



Instituto Superior de  
**Engenharia** do Porto

## **Relatório - Projeto Integrador Sprint II**

### **Estruturas de Informação**

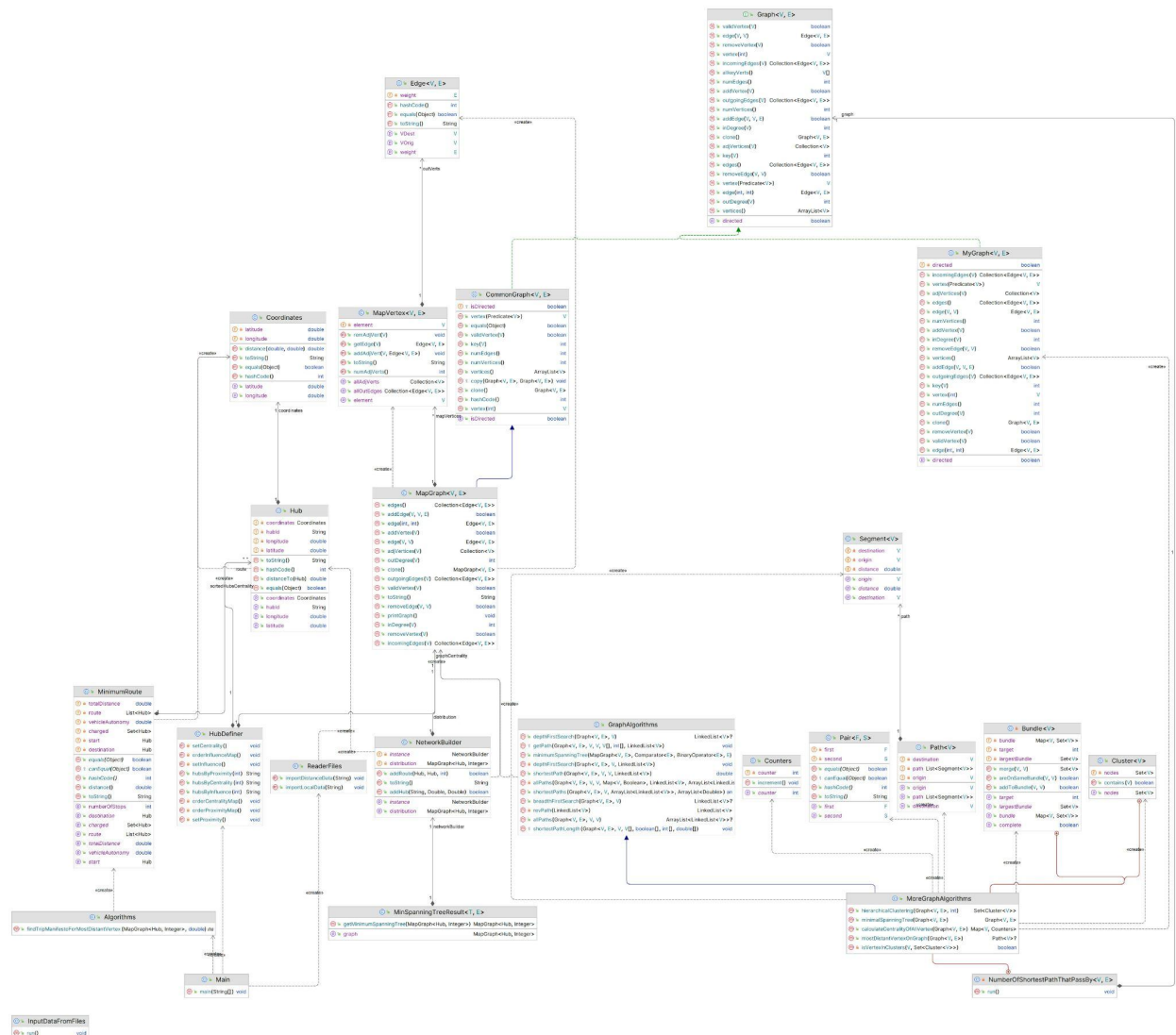
**Realizado por :**

**Lourenço Guimarães, 1220766**

**Diogo Ribeiro, 1220812**

**Francisco Silveira, 1220813**

**Rodrigo Brito, 1220842**



## Análise dos métodos implementados

Para a realização do projeto foram utilizadas diversas classes de implementação e manipulação de grafos, sendo que daremos maior destaque às classes 'GraphAlgorithms', 'MapGraph' e 'MapVertex'.

