Licenciatura em Engenharia Informática

Instituto Superior de Engenharia do Porto

Relatório Trabalho Prático 4

João Marques (1192221)

Paulo Couto (1200587)

Janeiro 2022

Table of Contents


```java
public static String mostEfficientCircuit(Graph<Position,Double> g) {


    List<LinkedList<Position>> paths = Algorithms.allCycles(g);
    int max=0;
    double dist=Double.MAX_VALUE;
    LinkedList<Position> tempPos= null;
    for(LinkedList<Position> p: paths) {
        if (p.size() > max) {
            max = p.size();
            tempPos = p;
            dist = pathDistance(p, g);
        } else if (p.size() == max && dist < pathDistance(p, g)) {
            tempPos = p;
            dist = pathDistance(p, g);
        }
    }
    return print(tempPos);
}
public static double pathDistance (LinkedList<Position> p,Graph<Position,Double> g){
    double temp=0;
    Position tempP= p.pop();

    for (Position pos: p) {
        temp=temp+g.edge(tempP,pos).getWeight();
        tempP=pos;
    }

    return temp;
}
```

US 401

As a Traffic manager I wish to know which ports are more critical (have greater centrality) in this freight network.

Nesta User Story, foi-nos pedido que como um Gestor de Tráfego, fosse possível ver quais são os portos mais críticos, ou seja, que têm maior centralidade nesta rede.

Para tal, criámos o método getNCentralPorts, que recebe um Grafo com uma posição do navio e a sua distância, e com um inteiro referente ao número de portos que se pretende saber que têm maior centralidade. Para isso o método cria um mapa em que as chaves primarias são os vértices do grafo e número de shortestsPaths que passam nesse vértice. No fim é chamado o método toString que imprime os portos já ordenados pelo maior número de ocorrências.

```
public String getNCentralPorts(Graph<Position, Double> g, int n) {
    ArrayList<LinkedList<Position>> paths = new ArrayList<>();
    ArrayList<Double> dists = new ArrayList<>();
    List<Position> allPositions = g.vertices();
    for (Position p : allPositions) {
        dists.clear();
        paths.clear();
        Algorithms.shortestPaths(g, p, Double::compare, Double::sum,  zero: 0.0, paths, dists);
        for (int i = 0; i < paths.size(); i++) {
            LinkedList<Position> positionPath = paths.get(i);
            for (int j = 1; j < positionPath.size() - 1; j++) {
                Position position = positionPath.get(j);
                if (position.getClass().equals(Place.class) || position.getClass().equals(Port.class)) {
                    if (centralityPort.containsKey(positionPath.get(j)))
                        centralityPort.put(position, centralityPort.get(position) + 1);
                    else {
                        centralityPort.put(position, 1);
                    }
                }
            }
        }
    }
    return toString(n,centralityPort);
}
```

Figura 1 - Class Centrality ,method getNCentralPorts

```java
public static <V, E> boolean shortestPaths(Graph<V, E> g, V vOrig,
                                           Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                           ArrayList<LinkedList<V>> paths, ArrayList<E> dists) {

    if (!g.validVertex(vOrig)) return false;

    int nverts = g.numVertices();
    boolean[] visited = new boolean[nverts];

    /unchecked/
    E[] dist = (E[]) Array.newInstance(zero.getClass(), nverts);
    /unchecked/
    V[] pathKeys = (V[]) Array.newInstance(vOrig.getClass(), nverts);
    shortestPathDijkstra(g, vOrig, ce, sum, zero, visited, pathKeys, dist);
    dists.clear();
    paths.clear();

    for (int i = 0; i < nverts; i++) {
        paths.add(null);
        dists.add(null);
    }
    ArrayList<V> vertices = g.vertices();
    for (int i = 0; i < nverts; i++) {
        LinkedList<V> shortPath = new LinkedList<>();
        if (dist[i] != null){
            getPath(g, vOrig, vertices.get(i), pathKeys, shortPath);
        }
        paths.set(i, shortPath);
        dists.set(i, dist[i]);
    }
    return true;
}
```

