Project 6 (Java): All paths shortest paths, by using Dijkstra's algorithm for the Single-Source-Shortest Paths problem N times.

The single-source-shortest pages problem Statement: Given a directed graph, G = <N, E>, and the source node, S, in G, the task is find the shortest paths from S to all other nodes in G, using the Dijkstra's algorithm.

*** Please note that this project, the source node will be 1, 2, 3, …, N. // i.e., Your program will produce *all pairs* shortest paths.

*** You will be given 2 data files: data1 and data2. data1 is the example given in the lecture note. except using 1, 2, 3, 4, and 5 for A, B, C, D, and E; data2 is a larger graph. What to do as follows:
1) Implement your program based on the specs given below.
2) Run and debug your program with data1 until your program produces the same result as the lecture note, when source node is A.
3) When the result is correct, then run your program with data2.

Include in your hard copy:
- cover page
- source code
- SSSfile for data1
- deBugFile for data1
- SSSfile for data2
- deBugFile for data2

****
Language: Java
****
Project points:10 pts
Due Date: Soft copy (*.zip) and hard copies (*.pdf):
+1 (11/10 pts): early submission, 11/30/2022, Sunday before midnight
-0 (10/10 pts): on time, 11/3/2022 Thursday before midnight
-1 (9/10 pts): 1 day late, 11/4/2022 Friday before midnight
-2 (8/10 pts): 2 days late, 11/5/2022 Saturday before midnight
(-10/10 pts): non submission, 11/5/2022 Saturday after midnight

*** Name your soft copy and hard copy files using the naming convention as given in the project submission requirement.
*** All on-line submission MUST include Soft copy (*.zip) and hard copy (*.pdf) in **the same email attachments** with correct email subject as stated in the email requirement; otherwise, your submission will be rejected.
.
****
I. inFile (args [0]): a directed graph, represented by a list of edges with costs, {<n_i, n_j, c>}
// You may assume that nodes' Id is from 1 to N (0 is not used)

The format of the input file is as follows:

The first text line is the number of nodes, N, follows by a list of triplets, <n_i, n_j, cost>
For example:

```
5                    // there are 5 nodes in the graph
1  5 10  // an  edge  from node 1 to node 5, the cost is 10
2  3  5
1  2  20
3  5  2
:
:
```

****

II. Outputs:

a) SSSfile (args [1]) : for the result of all pairs shortest paths. The format is given below:

// For example, if there are 7 nodes in the graph G. Then your output will be as follows:

==========

There are 7 nodes in the input graph. Below are all pairs of shortest paths:

==========

Source node = 1

The path from 1 to 1 : 1 ← 1 : cost = 0
The path from 1 to 2 : 2 ← … ← 1: cost = whatever
The path from 1 to 3 : 3 ← … ← 1: cost = whatever
:
:
The path from 1 to 7 : 7 ← … ← 1: cost = whatever

==========

The source node = 2

The path from 2 to 1 : 1 ← … ← 2 : cost = whatever
The path from 2 to 2 : 2 ← 2: cost = 0
The path from 2 to 3 : 3 ← … ← 2: cost = whatever
:
:
The path from 2 to 7 : 7 ← … ← 2: cost = whatever

==========

:
:

==========

The source node = 7

The path from 7 to 1 : 1 ← … ← 7 : cost = whatever
The path from 7 to 2 : 2 ← … ← 7 : cost = whatever
The path from 7 to 3 : 3 ← … ← 7 : cost = whatever
:
:
The path from 7 to 7 : 7 ← 7: cost = 0

b) deBugFile (args [2]): For all debugging outputs.

****

III. Data structure:

1) A DijktraSSS class

- (int) numNodes //number of nodes in G
- (int) sourceNode
- (int) minNode
- (int) currentNode
- (int) newCost
- (int) costMatrix[][]
  // a 2-D cost matrix (integer array), size of N+1 X N+1, should be dynamically allocated.
  // Initially, costMatrix[i][i] set to zero and all others set to infinity, 9999
  // Note: 0 is not used for node Id in this program.

- (int) father [] // a 1-D integer array, size of N+1, should be dynamically allocated. Initially set to source.
- (int) toDo [] // 1-D integer array, size of N+1, should be dynamically allocated;
//initially set to 1 (unsolved).
- (int) best [] // a 1-D integer array, size of N+1, should be dynamically allocated.

Methods:
- constructor (…) // Use it freely.
- loadCostMatrix (. . .)// read from input file and fill the costMatrix, on your own.
- setBest (sourceNode) // copy the row of source node from costMatrix,
- setFather (sourceNode) // set all to sourceNode
- setToDo (sourceNode) // set sourceNode to 0 and all other to 1
- int findMinNode (. . .) // find an *unsolved* node with minimum cost from best Ary
// Algorithm is given below
- (int) computeCost (minNode, currentNode)
// returns best [minNode] + costMatrix [minNode][currentNode]
- debugPrint (…) // This method for you to debug your program.
// Prints sourceNode to deBugFile (with proper heading, i.e., the sourceNode is:)
// Prints father array to deBugFile (with proper heading)
// Prints best array to deBugFile (with proper heading)
// Prints toDo array to deBugFile (with proper heading)
- (bool) doneToDo (…) // returns true of toDo array are all 0 (i.e., all nodes are solved), else return false.
- printShortestPath (currentNode, sourceNode, SSSfile) // on your own.
// The method traces from currentNode back to sourceNode (via fatherAry),
// print to SSSfile, the shortest path from currentNode to sourceNode with the total cost, using the format
//given in the above.

****
V. main (…)
****

step 0: open inFile, SSSfile, deBugFile
numNodes ← get from inFile
Allocate and initialize all members in the DijktraSSS class accordingly

step 1: loadCostMatrix (inFile)
sourceNode ← 1

step 2: setBest (sourceNode)
setFather (sourceNode)
setToDo (sourceNode)

step 3: minNode ← findMinNode (…)
toDo [minNode] ← 0
debugPrint (…)

step 4: // expanding the minNode
currentNode ← 1

step 5: if toDo[currentNode] > 0 { // only process those unsolved nodes.
newCost ← computeCost (minNode, currentNode)
if newCost < best [currentNode]
best [currentNode] ← newCost
father [currentNode] ← minNode
debugPrint (…)
}

step 6: currentNode ++

step 7: repeat step 5 to step 6 while currentNode <= numNodes

step 8: repeat step 3 to step 7 until doneToDo() == true

// begin printing the paths from sourceNode
step 9: currentNode ← 1
step 10: printShortestPath (currentNode, sourceNode, SSSfile)
step 11: currentNode ++
step 12: repeat 10 and step 11 while currentNode <= numNodes
step 13: sourceNode ++
step 14: repeat step 2 to step 13 while sourceNode <= numNodes
step 15: close all files

****
V. (int) findMinNode ()
****
Step 0: minCost ← 99999
minNode ← 0

Step 1: index ← 1

Step 2: if toDo [index] > 0 // if it is an unsolved node
if best [index] < minCost
minCost ← best[index]
minNode ←index
step3: index++

step 4: repeat step 2 to step 3 while index <= numNodes

step 5: return minNode