Universidade do Porto

FEUP Faculdade de Engenharia

Determinação de percursos turísticos

Mestrado Integrado em Engenharia Informática e Computação

(MIEIC)

Autores:

Ivo Lima da Silva – 120509192 – ei12192@fe.up.pt

João Manuel Ferreira Trindade – 110509118 – ei11118@fe.up.pt

Índice

Descrição

O objectivo do trabalho é, dado um mapa de ruas contendo monumentos calcular um percurso passando por todos as ruas que contêm monumentos acabando no ponto inicial.

Para tal usamos um algoritmo *greedy* que calculava qual o monumento mais perto do ponto em que estamos (usando o algoritmo *dijkstra*) e "viajava" para o ponto em questão. A partir daí repetia o processo até os monumentos estarem todos visitados e aí calculava o caminho mais curto para a origem.

Manual do utilizador

Ao iniciar o programa, o utilizador é apresentado um menu principal com acesso a todas as opções principais. Além disso, tem também o *Graph-Viewer* onde pode ver, em tempo real, todas as operações que efectua.

```
Bem vindo a aplicacao
1 - Adicionar Rua
2 - Remover Rua
3 - Listar Rua
4 - Adicionar Monumento
5 - Remover Monumento
6 - Listar Monumento

7 - Gerar Caminho

8 - Guardar em ficheiro
9 - Load de ficheiro

0 - Encerrar

Escolha opcao
```

Este é o menu do programa e ao qual o programa retorna sempre que executa uma das suas funções. As suas funções são:

1. Adicionar Rua: Esta função adiciona uma rua liga a dois vértices, existentes ou não. Quando se pretende criar uma rua nova em que um ou os dois vértices não existam é necessário dar o valor -1 aos vértices. Isto criará vértices novos.
2. Remover Rua: Remove a Rua cujo nome tenha sido indicada. Esta função não faz qualquer verificação se a rua contêm monumentos ou não, pelo qual é preciso cuidado ao usá-la.
3. Listar Rua: Lista todas as ruas inseridas tal a distância da rua.
4. Adicionar Monumento: A função adiciona monumentos na rua dado o seu nome. Caso a rua não exista o monumento não é inserido.
5. Remover Monumento: Remove o monumento dado o seu nome.
6. Listar Monumento: Lista as ruas e os seus monumentos.
7. Gerar caminho: indica o caminho desde a origem passando por todos os monumentos e voltando à origem. Mostra os IDs dos vértices por onde passou.
8. Guardar em ficheiro: Guarda em ficheiro *txt* todas as ruas e monumentos inseridos até à data.

9. *Load* de ficheiro: Carrega a informação guardada no ficheiro “mapa.txt” localizado na pasta do programa.

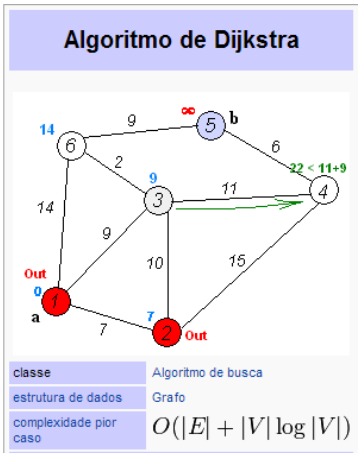
É ainda possível proceder a alteração directa do ficheiro de texto, onde se encontra armazenada a informação sobre cada rua. A estruturação desse ficheiro é a seguinte:

$$xx\ \ yy\ \ NOME\ DA\ RUA@MONUMENTO1\#MONUMENTO2\#$$

Uma rua por cada linha. Caracteres até ao primeiro espaço representam o ID do primeiro nó, ou cruzamento de ruas. Os caracteres até ao 2º espaço representam o ID do segundo nó. Os caracteres até ao @ representam o nome da rua, e todos os caracteres, daí a frente, representam os monumentos presentes nessa rua. O caracter ‘#’ é usado como separador de monumentos.