Figura 2 – Método shorstsPaths da Classe Algorithms

```java
private static <V, E> void shortestPathDijkstra(Graph<V, E> g, V vOrig,
                                                Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                                boolean[] visited, V [] pathKeys, E [] dist) {
    int vKey = g.key(vOrig);
    dist[vKey] = zero;
    pathKeys[vKey] = vOrig;
    while (vOrig != null) {
        vKey = g.key(vOrig);
        visited[vKey] = true;
        for (V vAdj : g.adjVertices(vOrig)) {
            Edge<V, E> edge = g.edge(vOrig, vAdj);
            int vKeyAdj = g.key(vAdj);
            if (!visited[vKeyAdj]) {
                E s = sum.apply(dist[vKey], edge.getWeight());
                if (dist[vKeyAdj] == null || ce.compare(dist[vKeyAdj], s) > 0) {
                    dist[vKeyAdj] = s;
                    pathKeys[vKeyAdj] = vOrig;
                }
            }
        }
        E minDist = null; //next vetice, that has minimun dist
        vOrig = null;
        for (int i = 0; i < g.numVertices(); i++) {
            if (!visited[i] && (dist[i] != null) && ((minDist == null) || ce.compare(dist[i], minDist) < 0)) {
                minDist = dist[i];
                vOrig = g.vertex(i);
            }
        }
    }
}
```

Figura 3 - Método shorstPathDijkstra da Classe Algorithms

```java
private static <V, E> void getPath(Graph<V, E> g, V vOrig, V vDest,
                                   V [] pathKeys, LinkedList<V> path) {
    if (vOrig.equals(vDest)) {
        path.push(vDest);
    } else {
        path.push(vDest);
        int vKey = g.key(vDest);
        vDest = pathKeys[vKey];
        getPath(g, vOrig, vDest, pathKeys, path);
    }
}
```

Figura 4 – Método getPath da Classe Algorithms

```
public String toString(int n, Map<Position, Integer> portsCentrality) {
    String output ="";
    Map<Position, Integer> d = sortByComparator(portsCentrality);
    int i = 0;
    for (Position p : d.keySet()) {
        output += p + " => with " + String.format("%d", d.get(p)) + " detections in shortest paths. \n\n";
        i++;
        if (i == n) {
            break;
        }
    }
    return output;
}
```

Figura 5 – Método toString da Classe Centrality

Analise de complexidade

A complexidade da US 401:

Complexidade *toString* é constituído por um ciclo logo a complexidade é O(n).

Complexidade *getPath* é constituído por um *if* e um *else* logo tem complexidade de O(n).

Complexidade *shortestPathDijkstra* é constituído por dois ciclos encadeados logo tem complexidade de $O(n^2)$.

Complexidade *shortestPaths* é constituído por pelo método *shortestPathDijkstra* e de seguida um ciclo com a invocação do método getPath logo tem complexidade de $O(n^2)+O(n*n)=O(n^2)$.

Complexidade *getNCentralPorts* é constituído por um ciclo e dentro desse ciclo é invocado o método shortestPaths e ainda desse mesmo ciclo são chamados mais 2 ciclos encadeados logo tem complexidade de $O(n*(n^2+n*n))=O(n^3+n^3)=O(n^3)$.

Visto que o método getNCentralPorts é o método que dá a solução da user story podemos concluir que as complexidades são iguais.

US 402

As a Traffic manager I wish to know the shortest path between two locals (city and/or port).

