Advanced Data Structures

Labwork 6: Implementation of Floyd-Warhsall algorithm

Implement a java class `FloydWarshall` with the following API:

`public class FloydWarshall`	
`FloydWarshall(IWeightedGraph G)`	Computes a predecessor matrix and a matrix of minimum weights from each node to every other node in the weighted graph `G`.
`boolean connected(int i,int j)`	Is there a path from node `i` to node `j`?
`double distance(int i,int j)`	The minimum weight of a path from node `i` to `j` in `G`.
`Integer predecessor(int i,int j)`	The prececessor of node `i` on a path with minimum weight from `i` to `j`. Returns `null` if such a predecessor does not exist.
`Iterable<Integer> path(int i,int j)`	The nodes of a path with minimum weight from node `i` to node `j`. Returns `null` if such a path does not exist.

1 Setting up the programming environment

1. Download the library of java classes `algs4.jar` from

   `https://algs4.cs.princeton.edu/code/`

2. Download the archive of java classes `ADS.zip` from

   `https://staff.fmi.uvt.ro/~mircea.marin/lectures/EduGraph/`

   and unzip it. You will get a directory `src` with source code of java classes, and sample graph files.

3. Use `eclipse` to create a java project, say `TGC`, and

   - add algs4.jar as external JAR to the build path of this java project.
   - overwrite the `src` directory of this project with the `src` directory downloaded in step 2.

   The java classes to work with graphs are in the package

   `ro.uvt.cs.graphs`

1.1 A java API for weighted graphs

The interface `IWeightedGraph` provides the following API to work with weighted graphs:

`public interface IWeightedGraph`	
`int V()`	Number of nodes
`int E()`	Number of edges
`boolean addEdge(int` i`,int` j`,double` w`)`	Add edge $i-j$ with weight w
`Iterable<WeightedEdge> adj(int` i`)`	Iterator of the outgoing weighted edges form node i
`int degree(int` i`)`	Outdegree of node i
`boolean directed()`	Is the graph directed?
`String toString()`	String representation of the weighted graph.

The outgoing weighted edges from a node are represented by instances of class `WeightedEdge`:

`public`	`class WeightedEdge`	
	`WeightedEdge(int i,double w)`	Creates an outgoing weighted edge to node i with weight w
`int`	`node()`	The destination node of the edge
`double`	`w()`	The weight of the edge

The classes which implement this interface are: (1) `WeightedGraph` for unoriented weighted graphs, and (2) `WeightedDigraph`: subclass of class `WeightedGraph` for weighted digraphs. Both classes implement a representation with adjacency lists where the nodes of a graph with n nodes are 0, 1, 2, ..., $n-1$. The additional capabilities of these classes are:

`public`	`class WeightedGraph`	
	`WeightedGraph(int` n`)`	Creates a weighted graph with n nodes and no edges
	`WeightedGraph(In` in`)`	Reads a weighted graph from input stream in

`public class WeightedDigraph`	`extends WeightedGraph`
`WeightedDigraph(int` n`)`	Creates a weighted graph with n nodes and no edges
`WeightedDigraph(In` in`)`	Reads a weighted digraph from input stream in
`int indegree(int` i`)`	Indegree of node i
`int outdegree(int` i`)`	Outdegree of node i
`WeightedDigraph reverse()`	Reverse of this weighted digraph

The constructors `WeightedGraph(In` in`)` and `WeightedDigraph(In` in`)` read a weighted graph from an input stream consisting of $3\,m+2$ numbers: the number n of nodes, the number m of edges, followed by m groups made of three numbers: two `int`s which indicate the endpoints of an edge, followed by a `double` which is the weight of the edge.

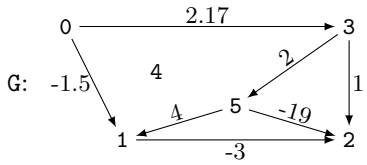
For instance, if `wgraph.txt` is a text file with content

```
6 7
0 1 -1.5
0 3 2.17
3 2 1
3 5 2
1 2 -3
5 1 4
5 2 -19
```

then the instruction

```
WeightedDigraph G = new WeightedDigraph(new In("wgraph.txt"));
```

creates an instance `G` which represents the weighted digraph



Object `G` is the internal representation (in computer memory) of this weighted digraph, and the content of file `wgraph.txt` is its external representation.

