Instituto Superior de Engenharia do Porto

1DCDD – GRUPO 034 - CODEFLOW

SPRINT 3 – US17 e US18

MATEMÁTICA DISCRETA



PEDRO COSTA (1221790)

RUI SANTIAGO (1221402)

FRANCISCO TROCADO (1230608)

# US17 e US18 – Procedimento implementado

No âmbito da disciplina de matemática discreta, nas User Stories 17 e 18 pretende-se colocar sinais para evacuar (em caso de emergência) as pessoas que estão no parque para um Assembly Point (us17) ou vários Assembly Points (us18).

Com o presente documento vamos mostrar o algoritmo em código que desenvolvemos para solucionar este problema.

Para fazer este algoritmo, a equipa optou por utilizar HashMaps em vez de vetores ou matrizes, um HashMap é uma estrutura de dados em Java que armazena pares de “chave-valor”, permitindo uma busca rápida baseada na chave. Cada chave única está associada a um valor, e a busca, inserção e remoção de elementos são muito eficientes, geralmente O(1), devido ao uso de uma tabela de dispersão (hash table).

Para resolver o problema da US18, o algoritmo de Dijkstra é repetido para a quantidade de Assembly Points encontrados, comparando os valores das distancias obtidos a cada iteração e escolhendo o caminho mais curto para o Assembly Point mais próximo.

Tendo em conta o contexto do semestre e a implementação da programação orientada ao objeto (POO), foram desenvolvidas sub-classes que representam diferentes objetos:

* **Class Vertex**– cria objetos representantes de um vértice do grafo, apenas tendo o nome e métodos de comparação.
* **Class Edge** – cria objetos que representam arestas do grafo, com uma referencia ao vértice de origem e ao vértice de chegada, assim como o respetivo custo desta ligação.
* **Class Graph** – cria um objeto representante de um grafo, tendo uma lista de objetos de vértices e arestas e que implementa os métodos necessários na execução do algoritmo.

# Class Vertex:

//Classe que representa um vertice do grafo  
public class Vertex {  
  
 private String name; //nome do vertice  
  
 // Construtor de um novo vertice  
 public Vertex(String name) {  
 this.name = name;  
 }  
  
 // metodo que retorna o nome do vertice  
 public String getName() {  
 return name;  
 }  
  
 // metodo que verifica se dois vertices sao iguais  
 @Override  
 public boolean equals(Object possibleVertex) {  
 if (this == possibleVertex)  
 return true;  
 if (possibleVertex == null)  
 return false;  
 if (getClass() != possibleVertex.getClass())  
 return false;  
 Vertex otherVertex = (Vertex) possibleVertex;  
 if (name == null) {  
 if (otherVertex.name != null)  
 return false;  
 } else if (!name.equals(otherVertex.name))  
 return false;  
 return true;  
 }  
}

# Class Edge:

// Classe que representa uma aresta de um grafo  
public class Edge {  
  
 private Vertex startVertex; // vértice de origem  
 private Vertex endVertex; // vértice de destino  
 private int weight; // peso da aresta  
  
 // Construtor de uma nova aresta  
 public Edge(Vertex startVertex, Vertex endVertex, int weight) {  
 this.startVertex = startVertex;  
 this.endVertex = endVertex;  
 this.weight = weight;  
 }  
  
 // metodo que retorna o vértice de origem  
 public Vertex getVertexFrom() {  
 return startVertex;  
 }  
  
 // metodo que retorna o vértice de destino  
 public Vertex getVertexTo() {  
 return endVertex;  
 }  
  
 // metodo que retorna o peso da aresta  
 public int getWeight() {  
 return weight;  
 }  
  
 //metodo que retorna o vertice na forma "Origem -Peso- Destino"  
 @Override  
 public String toString(){  
 return startVertex + " -" + weight + "- " + endVertex;  
 }  
  
