Instituto Superior de Engenharia do Porto

Projeto Integrador
Sprint 2 - ESINF

26/11

1150609 – Tiago Silva
1220716 - João Botelho
1220962 – Alfredo Ferreira
1220976 - Ricardo Dias

Índice:

Introdução -

Este projeto foi feito perante o enunciado fornecido na unidade curricular de Estruturas de Informação (ESINF), com o objetivo de incentivar os alunos a desenvolver uma biblioteca de classes, e os respetivos testes, que permitam gerir informação fornecida. Neste exemplo em concreto, o requisitado era a navegação nos dados relativos à rede de distribuição de cabazes de produtos agrícolas entre localidades registadas no sistema que podem conter, ou não, "hubs" de distribuição de mercadoria. Com este propósito em mente o sistema de distribuição foi mapeado com recurso às mais apropriadas estruturas de informação, "grafos".

Este relatório encontra-se dividido por cada exercício, no qual são apresentados os algoritmos de todas as funcionalidades implementadas e a sua respetiva análise de complexidade. O relatório é também iniciado com um diagrama de classes, no qual são demonstradas as interações entre as funcionalidades incluídas no programa.

Este trabalho foi realizado pelo grupo de trabalho com os elementos: Alfredo Ferreira (1220962), João Botelho (1220716), Ricardo Dias (1220976) e Tiago Silva (1150609); lecionados pelas professoras: Isabel Sampaio (PL), Ana Maria Madureira (TP) e Fátima Rodrigues (T).

USEI01 -

```java
public static void importData() throws FileNotFoundException {
    File distancias_big = new File( pathname: "distancias_big.csv");     //big
    File locais_big = new File( pathname: "locais_big.csv");             //big

    Map<String, Localidade> localidades = new HashMap<>();

    localidades = importLocalidades(locais_big, localidades);

    Scanner read = new Scanner(new FileReader(distancias_big));
    String line[] = new String[3];

    read.nextLine();
    while(read.hasNextLine()){
        line = read.nextLine().split( regex: ",");
        graph.addEdge(localidades.get((line[0])), localidades.get(line[1]),Double.parseDouble(line[2]));
    }
}
```

```java
public static Map<String, Localidade> importLocalidades(File dists, Map<String, Localidade> localidades) throws FileNotFoundException {
    Scanner read = new Scanner(new FileReader(dists));
    String line[] = new String[3];
    Localidade loc;

    read.nextLine();

    while(read.hasNextLine()){
        line = read.nextLine().split( regex: ",");
        loc = new Localidade(line[0], Double.parseDouble(line[1]), Double.parseDouble(line[2]));
        localidades.put((line[0]), loc);
    }

    return localidades;
}
```

A USEI1 envolve a criação do grafo, o formato escolhido para o grafo foi o de “MapGraph”, cujos vértices correspondem a localidades com possíveis hubs e cujas edges correspondem a estradas que conectam vértices existentes.

Para construir este grafo primeiro extraímos toda as localidades para objetos “localidade” que são armazenados num HashMap<String, Localidade>, cuja String corresponde ao “ID” da localidade que corresponde à key ligada a um value que armazena o objeto localidade correspondente.

Após isso o ficheiro csv com as distâncias é percorrido é executada uma operação por cada linha que selecionada os 2 vértices a partir dos IDs indicados na linha e cria uma edge no mapa entre os 2 vértices com o peso indicado na mesma linha.

A complexidade deste processo todo é $O(n) + O(n) \cdot O(1) = O(n)$.

