PRÁCTICA 4

Aplicaciones de grafos en Java. Red de carreteras

Objetivos

- Estudiar los grafos y su forma de representación (mapa de mapas).
- Aprender a utilizar la estructura de datos para grafos y los algoritmos básicos para aplicarlos sobre un problema relacionado con una red de carreteras.

Requerimientos

Para superar esta práctica se debe realizar lo siguiente:

- Dominar las estructuras de datos para implementar un grafo (no dirigido) y su uso para la resolución de problemas sencillos (en este caso, un problema de red de carreteras).
- Conocer cómo implementar algoritmos básicos sobre grafos para responder a consultas sencillas de determinados problemas (en este caso, un problema de red de carreteras).

Enunciado

Una de las principales aplicaciones prácticas de los grafos son las redes, como por ejemplo: la red de carreteras, la red de ferrocarriles, las redes informáticas, etc. En esta práctica vamos a estudiar problemas que se suelen plantear sobre una red o mapa de carreteras (de España) representada mediante un grafo ponderado (positivamente) no orientado (no dirigido) y representado en la Figura 1. En dicho grafo, cada nodo o vértice representaría a un núcleo de población, y cada arista indicaría el "camino real" que conecta a dos núcleos de población (el peso asociado a la arista, en este caso coincide con la distancia que separa a los núcleos de población).

Figura 1. Grafo correspondiente al mapa de carreteras de España.

Sobre dicho grafo, son muchas las preguntas que nos podemos plantear, entre ellas destacamos las siguientes:

- ¿Cuál es el camino más corto de una ciudad a otra?
- ¿Cuál es el camino más corto entre una ciudad y todas las demás ciudades del grafo?
- ¿Existen caminos mínimos entre todos los pares de ciudades?, ¿cuáles son?
- ¿Cuál es la ciudad más lejana a una dada?
- ¿Cuál es la ciudad más céntrica?
- ¿Cuántos caminos distintos existen de una ciudad a otra?
- ¿Cómo hacer un tour entre todas las ciudades en el menor coste (distancia) posible?
- etc.

Repasemos algunos de los conceptos más importantes sobre grafos:
Un **grafo G** es una tupla G = (V, A), donde V es un conjunto no vacío de nodos o vértices y A es un conjunto de aristas o arcos. Cada **arista** es un par (v_x, v_y) de nodos, donde v_x, $v_y \in V$. Las aristas son relaciones muchos a muchos entre nodos. Según el tipo de arista, existen dos tipos de grafos: orientados y no orientados. En un **grafo no orientado** (no dirigido), las aristas no están ordenadas: (v_x, v_y) = (v_y, v_x). Mientras que en un **grafo orientado** (dirigido o digrafo), las aristas son pares ordenados: $<v_x, v_y> \neq <v_y, v_x>$; $<v_x, v_y> \Rightarrow v_y$ = destino de la arista, v_x = origen. Un **grafo ponderado (valorado)**, es un grafo en el que cada arista tiene asociada un valor de cierto tipo (numérico). Es decir, un grafo ponderado: G = (V, A, W), con W: A $\rightarrow$ TipoEtiqueta (e.g. W: A $\rightarrow \Re$). En esta práctica haremos uso de **grafos ponderados no orientados** con valores de distancias ($\in \Re^+$) asociados a las aristas.

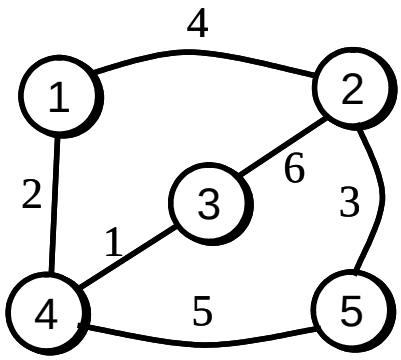
Nodos adyacentes a un nodo v son todos los nodos unidos a v mediante una arista. **Camino de un nodo v_1 a otro v_q**, es una secuencia $v_1, v_2, ..., v_q \in V$, tal que todas las aristas (v_1, v_2), (v_2, v_3), ..., (v_{q-1}, v_q) $\in$ A. La **longitud de un camino** es el número de aristas del camino = nº de nodos menos uno. **Camino simple** es aquel en el que todos los nodos son distintos (excepto el primero y el último que pueden ser iguales). **Ciclo** es un camino en el cual el primer y el último nodo son iguales. Un **subgrafo** de G = (V, A) es un grafo G' = (V', A') tal que V' $\subseteq$ V y A' $\subseteq$ A. Dados dos nodos v_x y v_y, se dice que están **conectados** si existe un camino de v_x a v_y. Un grafo es **conexo** (o conectado) si hay un camino entre cualquier par de nodos. Un grafo es **completo** si existe una arista entre cualquier par de nodos, es decir, cada nodo de G es adyacente a todos los demás nodos de G, por lo que el número de aristas de un grafo completo de v nodos es (v * (v – 1)) / 2.

