EDA (ETS Ingeniería Informática). Curso 2016-2017

Práctica 6. Obtención de caminos entre municipios de una red como aplicación de la exploración de un *Grafo*

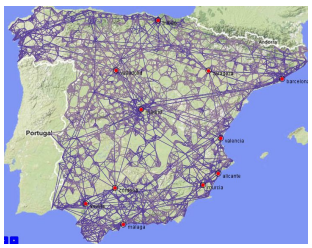
Departamento de Sistemas Informáticos y Computación. UPV

1. Descripción general, planificación y resultados de aprendizaje

Son muchos los problemas prácticos de interés que pueden ser planteados en términos de problemas sobre grafos. El objetivo de esta práctica es la implementación y utilización del algoritmo de **Dijkstra** para calcular de forma eficiente el camino más corto entre dos municipios en una red (mapa) cuyos datos se leen de un fichero.

La práctica se realizará durante 3 sesiones y al finalizarla el alumno debe ser capaz de:

- Usar la implementación de la jerarquía Java de representación de un Grafo Dirigido y Ponderado en el que los vértices están numerados entre 0 y $|V| - 1$; esto es de las clases `Adyacente`, `Grafo` y `GrafoDirigido`.
- Implementar en Java el algoritmo de Dijkstra.
- Implementar de forma eficiente un Grafo Dirigido y Ponderado en el que los vértices contienen la información sobre los municipios de la red; esto es, las clases `Municipio` y `GrafoMunicipios`. En particular se implementará la obtención del camino mínimo a partir de los datos obtenidos del uso del algoritmo de Dijkstra.
- Usar la implementación gráfica que se proporciona, `GuiGrafo`, para comprobar el funcionamiento del código realizado.



2. Jerarquía `grafos` e implementación del algoritmo de Dijkstra

El algoritmo de *Dijkstra*, también llamado de *caminos mínimos*, permite obtener el camino más corto desde un vértice origen al resto de vértices en un grafo dirigido, con pesos en cada arista y sin ciclos de peso negativo. Su nombre se refiere a *Edsger Dijkstra*, quien lo describió por primera vez en 1959.

La tarea del alumno consiste en implementar de forma eficiente este algoritmo y un método que, a partir de la información resultante de este algoritmo, obtenga la secuencia de vértices que conforman el camino más corto hasta un vértice destino dado. Como aplicación de grafos se planteará la respresentación de una red de municipios conectados por carretera y como aplicación de *Dijkstra* la obtención del camino más corto entre dos municipios dados.

Las actividades a realizar son las siguientes: creación del paquete grafos y codificación en Java de los métodos para el cálculo del camino más corto desde un vértice al resto (algoritmo de Dijkstra) y para la obtención de la secuencia de vértices (camino) desde el vértice origen a uno dado.

Actividad #1: Creación del paquete *librerias.estructurasDeDatos.grafos*

El alumno deberá crear el nuevo paquete *librerias.estructurasDeDatos.grafos*, que debe contener la jerarquía de clases definida para la implementación de un grafo dirigido y con pesos. Este paquete deberá incluir las siguientes clases (que se encuentran disponibles en *PoliformaT*):

- `Grafo`: clase abstracta que define la funcionalidad básica de un grafo, abstrayéndose de los detalles de implementación. Algunos métodos, como los recorridos y las búsquedas de caminos mínimos, se implementan en esta clase pues no dependen de las decisiones de implementación del grafo.
- `Adyacente`: esta clase contiene la información necesaria para representar los adyacentes a un vértice del grafo en su implementación mediante listas de adyacencia. Así, un *Adyacente* tiene un destino, que es el código del vértice adyacente, y un peso, que es el peso o coste de la arista que une ambos vértices.
- `GrafoDirigido`: implementación de un grafo dirigido con pesos mediante listas de adyacencia. Hereda la funcionalidad básica de la clase `Grafo`.

Actividad #3: implementación del algoritmo de *Dijkstra*

El alumno deberá completar en clase `Grafo` el método `dijkstra`; este método recibe como parámetro *origen*, que es el código del vértice de partida y debe rellenar los dos arrays (atributos de la clase) siguientes:

- `protected double[] distanciaMin;` // Distancia mínima del origen al resto de vértices
- `protected int[] caminoMin;` // Vértice anterior en el camino más corto

Un esquema de una implementación eficiente del algoritmo es el siguiente:

```
inicializar los arrays distanciaMin, caminoMin y visitados
inicializar la cola de prioridad cp

while (no vacía cp) {
    Seleccionar el vértice v, mínimo de cp;
    if (visitados[v] == 0) {
        visitados[v] = 1;
        Para todo w en adyacentesDe(v) {
            if (distanciaMin[w] > distanciaMin[v] + coste de v a w) {
                  distanciaMin[w] = distanciaMin[v] + coste de v a w;
                  caminoMin[w] = v;
                  insertar en cp el par (w, distanciaMin[w]);
            }
        }
    }
}
```

La interfaz `ColaPrioridad` se encuentra en el paquete *librerias.estructurasDeDatos.modelos* y una implementación eficiente de la misma es la clase `PriorityQColaPrioridad` que se encuentra ubicada

desde la práctica 1 en *librerias.estructurasDeDatos.jerarquicos*. Nótese que los elementos de la Cola de Prioridad que usa `dijkstra` son pares vértice alcanzado y coste hasta ese vértice. Será necesario definir una clase también en el paquete *librerias.estructurasDeDatos.grafos* para representar estos datos.

