Faculdade de Ciências e Tecnologia

Análise e Desenho de Algoritmos 2016/2017

Roman Warfare

Trabalho prático nº1

Alunos:

- Filipe Cabrita nº41892

- Pedro Carolina nº41665

Índice

Apresentação do Problema

Este problema situa-se temporalmente, na era Romana, onde existem exércitos Romanos e aldeias (que ainda não pertencem ao Império Romano) que poderão ser atacadas por esses mesmos exércitos.

Existem três fatores (ordenados por ordem de importância) que temos de ter em conta, antes de se enviar um exército para um ataque: Riqueza da aldeia (1<=Riqueza<=4000); Distância entre o exército que está a atacar e a aldeia que vai ser atacada; o Custo de manutenção desse exército (1<=Custo<=10000).

A localização é medida através de coordenadas ‘x’ e ‘y’ , onde o valor mínimo de ambas é 1 e o máximo 10000. Numa coordenada (x,y), apenas pode estar ou uma aldeia ou um exército. Nenhuma aldeia tem o mesmo nível de riqueza, bem como, nenhum exército tem o mesmo custo de manutenção.

O objetivo deste problema é maximizar a riqueza conquistada, recorrendo à menor distância e custos de manutenção possível, consoante um nº de exércitos e de aldeias. No entanto, existem regras que devem ser seguidas como: Uma aldeia apenas pode ser atacada por um exército; Se um exército com custo de manutenção X atacar uma aldeia com riqueza Y, o próximo exército que for atacar, tem que ter um custo de manutenção superior a X, se quiser atacar uma aldeia com riqueza superior a Y.

Resolução do Problema

A nossa implementação baseou-se em programação dinâmica, onde começámos por preencher a matriz a partir das posições iniciais. Abaixo, usamos um raciocínio onde assumimos que as posições estão todas preenchidas menos a última.

$$DP(nA, nP)\begin{cases} 0 & ,nA = 0 \\ 0 & ,nP = 0 \\ Best\left(\sum_{i=0,}^{i=nP} \big(DP(nA-i, nP-i)\big), DP(nA-1, nP)\right) & ,nA < size(Population) + nP \; e \; nA \geq nP \end{cases}$$

$$Best(a,b)\begin{cases} a, & r(a) > r(b) \\ & \vee \big(r(a) == r(b) \; \wedge d(a) < d(b)\big) \\ \vee\, (r(a) == r(b) \wedge d(a) == d(b) \; \wedge m(a) < m(b)) & \\ b, & r(b) > r(a) \\ & \vee \big(r(b) == r(a) \; \wedge d(b) < d(a)\big) \\ \vee\, (r(b) == r(a) \wedge d(b) == d(a) \; \wedge m(b) < m(a)) & \end{cases}$$

Assumindo que **r** representa a função que retorna a riqueza de um objeto que representará a população, **m** a função do custo de manutenção de um objeto que representará um exército e **d** a distância guardada nesse objeto.

Não modelámos o acesso às propriedades de cada objeto (riqueza, manutenção e distância) para cada população/exército porque assumimos que é trivial.

Implementação do Algoritmo

No nosso projeto, criámos 3 classes: *Main*, *RomanWarfare* e *People*.

Na **Main** apenas nos preocupamos em receber o *input* do utilizador, enviar esses mesmos dados para a *RomanWarfare*, que criará os objetos consoante os valores e nº de aldeias/exércitos que recebeu. No final, apenas se *printa* o resultado final, depois de a classe *RomanWarfare* e *People* completarem o seu papel.

A classe **People**, serve para definir as propriedades de uma aldeia ou exército. As propriedades são: as coordenadas (x,y) e a riqueza/custo manutenção, onde tudo é um inteiro. Esta classe implementa o *Comparable*, que nos permitirá ter uma lista ordenada na classe *RomanWarfare*, através da riqueza/custo, que simplificará a nossa resolução ao problema.

Na classe **RomanWarfare**, é onde se situa a parte mais interessante e importante do nosso projeto. Antes de receber os dados de cada exército/aldeia, são criadas duas *ArrayList<People>* com o número inserido pelo utilizador na 1ª linha do *input* na *Main*, que definirão a capacidade máxima de cada lista. Há medida que a *Main* envia uma coordenada x,y e um preço (riqueza ou manutenção), na *RomanWarfare*, é criado um objeto *People*, que é adicionado a uma desses listas *ArrayList <People>* que representará os exércitos ou a uma *ArrayList <People>* que representará as aldeias. Existe também um método que calcula a distância entre um exército e uma aldeia, utilizando para isso as coordenadas x,y de cada objeto. As listas serão ordenadas ascendentemente por ordem de custo ou riqueza quando se entra no *computeSolution*();

O método que calcula o resultado a ser enviado para a *Main*, tem três casos distintos:

- Quando o nº de exércitos é maior que o nº de aldeias (o único caso em que aplicámos a programação dinâmica);
- Quando o nº de exércitos é igual ao nº de aldeias;
- Quando o número de aldeias é maior que o nº de exércitos.

