Estructuras de Datos y Algoritmos I
Grado en Ingeniería Informática, Curso 2º

ACTIVIDAD 5

Implementar operaciones sobre Grafos

Objetivos

- Estudiar los grafos y su forma de representación (mapa de mapas).
- Aprender a utilizar la estructura de datos grafo y los algoritmos básicos para su uso en ejercicios concretos.

Requerimientos

Para superar esta actividad se debe realizar lo siguiente:

- Dominar las estructuras de datos para implementar un grafo (dirigido) y su uso para la resolución de actividades sencillas.
- Conocer cómo implementar algoritmos básicos sobre grafos para realización de ejercicios que refuercen los conocimientos adquiridos sobre grafos (en este caso, para un grafo dirigido).

Enunciado

En esta actividad vamos a complementar lo estudiado en las clases de teoría, relativo a grafos dirigidos ponderados, que aquí denominaremos `Network` (red). Esta actividad va a consistir en implementar las operaciones de la interface `Graph<T>` y otras operaciones adicionales que se explicarán a continuación.

```
public interface Graph<T> {
   public int numberOfVertices();
   public int numberOfEdges();
   public boolean isEmpty();
   public double getWeight(T v1, T v2);
   public double setWeight(T v1, T v2, double w);
   public Set<T> getNeighbors(T v);
   public boolean addEdge(T v1, T v2, double w);
   public boolean addVertex(T v);
   public boolean removeEdge(T v1, T v2);
   public boolean removeVertex(T v);
   public void clear();
   public Set<T> vertexSet();
   public boolean containsVertex(T v);
   public boolean containsEdge(T v1, T v2);
}
```

La clase `Network.java` que va a implementar la interface `Graph.java`, junto con una serie de operaciones adicionales. Destacar que `Network.java` es un grafo genérico y que en lugar de utilizar **`T`** se ha utilizado **`Vertex`** para indicar que los vértices del grafo pueden ser cualquier tipo de objeto.

Además, uno de los aspectos más destacables de esta clase son los datos miembro que serán los siguientes y 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*.

```
public class Network<Vertex> implements Graph<Vertex>, Iterable<Vertex> {
   protected boolean directed; // directed = false (unDirected), directed = true (DiGraph)
   protected TreeMap<Vertex, TreeMap<Vertex, Double>> adjacencyMap;
   protected TreeMap<Vertex, Boolean> visited;
   protected ArrayList<Vertex> result;
   protected ArrayList<ArrayList<Vertex>> resultSimplePaths;
   Double shortestPathWeight;
   protected ArrayList<Vertex> resultShortestPath;
   int largestLengthPath;
   protected ArrayList<Vertex> resultLargestLengthPath;
   int numberOfSimplePaths, sumOfLengthOfSimplePaths;
   public Network() { … }
   public Network(boolean uDOrD) { … }
   public Network (Network<Vertex> network) { … }
   public void setDirected(boolean uDOrD) { … }
   public boolean getDirected() { … }
   public boolean isEmpty() { … }
   public int numberOfVertices() { … }
   public int numberOfEdges() { … }
   public void clear() { … }
   public double getWeight (Vertex v1, Vertex v2) { … }
   public double setWeight (Vertex v1, Vertex v2, double w) { … }
   public boolean containsVertex(Vertex vertex) { … }
   public boolean containsEdge(Vertex v1, Vertex v2) { … }
   public boolean addVertex(Vertex vertex) { … }
   public boolean addEdge(Vertex v1, Vertex v2, double w) { … }
```

```
   public boolean removeVertex(Vertex vertex) { … }
   public boolean removeEdge (Vertex v1, Vertex v2) { … }
   public Set<Vertex> vertexSet(){ … }
   public Set<Vertex> getNeighbors(Vertex v) { … }
   public String toString(){ … }
   public int inDegree(Vertex v) { … }
   public int outDegree(Vertex v) { … }
   public static Network<String> readNetwork(String filename) throws FileNotFoundException  { … }
   public void showNetwork(){ … }
   public boolean isReachable(Vertex source, Vertex destination) { … }
   private boolean isReachableAux(Vertex current, Vertex destination) { … }
   public boolean isSource(Vertex v) { … }
   public boolean isSink(Vertex v) { … }
   public ArrayList<ArrayList<Vertex>> simplePaths(Vertex source, Vertex destination) { … }
   private void simplePathsAux(Vertex current, Vertex destination) { … }
   public ArrayList<Vertex> toArrayDFS(Vertex start) { … }
   private void toArrayDFSAux(Vertex current) { … }
   public ArrayList<Vertex> toArrayDFSIterative(Vertex start) { … }
   public ArrayList<Vertex> toArrayBFS(Vertex start) { … }
   public String shortestPathWithSimplePaths(Vertex source, Vertex destination) { … }
   private void shortestPathWithSimplePathsAux(Vertex current, Vertex destination) { … }
   public String largestLenghtPathWithSimplePaths(Vertex source, Vertex destination) { … }
   private void largestLenghtPathWithSimplePathsAux(Vertex current, Vertex destination) { … }
   public boolean isPathLength(Integer length, Vertex source, Vertex destination) { … }
   // Otras funciones adicionales
   public ArrayList<Object> DijkstraPQ(Vertex source, Vertex destination) { … }
   public String FloydEC() { … }
   public boolean isConnected() { … }
   public boolean isStronglyConnected() { … }
   public boolean isTree() { … }
   public Iterator<Vertex> iterator(){ … }
   public BreadthFirstIterator breadthFirstIterator (Vertex v) { … }
   public DepthFirstIterator depthFirstIterator (Vertex v) { … }
   protected class BreadthFirstIterator implements Iterator<Vertex> {
        protected LinkedQueue<Vertex> queue;
        protected TreeMap<Vertex, Boolean> reached;
        protected Vertex current;
        public BreadthFirstIterator(Vertex start) { … }
        public boolean hasNext() { … }
        public Vertex next() { … }
        public void remove() { … } }
   protected class DepthFirstIterator implements Iterator<Vertex> {
        ALStack<Vertex> stack;
        TreeMap<Vertex, Boolean> reached;
        Vertex current;
        public DepthFirstIterator(Vertex start) { … }
        public boolean hasNext() { … }
        public Vertex next() { … }
        public void remove(){ … } }
}
```

