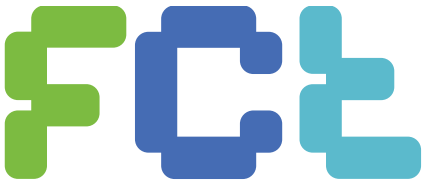

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Análise e Desenho de Algoritmos

Relatório do 3º Trabalho

Ana Garcia, MIEI 46783

Maio 2015

Conteúdo

1 Apresentação do Problema

Escaping zombies é um simples problema de pesquisa em largura. Existe um labirinto, com L linhas e C colunas, composto por um ponto de partida, um ponto de chegada, muros, caminhos, e uma câmara de veneno.

O objetivo é encontrar o caminho mais curto, evitando o veneno se possível, pois este 'veneno' liberta os zombies e torna o labirinto menos 'seguro'.

Se for possível, envia a mensagem de "YES" e a duração do caminho. Se só existir caminho do ponto de partida ao ponto de chegada passando pelo veneno, envia a mensagem "NO" e a duração do caminho, caso não exista caminho, envia a mensagem "IMPOSSIBLE".

2 Resolução do Problema

Após uma leitura cuidada do enunciado é possível verificar que é um simples problema de pesquisa em largura. Na leitura do labirinto, ao qual chamo de map, introduzi uma parede em volta, ou seja, um muro que limita o labirinto, para não me precupar com margens aquando a resolução do problema. Neste mapeamento de leitura, apenas registo se existe caminho ou se é muro, tomando o poison como sendo muro.

Após mandar para um método inteiro que pesquisa a existência do menor caminho, se receber um numero maior ou igual a 0 (ou seja, o tamanho) envia "YES" e o tamanho recebido, se receber -1, no mapeamento troca o poison de muro para caminho e volta a pedir ao método o caminho mais curto.

Se receber um numero maior ou igual a 0 (ou seja, o tamanho) envia "NO" e o tamanho recebido, se receber -1, significa que não existe caminho possível e envia "IMPOSSIBLE".

O metodo de resolução, contem um labirinto, do mesmo tamanho que o do labirinto original, chamado found, para registar os pontos que forem sendo percorridos, de modo a não entrar em ciclo. Começa no ponto "Start", percorre para cima, baixo, esquerda, direita, se houver caminho, pergunta se já está no ponto "End", se estiver envia o tamanho, pergunta se está na câmara de veneno, para saber que cada caminho passa a custar dois em vez de um, e soma o valor correcto ao tamanho do percurso.

3 Implementação do Algoritmo

Na estrutura do programa foi usada uma classe Main, com uma função resolution.

As estruturas de dados utilizadas na classe Main, é um vector booleano bidimencional map, para guardar o labirinto, cujo o tamanho é o número de linhas + 2, número de colunas + 2; no método contido na Main tenho uma fila FIFO para ir guardando as possições onde se encontra a percorrer o caminho; e um outro vector booleano bidimencional found, para guardar as possições já percorridas.

4 Análise do Algoritmo

Seja L o número de linhas e C o número de colunas.

A complexidade espacial desta implementação é $\Theta(|L+2| * |C+2|)$.

A complexidade temporal é de $\Theta(|L| * |C|)$ na leitura na main, no método resolution, no pior caso, analisa todas as posições excepto o muro de limitação, ou seja, $O(|L| * |C|)$, ou seja, o pior caso do método mais a leitura dá $\Theta(|L| * |C|)$.

5 Conclusões

Em suma, o algoritmo de pesquisa em largura resolveu este problema. Foi pensada na realição de uma optimização do algoritmo, pois, no pior caso, este executa o mesmo metodo duas vezes. Como não alterava a complexidade do problema não executei. Não foi implementada nenhuma TAD, pois não vi nenhuma vantagem na mesma, para este problema específico.

6 Código Fonte Completo

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.LinkedList;
import java.util.Queue;
public class Main {
  public static void main(String[] args) throws IOException {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String[] split = in.readLine().split("␣");
    int L = Integer.parseInt(split[0]);
    int C = Integer.parseInt(split[1]);
    // criacao de paredes a volta, para nao ter de verificar os
    // limites do map
    boolean[][] map = new boolean[L + 2][C + 2];
    int lineP = 0, columnP = 0, lineS = 0, columnS = 0, lineE = 0, columnE = 0;
    for (int i = 0; i < L; i++) {
      String line = in.readLine();
      for (int j = 0; j < C; j++) {
        if (line.charAt(j) == '.') {
          map[i + 1][j + 1] = true;
        } else if (line.charAt(j) == '#') {
          map[i + 1][j + 1] = false;
        } else if (line.charAt(j) == 'P') {
          map[i + 1][j + 1] = false;
          lineP = i + 1;
          columnP = j + 1;
        } else if (line.charAt(j) == 'S') {
          map[i + 1][j + 1] = true;
          lineS = i + 1;
          columnS = j + 1;
        } else if (line.charAt(j) == 'E') {
          map[i + 1][j + 1] = true;
          lineE = i + 1;
          columnE = j + 1;
        }
      }
    }
```

```
    int a = resolution(map, L, C, lineS, columnS, lineE, columnE, lineP,
    columnP);
    if (a == -1) {
      map[lineP][columnP] = true;
      int b = resolution(map, L, C, lineS, columnS, lineE, columnE,
      lineP, columnP);
      if (b == -1) {
        System.out.println("IMPOSSIBLE");
      } else {
        System.out.println("NO " + b);
      }
    } else
  System.out.println("YES " + a);
  }
```

```
  public static int resolution(boolean[][] map, int L, int C, int lineS,
  int columnS, int lineE, int columnE, int lineP, int columnP) {

    boolean[][] found = new boolean[L + 2][C + 2];
    boolean poison = false;

    // introducao aos trios, ou seja, introduzir line, column, length
    Queue<Integer> waiting = new LinkedList<Integer>();
    waiting.add(lineS);
    waiting.add(columnS);
    waiting.add(0);
    found[lineS][columnS] = true;

    while (!waiting.isEmpty()) {
      int line = waiting.poll();
      int column = waiting.poll();
      int length = waiting.poll();

      if (line == lineE && column == columnE) {
        return length;
      } else if (line == lineP && column == columnP) {
        poison = true;
      }
      length += poison ? 2 : 1;
      // cima
      add_waiting(map, found, waiting, line, column - 1, length);
      // direita
      add_waiting(map, found, waiting, line + 1, column, length);
      // baixo
      add_waiting(map, found, waiting, line, column + 1, length);
      // esquerda
      add_waiting(map, found, waiting, line - 1, column, length);

    }
    return -1;
  }

  private static void add_waiting(boolean[][] map, boolean[][] found,
  Queue<Integer> waiting, int line, int column, int length) {
    if (map[line][column] && !found[line][column]) {
      found[line][column] = true;
      waiting.add(line);
      waiting.add(column);
      waiting.add(length);
    }
  }
}
```