Existen dos maneras típicas de representar un grafo:

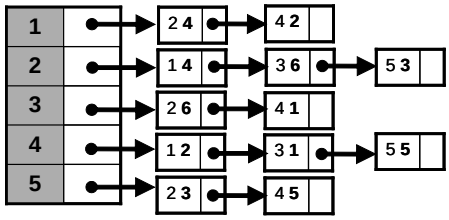
- Mediante matrices de adyacencia
- Mediante listas de adyacencia

Sea un grafo ponderado G = (V, A, W) de *n* nodos, donde suponemos que dichos nodos $v_1, v_2, ..., v_n \in$ V están ordenados y podemos representarlos por sus ordinales {1, 2, ..., n}. La **matriz de adyacencia** para el grafo G es una matriz M de dimensión **n × n** de elementos de W en la que: M[i, j] = w $\in$ W, si y sólo si existe una arista en G que va del nodo *i* al nodo *j*; M[i, j] = 0.0, en caso contrario.
En el caso de una **lista de adyacencia**, dicha representación consiste en *n* listas, de forma que la lista *i*-ésima contiene los nodos adyacentes al nodo *i*.

Veamos un ejemplo para un grafo ponderado no orientado con una matriz de adyacencia (problema: matriz simétrica M[i, j] = M[j, i] y se desperdicia la mitad de memoria) y con una lista de adyacencia (problema: las aristas también están repetidas ⇒ también se desperdicia memoria).



M	1	2	3	4	5
1		4		2	
2	4		6		3
3		6		1	
4	2		1		5
5		3		5	



Las operaciones básicas sobre grafos serían las siguientes:

- Crear un grafo vacío (o con *n* nodos)
- Insertar un nodo o una arista
- Eliminar un nodo o arista
- Consultar si existe una arista (obtener el valor asociado a dicha arista)
- Iteradores sobre las aristas de un nodo:
  - Para todo nodo v_y adyacente a v_x: hacer → acción sobre v_y
  - Para todo nodo v_y adyacente de v_x: hacer → acción sobre v_y

Algoritmos más comunes sobre grafos.

Recorrido en profundidad, es equivalente a un recorrido en preorden de un árbol. Para resolver este problema necesitamos tener un array que nos indique si hemos visitado un nodo o no. A este array [1..n] le llamaremos *visitados* y será de booleanos, donde *false* indicará nodo no visitado y *true* nos informará que ese nodo ya ha sido visitado.

```
algoritmo recorridoEnProfundidad()
        desde v = 1 hasta n hacer
                visitados[v] = false;
        fin desde
        desde v = 1 hasta n hacer
                si visitados[v] == false entonces dfs(v);
        fin desde
fin algoritmo

algoritmo dfs(v: Nodo)
        visitados[v] = true;
        para cada nodo w adyacente a v hacer
                si visitados[w] == false entonces dfs(w);
        fin para
fin algoritmo
```