- **GraphAlgorithms:** Nesta classe encontram-se todos os métodos que permitem a pesquisa no grafo.

Começamos pela análise do algoritmo breadthFirstSearch, que realiza uma pesquisa em largura ao grafo, visitando todos os vértices e arestas uma vez, o que se traduz numa complexidade  $O(|V| + |E|)$ , sendo  $|V|$  o número de vértices do grafo e  $|E|$  o número de arestas do mesmo.

```

/**
 * Performs breadth-first search of a Graph starting in a Vertex
 *
 * @param g Graph instance
 * @param vert information of the Vertex that will be the source of the search
 * @return qbfs a queue with the vertices of breadth-first search
 */
no usages Francisco Silveira
public static <V, E> LinkedList<V> breadthFirstSearch(Graph<V, E> g, V vert) {
    if (!g.validVertex(vert)) {
        return null;
    }
    LinkedList<V> neighbors = new LinkedList();
    LinkedList<V> auxiliar = new LinkedList<>();
    auxiliar.add(vert);
    neighbors.add(vert);
    while (!auxiliar.isEmpty()) {
        vert = auxiliar.remove();
        for (V adj : g.adjVertices(vert)) {
            if (!neighbors.contains(adj)) {
                neighbors.add(adj);
                auxiliar.add(adj);
            }
        }
    }
    return neighbors;
}

```

Já o algoritmo depthFirstSearch, realiza uma pesquisa em profundidade ao grafo, começando num vértice especificado. Apresenta por isso também uma complexidade do tipo  $O(|V| + |E|)$ , sendo  $|V|$  o número de vértices do grafo e  $|E|$  o número de arestas do mesmo.

```

private static <V, E> void depthFirstSearch(Graph<V, E> g, V vOrig, LinkedList<V> qdfs) {
    qdfs.add(vOrig);
    for (V adj : g.adjVertices(vOrig)) {
        if (!qdfs.contains(adj)) {
            depthFirstSearch(g, adj, qdfs);
        }
    }
}

/**
 * @param g    Graph instance
 * @param vert information of the Vertex that will be the source of the search
 * @return qdfs a queue with the vertices of depth-first search
 */
no usages  Francisco Silveira
public static <V, E> LinkedList<V> depthFirstSearch(Graph<V, E> g, V vert) {
    if (!g.validVertex(vert)) {
        return null;
    }
    LinkedList<V> path = new LinkedList<>();
    depthFirstSearch(g, vert, path);
    return path;
}

```

O algoritmo 'allPaths' retorna todos os caminhos entre um vértice fonte e um vértice destino passados por parâmetro, a complexidade deste método é exponencial, visto que no pior dos casos são gerados todos os caminhos possíveis do grafo.

```

private static <V, E> boolean allPaths(Graph<V, E> g, V vOrig, V vDest, Map<V, Boolean> visited, LinkedList<V> path, ArrayList<LinkedList<V>> paths) {
    if (vDest.equals(vOrig)) {
        path.add(vDest);
        paths.add(path);
        return true;
    }
    visited.put(vDest, true);
    for (V adj : g.adjVertices(vDest)) {
        if (visited.get(adj) != null) {
            continue;
        }
        if (allPaths(g, vOrig, adj, new HashMap<>(visited), new LinkedList<>(path), paths)) {
            path.add(adj);
        }
    }
    return false;
}

/**
 * @param g      Graph instance
 * @param vOrig   information of the Vertex origin
 * @param vDest   information of the Vertex destination
 * @return paths ArrayList with all paths from vOrig to vDest
 */
1 usage  @ Francisco Silveira
public static <V, E> ArrayList<LinkedList<V>> allPaths(Graph<V, E> g, V vOrig, V vDest) {
    if (!g.validVertex(vOrig) || !g.validVertex(vDest)) {
        return null;
    }
    ArrayList<LinkedList<V>> paths = new ArrayList<>();
}

