Project 9 (C++): Scheduling. Please read the supplemental guide to Lecture Note ahead of your reading of this specs.

Abstract: taking as inputs: (1) a dependency graph (where nodes are tasks, and edges are dependencies) and (2) the number of processors allows to use to process the tasks, this project can construct a schedule table for you or your customer, showing a complete layout of schedule to finish all tasks. The schedule table shows which time slot and which processor is doing which job, the maximum number of processors you need to do the tasks at any time lot, (good resource management) and length of time from the begin to end, to complete the entire tasks (good time management).

For example, you own a moving company and only own 6 trucks. You have 41 moving jobs to do this month. Each moving job has the dates to load and un-load, the places to load and unload, the distance of moving from places to load and unload and which job has to finish before which. You would want to have a schedule to complete all 41 jobs in a shortest time spend and use minimum trucks. This is a scheduling problem. Here, the nodes are moving jobs and dependency edges are which jobs need to do prior to which jobs.

Another example, you enter QC and want to get the CS BS degree. You can find a CS BS degree requirement chart the CS department. It shows which course have to take prior to which course, and the number of courses you must take (and passed) to obtain a BS in CS degree in QC. The CS BS chart is a dependency graph. Now, (1) you may want to know how many semesters you need to pay your tuitions to get your CS BS degree at QC; (2) you may want to know which courses to take in which semester; and (3) how many courses and which courses you need to take in each semester, giving that you pass each course in each semester. This is a scheduling problem. Here, the nodes are courses, and dependency edges are course pre-requisites.

To make a schedule table for “getting a CS BS degree, you need to provide the program with (1) a list of course pre-requisites (you can obtain the list from the CS BS chart), and the number of courses you have time for it in each semester. For example, say, course CSCI 211 and Math 141 need to take prior to taking 311; then, <211, 311> and <141, 311> are two dependency edges, meaning 211 and 141 are two parents of 311; another ward, or 311 is a dependent of 211 and 141. The number of courses you can take in each semester is either limited or un-limited processors).

In the implementation of the scheduling algorithm for this project, two input files are given: the first file contains the dependency graph, and the second file contains the processing times for nodes/jobs. For nodes with equal processing time, all nodes in the second file will be 1. In the data structures, you will use: an adjacent matrix to represent the input dependency graph; a 2D array for schedule table, a linked list, called OPEN, to maintain a list of nodes that are in the queue waiting to be processing / put on to take; a few 1D arrays for: nodes’ parents counts, nodes’ dependent counts, and status of nodes and processors , etc.

The scheduleTable is a 2D arrays of rows and columns, where index of rows represents processors ids, and the index of column represents time-slots. Each cell in the scheduleTable [i][j] is a job the processor i is processing the job t time j.

As you can see, given a dependency graph where nodes/jobs either have same processing or different processing time, and the number of processors can be either limited or unlimited. Therefore, there are four possibilities (listed below) for constructing a scheduling table (as well as given in the Lecture Notes and in the supplemental guide.) In this specs, nodes and jobs are the same thing and used interchangeable.

Prob 1: using limited processors and all nodes/jobs have equal (constant) processing time.

Prob 2: using un-limited processors and all nodes/jobs have equal (constant) processing time

Prob 3: using limited processors and nodes/jobs may have different processing time

Prob 4: using un-limited processors and nodes/job may have different processing time

In this project, you will implement an algorithm that covers the above four possibilites without writing four different methods to handle the four probabilities The abstract algorithm steps is given below; the implementation of the algorithm is to follow.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\* Algorithm steps for the Scheduling for all 4 options

Step 0: GϜ 🡨 given // a directed graph is given

numProc 🡨 given

// if numProc > number of nodes in GF, meaning – unlimited usage of processors;

//otherwise, it is limited to the given numProc.

currentTime 🡨 0 // Current processing time starts at zero

Step 1: orphan 🡨 get the next un-marked orphan node in GF // an orphan node does not have any parent

