CSC 323 Spring 2015: Dependency Graph (Java)

Ravi Patel

May 12, 2015

**Algorithm steps:**

\*\* Algorithm steps for dependency job scheduling for all cases.

step 0: prepare everyhing

ProcLimit <-- get from user or the maximum of number of jobs.

ProcUsed <-- 0 // the actual number of processors used

Time <-- 0

step 1: orphenNode <-- find an unmarked node on the dependancy

graph that does not have any parent

mark orphenNode

put orphenNode onto OPEN list

step 2: Repeat step 1 as long as there is orphenNode.

step 3: availProc <-- find an available used processor

If no used processor available and ProcUsed < ProcLimit

ProcUsed ++

availProc <-- get the next unused processor

If there is availProc // a processor either used or new

newJob <-- remove a job from OPEN

give newJob to the avalilProc for processing

record the newJob processing time

put the job onto scheduleTable[availProc]

(according to TIME status and job's time units)

step 4: repeat 3 as long as OPEN is not empty

\*and\* ProcUsed < ProcNeed

step 5: if OPEN is empty and D.G is not empty

and all processors are idled (not working)

report error and exit (a cycle in the graph)

step 6: Time++

step 7: Decrease all processor's time by 1

step 8: job <-- find a job that is finished by its processor

delete the job from its processor

delete the job from the graph

delete all it's outgoing arcs

step 9: repeat 8 until no more finished job

step 10: repeat step 1 to step 9 until the D.G. graph is empty

Source Code:

public class Node {

private int JobNode;

private Node next;

public Node()

{

JobNode = 0;

next = null;

}

public Node (int newJobID)

{

JobNode = newJobID;

next = null;

}

public int getJobNode()

{

return JobNode;

}

public Node getNext()

{

return next;

}

public void setNext(Node newNext)

{

next = newNext;

}

}

import java.io.PrintWriter;

public class LinkedList {

private Node head;

private int size;

public LinkedList()

{

head = new Node(0);

size = 0;

}

public void addToList(int job)

{

Node newNode = new Node(job);

if (isEmpty())

{

head.setNext(newNode);

size++;

}

else

{

Node current = head;

while (current != null) {

if (current.getNext() == null)

{

break;

}

current = current.getNext();

}

current.setNext(newNode);

size++;

}

}

void removeFirst()

{

Node temp;

temp = head.getNext().getNext();

head.setNext(temp);

size--;

}

void printLinkedList(PrintWriter outputFile)

{

for (Node walker = head.getNext(); walker != null; walker = walker.getNext())

{

outputFile.print(walker.getJobNode()+" ");

}

outputFile.print("\n");

}

public int getSize()

{

return size;

}

public boolean isEmpty()

{

return size == 0;

}

public Node getHeadNode()

{

return head;

}

}

import java.io.\*;

import java.util.Scanner;

public class Main {

public static void main(String[] args) throws IOException{

if(args.length==0) System.out.println("File not specified");

else{

Scanner DependencyFile = new Scanner(new FileReader(args[0]));

Scanner JobTimeFile = new Scanner(new FileReader(args[1]));

int TotalProcessors = (Integer)Integer.parseInt(args[2]);

PrintWriter outputFile = new PrintWriter(args[3]);

int NumberOfNodes1, NumberOfNodes2;

NumberOfNodes1 = DependencyFile.nextInt();

NumberOfNodes2 = JobTimeFile.nextInt();

if ( NumberOfNodes1 != NumberOfNodes2 ){

outputFile.println("There is an error in number of nodes");

}

if (TotalProcessors > NumberOfNodes1){

TotalProcessors = NumberOfNodes1;

}

int[] jobTime = new int[NumberOfNodes1];

int[] processJob = new int[TotalProcessors];

int[] processTime = new int[TotalProcessors];

int[] parentCount = new int[NumberOfNodes1];

int[] jobMarked = new int[NumberOfNodes1];

int[] jobDone = new int[NumberOfNodes1];

int[][] scheduleTime;

LinkedList openList = new LinkedList();

LinkedList[] hashTable = new LinkedList[NumberOfNodes1];

int TotalTime = 0, NodeNumber, Time;

while (JobTimeFile.hasNext()){

NodeNumber = JobTimeFile.nextInt();

Time = JobTimeFile.nextInt();

for (int row = 0; row < NumberOfNodes1; row++){

if ( row == NodeNumber-1){

jobTime[row] = Time;

TotalTime += Time;

}

}

}

scheduleTime = new int[TotalProcessors][TotalTime];

