Legionellosis

André Santos, nº55415
Tiago Teles, nº54953

Complexidade Temporal

Criar grafo de listas de sucessores: Θ(L)

Preencher grafo: Θ(C)

Criar e preencher vetor de entrevistas: Θ(S)

Preencher vetor de booleanos de nós atingidos em cada percurso (no máximo S percursos): O(S*L)

Percurso do grafo em largura com profundidade limitada (no máximo S percursos): O(S*(L+C))

Preencher StringBuilder para preparação do output: Θ(L) (L operações em Θ(1) de complexidade amortizada – resizes de vetor para o dobro)

Complexidade total: O(S*(L+C))

Complexidade Espacial

Grafo de listas de sucessores: Θ(L+C)

Vetor de entrevistas: Θ(S)

Vetor de contadores de chegadas a nós: Θ(L)

Vetor de booleanos de nós atingidos em cada percurso do grafo: Θ(L)

Fila de nós em espera para o percurso do grafo: O(L)

StringBuilder para preparação do output: O(L)

Complexidade total: Θ(L+C)

Conclusões

Em termos de complexidade temporal cremos que a nossa solução não possa ser melhorada pois a componente máxima é a resultante dos S percursos do grafo em largura com profundidade limitada e estes são todos necessários pois é necessário marcar todos os nós por eles atingidos e dado que os grafos têm formas arbitrárias não há padrões que permitam inferir a partir de um subpercurso informação acerca dos nós que seriam atingidos por um completo.

Em termos de complexidade espacial, dado que o grafo implementado em listas de sucessores, que representa a maior componente da mesma, é necessário para ter uma boa complexidade temporal e armazena de forma mínima a informação necessária, não vemos melhorias passíveis de serem aqui aplicadas.

Já em termos absolutos espaciais seria possível armazenar as listas de sucessores em listas ligadas simples em vez de listas duplamente ligadas, visto que estas não nos trazem mais valias.

Na vertente temporal, ainda em termos absolutos, no que toca ao uso do StringBuilder ter-nos-ia sido possível manter conta do número de localizações perigosas para alocar a memória do mesmo em ligeiro excesso para evitar resizes e teria sido também possível reduzir o número de vezes que os ifs na iteração dos sucessores de cada nó (linhas 46 e 48 – Legionellosis.java) são atingidos dado que estas verificações não são sempre necessárias. No entanto, como o impacto é relativamente pequeno face à execução do resto do programa e para manter o código simples, o referido não foi implementado.

Implementámos ainda um mecanismo para desistir da pesquisa quando se torna evidente que já não há posições que possam vir a ser consideradas perigosas (quando uma posição não foi atingida por todas as vizinhanças de doentes já iteradas).
Este mecanismo poderia beneficiar de uma ordem de iteração das vizinhanças dos doentes das mais pequenas para as maiores, com o objetivo de propiciar a descoberta de vizinhanças que não se intersetam mais cedo no caso de existirem. Mais ainda, deste modo as vizinhanças que ficariam por percorrer seriam as maiores, pelo que em casos muito evidentes as poupanças seriam substanciais. A reordenação recorrendo a um quicksort teria complexidade O(S log S), um custo negligenciável dados os baixos valores de S. No entanto, como esta reordenação só seria útil em casos de falha e o critério de ordenação requerido torna o código ligeiramente verboso acabámos por não a implementar.

```java
import java.io.IOException;
import java.util.LinkedList;
import java.util.List;

public class Main {
    public static void main(String[] args) throws IOException {
        TurboScanner in = new TurboScanner(System.in);
        int L = in.nextInt();
        List<Short>[] graph = newGraph(L);
        int C = in.nextInt();
        for (int i = 0; i < C; i++) {
            int l1 = in.nextInt() - 1;
            int l2 = in.nextInt() - 1;
            graph[l1].add((short) l2);
            graph[l2].add((short) l1);
        }
        int S = in.nextInt();
        short[] interviews = new short[2 * S];
        for (int i = 0; i < S; i++) {
            interviews[i * 2] = (short) (in.nextInt() - 1);
            interviews[i * 2 + 1] = (short) in.nextInt();
        }
        System.out.println(new Legionellosis(graph, interviews).perilousLocations());
    }

    private static List<Short>[] newGraph(int vertices) {
        @SuppressWarnings("unchecked") List<Short>[] graph = (List<Short>[]) new List[vertices];
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new LinkedList<>();
        }
        return graph;
    }

}

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

public class Legionellosis {

    private final List<Short>[] graph;
    //[home,maxDistance]
    private final short[] interviews;
    private final byte numSick;

    public Legionellosis(List<Short>[] graph, short[] interviews) {
        this.graph = graph;
        this.interviews = interviews;
        numSick = (byte) (interviews.length / 2);
    }

    public String perilousLocations() {
        byte[] hits = new byte[graph.length];
        boolean[] processed = new boolean[graph.length];
        for (short i = 0; i < numSick; i++) {
            Arrays.fill(processed, false);
            if (!bfs(hits, processed, interviews[i * 2], interviews[i * 2 + 1], (short) (i + 1))) {
                return "0";
            }
        }
        return formatOutput(hits);
    }
```

```
    private boolean bfs(byte[] hits, boolean[] processed, short current, short depthLimit, short
minHits) {
        Queue<Short> border = new LinkedList<>();
        boolean valid = false;
        border.add(current);
        processed[current] = true;
        if (++hits[current] == minHits) {
            valid = true;
        }
        int lastBorderSize = 1;
        short currentDepth = 0;
        do {
            current = border.remove();
            if (--lastBorderSize == 0) {
                currentDepth++;
            }
            for (Short node : graph[current]) {
                if (!processed[node]) {
                    processed[node] = true;
                    if (++hits[node] == minHits) {
                        valid = true;
                    }
                    if (currentDepth < depthLimit) {
                        border.add(node);
                    }
                }
            }
            if (lastBorderSize == 0) {
                lastBorderSize = border.size();
            }
        } while (!border.isEmpty() && currentDepth < depthLimit);
        return valid;
    }

    private String formatOutput(byte[] hits) {
        StringBuilder output = new StringBuilder();
        for (int i = 0; i < hits.length; i++) {
            if (hits[i] == numSick) {
                output.append(i + 1).append(' ');
            }
        }
        if (output.length() == 0) {
            return "0";
        } else {
            output.setLength(output.length() - 1);
        }
        return output.toString();
    }

}
```