**CSC 323 Project 6 (C++)**

Tyler Gaugler

Due Date: 12/6/2016

Algorithm Steps:

step 0: - open input-1 (argv[1]) and input-2 (argv[2]) and output (argv[4])

- numJobs <-- get from input1.

- numProcessor <-- get from argv[3]

- if numProcessor > numJobs

numProcessor <-- numJobs

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

- Time <-- 1

Step 1: graphHashTable <-- dynamically allocated and initallized of all fields

step 2: <ni, nj> <-- read from input-1

job <-- ni

child <-- nj

newNode <-- create a graphNode for <child>

push newNode on the top of graphHashTable[index].stackTop

graphHashTable[job].childCount ++

graphHashTable[child].fatherCount ++

Step 3: repeat step 2 until input-1 is end of file

step 4: totalJobTime <-- 0

step 5: <job, jobTime> <-- read from input-2

graphHashTable[job].jobTime <-- jobTime

totalJobTime += jobTime

step 6: repeat step 5 until input-2 is end of file

Step 7: processorSchedule <-- dynamically allocated and initallized

// The dimension is (numProcessors + 1) by (totalJobTimes + 1)

// prcessorSchedule[i][0],(1st column) to be used as availablity of processor[i]

// prcessorSchedule[i][0] initially set to 0

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

// if prcessorSchedule[i][0] > 0, means processor[i] is currently busy, not available.

// i prcessorSchedule[i][Time] == j, means processor[i] is busy on job j.

step 8: job <-- 1

step 9: if graphHashTable[job].fatherCount == 0

orphen <-- job

newNode <-- create a graphNode for <orphen>

insert(OPEN, newNode) // Open is ordered by the jobTime,

// if two jobs with the same jobTime, job with more childCount will be in front

// of the job with fewer childCount. jobTime is stored at graphHashTable[job].jobTime

graphHashTable[orphen].fatherCount -- // is -1 now, job is now on the Open

step 10: job++

step 11: repeat step 9 - step 10 while job <= numJobs

step 12: processor <-- 1

step 13: if OPEN is not empty and

if processorSchedule[processor][0] <= 0 // processor is not busy, available

availProc <-- processor

processorSchedule[availProc][0]++ // set to busy, un-available

job <-- remove from Open

jobTime <-- graphHashTable[job].jobTime

step 14: slot <-- Time

step 15: processorSchedule[availProc][slot] <-- job

step 16: slot ++

step 17: repeat step 15 to step 16 while slot <= Time + jobTime

step 18: processor ++

step 19: repeat step 13 - step 18 while OPEN is \*NOT empty\* and (processor <= numProcessor)

step 20: Time++

step 21: processor <-- 1

step 22: if (processorSchedule[processor][0] > 0) // was busy on a job

and (processorSchedule[processor][Time] <= 0) // current job is done

doneJob <-- processorSchedule[processor][Time - 1]

graphHashTable[doneJob].fatherCount-- // is equal to -2 now, job is done

processorSchedule[processor][0] <-- 0 // becomes available

step 23: processor ++

step 24: repeat step 22 to 23 while processor <= numProcessor

step 25: repeat which steps until what condition? (On your own)

Source Code:

#include <iostream>

#include <fstream>

#include <string>

#include <stdlib.h>

#include <math.h>

#include <iomanip>

using namespace std;

ifstream inputFile1;

ifstream inputFile2;

ofstream outputFile1;

class graphNode{

friend class hashNode;

public:

int jobId;

graphNode\* next ;

graphNode(int id);

graphNode();

void insert(graphNode\* newNode);

void print(graphNode\* current);

int popJob(graphNode\* nodeToBeRemoved);

};

graphNode::graphNode(int id){

jobId = id;

next= NULL;

}

graphNode::graphNode(){

jobId=0;

next=NULL;

}

void graphNode::insert(graphNode\* newNode){

newNode->next= this->next;

this->next= newNode;

}

void graphNode::print(graphNode\* current){

while(current){

cout<<"job("<< current->jobId<<")-> ";

if(current->next) current=current->next;

else break;

}

cout<<endl;

}

int graphNode::popJob(graphNode\* nodeToBeRemoved){

graphNode\* temp = nodeToBeRemoved->next;

this->next= temp;

return nodeToBeRemoved->jobId;

}

class hashNode{

public:

int jobTime;

int fatherCount;

int childCount;

graphNode\* stackTop;

hashNode();

void push(graphNode\* child);

bool hasNodes(hashNode\* graphHashTable, int numJobs);

void removeKids(hashNode\* graphHashTable, int jobDone);

};

hashNode::hashNode(){

jobTime=0;

fatherCount=0;

childCount=0;

stackTop= NULL;

}

void hashNode::push(graphNode\* child){

if(this->stackTop == NULL){

this->stackTop = child;

}

else{

graphNode\* temp = this->stackTop;

this->stackTop = child;

child->next = temp;

}

}

bool hashNode::hasNodes(hashNode\* graphHashTable, int numJobs){

int count=0;

int index=0;

for(int i=1; i <= numJobs; i++){

if(graphHashTable[i].fatherCount > -2){

count++;

index= i;

}

}

if(count==1) graphHashTable[index].fatherCount--;

if(count > 0) return true;

return false;

}

void hashNode::removeKids(hashNode\* graphHashTable, int jobDone){

graphNode\* current= graphHashTable[jobDone].stackTop;

while(current){

graphHashTable[current->jobId].fatherCount--;

current= current->next;

}

}

void insert(graphNode\* open\_queue, graphNode\* newNode, int newNodeindex, hashNode\* graphHashTable);