FillScheduleTime(scheduleTime, TotalProcessors, TotalTime);

for (int row = 0; row < NumberOfNodes1; row++){

hashTable[row] = new LinkedList();

}

int BeforeNode, AfterNode;

while (DependencyFile.hasNext()){

BeforeNode = DependencyFile.nextInt();

AfterNode = DependencyFile.nextInt();

for (int row = 0; row < NumberOfNodes1; row++){

if (row == (BeforeNode-1) ){

hashTable[row].addToList(AfterNode);

}

}

}

for (int row = 0 ; row < NumberOfNodes1; row++){

parentCount[row]=0;

jobDone[row]=0;

jobMarked[row]=0;

}

for (int row = 0 ; row < TotalProcessors; row++)

{

processJob[row]=0;

processTime[row]=0;

}

int availProcess, ProcessorUsed = 0, time=0, orphaneNode;

for (int row = 0; row < NumberOfNodes1; row++)

{

for (Node walker = hashTable[row].getHeadNode().getNext(); walker!=null;

walker = walker.getNext())

{

parentCount[walker.getJobNode()-1] +=1;

}

}

while (true) //<----- needs work on

{

for (int row = 0; row < NumberOfNodes1; row++)

{

if (parentCount[row] == 0 && jobMarked[row] == 0)

{

orphaneNode = row;

jobMarked[orphaneNode] = 1;

openList.addToList(orphaneNode+1);

}

}

//========================= Step 2-3 ===========================

while (openList.getSize() != 0 || ProcessorUsed > TotalProcessors)

{

availProcess = findProcess(processJob, ProcessorUsed);

//cout<<"-----"<<availProcess<<endl;

if (availProcess == -1)

{

ProcessorUsed++;

availProcess = ProcessorUsed;

}

if (ProcessorUsed < TotalProcessors)

{

int newJob = openList.getHeadNode().getNext().getJobNode();

openList.removeFirst();

processJob[availProcess] = newJob;

processTime[availProcess] = jobTime[newJob - 1];

scheduleTime[availProcess][time] = newJob;

}

}

//============================ Step 4 ===========================

if (!openList.isEmpty() && DG(jobMarked, NumberOfNodes1) && JobsFinished(processJob, TotalProcessors))

{

outputFile.println("There is a cycle in the graph");

System.exit(0);

}

outputFile.println("OpenList");

openList.printLinkedList(outputFile);

//============================ Step 5 ============================

outputFile.println("Time: "+time+" ProcessorUsed "+ProcessorUsed);

outputFile.println("Hash Table");

for (int row = 0; row < NumberOfNodes1; row++)

{

outputFile.print(row+1+" ");hashTable[row].printLinkedList(outputFile);

}

outputFile.println("Schedule");

PrintSchedule(scheduleTime, TotalProcessors, TotalTime, outputFile);

outputFile.println("Job Time");

PrintArray(jobTime, NumberOfNodes1, outputFile);

outputFile.println("Process Job");

PrintArray(processJob, TotalProcessors, outputFile);

outputFile.println("Process Time");

PrintArray(processTime, TotalProcessors, outputFile);

outputFile.println("Parent Count");

PrintArray(parentCount, NumberOfNodes1, outputFile);

outputFile.print("Job Marked");

PrintArray(jobMarked, NumberOfNodes1, outputFile);

outputFile.println("Job Done");

PrintArray(jobDone, NumberOfNodes1, outputFile);

//============================ Step 6-7 ============================

time++;

decreaseTime(processTime, TotalProcessors);

//============================ Step 8-9 ============================

while ( processJobisFull(processJob, TotalProcessors) ){

int finishedJob = findFinishedJob(processTime, processJob,

TotalProcessors);

if (finishedJob == 0)

{

break;

}

outputFile.println("\*\*\*\*\*"+finishedJob+"\*\*\*\*\*\*");

deleteProcessJob(processJob, TotalProcessors, finishedJob);

for (Node walker = hashTable[finishedJob-1].getHeadNode().getNext();

walker!=null; walker = walker.getNext())

{

int node = walker.getJobNode();

parentCount[node-1] -= 1;

}

updateJobDone(jobDone, finishedJob);

}

//=========================== Step 10-11 ===========================

outputFile.println("Schedule");

outputFile.println("Time: "+time+" ProcessorUsed "+ProcessorUsed);

outputFile.println("Job Time");

PrintArray(jobTime, NumberOfNodes1, outputFile);

outputFile.println("Process Job");

PrintArray(processJob, TotalProcessors, outputFile);

outputFile.println("Process Time");

PrintArray(processTime, TotalProcessors, outputFile);

outputFile.println("Parent Count");

