

Relatório Trabalho de ADA – Legionellosis

Ano Letivo	2020/2021	Semestre	2	Cadeira	ADA
Alunos	Gonçalo Martins Lourenço nº55780				
	Joana Soares Faria nº 55754				

Complexidade Temporal

Na análise da complexidade temporal consideramos as seguintes variáveis:

p - número de pessoas doentes

l - número de localizações existentes

c - número de conexões entre localidades

a - número de localizações perigosas

Em termos de complexidade temporal começamos por analisar a inicialização da lista ligada de sucessores (connections), que será executada l vezes e que em cada passo tem custo constante. Este ciclo tem então custo $\Theta(l)$.

Para resolver o problema temos um método que será executado para cada pessoa doente, ou seja, será executado p vezes. Nesse método é feita uma pesquisa em largura com limite de profundidade. A origem do grafo a considerar em cada passo é variável e a profundidade a explorar também é variável. Para doentes diferentes podemos ter de explorar vértices e arcos já explorados anteriormente.

Tendo todos esses aspetos em consideração, no pior dos casos podemos ter de explorar todos os arcos e todos os vértices, para todos os doentes, mas noutros casos, pode-se nem ter que explorar todos os vértices do grafo.

Para devolver o vetor ordenado e obedecer aos requisitos do enunciado temos uma complexidade de $O(a \times \log(a))$, sendo que a será sempre menor ou igual a l .

Assim se justifica uma complexidade de $O(p \times (l \times 2c))$. Consideramos $2 \times c$ porque cada conexão apresentada é bidirecional.

Complexidade Espacial

Na análise da complexidade espacial consideramos as seguintes variáveis:

p - número de pessoas doentes

l - número de localizações existentes

c - número de conexões

a - número de localizações perigosas

Fazendo a análise da complexidade do nosso código podemos aferir uma complexidade espacial de $\Theta(l + a + 2c)$.

Para a execução do código contamos com uma variável que guarda o número de localizações $\Theta(1)$, um vetor com todas as localizações $\Theta(l)$, assim como uma lista ligada com as localizações perigosas $\Theta(a)$. Contamos ainda com um vetor de l listas ligadas, para guardar, para cada ponto, as suas conexões. Podemos por isso dizer que cada conexão é guardada duas vezes. Sendo assim a complexidade espacial desta parte será $\Theta(2c) = \Theta(c)$.

Assim sendo, e considerando que $a \leq l \leq c + 1$, a complexidade apresentada inicialmente pode ser simplificada para $\Theta(c)$.

Conclusões

Para a implementação do algoritmo foi escolhido um percurso em largura porque é necessário percorrer todos os vértices do grafo (localizações), sem repetições, a partir de uma dada origem e limitar a profundidade alcançada. Como o percurso em largura é feito por nível, mostrou-se a forma ideal de explorar todos os nós, nível a nível, e parando no último nível a considerar.

Para a estrutura que guarda as localizações perigosas encontradas foram consideradas várias opções. A primeira opção era apenas o vetor com o número total de doentes que passavam em cada localização e no final fazer uma verificação de quais as localizações que tinham um número de doentes igual ao número total de doentes. Isto obrigaria a percorrer todo o vetor das localizações e chegar a uma complexidade de $\Theta(l)$. Sendo de esperar que o número de localizações perigosas seja várias vezes inferior ao número total de localizações, esta solução não nos pareceu adequada.

A segunda opção era considerar uma estrutura de dados ordenada para guardar as localizações perigosas. Porém o custo de inserção e remoção seria de $O(a \times \log(a))$, pelo que excluímos esta opção.

Dado que a ordenação só é de realização necessária uma vez, optou-se por uma lista ligada para as localizações perigosas pois esta apresenta inserção e remoção constantes. O custo de ordenação desta estrutura é de $O(a \times \log(a))$ pelo que se demonstrou a melhor solução.

Em estruturas de dados que não sabíamos que tamanho seria necessário escolhemos estruturas de dados dinâmicas a fim de diminuir a complexidade espacial e temporal do algoritmo.