Para o primeiro caso criou-se quatro matrizes de inteiros. Uma para se ir guardando os valores da riqueza, outra para os custos e outra para as distâncias entre os exércitos e as populações. A quarta matriz serve para melhorar a eficiência do nosso programa, ao evitar que ataques que seriam impossíveis de acontecer, não sejam tidos em conta.

É no método *PreProcess* que inicializamos estas quatro matrizes (não há necessidade de o fazer se se tratar de um dos outros dois casos). Para arranjarmos uma lógica para a quarta matriz (a que melhora a eficiência) que funcionasse, tivemos de analisar vários casos e reparámos que o padrão era preencher as diagonais até se se chegar ao preenchimento do canto inferior direito da matriz.

O cálculo de cada posição das matrizes de programação dinâmica estão retratados na secção de "Resolução do Problema".

No segundo caso, quando temos um nº de aldeias igual ao nº de exércitos, todas as aldeias serão atacadas e para que se verifiquem as restrições requeridas no enunciado, estando ambas as listas ordenadas, a primeira população [j] (0<=j<=pop.size()-1) é atacada pelo primeiro exército i (0<=i<=army.size()-1) em que j=i e assim sucessivamente nas iterações seguintes.

No terceiro caso, quando temos um nº de aldeias maior que o nº de exércitos, como queremos maximizar a riqueza, as aldeias com maior riqueza terão de ser atacadas. Para isso basta começarmos por atacar as aldeias a partir do final da lista e pararmos quando o nº de aldeias atacadas for igual ao nº de exércitos.

Análise do Algoritmo

Complexidade Temporal

Sendo **a**, o nº de exércitos e **p** o nº de populações,

- Nº de populações > nº de exércitos, tem-se Θ(a)
- Nº de populações = nº de exércitos, tem-se Θ(a)
- Nº de populações < nº de exércitos, tem-se Θ(p x (a-p+1))

Se não tivéssemos uma matriz que guarda as posições dos ataques que fazem sentido, teríamos complexidade temporal de Θ(a x p).

Complexidade Espacial

Sendo **a**, o nº de exércitos e **p** o nº de populações,

- Nº de populações > nº de exércitos, tem-se Θ(a) + Θ(p) + Θ(1) = Θ(a+p)
- Nº de populações = nº de exércitos, tem-se Θ(a) + Θ(p) + Θ(1) = Θ(a+p)
- Nº de populações < nº de exércitos, tem-se

  3 * Θ((a+1) x (p+1)) (*distances*, *wealth*, *maint*)
  + Θ(a x p) (*neoTheChosenOne*)
  + Θ(a+p) (*armies*, *populations*)
  + Θ (1) (*wealthToRet*, *distToRet*, *mainToRet*, *nArmies*, *nPops*)
  = Θ((a+1) x (p+1))

Conclusões

Dois dos pontos fortes do nosso trabalho são termos usado programação dinâmica (ao invés de recursividade) e de termos criado uma matriz que guarda os possíveis ataques, evitando deste modo, cálculos de ataques desnecessários, na computação da solução.

No início do nosso projeto optámos por estudar uma solução que tentasse cobrir todos os casos possíveis do problema, no entanto, surgia sempre um ou outro caso, que invalidava a nossa solução.

Relativamente aos pontos a melhorar do nosso projeto, um deles seria ignorar a matriz que guarda as possibilidades de ataques entre exércitos e aldeias e arranjar uma solução que faça essa verificação através de um *if*, quando se está a utilizar a programação dinâmica. No entanto, isto não se traduziria em melhoramentos de complexidade, tanto espacial como temporal.

Anexo

Main

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

public class Main {

	public static void main(String[] args) {
		BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
		try{
			String[] numArmiesPop = br.readLine().split(" ");
			int nArmies = Integer.parseInt(numArmiesPop[0]);
			int nPops = Integer.parseInt(numArmiesPop[1]);
			RomanWarfare rw = new RomanWarfare(nArmies, nPops);

			for(int i=0;i< nArmies;i++){
				String[] armyInfo = br.readLine().split(" ");
				rw.addArmy(Integer.parseInt(armyInfo[0]),
Integer.parseInt(armyInfo[1]), Integer.parseInt(armyInfo[2]));
			}
			for(int i=0;i< nPops;i++){
				String[] popInfo = br.readLine().split(" ");
				rw.addPop(Integer.parseInt(popInfo[0]),
Integer.parseInt(popInfo[1]), Integer.parseInt(popInfo[2]));
			}

			rw.computeSolution();

			String output = rw.getWealth() + " " + rw.getTotalDistance() + " " +
rw.getMaintenance();

			System.out.println(output);
			//rw.printMatrix();
		}catch(IOException e){
			System.out.println("Error reading from stdin");
		}
	}