Nesta user story era pedido que enquanto gestor de tráfego fosse possível encontrar o caminho mais curto entre dois locais. Podendo esses locais ser uma cidade e/ou um porto.

```java
public void shortPathMenu() throws IOException, InterruptedException {
    String option = "";
    LinkedList<Position> s = new LinkedList<>();
    BufferedReader read = new BufferedReader(new InputStreamReader(System.in));
    pmg.fillMatrixGraph( 6, cs.getCountryList(), ps.getPortList(), sds.getSeaDistArrayList(), bs.toMap(bs.getBorderArray(), cs.getCountryArray()));
    System.out.println(INITIALOPTIONPHRASE);
    System.out.println("1) Land path");
    System.out.println("2) Maritime path");
    System.out.println("3) Land or sea path");
    option = read.readLine();
        switch (option) {
            case "1":
                Graph<Position, Double> land = pmg.getLandMap();
                System.out.println("First location:");
                String string = read.readLine();
                System.out.println("Secound location:");
                String string1 = read.readLine();
                int keyp1 = -1;
                int keyp2 = -1;
                for (Position p : land.vertices()) {
                    if (p.getName().equalsIgnoreCase(string) || p.getCountryName().equalsIgnoreCase(string))
                        keyp1 = land.key(p);
                    if (p.getName().equalsIgnoreCase(string1) || p.getCountryName().equalsIgnoreCase(string1))
                        keyp2 = land.key(p);
                }
                if (keyp1 == -1)
                    System.out.println(string + " doesn't exist in the graph");
                if (keyp2 == -1)
                    System.out.println(string1 + " doesn't exist in the graph");
                Algorithms.shortestPath(land, land.vertex(keyp1), land.vertex(keyp2), Double::compare, Double::sum,  0.0, s);
                if (s.size() == 0) {
                    System.out.println("The two locations don't have a land connenction!");
                } else {
                    System.out.println(s);
                }
                TimeUnit.SECONDS.sleep( 10);
                break;
```

Figura 6- Class RolesUI, method shortPathMenu

```java
case "2":
    Graph<Position, Double> sea = pmg.getSeaMap();
    System.out.println(sea);
    System.out.println("First location:");
    string = read.readLine();
    System.out.println("Secound location:");
    string1 = read.readLine();
    keyp1 = -1;
    keyp2 = -1;
    for (Position p : sea.vertices()) {
        if (p.getName().equalsIgnoreCase(string) || p.getCountryName().equalsIgnoreCase(string))
            keyp1 = sea.key(p);
        if (p.getName().equalsIgnoreCase(string1) || p.getCountryName().equalsIgnoreCase(string1))
            keyp2 = sea.key(p);
    }
    if (keyp1 == -1)
        System.out.println(string + " doesn't exist in the graph");
    if (keyp2 == -1)
        System.out.println(string1 + " doesn't exist in the graph");
    Algorithms.shortestPath(sea, sea.vertex(keyp1), sea.vertex(keyp2), Double::compare, Double::sum,  zero: 0.0, s);
    if (s.size() == 0) {
        System.out.println("The two locations don't have a land connenction!");
    } else {
        System.out.println(s);
    }
    TimeUnit.SECONDS.sleep( timeout: 10);
    break;
```

Figura 7 - continuation of method shortPathMenu

```java
case "3":
    System.out.println("First location:");
    string = read.readLine();
    System.out.println("Secound location:");
    string1 = read.readLine();
    keyp1 = -1;
    keyp2 = -1;
    for (Position p : pmg.getCompleteMap().vertices()) {
        if (p.getName().equalsIgnoreCase(string) || p.getCountryName().equalsIgnoreCase(string))
            keyp1 = pmg.getCompleteMap().key(p);
        if (p.getName().equalsIgnoreCase(string1) || p.getCountryName().equalsIgnoreCase(string1))
            keyp2 = pmg.getCompleteMap().key(p);
    }
    if (keyp1 == -1)
        System.out.println(string + " doesn't exist in the graph");
    if (keyp2 == -1)
        System.out.println(string1 + " doesn't exist in the graph");
    Algorithms.shortestPath(pmg.getCompleteMap(), pmg.getCompleteMap().vertex(keyp1), pmg.getLandMap().vertex(keyp2), Double::compare, Double::sum,
    if (s.size() == 0) {
        System.out.println("The two locations don't have a land connenction!");
    } else {
        System.out.println(s);
    }
    TimeUnit.SECONDS.sleep( timeout: 10);
    break;
case "0":
    System.out.println("bye");
    break;
default:
    System.out.println("The option doens't exist");
    break;
```