Actividad #4: implementación del método de decodificación del camino más corto

Completar en la clase `Grafo` el método `caminoMinimo` que recibe dos parámetros: *origen*, que es el código del vértice de partida (municipio origen), y *destino*, que es el código del vértice destino. Como resultado, el método debe devolver una *Lista con Punto de Interés* con los códigos de los vértices que conforman el camino mínimo entre *origen* y *destino*. Para ello se sugiere seguir los siguientes pasos:

1. invocar al algoritmo de *Dijkstra*.
2. recuperar el camino mínimo a partir de los arrays *distanciaMin* y *caminoMin*, guardando el camino resultante en una *Lista con Punto de Interés*.

En caso de que *origen* o *destino* no estén en el grafo, o que no exista camino entre ambos municipios, el método deberá devolver una lista vacía.

Actividad #5: Comprobación de la corrección de la clase `Grafo`

Copiar los ficheros `TestGrafos1.java` y `TestGrafos2.java`, disponibles en poliformaT, en el paquete (carpeta) *librerias.estructurasDeDatos.grafos*, ejecútalos y comprueba que el resultado que se obtiene es el que se indica.

3. La aplicación `municipios`

Para poder representar la red de municipios utilizando la jerarquía grafos desarrollada, resulta necesario etiquetar cada vértice del grafo con un municipio de la red; es decir, es necesario identificar cada vértice con un municipio. Para establecer esta relación de forma eficiente se utilizarán dos diccionarios (`Map`), implementados mediante sendas `TablasHash`:

- `ver2Munis`, en el que las claves serán los índices de los vértices y los valores serán los municipios. Así, se puede recuperar con coste constante el municipio asociado a un vértice `i`: `ver2Munis.recuperar(i)`.
- `munis2Ver`, en el que las claves serán los municipios y los valores los índices de los vértices. Así, también se puede recuperar con coste constante el vértice asociado a un municipio `m`: `munis2Ver.recuperar(m)`.

Las actividades a realizar son las siguientes: crear el paquete `municipios`, completar el código de las clases `Municipio` y `GrafoMunicipios` y usar la aplicación que se proporciona `GuiGrafo`.

Actividad #6: el paquete *aplicaciones.municipios*

Se deberá crear un nuevo paquete, denominado *aplicaciones.municipios*, que contendrá todas las clases relacionadas con la aplicación. Dentro de este paquete se deberá añadir las siguientes clases disponibles en PoliformaT:

- La clase `Municipio` que permite guardar la información relativa a un municipio (nombre, población, extensión y posición dentro del mapa).
- La clase `GrafoMunicipios` dispone internamente de un grafo de municipios y se encarga de gestionar las operaciones habituales del mismo, así como su inicialización a partir de los ficheros de datos.

Actividad #7: la clase `Municipio`

Se deberá completar la clase `Municipio` para que pueda instanciar la clase de las claves de un `Map` implementada mediante una `TablaHash`.

Actividad #8: la clase `GrafoMunicipios`

Se debe completar la clase `GrafoMunicipios` siguiendo los comentarios de la misma. En concreto se debe incluir los `Maps` de vértices a municipios (`ver2Munis`) y viceversa (`munis2Ver`, implementados mediante `TablaHashs`, para permitir el etiquetado de los vértices del grafo. Uno de los métodos a completar será *caminoMinimo* para que traduzca la lista de códigos obtenida a una lista de etiquetas. Puesto que cada vértice del grafo corresponde a un municipio, es conveniente poder obtener el camino mínimo como una lista de los nombres de los municipios que lo componen en lugar de como una lista de códigos.

Actividad #9: aplicación de prueba

Añade la aplicación de prueba, `GuiGrafo`, al paquete *aplicaciones.municipios*. Para poder ejecutarla es necesario copiar los ficheros *municipios.txt*, *distancias.txt* y *spain.jpg*, disponibles en *PoliformaT*, a la carpeta `$HOME/eda/aplicaciones/municipios`.

Al iniciar la aplicación gráfica se puede observar el grafo de municipios con todas sus aristas y alguno de sus vértices más importantes. El grafo consta de un total de 4016 vértices y 47962 aristas. A pesar de su considerable tamaño, se puede comprobar la eficiencia del algoritmo de *Dijkstra*, que es capaz de calcular el camino mínimo entre un municipio y todos los demás de forma casi instantánea.

El resultado que se debe obtener si se consulta a través de la aplicación `GuiGrafo` el camino más corto para llegar de Albaida a Valencia, es el que se muestra en la siguiente figura: