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.

Requisitos

Para superar esta práctica se debe realizar lo siguiente:

- Dominar las estructuras de datos para implementar un grafo y su uso para la resolución de problemas sencillos (en este caso, un problema de red de carreteras).
- Saber cómo implementar algoritmos básicos sobre grafos para responder a consultas sencillas en determinados problemas.

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, redes sociales, 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 (o 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 las que destacamos:

- ¿Cuál es el camino más corto de una ciudad a otra?
- ¿Cuál es el camino más corto entre una ciudad y el resto de ciudades?
- ¿Existen caminos mínimos entre todos los pares de ciudades?. En caso afirmativo, ¿cuáles?
- ¿Cuál es la ciudad más lejana a una ciudad especificada por el usuario?
- ¿Cuál es la ciudad más céntrica?
- ¿Cuántos caminos distintos existen de una ciudad a otra?
- ¿Cómo hacer un tour entre todas las ciudades con el menor coste (distancia) posible?
- Y un largo etcétera...

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: $\langle v_x, v_y \rangle \neq \langle v_y, v_x \rangle$; $\langle v_x, v_y \rangle \Rightarrow v_y$ = 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 → TipoEtiqueta (e.g. W: A → $\Re$). En esta práctica haremos uso de **grafos ponderados no orientados** con valores de distancias ($\in \Re^+$) asociados a las aristas.

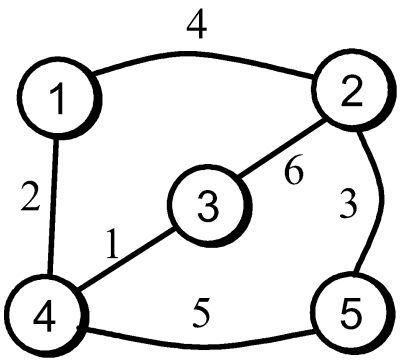
Nodos adyacentes a un nodo v son todos los nodos unidos a v mediante una arista. **Camino de un nodo v_1 a otro v_q**, es una secuencia $v_1, v_2, ..., v_q \in V$, tal que todas las aristas $(v_1, v_2), (v_2, v_3), ..., (v_{q-1}, v_q) \in A$. La **longitud de un camino** es el número de aristas del camino = 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:

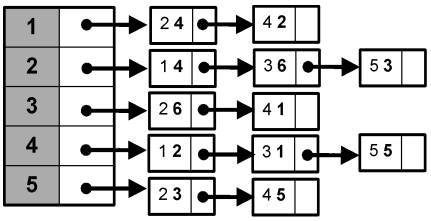
- Mediante matrices de adyacencia
- Mediante listas de adyacencia

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 **nxn** de elementos de W en la que: M[i, j] = w ∈ 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]…desperdicio de memoria) y con una lista de adyacencia (con el mismo problema, pero mucho menor).



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
```

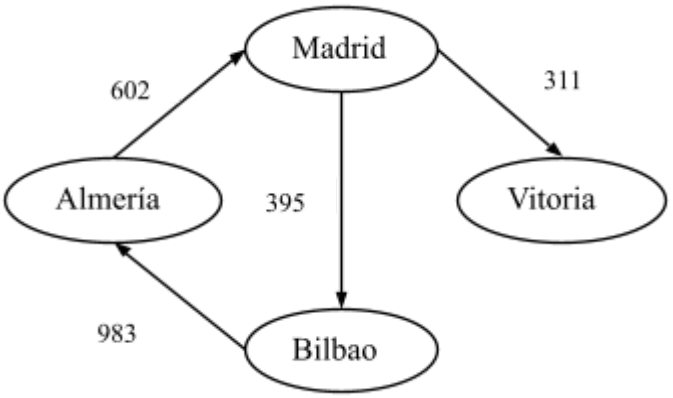
Ejercicios prácticos a realizar.

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

```
%Ejemplo de grafo
@Type
Not Directed

@Vertex
v1
v2
…
v_n

@Edges
v_i v_i w_ij
```

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

```
@Vertex
Almería
Madrid
Vitoria
Bilbao

@Edges
Almería 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 específicamente 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 "**data.txt**".

