Faculdade de Ciências e Tecnologia

Análise e Desenho de Algoritmos 2016/2017

LionLand

Trabalho prático nº2

Docente Margarida Mamede

- Filipe Cabrita nº41892

- Pedro Carolina nº41665

Índice

Apresentação do Problema

No problema apresentado, está representado um mundo onde existem dois leões (Simba e Nala) que têm como objetivo encontrar-se na célula do ❤. Neste mundo, com dimensões de 1<=R<=40 (sendo R, nº de linhas) por 1<=C<=40 (sendo C, nº de colunas), são atribuídas posições (1<=x<=R, 1<=y<=C) ao Simba, à Nala e à célula do amor.

O Simba e a Nala podem mover-se nas direções cardeais (sul,norte, este ou oeste), com um custo constante de 1, sempre que se deslocam de uma posição para outra e desde que não exista uma parede a bloquear o caminho! Como não poderia deixar de ser, este mundo encontra-se amaldiçoado, sendo que, sempre que um dos leões se move, o movimento do outro está dependente da direção que o anterior optou. Temos quatro situações a considerar:

- Se um dos leões se mover para norte, então o outro também tem de ir para norte. No entanto, se um dos leões tiver uma parede a bloquear o caminho para norte, fica na mesma posição.
- Se um dos leões se mover para sul, então o outro também tem de ir para sul. No entanto, se um dos leões tiver uma parede a bloquear o caminho para sul, fica na mesma posição.
- Se um dos leões se mover para oeste, então o outro tem de ir para este. Se o segundo leão tiver uma parede a bloquear o caminho para este, fica na mesma posição.
- Se um dos leões se mover para este, então o outro também tem de ir para oeste. Se o segundo leão tiver uma parede a bloquear o caminho para oeste, fica na mesma posição.

Apesar de haver várias combinações de caminhos para se chegar ao objetivo do problema, a que interessa obter como resultado é a que contém menor custo.

Resolução do Problema

O problema foi resolvido recorrendo a brute forcing, que no âmbito do projeto consiste em analisar todas os caminhos possíveis dos dois leões em simultâneo, até se encontrar a célula pretendida, apresentando no final o menor custo.

Inicialmente considerámos uma implementação de grafos com pesquisa em largura, no entanto devido às características do problema e de serem dois leões em vez de um a movimentarem-se em simultâneo, somando as restrições impostas de um movimento de um leão estar dependente do outro, tivemos consciência da dificuldade da implementação do problema nestas condições. Tendo em conta que no máximo temos apenas 40 linhas e 40 colunas, o número máximo de combinações apenas poderá ascender aos (40*40)^2 = 2560000, o que consideramos aceitável.

Implementação do Algoritmo

Estrutura do programa

Main

Na classe Main tratamos de todas as operações de input/output. Começamos por inicializar a nossa classe de topo "Land" com os argumentos que recebemos da consola e de seguida preenchemos as informações acerca do "mapa" onde os leões se encontram. Com isto temos todos os dados necessários para resolver o problema e escrevemos na consola o resultado que é pretendido no problema: o custo mínimo no caso de haver solução, ou "NO LOVE" caso contrário.

Land

Esta é a classe que implementa toda a lógica do nosso programa. Inicialmente são criadas as estruturas de dados necessárias à resolução do nosso problema.

As posições iniciais dos leões, da célula do amor e o número de linhas e colunas da matriz são guardadas como variáveis globais.

A nossa função principal – *computeSolution()* – é a mais importante desta classe. É nesta função que são fornecidas as posições iniciais dos leões, sendo que no início apenas temos um objeto para se expandir, onde avaliamos os possíveis caminhos que este pode tomar (no máximo 4). Na segunda iteração temos 4 objetos para expandir (16 combinações possíveis de caminhos) e assim em diante, até que as posições dos dois leões cheguem à célula objetivo simultaneamente. Sempre que um leão avança para uma determinada célula, esta é guardada na estrututa de células visitadas, de modo a impedir um movimento repetido de combinações numa iteração futura. Temos ainda duas funções auxiliares: uma que preenche as posições das paredes numa matriz; outra que indica se o leão está a tentar movimentar-se para um limite exterior ao da matriz ou em direção a uma parede.