Algoritmos implementados

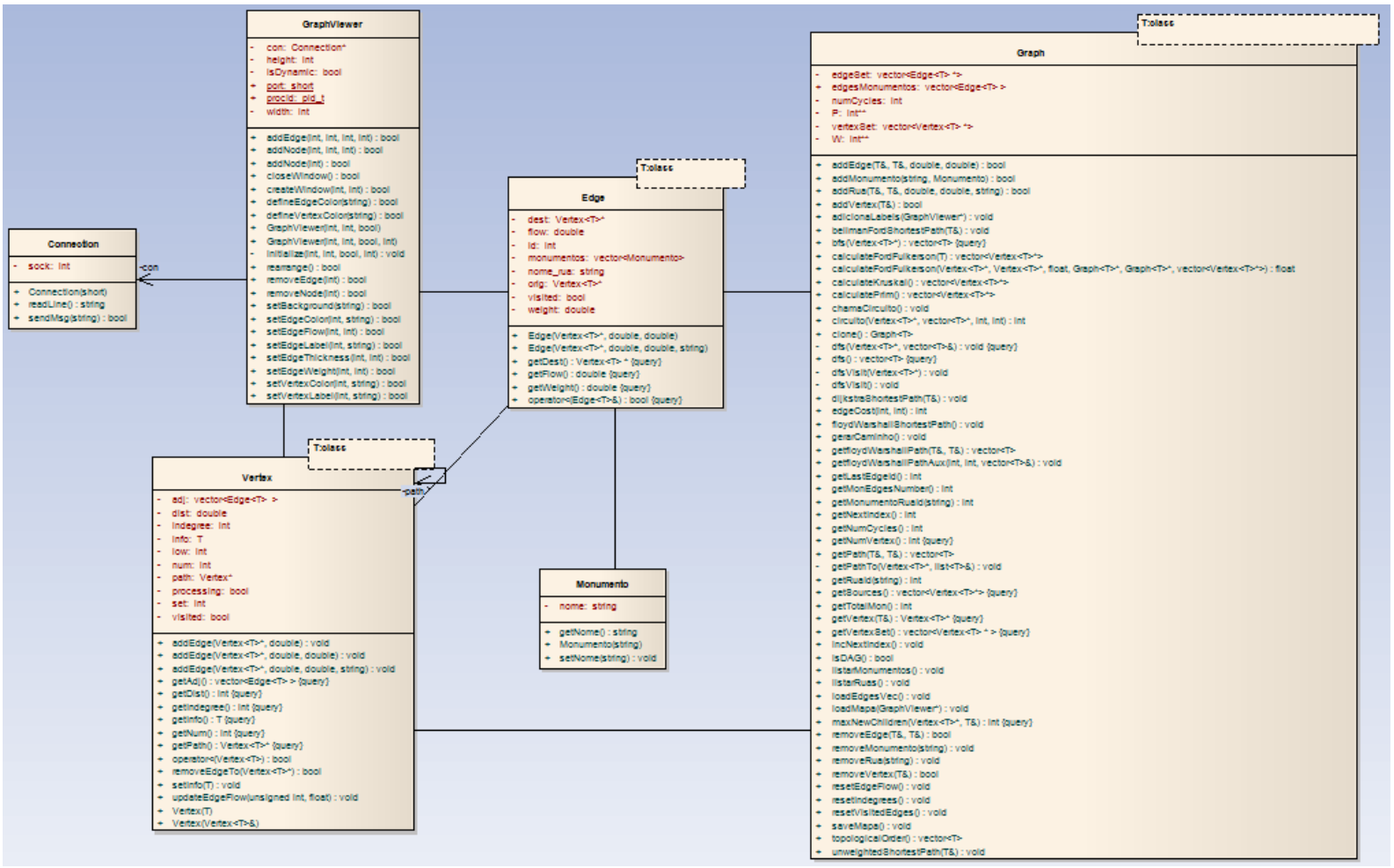
Para solucionar o problema do caminho mais curto entre cada dois vértices, optamos pelo algoritmo de Dijkstra, uma vez que as arestas do nosso grafo, ruas da cidade, não podem ter distâncias negativas, o que é um dos impedimentos ao uso deste algoritmo.



Este algoritmo permite-nos uma complexidade temporal correspondente ao numero de arestas + num_vertices* log(num_vertices).

Foi implementado também o algoritmo *greedy* para decidir qual aresta seguir em caso de empate. É um algoritmo bastante simples pois opta sempre pela aresta de menor comprimento, o que é um caso de desempate bastante seguro pois o objectivo é devolver o caminho mais perto do ideal (com o menor percurso percorrido).

