

## Análise e Desenho de Algoritmos 2016/2017

### Administrative Reform

### Trabalho prático nº3

Docente Margarida Mamede

- Filipe Cabrita nº41892

- Pedro Carolina nº41665

# Índice

Apresentação do Problema	1
Resolução do Problema	2
Implementação do Algoritmo	3
Estrutura do programa	3
Estruturas de dados	4
Análise do Algoritmo	5
Complexidade Temporal	5
Complexidade Espacial	7
Conclusões	8
Anexo	9

## Apresentação do Problema

O governo português tem estudado um plano de reforma administrativa, no qual cada distrito será dividido por dois municípios, sendo atribuído a cada um deles uma capital. As comunidades (aldeias, vilas ou cidades) de cada distrito terão de pertencer obrigatoriamente a uma dessas capitais. De modo a se fazer essa atribuição, ficou estipulado que uma comunidade pertenceria a uma capital, caso a distância a essa fosse menor do que a distância calculada à outra capital.

O objetivo do problema será calcular quantas comunidades pertencem a cada capital, bem como quantas poderão escolher a qual pertencer, caso a distância seja igual.

Obrigatoriamente tem de haver pelo menos duas comunidades (consequentemente duas capitais), pelo menos uma estrada (ligação entre duas comunidades) e a distância mínima entre duas comunidades ser pelo menos um.

## Resolução do Problema

Para resolver este problema, utilizámos uma estrutura de grafos.

Os vértices guardam o nº da cidade enquanto os arcos representam a distância entre duas cidades; o grafo é não orientado (não é de sentido único) e cíclico (o primeiro e o último vértice podem ser iguais, enquanto os arcos entre eles poderão ser diferentes).

O facto do peso dos arcos ser sempre positivo e a necessidade de se calcular o menor caminho de um nó a todos os outros, fez-nos optar por implementar o algoritmo de **Dijkstra**. Realizámos algumas alterações ao algoritmo original, como não ter utilizado um vetor *via* que guardasse os caminhos entre as comunidades.

No contexto deste problema apenas nos interessa saber a menor distância de dois nós – o das capitais - a todos os outros. Por essa mesma razão, o nosso algoritmo passou a ter dois vetores de distâncias com a finalidade de alocar a distância mais curta de cada nó à capital respetiva. Observemos o seguinte exemplo com as distâncias da capital nº 1 e da capital nº7:

Consideremos **C** como sendo abreviação de comunidade,

C0	<b>C1</b>	C2	C3	C4	C5	C6	C7
15	0	12	7	25	30	33	50

Figura 1 – Vetor de distâncias da capital 1 a todas as outras comunidades

C0	C1	C2	C3	C4	C5	C6	<b>C7</b>
65	50	62	57	25	20	83	0

Figura 2 – Vetor de distâncias da capital 7 a todas as outras comunidades

Após o preenchimento de ambos os vetores, podemos afirmar a que capital pertencerá cada comunidade, bem como se uma comunidade poderá escolher a capital que quer pertencer. Por exemplo, olhando para os vetores a cima, conseguimos perceber que para a C4, ambas as capitais têm a distância de 25, o que significa que esta comunidade tem a possibilidade de escolher a capital que deseja pertencer.

# Implementação do Algoritmo

## Estrutura do programa

### Main

É a classe responsável pelas operações de input/output, bem como pela instanciação do problema. Esta instanciação começa com o envio para a classe *AdministrativeReform* de o nº de comunidades e de estradas. Seguidamente, envia-se cada ligação entre duas comunidades, enviando o nº de cada cidade e a distância entre as duas. Por fim, endereça-se os nºs das duas comunidades que são capitais.

No final o output que se espera, é o nº de comunidades que pertencem:

- À primeira capital inserida pelo utilizador
- À segunda capital inserida pelo utilizador
- À possibilidade de escolher entre a primeira ou a segunda capital inserida pelo utilizador

### AdministrativeReform

Trata-se da classe que trabalha os inputs enviados pela **Main**. No construtor da mesma, são inicializadas todas as estruturas de dados necessárias.