 //metodo que verifica se duas arestas são iguais  
 @Override  
 public boolean equals(Object obj) {  
 if (obj instanceof Edge) {  
 Edge edge = (Edge) obj;  
 boolean equal = false;  
 if(startVertex.equals(edge.getVertexFrom()) && endVertex.equals(edge.getVertexTo()))  
 equal = true;  
 if(startVertex.equals(edge.getVertexTo()) && endVertex.equals(edge.getVertexFrom()))  
 equal = true;  
 return equal;  
 }  
 return false;  
 }  
}

# Class Graph:

// Classe que representa um grafo  
public class Graph {  
  
 private final List<Vertex> vertices; // Lista de vértices do grafo  
 private final List<Edge> edges; // Lista de arestas do grafo  
  
 // Construtor de um novo grafo  
 public Graph(List<Vertex> vertices, List<Edge> edges) {  
 this.vertices = vertices;  
 this.edges = edges;  
 }  
 // Métodos que retorna a lista dos vértices do grafo  
 public List<Vertex> getVertexes() {  
 return vertices;  
 }  
 // Método que retorna a lista das arestas do grafo  
 public List<Edge> getEdges() {  
 return edges;  
 }  
 // Método que retorna a lista das arestas que saem de um outro vértice  
 public List<Edge> getEdgesFromVertex(Vertex vertex) {  
 return edges.stream()  
 .filter(edge -> edge.getVertexFrom().equals(vertex) || edge.getVertexTo().equals(vertex))  
 .collect(Collectors.*toList*());  
 }  
   
 // Método que mostra o grafo na consola  
 public static void display(Graph graph){  
 for(Edge edge : graph.getEdges()){  
 System.*out*.println(edge);  
 }  
 }  
}

# Algortimo de Dijkstra – US17

Este algoritmo determina o caminho de custo mínimo de um vértice inicial para todos os outros vértices de um grafo. Começamos por inicializar uma lista de distâncias, onde todas as distâncias são definidas como infinitas, exceto a distância do vértice inicial para ele mesmo, que é zero. Também inicializamos um conjunto de vértices visitados, que começa vazio por ainda não termos visitado nenhum vértice.

O próximo passo é iterar os vértices do grafo enquanto o número de vertices na lista de vértices visitados não for igual ao número de vertices total do grafo.

A cada iteração, escolhemos o vértice com a menor distância atualmente conhecida, removemos esse vértice do conjunto de vértices não visitados e analisamos todas as suas arestas adjacentes. Para cada aresta adjacente em que o destino ainda não esteja na lista de visitados, calculamos a distância potencial até o vértice de destino através do vértice atualmente a visitar. Se essa distância calculada for menor do que a distância atualmente registrada para o vértice de destino, atualizamos a distância e registramos o vértice atual como predecessor.

Esse processo é repetido até que todos os vértices tenham sido visitados ou até que a menor distância conhecida seja infinita (indicando que os vértices restantes são inacessíveis a partir do vértice inicial). O resultado final é uma lista de distâncias mínimas do vértice inicial para todos os outros vértices e, opcionalmente, uma árvo de caminhos de custo mínimo, que pode ser reconstruída a partir dos predecessores de cada vértice.

public static void DijkstraAlgorithm(Graph graph,Map<Vertex, Integer> distance, Map<Vertex, Vertex> previous, Vertex source) {  
 // Lista de vértices visitados  
 List<Vertex> visited = new ArrayList<>();  
  
 // Inicializa as distâncias e os vértices anteriores  
 for (Vertex vertex : graph.getVertexes()) {  
 distance.put(vertex, Integer.*MAX\_VALUE*); // todas as distâncias são inicializadas com infinito  
 previous.put(vertex, null); // todos os vértices anteriores são inicializados com null  
 }  
 distance.put(source, 0); // a distância do vértice AP começará em 0  
 previous.put(source, source); // o vértice anterior do vértice AP será ele mesmo  
  