USEI02 -

```java
public List<String> ordernarLocalidades(MapGraph<Localidade, Double> graph) {
    List<String> ret = new ArrayList<>();
    Map<Localidade, Integer> betMap = new HashMap<>();
    Map<Localidade, Double> closMap = new HashMap<>();

    List<Localidade> helper = graph.vertices();
    for (Localidade l : helper) {
        betMap.put(l, value:0);
    }
    for (Localidade l : helper) {
        closMap.put(l, value:0.0);
    }

    Comparator<Double> cmp = Double::compare;
    BinaryOperator<Double> sum = Double::sum;
    ArrayList<LinkedList<Localidade>> paths = new ArrayList<>();
    ArrayList<Double> dst = new ArrayList<>();

    Localidade local;
    boolean res;
    int betCrit;
    Double closCrit;
```

```java
for (int i = 0; i < graph.numVertices(); i++) {
    local = graph.vertices.get(i);
    closCrit = 0.0;

    res = shortestPaths(graph, local, cmp, sum, zero:0.0, paths, dst);

    if (res) {
        for (Double d : dst)
            closCrit += d;
        closMap.put(local, closCrit);

        for (LinkedList<Localidade> l : paths) {
            if (l.size() <= 2)
                continue;

            for (Localidade lc : l) {
                if (lc.equals(l.getFirst()) || lc.equals(l.getLast()))
                    continue;

                betCrit = betMap.get(lc);
                betMap.put(lc, betCrit + 1);
            }
        }
    }
}
```

```java
String horario;
int ilc;
for (Localidade l : betMap.keySet()) {

    ilc = Integer.parseInt(l.getIdLocalidade().substring(beginIndex:2));
    if (ilc <= 105)
        horario = "9h:00 - 14h:00";
    else if (ilc <= 215)
        horario = "11h:00 - 16h:00";
    else
        horario = "12h:00 - 17h:00";

    ret.add(String.format(format:"Localidade: %s, horario: %s | Criterios: influencia: %d, proximidade: %.0f, centralidade: %d.",
            l.getIdLocalidade(), horario, graph.outDegree(l), closMap.get(l), betMap.get(l)));
}

ret.sort(ordernarPorCriterios);
return ret;
}
```

```java
private final Comparator<String> ordernarPorCriterios = new Comparator<String>() {
    @Override
    public int compare(String l1, String l2) {
        double valsL1[] = new double[3];
        double valsL2[] = new double[3];
        Pattern p = Pattern.compile(regex:"-?\\d+(\\.\\d+)?");
        Matcher m = p.matcher(l1.substring(l1.indexOf(ch:'|')));
        int i = 0;
        while (m.find())
            valsL1[i++] = Double.parseDouble(m.group());
        m = p.matcher(l2.substring(l1.indexOf(ch:'|')));
        i = 0;
        while (m.find())
            valsL2[i++] = Double.parseDouble(m.group());
        if (valsL1[2] != valsL2[2])
            return Double.compare(valsL2[2], valsL1[2]);
        else if (valsL1[1] != valsL2[1])
            return Double.compare(valsL2[0], valsL1[0]);
        else
            return Double.compare(valsL1[1], valsL2[1]);
    }
};
```

A USEI02 pedia que fossem ordenadas as localidades por diversos critérios: influência, proximidade e centralidade. Como tal foram utilizados algoritmos vagamente baseados nos algoritmos de closeness e betweenness, e utilizando o grau do vértice para determinar a sua influência.

Devido à necessidade de obter o caminho mínimo dos vértices com todos os outros, foi utilizado o algoritmo de Dijkstra n vezes, em que n equivale o número de vértices, obtendo O(n*logn). Para cada um dos resultados obtidos são, primeiramente, somadas as distâncias dos caminhos mínimos, que servirá para calcular a proximidade do vértice de origem, tendo isto complexidade de O(n).

Seguidamente verifica-se quais os caminhos que tenham mais que 2 localizações, registando quais são as intermediárias e guardando num hashmap, do qual se faz get e put, que, em pior dos casos têm complexidade de O(n), levando a uma complexidade de O(n * (n + n)) = 0(n ^ 2), que por sua vez é efetuada n vezes, levando a O(n^3).

Finalmente, estes valores são formatados numa string e depois guardados num ArrayList, que depois será sorted utiizando o sort do Collections e um comparator que extrai os valores da string para os poder comparar.