Recorrido en anchura, es equivalente a recorrer un árbol por niveles (en amplitud). Este recorrido empieza visitando un nodo v, luego se visitan todos sus adyacentes, luego los adyacentes de estos, y así sucesivamente. El algoritmo que implementa este recorrido utiliza una cola de nodos, *cola*, cuyas operaciones básicas empleadas son: sacar un nodo del frente de la cola (peek y pop), y añadir a la cola sus adyacentes no visitados (push). Además, para resolver este problema, también necesitamos tener un array (de booleanos) que nos indique si hemos visitado ya un nodo o no. Es decir, *visitados* es un array [1..n] de booleanos.

```
algoritmo recorridoEnAnchura
    desde v = 1 hasta n hacer
        visitados[v] = false;
    para v = 1 hasta n hacer
        si visitados[v] == false entonces bfs(v);
    fin para
fin algoritmo

algoritmo bfs(v: Nodo)
    queue<nodo> cola;
    visitados[v] = true;
    cola.push(v);
    mientras (!cola.empty()) hacer
        x = cola.peek();
        cola.pop();
        para cada nodo y adyacente a x hacer
            si visitado[y] == false entonces
                visitados[y] = true;
                cola.push(y);
            fin si
        fin para
    fin mientras
fin algoritmo
```

El **algoritmo de Dijkstra** encuentra el *camino mínimo entre un nodo origen y cada uno de los otros nodos de un grafo* G = (V, A). Se considera el nodo 1 como nodo origen. El número de nodos del grafo es *n*. M es un array bidimensional [1..n, 1..n] de reales ⇒ Matriz de adyacencia (costes) de tamaño n × n, almacenado M[i, j] = 0 si i = j, M[i, j] = w_{ij} si los nodos *i* y *j* están conectados, y M[i, j] = ∞ en caso contrario. D es un array [2..n] de reales ⇒ array costes de caminos mínimos. D es un array formado por las distancia del nodo origen a cada uno de los otros nodos de G. Es decir, D[i] almacena la menor distancia entre el nodo origen y el nodo *i*. S es un array [2..n] de booleanos ⇒ array relativo a los nodos de los cuales ya se conoce la distancia mínima entre ellos y el origen.

```
algoritmo Dijkstra
    desde j = 2 hasta n hacer
        D[j] = M[1, j];
        S[j] = false;
    fin desde
    desde i = 2 hasta n hacer
        Elegir v = nodo con S[v] = false (v ∈ V – S) y mínimo D[v];
        S[v] = true;
        para cada nodo w adyacente a v (w ∈ V – S) hacer
            // D[w] = min{D[w], D[v] + M[v, w]}
            si ((S[w] == false) && (D[v] + M[v, w] < D[w]) entonces
                D[w] = D[v] + M[v, w];
            fin si
        fin para
    fin desde
fin algoritmo
```

El **algoritmo de Floyd** encuentra *el camino mínimo entre todos los posibles pares de nodos del grafo,* siendo n el número de nodos. M es un array bidimensional [1..n, 1..n] de reales ⇒ Matriz de costes de tamaño n × n. D es un array bidimensional [1..n, 1..n] de reales ⇒ matriz de costes de caminos mínimos de tamaño (n × n), con los costes o pesos de las aristas, de tal forma que cada elemento de la matriz representa el peso coste c_{ij} asociado a la arista (v_i, v_j). Usaremos $c_{ij} = \infty$ para indicar que no existe arco entre los vértices v_i y v_j y además debemos fijar $c_{ii} = 0$. En este array D se almacenará el resultado final del algoritmo. Es decir, en la *k*-ésima iteración D[i, j] tendrá el camino de menor coste para llegar de *i* a *j*, pasando por un número de nodos menor que *k* (matriz resultado), el cual se calcula según la siguiente expresión:

$$D_k[i, j] = \begin{cases} c_{ij} & si \;\; k = -1 \\ \min\{D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j]\} & si \;\; k \geq 0 \end{cases}$$

Se elegirá el camino más corto entre el valor obtenido en la iteración *k – 1*, y el que resulta de pasar por el nodo *k*.

```
algoritmo Floyd
    D = M;      // Haciendo 0 la diagonal principal (coste de las aristas (v_i, v_i), c_ii = 0)
    P = 0;      // Se utiliza para indicar el camino seguido
    desde k = 1 hasta n hacer
        desde i = 1 hasta n hacer
            desde j = 1 hasta n hacer
                D[i, j] = min{D[i, j], D[i, k] + D[k, j]};
                P[i, j] = k;
            fin desde
        fin desde
    fin desde
fin algoritmo
```

