Project 7 (Java): Scheduling. You are to implement the dependency graph, and scheduling problem.

There are four options in scheduling:

You will be given 4 test data sets where each set includes two files: one contains the dependency graph and one contains processing time for each node. Nodes in the graphs represent jobs. Note: in the specs, jobs and nodes means the same thing.

Set1: graph1 and jobTime1
Set2: graph2 and jobTime2
Set3: graph3 and jobTime3 // graph3 contains a cycle, see if your program can detect the cycle.
Set4: graph4 and jobTime4 // A larger graph.

What to do as follows:
1) Implement your program based on the specs given below.
2) Hand draw to illustrate the schedule table using Set1 with 3 processors as taught in class.
3) Run and debug your program on Set1 with 3 processors until your program can produce the same result as your illustration.
4) Hand draw to illustrate the schedule table using unlimited (numNodes +2) processors.
5) Run and debug your program until your program can produce the same result as your illustraction
6) Run and debug your program on Set3 with 3 processors until your program can detect the cycle.
7) Run your program on Set4 with 3 processors.
8) Run your program on Set4 with (numNodes + 2) processors.

*** Include in your hard copies:

- cover page
- illustration of 2) in the above.
- illustration of 4) in the above.
- source code
- outFile1 and outFile2 from the results of 3) in the above.
- outFile1 and outFile2 from the results of 5) in the above.
- outFile1 and outFile2 from the results of 6) in the above.
- outFile1 and outFile2 from the results of 7) in the above.
- outFile1 and outFile2 from the results of 8) in the above.

****

Language: Java
Project points: 12 pts
Due Date: Soft copy (*.zip) and hard copies (*.pdf):

-0 (12/12 pts): on time, 4/24/2022 Sunday before midnight
+1 (13/12 pts): early submission, 4/19/2022, Tuesday before midnight
-1 (11/12 pts): 1 day late, 4/25/2022 Monday before midnight
-2 (10/12 pts): 2 days late, 4/26/2022 Tuesday before midnight
(-12/12 pts): non submission, 4/26/2022 Tuesday 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. Inputs: There are three (3) inputs to the program.

****

1) inFile1 (use args[0]): a text file representing the dependency graph, G=<N, E>.
The first number in inFile1 is the number of nodes in the graph;
then follows by a list of directed edges (dependency) <n_i, n_j>, where n_i, n_j nodeIDs;
nodeID is from 1 to numNodes, 0 is not used.
For example:

```
6       // Graph has 6 nodes
1  2    // 2 is a dependent of 1 & 1 is a parent of 2
1  4    // 4 is a dependent of 1 & 1 is a parent of 4
4  3    // 3 is a dependent of 4 & 4 is a parent of 3
4  2    // 2 is a dependent of 4 & 4 is a parent of 2
6  1    // 1 is a dependent of 1 & 6 is a parent of 1
:
:
```

2) inFile2 (use args[1]): a text file contains the processing times for nodes.
The first number in inFile2 is the number of nodes in the graph;
then follows by a list of pairs, <n_i, t_i>, where n_i is the node's id and t_i is the unit of processing times for node n_i.

For example: all jobs take the same unit of processing time;

```
6       // Graph has 6 nodes
1  1    // job time for node 1 is 1
2  1    // job time for node 2 is 1
3  1    // job time for node 3 is 1
        :
        :
```

another example: jobs take variable of processing time

```
6       // there are 6 nodes in the input graph
1  3    // job time for node 1 is 3
2  4    // job time for node 2 is 4
3  1    // job time for node 3 is 1
        :
        :
```

3) number processors (use args[2]) // > 0, use at least one processor!

****
II. Outputs: There are two (2) output files
****

1) outFile1: (use args[3]) for the intermediate and final results of the schedule table, nicely formatted.

For example:

```
==================================
ProcUsed : 2   currentTime: 7

Time:    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 . .
         ---------------------------------
Proc: 1  | 1 | 1 | 7 | 3 | 3 | 3 | - | 6 . . .
         ---------------------------------
Proc: 2  | 2 | 4 | 4 | 4 | - | 5 | 5 | - . . .
         ---------------------------------
```

2) outFile2 (use args[4]): for all debugging outputs <u>to get partial credits if your program does not work completely!!</u>

****
III. Data structure:
****

- A node class
  - (int) jobID

- (int) jobTime
- (node) next

Method:
- constructor (…)

- A schedule class
  - (int) numNodes // the total number of nodes in the input graph.
  - (int) numProcs // the number of processors can be used.
  - (int) procUsed // number of processors used so far; initialized to 0.
  - (int) currentTime // initialize to 0.
  - (int) totalJobTimes // the sum of all job times of nodes in the graph.
  - (int []) jobTimeAry // an 1D array to store the job time of each node in the graph;
    // to be dynamically allocated, size of numNodes +1; initialized to 0.
  - (int [][]) Matrix // a 2-D integer array, size numNodes+1 by numNodes+1,
    // to represent the dependency graph; to be dynamically allocated, initialized to zero.

