Departamento de Informática
Estruturas de Dados e Algoritmos II
Ano Letivo de 2020/2021

Hard Weeks

Trabalho por:
Miguel Rodrigues - 45424
André Rato - 45517

Docente:
Vasco Pedro
Mooshak: g208

Índice

1. Introdução 3
2. Análise do Problema 3
3. Estruturas Utilizadas 4
4. Algoritmo 4
5. Código 6
6. Complexidade Temporal 8
7. Complexidade Espacial 9
8. Bibliografia 10

1. Introdução

Este relatório é referente ao problema H, de nome “Hard Weeks”.

O problema consiste em organizar, em semanas de trabalho, um conjunto de tarefas, de modo a executá-las apenas quando todas as tarefas de que dependerem terem sido executadas. Após essa organização, é pedido que seja determinado o número máximo de tarefas a ser realizado numa semana e quantas semanas são consideradas *hard weeks*. Para que uma semana seja considerada *hard week* é necessário que o número de tarefas executadas nessa semana seja maior que o limite definido no *input*.

2. Análise do Problema

Para a realização deste trabalho foi utilizado um grafo orientado não pesado, implementado através de um *array* de vértices (ou nós). As relações de dependência entre as várias tarefas são representadas através das arestas (ou arcos) do grafo, em que o vértice de destino da aresta representa a tarefa que depende do vértice de origem da mesma.

O grafo que representa o problema é construído a partir de um *input* (Figura 1). Após a construção do grafo, é necessário definir-se a ordem de execução das tarefas, semelhante à ordenação topológica de um grafo (Figura 2).

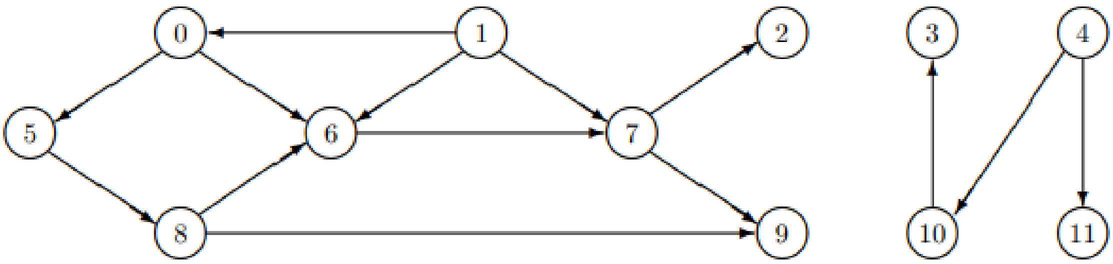

Figura 1 - Exemplo de um grafo representativo do problema

Weeks	1st	2nd	3rd	4th	5th	6th	7th
Tasks	1	0	5	8	6	7	2
	4	10	3				9
		11					

Figura 2 - Distribuição das tarefas por semanas

Tendo em conta a distribuição das tarefas pelas semanas, é possível determinar o número máximo de tarefas a ser realizado numa semana e quantas semanas são consideradas *hard weeks*.

3. Estruturas Utilizadas

Foi implementada uma classe designada por *Vertex*, responsável por representar os vértices do grafo. A classe possuí três variáveis de classe:

- int number: número da tarefa;
- ArrayList<Vertex> edges: lista com todos os vértices adjacentes do vértice;
- int i: inteiro que representa o número de dependências relativas desse vértice.

Para além da classe *Vertex*, também se implementou a classe *Graph*, representativa de um grafo. A classe tem as seguintes variáveis de classe e métodos:

- Vertex[] vertexes: *array* com todos os vértices que constituem o grafo;
- int size: número de vértices do grafo;
- void addAdj(int origin, int destination): método que adiciona uma aresta do vértice *origin* ao vértice *destination*.

No método que calcula o resultado do problema, recorreu-se à utilização de duas *Queues*, mais especificamente, duas *LinkedLists*, ambas utilizadas alternadamente para guardar as tarefas a realizar em cada semana:

- Queue<Graph.Vertex> S = new LinkedList<>();
- Queue<Graph.Vertex> P = new LinkedList<>().

4. Algoritmo

Com o objetivo de ordenar as tarefas, de modo a preservar todas as suas dependências e, por fim, calcular o número de *hard weeks* e o número máximo de tarefas por semana, modificou-se o algoritmo de ordenação topológica fornecido durante as aulas da disciplina, de modo a adequar-se ao problema em causa.