El **algoritmo de Prim** se utiliza para determinar el *árbol de recubrimiento de coste mínimo de un grafo* G = (V, A) con *n* nodos. U y L son estructuras de datos (arrays o listas) que permiten guardar los nodos añadidos a la solución ($U \subseteq V$) y de aristas que se va formando con las aristas de menor coste que se van seleccionando, respectivamente. Inicialmente, L está vacía ($L = \{\varnothing\}$) y $U = \{\{v_1\}\}$

algoritmo Prim
  mientras $V \neq U$ **hacer**
    Elegir una arista $(v_x, v_y) \in A$ que tenga el menor coste, tal que $v_x \in U$ y $v_y \in (V - U)$
    $L = L \cup \{(v_x, v_y)\}$
    $U = U \cup \{v_y\}$
  fin mientras
fin algoritmo

Otro esquema equivalente del algoritmo de **Prim** es el siguiente: **(1)** Empezar en un nodo cualquiera v. El árbol consta inicialmente sólo del nodo v. **(2)** Del resto de nodos, buscar el que esté más próximo a v (es decir, con la arista (v, w) de coste mínimo). Añadir el nodo w y la arista (v, w) al árbol. **(3)** Buscar el nodo más próximo a cualquiera de estos dos. Añadir ese nodo y la arista al árbol de expansión. **(4)** Repetir sucesivamente hasta añadir los *n* nodos.

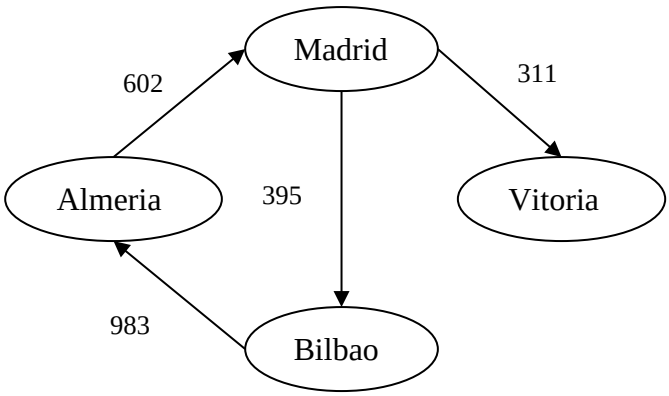
Ejercicios prácticos a realizar.

El formato de archivo de entrada para el grafo de carreteras será el que se muestra a continuación:

```
(0) No Dirigido || (1) Dirigido
# de nodos
Nodo_1
…
Nodo_n
# de aristas
Nodo_i Nodo_j Coste_i_j
```

Por ejemplo, el siguiente archivo daría lugar al grafo que se muestra a continuación:

```
1
4
Almeria
Madrid
Vitoria
Bilbao
4
Almeria Madrid 602
Madrid Vitoria 311
Bilbao Almería 983
Madrid Bilbao 395
```



Dada la Figura 2 que represente al grafo (ponderado no orientado) que representa un mapa parcial de carreteras de España (reducido, sólo para esta práctica), se debe crear el archivo de entrada, sabiendo que es no orientado (0), que tiene un total de 21 ciudades y 29 aristas. El archivo se denominará "**graphSpain.txt**".

