Problema 3: Grafos

Disponemos de un mapa de carreteras modelado como un grafo no dirigido compuesto por localizaciones y carreteras. Las carreteras representan los desplazamientos directos entre dos localizaciones.

Los objetos `Localizacion` tendrán las siguientes propiedades:

- id: String **// Identificador de la localización.**
- atractivo: Integer // **Atractivo turístico del enclave. El atractivo podrá tomar valores entre 0 y 3. 0 para las localizaciones sin interés, 1 para las localizaciones con interés paisajístico, 2 para las localizaciones con playa y 3 para las localizaciones con interés histórico.**
- hayCamasDisponibles: Boolean **// Indica si en la localización hay disponibilidad de camas hoteleras para pernoctar.**

Y los objetos `Trayecto` tendrán las siguientes propiedades:

- origen: Localizacion **// Una de las localizaciones que une el trayecto.**
- destino: Localizacion **// La otra localización que une el trayecto.**
- duracionEnHoras: Integer **// Duración del trayecto (en minutos).**

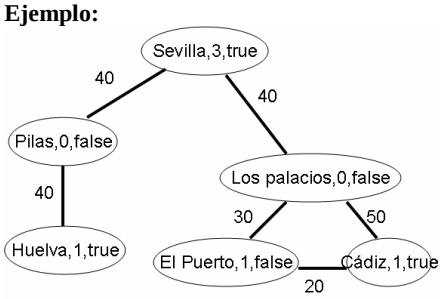
También se representará las travesías turísticas mediante listas de localizaciones, `List<Localizacion>`. La travesía turística representa la secuencia de escalas que realiza el turista donde cada par de escalas representan un trayecto directo entre dos localizaciones.

A partir de la información disponible en el grafo de rutas nos piden completar el código de la clase `MapaDeRutas` y el resto de clases necesarias:

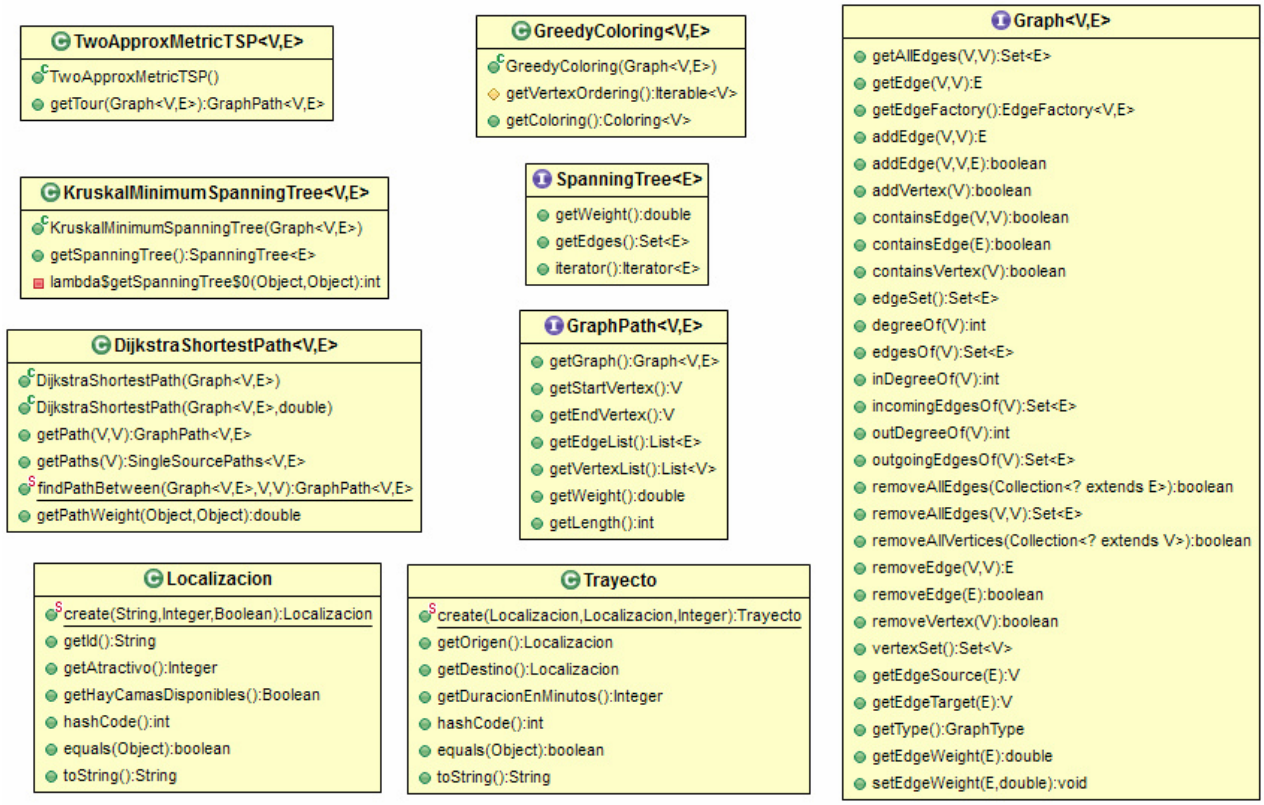
```
public class MapaDeRutas {

  private static Graph<Localizacion,Trayecto> g = null;

  1) Devuelve la localización asociada al identificador id.
  static public Localizacion getLocalizacionById(String id) { // TODO  }

  2) Devuelve un subgrafo con todas las localizaciones con cierto atractivo, los atractivos para el  filtrado se pasan en
  forma de array por parámetro. El subgrafo también contendrá los trayectos entre las localizaciones filtradas.
  static public Graph<Localizacion,Trayecto> getSubMapa(Integer ... atractivos) { // TODO    }

  3) Devuelve las zonas de influencia de una localización inicial condicionada por el atributo
  limiteDeMinutosXDesplazamiento. Se define la zona de influencia de la localización inicial como todas las
  localizaciones alcanzables desde la localización inicial con una duración menor o igual que el
  limiteDeMinutosXDesplazamiento. Además solo formarán parte de la zona de influencia las localizaciones que
  tengan camas para pernotar y un atractivo turístico mayor de cero.
  static public Set<Localizacion> getZonaDeInfluencia(String id,
                                int limiteDeMinutosXDesplazamiento) { // TODO }

  4) Devuelve el recubrimiento mínimo de trayectos necesarios para acceder a todas las localizaciones del mapa
  minimizando las horas de viaje.
  static public Set<Trayecto> getRedDeRecubrimiento() { // TODO }

  5) Devuelve la duración en horas de una travesía turísticas.
  static public Double getDuracionDeTravesia(List<Localizacion> travesia) { // TODO  }

}
```

