FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

UNIVERSIDADE NOVA DE LISBOA
OMNIS CIVITAS CONTRA SE DIVISA NON STABIT

Engenharia Informática

Análise e Desenho de Algoritmos

Trabalho Prático 2

Everything under Control

Trabalho realizado por:

Rafael Gameiro nº50677

Rui Santos nº50833

Índice

Complexidade Espacial

Vetor de Listas Ligadas de Adjacências de Sucessores (*nodesSuc*):	$\Theta(\|V\| + \|A\|)$
Vetor de Listas Ligadas de Adjacências de Antecessores (*nodesAnt*):	$\Theta(\|V\| + \|A\|)$
Lista de nós do Grafo (*nodesGraph*):	$\Theta(\|V\|)$
Lista de tarefas adiadas (*delays*):	$\Theta(\|A\|)$
Queue *ready*:	$\Theta(\|V\|)$
Array *inDegree*:	$\Theta(\|V\|)$
Array *maxLength*:	$\Theta(\|V\|)$
Total:	$\Theta(\|V\| + \|A\|)$

Complexidade Temporal

Classe Main:

Preenchimento do *nodesSuc* e do *nodesAnt* $\Theta(|A|)$

Criação dos nós e a inserção destes no objeto *Graph* $\Theta(|V|)$

Classe UnderControl:

Preenchimento do vetor *inDegree* $\Theta(|V|)$

Execução do algoritmo de Ordenação Topológica

Ciclo (executado |V| - 1 vezes) $\Theta(|V|)$

Ciclo (executado |V| - 1 vezes) $O(|V|)$

Exploração de todos os sucessores do grafo e
diminuição do número de antecessores de cada nó explorado

método exploreDelays():

Cálculo do custo máximo para chegar ao nó *current* a partir do nó *antecessor*

$$O(|V| - 1) = O(|V|)$$

Ciclo (executado no máximo |A| - 1 vezes)

Cálculo do custo máximo com os restantes antecessores
do nó current sendo feita a comparação
com o último custo máximo calculado. $O\big((|V| - 1) * (|V| - 1)\big) = O(|V|^2)$

Ordenação da lista delays $\Theta(n \log n)$
sendo n = $\text{delays.size}()$

Total: $O(|V|^2)$

Conclusão

Concluímos que a nossa versão do algoritmo não é a mais otimizada. Tentámos implementar uma versão mais otimizada, mas sem sucesso. Atualmente o nosso código percorre todos os antecessores de um vértice para ver o tamanho máximo do caminho até este vértice.

A versão otimizada iria a cada sucessor do vértice e calculava o custo máximo para chegar a esse sucessor. Com esta versão otimizada, conseguia-se reduzir o número de iterações dos arcos, e assim diminuir a complexidade temporal do algoritmo.