Desta forma, a complexidade final será O(n*logn + n + n^3 + n*logn) = O(n^3).

USEI03 -

```java
public Path getLongestTrip(MapGraph<Localidade, Double> graph) {
    double maxDistance = 0;
    LinkedList<Localidade> longestPath = new LinkedList<>();

    Comparator<Double> comparator = Double::compare;
    BinaryOperator<Double> sumOperator = Double::sum;
    ArrayList<LinkedList<Localidade>> paths = new ArrayList<>();
    ArrayList<Double> distances = new ArrayList<>();


    for (int i = 0; i < graph.numVertices(); i++) {
        boolean result = shortestPaths(graph, graph.vertices.get(i), comparator, sumOperator,  zero: 0.0, paths, distances);
        if (result) {
            for (int j = 0; j < distances.size(); j++) {
                if (maxDistance < distances.get(j)) {
                    maxDistance = distances.get(j);
                    longestPath = paths.get(j);
                }
            }
        }
    }

    return new Path(longestPath, maxDistance);
}
```

```java
public String checkAutonomy(MapGraph<Localidade, Double> graph, double autonomia, Path path) {
    StringBuilder output = new StringBuilder();
    int count = 0;
    boolean impossible = false;
    double deposito = autonomia;
    Localidade vOrigem = path.getPathStops().getFirst();
    Localidade vDestino;

    for (int i = 1; i < path.getPathStops().size(); i++) {
        vDestino = path.getPathStops().get(i);
        if (graph.edge(vOrigem, vDestino).getWeight()>autonomia) {...} else {
            deposito = deposito - graph.edge(vOrigem, vDestino).getWeight();

            if (deposito < 0 && output.isEmpty()) {
                output.append(vOrigem.getIdLocalidade());
                deposito = autonomia - graph.edge(vOrigem, vDestino).getWeight();
                count++;
            } else if (deposito < 0) {
                output.append(", " + vOrigem.getIdLocalidade());
                deposito = autonomia - graph.edge(vOrigem, vDestino).getWeight();
                count++;
            } else if (deposito == 0 && output.isEmpty()){
                output.append(vDestino.getIdLocalidade());
                deposito = autonomia;
                count++;
            } else if (deposito == 0){
                output.append(", " + vDestino.getIdLocalidade());
                deposito = autonomia;
                count++;
            }
            vOrigem = vDestino;
        }
    }
    if(!impossible) {
        output.append("\nNúmero de Paragens: " + count);
    }
    return output.toString();
}
```

A USEI03 pretendia o número de paragens necessárias entre os dois pontos mais distantes, através de uma autonomia fornecida como parâmetro. Para a realização deste problema é necessário determinar primeiro quais é que são os dois pontos mais distantes no grafo e o seu menor trajeto possível. Após este passo, definir quais serão as paragens necessárias através da autonomia fornecida.

Portanto, para avaliar o pior caso de complexidade, temos de analisar cada algoritmo, como referido. O primeiro algoritmo "getLongestTrip" usufrui do algoritmo de Dijkstra para determinar o menor caminho possível de um vértice para todos os restantes. Efetua este processo para todos os vértices, comparando cada caminho registando o maior de todos. Conseguimos assim, um *loop* que é repetido V vezes, que realiza um algoritmo com complexidade de log V, V vezes e um *loop* que se repete outras A vezes, em que V é o número de vértices e A é o número de arestas/caminhos. Concluímos que a complexidade do pior caso deste algoritmo é $O(n[n + (n \cdot \log n)]) = O(n^2 \cdot \log n)$.

Para o segundo algoritmo, possuímos apenas um *loop* que percorre os vértices do caminho enviado pelo algoritmo anterior. Este *loop* repete-se V vezes, sendo V o número de vértices do caminho de menor distância entre os dois pontos mais distantes. Se adaptarmos para a complexidade de pior caso, obtemos $O(n)$.