Existem três métodos particularmente importantes para a implementação do algoritmo de Dijkstra:

- ***init(int origin)*** – Preenchimento dos vetores *distance1*, *distance2* pelo maior valor inteiro e *selected* com as posições todas a *false*, visto que nesta fase de execução ainda nenhum nó foi explorado e nenhuma distância comparada. Coloca-se na lista por explorar apenas a capital recebida no argumento do método e a atribui-se uma distância de zero no vetor de distâncias respetivo (*distance1* caso seja a primeira capital a ser inserida, *distance2* caso contrário).
- ***computeSolution()*** – Terá a função de controlar o *flow* do nosso algoritmo, pois é a partir desta que:
  - inicializam-se os vetores relativos a cada capital, através de uma chamada a *init*
  - guarda-se o nó que foi explorado, que se encontra na cabeça da lista ordenada
  - invoca-se a função *exploreNode*
  - atribui-se as capitais às comunidades
- ***exploreNode(int node, int cap)*** – Recebe como argumentos, o nó a explorar e a capital. Caso este nó não tenha sido ainda explorado, verifica-se a soma do custo a explorar, de cada ligação que este nó tem, guardando essa distância num vetor para cada nó. À medida que o algoritmo vai sendo executado, o vetor de distâncias pode

sofrer alterações caso a distância de um nó para o outro diminua. A *PriorityQueue* vai guardando estas distâncias e ordenando-as por menor distância, de modo a que no *computeSolution* se obtenha o melhor nó a explorar.

## Connection

Classe que serve para instanciar um objeto que terá um inteiro que representará a distância e outro para identificar a comunidade. O método de equals foi *overwritten* para se considerar que dois nós com o mesmo nº de comunidade são iguais.

## Estruturas de dados

```
private boolean[] selected;
```

- Vetor de *booleanos* com a funcionalidade de guardar as comunidades que já foram exploradas (ou expandidas), de modo a otimizar o nosso programa, evitando chamadas desnecessárias durante a execução do mesmo.

```
private int[] distance1;
```

- Vetor de inteiros que tem a finalidade de guardar a distância da primeira capital inserida como *input* pelo utilizador a todas as outras comunidades.

```
private int[] distance2;
```

- Vetor de inteiros que tem a finalidade de guardar a distância da segunda capital inserida como *input* pelo utilizador a todas as outras comunidades.

```
private Queue < Connection > connected;
```

- Uma *Queue* implementada através da classe *PriorityQueue* presente na biblioteca do *Java*, que serve para meter na cabeça da lista, a comunidade a explorar, ordenado pela menor distância relativamente à capital em causa.

```
private List < Connection > [] nodes;
```

- Vetor de lista de objetos da classe *Connection* que serve para criarmos a nossa estrutura de grafo, atribuindo as ligações entre cada comunidade. Estes objetos ficarão guardados numa *LinkedList*, uma classe pertencente à biblioteca do *Java*.

# Análise do Algoritmo

## Complexidade Temporal

Sendo **n** o nº de comunidades e **m** o nº de estradas, tem-se:

```
private void init(int origin) {
    if (origin == cap1) {
        for (int i = 0; i < distance1.length; i++) {
            distance1[i] = Integer.MAX_VALUE;
            selected[i] = false;
        }
        distance1[origin] = 0;
    } else if (origin == cap2) {
        ...
    }
    connected.offer(new Connection(origin, 0));
}
```

Complexity analysis for the first part of the code:

- `for (int i = 0; i < distance1.length; i++)`:  $\Theta(n)$
- `distance1[i] = Integer.MAX_VALUE;`:  $\Theta(1)$
- `selected[i] = false;`:  $\Theta(1)$
- `distance1[origin] = 0;`:  $\Theta(1)$
- `connected.offer(new Connection(origin, 0));`:  $\Theta(1)$

**Nota:**

Não se colocou a 2ª parte do *else* do método *init*, pois a complexidade é exatamente a mesma que a primeira parte da condição, só que trata a segunda capital.

```
public void computeSolution() {
    init(cap1);
    int node;
    do {
        node = connected.remove().getNode();
        selected[node] = true;
        exploreNode(node, cap1);
    } while (!connected.isEmpty());
}
```

Complexity analysis for the `computeSolution` method:

- `init(cap1);`:  $\Theta(n)$
- `int node;`:  $\Theta(1)$
- `do {`:  $\Theta(1)$
- `node = connected.remove().getNode();`:  $O(\log(n))$
- `selected[node] = true;`:  $\Theta(1)$
- `exploreNode(node, cap1);`:  $O(m) + O(\log(n))$
- `} while (!connected.isEmpty());`:  $O(n)$

(Neste espaço situaria-se uma réplica do código acima, para a segunda capital)

```
for(int j=0; j<distance1.length; j++){
    if(distance1[j]<distance2[j])
        first++;
    else if(distance1[j]>distance2[j])
        second++;
    else
        any++;
}
```

