



UNIVERSIDADE  
DE ÉVORA

# **Relatório - 2º Trabalho - Estruturas de Dados e Algoritmos II**

**Trabalho realizado pelo grupo g226 (Mooshak) composto por:**

- Rui Roque nº42720
- Tomás Dias nº42784

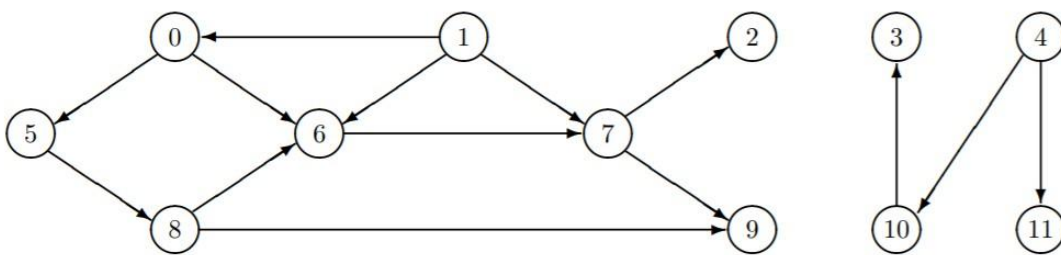
## Introdução ao problema

Este trabalho teve como objetivo implementar uma solução para o problema *Hard Weeks*.

Para tal, foi pedido um programa que dado o número de tarefas, as precedências entre elas, e o limite de tarefas a serem realizadas numa semana (caso esse limite seja ultrapassado, a semana é considerada uma *hard week*), fosse calculado o número máximo de tarefas a ser realizado numa única semana e o número de *hard weeks*.

## Construção e descrição do algoritmo

De forma a elaborar o algoritmo implementado, focou-se no grafo de exemplo apresentado no enunciado:



Recebendo a informação relativa ao grafo (vértices e arcos), o algoritmo começa por calcular o número de vértices antecessores de cada vértice do mesmo.

Utilizando os conceitos do algoritmo de **ordenação topológica**, é identificado quais os vértices do grafo que não têm antecessores, sendo estes posteriormente adicionados a uma **fila** (por ordem crescente). De forma a identificar quais as tarefas a serem realizadas em cada semana, cada vértice é etiquetado com o número da semana em que a tarefa que lhe corresponde é realizada. No exemplo acima esses vértices correspondem a 1 e 4 sendo que cada um destes tem associado o número 1 que corresponde à primeira semana.

Depois, é retirado o primeiro vértice da **fila** e é feita a verificação dos seus vértices adjacentes. Em cada vértice adjacente analisado é avaliado se este não tem outros vértices antecessores para além do vértice retirado da **fila** anteriormente. Se isso se verificar, esse vértice é etiquetado com o número da semana que lhe corresponde (número da semana do antecessor incrementado) e é adicionado à **fila**.

Após a verificação de todos os vértices adjacentes do vértice retirado da **fila** anteriormente, no caso em que o valor da etiqueta correspondente à semana do vértice seja igual ao valor da variável **week** (responsável por auxiliar a contagem de tarefas numa semana, inicializada com o valor 1), é incrementado o número de tarefas realizadas na semana em questão. Caso contrário, é avaliado se o número de tarefas calculado para a semana atual (determinada pela variável **week**) corresponde ao máximo calculado e se se verificar é guardado esse valor na variável **max\_num\_tasks**. Por fim, é verificado se o número de tarefas calculado para a semana atual é superior ao **limite**, e se sim, é incrementada a variável **hard\_weeks**. Após as verificações, a variável **week** é atualizada.

O processo descrito é repetido até a **fila** ficar vazia.

## Análise das complexidades temporal e espacial

Inicialmente, foi necessário guardar a informação sobre o grafo, pelo que foi criada uma lista de adjacências **adj**, pelo que a complexidade espacial é de  $\Theta(V+E)$ .

Por se tratar de uma adaptação direta do algoritmo (2) de ordenação topológica lecionado nas aulas teóricas, as complexidades temporal e espacial serão bastante idênticas às calculadas nas mesmas.

Sendo que **V** representa o número de vértices (variável **num\_tasks** no código abaixo) e **E** o número de arcos (variável **temp** que guarda a lista de adjacências de um vértice), a complexidade temporal do código seguinte é:

```
for(i = 0; i < num_tasks; i++)
{
    ArrayList<Integer> temp = adj.get(i);

    for (int node : temp)
        parents[node]++;
}

Queue<Task> queue = new LinkedList<>();

for(i = 0; i < num_tasks; i++)
{
    if(parents[i] == 0)
    {
        Task t = new Task();
        t.value = i;
        t.week = week;
        queue.add(t);
    }
}
```

Diagrama de complexidade temporal:

- $\Theta(V)$  para a primeira iteração do loop `for(i = 0; i < num_tasks; i++)` (acima).
- $\Theta(E)$  para o loop interno `for (int node : temp)` (ao lado).
- $\Theta(V)$  para a segunda iteração do loop `for(i = 0; i < num_tasks; i++)` (abaixo).

Ou seja,  $\Theta(V+E) + \Theta(V)$ .

**Nota:** A condição *if* é executada no máximo **V** vezes, podendo este valor ser menor dependendo do grafo.

Já para o troço de código seguinte, como o ciclo *while* é executado uma vez para cada vértice, sendo que **V** representa o número de vértices e **E** o número de arcos, a complexidade temporal é:


```
while (!queue.isEmpty())
{
    Task u = queue.poll();

    for(int node : adj.get(u.value))
    {
        if (--parents[node] == 0)
        {
            Task v = new Task();
            v.value = node;
            v.week = u.week + 1;
            queue.add(v);
        }
    }

    if(u.week == week)
        nt++;
    else
    {
        if(nt > max_num_tasks)
            max_num_tasks = nt;

        if(nt > limit)
            hard_weeks++;

        nt = 1;
        week++;
    }
}
```



Ou seja,  **$\Theta(V+E)$**

**Nota:** A instrução `queue.add` (representada na primeira e segunda imagem) é executada **V** vezes em conjunto.

Portanto a complexidade temporal total é dada por:

$$\Theta(V+E) + \Theta(V) + \Theta(V+E) = \Theta(V+E)$$

Quanto à complexidade espacial, é utilizada uma fila para guardar todos os vértices do grafo, sendo dada por:  **$\Theta(V)$**

Adicionando a complexidade espacial resultante da criação de ***adj***, a complexidade espacial total dada por:

$$\Theta(V+E) + \Theta(V) = \Theta(V+E)$$