O algoritmo começa por percorrer os vértices do grafo e as suas arestas, atribuindo a cada vértice o número de arestas, das quais o vértice é o destino, guardando esse valor na variável *i* de cada vértice. Em seguida, é calculado o número de vértices que não têm arestas nesse termos, ou seja, em que o valor de *i* é 0, vértices esses que são associados à primeira semana, visto que correspondem às tarefas que podem ser associadas à mesma. Durante este cálculo, os vértices que seguem estas condições são adicionados a uma q*ueue* (*S*), modo a serem utilizados futuramente. Depois define-se o valor máximo e, caso o número de vértices nestas condições seja maior que o limite (*L*), o número de *hard weeks* é incrementado.

Após esta primeiro conjunto de operações, o algoritmo entra num ciclo, que terminará após as *queues* estarem vazias, ou seja, todos os vértices terem passado por, uma das *queues*. O bloco de operações a ser executado depende do valor de *queueChanger* (*true* ou *false*):

- caso o valor desta seja *true*, é retirado um vértice da *queue S*; após essa remoção, todos os vértices adjacentes desse mesmo vértice são avaliados e, a sua variável *i* decrementada e, se for 0, o vértice é adicionado à *queue P* e a variável *counter* (responsável pela contagem do número de tarefas correspondestes à semana) é incrementada; de seguida, e apenas se a *queue S* estiver vazia, o valor booleano de *queueChanger* é alterado, o máximo é atualizado, caso necessário, o número de *hard* *weeks* é incrementado caso se verifique a condição para tal e o contador (*counter*) é colocado a 0;

- caso o valor seja *false*, o processo é o mesmo, apenas trocando as *queues*, ou seja, os vértices são removidos da *queue P* e são adicionados à *queue S*.

Esta alternância entre as *queues* foi implementada para separar as tarefas de semanas consecutivas, deixando assim em cada *queue* as tarefas que serão executadas na mesma semana (Figura 3).

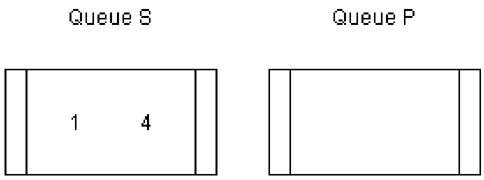

Figura 3.1 - Tarefas da primeira semana na queue S

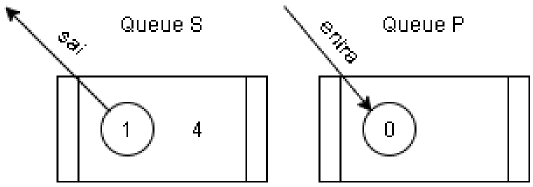

Figura 3.2 - Primeira tarefa da segunda semana na queue P

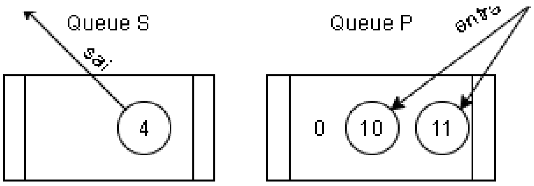

Figura 3.3 - Tarefas da segunda semana na queue P

Este algoritmo retorna um *array* de dimensão 2, em que a primeira posição contém o número máximo de tarefas realizadas numa semana, e a segunda contém o número total de *hard weeks*.

5. Código

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.*;

public class HardWeeks {
    public static class Graph {

        public static class Vertex {
            int number;
            ArrayList<Vertex> edges;
            int i;

            public Vertex(int number) {
                this.number = number;
                this.edges = new ArrayList<>();
                this.i = 0;
            }
        }

        Vertex[] vertexes;
        int size;

        public Graph(int nVertex) {
            this.vertexes = new Vertex[nVertex];
            this.size = nVertex;

            for (int i = 0; i < nVertex; i++)
                this.vertexes[i] = new Vertex(i);
        }

        void addAdj(int origin, int destination) {
            this.vertexes[origin].edges.add(this.vertexes[destination]);
        }
    }

    static int[] hardWeekSolver(Graph G, int L) {
        int counter = 0, nHardWeeks = 0, max;
        boolean queueChanger = true;

        for (Graph.Vertex u : G.vertexes)
            for (Graph.Vertex v : u.edges)
                v.i++;

        Queue<Graph.Vertex> S = new LinkedList<>();
        Queue<Graph.Vertex> P = new LinkedList<>();

        for (Graph.Vertex u : G.vertexes)
            if (u.i == 0) {
                S.add(u);
                counter++;
            }

        max = counter;

        if (counter > L)
            nHardWeeks++;

        counter = 0;

        while (!S.isEmpty() || !P.isEmpty()) {
            Graph.Vertex u;
            if (queueChanger) {
                u = S.poll();
                for (Graph.Vertex v : u.edges) {
                    v.i--;
                    if (v.i == 0) {
                        P.add(v);
```

```
                        counter++;
                    }
                }
                if (S.isEmpty()) {
                    queueChanger = false;
                    if (counter > max)
                        max = counter;
                    if (counter > L)
                        nHardWeeks++;
                    counter = 0;