Step 2: mark the orphan // don’t get the same node twice

Step 3: newNode 🡨 get a node for the orphan

Step 4: insert newNode into OPEN // OPEN is the same as listHead used in all your previous projects dealing with an ordered linked list, OPEN points to a dummy node. Nodes in OPEN are sorted in the descending order with respect to the number of dependents of nodes in GF; i.e., a node with more dependents will be in the front of nodes with less dependents

Step 5: repeat Step 1 to Step 4 until no more un-marked orphan node can be found

Step 6: availableProc 🡨 ﬁnd the next available processor

// a processor is available if it does not currently processing any job

Step 7: nextNode 🡨 remove the front node, after dummy node, of OPEN linked list

Step 8: t 🡨 currentTime

jobTime 🡨 nextNode’s job time // it would 1, in the case of all nodes have equal processing time

Stee 9: scheduleTable [availableProc, t] 🡨 nextNode’s Id

// write the node’s id on the time slice on scheduleTable at the available processor’s row of the table

Step 10: t++

jobTime - - // decreased by 1

Step 11: repeat step 9 to step 10 while jobTime > 0 // for fix processing time, this loop will only goes once.

Step 12: repeat steps 6 to 11 until OPEN is empty or NO processor is available ( in the case of limited processor)

Step 13: currentTime++

Step 14: finishedNode 🡨 find the next job that is done processing by its processor

Step 15: delete the finishedNode from GF, also delete all finishedNode’s outgoing edges

Step 16: repeat Step 14 to Step 15 until no more finished nodes

Step 17: repeat step 1 to 16 until GF is empty or no more orphan node to be found.

// If GF is not empty and no more orphan node to be found, meaning there is a cycle in GF!

There are three major components in the abstract algorithm steps given in the above:

1. loading orphan nodes onto the OPEN linked list. (Step 1 to Step 5 of the algorithm)

2. moving nodes in OPEN to available (idle) processors, and fill the schedule table. (Step 6 to step 12)

3. deleting processed nodes and their dependents edges from the graphs. (Step 13 – Step 17)

A nice by-product of this algorithm is that, if the given dependency graph contains a cycle, it will detect it.

You will be given 2 sets of test data to test the correctness of your program, where each set includes two files -- a file contains the dependency graph (a list of partial relations) and a file contains processing time for each node. The number of processors to be used is given by you via command-line.

What you need to do for this project:

- Run your program once on data1 set (GraphData1 and JobTimeData1) using 3 processors.

GraphData1contains cycles, your program should detect it and report it. No schedule table!

- Run your program once on data2 set (GraphData2 and JobTimeData2) using 2 processors.

- Run your program once on data2 set (GraphData2 and JobTimeData2) using 3 processors.

- Run your program once on data2 set (GraphData2 and JobTimeData2) using numNodes + 3 processors

- Draw the dependency graph for GraphData1

- Draw the dependency graph for GraphData2

- (+1 pt) Draw an illustration (similar to the illustration shown in Lecture Notes) of data2 using 3 processors;

for each time cycle, showing the content changes of OPEN, the changes of the graph,

and the schedule table.

\*\*\* Include in your \*hard copies:

- cover page

- the drawings of GraphData1

- the drawings of GraphData2

- illustrations of GraphData using 3 processors, if you draw it for +1 pt

- source code

- outFile1 // result of data1 using 3 processors

- outFile2 // result of data1 using 3 processors

- outFile1 // result of data2 using 2 processors

- outFile2 // result of data2 using 2 processors

- outFile1 // result of data2 using 3 processors

- outFile2 // result of data2 using 3 processors

- outFile1 // result of data2 using numNodes + 3 processors

- outFile2 // result of data2 using numNodes + 3 processors

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Language: C++

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Project points: 12 pts

Due Date: Soft copy and pdf hard copies: 4/17/2020 Friday before midnight

+1 early submission: 4/13/2020 Monday before midnight

- 1 pt 1 day late: 4/18/2020 Saturday before midnight

- 3 pts 2 days late: 4/19/2020 Sunday before midnight

-12 pts: after 4/19/2020 Sunday after midnight

\*\*\* Name your pdf file using the same naming convention as given prior

\*\*\* All on-line submission MUST include Soft copy and pdf hard copy in the same email with correct file names; otherwise, it would not count as submission.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

I. Inputs:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1) inFile1 (use argv[1]): 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 edges (dependency) <ni, nj>, where 0 < ni , nj <= numNodes.

