CSC 323 Spring 2015: Dependency Graph (C++)

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:

#include <stdio.h>

#include <fstream>

#include <iostream>

#include <sstream>

using namespace std;

//=========================== Node ===============================

class Node {

private:

int JobNode;

Node\* next;

public:

Node()

{

JobNode = 0;

next = NULL;

}

Node (int newJobID)

{

JobNode = newJobID;

next = NULL;

}

int getJobNode()

{

return JobNode;

}

Node\* getNext()

{

return next;

}

void setNext(Node\* newNext)

{

next = newNext;

}

};

//==================== Linked List ================================

class LinkedList {

private:

Node\* head;

int size;

public:

LinkedList()

{

head = new Node(0);

size = 0;

}

void addToList(int job)

{

Node \*newNode = new Node(job);

if (isEmpty())

{

head->setNext(newNode);

size++;

}

else

{

Node \*current = head;

while (current) {

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(ostream& outputFile)

{

for (Node\* walker = head->getNext(); walker != NULL; walker = walker->getNext())

{

outputFile<<walker->getJobNode()<<" ";

}

outputFile<<endl;

}

int getSize()

{

return size;

}

bool isEmpty()

{

return size == 0;

}

Node\* getHeadNode()

{

return head;

}

};

//=================== Main =======================================

void PrintArray (int\* Array, int Size, ostream& outputFile)

{

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

{

outputFile<<row+1<<" "<<Array[row]<<endl;

}

}

void FillSchedule (int\*\* scheduleTime, int TotalProcessors, int TotalTime)

{

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

{

for (int col = 0 ; col < TotalTime; col++)

{

scheduleTime[row][col]=0;

}

}

}

void PrintSchedule (int\*\* scheduleTime, int TotalProcessors, int TotalTime, ostream& outputFile)

{

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

{

for (int col = 0 ; col < TotalTime; col++)

{

outputFile<<scheduleTime[row][col]<<" ";

}

outputFile<<endl;

}

}

int findProcess(int\* processJob, int ProcessorUsed)

{

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

{

if (processJob[row] <= 0)

{

return row;

}

}

return -1;

}

bool DG(int\* jobMarked, int NumberOfNodes1)

{

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

if (jobMarked[row] == 0) {

return true;

}

}

return false;

}

bool JobsFinished(int\* processJob, int TotalProcessors)

{

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

{

if (processJob[row] != 0){

return false;

}

}

return true;

}

void decreaseTime(int\* processTime, int TotalProcessors)

{

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

{

if (processTime[row] > 0)

{

processTime[row] -= 1;

}

}

}

int findFinishedJob(int\* processTime, int\* processJob, int TotalProcessors)

{

int finishedJob = 0;

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

{

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

{

finishedJob = processJob[row];

return finishedJob;

}

}

return finishedJob;

}

void deleteProcessJob(int\* processJob, int TotalProcessors, int finishedJob)

{

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

{

if (processJob[row] == finishedJob)

{

processJob[row] = 0;

}

}

}

void updateJobDone(int\* jobDone, int finishedJob)

{

if (finishedJob == 0)

{

return;

}

else jobDone[finishedJob - 1] = 1;

}

bool processJobisFull(int\* processJob, int TotalProcessors)

{

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

{

if (processJob[row] > 0)

{

return true;

}

}

return false;

}

bool jobDoneisNotFull(int\* jobDone, int NumberOfNodes1)

{

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

{

if (jobDone[row] != 1)

{

return false;

}

}

return true;

}

int main(int argc, char\* argv[])

{

//TWO INPUT FILES && ONE INPUT VALUE && ONE OUTPUTFILE

ifstream DependencyFile;

DependencyFile.open(argv[1]);

ifstream JobTimeFile;

JobTimeFile.open(argv[2]);

int TotalProcessors = stoi(argv[3]);

ofstream outputFile;

outputFile.open(argv[4]);

//====================================//

int NumberOfNodes1, NumberOfNodes2;

DependencyFile>>NumberOfNodes1;

JobTimeFile>>NumberOfNodes2;

if (NumberOfNodes1 != NumberOfNodes2)

{

outputFile<<"There is a error in number of nodes";

}

if (TotalProcessors > NumberOfNodes1)

{

TotalProcessors = NumberOfNodes1;