Anexo – Código Main

```

1  /*
2   * Ada Trabalho 2 - Legionellosis
3   *
4   * @author Joana Soares Faria n55754
5   * @author Gonçalo Martins Lourenço n55780
6   */
7
8
9  import java.io.BufferedReader;
10 import java.io.IOException;
11 import java.io.InputStreamReader;
12 import java.util.List;
13
14 public class Main {
15
16     public static void main(String[] args) throws IOException {
17
18         BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
19         String[] inputLine = input.readLine().split(" ");
20         int numLocations = Integer.parseInt(inputLine[0]);
21         int numConnections = Integer.parseInt(inputLine[1]);
22
23         Legionellosis problem = new Legionellosis(numLocations);
24
25         for (int i = 0; i < numConnections; i++) {
26             String[] connection = input.readLine().split(" ");
27             int l1 = Integer.parseInt(connection[0]);
28             int l2 = Integer.parseInt(connection[1]);
29             problem.addConnection(l1, l2);
30         }
31
32         int numSick = Integer.parseInt(input.readLine());
33         for (int i = 0; i < numSick; i++) {
34             String[] sick = input.readLine().split(" ");
35             int home = Integer.parseInt(sick[0]);
36             int distance = Integer.parseInt(sick[1]);
37             problem.addSick(home, distance, numSick);
38         }
39
40         List<Integer> perilousLoc = problem.perilousLocations();
41
42
43         if (perilousLoc.size() == 0) {
44             System.out.println(0);
45         } else {
46             int initial = perilousLoc.remove(0);
47             System.out.printf("%d", initial);
48             for (int a : perilousLoc) {
49                 System.out.printf(" %d", a);
50             }
51             System.out.println();
52         }
53     }
54 }
55

```

Anexo – Código Class Legionellosis

```

1  /*
2   * Ada Trabalho 2 - Legionellosis
3   *
4   * @author Joana Soares Faria n55754
5   * @author Goncalo Martins Lourenco n55780
6   */
7
8  import java.util.Collections;
9  import java.util.LinkedList;
10 import java.util.List;
11 import java.util.Queue;
12
13 public class Legionellosis {
14
15     /**
16      * number of location in the problem
17      */
18     private final int numLocations;
19     /**
20      * number of sick person that passed in which location
21      */
22     private final int[] locationsCheck;
23     /**
24      * list of all perilous locations identified
25      */
26     private final List<Integer> perilousLocations;
27     /**
28      * connections between locations - adjacency's linked list of successors in the graph
29      */
30     private List<Integer>[] connections;
31
32
33     public Legionellosis(int numLocations) {
34         this.numLocations = numLocations;
35         locationsCheck = new int[numLocations + 1];
36         initConnections();
37         perilousLocations = new LinkedList<>();
38     }
39
40     /**
41      * Initiates the array of connections in the graph (adjacency's linked list of
42      successors)
43      */
44     @SuppressWarnings("unchecked")
45     private void initConnections() {
46         connections = new List[numLocations + 1];
47         for (int i = 0; i <= numLocations; i++) {
48             connections[i] = new LinkedList<>();
49         }
50     }
51
52     /**
53      * Adds a connection between to locations
54      *
55      * @param l1 first location
56      * @param l2 second location
57      */
58     public void addConnection(int l1, int l2) {
59         connections[l1].add(l2);
60         connections[l2].add(l1);
61     }
62
63     /**
64      * Computes the locations where a sick person might have been, that information is
65      recorded
66      * in the locationsCheck array
67      */

```

```

66      * @param home      location of the of the home of the sick person
67      * @param distance maximum distance from home the patient has been on the days
    before the
68      *                  first symptoms
69      * @param numSick    total number of sick people
70      */
71      public void addSick(int home, int distance, int numSick) {
72          boolean[] found = new boolean[numLocations + 1];
73          //explore level by level (each level represents a distance)
74          //current level being explored
75          Queue<Integer> currentLevel = new LinkedList<>();
76          //next level to explore
77          Queue<Integer> nextLevel = new LinkedList<>();
78          //current level
79          int level = 0;
80
81          //start the exploration from the sick person's home
82          currentLevel.add(home);
83
84          int loc;
85
86          //the level cannot exceed the given distance
87          while (!currentLevel.isEmpty() && level <= distance) {
88              while (!currentLevel.isEmpty()) {
89
90                  loc = currentLevel.remove();
91                  found[loc] = true;
92
93                  //increases the number of sick persons that passed by the location loc
94                  locationsCheck[loc]++;
95                  //check if all sick persons passed by the location
96                  if (locationsCheck[loc] == numSick) {
97                      //if all the sick persons passed by the location then the location
    is perilous
98                      perilousLocations.add(loc);
99                  }
100
101                  //add the node descendants to the next level to explore
102                  for (int l : connections[loc]) {
103                      if (!found[l]) {
104                          nextLevel.add(l);
105                          found[l] = true;
106                      }
107                  }
108              }
109
110              //level finished, go to next level
111              level++;
112              Queue<Integer> temp = currentLevel;
113              currentLevel = nextLevel;
114              nextLevel = temp;
115          }
116      }
117
118      /**
119       * Returns the list with all the perilous locations identified, ordered
120       *
121       * @return an ordered list with all the identified perilous locations
122       */
123      public List<Integer> perilousLocations() {
124          Collections.sort(perilousLocations);
125          return perilousLocations;
126      }
127 }
128

```