Figura 8 - continuation of method shortPathMenu

```java
public static <V, E> E shortestPath(Graph<V, E> g, V vOrig, V vDest,
                                    Comparator<E> ce, BinaryOperator<E> sum, E zero,
                                    LinkedList<V> shortPath) {

    shortPath.clear();
    if (!g.validVertex(vOrig) || !g.validVertex(vDest))
        return null;

    int nverts = g.numVertices();
    boolean[] visited = new boolean[nverts];

    /unchecked/
    E[] dist = (E[]) Array.newInstance(zero.getClass(), nverts);
    /unchecked/
    V[] vertices = (V[]) Array.newInstance(vOrig.getClass(), nverts);
    shortestPathDijkstra(g, vOrig, ce, sum, zero, visited, vertices, dist);
    E lengthPath = dist[g.key(vDest)];

    if (lengthPath != null) {
        getPath(g, vOrig, vDest, vertices, shortPath);
        return lengthPath;
    }
    return null;
}
```

Figura 9 - Class Algorithms, method shortestPath

```java
public Graph<Position, Double> getLandMap(){
    Graph<Position, Double> portMap = completeMap.clone();
    for(Position p : portMap.vertices()){
        for (Position p1 : portMap.adjVertices(p)){
            if ((p.getClass().equals(Port.class) && p1.getClass().equals(Port.class)) || (p.getClass().equals(Place.class) && p1.getClass().equals(Place.
                portMap.removeEdge(p,p1);
        }
    }
    return portMap;
}

public Graph<Position, Double> getSeaMap(){
    Graph<Position, Double> seaMap = completeMap.clone();
    for(Position p : seaMap.vertices()){
        for (Position p1 : seaMap.adjVertices(p)){
            if (!((p.getClass().equals(Port.class) && p1.getClass().equals(Port.class)) || (p.getClass().equals(Place.class) && p1.getClass().equals(Plac
                seaMap.removeEdge(p,p1);
        }
    }
    return seaMap;
}
```

Figura 10 - Class PositionMatrixGraph, methods getLandMap and getSeaMap

Analise de complexidade

A complexidade da US 402:

Complexidade *getLandMap* é constituído por um ciclo for encadeado logo a complexidade é $O(n^2)$.

Complexidade *getSeaMap* é constituído por um ciclo for encadeado logo a complexidade é $O(n^2)$. Complexidade *shorstPathDijkstra* é constituído por dois ciclos encadeados logo tem complexidade de $O(n^2)$.

Complexidade *shortestPath* é constituído por pelo método *shorstPathDijkstra* e de seguida um ciclo com a invocação do método getPath logo tem complexidade de $O(n^2)+O(n*n)=O(n^2)$.

Como todos os cases do switch têm a mesma complexidade pois acabam por chamar os mesmos métodos, a complexidade total do método shortPathMenu é $O(n^2) + O(n^2) = O(n^2)$.

A complexidade da us402 é então $O(n^2)$.

US 403

As a Traffic manager I wish to know the most efficient circuit that starts from a source location and visits the greatest number of other locations once, returning to the starting location and with the shortest total distance.

Nesta user story era pedido que enquanto gestor de tráfego fosse possível encontrar o circuito mais eficiente que começa numa localização, visita o maior número de outras localizações uma única vez, retornando à mesma localização inicial com a menor distância total possível.