La función que permite mostrar los datos contenidos en la red (`Network`) es `showNetwork`. Esta función lee y muestra en pantalla la estructura **`TreeMap<Vertex, TreeMap<Vertex, Double>>`**.

```
   public void showNetwork(){ … }
```

Actividades a implementar:

A continuación vamos a explicar en qué van a consistir las funciones adicionales a implementar. Recordar que estas nuevas funciones deben estar incluidas en el archivo `Network.java` (implementación de la interface `Graph.java`).

```
public int inDegree(Vertex v) { … }
```
Consultar el grado de entrada de un vértice dado.

```
public int outDegree(Vertex v) { … }
```
Consultar el grado de salida de un vértice dado.

```
public static Network<String> readNetwork(String filename) throws FileNotFoundException  { … }
```
Leer de un archivo de texto (*graphFile.txt*) la estructura del grafo y crear una estructura `Network` (grafo dirigido ponderado según la declaración descrita anteriormente). El archivo debe indicar en la primera línea si es un grafo dirigido (1) o no dirigido (0). En la segunda línea aparece el número de vértices y a continuación aparecen tantos nombres de vértices en líneas separadas como ha indicado dicho número (número de vértices). En la tercera línea tendremos el número de aristas o arcos que tiene dicho grafo y a partir de dicha línea tendremos todas las aristas siguiendo el formato *<vértice_origen vértice_destino peso_arista>* (tantas líneas independientes como aristas tiene el grafo).

```
public boolean isReachable(Vertex source, Vertex destination) { … }
private boolean isReachableAux(Vertex current, Vertex destination) { … }
```
Comprobar si un vértice es alcanzable desde otro vértice dado.

```
public boolean isSource(Vertex v) { … }
```
Comprobar si un vértice es fuente, es decir, si es un vértice del que sólo salen aristas. No se puede hacer uso de la función `inDegree`, ya que un vértice fuente es aquél con grado de entrada igual a cero.

```
public boolean isSink(Vertex v) { … }
```
Comprobar si un vértice es un sumidero (es decir, un vértice al que sólo llegan aristas) al que llegan aristas de todos los demás vértices del grafo. No se puede hacer uso de la función `outDegree`, ya que un vértice sumidero es aquél con grado de salida igual a cero.

```
public ArrayList<ArrayList<Vertex>> simplePaths(Vertex source, Vertex destination) { … }
private void simplePathsAux(Vertex current, Vertex destination) { … }
```
Implementar una función que dado un grafo (`Network`) con la estructura descrita anteriormente, y dos vértices de dicho grafo, muestre todos los posibles *caminos simples* de un vértice a otro. Recuerde que un camino simple es aquel en el que todos los vértices son distintos (excepto el primero y el último que pueden ser iguales).

```
public ArrayList<Vertex> toArrayDFS(Vertex start) { … }
private void toArrayDFSAux(Vertex current) { … }
```
Implementar una función para obtener en un `ArrayList` de vértices, los vértices que se visitan siguiendo un recorrido en profundidad (Depth-First traversal) recursivo.

```
public ArrayList<Vertex> toArrayDFSIterative(Vertex start) { … }
```
Implementar una función para obtener en un `ArrayList` de vértices, los vértices que se visitan siguiendo un recorrido en profundidad Depth-First traversal iterativo (sin utilizar la recursividad).