```

O método ‘minimumSpanningTree’ computa a minimum spanning tree de um grafo recorrendo ao algoritmo ‘Prim’. Apresenta também complexidade do tipo  $O(|V| + |E|)$ , sendo  $|V|$  o número de vértices do grafo e  $|E|$  o número de arestas do mesmo, devido a usar uma queue prioritária.

```

public static <V, E> MapGraph<V, E> minimumSpanningTree(MapGraph<V, E> g, Comparator<E> ce, BinaryOperator<E> sum, E zero) {
    // Create a set to keep track of visited vertices
    Set<V> visitedVertices = new HashSet<>();

    // Create a priority queue to store edges based on their weights
    PriorityQueue<Edge<V, E>> edgeQueue = new PriorityQueue<>(Comparator.comparing(e -> e.getWeight(), ce));

    // Create a graph to represent the minimum spanning tree
    MapGraph<V, E> minimumSpanningTree = new MapGraph<>(directed: true);

    // Add an arbitrary vertex to start the process
    V startVertex = g.vertices().iterator().next();
    visitedVertices.add(startVertex);

    // Add all edges connected to the start vertex to the priority queue
    for (Edge<V, E> edge : g.outgoingEdges(startVertex)) {
        edgeQueue.add(edge);
    }

    // Continue adding edges until all vertices are visited
    while (visitedVertices.size() < g.numVertices()) {
        // Obtenha a aresta de peso mínimo da fila de prioridade
        Edge<V, E> minEdge = edgeQueue.poll();
        // Check if the priority queue is empty
        if (minEdge == null) {
            // The graph is not connected
            break;
        }
    }
}

```

- **MapVertex:** Esta classe tem como objetivo representar um vértice no grafo. Cada instância desta classe armazena dados dos vértices, incluindo o elemento associado e as arestas que o conectam aos vértices adjacentes. A classe mantém uma coleção de vértices adjacentes, onde cada vértice adjacente é associado a uma aresta. Fornece métodos para adicionar e remover vértices adjacentes, bem como para recuperar informações sobre as arestas que conectam o vértice aos vértices adjacentes, contém métodos para obter o número total de vértices adjacentes, etc.

```

public class MapVertex<V, E> {

    3 usages
    final private V element; // Vertex information
    10 usages
    final private Map<V, Edge<V, E>> outVerts; // Adjacent vertices

    Constructs a vertex with the specified element.
    Params: vert – The information associated with the vertex.
    Throws: RuntimeException – if the vertex information is null.

    1 usage  ⚡ Diogo_1220812
    public MapVertex(V vert) {
        if (vert == null) throw new RuntimeException("Vertice information cannot be null!");
        element = vert;
        outVerts = new LinkedHashMap<>();
    }

    Gets the element associated with the vertex.
    Returns: The element associated with the vertex.

    6 usages  ⚡ Diogo_1220812
    public V getElement() { return element; }

    Adds an adjacent vertex along with the connecting edge.
    Params: vAdj – The adjacent vertex to be added.
           edge – The connecting edge to the adjacent vertex.

    2 usages  ⚡ Diogo_1220812
    public void addAdjVert(V vAdj, Edge<V, E> edge) { outVerts.put(vAdj, edge); }

    Removes an adjacent vertex.
    Params: vAdj – The adjacent vertex to be removed.
}

```

- **MapGraph:** A classe MapGraph representa um grafo usando uma lista de adjacências baseada em mapas, incluindo métodos específicos para manipular um grafo representado por vértices e arestas armazenados num LinkedHashMap. Abaixo estão os principais métodos da classe MapGraph.



```

public static <V, E> MapGraph<V, E> minimumSpanningTree(MapGraph<V, E> g, Comparator<E> ce, BinaryOperator<E> sum, E zero) {
    // Create a set to keep track of visited vertices
    Set<V> visitedVertices = new HashSet<>();

    // Create a priority queue to store edges based on their weights
    PriorityQueue<Edge<V, E>> edgeQueue = new PriorityQueue<>(Comparator.comparing(e -> e.getWeight(), ce));

    // Create a graph to represent the minimum spanning tree
    MapGraph<V, E> minimumSpanningTree = new MapGraph<>(directed: true);

    // Add an arbitrary vertex to start the process
    V startVertex = g.vertices().iterator().next();
    visitedVertices.add(startVertex);

    // Add all edges connected to the start vertex to the priority queue
    for (Edge<V, E> edge : g.outgoingEdges(startVertex)) {
        edgeQueue.add(edge);
    }

    // Continue adding edges until all vertices are visited
    while (visitedVertices.size() < g.numVertices()) {
        // Obtenha a aresta de peso mínimo da fila de prioridade
        Edge<V, E> minEdge = edgeQueue.poll();
        // Check if the priority queue is empty
        if (minEdge == null) {
            // The graph is not connected
            break;
        }
    }
}

```

**Exercício 1:** No exercício 1 pretende-se a construção da rede de distribuição a partir dos dados fornecidos nos ficheiros csv disponibilizados, o que se garante com a construção do grafo.

