Project 8: 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 are also refer to as jobs.

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

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

(2)using unlimited processors where each job takes only 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 in 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 test files:

1) Data1: The graph in Data1 contains cycles.

a) Run your program once on Data1 using 3 processors.

2) Data2: The graph in Data2 does not contain cycles.

b) Run your program once on Data2 using 2 processors.

c) Run your program once on Data2 using 3 processors.

d) Run your program with numNodes + 3 processors

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

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

Language: C++

Due date: soft copies: 11/19/2018 Monday before Midnight

+2 Early submission deadline: 11/17/2018, Saturday before midnight

Due date: hard copies: 11/20/2018 Tuesday in class

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

1) input-1 (use argv):

a text file representing the dependency graph, G=<N, E>.

The first number in the input file is the number of nodes in the graph;

then follows by a list of edges (dependency) <ni, nj>.

for example:

6 // there are 6 nodes in the input graph

1 2 // 1 must be done before 2

1 4 // 1 must be done before 4

4 3 // 4 must be done before 3

4 2 // 4 must be done before 2

:

:

2) input-2 (use argv): a text file contains the time requirements for jobs.

The first number in the input file is the number of nodes in the graph;

then follows by a list of pairs, <ni, ti>, where

ni is the node id and ti is the unit of times require by ni.

for example:

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

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

II. Outputs:

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

1) outFile\_1: (use argv)

The scheule 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) outFile\_2: print all your debugging needs, but do \*not\* include them in your hardcopy!!

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

II. Data structure:

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

- A Scheduling class

- A node class

- jobId (int) //the node's Id.

- jobTime (int) // the processing time for the node

- constructor()

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

- totalJobTimes (int) // the total job times for all nodes.

- adjacencyMatrix (int\*\*)

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

// representing the input dependency graph,

// need to be dynamically allocated;

// adjacencyMatrix[i][j] >= 1, means job i has to finish before job j.

- scheduleTable (int\*\*)

// a 2-D integer array, (need to be dynamically allocated)

// to record the schedule.

// The dimension should be determined at run time,

// (numNodes +1) by (totalJobTimes +1)

- jobTimeAry (int\*)

// an 1-D array (initialize to 0)

// to store the time requirement for each job; get it from input2

// If you like, you can also store the job time of i,

// in adjacencyMatrix[i][i]

- OPEN (node \*)

// OPEN is a linked list (with a dummy node),

// to store jobs that do not have any parents.

// Nodes in OPEN are sorted in descending order

// by the # of dependents. i.e.,

// nodes with more dependents will be in the front

// This can be counted from adjacencyMatrix [i]

- processJob (int\*) // an 1-D array (initialize to 0) to indicate

// which job is being processed by processor[i].

// i.e., if processJob[i] is 5,

// it means processor, i, is currently processing job 5.

- processTime (int\*) // an 1-D array (initialize to 0) to keep track of

// the processing time remain on processor, i,

// When ProcessTime[i] > 0,

//it means the processor, i, is busy

// When ProcessTime[i] <= 0,

// it means the processor, i, is available.

- parentCount (int\*) // an 1-D array (initialize to 0)

// to store number of parents of each job

// (0 means no parent, an orphan.)

// the sum of each \*column j\* in the matrix is

// the total parent counts of node j.

- kidCount (int\*) // an 1-D array (initialize to 0)

// to store number of dependents of each job

- jobDone (int\*) // an 1-D array (initialize to 0)

// to keep track which jobs remain in the graph

// jobDone[i] == 1 means job i has been deleted from the graph,

// == 0 means still in the graph.

- jobMarked(int\*)// an 1-D array (initialize to 0) to keep track

// to see if jobs are currently on Open or

// on processor.(1 means is on OPEN/processor, 0 means is not).

>>> The dimension of all 1-D arrays should be (numberNodes + 1) array index 0 will not be used.

>>>> Methods: You must write each of the following as a method!!

Some of the methods will be on the next exam.

- loadMatrix (inFile1) // load the matrix from inFile1

- computeTotalJobTimes (inFile2)

// returns the sum of all jobs times given in inFile2.