 Vertex visiting = source; // colocar o vértice AP como o vertice que estamos a visitar  
  
 while(visited.size() != graph.getVertexes().size()){  
 // Obter as arestas adjacentes ao vértice que estamos a visitar  
 List<Edge> adjacentEdges = graph.getEdgesFromVertex(visiting);  
 for(Edge adjacentEdge : adjacentEdges){ // para cada aresta  
 Vertex neighbor; // obter o vértice vizinho ao vértice que estamos a visitar  
 if(adjacentEdge.getVertexFrom().getName().equals(visiting.getName())){  
 neighbor = adjacentEdge.getVertexTo();  
 } else {  
 neighbor = adjacentEdge.getVertexFrom();  
 }  
 // se a distancia alocada ao vertice vizinho for maior que a distancia da aresta + a distancia do vertice que estamos a visitar  
 // alocar a distancia do vertice vizinho como a distancia da aresta + a distancia do vertice que estamos a visitar  
 // alocar o vertice anterior do vertice vizinho como o vertice que estamos a visitar  
 if(distance.get(neighbor) > distance.get(visiting) + adjacentEdge.getWeight() && !visited.contains(neighbor) ){  
 distance.put(neighbor, distance.get(visiting) + adjacentEdge.getWeight());  
 previous.put(neighbor, visiting);  
 }  
 }  
 // adicionar o vertice que estamos a visitar à lista de visitados  
 visited.add(visiting);  
 // obter o vertice com a menor distancia que ainda não foi visitado  
 visiting = *getMinDistanceVertex*(graph, distance, visited);  
 }  
}  
  
// Metodo para obter o vertice com a menor distancia que ainda não foi visitado  
public static Vertex getMinDistanceVertex(Graph graph, Map<Vertex, Integer> distance, List<Vertex> visited) {  
 Vertex nextVertex = null;  
 for(Vertex vertex : graph.getVertexes()){  
 if(!visited.contains(vertex)){  
 if(nextVertex == null){  
 nextVertex = vertex;  
 }  
 if(distance.get(vertex) < distance.get(nextVertex)){  
 nextVertex = vertex;  
 }  
 }  
 }  
 return nextVertex;  
}

# Obter o caminho mais curto do vértice X ao vertice Y

Para reconstruir o caminho mais curto entre um vértice X ao vertice inicial Y, implementamos um método que percorre a lista de distancias e de predecessores para obter uma representação em texto, por exemplo: (X; ... ; Y); 10 path cost

public static String shortestPathToString(Vertex vertex, List<Vertex> sources, Map<Vertex, Integer> distance , Map<Vertex, Vertex> previous){  
 String shortestPath = "";  
 Vertex currentVertex = vertex;  
 Vertex sourceOfVertex = null;  
 while(!sources.contains(currentVertex)){  
 shortestPath += currentVertex.getName() + ";";  
 currentVertex = previous.get(currentVertex);  
 sourceOfVertex = currentVertex;  
 }  
 shortestPath += sourceOfVertex + "); ";  
 shortestPath += distance.get(vertex);  
 return shortestPath;  
}

# Execução da US18

A US18 executa o algoritmo de Dijkstra para cada AP encontrado no grafo do ficheiro da US18, salvando as distancias e precedentes definitivos em HashMaps diferentes até ter-mos os caminhos minimos de todos os vértices para o AP “mais perto”

for(Vertex vertexI : us18\_APs){  
 Main.*DijkstraAlgorithm*(us18Graph, *distance*, *previous*, vertexI);  
 for(Vertex vertexJ : us18Graph.getVertexes()){  
 if(*distanceDefinitive*.get(vertexJ) == null || *distanceDefinitive*.get(vertexJ) > *distance*.get(vertexJ)){  
 *distanceDefinitive*.put(vertexJ, *distance*.get(vertexJ));  
 *previousDefinitive*.put(vertexJ, *previous*.get(vertexJ));  
 }  
 }  
}