Position

A estrutura de dados Position guarda informação acerca de uma combinação de posições dos dois leões e do número de movimentos(ou seja, o custo) que foram efectuados para chegar àquela combinação.

Estruturas de dados.

boolean[][] land; Esta estrutura de dados indica-nos se uma dada posição do nosso mapa está livre ou tem uma parede.

Queue<Position> toExpand; Estrutura onde vamos guardar as casas que são possíveis de ser visitadas e que ainda não foram.

boolean[][][][] visitedStates; Estrutura que nos indica se uma dada posição já foi visitada ou não.

Análise do Algoritmo

Complexidade Temporal

Sendo:

n o nº de linhas

m o nº de colunas

Inicialização de char[][] land: Θ(m ×n)

Expansão O ((m × n)2)

Complexidade Total: O ((m × n)2)

Complexidade Espacial

Sendo:

n o nº de linha

m o nº de colunas

private Queue < Position > toExpand; O(4 × (m × n) – (n ×m)) = O(3(n ×m)) = O(n ×m)

private boolean[][][][] visitedStates; Θ((m × n)2)

private boolean[][] land; Θ(m ×n)

Complexidade total: Θ((m × n)2)

Conclusões

Um dos nossos grandes pontos fortes é o facto de termos acessos constantes na verificação de posições já visitadas, uma vez que é uma operação realizada frequentemente. O mesmo acontece com a verificação da existência de uma parede numa dada posição.

Optámos por sacrificar a complexidade espacial em prol da temporal. Poderíamos ter utilizado a memória de maneira mais eficiente, se em vez de guardar o mapa, guardássmos apenas as posições que têm uma parede.

Na fase incial, pensámos em abdicar da matriz que guarda as paredes, passando a ter as paredes na estrutura de dados das posições visitadas. Para isso, se tivessemos uma parede na posição x, y e sendo n o número de linhas do nosso mapa e m o número de colunas, teríamos então:

$$\forall a, b: 0 \leq a \leq n-1 \ \wedge \ 0 \leq b \leq m-1$$

$$=> \ Visitados[x][y][a][b] = true \ \wedge \ Visitados[a][b][x][y] = true$$

No entanto, (sendo **p** o numero de paredes) isto teria uma complexidade temporal de:

$\Theta(p \times n \times m \times 2)$

Outra solução, seria guardar as paredes num *hashmap*, no entanto, com o *overhead* desta solução, optámos por manter a ideia da matriz de quatro dimensões.

Também pensámos em utilizar uma fila com prioridade de duas formas:

- Na primeira, as expansões seriam executadas pelos nós com o maior custo
- Na segunda, as expansões seriam executadas pelos nós com o menor custo

Após analisarmos esta solução a fundo, verificámos que não seria a melhor.

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 line = "";

		while ((line = br.readLine()) != null) {
			String[] matrixDimensions = line.split(" ");
			if (matrixDimensions.length != 2)
				System.exit(0);

			int nRows = Integer.parseInt(matrixDimensions[0]);
			int nCols = Integer.parseInt(matrixDimensions[1]);

			String[] objectPositions = br.readLine().split(" ");

			Land l = new Land(nRows, nCols,
Integer.parseInt(objectPositions[0]), Integer.parseInt(objectPositions[1]),
					Integer.parseInt(objectPositions[2]),
Integer.parseInt(objectPositions[3]),
					Integer.parseInt(objectPositions[4]),
Integer.parseInt(objectPositions[5]));

			for (int i = 0; i < nRows; i++) {
				String fillingMatrix = br.readLine();
				l.fillWall(fillingMatrix, i);
			}

			int result = l.computeSolution();
			if (result != -1)
				System.out.println(result);
			else
				System.out.println(Land.NOLOVE);
		}
	}
}
```