Para a construção deste, foram necessários, principalmente, os métodos addHub e addRoute.

- **addHub** : este método recebe por parâmetros um hubId, uma latitude e uma longitude e tem como principal objetivo a adição de um hub (vértice do grafo) à rede de distribuição, apresentando um ID, latitude e longitude. Apresenta uma **complexidade O(1)** por ser um método de inserção.

```

/**
 * Adds a route to the distribution network between the specified origin and destination hubs
 * with the given distance.
 *
 * @param orig The origin hub.
 * @param dest The destination hub.
 * @param distance The distance between the origin and destination hubs.
 * @return `true` if the route is successfully added; `false` otherwise.
 */
4 usages  ± Diogo_1220812
public boolean addRoute(Hub orig, Hub dest, int distance) { return distribution.addEdge(orig, dest, distance); }

```

- **addRoute** : Neste método são passados por parâmetro dois objetos Hub e uma distância, sendo este responsável pela inserção da route(edge) entre dois hubs, sendo que a distância fornecida é o comprimento da route criada. A **complexidade é também  $O(1)$** , por se tratar de um método de inserção.

```
/**
 * Adds a hub to the distribution network with the specified hubId, latitude, and longitude.
 *
 * @param hubId The unique identifier of the hub.
 * @param latitude The latitude of the hub's geographical coordinates.
 * @param longitude The longitude of the hub's geographical coordinates.
 * @return `true` if the hub is successfully added; `false` otherwise.
 */
8 usages  Diogo_1220812
public boolean addHub(String hubId, Double latitude, Double longitude){
    Hub vert = new Hub(hubId,latitude, longitude);
    return distribution.addVertex(vert);
}
```

### Exercício 3:

```
public MinimumRoute findTripManifestoForMostDistantVertex(MapGraph<Hub, Integer> graph, double vehicleAutonomy) {
    Path<Hub> path = MoreGraphAlgorithms.mostDistantVertexOnGraph(graph);
    double autonomyLeft = vehicleAutonomy;
    double distance = 0;
    Hub lastHub = null;
    if (path != null && path.getPath() != null && !path.getPath().isEmpty()) {
        lastHub = path.getPath().get(0).getOrigin();
    }

    Set<Hub> chargingStops = new HashSet<>();
    List<Hub> route = new LinkedList<>();
    route.add(lastHub);

    ESINF.Structure.Auxiliary.Segment<Hub> v = null;
    if (path != null && path.getPath() != null && !path.getPath().isEmpty()) {
        v = path.getPath().remove(index: 0);
    }

    if (v != null) {
        autonomyLeft -= v.getDistance();
    }

    if (v != null) {
        distance += v.getDistance();
    }
}
```

```

if (path != null && path.getPath() != null) {
    for (ESINF.Structure.Auxiliary.Segment<Hub> hubSegment : path.getPath()) {

        if (hubSegment.getDistance() > autonomyLeft) {
            chargingStops.add(lastHub);
            autonomyLeft = vehicleAutonomy;
        }
        autonomyLeft -= hubSegment.getDistance();
        distance += hubSegment.getDistance();
        route.add(hubSegment.getOrigin());
        lastHub = hubSegment.getDestination();
    }
    ESINF.Structure.Auxiliary.Segment<Hub> last = path.getPath().get(path.getPath().size() - 1);
    route.add(last.getDestination());
}
if (path != null) {
    return new MinimumRoute(path.getOrigin(), path.getDestination(), vehicleAutonomy, route, chargingStops, distance);
}
return null;
}

```

## findTripManifestoForMostDistanteVertex:

Este método rastreia a autonomia restante, a distância total percorrida e os pontos de carregamento ao longo do caminho. Utiliza também a classe MoreGraphAlgoritms para calcular o caminho mais distante no grafo (graph)

O resultado é armazenado na variável path, que é do tipo Path<Hub> e representa o caminho do vértice mais distante encontrado. Itera também sobre os segmentos do caminho, calculando a distância percorrida e atualizando a autonomia restante.

Quando a autonomia restante não é suficiente para cobrir a distância até o próximo hub, adiciona o hub atual aos pontos de carregamento e reinicia a autonomia. Constrói um objeto **MinimumRoute** com base nas informações obtidas.