Complexity analysis for the comparison loop:

- `for(int j=0; j<distance1.length; j++)`:  $\Theta(n)$
- `if(distance1[j]<distance2[j])`:  $\Theta(1)$
- `first++;`:  $\Theta(1)$
- `else if(distance1[j]>distance2[j])`:  $\Theta(1)$
- `second++;`:  $\Theta(1)$
- `else`:  $\Theta(1)$
- `any++;`:  $\Theta(1)$

**Nota:**

Não se colocou o trecho de código relativo á segunda capital no método *computeSolution*, pois a complexidade temporal é exatamente a mesma de quando se inicializa a primeira.

```
private void exploreNode(int node,int cap) {
    int length;
    if(cap==cap1) {
        for (Connection c : nodes[node]) {
            if (!selected[c.node]) {
                length = distance1[node] + c.cost;
                if (length < distance1[c.getNode()]) {
                    boolean nodeIsInQueue = distance1[c.node]<
                    Integer.MAX_VALUE;
                    distance1[c.node] = length;
                    if (nodeIsInQueue) {
                        Connection explored = new Connection(c.node, length);
                        connected.remove(explored);
                        connected.offer(explored);
                    } else {
                        connected.offer(new Connection(c.node, distance1[c.node]));
                    }
                }
            }
        }
    } else {
        ...
    }
}
```

Complexity annotations from the image:

- $\Theta(1)$  for `int length;`
- $\Theta(1)$  for `if(cap==cap1) {`
- $O(m)$  for `for (Connection c : nodes[node]) {`
- $\Theta(1)$  for the inner `if` block (lines 14-19)
- $O(\log(n))$  for `connected.remove(explored);`
- $\Theta(1)$  for `connected.offer(explored);`
- $\Theta(1)$  for `connected.offer(new Connection(c.node, distance1[c.node]));`

**Nota:**

Não se colocou a 2ª parte do *else* do método *exploreNode*, pois a complexidade é exatamente a mesma que a primeira parte da condição, só que trata a segunda capital.

Complexidade temporal do algoritmo = complexidade temporal (*init*) + complexidade temporal (*computeSolution*) + complexidade temporal (*exploreNode*)

$$\begin{aligned}
 &= 2 \times \Theta(n) \text{ init} \\
 &+ 2 \times (O(\log(n)) + O(m) + O(\log(n)) + O(n)) + 3 \times \Theta(n) \text{ computeSolution} \\
 &+ 2 \times (O(\log(n)) + O(m)) \text{ exploreNode} \\
 &= 7\Theta(n) + 6O(\log(n)) + 4O(m) \\
 &= \Theta(n) + O(\log(n) + m)
 \end{aligned}$$



## Complexidade Espacial

Sendo  $n$  o nº de comunidades, tem-se:

```
private boolean[] selected;  $\Theta(n)$ 
```

```
private int[] distance1;  $\Theta(n)$ 
```

```
private int[] distance2;  $\Theta(n)$ 
```

```
private Queue < Connection > connected;  $O(n)$ 
```

```
private List < Connection > [] nodes;  $O(n \times (n-1))$ 
```

Complexidade espacial do algoritmo =  $\Theta(n) \times 3 + O(n) + O(n \times (n-1)) =$   
 $O(n \times (n-1))$

## Conclusões

Tanto a complexidade temporal como espacial do nosso trabalho poderia ter melhorado significativamente, nomeadamente, para praticamente metade. Poderíamos ter evitado utilizar dois vetores de distâncias e deste modo ignorado algumas iterações de exploração de nós.

Para isso teríamos de ter implementado o algoritmo de Dijkstra com mais um vetor, e.g., *via*, que guardasse a informação de que comunidade era preciso passar para se chegar a uma determinada outra. Olhemos para o seguinte exemplo onde se apenas calculou a distância de uma capital (assumindo que a primeira capital é a nº1 e a segunda é a nº7) relativamente às outras comunidades e se utilizou o presumido vetor *via*:

Consideremos **C** como sendo abreviação de comunidade,

*Index*   C0   C1   C2   C3   C4   C5   C6   C7

1	0	1	1	1	4	2	5
---	---	---	---	---	---	---	---

Figura 3 – Vetor *via*

15	0	12	7	25	30	33	50
----	---	----	---	----	----	----	----

Figura 4 – Vetor *distância*

Sabendo as comunidades que de certeza absoluta terão de passar pela **C1** se tiverem de ir para a **C7**, conseguimos automaticamente saber que as mesmas pertencem a **C1**. Por exemplo, analisando o vetor *via*, conseguimos entender que as comunidades 0,3,2,6 não têm maneira de chegar à segunda capital sem passar pela primeira.