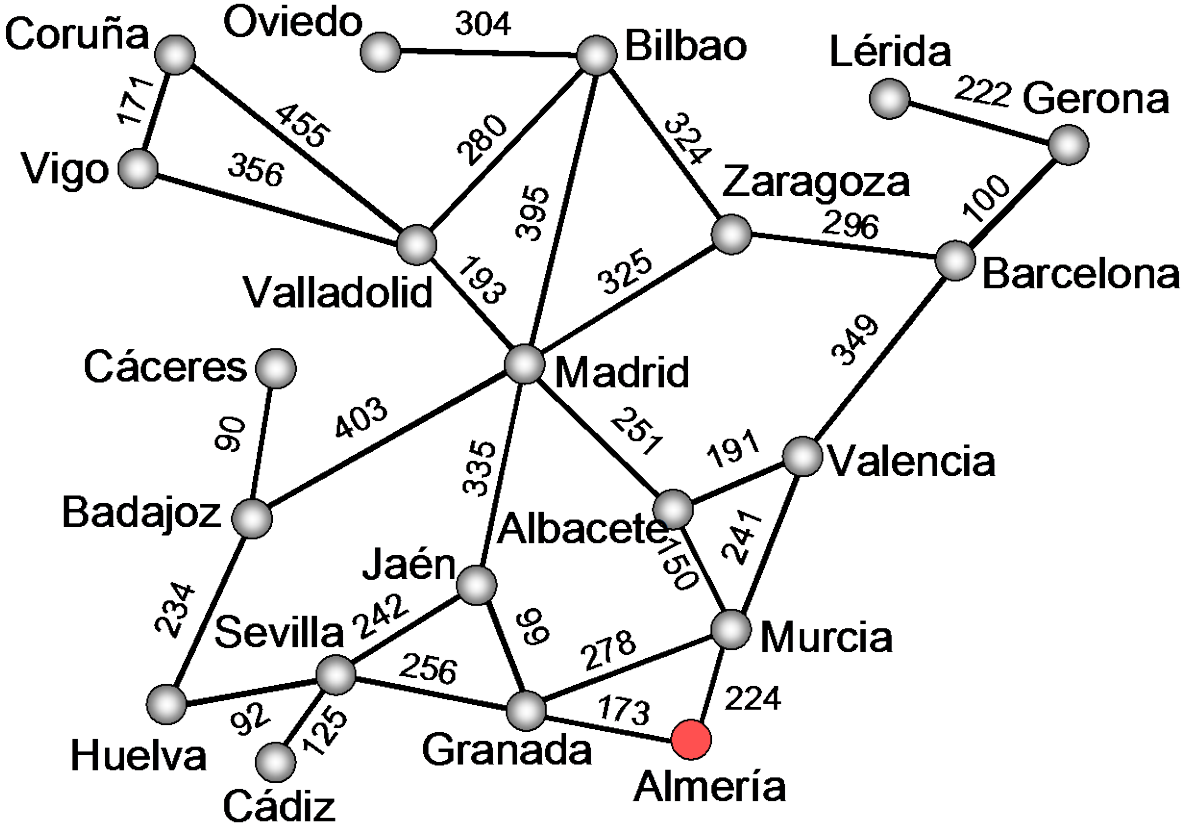


Figura 2. Grafo correspondiente al mapa parcial de carreteras de España.

Igual que para la actividad correspondiente a grafos, se va a proporcionar parcialmente la clase `RoadNetwork` (`RoadNetwork.java`) que va a implementar la interface `Graph` (proporcionada en el paquete `estructurasdedatos`), junto con una serie de operaciones adicionales. Uno de los aspectos más destacables de esta clase son los atributos o datos miembro que ésta contiene, siendo algunos de ellos los siguientes (cuya descripción se detalla a continuación).

```
protected TreeMap<Vertex, TreeMap<Vertex, Double>> adjacencyMap;
```

Declaración de la estructura de datos donde se almacenará el grafo (`Network`). Ésta es un mapa (`TreeMap`) de vértices (`Vertex`), cuyos vértices adyacentes se almacenan también en un mapa (`TreeMap`) de vértices (`Vertex`) con sus respectivos pesos (`Double`). Destacar que `Vertex` debe ser una clase que implemente `Comparable`.

```
protected TreeMap<Vertex, Boolean> visited;
```

Este mapa (estructura de datos auxiliar) se utilizará en muchas funciones para ir marcando (`true`) los vértices que se vayan procesando, como por ejemplo, recorridos y funciones derivadas de ellos. Existen otras posibles alternativas para implementar **`visited`**, otra podría ser con un `TreeSet<Vertex>`.

```
protected ArrayList<Vertex> result;
```

Este `ArrayList` es una estructura de datos auxiliar que contendrá el resultado final de una determinada operación, siendo realmente una colección de vértices resultado.

```
protected ArrayList<ArrayList<Vertex>> resultSimplePaths;
```

Estructura de datos auxiliar para almacenar el resultado de la operación *caminos simples*.

También se proporcionará el archivo de texto con la estructura del grafo de la red de carreteras de España (*graphSpain.txt*), que se deberá de cargar en memoria para poder realizar las consultas que se indican a continuación

Una vez cargada la estructura en memoria (haciendo uso de la función `readRoadNetwork`), realizar la correspondiente implementación en Java, para responder a los siguientes ejercicios, aplicando los algoritmos apropiados.