Este método apresenta uma **complexidade de  $O(V^2)$** , onde V é o número de vértices no grafo. Isto ocorre devido à chamada do método mostDistantVertexOnGraph dentro do loop que itera sobre todos os vértices no grafo. A complexidade é quadrática em relação ao número de vértices.

```

public static <V, E> Path<V> mostDistantVertexOnGraph(Graph<V, E> graph) {
    double distance = Double.MIN_VALUE;
    Pair<V, V> pair = null;
    LinkedList<V> shortestPath = null;

    Map<V, Set<V>> hitMap = new HashMap<>();

    for (V v1 : graph.vertices()) {
        if (!hitMap.containsKey(v1)) {
            hitMap.put(v1, new HashSet<>());
        }
        for (V v2 : graph.vertices()) {
            if (v1.equals(v2)) continue;
            if (hitMap.get(v1).contains(v2)) continue;
            LinkedList<V> path = new LinkedList<>();
            double delta = shortestPath(graph, v1, v2, path);
            if (delta > distance) {
                distance = delta;
                pair = new Pair<>(v1, v2);
                shortestPath = path;
            }
            if (!hitMap.containsKey(v2)) {
                hitMap.put(v2, new HashSet<>());
            }
            hitMap.get(v2).add(v1);
            hitMap.get(v1).add(v2);
        }
    }
}

```

```

V start = null;
List<ESINF.Structure.Auxiliary.Segment<V>> segments = new LinkedList<>();

if (shortestPath != null) {
    for (V v : shortestPath) {
        if (start == null) {
            start = v;
            continue;
        }
        segments.add(new ESINF.Structure.Auxiliary.Segment<>(start, v, (Integer) graph.edge(start, v).getWeight()));
        start = v;
    }
}

if (pair != null) {
    return new Path<>(pair.getFirst(), pair.getSecond(), segments);
} else {
    return null;
}

```

## mostDistanteVertexOnGraph:

Este método tem o objetivo de encontrar o par de vértices no grafo que maximiza a distância entre eles, utilizando o algoritmo de menor caminho para calcular as distâncias. O uso do hitMap evita cálculos redundantes para pares de vértices já encontrados. O resultado é retornado como uma instância de **Path**.

Este método tem uma **complexidade de  $O(V^3)$** , onde  $V$  é o número de vértices no grafo. Isso ocorre porque ele realiza um loop duplo sobre todos os vértices, calculando os caminhos mais curtos entre pares de vértices. O cálculo dos caminhos mais curtos, por sua vez, tem uma **complexidade de  $O(V)$**  no pior caso. Assim, a complexidade geral é cúbica em relação ao número de vértices.

## Exercício 4:

```
package ESINF.US04;

import ...

6 usages  ▸ SobreRodasPlays
public class MinSpanningTreeResult<T, E> {
    1 usage
    NetworkBuilder networkBuilder = NetworkBuilder.getInstance();

    ▸ SobreRodasPlays
    public MapGraph<Hub, Integer> getGraph() { return networkBuilder.getDistribution(); }

    1 usage  ▸ SobreRodasPlays
    public static MapGraph<Hub, Integer> getMinimumSpanningTree(MapGraph<Hub, Integer> graph) {
        if (graph == null) {
            return null;
        } else {
            return minimumSpanningTree(graph, Integer::compare, Integer::sum, zero: 0);
        }
    }
}
```

A classe `MinSpanningTreeResult` é destinada a trabalhar com árvores de spanning mínimas em um grafo. Vamos explicar cada método da classe, incluindo a explicação do método `minimumSpanningTree`.

## getGraph e getMinimumSpanningTree:

O método `getGraph()` na classe `MinSpanningTreeResult` tem a finalidade de retornar o grafo associado à instância dessa classe. O método `getMinimumSpanningTree` na mesma classe tem como objetivo calcular a árvore de abrangência mínima (MST) para um grafo dado.

```
public static void printMinimumSpanningTree(MapGraph<Hub, Integer> minimumSpanning) {
    if (minimumSpanning == null) {
        System.out.println("The minimum spanning tree is null. Make sure you initialize it correctly.");
    } else {
        System.out.println("Minimum Spanning Tree:");
        System.out.printf("%-10s%-20s%s\n", "Origin", "--(Distance(m))--", "Destination");

        for (Edge<Hub, Integer> edge : minimumSpanning.edges()) {
            Hub vOrig = edge.getVOrig();
            Hub vDest = edge.getVDest();
            String hubOrigin = vOrig.getHubId();
            String hubDestination = vDest.getHubId();
            Integer weight = edge.getWeight();

            System.out.printf("%-10s%-20s%s\n", hubOrigin, "--(" + weight + ")--", hubDestination);
        }
    }
}

2 usages  ▴ SobreRodasPlays
public static int calculateTotalDistance(MapGraph<Hub, Integer> minimumSpanning) {
    int totalDistance = 0;

    if (minimumSpanning != null) {
        for (Edge<Hub, Integer> edge : minimumSpanning.edges()) {
            totalDistance += edge.getWeight();
        }
    }

    return totalDistance;
}
```