PrintArray(parentCount, NumberOfNodes1, outputFile);

outputFile.print("Job Marked");

PrintArray(jobMarked, NumberOfNodes1, outputFile);

outputFile.println("Job Done");

PrintArray(jobDone, NumberOfNodes1, outputFile);

if (jobDoneisNotFull(jobDone, NumberOfNodes1)) {

break;

}

}

PrintSchedule(scheduleTime, TotalProcessors, TotalTime, outputFile);

DependencyFile.close();

}

}

private static int findProcess(int[] processJob, int processorUsed) {

for (int row = 0; row <= processorUsed; row++)

{

if (processJob[row] <= 0)

{

return row;

}

}

return -1;

}

private static boolean DG(int[] jobMarked, int numberOfNodes1) {

for (int row = 0; row < numberOfNodes1; row++) {

if (jobMarked[row] == 0) {

return true;

}

}

return false;

}

private static boolean JobsFinished(int[] processJob, int totalProcessors) {

for (int row = 0; row < totalProcessors; row++)

{

if (processJob[row] != 0){

return false;

}

}

return true;

}

private static void decreaseTime(int[] processTime, int totalProcessors) {

for (int row = 0; row < totalProcessors; row++)

{

if (processTime[row] > 0)

{

processTime[row] -= 1;

}

}

}

private static boolean processJobisFull(int[] processJob,

int totalProcessors) {

for (int row = 0; row < totalProcessors; row++)

{

if (processJob[row] > 0)

{

return true;

}

}

return false;

}

private static int findFinishedJob(int[] processTime, int[] processJob,

int totalProcessors) {

int finishedJob = 0;

for (int row = 0; row < totalProcessors; row++)

{

if (processTime[row] == 0 && processJob[row] != 0)

{

finishedJob = processJob[row];

return finishedJob;

}

}

return finishedJob;

}

private static void deleteProcessJob(int[] processJob, int totalProcessors,

int finishedJob) {

for (int row = 0; row < totalProcessors; row++)

{

if (processJob[row] == finishedJob)

{

processJob[row] = 0;

}

}

}

private static void updateJobDone(int[] jobDone, int finishedJob) {

if (finishedJob == 0){

return;

}

else jobDone[finishedJob - 1] = 1;

}

private static boolean jobDoneisNotFull(int[] jobDone, int numberOfNodes1) {

for (int row = 0; row < numberOfNodes1; row++){

if (jobDone[row] != 1){

return false;

}

}

return true;

}

private static void PrintSchedule(int[][] scheduleTime,

int totalProcessors, int totalTime, PrintWriter outputFile) {

for(int row = 0; row < totalProcessors; row++){

for(int col = 0; col < totalTime; col++){

outputFile.println(scheduleTime[row][col]+" ");

}

outputFile.println();

}

}

private static void PrintArray(int[] Array, int size,

PrintWriter outputFile) {

for(int row = 0; row < size; row++){

outputFile.println(row+1 +" "+ Array[row]);

}

}

private static void FillScheduleTime(int[][] scheduleTime,

int totalProcessors, int totalTime) {

for (int row = 0; row < totalProcessors; row++){

for (int col =0; col < totalTime; col++){

scheduleTime[row][col] = 0;

}

}

}

}

Input File:

15

1 2

1 3

7 3

9 3

4 2

4 6

4 5

2 6

3 6

3 10

3 8

5 13

8 15

8 11

6 11

6 12

6 13

5 14

12 14

Input File 2:

15

1 2

2 3

3 1

4 2

5 4

6 2

7 4

8 2

9 3

10 1

11 2

12 1

13 5

14 1

15 3

Output File 1:

1 0 2 0 0 6 0 11 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

4 0 5 0 0 0 0 12 14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

7 0 0 0 3 8 0 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

9 0 0 0 0 10 0 15 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Output File 2:

1 0 2 0 0 6 0 11 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

4 0 5 0 0 0 0 12 14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

7 0 0 0 3 8 0 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

9 0 0 0 0 10 0 15 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Output 3:

1 2 6 11 14 0 0 0 0 0 0 0 0 0 0

4 3 8 12 0 0 0 0 0 0 0 0 0 0 0

7 5 10 13 0 0 0 0 0 0 0 0 0 0 0

9 0 0 15 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Output 4:

1 2 6 11 14 0 0 0 0 0 0 0 0 0 0

4 3 8 12 0 0 0 0 0 0 0 0 0 0 0

7 5 10 13 0 0 0 0 0 0 0 0 0 0 0

9 0 0 15 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0