```
import java.util.LinkedList;
import java.util.Queue;

public class Land {

	public static final char WALL = '#';
	public static final String NOLOVE = "NO LOVE";

	private boolean[][] land;
	private int treasureX, treasureY, initialNalaX, initialNalaY,
initialSimbaX, initialSimbaY;
	private int nRows, nCols;
	private int cost;
	private Queue<Position> toExpand;
	private boolean[][][][] visitedStates;

	public Land(int nRows, int nCols, int treasureX, int treasureY, int NalaX,
int NalaY, int SimbaX, int SimbaY) {
		land = new boolean[nRows][nCols];
		this.treasureX = treasureX - 1;
		this.treasureY = treasureY - 1;
		this.initialNalaX = NalaX - 1;
		this.initialNalaY = NalaY - 1;
		this.initialSimbaX = SimbaX - 1;
		this.initialSimbaY = SimbaY - 1;
		this.nRows = nRows - 1;
		this.nCols = nCols - 1;
		visitedStates = new boolean[nRows][nCols][nRows][nCols];
	}

	public int computeSolution() {

		Position NalaSimbaFrom = new Position(initialNalaX, initialNalaY,
initialSimbaX, initialSimbaY, cost);
		toExpand = new LinkedList<>();
		toExpand.add(NalaSimbaFrom);

		while (toExpand.size() != 0) {
			Position expanded = toExpand.poll();

			int xNala = expanded.getNalaX();
			int yNala = expanded.getNalaY();

			int xSimba = expanded.getSimbaX();
			int ySimba = expanded.getSimbaY();

			int currentCost = expanded.getCurrentCost();

			if (xNala == treasureX && xSimba == treasureX && yNala ==
treasureY && ySimba == treasureY)
				return currentCost;
```

```
                    if (canMove(xNala - 1, yNala) && canMove(xSimba - 1, ySimba)
                                && !visitedStates[xNala - 1][yNala][xSimba -
1][ySimba]) {
                          Position NalaSimbaTo = new Position(xNala - 1, yNala,
xSimba - 1, ySimba, currentCost + 1);
                          toExpand.add(NalaSimbaTo);
                          visitedStates[xNala - 1][yNala][xSimba - 1][ySimba] =
true;
                    } // both moved north
                    else if (canMove(xNala - 1, yNala) && !canMove(xSimba - 1,
ySimba)
                                && !visitedStates[xNala -
1][yNala][xSimba][ySimba]) {
                          Position NalaTo = new Position(xNala - 1, yNala,
xSimba, ySimba, currentCost + 1);
                          toExpand.add(NalaTo);
                          visitedStates[xNala - 1][yNala][xSimba][ySimba] = true;
                    } // only nala moved north
                    else if (!canMove(xNala - 1, yNala) && canMove(xSimba - 1,
ySimba)
                                && !visitedStates[xNala][yNala][xSimba -
1][ySimba]) {
                          Position SimbaTo = new Position(xNala, yNala, xSimba -
1, ySimba, currentCost + 1);
                          toExpand.add(SimbaTo);
                          visitedStates[xNala][yNala][xSimba - 1][ySimba] = true;
                    } // only simba moved north

                    if (canMove(xNala + 1, yNala) && canMove(xSimba + 1, ySimba)
                                && !visitedStates[xNala + 1][yNala][xSimba +
1][ySimba]) {
                          Position NalaSimbaTo = new Position(xNala + 1, yNala,
xSimba + 1, ySimba, currentCost + 1);
                          toExpand.add(NalaSimbaTo);
                          visitedStates[xNala + 1][yNala][xSimba + 1][ySimba] =
true;
                          // System.out.println("Nala and Simba moved south");
                    }
                    else if (canMove(xNala + 1, yNala) && !canMove(xSimba + 1,
ySimba)
                                && !visitedStates[xNala +
1][yNala][xSimba][ySimba]) {
                          Position NalaTo = new Position(xNala + 1, yNala,
xSimba, ySimba, currentCost + 1);
                          toExpand.add(NalaTo);
                          visitedStates[xNala + 1][yNala][xSimba][ySimba] = true;
                          // System.out.println("Only Nala moved south");
                    } else if (!canMove(xNala + 1, yNala) && canMove(xSimba + 1,
ySimba)
                                && !visitedStates[xNala][yNala][xSimba +
1][ySimba]) {
                          Position SimbaTo = new Position(xNala, yNala, xSimba +
1, ySimba, currentCost + 1);
                          toExpand.add(SimbaTo);
                          visitedStates[xNala][yNala][xSimba + 1][ySimba] = true;
                          // System.out.println("Only Simba moved south");
```