```
public ArrayList<Vertex> toArrayBFS(Vertex start) { … }
```
Implementar una función para obtener en un array de vértices, los vértices que se visitan siguiendo un recorrido en anchura o amplitud (Breadth-First traversal).

```
public String shortestPathWithSimplePaths(Vertex source, Vertex destination) { … }
private void shortestPathWithSimplePathsAux(Vertex current, Vertex destination) { … }
```
Implementar una función que devuelva el camino más corto (si hay más de uno, que devuelva uno de ellos) según el peso de las aristas entre un vértice `source` y otro `destination` haciendo uso del algoritmo de *caminos simples*. Además debe ser posible calcular fácilmente su coste total.

```
public String largestLenghtPathWithSimplePaths(Vertex source, Vertex destination) { … }
private void largestLenghtPathWithSimplePathsAux(Vertex current, Vertex destination) { … }
```
Implementar una función que devuelva el camino con mayor longitud (si hay más de uno, que devuelva uno de ellos) entre un vértice `source` a un vértice `destination`, y su correspondiente longitud de camino (recordad que la longitud de un camino es su número de vértices menos uno o equivalentemente el número de aristas). Además, y al final, debe devolver la longitud media de los caminos que van desde dicho vértice `source` hasta el vértice `destination`.

```
public boolean isPathLength(Integer length, Vertex source, Vertex destination) { … }
```
Implementar una función que devuelva `true` en el caso de que exista un camino entre el vértice `source` y el `destination` y tiene una longitud de `length`, `false` en cualquier otro caso. Recordad que la longitud de un camino es su número de vértices menos uno o equivalentemente el número de aristas.

```
public boolean isTree() { … }
```
Implementar una función para determinar si un grafo no orientado es un *árbol libre* (tree). Un grafo no orientado es un árbol libre si es conexo y acíclico (sin ciclos), o equivalentemente, si es conexo y tiene exactamente *n – 1* aristas para *n* nodos.

```
public ArrayList<Object> DijkstraPQ(Vertex source, Vertex destination)
```
Hemos estudiado e implementado el algoritmo de *Dijkstra*, partiendo desde un vértice *fuente*, `public ArrayList<Object> Dijkstra(Vertex source, Vertex destination)`, sobre un grafo orientado/no orientado, conexo y valorado positivamente, utilizando mapas (`TreeMap`) y conjuntos (`TreeSet`). En este ejercicio se pide la implementación del algoritmo de *Dijkstra* utilizando una *cola de prioridad* (`PriorityQueue<VertexWeightPair> pQ`) en lugar de un `TreeSet` (`V_minus_S`) para determinar los vértices que se van procesando. Como una aproximación para acometer esta función es recomendable estudiar y comprobar el correcto funcionamiento de la función `public ArrayList<Object> PrimPQ(Vertex source)`. Nótese que también hay que definir e implementar las clases: `protected class VertexWeightPair implements Comparable<VertexWeightPair> { Vertex vertex; double weight; // … }` y `protected class EdgeWeight implements Comparable<EdgeWeight> { Vertex from; Vertex to; double weight; // … }`.

```
public String FloydEC()
```