Para tal criámos o método mostEfficientCircuit, que cria uma Lista de LinkedLists de Posições recorrendo à utilização do algoritmo allCycles. O allCycles utiliza o algoritmo allPaths, que retorna todos os caminhos possíveis, sendo estes caminhos uma sequência alternante de vértices adjacentes e os seus ramos e que não contém nenhum ramo repetido. Em seguida, allCycles utiliza estes caminhos todos e verifica neles todos os circuitos que passam no maior número de vértices. Um circuito é um caminho fechado que não contém qualquer ramo repetido.

```java
public static String mostEfficientCircuit(Graph<Position,Double> g) {


    List<LinkedList<Position>> paths = Algorithms.allCycles(g);
    int max=0;
    double dist=Double.MAX_VALUE;
    LinkedList<Position> tempPos= null;
    for(LinkedList<Position> p: paths) {
        if (p.size() > max) {
            max = p.size();
            tempPos = p;
            dist = pathDistance(p, g);
        } else if (p.size() == max && dist < pathDistance(p, g)) {
            tempPos = p;
            dist = pathDistance(p, g);
        }
    }
    return print(tempPos);
}
public static double pathDistance (LinkedList<Position> p,Graph<Position,Double> g){
    double temp=0;
    Position tempP= p.pop();

    for (Position pos: p) {
        temp=temp+g.edge(tempP,pos).getWeight();
        tempP=pos;
    }

    return temp;
}
```

Figura 11 - Class Circuit, methods mostEfficientCircuit and pathDistance

```java
public static String print(LinkedList<Position> p) {
    String output = "";
    for (Position port : p) {
        System.out.println(port.getCountryName());
        output += port.getCountryName() + "\n";
    }
    return output;
}
```

Figura 12 - Class Circuit, method print

```java
public static <V, E> List<LinkedList<V>> allCycles(Graph<V, E> g) {

    ArrayList<LinkedList<V>> paths = new ArrayList<>();
    LinkedList<V> path = new LinkedList<>();
    for (V vOrig : g.vertices()) {
        boolean[] visited = new boolean[g.numVertices()];
        allPaths(g, vOrig, vOrig, visited, path, paths);

    }
    return paths;
}

private static <V, E> void allPaths(Graph<V, E> g, V vOrig, V vDest, boolean[] visited,
                                    LinkedList<V> path, ArrayList<LinkedList<V>> paths) {

    path.push(vOrig);
    visited[g.key(vOrig)] = true;
    for (V vAdj : g.adjVertices(vOrig)) {
        if (vAdj == vDest) {
            path.push(vDest);
            LinkedList<V> pathClone = (LinkedList<V>) path.clone();
            Collections.reverse(pathClone);
            paths.add(pathClone);

            path.pop();
        } else {
            if (!visited[g.key(vAdj)]) {
                allPaths(g, vAdj, vDest, visited, path, paths);
            }
        }
    }
    path.pop();
}
```

Figura 13 - Class Algorithms, methods allCycles and allPaths

Analise de complexidade

A complexidade da US 403:

Complexidade *print* é constituído por um ciclo for:each logo a complexidade é O(n).

Complexidade *pathDistance* é constituído por um *ciclo for:each* logo tem complexidade de O(n).

Complexidade *allCycles* é constituído por um ciclo for:each, mas como recorre ao método allPaths que tem um for:each com um if e um else, a complexidade do método fica $O(n^{*}n^{2}) = O(n^{3})$.

Complexidade *allPaths* é constituído por um ciclo for:each e um if and else com um if encadeado logo fica $O(n*n) = O(n^{2})$

Complexidade *mostEfficientCircuit* é utilizado o método allCycles para criar uma Lista de LinkedLists das posições do grafo e depois é utilizado um for:each onde se chama o método pathDistance e no final retorna-se o método print com a lista de portos. Logo a complexidade é $O(n^3) + O(n) + O(n)$, como nas somas só se conta o maior termo, a complexidade é = $O(n^3)$

Complexidade da us403 = $O(n^3)$.