(Note: In this project, we do not use 0 for nodeID; nodeID from 1 to numNodes are in the graph.

For example:

6 // Graph has 6 nodes

1 2 // an edge from 1 to 2, i.e., 2 is a dependent of 1

1 4 // an edge from 1 to 4; 4 is a dependent of 1

4 3 // an edge from 4 to 3; 3 is a dependent of 4

4 2 // an edge from 4 to 2; 2 is a dependent of 4

:

2) inFile2 (use argv[2]): 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, <ni, ti>, where

ni is nodeID i, and ti is the unit of processing times for node ni.

An example for jobs take 1 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

:

:

An example for 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) numProc (use argv[3]): //to provide the number of processors are needed.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

II. Outputs:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1) outFile1: (use argv) . Use for all intermediate outputs and the final schedule table.

The schedule table needs to be nicely formatted.

For example:

--0---1---2---3---4---5---6---7---. . .

P(1) | 1 | 1 | 7 | 3 | 3 | 3 | - | 6 . . .

--------------------------------- . . .

P(2) | 2 | 4 | 4 | 4 | - | 5 | 5 | - . . .

--------------------------------- . . .

P(3) | etc.

2) outFile2: print all linked list as it calls for

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

III. Data structure:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- A Scheduling class

- A node class // for linked list

- jobId (int)

- jobTime (int) // node’s processing time

- dependentCount (int)

- next (node\*)

Method:

- constructor (jobId, jobTime, dependentCount)

- printNode (…) // copy from your previous project

* A JOBS struct or class

- jobTime (int) // processing time required for the job, provided in inFile2.

- onWhichProc (int) // initialized to 0; means not on any processor.

- onOpen (int) // initialized to 0; means not on Open.

- parentCount (int)

- dependentCount (int)

Method:

- constructor (. . .)

* A Proc struct or class

- doWhichJob (int) //which job it is processing, initialize -1, meaning - available

- timeRemain (int) //time remain on a job; <= 0 means it is available

- numNodes (int) // the total number of nodes in the input graph.

- numProcs (int) // the number of processors is avaiable be used.

- procUsed (int) // number of processors are used so far; initialized to 0.

- jobAry [] (JOBS) // a 1-D struct JOBS array of size of numNodes +1,

//the array index served as job id, to be dynamically allocated.

- procAry [] (Proc) // a 1-D Proc array of size of numProcs +1, the array index

//serve as Proc id, to be dynamically allocated.

- OPEN (node \*) // OPEN is a node pointer, the list head of the ordered linked list, points to a dummy node.

// Nodes in OPEN are sorted in descending order

// by the # of dependents of orphan nodes. i.e., nodes with more dependents will be in the front of nodes with less dependents.

- adjMatrix (int\*\*)

// a 2-D integer array, size numNodes+1 by numNodes+1,

// representing the input dependency graph;

// adjMatrix[i][j] == 1, means node i is a parent of j, or, node j is a dependent of node i.

// need to be dynamically allocated and initialized to zero.

- parentCountAry (int\*) // 1-D integer array to hold nodes’ parent counts;

// needs to be dynamically allocated.

// Since row 0 of adjMatrix[0][j] is not used, instead of using this array, you may use of //adjMatrix[0][j] to hold parentCount, j = 1 to numNodes, for the same purpose.

- dependentCountAry (int\*) // 1-D integer array to hold nodes’ dependent counts,

//needs to be dynamically allocated.

// Since column 0 of adjMatrix[i][0] is not used, instead of using this array, you may use //adjMatrix[i][0] to hold dependentCount, i = 1 to numNodes, for the same purpose.

- onGraphAry (int\*) // 1-D integer array to indicate whether a node has been remove from the graph;

//needs to be dynamically allocated and initialized to 1;

// onGraphAry[i] == 1 means node i is on the graph, 0 means it had been deleted.