Hemos estudiado tanto teoría como en prácticas el algoritmo de *Floyd* para obtener el camino mínimo entre todos los posibles pares de vértices del grafo (`public void Floyd()`). En este contexto, sea *G* un grafo orientado y valorado, y v_i un vértice de *G*. Se define la ***excentricidad*** del vértice v_i respecto del grafo *G* como el máximo de los costes de los caminos mínimos (matriz D del algoritmo de *Floyd*) que van desde cualquier otro vértice v_j del grafo al vértice v_i. También se define el ***centro*** del grafo como el vértice que tiene mínima excentricidad. Se pide implementar en Java (**`public String FloydEC()`**) una función que calcule la *excentricidad* de cada uno de los vértices v_i de *G* y el *centro* del grafo junto con el coste del camino mínimo de cada uno de los vértices de *G* accesibles desde dicho *centro*. Debe ayudarse de la función `public void Floyd()` estudiada en clase.


org.eda1.actividad05.Network<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.actividad05.Network<Vertex>.DepthFirstIterator
stack: ALStack<Vertex>
reached: TreeMap<Vertex,Boolean>
current: Vertex
DepthFirstIterator(start: Vertex)
hasNext(): boolean
next(): Vertex
remove(): void
org.eda1.actividad05.Network<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.actividad05.Network<Vertex>.BreadthFirstIterator
queue: LinkedQueue<Vertex>
reached: TreeMap<Vertex,Boolean>
current: Vertex
BreadthFirstIterator(start: Vertex)
hasNext(): boolean
next(): Vertex
remove(): void
org.eda1.actividad05.Network<Vertex>
directed: boolean
adjacencyMap: TreeMap<Vertex,TreeMap<Vertex,Double>>
visited: TreeMap<Vertex,Boolean>
visitedWithSet: TreeSet<Vertex>
result: ArrayList<Vertex>
resultSimplePaths: ArrayList<ArrayList<Vertex>>
shortestPathWeight: Double
resultShortestPath: ArrayList<Vertex>
largestLengthPath: int
resultLargestLengthPath: ArrayList<Vertex>
numberOfSimplePaths: int
sumOfLengthOfSimplePaths: int
inDegree(v: Vertex): int
outDegree(v: Vertex): int
readNetwork(filename: String): Network<String>
isReachable(source: Vertex, destination: Vertex): boolean
isReachableAux(current: Vertex, destination: Vertex): boolean
isSource(v: Vertex): boolean
isSink(v: Vertex): boolean
simplePaths(source: Vertex, destination: Vertex): ArrayList<ArrayList<Vertex>>
simplePathsAux(current: Vertex, destination: Vertex): void
simplePathsVisited(source: Vertex, destination: Vertex): ArrayList<ArrayList<Vertex>>
simplePathsVisitedAux(current: Vertex, destination: Vertex): void
simplePathsBFS(source: Vertex, destination: Vertex): ArrayList<ArrayList<Vertex>>
toArrayDFS(start: Vertex): ArrayList<Vertex>
toArrayDFSAux(current: Vertex): void
toArrayDFSWithSet(start: Vertex): ArrayList<Vertex>
toArrayDFSAuxWithSet(current: Vertex): void
toArrayDFS(): ArrayList<Vertex>
toArrayDFSIterative(start: Vertex): ArrayList<Vertex>
toArrayBFS(start: Vertex): ArrayList<Vertex>
toArrayBFS(): ArrayList<Vertex>
shortestPathWithSimplePaths(source: Vertex, destination: Vertex): String
shortestPathWithSimplePathsAux(current: Vertex, destination: Vertex): void
largestLenghtPathWithSimplePaths(source: Vertex, destination: Vertex): String
largestLenghtPathWithSimplePathsAux(current: Vertex, destination: Vertex): void
isPathLength(source: Vertex, destination: Vertex, length: Integer): boolean
isConnected(): boolean
isStronglyConnected(): boolean
isTree(): boolean
Dijkstra(source: Vertex, destination: Vertex): ArrayList<Object>
DijkstraPQ(source: Vertex, destination: Vertex): ArrayList<Object>
isFloydGraph(): boolean
adaptToFloydGraph(): void
FloydEC(): String
writeObject(out: java.io.ObjectOutputStream): void
readObject(in: java.io.ObjectInputStream): void
iterator(): Iterator<Vertex>
breadthFirstIterator(v: Vertex): BreadthFirstIterator
depthFirstIterator(v: Vertex): DepthFirstIterator
removeVertex(vertex: Vertex): boolean
addEdge(v1: Vertex, v2: Vertex, w: double): boolean
addVertex(vertex: Vertex): boolean
containsEdge(v1: Vertex, v2: Vertex): boolean