1. Partiendo desde **Almería**, mostrar un recorrido en profundidad y otro en anchura del grafo de carreteras, utilizando los algoritmos vistos anteriormente y estudiados en clase. Es general, implementar una función que partiendo de una ciudad elegida por el usuario, muestre un recorrido en profundidad y otro en anchura del grafo de carreteras.

2. ¿Cuál es el camino más corto de **Almería** a **Oviedo**?, ¿y cuál el de **Almería** a **Cáceres**?. En ambos casos, mostrar el itinerario seguido (ciudades visitadas). En general, implemente un método que realice esta tarea (camino más corto entre dos ciudades dadas, utilizando el algoritmo de *Dijkstra*) para un par de ciudades cualquiera que decida el usuario. Conociendo esto, ¿Cuál es la distancia del camino más corto de **Almería** a **Lérida**, pasando por **Madrid**?, indicando también su itinerario.

3. ¿Cuál es el árbol de caminos mínimos (o Shortest-Paths Tree, SPT) que permite calcular el camino mínimo entre **Almería** y todas los demás ciudades del grafo (árbol de caminos mínimos de *Dijkstra*)?. ¿Debe de coincidir este árbol con el árbol de recubrimiento de coste mínimo empezando desde **Almería**?. Razone las respuestas.

4. ¿Cuál es la ciudad más lejana a **Almería** y la siguiente más lejana (segunda más lejana), en función del algoritmo que determina el camino mínimo?. Mostrar los itinerarios seguidos para cada cuidad destino. En general, implemente una función que, partiendo de una ciudad dada, muestre las ciudades con las que puede conectarse en orden decreciente según la distancia del camino mínimo que las une y sus correspondientes itinerarios.

5. ¿Cuántos caminos distintos existen entre **Almería** y **Bilbao**?. Indicar en la memoria cuáles son dichos caminos (uno por uno). En general, implemente una función que muestre todos los posibles caminos distintos que existen entre dos ciudades cualesquiera dadas por el usuario (el algoritmo para su resolución es el algoritmo de *caminos simples*).

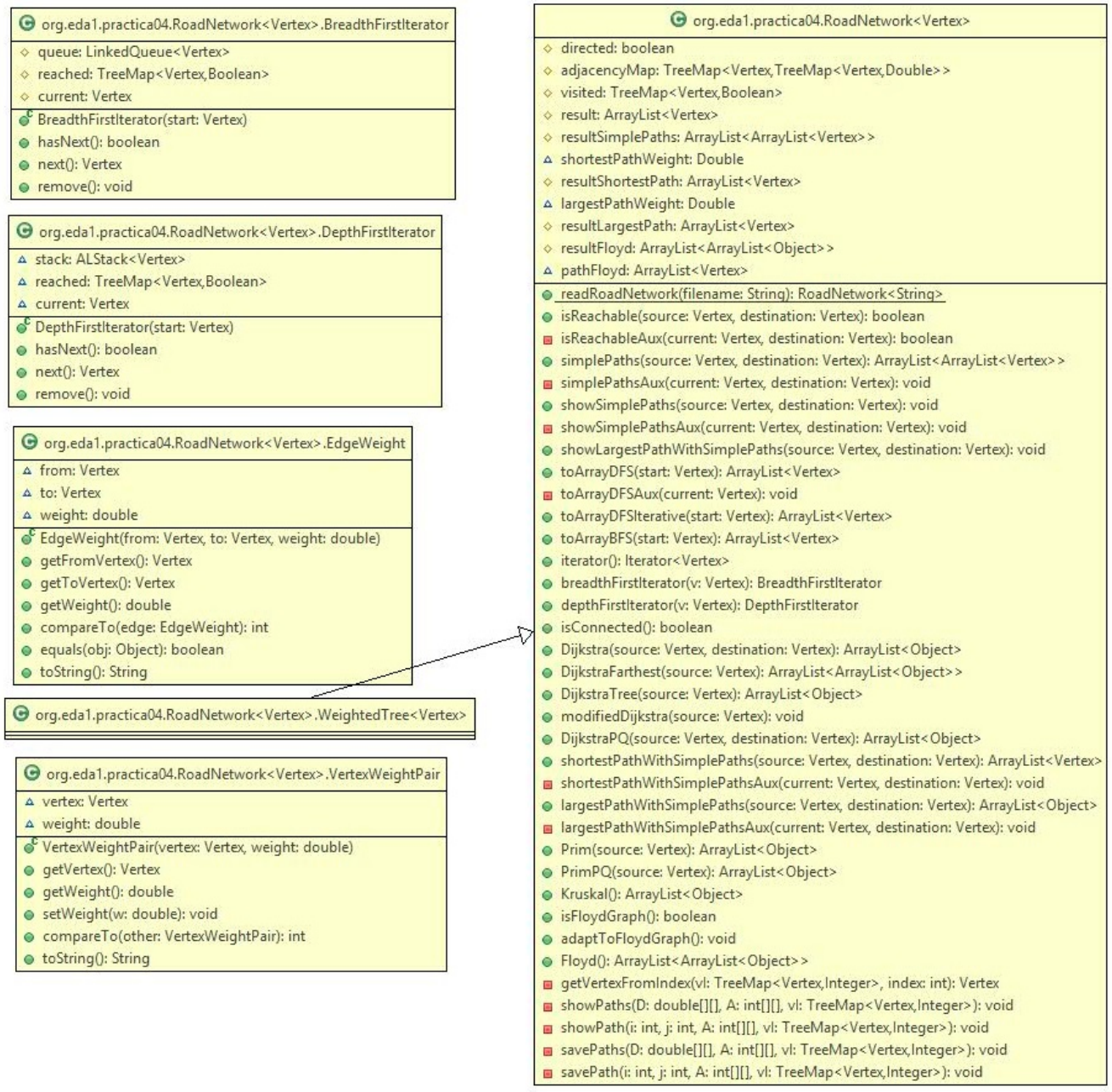
6. ¿Cuál es el camino más largo (con mayor distancia posible) de **Almería** a **Oviedo**?. Mostrar el itinerario seguido (ciudades visitadas). En general, implemente un método que realice esta tarea (camino más largo) para un par de ciudades cualesquiera que decida el usuario.