}
```

People

```
public class People implements Comparable<People>{

	private int x;
	private int y;
	private int price;

	public People(int x, int y, int price){
		this.x = x;
		this.y = y;
		this.price=price;
	}

	public int getX() {
		return x;
	}

	public void setX(int x) {
		this.x = x;
	}

	public int getY() {
		return y;
	}

	public void setY(int y) {
		this.y = y;
	}

	public int getPrice() {
		return price;
	}

	public void setPrice(int price) {
		this.price = price;
	}

	public int compareTo(People other) {
		if(this.getPrice()>other.getPrice())
			return 1;
		else if(this.getPrice()<other.getPrice())
			return -1;
		else
		return 0;		} }
```

RomanWarfare

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class RomanWarfare {
	private List<People> armies;
	private List<People> populations;
	private int nArmies;
	private int nPops;
	private int[][] distances;
	private int[][] wealth;
	private int[][] maint;
	private int[][] neoTheChosenOne;
	private int wealthToRet;
	private int distToRet;
	private int maintToRet;

	public RomanWarfare(int nArmies, int nPops) {
		this.nArmies = nArmies;
		this.nPops = nPops;
		armies = new ArrayList<People>(nArmies);
		populations = new ArrayList<People>(nPops);
	}

	public void addArmy(int x, int y, int cost) {
		armies.add(new People(x, y, cost));
	}

	public void addPop(int x, int y, int wealth) {
		populations.add(new People(x, y, wealth));
	}

	public int getWealth() {
		return wealthToRet;
	}

	public int getTotalDistance() {
		return distToRet;
	}

	public int getMaintenance() {
		return maintToRet;
	}
```

```
	private int getDistance(People p1, People p2) {
		return Math.abs(p1.getX() - p2.getX()) + Math.abs(p1.getY() - p2.getY());
	}

	private void preProcess() {
		distances = new int[nArmies + 1][nPops + 1];
		wealth = new int[nArmies + 1][nPops + 1];
		maint = new int[nArmies + 1][nPops + 1];
		neoTheChosenOne = new int[nArmies][nPops];

		int k = 0;
		do {
			for (int i = 0, j = 0; j < nPops; i++, j++) {
				neoTheChosenOne[i + k][j] = -1;
			}
			k++;
		} while (neoTheChosenOne[nArmies - 1][nPops - 1] == 0);
	}

	private void sort(){
		Collections.sort(populations);
		Collections.sort(armies);
	}

	public void computeSolution() {
		sort();

		if(nArmies>nPops){
		preProcess();
		for (int currentArmy = 1; currentArmy < nArmies + 1; currentArmy++) {
			for (int currentPop = 1; currentPop < nPops + 1; currentPop++) {
				if (neoTheChosenOne[currentArmy - 1][currentPop - 1] == -1) {
					int w = wealth[currentArmy - 1][currentPop - 1] +
populations.get(currentPop - 1).getPrice();
					int d = distances[currentArmy - 1][currentPop - 1] +
getDistance(populations.get(currentPop - 1), armies.get(currentArmy - 1));
					int m = maint[currentArmy - 1][currentPop - 1] +
armies.get(currentArmy - 1).getPrice();

					if (w > wealth[currentArmy - 1][currentPop]) {
						setProperties(w, d, m, currentArmy, currentPop);
					} else if (d < distances[currentArmy - 1][currentPop] && w
== wealth[currentArmy - 1][currentPop]) {
						setProperties(w, d, m, currentArmy, currentPop);
					} else if (m < maint[currentArmy - 1][currentPop] && d ==
```

```
distances[currentArmy - 1][currentPop] && w == wealth[currentArmy - 1][currentPop]) {
                                                setProperties(w, d, m, currentArmy, currentPop);
                                        } else {
                                                wealth[currentArmy][currentPop] =
wealth[currentArmy - 1][currentPop];
                                                distances[currentArmy][currentPop] =
distances[currentArmy - 1][currentPop];
                                                maint[currentArmy][currentPop] =
maint[currentArmy - 1][currentPop];
                                        }
                                } else {
                                        continue;
                                }
                        }
                }
                wealthToRet=wealth[nArmies][nPops];
                distToRet = distances[nArmies][nPops];
                maintToRet = maint[nArmies][nPops];
                } else  if (nPops==nArmies){
                        for (int i = 0; i < nArmies; i++) {
                                People currArmy = armies.get(i);
                                People currPop = populations.get(i);
                                wealthToRet += currPop.getPrice();
                                maintToRet += currArmy.getPrice();
                                distToRet += getDistance(currArmy, currPop);
                        }
                }else{
                        for(int i = nArmies-1, j = nPops-1; i>=0; i--, j--){
                                People currArmy = armies.get(i);
                                People currPop = populations.get(j);
                                wealthToRet += currPop.getPrice();
                                maintToRet += currArmy.getPrice();
                                distToRet += getDistance(currArmy, currPop);
                        }
                }
        }

        private void setProperties(int w, int d, int m, int currentArmy, int currentPop) {
                wealth[currentArmy][currentPop] = w;
                distances[currentArmy][currentPop] = d;
                maint[currentArmy][currentPop] = m;
        }

        /*
         * public void printMatrix() { computeSolution(); for (int i = 0; i <
         * neoTheChosenOne.length; i++) { for (int j = 0; j <
```

```
        * neoTheChosenOne[i].length; j++) { System.out.print(neoTheChosenOne[i][j]
        * + " "); } System.out.println(); } }
        */

}
```