Anexos

```java
import java.io.BufferedReader;

/**
 *
 * @author Rafael Gameiro (50677)
 * @author Rui Santos (50833)
 *
 */
public class Main {

    @SuppressWarnings("unchecked")
    public static void main(String[] args) throws IOException {

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

        String inputNT = br.readLine();
        String[] inputNTArray = inputNT.split(" ");
        int nodes = Integer.parseInt(inputNTArray[0]);
        int tasks = Integer.parseInt(inputNTArray[1]);

        List<Edge>[] nodesSuc = new List[nodes];
        for (int i = 0; i < nodes; i++) {
            nodesSuc[i] = new LinkedList<>();
        }

        List<Integer>[] nodesAnt = new List[nodes];
        for (int i = 0; i < nodes; i++) {
            nodesAnt[i] = new LinkedList<>();
        }

        List<Node> nodesGraph = new ArrayList<Node>(nodes);

        int counter = 0;
        while (counter < tasks) {
            String input = br.readLine();
            String[] inputArray = input.split(" ");
            int nodeInicial = Integer.parseInt(inputArray[0]);
            int nodeFinal = Integer.parseInt(inputArray[1]);
            int duration = Integer.parseInt(inputArray[2]);

            Edge e = new Edge(nodeInicial - 1, nodeFinal - 1, duration);

            nodesSuc[nodeInicial - 1].add(e);
            nodesAnt[nodeFinal - 1].add(nodeInicial - 1);

            counter++;
        }
```

```java
        int makingGraph = 0;
        while (makingGraph < nodes) {
            Node n = new Node(makingGraph, nodesSuc[makingGraph], nodesAnt[makingGraph]);
            nodesGraph.add(makingGraph, n);
            makingGraph++;
        }

        Graph g = new Graph(nodesGraph);

        UnderControl uc = new UnderControl(g);
        List<Edge> delays = uc.ordTopologic();

        System.out.println(delays.size());
        for (Edge edge : delays) {
            System.out.println((edge.returnStartNode() + 1) + " " + (edge.returnEndNode() + 1));
        }
    }

}
```

```java
import java.util.Comparator;

/**
 *
 * @author Rafael Gameiro (50677)
 * @author Rui Santos (50833)
 *
 */
public class UnderControl {

    private Graph graph;

    public UnderControl(Graph graph) {
        this.graph = graph;
    }

    /**
     * Executes the topological sort algorithm.
     * Before inserting the next node to be processed in the ready queue,
     * the method exploreDelays will compute the maximum path cost to reach that node.
     */
    public List<Edge> ordTopologic() {

        Comparator<Edge> comparatorEdges = new EdgeComparator();
        List<Edge> delays = new LinkedList<>();

        Queue<Node> ready = new LinkedList<>();

        int[] inDegree = new int[graph.numNodesInGraph()];
        int[] maxLength = new int[graph.numNodesInGraph()];

        for (Node node : graph.returnNodes()) {
            int pos = node.getState();
            inDegree[pos] = node.numAnt();
            if (inDegree[pos] == 0)
                ready.add(node);
        }

        do {
            Node node = ready.remove();
            for (Edge e : node.returnSuc()) {
                int pos = e.returnEndNode();
                inDegree[pos]--;
                if (inDegree[pos] == 0) {
                    Node n = graph.returnNode(pos);
                    exploreDelays(n, node, maxLength, delays);
                    ready.add(n);
                }

            }

        } while (!ready.isEmpty());

        delays.sort(comparatorEdges);
        return delays;
    }
```

```java

    /***
     * Computes the maximum path cost to reach the Node current,
     * through the sum of the maximum path cost to reach Node antecessor plus the cost of the edge (antecessor -> current).
     * If the current node has more than one antecessor, it will go through all its antecessors and compute the maximum path.
     * The biggest cost will be the final maximum cost of the node,
     * while the other edges formed by the node and the other antecessors will be inserted in the delay queue.
     *
     * @param current node that is going to be processed
     * @param antecessor the previous node before the current node
     * @param maxLength array with all the maximum path costs to reach each node
     * @param delays list with all the edges that can be delayed
     */
    private void exploreDelays(Node current, Node antecessor, int[] maxLength, List<Edge> delays) {

        for (Edge edge : antecessor.returnSuc()) {
            if (edge.returnEndNode() != current.getState())
                continue;

            int nlength = maxLength[antecessor.getState()] + edge.returnDuration();
            maxLength[current.getState()] = nlength;
        }

        for (int ant : current.returnAnt()) {
            Node node = graph.returnNode(ant);
            for (Edge edge : node.returnSuc()) {
                if (edge.returnEndNode() != current.getState())
                    continue;

                int nlength = maxLength[ant] + edge.returnDuration();

                if (maxLength[current.getState()] < nlength) {
                    maxLength[current.getState()] = nlength;
                } else if (maxLength[current.getState()] > nlength) {
                    delays.add(edge);
                }

            }
        }

    }

}
```

```java
import java.util.List;

/**
 *
 * @author Rafael Gameiro (50677)
 * @author Rui Santos (50833)
 *
 */
public class Graph {

    private List<Node> nodesGraph;

    public Graph(List<Node> nodesGraph) {
        this.nodesGraph = nodesGraph;
    }

    /***
     *
     * @return the nodes that represent the Graph
     */
    public List<Node> returnNodes() {
        return nodesGraph;
    }

    /***
     *
     * @param position
     * @return a specific node from the graph
     */
    public Node returnNode(int position) {
        return nodesGraph.get(position);
    }

    /***
     *
     * @return the number of nodes that represent the Graph
     */
    public int numNodesInGraph() {
        return nodesGraph.size();
    }
}
```

```java
import java.util.List;

/**
 *
 * @author Rafael Gameiro (50677)
 * @author Rui Santos (50833)
 *
 */
public class Node {

    private int state;
    private List<Edge> successors;
    private List<Integer> antecessors;

    public Node(int state, List<Edge> successors, List<Integer> antecessors) {
        this.state = state;
        this.successors = successors;
        this.antecessors = antecessors;
    }

    /**
     *
     * @return the state (number) of the respective node
     */
    public int getState() {
        return state;
    }

    /***
     *
     * @return the successors of the respective node
     */
    public List<Edge> returnSuc() {
        return successors;
    }

    /***
     *
     * @return the antecessors of the respective node
     */
    public List<Integer> returnAnt() {
        return antecessors;
    }

    /***
     *
     * @return the number of antecessors of the respective node
     */
    public int numAnt() {
        return antecessors.size();
    }
}

```

```java

/**
 *
 * @author Rafael Gameiro (50677)
 * @author Rui Santos (50833)
 *
 */
public class Edge {

    private int startNode;
    private int endNode;
    private int duration;

    public Edge(int startNode, int endNode, int duration) {
        this.startNode = startNode;
        this.endNode = endNode;
        this.duration = duration;
    }

    /***
     *
     * @return the startNode of the respective Edge
     */
    public int returnStartNode() {
        return startNode;
    }

    /***
     *
     * @return the endNode of the respective Edge
     */
    public int returnEndNode() {
        return endNode;
    }

    /***
     *
     * @return the duration (cost) of the respective Edge
     */
    public int returnDuration() {
        return duration;
    }
}
```

```java
import java.util.Comparator;

/**
 *
 * @author Rafael Gameiro (50677)
 * @author Rui Santos (50833)
 *
 */
public class EdgeComparator implements Comparator<Edge> {

    @Override
    public int compare(Edge e1, Edge e2) {

        if (e1.returnStartNode() == e2.returnStartNode()) {
            if (e1.returnEndNode() < e2.returnEndNode())
                return -1;
            else
                return 1;
        } else {
            if (e1.returnStartNode() < e2.returnStartNode())
                return -1;
            else
                return 1;
        }
    }

}
```