7. ¿Cuántos caminos mínimos *distintos* existen entre todas las ciudades del mapa (entre todos los posibles pares de ciudades de España)?. Indicar de qué forma se pueden obtener. Además, se debe comprobar que el número total de caminos mínimos es 420, enumerándolos todos ellos por pantalla. Como aplicación de este algoritmo (algoritmo de *Floyd*), implementar una función que devuelva los itinerarios de todos los caminos mínimos de pares de ciudades cuya distancia esté comprendida entre dos distancias dadas $[d_1, d_2]$, siendo $d_1 < d_2$. Además, implemente otra función que devuelva el camino mínimo (con su itinerario y distancia) entre dos pares de ciudades, con la menor y mayor distancia. Finalmente, implemente una última función que devuelva todos los caminos mínimos que tienen como origen **Madrid** y destino las restantes ciudades del grafo.

8. Conectar y mostrar todas las ciudades de España con la menor distancia total (es decir, dado un grafo ponderado no orientado, encontrar el árbol de recubrimiento de coste mínimo o Minimum Spanning Tree, MST). Con este árbol resultante, ¿podemos determinar el camino mínimo entre dos pares de ciudades cualquiera?.

Además de pasar los test que se proporcionan, se deberá desarrollar una memoria que consistirá en un archivo PDF con nombre **practica04** y que se almacenará en la carpeta denominada **Memorias** en el proyecto de Practicas de cada alumno. El archivo deberá contener las respuestas a todas las preguntas planteadas en todos los ejercicios propuestos, independientemente de que las salidas de algunos de ellos sean muy extensas.

Para la resolución de estos ejercicios se ha propuesto utilizar el `TreeMap`, como estructura de datos para representar un grafo (en este caso un grafo no dirigido y valorado). ¿Sería muy complicado realizar la misma implementación con `HashMap`?, ¿qué habría que hacer?. ¿Qué diferencia existe entre `TreeMap` y `HashMap`?. Reflejad las respuestas a estas preguntas en el archivo **practica04** creado anteriormente en la carpeta **Memorias**.