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

int numJobs=0;

int numProcessors= 1;

if(argv[3])numProcessors= stoi(argv[3]);

int totalJobTime=0;

int time=1;

int\*\* processorSchedule;

hashNode\* graphHashTable;

graphNode\* OPEN = new graphNode();

int index=0;

inputFile1.open(argv[1]);

if(inputFile1.is\_open()){

string data;

while(inputFile1 >> data ){

if(index==0){

numJobs = stoi(data);

if(numJobs < numProcessors) numProcessors =numJobs;

graphHashTable = new hashNode[numJobs+1];

for(int i=1;i<numJobs+1;i++){

graphHashTable[i] = hashNode();

}

OPEN = new graphNode[numJobs+1];

}

else{

int job= stoi(data);

inputFile1 >> data;

int child= stoi(data);

graphNode\* newNode= new graphNode(child);

graphHashTable[job].push(newNode);

graphHashTable[job].childCount++;

graphHashTable[child].fatherCount++;

}

index++;

}

time=1;

}

inputFile2.open(argv[2]);

if(inputFile2.is\_open()){

string data;

inputFile2 >>data;

while(inputFile2 >> data){

int job= stoi(data);

inputFile2 >> data;

int jobTime = stoi(data);

totalJobTime+=jobTime;

graphHashTable[job].jobTime= jobTime;

}

}

// instantiate processor schedule

processorSchedule = new int\*[numProcessors+1];

for(int i=0;i<numProcessors+1;i++){

processorSchedule[i] = new int[totalJobTime+1];

}

for(int r=0;r<numProcessors+1;r++){

for(int c=0;c<totalJobTime+1;c++){

processorSchedule[r][c] = 0;

}

}

int count=0;

while(graphHashTable->hasNodes(graphHashTable, numJobs)){

//get orphans

int job = 1;

int orphan = 0;

while(job <= numJobs){

if(graphHashTable[job].fatherCount == 0){

cout<<"job: "<<job<<" is an orphan"<<endl;

orphan = job;

graphNode\* orphanNode = new graphNode(orphan);

insert(OPEN, orphanNode, job, graphHashTable);

OPEN->print(OPEN);

graphHashTable[orphan].fatherCount--;

// cout<<" job has "

}

job++;

}

OPEN->print(OPEN);

// put on processor

int processor= 1;

int availableProcessor=0;

int jobToPutOn=0;

int jobTime=0;

int jobDone=0;

while( OPEN->next && (processor <= numProcessors) ){

if(OPEN->next){

if(processorSchedule[processor][0] <= 0){

availableProcessor = processor;

processorSchedule[availableProcessor][0]++;

jobToPutOn = OPEN->popJob(OPEN->next);

cout<<"job to be put on: "<<jobToPutOn<<endl;

OPEN->print(OPEN);

jobTime = graphHashTable[jobToPutOn].jobTime;

}

}

int slot = time;

while (slot < time + jobTime){

processorSchedule[availableProcessor][slot] = jobToPutOn;

slot++;

}

processor++;

}

time++;

processor=1;

while (processor <= numProcessors){

if ((processorSchedule[processor][0] > 0) && (processorSchedule[processor][time] <= 0) ){

jobDone= processorSchedule[processor][time-1];

graphHashTable[jobDone].fatherCount--;

graphHashTable->removeKids(graphHashTable, jobDone);

processorSchedule[processor][0] = 0;

cout<<"job "<<jobDone<<" just finished"<<endl<<endl;

}

processor++;

}

for(int r=1;r<numProcessors+1;r++){

for(int c=1;c<totalJobTime+1;c++){

cout<< processorSchedule[r][c]<<" ";

}

cout<<endl;

}

for(int i=1;i<numJobs+1;i++){

cout<<"job "<<i<<" has "<<graphHashTable[i].fatherCount<<" fathers"<<endl;

}

cout<<endl;

count++;

}// while there are still nodes left

outputFile1.open(argv[4]);

if(outputFile1.is\_open()){

for(int r=0;r<numProcessors+1;r++){

for(int c=1;c<time;c++){

if(r==0) outputFile1<< setw(5)<< left <<"\_\_\_\_\_";

else if(c==1)outputFile1<<"p"<<r<<"|"<< setw(5)<< left << processorSchedule[r][c];

else outputFile1<< setw(5)<< left << processorSchedule[r][c];

}

outputFile1<<endl;

}

for(int r=0;r<time-1;r++) outputFile1<< setw(5)<< left <<"-----";

}

}

void insert(graphNode\* open\_queue, graphNode\* newNode, int newNodeindex, hashNode\* graphHashTable){

bool inserted = false;

if(open\_queue->next != NULL){

graphNode\* current= open\_queue;

while(current->next && !inserted){

if(graphHashTable[current->next->jobId].jobTime > graphHashTable[newNode->jobId].jobTime){

current->insert(newNode);

inserted=true;

}

else if(graphHashTable[current->next->jobId].jobTime == graphHashTable[newNode->jobId].jobTime){

if(graphHashTable[current->next->jobId].childCount < graphHashTable[newNode->jobId].childCount){

current->insert(newNode);

inserted = true;

}

else{

while(current->next && (graphHashTable[current->next->jobId].jobTime == graphHashTable[newNode->jobId].jobTime) && !inserted){

if(graphHashTable[current->next->jobId].childCount <= graphHashTable[newNode->jobId].childCount){

current->insert(newNode);

inserted= true;

}

if(current->next && !inserted){

current = current->next;

if(current->next == NULL){

current->insert(newNode);

inserted= true;

}

}

else break;

} // while loop for number of kids spot

}

} // if they have the same jobTime

if(current->next && !inserted){

current = current->next;

}

if( current->next == NULL){

current->insert(newNode);

inserted=true;

}

// else break;

} // outer while loop

}

else{

open\_queue->next = newNode;

}

}