Em geral, conseguimos uma complexidade de $O(n^2 \cdot \log n) + O(n) = O(n^2 \cdot \log n)$.

USEI04 -

```
public static void kruskal(int V, List<Edge<Localidade, Double>> graph, List<Edge<Localidade, Double>> results){
    int noOfEdges = 0;
    int j = 0;
    int[][] subset = new int[V+1][2];

    for (int i = 0; i <= V; i++) {
        subset[i][0] = i;
        subset[i][1] = 0;
    }

    while(noOfEdges < V){
        if(j < graph.size()){
            Edge<Localidade, Double> nextEdge = graph.get(j);
            int orig = Integer.parseInt(nextEdge.getVOrig().getIdLocalidade().substring( beginIndex: nextEdge.getVOrig().getIdLocalidade().indexOf("T")+1))-1;
            int dest = Integer.parseInt(nextEdge.getVDest().getIdLocalidade().substring( beginIndex: nextEdge.getVDest().getIdLocalidade().indexOf("T")+1))-1;

            int x = findRoot(subset, orig);
            int y = findRoot(subset, dest);

            if (x != y) {
                results.add(noOfEdges, nextEdge);
                union(subset, x, y);
                noOfEdges++;
            }

            j++;
        }else{
            break;
        }

    }
}
```

```
private static void union(int[][] subsets, int x, int y)
{
    int rootX = findRoot(subsets, x);
    int rootY = findRoot(subsets, y);

    if (subsets[rootY][1] < subsets[rootX][1]) {
        subsets[rootY][0] = rootX;
    }
    else if (subsets[rootX][1] < subsets[rootY][1]) {
        subsets[rootX][0] = rootY;
    }
    else {
        subsets[rootY][0] = rootX;
        subsets[rootX][1]++;
    }
}

private static int findRoot(int[][] subsets, int i)
{
    if (subsets[i][0] == i)
        return subsets[i][0];

    subsets[i][0] = findRoot(subsets, subsets[i][0]);
    return subsets[i][0];
}
```

```java
private static final Comparator<Edge<Localidade, Double>> comparatorFixe = new Comparator<Edge<Localidade, Double>>(){
    @Override
    public int compare(Edge<Localidade, Double> e1, Edge<Localidade, Double> e2){
        return Double.compare(e1.getWeight(), e2.getWeight());
    }
};
public static List<Edge<Localidade, Double>> shortestPathBetweenAll(MapGraph<Localidade, Double> graph){
    int vertices = graph.numVerts-1;
    List<Edge<Localidade, Double>> edges = new ArrayList<>(graph.edges().stream().toList());
    List<Edge<Localidade, Double>> results = new ArrayList<>();

    edges.sort(comparatorFixe);

    kruskal(vertices, edges, results);

    return results;
}
```

A USEI04 tem como objetivo determinar a rede de distância total mínima que liga todas as localidades. Para alcançar o objetivo da US, foi preciso de criar uma MST (Minimum Spanning Tree). Esta MST foi criada através do uso do algoritmo de Kruskal.

O algoritmo de Kruskal recebe como parâmetros o número de vértices-1 do grafo (a quantidade de ligações necessárias), uma List de Edges que é ordenada através de um comparator, para ser crescente em termos de peso total, e outra List de Edges na qual serão guardados o caminho necessário para obter a MST. Dentro do Algoritmo, é criado uma matrix bi-dimensional na qual serão guardados o número do vértice a que estão ligados e quantas ligações cada vértice tem. Após a criação desta matrix, o algoritmo itera por cada Edge do grafo, e por cada um, vai pegar na origem e no destino deste e, depois de verificar se são iguais, adiciona este ao MST e, coloca na matrix, o vértice a qual este se ligará. Depois de iterar por todos os Edges vai devolver a MST.

Devido ao uso da função sort dos Collections, a complexidade deste processo é O(n log n)