}

//============== STEP 0 =====================//

int\* jobTime;//1

int\* processJob;//2

int\* processTime;//3

int\* parentCount;//4

int\* jobMarked;//5

int\* jobDone;//6

int\*\* scheduleTime;//7

LinkedList openList;//8

LinkedList\* hashTable[NumberOfNodes1];//9

jobTime = new int[NumberOfNodes1];//1

jobMarked = new int[NumberOfNodes1];//2

jobDone = new int[NumberOfNodes1];//3

parentCount = new int[NumberOfNodes1];//4

processJob = new int[TotalProcessors];//5

processTime = new int[TotalProcessors];//6

openList = LinkedList();//7

int TotalTime = 0;

int NodeNumber, Time;

while (JobTimeFile>>NodeNumber and JobTimeFile>>Time)

{

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

{

if (row == NodeNumber-1)

{

jobTime[row] = Time;

TotalTime += Time;

}

}

}

scheduleTime = new int\*[TotalProcessors];//8

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

{

scheduleTime[row] = new int[TotalTime];

}

FillSchedule(scheduleTime, TotalProcessors, TotalTime);

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

{

hashTable[row] = new LinkedList();//9

}

int BeforeNode, AfterNode;

while (DependencyFile>>BeforeNode and DependencyFile>>AfterNode)

{

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;

int orphaneNode;

//======================= Step 1 ===========================

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 and jobMarked[row] == 0)

{

orphaneNode = row;

jobMarked[orphaneNode] = 1;

openList.addToList(orphaneNode+1);

}

}

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

while (openList.getSize() != 0 or 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() and DG(jobMarked, NumberOfNodes1) and JobsFinished(processJob, TotalProcessors))

{

outputFile<<"There is a cycle in the graph"<<endl;

return 0;

}

outputFile<<"OpenList"<<endl;

openList.printLinkedList(outputFile);

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

outputFile<<"Time: "<<time<<" ProcessorUsed "<<ProcessorUsed<<endl;

outputFile<<"Hash Table"<<endl;

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

{

outputFile<<row+1<<" ";hashTable[row]->printLinkedList(outputFile);

}

outputFile<<"Schedule"<<endl;

PrintSchedule(scheduleTime, TotalProcessors, TotalTime, outputFile);

outputFile<<"Job Time"<<endl;

PrintArray(jobTime, NumberOfNodes1, outputFile);

outputFile<<"Process Job"<<endl;

PrintArray(processJob, TotalProcessors, outputFile);

outputFile<<"Process Time"<<endl;

PrintArray(processTime, TotalProcessors, outputFile);

outputFile<<"Parent Count"<<endl;

PrintArray(parentCount, NumberOfNodes1, outputFile);

outputFile<<"Job Marked"<<endl;

PrintArray(jobMarked, NumberOfNodes1, outputFile);

outputFile<<"Job Done"<<endl;

PrintArray(jobDone, NumberOfNodes1, outputFile);

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

time++;

decreaseTime(processTime, TotalProcessors);

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

while ( processJobisFull(processJob, TotalProcessors) )//<--- needs work on

{

int finishedJob = findFinishedJob(processTime, processJob,

TotalProcessors);

if (finishedJob == 0)

{

break;

}

outputFile<<"\*\*\*\*\*"<<finishedJob<<"\*\*\*\*\*\*"<<endl;

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);

//cout<<endl<<endl;

}

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

outputFile<<"Time: "<<time<<" ProcessorUsed "<<ProcessorUsed<<endl;

outputFile<<"Job Time"<<endl;

PrintArray(jobTime, NumberOfNodes1, outputFile);

outputFile<<"Process Job"<<endl;

PrintArray(processJob, TotalProcessors, outputFile);

outputFile<<"Process Time"<<endl;

PrintArray(processTime, TotalProcessors, outputFile);

outputFile<<"Parent Count"<<endl;

PrintArray(parentCount, NumberOfNodes1, outputFile);

outputFile<<"Job Marked"<<endl;

PrintArray(jobMarked, NumberOfNodes1, outputFile);

outputFile<<"Job Done"<<endl;

PrintArray(jobDone, NumberOfNodes1, outputFile);

if (jobDoneisNotFull(jobDone, NumberOfNodes1)) {

break;

}

}

// cout<<"Schedule"<<endl;

PrintSchedule(scheduleTime, TotalProcessors, TotalTime, outputFile);

delete[] jobTime;

delete[] jobMarked;

delete[] jobDone;

delete[] processJob;

delete[] processTime;

delete[] parentCount;

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

delete[] scheduleTime[row];

}

}

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