UML


GraphViewer
- con: Connection*
- height: int
- isDynamic: bool
+ port: short
+ procId: pid_t
- width: int
+ addEdge(int, int, int, int) : bool
+ addNode(int, int, int) : bool
+ addNode(int) : bool
+ closeWindow() : bool
+ createWindow(int, int) : bool
+ defineEdgeColor(string) : bool
+ defineVertexColor(string) : bool
+ GraphViewer(int, int, bool)
+ GraphViewer(int, int, bool, int)
- initialize(int, int, bool, int) : void
+ rearrange() : bool
+ removeEdge(int) : bool
+ removeNode(int) : bool
+ setBackground(string) : bool
+ setEdgeColor(int, string) : bool
+ setEdgeFlow(int, int) : bool
+ setEdgeLabel(int, string) : bool
+ setEdgeThickness(int, int) : bool
+ setEdgeWeight(int, int) : bool
+ setVertexColor(int, string) : bool
+ setVertexLabel(int, string) : bool
Connection
- sock: int
+ Connection(short)
+ readLine() : string
+ sendMsg(string) : bool
con
T:class
Edge
- dest: Vertex<T>*
- flow: double
- id: int
- monumentos: vector<Monumento>
- nome_rua: string
- orig: Vertex<T>*
- visited: bool
- weight: double
+ Edge(Vertex<T>*, double, double)
+ Edge(Vertex<T>*, double, double, string)
+ getDest() : Vertex<T> * {query}
+ getFlow() : double {query}
+ getWeight() : double {query}
+ operator<(Edge<T>&) : bool {query}
T:class
Graph
- edgeSet: vector<Edge<T> *>
+ edgesMonumentos: vector<Edge<T> >
- numCycles: int
- P: int**
- vertexSet: vector<Vertex<T> *>
- W: int**
+ addEdge(T&, T&, double, double) : bool
+ addMonumento(string, Monumento) : bool
+ addRua(T&, T&, double, double, string) : bool
+ addVertex(T&) : bool
+ adicionaLabels(GraphViewer*) : void
+ bellmanFordShortestPath(T&) : void
+ bfs(Vertex<T>*) : vector<T> {query}
+ calculateFordFulkerson(T) : vector<Vertex<T>*>
+ calculateFordFulkerson(Vertex<T>*, Vertex<T>*, float, Graph<T>*, Graph<T>*, vector<Vertex<T>*>) : float
+ calculateKruskal() : vector<Vertex<T>*>
+ calculatePrim() : vector<Vertex<T>*>
+ chamaCircuito() : void
+ circuito(Vertex<T>*, vector<T>*, int, int) : int
+ clone() : Graph<T>
- dfs(Vertex<T>*, vector<T>&) : void {query}
+ dfs() : vector<T> {query}
- dfsVisit(Vertex<T>*) : void
- dfsVisit() : void
+ dijkstraShortestPath(T&) : void
+ edgeCost(int, int) : int
+ floydWarshallShortestPath() : void
+ gerarCaminho() : void
+ getfloydWarshallPath(T&, T&) : vector<T>
+ getfloydWarshallPathAux(int, int, vector<T>&) : void
+ getLastEdgeId() : int
+ getMonEdgesNumber() : int
+ getMonumentoRua(string) : int
+ getNextIndex() : int
+ getNumCycles() : int
+ getNumVertex() : int {query}
+ getPath(T&, T&) : vector<T>
- getPathTo(Vertex<T>*, list<T>&) : void
+ getRua(string) : int
+ getSources() : vector<Vertex<T>*> {query}
+ getTotalMon() : int
+ getVertex(T&) : Vertex<T>* {query}
+ getVertexSet() : vector<Vertex<T> * > {query}
+ incNextIndex() : void
+ isDAG() : bool
+ listarMonumentos() : void
+ listarRuas() : void
+ loadEdgesVec() : void
+ loadMaps(GraphViewer*) : void
+ maxNewChildren(Vertex<T>*, T&) : int {query}
+ removeEdge(T&, T&) : bool
+ removeMonumento(string) : void
+ removeRua(string) : void
+ removeVertex(T&) : bool
+ resetEdgeFlow() : void
+ resetIndegrees() : void
+ resetVisitedEdges() : void
+ saveMaps() : void
+ topologicalOrder() : vector<T>
+ unweightedShortestPath(T&) : void
T:class
Vertex
- adj: vector<Edge<T> >
- dist: double
- indegree: int
- info: T
- low: int
- num: int
- path: Vertex*
- processing: bool
- set: int
- visited: bool
+ addEdge(Vertex<T>*, double) : void
+ addEdge(Vertex<T>*, double, double) : void
+ addEdge(Vertex<T>*, double, double, string) : void
+ getAdj() : vector<Edge<T> > {query}
+ getDist() : int {query}
+ getIndegree() : int {query}
+ getInfo() : T {query}
+ getNum() : int {query}
+ getPath() : Vertex<T>* {query}
+ operator<(Vertex<T>) : bool
+ removeEdgeTo(Vertex<T>*) : bool
+ setInfo(T) : void
+ updateEdgeFlow(unsigned int, float) : void
+ Vertex(T)
+ Vertex(Vertex<T>&)
path*
Monumento
- nome: string
+ getNome() : string
+ Monumento(string)
+ setNome(string) : void