## printMinimumSpanningTree e calculateTotalDistance:

O penúltimo método da classe `MinSpanningTreeResult` é o `printMinimumSpanningTree`. Este método imprime na saída padrão a representação visual da árvore de abrangência mínima. Se a árvore fornecida (`minimumSpanning`) for null, imprime uma mensagem indicando que a árvore é nula. Caso contrário, imprime as arestas da árvore no formato "Origem --(Distância(m))-- Destino". Por último temos o método `calculateTotalDistance`. Este método calcula e retorna a distância total da árvore de abrangência mínima. Se a árvore fornecida

(minimumSpanning) for null, retorna 0 indicando que a distância total é zero. Caso contrário, soma os pesos de todas as arestas na árvore para obter a distância total.

### Complexidade dos métodos da classe **MinSpanningTreeResult**:

getGraph():  $O(1)$

getMinimumSpanningTree: Geralmente  $O(E + V * \log(V))$  dependendo da implementação específica do algoritmo de Prim.

printMinimumSpanningTree:  $O(E)$

calculateTotalDistance:  $O(E)$

```
public static <V, E> MapGraph<V, E> minimumSpanningTree(MapGraph<V, E> g, Comparator<E> ce, BinaryOperator<E> sum, E zero) {
    Set<V> visitedVertices = new HashSet<>();

    PriorityQueue<Edge<V, E>> edgeQueue = new PriorityQueue<>(Comparator.comparing(e -> e.getWeight(), ce));

    MapGraph<V, E> minimumSpanningTree = new MapGraph<>( directed: true);

    V startVertex = g.vertices().iterator().next();
    visitedVertices.add(startVertex);

    for (Edge<V, E> edge : g.outgoingEdges(startVertex)) {
        edgeQueue.add(edge);
    }

    while (visitedVertices.size() < g.numVertices()) {
        Edge<V, E> minEdge = edgeQueue.poll();
        if (minEdge == null) {
            break;
        }

        V destVertex = minEdge.getVDest();

        if (!visitedVertices.contains(destVertex)) {
            visitedVertices.add(destVertex);

            minimumSpanningTree.addEdge(minEdge.getVOrig(), destVertex, minEdge.getWeight());

            for (Edge<V, E> edge : g.outgoingEdges(destVertex)) {
                edgeQueue.add(edge);
            }
        }
    }

    return minimumSpanningTree;
}
```

## **minimumSpanningTree:**

Este método é fundamental para a classe `MinSpanningTreeResult`, apesar de estar presente na classe `GraphAlgorithms`, pois é responsável por calcular a árvore de abrangência mínima utilizada nos métodos relacionados a essa classe, como `getMinimumSpanningTree` e outros. Ele representa uma implementação eficiente do algoritmo de Prim para encontrar uma árvore de abrangência mínima em um grafo ponderado não direcionado. Falando agora do método propriamente dito, este inicia as estruturas de dados, adiciona o `startVertex` aos vértices visitados e insere todas as arestas conectadas a `startVertex` na fila de prioridade. Executa um loop até que todos os vértices sejam visitados: Obtém a aresta de menor peso (`minEdge`) da fila de prioridade. Verifica se a fila de prioridade está vazia. Se sim, o grafo não está conectado e o loop é interrompido. Obtém o vértice de destino (`destVertex`) da aresta. Verifica se o vértice de destino já foi visitado. Se não, adiciona o vértice aos visitados, adiciona a aresta à árvore de abrangência mínima e insere todas as arestas conectadas ao vértice de destino na fila de prioridade. Por último retorna o grafo representando a árvore de abrangência mínima.

A complexidade geral do método `minimumSpanningTree` é aproximadamente  $O(E + V * \log(V))$  em termos de tempo e  $O(V + E)$  em termos de espaço. Esta complexidade faz do algoritmo de Prim uma escolha eficiente para encontrar árvores de abrangência mínima em grafos ponderados não direcionados.