2 Algorithm description

Suppose G is a graph with n nodes numbered from 0 to $n-1$. The Floyd-Warshall algorithm works in two stages:

Initialization: We set up two arrays of dimension $n \times n$: the array of `doubles` $\mathtt{d}_0$ and the array of `Integers` $\mathtt{P}_0$, such that

- $\mathtt{P}_0[i][j] = i$ if G has an edge from i to j; and $\mathtt{P}_0[i][j]$ undefined (that is, `null`), otherwise.
- $\mathtt{d}_0[i][i] = 0$; $\mathtt{d}_0[i][j]$ =the weight of edge from i to j, if there is one; $\mathtt{d}_0[i][j] = +\infty$ otherwise.

For the weighted digraph G depicted above we have

$$\mathtt{P}_0 = \begin{pmatrix} \bullet & 0 & \bullet & 0 & \bullet & \bullet \\ \bullet & \bullet & 1 & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & 3 & \bullet & \bullet & 3 \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & 5 & 5 & \bullet & \bullet & \bullet \end{pmatrix}, \quad \mathtt{d}_0 = \begin{pmatrix} 0 & -1.5 & +\infty & 2.17 & +\infty & +\infty \\ +\infty & 0 & -3 & +\infty & +\infty & +\infty \\ +\infty & +\infty & 0 & +\infty & +\infty & +\infty \\ +\infty & +\infty & 1 & 0 & +\infty & 2 \\ +\infty & +\infty & +\infty & +\infty & 0 & +\infty \\ +\infty & 4 & -19 & +\infty & +\infty & 0 \end{pmatrix}$$

where $\bullet$ stands for `null`.

Update: For $k = 1$ to n, compute the arrays $\mathtt{P}_k$ and $\mathtt{d}_k$ of dimension $n \times n$, as follows:

$$\mathtt{d}_k[i][j] = \min(\mathtt{d}_{k-1}[i][j], \mathtt{d}_{k-1}[i][k-1] + \mathtt{d}_{k-1}[k-1][j])$$

$$\mathtt{P}_k[i][j] = \begin{cases} \mathtt{P}_{k-1}[i][j] & \text{if } \mathtt{d}_{k-1}[i][j] = \mathtt{d}_k[i][j], \\ \mathtt{P}_{k-1}[k][j] & \text{otherwise.} \end{cases}$$

For the weighted digraph G depicted above, the algorithm computes

$$\mathtt{d}_1 = \ldots \quad \mathtt{d}_2 = \ldots \quad \mathtt{d}_3 = \ldots \quad \mathtt{d}_4 = \ldots \quad \mathtt{d}_5 = \begin{pmatrix} 0 & -1.5 & -14.83 & 2.17 & +\infty & 4.17 \\ +\infty & 0 & -3 & +\infty & +\infty & +\infty \\ +\infty & +\infty & 0 & +\infty & +\infty & +\infty \\ +\infty & 6 & -17 & 0 & +\infty & 2 \\ +\infty & +\infty & +\infty & +\infty & 0 & +\infty \\ +\infty & 4 & -19 & +\infty & +\infty & 0 \end{pmatrix}$$

$$\mathtt{P}_1 = \ldots \quad \mathtt{P}_2 = \ldots \quad \mathtt{P}_3 = \ldots \quad \mathtt{P}_4 = \ldots \quad \mathtt{P}_5 = \begin{pmatrix} \bullet & 0 & 5 & 0 & \bullet & 3 \\ \bullet & \bullet & 1 & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & 5 & 5 & \bullet & \bullet & 3 \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & 5 & 5 & \bullet & \bullet & \bullet \end{pmatrix}$$

The algorithm guarantees that, if G is a weighted graph where all cycles have nonnegative weight, then arrays $\texttt{d}_n$ and $\texttt{P}_n$ have the following properties for all $0 \leq i, j < n$:

1. $\texttt{d}_n[i][j] = +\infty$ if there is no path from i to j; otherwise, $\texttt{d}_n[i][j]$ is the minimum weight of a path from i to j.

2. $\texttt{P}_n[i][j]$ is `null` if $i = j$ or there is no path from i to j; otherwise, $\texttt{P}_n[i][j]$ is the predecessor of j in a path with minimum weight from i to j.