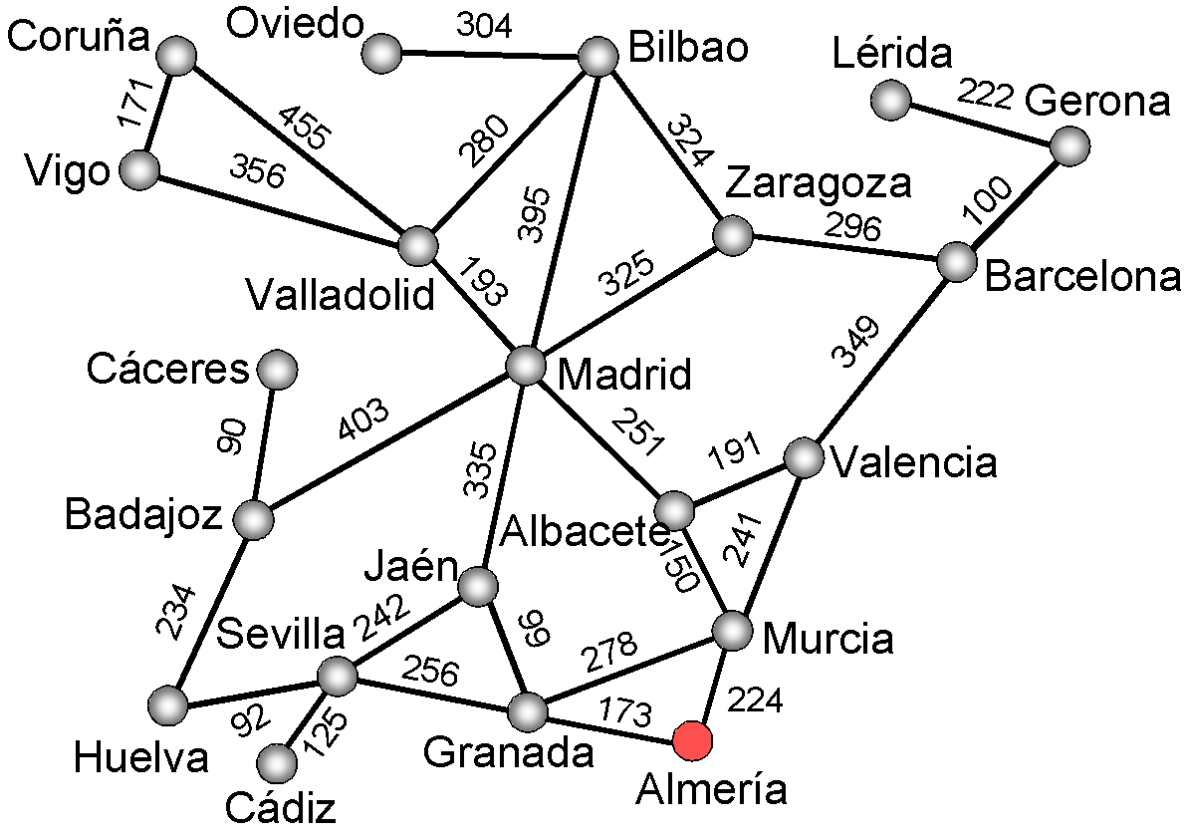


Figura 2. Grafo correspondiente al mapa parcial de carreteras de España.

Para la realización de la práctica, se os va a proporcionar la clase `RoadNetwork` (`RoadNetwork.java`) que extiende la clase `Network<String>`, ambas parcialmente implementadas. A su vez, `Network<Vertex>` implementa la interfaz `Graph<T>`, extendiéndola con una serie de operaciones adicionales. Uno de los aspectos más destacables de esta clase son los atributos o datos miembro que contiene y que describimos a continuación:

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

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

También se proporcionará el archivo de texto con la estructura del grafo de la red de carreteras de España (*data.txt*), que se deberá de cargar en memoria para poder realizar las consultas que se indican en el test proporcionado. Para la resolución de esta práctica se ha propuesto utilizar la estructura `TreeMap<K,V>` 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<K,V>`? ¿Qué habría que hacer? ¿Qué diferencias fundamentales existen entre las estructuras `TreeMap<K,V>` y `HashMap<K,V>`?.