    // We use this matrix for the followings:

    // Matrix [0][0] to store number of nodes remain in the graph; initialize to numNodes;
    // decreases by 1, when a node is deleted; So, to check if the graph is empty,
    // just check if Matrix[0][0] == 0.

    // Matrix [i][j] >= 1, means node i is a parent of node j (i.e., node j is a dependent of node i).

    // Matrix [0][j], we use row 0 to store the parent counts of node j.
    //The parent counts of node j is the total count of none zero rows in **j-th column**.
    // So if we want to know the parent counts of node j, we check Matrix[0][j].

    // Matrix [i][0], we use column 0 to store the dependent counts of node i.
    //The dependent counts of node i is the total count of none zero columns in **i-th row**.
    // So if we want to know the dependent counts of node i, we check Matrix [i][0].

    // Matrix [i][i], the diagonal, [i][i] to indicate the status of the node, where i = 1 … numNodes
    // Matrix [i][i] == 0 means node i is not in the graph. (i.e., done and deleted.)
    // Matrix [i][i] == 1 means node i is in the graph and not marked yet.
    // Matrix [i][i] == 2 means node i is marked.
  - (int [][]) Table // a 2-D integer array, size of (numProcs +1) by (totalJobTimes +1) for scheduling;
    // to be dynamically allocated, and initialized to 0.
    // Index of Table row represent processor's id, and index of Table column represent time slot.
    // Table [i][T] > 0 means the processor i is processing a job, (Table[i]T]), at time slot T.
    // Table [i][T] <= 0 means the processor i is idled (available) at time slot T.
    // where i = 1 to numProcs and T = 0 to totalJobTimes.
  - (node) OPEN // OPEN acts as the list head of a linked list with a dummy node.
    // Nodes in OPEN are sorted in **ascending order** by jobTime.

Methods:
- constructor (…) // take care all member allocations, initialization, etc.
- loadMatrix (inFile1) // read an edge <n_i, n_j> from inFile1and load.
- (int) loadJobTimeAry (inFile2) // read each pair <jobID, time> from inFile2 and load to jobTimeAry;
  // set jobTimeAry[jobID] ← time; this method needs to keep track of total job times;
  // it returns totalJobTimes
- setMatrix (…) // Compute parent counts and store in Matrix[0][j],
  // computes dependent counts and store in Matrix[i][0],

//set diagonal entries Matrix[i][i] to 1; set Matrix [0][0] to numNodes.
// If you like, you may let the constructor do these.

- printMatrix (outFile2) // Print the entire content of Matrix with row and column indices in readable format.

- (int) findOrphan (…)
// Check Matrix [0][j] to find the next un-marked orphan node, j, i.e., Matrix [0][j] == 0 &&
//Matrix [j][j] == 1. If found, mark the orphan, i.e., set Matrix[j][j] ← 2, then returns j;
// if no such j, returns -1.

- OpenInsert (node) // inserts node into OPEN in ascending order with respect to the node's jobTime.
// Re-use codes similar to your earlier projects. On your own.

- printOPEN (outFile2) // Prints to outFile2, nodes in OPEN linked list.
// Re-use codes similar to the printList method in your earlier projects.
- (int) getNextProc (currentTime) // on your own
// check Table [i][currentTime] to find the first i where Table [i][currentTime] == 0
// if found returns i, else returns -1, means no available processor.

- fillOPEN (…) // populate OPEN from orphan nodes in the graph; see algorithm below.
- fillTable (…) // populate the table from jobs in OPEN; see algorithm below.

- putJobOnTable (availProc, currentTime, jobID, jobTime) // see algorithm below.

- printTable (outFile1, currentTime) // print the scheduleTable up to the currentTime slot to outFile1,
// On your own, see the format description given in the above.

- (bool) checkCycle (…) // on your own.
Check the followings:
(1) OPEN is empty.
(2) Graph is not empty. // you should know where to check.
(3) all processors are available. // you should know where to check.

if all 3 conditions in the above are true, returns true, else returns false.