NOTA: Las funciones deberán reutilizar el código siempre que sea posible.

Ejemplo:




```
getLocalizacionById("Sevilla")=Sevilla(a:3, c:true)

getSubMapa(0,1)=[[Pilas, Los Palacios, Huelva, Cadiz,
Puerto], [Huelva–Pilas, Los Palacios–Cadiz,Los Palacios–
Puerto, Puerto–Cadiz]]

getDuracionDeTravesia([sevilla,lpalacios,puerto])=70

getZonaDeInfluencia("Sevilla",80)={Huelva, Puerto}

getRedDeRecubrimiento()=[Puerto--Cadiz, Huelva--Pilas,
Los Palacios–Sevilla, Sevilla--Pilas, Los Palacios--Puerto]
```



Solución:

```
static public Localizacion getLocalizacionById(String id) {
  return g.vertexSet().stream().filter(v -> v.getId().equals(id)).findFirst().orElse(null);
}

static public Graph<Localizacion,Trayecto> getSubMapa(Integer ... atractivos) {
  final Graph<Localizacion,Trayecto> gAux =
         new SimpleWeightedGraph<Localizacion, Trayecto>(Trayecto.class);
  Arrays.asList(atractivos).stream().forEach(
       a -> g.vertexSet().stream().filter(v -> v.getAtractivo().equals(a))
              .forEach (vv -> gAux.addVertex(vv)));

  g.edgeSet().stream()
       .filter(e -> gAux.vertexSet().contains(e.getDestino()))
       .filter(e -> gAux.vertexSet().contains(e.getOrigen()))
       .forEach(e -> gAux.addEdge(e.getOrigen(), e.getDestino(),e));

  gAux.edgeSet().stream().forEach(e -> gAux.setEdgeWeight(e, e.getDuracionEnMinutos()));

  return gAux;
}

static public Set<Localizacion> getZonaDeInfluencia(String id, int limiteDeMinutosXDesplazamiento) {
  final Localizacion lini = getLocalizacionById(id);
  ShortestPathAlgorithm<Localizacion,Trayecto> a = new DijkstraShortestPath<Localizacion,Trayecto>(g);
  return g.vertexSet().stream().filter(v -> v.getAtractivo()!=0)
       .filter(v -> !v.equals(lini))
       .filter(v -> a.getPath(lini,v).getWeight()<=limiteDeMinutosXDesplazamiento)
       .collect(Collectors.toSet());
}

static public Set<Trayecto> getRedDeRecubrimiento() {
  SpanningTreeAlgorithm<Trayecto> ast = new KruskalMinimumSpanningTree<>(g);
  SpanningTree<Trayecto> r = ast.getSpanningTree();
  return r.getEdges();
}

static public Double getDuracionDeTravesia(List<Localizacion> travesia) {
  return IntStream.range(0,travesia.size()-1).boxed()
   .mapToDouble(i -> g.getEdge(travesia.get(i), travesia.get(i+1)).getDuracionEnMinutos())
   .sum();
}
```