Trabalho Programação Orientada a Objetos

- 2017089014 - Alex Souza
- 2017001320 - Rafael 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;

- Resolução (a)

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

- Resolução (b)

> 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.

- Resolução (c)

> 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.

- Resolução (d)

> 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.

> 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.

##Código Fonte

###main.cpp

```
{{embed 'src/main.cpp' 'cpp')}}
```

```
{{embed}}:..
#ifndef MAIN
#define MAIN

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

#define NUMERO_DE_VERTICES 5

using namespace std;

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

/**
 * Função que efetua a inclusão de algumas arestas 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);

/**
 * 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);
```

```
void exibirDijkstra(Graph &graph);

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);
    graph.mst();
    char a = 'A';
    graph.travellingSalesman(a);


    cout << endl << endl << "Total de conexões: " <<
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);
    graphCompleto.mst();
    cout << endl << endl << "Total de conexões: " <<
graphCompleto.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++){
        graph.dijkstra(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 << "Aresta adicionada com sucesso (" << v1 << ", " << v2 <<
")." << endl;
    }else{
        cout << "Não foi possível adicionar a aresta (" << v1 << ", " << v2
<< ")." << endl;
    }

    v1=3, v2=1;
    if(graph.insert(Edge(v1,v2))){
        cout << "Aresta adicionada com sucesso (" << v1 << ", " << v2 <<
")." << endl;
    }else{
        cout << "Não foi possível adicionar a aresta (" << v1 << ", " << v2
<< ")." << endl;
    }

    v1=3, v2=4;
    if(graph.insert(Edge(v1,v2))){
        cout << "Aresta adicionada com sucesso (" << v1 << ", " << v2 <<
")." << endl;
    }else{
        cout << "Não foi possível adicionar a aresta (" << v1 << ", " << v2
<< ")." << endl;
    }

    v1=2, v2=2;
    if(graph.insert(Edge(v1,v2))){
        cout << "Aresta adicionada com sucesso (" << v1 << ", " << v2 <<
")." << endl;
    }else{
        cout << "Não foi possível adicionar a aresta (" << v1 << ", " << v2
<< ")." << endl;
    }

    v1=1, v2=4, weight=2;
    if(graph.insert(Edge(v1,v2, weight))){
        cout << "Aresta adicionada com sucesso (" << v1 << ", " << v2 <<
")." << endl;
    }else{
        cout << "Não foi possível adicionar a aresta (" << v1 << ", " << v2
<< ")." << endl;
    }
```

```
    v1=4, v2=3;
    if(graph.insert(Edge(v1,v2))){
        cout << "Aresta adicionada com sucesso (" << v1 << ", " << v2 <<
").." << endl;
    }else{
        cout << "Não foi possível adicionar a aresta (" << v1 << ", " << v2
<< ")." << endl;
    }

    v1=4, v2=3;
    if(graph.insert(Edge(v1,v2))){
        cout << "Aresta adicionada com sucesso (" << v1 << ", " << v2 <<
").." << endl;
    }else{
        cout << "Não foi possível adicionar a aresta (" << v1 << ", " << v2
<< ")." << endl;
    }

}

void removerEdges(Graph &graph){

    int v1=3, v2=4;
    if(graph.remove(Edge(v1,v2))){
        cout << "Aresta removida com sucesso (" << v1 << ", " << v2 << ")."
<< endl;
    }else{
        cout << "Não foi possível remover a aresta (" << v1 << ", " << v2
<< ")." << endl;
    }

    v1=4, v2=2;
    if(graph.remove(Edge(v1,v2))){
        cout << "Aresta removida com sucesso (" << v1 << ", " << v2 << ")."
<< endl;
    }else{
        cout << "Não foi possível remover a aresta (" << v1 << ", " << v2
<< ")." << endl;
    }

}

#endif
```

Plugin	README
Dropbox	[plugins/dropbox/README.md][PlDb]
Github	[plugins/github/README.md][PlGh]
Google Drive	[plugins/googledrive/README.md][PlGd]

Plugin	README
OneDrive	[plugins/onedrive/README.md][PlOd]
Medium	[plugins/medium/README.md][PlMe]
Google Analytics	[plugins/googleanalytics/README.md][PlGa]