FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

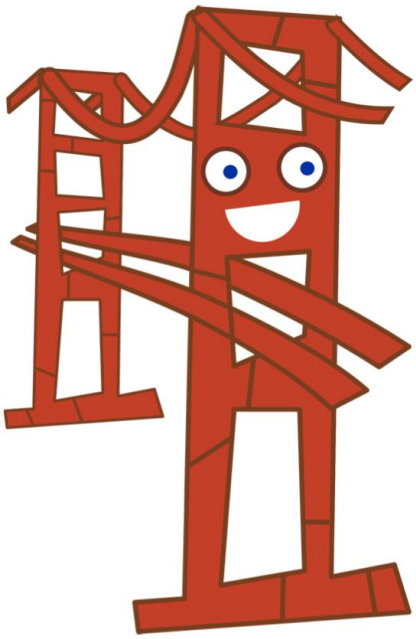
UNIVERSIDADE NOVA DE LISBOA
OMNIS CIVITAS CONTRA SE DIVISA NON STABIT

Engenharia Informática

Análise e Desenho de Algoritmos

Trabalho Prático 1

Complexity Bridges



Trabalho realizado por:

Rafael Gameiro nº50677

Rui Santos nº50833

Índice

Resolução do Problema

O problema que nos foi apresentado pode ser resolvido através da seguinte função recursiva por ramos:

$$P(i,j)\begin{cases} 0 & ,i=0 \vee j=0 \\ B(0) & ,i=1 \wedge j=1 \\ \min\big(P(i,j-1),B(j)\big) & ,i=1 \wedge j>1 \\ \min\big(P(i,j-1),P(i-1,j-S-1)+B(j)\big) & ,\ i\geq 2 \wedge j \geq 1 \end{cases}$$

- no qual *P(i, j)* é uma função que determina qual o custo mínimo para construir *i* pontes em *j* colunas.

- *B(j)* é uma estrutura de dados auxiliar que representa o tamanho das pontes em cada coluna.

Explicação dos diferentes casos da função recursiva por ramos:

- 1º caso: A primeira linha e primeira coluna da matriz (que representa não haver pontes para fazer / não haver colunas) serão colocados a 0 porque são pré-condições do enunciado que devem ser respeitadas.
- 2º caso: A posição (1,1) da matriz (que representa haver uma só ponte e uma só coluna) irá ser a primeira posição de B (o tamanho da ponte da primeira coluna).
- 3º caso: A segunda linha irá ser percorrida a partir de j = 2, em que na posição (1,j) da matriz irá ficar o valor mínimo entre a posição anterior da matriz (j-1) e a posição j de B (tamanho da ponte na coluna j).
- 4º caso (caso geral): A partir da terceira linha (consoante o input dado), na posição (i,j) da matriz irá ficar o valor mínimo entre a posição anterior da matriz e a soma da posição(i–1,j-*S*-1) da matriz(que representa o comprimento total mínimo de pontes que se pode fazer após se ter escolhido fazer uma ponte na posição (i,j), respeitando a regra do espaço mínimo entre pontes) com a posição j de B (tamanho da ponte na coluna j).

(a matriz referida na explicação dos diferentes casos, é a matriz que está a ser construída a partir da função recursiva por ramos $P(i,j)$).

Chamada Inicial:

- Para a resolução de um problema genérico, a função P irá receber como parâmetro *i* e *j*, no qual *i* é o número de pontes e *j* é o número de colunas.

Complexidade Espacial

Para a resolução de um problema genérico, temos os seguintes parâmetros:

Número de pontes: i

Número de colunas: j

Matriz minVal: $\Theta\big((i+1)*j\big) = \Theta(i*j)$

Vetor nBridges: $\Theta(j)$

Total: $\Theta\big((i*j)+j\big) = \Theta(i*j)$

Complexidade Temporal

Para a resolução de um problema genérico, temos os seguintes parâmetros:

Número de pontes: i

Número de colunas: j

(**Main** -> processamento das linhas do mapa)

Primeiro ciclo: $\Theta(\mathrm{i} * \mathrm{j})$

(1º *for* do método **solve**)