// Since the diagonal of adjMatrix[i][i] is not used, instead of using this array, you may use //adjMatrix[i][i], for the same purpose.

- totalJobTimes (int) // the sum of all nodes’ job provided in inFile2.

- scheduleTable (int\*\*) // a 2-D integer array, size of (numProcs +1) by (totalJobTimes +1),

// where the index of rows are processors’ ID and columns are the time slices during the

// construction to record the schedul, initialized to 0.

// to be dynamically allocated.

- loadMatrix (…) // read each pair of (dependency) <ni, nj> from inFile1

// set adjMatrix[ni][ nj] <-- 1

- (int) constructJobAry(inFile2, adjMatrix) // it returns the totalJobTIme

// see algorithm below

- computeParentCount(adjMatrix, parentCountAry) // For each nodeId,

// parentCountAry[nodeId] 🡨 sum of adjMatrix[i][nodeId],

// jobAry[nodeId].parentCount 🡨 parentCountAry[nodeId]

// where i = 1 to numNodes. On your own.

- computeDependentCount(adjMatrix, dependentCountAry) // for each nodeId,

// dependentCountAry[nodeId] 🡨 sum of adjMatrix[nodeId][j],

// jobAry[nodeId].dependentCount 🡨 dependentCountAry[nodeId]

// where j = 1 to numNodes. On your own.

- (int) findOrphan() // Go thru jobAry to find the next orphan node, i,

// if parentCountAry[i] <= 0 && jobAry[i].onOpen == 0 &&

//jobAry[i].onWhichProc == 0

// returns i, else return -1, if there is none orphan can be found.

// On your own.

- listInsert (…) // similar to your project 1 and 2 and 3

// insert node into OPEN in the order by the # of dependents of node,

// in descending order. Re-use codes in your earlier project.

- findSpot (…) // similar to your project 1 and 2 and 3. Re-use codes in your earlier project.

- printList (outFile2) // print to outFile2, nodes in OPEN linked list using the similar format as

// in your earlier project.

- printScheduleTable (outFile1) // print the scheduleTable to outFile1,

// see the format given in the above.

- (int) findProcessor( ) // Go thru ProcAry to find the 1st available processor;

// if ProcAry[i].timeRemain <= 0

// return i, else return -1 if all processors are busy

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

// newJob will take up as many time slots on the schedule table

//as the newJob’s jobTime requires.

- (bool) checkCycle ()

Check the followings:

(1) OPEN is empty

(2) Graph is not empty // you should know where to check

(3) all processors finished all the jobs

// you should know where to check

if all 3 conditions in the above are true,

returns true

else returns false

* (bool) graphIsEmpty () // if onGraph[i] == 0 for all i

// return true else return false

* updateProcTime () // decrease procAry[i].timeRemain by 1 for all used processors.
* (int) findDoneProc() //Go thru ProcAry, find an available next processor just finished a job, i.e.,

// if ProcAry[i].doWhichJob >0 && ProcAry[i].timeRemain <= 0

// jobID 🡨 ProcAry[i].doWhichJob

// ProcAry[i].doWhichJob 🡨 -1 // ProcAry[i] is now not busy

// return jobID

- deleteFinishedNodes (…) //See algorithm below.

- deleteEdge (jobId) // delete all jobId’s outgoing edges, i.e.,

// Go thru jobID’s dependents, check the jobId row of adjMatrix,

// if adjMatrix[jobId][dependent] > 0

//decrease parentCount[dependent] by 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

IV. main(..)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Algorithms may contain bugs, debugging is yours.

Step 0: inFile1, inFile1, outFile1, outFile2 🡨open

numProcs 🡨 from argv[3]

if (numProcs > numNodes)

numProcs 🡨 numNodes // means unlimited processors, why?

Step 1: initialization (…) // see algorithm below.

Step 2: loadOpen(…) // see algorithm below.

Step 3: printList(Open, outFile2) // debug print

Step 4 loadProcAry(…) // see algorithm below.