- getUnMarkOrphen() // get the next unmarked orphan node, i,

// if jobMarked[i] == 0 \*and\* parentCount[i] == 0

// returns i,

// if none exist, returns -1

- insert2Open(node)

// insert node on to OPEN in the order of node’s jobTime

- printList(OPEN) // print to the \*console\* OPEN linked list,

// use the similar format as in your previous project.

- printTable()

// print the scheduleTable to outFile,

// using the format given in the above.

- findProcessor( ) // For i from 1 to ProcGiven

// if ProcessTime[i] <= 0, means processor i is available.

return i,

else return -1

- updateTable(availProc, newJob, currentTime)

// from scheduleTable[availProc][currentTime] to

// scheduleTable[availProc][currentTime + newJob’s jobTime]

// write with newJob’s jobId

- checkCycle ( )

//if (1), (2) and (3) below are all true,

// then the graph contains cycles:

(1) OPEN is empty

(2) Graph is not empty

// how do you know? Which array do you check?

(3) all processors finished all the jobs

// how do you know? Which array do you check?

if all 3 conditions are true,

returns 1

else returns 0

- findDoneJob() // find the job that is done by processor i,

// i.e., processTIME [i] == 0 ;

// for each processTIME [i] == 0,

// do the following in sequence:

// (1) \*\*save jobId, i.e.,

// the job has been processed by processor i;

// (2) set processJob[i] to 0, i.e.,

// processor i, no longer processing that job;

// (3) returns the saved jobId.

- deleteNode(node) // delete the node from the graph

// by setting jobDone[node’s jobId] <--1

- deleteEdge (node)

// delete all node's outgoing edges // node's kids

// for kidIndex from 1 to numNodes

// if adjacencyMatrix [node.jobId][kidIndex] > 0,

// parentCount[kidIndex] --

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

III. main(..)

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

step 0: inFile1 <-- open input-1 file from argv

inFile2 <-- open input-2 file from argv

outFile <-- open output file from argv

numNodes <-- get from inFile1.

loadMatrix (inFile1)

totalJobTimes <-- computeTotalJobTimes (inFile2)

procGiven <-- from console, ask user how many processors are needed

if (procGiven <= 0) returns with error message

if (procGiven > numNodes)

procGiven <-- numNodes

// There is no need to have more processors than the number of nodes.

- dynamically allocate all 1-D and 2-D arrays and initializing all arrays.

- procUsed <-- 0 // the actual number of processors used is 0 at the beginning.

- currentTime <-- 0

step 1: orphenNode <-- getUnMarkOrphen()

jobMarked[orphenNode] <-- 1

newNode <-- create a new node for orphenNode with its jobId and jobTime

// newNode has two fields, jobId and jobTime

// jobTime can be found in jobTime[] array

Insert2Open(newNode)

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

// orphenNode is -1.

step 3: printList(OPEN)

step 4: availProc <-- findProcessor( )

if availProc > 0 // there is a processor available

ProcUsed ++

newJob <-- remove from the front of OPEN list

processJob[availProc]<-- newJob’s jobId

processTime[availProc] <-- newJob’s jobTime

updateTable(availProc, currentTime, newJob)

step 5: repeat 4 while OPEN is not empty \*and\* ProcUsed < ProcGiven

step 6: if checkCycle ( ) >= 1

output error message and exit (there is cycle in the graph)

step 7: printTable()

step 8: currentTime++

step 9: Decrease all processTime[i] by 1

step 10: job <-- findDoneJob()

// find a job that is done, ie., processTIME [i] == 0 ;

// findDoneJob also deletes the job from the processJob[i]

// (set processJob[i] to 0)

Step 11 deleteNode(job)

deleteEdge(job)

step 12: repeat step 10 – step 11 until no more finished job

step 13: debugging print the following to outFile2

with readable headings for each:

- currentTime,

- jobMarked[ ] 1D array

- processTime[ ] 1D array

- processJob[ ] 1D array

- jobDone [ ] 1D array

step 14: repeat step 1 to step 13 until the graph is empty

(i.e., jobDone[i] are == 1. )

step 15: printTable()

step 16: close all files