org.eda1.practica04.RoadNetwork<Vertex>.BreadthFirstIterator
queue: LinkedQueue<Vertex>
reached: TreeMap<Vertex,Boolean>
current: Vertex
BreadthFirstIterator(start: Vertex)
hasNext(): boolean
next(): Vertex
remove(): void
org.eda1.practica04.RoadNetwork<Vertex>.DepthFirstIterator
stack: ALStack<Vertex>
reached: TreeMap<Vertex,Boolean>
current: Vertex
DepthFirstIterator(start: Vertex)
hasNext(): boolean
next(): Vertex
remove(): void
org.eda1.practica04.RoadNetwork<Vertex>.EdgeWeight
from: Vertex
to: Vertex
weight: double
EdgeWeight(from: Vertex, to: Vertex, weight: double)
getFromVertex(): Vertex
getToVertex(): Vertex
getWeight(): double
compareTo(edge: EdgeWeight): int
equals(obj: Object): boolean
toString(): String
org.eda1.practica04.RoadNetwork<Vertex>.WeightedTree<Vertex>
org.eda1.practica04.RoadNetwork<Vertex>.VertexWeightPair
vertex: Vertex
weight: double
VertexWeightPair(vertex: Vertex, weight: double)
getVertex(): Vertex
getWeight(): double
setWeight(w: double): void
compareTo(other: VertexWeightPair): int
toString(): String
org.eda1.practica04.RoadNetwork<Vertex>
directed: boolean
adjacencyMap: TreeMap<Vertex,TreeMap<Vertex,Double>>
visited: TreeMap<Vertex,Boolean>
result: ArrayList<Vertex>
resultSimplePaths: ArrayList<ArrayList<Vertex>>
shortestPathWeight: Double
resultShortestPath: ArrayList<Vertex>
largestPathWeight: Double
resultLargestPath: ArrayList<Vertex>
resultFloyd: ArrayList<ArrayList<Object>>
pathFloyd: ArrayList<Vertex>
readRoadNetwork(filename: String): RoadNetwork<String>
isReachable(source: Vertex, destination: Vertex): boolean
isReachableAux(current: Vertex, destination: Vertex): boolean
simplePaths(source: Vertex, destination: Vertex): ArrayList<ArrayList<Vertex>>
simplePathsAux(current: Vertex, destination: Vertex): void
showSimplePaths(source: Vertex, destination: Vertex): void
showSimplePathsAux(current: Vertex, destination: Vertex): void
showLargestPathWithSimplePaths(source: Vertex, destination: Vertex): void
toArrayDFS(start: Vertex): ArrayList<Vertex>
toArrayDFSAux(current: Vertex): void
toArrayDFSIterative(start: Vertex): ArrayList<Vertex>
toArrayBFS(start: Vertex): ArrayList<Vertex>
iterator(): Iterator<Vertex>
breadthFirstIterator(v: Vertex): BreadthFirstIterator
depthFirstIterator(v: Vertex): DepthFirstIterator
isConnected(): boolean
Dijkstra(source: Vertex, destination: Vertex): ArrayList<Object>
DijkstraFarthest(source: Vertex): ArrayList<ArrayList<Object>>
DijkstraTree(source: Vertex): ArrayList<Object>
modifiedDijkstra(source: Vertex): void
DijkstraPQ(source: Vertex, destination: Vertex): ArrayList<Object>
shortestPathWithSimplePaths(source: Vertex, destination: Vertex): ArrayList<Vertex>
shortestPathWithSimplePathsAux(current: Vertex, destination: Vertex): void
largestPathWithSimplePaths(source: Vertex, destination: Vertex): ArrayList<Object>
largestPathWithSimplePathsAux(current: Vertex, destination: Vertex): void
Prim(source: Vertex): ArrayList<Object>
PrimPQ(source: Vertex): ArrayList<Object>
Kruskal(): ArrayList<Object>
isFloydGraph(): boolean
adaptToFloydGraph(): void
Floyd(): ArrayList<ArrayList<Object>>
getVertexFromIndex(vl: TreeMap<Vertex,Integer>, index: int): Vertex
showPaths(D: double[][], A: int[][], vl: TreeMap<Vertex,Integer>): void
showPath(i: int, j: int, A: int[][], vl: TreeMap<Vertex,Integer>): void
savePaths(D: double[][], A: int[][], vl: TreeMap<Vertex,Integer>): void
savePath(i: int, j: int, A: int[][], vl: TreeMap<Vertex,Integer>): void