Para todas as outras, teria-se de calcular a diferença entre a distância de **C1** a **C7** menos a distância de **C1** a **C5**. Por exemplo, para se saber a que capital pertence a comunidade 5 calculamos: 50 (distância **C1** à **C7**) – 30 (distância **C1** à **C5**) = 20. Como a distância de **C7** a **C5** (20) é inferior à distância de **C1** a **C5** (30), neste caso, **C5** pertenceria à segunda capital.

## Anexo

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

public class Main {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        String[] nums = br.readLine().split(" ");

        AdministrativeReform ar = new
AdministrativeReform(Integer.parseInt(nums[0]), Integer.parseInt(nums[1]));
        for(int i=0; i<Integer.parseInt(nums[1]); i++){
            String[] line= br.readLine().split(" ");
            ar.addConnection(Integer.parseInt(line[0]),
Integer.parseInt(line[1]), Integer.parseInt(line[2]));
        }
        String[] line = br.readLine().split(" ");

        ar.setCapitals(Integer.parseInt(line[0]), Integer.parseInt(line[1]));

        ar.computeSolution();
        System.out.println(ar.getFirst() + " " + ar.getSecond() + " "
+ ar.getAny() );
    }
}
```

```

import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Queue;

public class AdministrativeReform {
    private boolean[] selected;
    private int[] distance1;
    private int[] distance2;
    private Queue<Connection> connected;
    private int first, second, any;
    private List<Connection>[] nodes;
    private int cap1, cap2;

    @SuppressWarnings("unchecked")
    public AdministrativeReform(int numNodes, int numEdges) {
        nodes = new LinkedList[numNodes];
        for(int i=0; i<nodes.length; i++){
            nodes[i] = new LinkedList<Connection>();
        }
        selected = new boolean[numNodes];
        distance1 = new int[numNodes];
        distance2 = new int[numNodes];
        connected = new PriorityQueue<Connection>(numEdges, new
Comparator<Connection>() {
            public int compare(Connection arg0, Connection arg1) {
                if (arg0.cost < arg1.cost)
                    return -1;
                else if (arg0.cost > arg1.cost)
                    return 1;
                else
                    return 0;
            }
        });
    }

    private void init(int origin) {
        if (origin == cap1) {
            for (int i = 0; i < distance1.length; i++) {
                distance1[i] = Integer.MAX_VALUE;
                selected[i] = false;
            }
            distance1[origin] = 0;
        } else if (origin == cap2) {
            for (int i = 0; i < distance2.length; i++) {
                distance2[i] = Integer.MAX_VALUE;
                selected[i] = false;
            }
            distance2[origin] = 0;
        }
        connected.offer(new Connection(origin, 0));
    }

    public void addConnection(int from, int to, int cost){
        nodes[from].add(new Connection(to, cost));
        nodes[to].add(new Connection(from, cost));
    }
}

```

```

    }

    public void setCapitals(int cap1, int cap2){
        this.cap1=cap1; this.cap2=cap2;
    }

    public void computeSolution() {
        init(cap1);
        int node;
        do {
            node = connected.remove().getNode();
            selected[node] = true;
            exploreNode(node, cap1);
        } while (!connected.isEmpty());
        init(cap2);
        do {
            node = connected.remove().getNode();
            selected[node] = true;
            exploreNode(node, cap2);
        } while (!connected.isEmpty());

        for(int j=0; j<distance1.length; j++){
            if(distance1[j]<distance2[j])
                first++;
            else if(distance1[j]>distance2[j])
                second++;
            else
                any++;
        }
    }

    public int getFirst(){
        return first;
    }
    public int getSecond(){
        return second;
    }
    public int getAny(){
        return any;
    }

    private void exploreNode(int node,int cap) {
        int length;
        if(cap==cap1){
            for (Connection c : nodes[node]) {
                if (!selected[c.node]) {
                    length = distance1[node] + c.cost;
                    if (length < distance1[c.getNode()]) {
                        boolean nodeIsInQueue = distance1[c.node] <
Integer.MAX_VALUE;

                        distance1[c.node] = length;
                        if (nodeIsInQueue) {
                            Connection explored = new
Connection(c.node, length);

                            connected.remove(explored);
                            connected.offer(explored);
                        } else {
                            connected.offer(new Connection(c.node,
distance1[c.node]));

```