Segundo ciclo: $\Theta(\mathrm{j}-1) = \Theta(\mathrm{j})$

(2º e 3º *for* do método **solve**)

Terceiro ciclo:

$$\sum_{x=2}^{i} \sum_{y=aux}^{j-1} 1 = \sum_{x=2}^{i} (j-1) - aux + 1 = \sum_{x=2}^{i} j - aux = (i-1) * (j - aux)^{*} =$$

$$\mathrm{O}(\mathrm{i} * \mathrm{j})$$

em que aux = (*i* - 1) * (*S* + 1)

*como j é sempre maior ou igual do que aux optou-se pelo o de maior ordem que neste caso é j

Total: $\Theta(\mathrm{i} * \mathrm{j})$

Conclusão

No final deste trabalho chegamos à conclusão de que a nossa solução é a mais otimizada de acordo com o algoritmo desenvolvido. Um dos aspetos fortes do nosso trabalho tem haver com o facto de que no terceiro ciclo (2º e 3º *for* do método solve) conseguirmos que a complexidade temporal desse ciclo fosse reduzida de $\Theta(i * j)$ (esta era a complexidade do nosso algoritmo no terceiro ciclo antes da otimização) para $O(i * j)$. Isto tornou o nosso algoritmo mais eficiente.

Anexos

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

/**

/**
 * @author Rafael Gameiro n50677
 * @author Rui Santos n50833
 *
 */
public class Main {

    private static final char SEA = '.';

    public static void main(String[] args) throws IOException {
        // TODO Auto-generated method stub

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

        String inputRC = br.readLine();
        String[] inputRCArray = inputRC.split(" ");
        int row = Integer.parseInt(inputRCArray[0]);
        int col = Integer.parseInt(inputRCArray[1]);

        String inputBS = br.readLine();
        String[] inputBSArray = inputBS.split(" ");
        int bridges = Integer.parseInt(inputBSArray[0]);
        int minSpaces = Integer.parseInt(inputBSArray[1]);

        Bridges b = new Bridges(row, col, bridges, minSpaces);

        /**
        * Processes a line from the map inputed by the user.
        * For every '.' char, a specific position in the nBridges array is increased by 1.
        */
        int[] nBridges = new int[col];
        for(int i = 0; i < row ; i++) {
            char[] line = br.readLine().toCharArray();
            for (int j = 0; j < line.length; j++) {
                if (line[j] == SEA)
                    nBridges[j]++;
            }
        }

        br.close();
        b.matrixNBridges(nBridges);

        System.out.println(b.solve());

    }

}
```

```java
/**
 *
 */

/**
 * @author Rafael Gameiro n50677
 * @author Rui Santos n50833
 *
 */
public class Bridges {

    int row;
    int col;
    int bridges;
    int space;
    int[] nBridges;

    public Bridges(int row, int col, int bridges, int space) {
        this.row = row;
        this.col = col;
        this.bridges = bridges;
        this.space = space;
        nBridges = new int[col];
    }

    /**
     * Receives an array with the bridge size for each column.
     */
    public void matrixNBridges(int[] array) {
        nBridges = array;
    }

    /**
     * Computes the minimum cost to build a specific numbers of bridges within j columns.
     * Initially fills the second row of the matrix, and after that computes the remaining rows,
     * based on the values in the previous row.
     *
     * @return the minimum cost to the number of bridges requested by the user,
     *         with a specific number of columns
     */
    public int solve() {
        int[][] minVal = new int[bridges + 1][col];

        minVal[1][0] = nBridges[0];
        for (int j = 1; j < col; j++)
            minVal[1][j] = Math.min(nBridges[j], minVal[1][j - 1]);

        for (int i = 2; i < bridges + 1; i++) {
            int aux = (i - 1) * (space + 1);
            for (int j = aux; j < col; j++) {
                if (j == aux)
                    minVal[i][j] = minVal[i - 1][j - space - 1] + nBridges[j];
                else if (j > i) {
                    minVal[i][j] = Math.min(minVal[i][j - 1], minVal[i - 1][j - space - 1] + nBridges[j]);
                }
            }
        }

        return minVal[bridges][col - 1];
    }

}
```