- (bool) isGraphEmpty (…) // on your own
//if Matrix [0][0] == 0 returns true, else returns false. When you printMatrix, pay attention to
// the content of Matrix [0][0].

- deleteDoneJobs (…) // delete all done jobs from the graph; see algorithm steps below.

- deleteJob (jobID) // see algorithm steps below.
// When a job is done, we delete the job and its outgoing edges from the graph, as follows.
// 1) To delete a job in the graph is to set Matrix [jobID][jobID] to 0 and decrease the
number of nodes in graph by 1, i.e., Matrix [0][0] --
// 2) To delete a job's outgoing edges is to decrease the parent counts all its dependents by 1.
Note: the job's dependents are those none zero adjMatrix [jobID][j] > 0, on jobID row
For example, if Matrix [jobID][j] > 0, then decrease Matrix [0][j] by 1; i.e., Matrix [0][j]--

****

IV. main(..) // If you like, Step 1 to Step 4 can be done in the class constructor.

****

Step 1: inFile1, inFile2, outFile1, outFile2 ← open
numNodes ← read from inFile1.
numProcs ← get from argv [3]
if (numProcs <= 0) exit with error message "need 1 or more processors".
else if (numProcs > numNodes)
numProcs ← numNodes // means unlimited processors.
Step 2: Matrix ← dynamically allocate, size of numNodes+1 by numNodes+1, initialize to zero

loadMatrix (inFile1)
setMatrix (…)
printMatrix(outFile2) // check for your self to make sure the matrix is correctly set.
Step 3: OPEN ← get a dummy node for OPEN to point to
currentTime ← 0 // at the beginning of scheduling
procUsed ← 0 // no processor is used at the beginning
Step 4: totalJobTimes ← loadJobTimeAry (inFile2)
Table ← dynamically allocate, size of numProcs by totalJobTimes, initialize to zero
printTable (outFile1, currentTime)
Step 5: fillOPEN (…)
printOPEN (outFile2)
Step 6: fillTable (…)
printTable (outFile1, currentTime)
Step 7: currentTime ++
Step 8: deleteDoneJobs (…)
Step 9: if checkCycle (…) is true
output error message to **outFile1**: "there is cycle in the graph!!!" and exit the program
step 10: repeat step 3 to step 7 until isGraphEmpty (…)
step 11: printTable (outFile1) // The final schedule table.
step 12: close all files

****
V. fillOPEN (…)
****
Step 1: jobID ← findOrphan (…)
Step 2: if jobID > 0
newNode ← get a node with (jobID, jobTimeAry[jobID], null)
OpenInsert (newNode)
printOPEN(outFile2) // debug print
Step 3: repeat step 1 to step 2 until no more unmarked orphan

****
VI. fillTable (…)
****
Step 1: availProc ← getNextProc (currentTime)
if availProc >= 0 // means there is a processor available
{ newJob ← remove the front node of OPEN after dummy node // newJob is a node!
putJobOnTable (availProc, currentTime, newJob.jobID, newJob.jobTime)
if availProc > procUsed
procUsed ++
}
step 2: repeat step 1 while availProc >= 0 && OPEN is not empty && procUsed < numProcs

****
V. putJobOnTable (availProc, currentTime, jobId, jobTime)
****
Step 0: Time ← currentTime
EndTime ← Time + jobTime

Step 1: Table[availProc][Time] ← jobID
Step 2: Time ++
Step 3: repeat step 1 to step 2 while Time < EndTime

****
VII. deleteDoneJobs (…)

****
Step 1: proc← 0
Step 2: if Table[proc][currentTime] <= 0 && Table[proc][currentTime - 1] > 0
// meaning, the processor, proc, just finished a job in the previous time cycle.
{
jobID ← Table[proc][currentTime - 1]
deleteJob(jobID) // see algorithm steps below.
}
Step 3: printMatrix (outFile2)
step 4: proc ++
step 5: repeat step 2 to step 4 while proc < procUsed

****
VI. deleteJob (jobID) // When a job is done, we delete the job and its outgoing edges.
****
Step 1: Matrix [jobID][jobID] ← 0 // delete jobID from the graph
Matrix [0][0] -- // one less node in the graph

Step 2: j ← 1
Step 3: if Matrix [jobID][j] > 0 // means j is a dependent of jobID
Matrix [0][j] -- // decrease j's parent count by 1
Step 4: j ++
Step 5: repeat Step 3 to Step 4 while j <= numNodes