Project 9: Scheduling. You are to implement the Partial Ordering, dependency graph, and scheduling problem.

Problem statement:

Given a directed (dependency) graph, G = <N, E>, the task is to construct a schedule table based on the given partial-ordering (list of edges) in the graph. If the graph has a cycle, you will report to the user that the given graph contains cycles.

In this specs, nodes and jobs are the same thing and used interchangeable.

As taught in the class, there are four options in the scheduling:

(1) using limited processors where each job takes 1 unit of processing time;

(2) using unlimited processors where each job takes 1 unit of processing time;

(3) using limited processors where jobs take variable unit of processing time;

(4) using unlimited processors where jobs take variable unit of processing time.

In this project we will combine the above four options into one.

That is, your project will be able to handle all these four options!

The number of processors are given by user from console.

You will be given two sets of test data, where each set include a file contains the dependency graph and a file contains processing time for each node.

1) The graph in data1 contains cycles and job time for each node is 1.

a) Run your program once on data1 set using 3 processors.

2) The graph in data2 does not contain cycles and job times for nodes are variable. Run your program three times as follows:

b) Run your program once on data2 set using 2 processors.

c) Run your program once on data2 set using 3 processors.

d) Run your program with numNodes + 3 processors

\*\*\* Include in your \*hard copies the four results of the four runs (a, b, c, d) of the above.

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

Language: **C++**

Due date (soft copy): 5/7/2019 Tuesday before Midnight

Early submission +5 deadline: 5/2/2019 Thursday before Midnight

Early submission +2 deadline: 5/5/2019 Sunday before Midnight

-1 pt due: 5/8/2019 Wednesday before midnight

After 5/8/2019, -5 pts for all students who did not submit soft copy

Due Date (hard copy): 5/9/2019 Thursday in class,

-1 pt for late hard copy submission after Thursday 5/9/2019

All projects without hard copy after 5/10/2019 will receive 0 pts even you have submitted soft copy on time and even if it works.

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

1) inFile1 (use argv): 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 as nodeId and nodeId from 1 to numNodes are in the graph.

For example:

6 // Graph has 6 nodes

1 2 // 2 is a dependent of 1

1 4 // 4 is a dependent of 1

4 3 // 3 is a dependent of 4

4 2 // 2 is a dependent of 4

:

:

2) inFile2 (use argv): 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 the node’s id and ti is the unit of processing times for node ni.

For example: 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

:

:

for 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) Console: \*\*\* Ask user from console to provide the number of processors are needed.

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

II. Outputs:

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

1) outFile1: (use argv)

The schedule table, 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 your debugging needs, but do \*not\* include them in your hardcopy!!

3) console output as indicate in the algorithms.

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

III. Data structure:

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

- A Scheduling class

- A node class

- jobId (int)

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

- dependentCount (int)

- next (node\*)

Method:

- constructor (jobId, jobTime, dependentCount)

* A JOBS struct or class
* jobTime (int) // processing time required, 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
* 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 can be used.

- procUsed (int) // number of processors 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 served as Proc id, to be dynamically allocated.

- OPEN (node \*) // OPEN is a node pointer, acts as the list head of the //linked list with 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 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, //of node 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 of adjMatrix[i][0] is not used, instead of using //this array, you may use adjMatrix[i][0] to hold dependentCount, //node -i = 1 to numNodes, for the same purpose.

- onGraphAry (int\*) // 1-D integer array to indicate whether a node is in //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),

// to record the schedule, initialized to 0.

// to be dynamically allocated.

Methods:

- initialization (inFile1, inFile2)) // see algorithm steps below

- loadMatrix (inFile1) // read an edge <ni, nj>from inFile1 and

// set adjMatrix[ni, nj] to 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],

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

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

// dependentCountAry[nodeId] 🡨 sum of adjMatrix[nodeId][j], the // where j = 1 to numNodes. On your own.

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

// if jobAry[i].parentCount <= 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.

- OpenInsert(node) // Perform a linked list insertion sort;

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

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

- printList(outFile2) // print to outFile2, nodes in OPEN linked list // Re-use codes of your printList method in your earlier project.

- printScheduleTablfe (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 next processor just finished

// a job, returns the finish jobId, i.e.,

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

// jobID 🡨 ProcAry[i].doWhichJob

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

// return jobID

- 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

//parentCount[dependent] –-

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

IV. main(..)

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

step 0: inFile1, inFile1, outFile1, outFile2 <-- open

step 1: initialization (inFile1, inFile2) // see algorithm below.

Step 2: procUsed <-- 0 // 0 processor is used at the beginning.

Step 3: currentTime <-- 0 // at the beginning of scheduling

Step 4: loadOpen() // see algorithm below.

step 5: printList(Open, outFile2) // debug print

step 6: loadProcAry() // see algorithm below.

step 7: 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 8: printScheduleTable (outFile1) // print intermediate schedule table

step 9: currentTime++

step 10: updateProcTime () // on your own, see the description in the above.

step 11: jobId 🡨 findDoneProc() // on your own, see the description in the above.

Step 12: if jobId > 0

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

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

step 13: repeat step 11 to step 12 while jobId > 0

// means no more processor finished its job

step 14: repeat step 4 to step 13 until graphIsEmpty ()

step 15: printScheduleTable (outFile1) // The final schedule table.

step 16: close all files

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

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

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

Step 0: Open 🡨 make a dummy node for Open to point to

Step 1: numProcs <-- from console, ask user how many processors are allow to use.

if (numProcs <= 0) ask the user again with message “need 1 or more proc”.

else if (numProcs > numNodes)

numProcs <-- numNodes // means unlimited processors.

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 numNodes+1

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

Step 5: totalJobTimes <-- constructJobAry(infile2, adjMatrix)

// see algorithm below.

Step 6: computeParentCount(adjMatrix, parentCountAry) // on your own

Step 7: computeDependentCount(adjMatrix, dependentCountAry) // on your own

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

VI. int constructJobAry(inFile2, jobAry, adjMatrix)

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

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])

OpenInsert(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

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

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