                }
            } else {
                u = P.poll();
                for (Graph.Vertex v : u.edges) {
                    v.i--;
                    if (v.i == 0) {
                        S.add(v);
                        counter++;
                    }
                }
                if (P.isEmpty()) {
                    queueChanger = true;
                    if (counter > max)
                        max = counter;
                    if (counter > L)
                        nHardWeeks++;
                    counter = 0;
                }
            }
        }
        return new int[]{max, nHardWeeks};
    }

    public static void main(String[] args) throws IOException {
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

        String[] line = input.readLine().split(" ");

        int T = Integer.parseInt(line[0]);
        int P = Integer.parseInt(line[1]);
        int L = Integer.parseInt(line[2]);

        Graph G = new Graph(T);

        for (int i = 0; i < P; i++) {
            line = input.readLine().split(" ");
            G.addAdj(Integer.parseInt(line[0]), Integer.parseInt(line[1]));
        }

        int[] answer = hardWeekSolver(G, L);

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

        input.close()
    }
}
```

6. Complexidade Temporal

As afetações das linhas 2 e 3 têm todas complexidade temporal $O(1)$:

$$O(1) + O(1) + O(1) + O(1) = O(1)$$

O ciclo das linhas 5-7 é executado |V| + |E| vezes, tendo apenas uma instrução de complexidade $O(1)$, ou seja, a complexidade temporal do ciclo é $\theta(|V| + |E|)$, sendo |V| o número de vértices e |E| o número de arestas do grafo.

A criação das *queues* (linhas 9 e 10) têm complexidade temporal $O(1)$, logo:

$$O(1) + O(1) = O(1)$$

O ciclo das linhas 12-16 é executado |V| vezes; a comparação da linha 13 tem complexidade constante, assim como a inserção do elemento na *queue S* e a soma e afetação da variável *counter*, logo a complexidade temporal é $O(|V|)$:

$$O(1) \,.\, O(1) + O(1) = O(1)$$

$$O(1) \,.\, |V| = O(|V|)$$

As afetações, comparação e soma das linhas 18, 20, 21 e 23 têm complexidade temporal:

$$O(1) + O(1) \,.\, O(1) + O(1) = O(1)$$

Os ciclos das linhas 29-35 e 47-57 são executados alternadamente |E| vezes a cada iteração do ciclo principal *while* da linha 25 e as suas afetações, somas, comparações, remoções e acessos têm todas complexidade $O(1)$, logo a complexidade temporal dos ciclos é $O(|E|)$.

Os blocos condicionais das linhas 36-44 e 54-61 são executados alternadamente a cada iteração do ciclo principal *while*, tal como os ciclos *for* previamente referidos. Deste modo, conclui-se que a complexidade temporal destes dois blocos é $O(1)$.

Agregando os dois resultados obtidos anteriormente e sabendo que o ciclo é executado, no pior dos casos, |V| vezes, conclui-se que a complexidade temporal desse ciclo é $\theta(|V| + |E|)$.

Concluindo,

$$O(1) + \theta(|V| + |E|) + O(1) + O(|V|) + O(1) + \theta(|V| + |E|) = \theta(|V| + |E|).$$

A complexidade temporal do algoritmo utilizado é $\theta(|V| + |E|)$.

Relativamente à classe do grafo (*Graph*) podemos concluir que, no construtor da classe *Vertex*, o ciclo que cria os vários vértices do grafo, dado o tamanho, tem complexidade temporal $O(|V|)$. As restantes operações têm todas complexidade temporal $O(1)$.

Na *main* do programa, todas as operações têm complexidade temporal $O(1)$, exceto o ciclo *for*, que tem complexidade $O(P)$, sendo P o número de arestas (mesmo significado de |E|).

Conclui-se assim, que o programa tem complexidade temporal $\theta(|V| + |E|)$.

7. Complexidade Espacial

No grafo, cada vértice tem dois valores escalares, *number* e *i*, e tem uma lista de tamanho |E|, no pior dos casos (se o vértice for adjacente a todos os outros) logo, a complexidade espacial do array de vértices é $\theta(|V| + |E|)$.

No algoritmo que resolve o problema, existem duas filas (*queues*) que, no pior dos casos, todos os |V| elementos estão distribuídos pelas duas filas. Caso uma das filas tenha todos os elementos, a outra está vazia. Portanto, conclui-se que o algoritmo tem complexidade espacial $O(|V|)$.

Concluindo, determinou-se que o programa tem complexidade espacial $\theta(|V| + |E|)$.

8. Bibliografia

https://www.bigocheatsheet.com/ - consulta de complexidades tabeladas;
Slides 1-10 e 82-90 das aulas teóricas de EDA2, fornecidos pelo docente.