```
                }

                if (canMove(xNala, yNala - 1) && canMove(xSimba, ySimba + 1)
                                && !visitedStates[xNala][yNala -
1][xSimba][ySimba + 1]) {
                        Position NalaSimbaTo = new Position(xNala, yNala - 1,
xSimba, ySimba + 1, currentCost + 1);
                        toExpand.add(NalaSimbaTo);
                        visitedStates[xNala][yNala - 1][xSimba][ySimba + 1] =
true;
                        // System.out.println("Nala and Simba moved westEast");
                }

                else if (canMove(xNala, yNala - 1) && !canMove(xSimba, ySimba
+ 1)
                                && !visitedStates[xNala][yNala -
1][xSimba][ySimba]) {
                        Position NalaTo = new Position(xNala, yNala - 1,
xSimba, ySimba, currentCost + 1);
                        toExpand.add(NalaTo);
                        visitedStates[xNala][yNala - 1][xSimba][ySimba] = true;
                        // System.out.println("Only Nala moved west");
                } else if (!canMove(xNala, yNala - 1) && canMove(xSimba,
ySimba + 1)
                                && !visitedStates[xNala][yNala][xSimba][ySimba +
1]) {
                        Position SimbaTo = new Position(xNala, yNala, xSimba,
ySimba + 1, currentCost + 1);
                        toExpand.add(SimbaTo);
                        visitedStates[xNala][yNala][xSimba][ySimba + 1] = true;
                        // System.out.println("Only Simba moved east");
                }

                if (canMove(xNala, yNala + 1) && canMove(xSimba, ySimba - 1)
                                && !visitedStates[xNala][yNala +
1][xSimba][ySimba - 1]) {
                        Position NalaSimbaTo = new Position(xNala, yNala + 1,
xSimba, ySimba - 1, currentCost + 1);
                        toExpand.add(NalaSimbaTo);
                        visitedStates[xNala][yNala + 1][xSimba][ySimba - 1] =
true;
                        // System.out.println("Nala and Simba moved eastWest");
                }
                else if (canMove(xNala, yNala + 1) && !canMove(xSimba, ySimba
- 1)
                                && !visitedStates[xNala][yNala +
1][xSimba][ySimba]) {
                        Position NalaTo = new Position(xNala, yNala + 1,
xSimba, ySimba, currentCost + 1);
                        toExpand.add(NalaTo);
                        visitedStates[xNala][yNala + 1][xSimba][ySimba] = true;
                        // System.out.println("Only Nala moved east");
                } else if (!canMove(xNala, yNala + 1) && canMove(xSimba,
ySimba - 1)
                                && !visitedStates[xNala][yNala][xSimba][ySimba -
1]) {
```

```
                        Position SimbaTo = new Position(xNala, yNala, xSimba,
ySimba - 1, currentCost + 1);
                        toExpand.add(SimbaTo);
                        visitedStates[xNala][yNala][xSimba][ySimba - 1] = true;
                        // System.out.println("Only Simba moved west");
                  }
            }
            return -1;
      }

      // AUXILIARY FUNCTIONS

      public void fillWall(String row, int pos) {
            for (int i = 0; i < row.length(); i++) {
                  if (row.charAt(i) == WALL)
                        land[pos][i] = true;
            }
      }

      public boolean canMove(int nRow, int nCol) {
            return !(nRow > nRows || nCol > nCols || nRow < 0 || nCol < 0 ||
land[nRow][nCol]);
      }

}



public class Position {

      private int NalaX, NalaY, SimbaX, SimbaY, currentCost;

      public Position(int NalaX, int NalaY, int SimbaX, int SimbaY, int
currentCost) {
            this.NalaX = NalaX;
            this.NalaY = NalaY;
            this.SimbaX = SimbaX;
            this.SimbaY = SimbaY;
            this.currentCost = currentCost;
      }

      public int getNalaX() {
            return NalaX;
      }

      public int getNalaY() {
            return NalaY;
      }

      public int getSimbaX() {
            return SimbaX;
      }

      public int getSimbaY() {
            return SimbaY;
      }
```

```
	public int getCurrentCost() {
		return currentCost;
	}

	public void setCost(int cost) {
		this.currentCost = cost;
	}

}
```