Departamento de Informática

Análise e Desenho de Algoritmos

2014/2015

**Relatório**  
Trabalho Prático 2

Hard Weeks

Docente:

Margarida Mamede

Turno prático 4

José Pedro Soares Nº 42016

Ricardo Korn Moreira Nº 44568

Introdução

O problema do Hard Weeks consiste no cálculo do número máximo de tarefas a ser realizadas em apenas uma semana e o número de semanas dificeis, dado um conjunto de informação de forma a possibilitar o mesmo calculo.   
 Restrições essas que consistem no número de tarefas, número de precedências entre tarefas e o limite usado como definição de semana dificil.

Neste problema as tarefas apenas são executadas apenas depois de todos os seus antecessores estarem concluídos, daí a necessidade das verificações apresentadas mais a frente.

Resolução do Problema

Para a resolução deste problema, utilizamos o algoritmo apresentado nas aulas teóricas de ordenação topológica mas modificado para se adequar ao problema a resolver.

Modificações ao algoritmo Todos os vetores no vetor 'ready' são tratados de uma só vez com auxilio de um vetor auxiliar 'inQueue', isto porque várias 'tasks' podem estar a correr de forma concorrente. Desta forma a cada iteração do vetor 'ready' podem ser verificadas as condições de 'hard week' e de maior número de tasks por semana.

Implementação do Algoritmo

A solução do problema Hard Weeks foi implementada através do algoritmo de ordenação topológica. Para a implementação deste algoritmo criamos apenas três classes, a Vertex, a HardWeeks e a Main.

A classe Vertex é apenas uma classe para guardar a informação sobre um vertex, como tal tem a informação relativa ao número de vertex, o número de antecessores e o número de sucessores.

A classe HardWeeks é uma classe que vai realizar todo o tipo de cálculos associados ao resultado final ficando os resultados guardados num vector de inteiros que apenas irá ter o número máximo de tarefas numa semana e o número de semanas difíceis. Esta classe tem apenas uma função, “**int**[] sort()”. Esta função é a mais importante do trabalho visto que é ela que vai permitir o calculo dos resultados finais.

A função vai ter um primeiro for, que vai utilizar list uma chamada “ready”, e um vector “inCounter”, estes vão permitir guardar as tarefas que estão prontas a executar, e número de antecessores dos respectivos vertex. Vamos então correr todo o grafo de forma a guardar todos os antecessores no “inCounter” e a adicionar todos os que estão prontos a executar na lista ready. O primeiro if verifica para todos os vertex que tenham sido tratados na main, o else vai tratar do restantes casos(ou seja dos gravos que não se encontram na lista dos sucessores dada como input). De seguida temos um while em que executamos sempre que ainda se existir algum número a tratar na lista “ready”, alteramos os valores do número máximo de semanas e o número de semanas difícies, isto caso as verificações se confirmem. Por fim temos dois ciclos, um que serve para correr o “inQueue”(cópia do “ready”) e um para iterar todos os sucessores do mesmo vertex. Vamos então a cada um dos seus sucessores verificar se ainda contém antecessores por tratar, caso não tenha o vertex é adicionado ao ready.

Tratamos assim todos os casos de forma a guardar sempre o número máximo de tarefas por semana e o número de semanas difícies.

Análise do Algoritmo

Complexidade Espacial

Na nossa solução utilizamos uma *ArrayList* para guardar o número de vértices a ser iterados a seguir.

List<Vertex> ready = **new** ArrayList<>(**numTasks**);

A complexidade espacial deste vetor é de *numTasks*, ou seja, o número de vértices do problema.

É também utilizado um *ArrayList* auxiliar

List<Vertex> inQueue;

cuja complexidade é equivalente ao *ready.* Além destes dois, existe um array de inteiros para guardar a informação sobre os *in degrees* dos vértices.

**int**[] inCounter = **new int**[**numTasks**];

A complexidade espacial deste vetor é do número de vértices. Por fim temos um vetor para guardar o grafo todo.

Vertex[] vertexes = **new** Vertex[numTasks];

Complexidade Temporal

No nosso algoritmo temos vários ciclos:

**for**(**int** j = 0; j < **graph**.**length**; j++)

cuja complexidade temporal é de O(V) em que V: número de vértices

**for** (Vertex vertex : inQueue) {

cuja complexidade temporal é de O(A) em que A: número de arcos

**while**(!ready.isEmpty())

em que ready ≤ nº de vértices.

