Nova School of Science & Technology

Universidade Nova de Lisboa

Legionellosis

Relatório de Análise e Desenho de Algoritmos 2020/2021(MIEI)

2º Trabalho

Alunos : José Murta (55226) e Diogo Rodrigues (56153)

Curso: Mestrado Integrado em Engenharia Informática

Turno: Po5

Docente prático: Margarida Mamede

Complexidade Temporal

Método calculate:

- Criação do vetor results $\Theta(1)$
- Por cada doente:
    1. Criação do vetor found $\Theta(1)$
    2. Criação das filas waiting e waitingNext $\Theta(1)$
    3. Adiciona-se a casa do doente à fila waiting $\Theta(1)$

    Por cada localização (vértice):
        1. Remove-se node da fila $\Theta(1)$
        2. Iteram-se os sucessores de node $\Theta(|Suc(node)|)$
        3. Inserem-se os sucessores nunca inseridos de node na fila waitingNext $O(|Suc(node)|)$
        *restantes operações de verificação $\Theta(1)$
- Para todos os doentes: $\Theta(S * (|L| + |C|))$
- Preenchimento da lista com as localizações perigosas $\Theta(L)$

Total do método: $\mathbf{O}(\boldsymbol{S} * (|\boldsymbol{L}| + |\boldsymbol{C}|))$

Complexidade Espacial

Vetor de restrições de cada doente $\Theta(|S|)$

- Vetor com 2xS posições (2 posições para cada doente)

Vetor de listas de adjacências $\Theta(|L| + |C|)$

- Vetor com L posições (uma posição para cada localização), em que cada posição contém uma lista das posições adjacentes

Vetor de contagem de contactos em cada localização $\Theta(|L|)$

- Vetor com L posições (uma posição para cada localização)

Vetor found $\Theta(|L|)$

- Vetor com L posições (uma posição para cada localização)

Fila waiting e fila waitingNext $O(|L|)$

- Filas com L posições no pior caso (uma posição para cada localização adjacente)

Total: $\boldsymbol{\Theta(|L| + |C|)}$

Conclusões

Ao longo da implementação deste projeto, pensámos em diversas alternativas tendo chegado à conclusão que esta, baseada na sugestão de resolução indicada pelos docentes, seria a mais vantajosa, tanto em complexidade espacial como em temporal.

Este problema poderia ser resolvido apenas com uma fila, onde os diferentes níveis seriam separados por um inteiro maior ou menor que o intervalo do numero de localizações/vértices. No entanto, optámos pela utilização de duas filas, pois vai ao encontro da técnica sugerida nas aulas práticas para a resolução do problema HardWeeks, em que o grafo também é percorrido por níveis.

Finalmente, ao termos a necessidade de retornar localizações perigosas sem sabermos ao certo a quantidade das mesmas, decidimos criar uma lista ligada de inteiros. Esta escolha facilita-nos também a obtenção do output correto, pois este varia consoante seja a ultima localização perigosa ou não.

```java
 * @author Diogo Rodrigues 56153 && Jose Murta 55226


import java.io.BufferedReader;

public class Main {

    public static void main(String[] args) throws IOException {
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

        String[] aux = input.readLine().split(" ");
        int nNodes = Integer.parseInt(aux[0]);
        int nEdges = Integer.parseInt(aux[1]);

        int[] temp = new int[nEdges * 2];
        int cont = 0;
        for (int i = 0; i < nEdges; i++) {
            String[] aux2 = input.readLine().split(" ");
            temp[cont++] = Integer.parseInt(aux2[0]);
            temp[cont++] = Integer.parseInt(aux2[1]);
        }

        int nSick = Integer.parseInt(input.readLine());
        int[] temp2 = new int[nSick * 2];
        int cont2 = 0;
        for (int j = 0; j < nSick; j++) {
            String[] aux3 = input.readLine().split(" ");
            temp2[cont2++] = Integer.parseInt(aux3[0]);
            temp2[cont2++] = Integer.parseInt(aux3[1]);
        }

        Legio l = new Legio(nNodes, nSick, temp, temp2);
        List<Integer> list = l.calculate();

        if (list.isEmpty()) {
            System.out.println(0);
        }
        else {
            Iterator<Integer> it = list.iterator();
            while (it.hasNext()) {
                int x = it.next();
                if (!it.hasNext()) {
                    System.out.println(x);
                } else {
                    System.out.print(x + " ");
                }
            }
        }
    }

}

```

```java
/**
 * @author Diogo Rodrigues 56153 && Jose Murta 55226
 */

import java.util.ArrayDeque;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class Legio {

    private int nNodes;
    private int nSicks;
    private int[] restri;

    private List<Integer>[] adja;

    /**
     *
     * @param nNodes - number of locations
     * @param nSicks - number of sick people
     * @param edges - all edges of the graph
     * @param restri - restrictions for each sick people
     */
    public Legio(int nNodes, int nSicks, int[] edges, int[] restri) {
        createDataStructure(nNodes);
        this.nNodes = nNodes;
        this.nSicks = nSicks;
        this.restri = restri;
        this.fill(edges);
    }

    /**
     * Calculate the perilous locations.
     * @return list with the perilous locations
     */
    public List<Integer> calculate() {
        int[] results = new int[nNodes];
        for (int i = 0; i < nSicks * 2; i += 2) {
            boolean[] found = new boolean[nNodes];
            int home = restri[i] - 1;
            int distance = restri[i + 1];

            Queue<Integer> waiting = new ArrayDeque<Integer>(nNodes);
            Queue<Integer> waitingNext = new ArrayDeque<Integer>(nNodes);
            waiting.add(home);
            found[home] = true;
            do {
                int node = waiting.remove();
                results[node]++;
                for (int j = 0; j < adja[node].size(); j++) {
                    int ad = adja[node].get(j);
                    if (!found[ad]) {
                        waitingNext.add(ad);
                        found[ad] = true;
                    }
                }
                if (waiting.isEmpty()) {
                    Queue<Integer> aux = waiting;
                    waiting = waitingNext;
                    waitingNext = aux;
                    distance--;
```

```
                }
                if (distance <0) {
                    break;
                }
            } while (!waiting.isEmpty());
        }

        List<Integer> list = new LinkedList<Integer>();
        for (int a = 0; a < nNodes; a++) {
            if (results[a] == nSicks) {
                list.add(a+1);
            }
        }
        return list;
    }


    /**
     * Fill the adjacent location of every location on the array of linked lists.
     * @param edges - int array wiht all edges of the graph
     */
    private void fill(int[] edges) {
        for (int i = 0; i < edges.length; i += 2) {
            adja[edges[i] - 1].add(edges[i + 1] - 1);
            adja[edges[i + 1] - 1].add(edges[i] - 1);
        }
    }


    @SuppressWarnings("unchecked")
    private void createDataStructure(int vecLength) {
        adja = new List[vecLength];
        for (int i = 0; i < adja.length; i++) {
            adja[i] = new LinkedList<>();
        }
    }
}
```