Step 5: hasCycle 🡨 checkCycle (…) // on your own, see the description in the above.

if hasCycle == true

output error message to console: “there is cycle in the graph!!!”

and exit the program

step 6: printScheduleTable (outFile1) // print intermediate schedule table to outFile1

step 7: currentTime++

step 8: updateProcTime (…) // on your own, see the description in the above.

step 9: deleteFinishedNodes (…)

step 10: repeat step 2 to step 11 until graphIsEmpty (…)

step 11: printScheduleTable (outFile1) // The final schedule table to outFile1

step 12: close all files

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

V. initialization (inFile1, inFile2) // can be done within constructor(s)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 1: procUsed 🡨 0 // 0 processor is used at the beginning.

currentTime 🡨 0 // at the beginning of scheduling

Open 🡨 make a new linked list with a dummy node for Open to point to

Step 2: numNodes 🡨 read from inFile1.

Step 3: adjMatrix 🡨 dynamically allocate, size of numNodes+1 by numNodes+1

// initialized to zero

jobAry 🡨 dynamically allocate, size of numNodes+1

procAry 🡨 dynamically allocate, size of numProcs+1

scheduleTable 🡨 dynamically allocate size of (numProcs +1) by (totalJobTimes +1),

Step 4: loadMatrix (inFile1) // on your own, see description in the above.

Step 5: computeParentCount(adjMatrix, parentCountAry) // on your own

Step 6: computeDependentCount(adjMatrix, dependentCountAry) // on your own

Step 7: totalJobTimes 🡨 constructJobAry(infile2, adjMatrix) // see algorithm below.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

VIIII. putJobOnTable (availProc, currentTime, jobId, jobTime)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 0: Time 🡨 currentTime

EndTime 🡨 Time + jobTime

Step 1: scheduleTable[availProc][Time] 🡨 jobId

Step 2: Time ++

Step 3: repeat step 1 to step 2 while Time < EndTime

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

VI. (int) constructJobAry(inFile2, …)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 0: totalTime 🡨 0

Step 1: nodeID 🡨 read from inFile2

jobTime 🡨 read from inFile2

totalTime += jobTime

Step 2: jobAry[nodeID].jobTime 🡨 jobTime

jobAry[nodeID].onWhichProc 🡨 -1

jobAry[nodeID].onOpen 🡨 0

jobAry[nodeID].parentCount 🡨 parentCountAry [nodeID]

jobAry[nodeID]. dependentCount 🡨 dependentCount [nodeID]

Step 3: repeat step 1 to step 2 until eof inFile2

Step 4: return totalTime

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

VII. loadOpen() // find orphan nodes and put them on Open.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

step 1: orphanNode <-- findOrphan()

Step 2: if orphanNode > 0

jobId 🡨 orphanNode

jobTime 🡨 jobAry[jobId].jobTime

newNode <-- make a node (jobId, jobTime, dependentCountAry [jobId])

listInsert(newNode)

jobAry[jobId].onOpen 🡨 1

step 3: repeat step 1 – step 2 until no more orphenNode in the graph,

// i.e., orphanNode == -1.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

VIII. loadProcAry() // find available processor to process nodes on the open.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

step 1: availProc <-- findProcessor( )

Step 2: if availProc > 0 // there is a processor available

ProcUsed ++

newJob <-- remove the front node of OPEN // newJob is a node!

jobId 🡨 newJob’s jobId

jobTime 🡨 newJob’s jobTime

procAry [availProc]. doWhichJob <-- jobId

procAry [availProc]. timeRemine <-- jobTime

putJobOnTable (availProc, currentTime, jobId, jobTime)

step 3: repeat step 1 to step 2 while availProc > 0

&& OPEN is not empty && ProcUsed < numProcs

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

VIII. deleteFinishedNodes (…)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 1: jobId 🡨 findDoneProc() // on your own, see the description in the above.

Step 2: if jobId > 0

onGraphAry[jobId] 🡨 0 // delete jobId from the graph

deleteEdge(jobId) // on your own, see the description in the above.

step 3: repeat step 1 to step 2 while jobId > 0

// means no more processor finished its job