Conclusões

Em suma o nosso trabalho foi realizado com bastante deliberação em termos de observação de exemplos de grafos, utilizando assim os mesmos para a abordagem ao problema e sua resolução. No entanto tivemos apenas uma dificuldade visto que nos esquecemos de tratar os vertex que se encontravam sem sucessores e sem antecessores, faltando assim tratar esse caso.

Contudo através das aulas teóricas e do trabalho chegamos a uma conclusão que nos permitiu obter bons resultados.

Anexo

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.util.ArrayList;

**import** java.util.List;

/\*\*

\* Ricardo Korn Moreira - MIEI 44568

\* Jose Pedro Soares - MIEI 42016

\*/

**public** **class** Main {

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

BufferedReader in = **new** BufferedReader(**new** InputStreamReader(System.***in***));

String[] line;

line = in.readLine().split(" ");

**int** numTasks = Integer.*parseInt*(line[0]);

**int** numPrecedents = Integer.*parseInt*(line[1]);

Vertex[] vertexes = **new** Vertex[numTasks];

**int** hardLimit = Integer.*parseInt*(line[2]);

**for**(**int** i = 0; i < numPrecedents; i++){

line = in.readLine().split(" ");

**int** first = Integer.*parseInt*(line[0]);

**int** second = Integer.*parseInt*(line[1]);

**if**(vertexes[first] == **null**) vertexes[first] = **new** Vertex(first);

**if**(vertexes[second] == **null**) vertexes[second] = **new** Vertex(second);

vertexes[second].addInDegree(1);

vertexes[first].addOutDegree(vertexes[second]);

}

HardWeeks calc = **new** HardWeeks(numTasks, numPrecedents, hardLimit, vertexes);

**int**[] result = calc.sort();

System.***out***.println(result[0]+" "+result[1]);

}

}

**class** Vertex {

**int** number;

Vertex[] outVertexes;

**int** counter;

**int** inDegree;

**public** Vertex(**int** number) {

**this**.number = number;

**this**.counter = 0;

**this**.outVertexes = **new** Vertex[5000];

}

**public** Vertex[] getOutVertexes() {

**return** outVertexes;

}

**public** **int** getNumber() {

**return** number;

}

**public** **int** getInDegree() {

**return** inDegree;

}

**public** **void** addInDegree(**int** inDegree) {

**this**.inDegree += inDegree;

}

**public** **void** addOutDegree(Vertex vertex){

**this**.outVertexes[counter] = vertex;

counter++;

}

}

**class** HardWeeks {

**int** numTasks;

**int** numPrecedents;

**int** hardLimit;

Vertex[] graph;

**public** HardWeeks(**int** numTasks, **int** numPrecedents, **int** hardLimit, Vertex[] graph) {

**this**.numTasks = numTasks;

**this**.numPrecedents = numPrecedents;

**this**.hardLimit = hardLimit;

**this**.graph = graph;

}

**int**[] sort(){

**int** maxTasksWeek = 0;

**int** hardWeeks = 0;

List<Vertex> ready = **new** ArrayList<>(numTasks);

List<Vertex> inQueue;

**int**[] inCounter = **new** **int**[numTasks];

**for**(**int** j = 0; j < graph.length; j++){

Vertex v = graph[j];

**if**(v != **null**) {

inCounter[v.getNumber()] = v.getInDegree();

**if** (inCounter[v.getNumber()] == 0) ready.add(v);

}

**else**{

ready.add(**new** Vertex(j));

}

}

**while**(!ready.isEmpty())

{

**if**(ready.size() > maxTasksWeek) maxTasksWeek = ready.size();

**if**(ready.size() > hardLimit) hardWeeks++;

inQueue = **new** ArrayList<>(ready);

ready.clear();

**for** (Vertex vertex : inQueue) {

Vertex[] outVertexes = vertex.getOutVertexes();

Vertex v = outVertexes[0];

**int** count = 0;

**while** (v != **null**) {

inCounter[v.getNumber()]--;

**if** (inCounter[v.getNumber()] == 0) ready.add(v);

v = outVertexes[++count];

}

}

}

**return** **new** **int**[]{maxTasksWeek, hardWeeks};

}

}