ISEP / LEI
ESINF
2021/2022
G76
Autores – Jorge Ferreira (1201564) Rafael Leite (1201566) Rui Pina (1201568)

RELATÓRIO ESINF

 # Abstract #

Uma empresa de logística de transporte requer um Sistema de software capaz de gerir as suas logísticas. Esta empresa opera tanto no mar, como em terra, por vários continentes e possui diversos armazéns espalhados pelo globo.

 # Introdução #

A finalidade deste relatório é de documentar o desenvolvimento das User Stories, no âmbito de ESINF. Todos os passos percorridos para a elaboração de uma solução para os problemas apresentados, assim como a complexidade temporal de cada algoritmo adotado.

De acordo com as boas práticas aprendidas em ESOFT, no semestre passado foi adotado uma metodologia ágil de Scrum e as práticas de Programação Orientada a Objetos.

Com isto as UserStories foram divididas de maneira justa por todos os membros da equipa.

 # Responsáveis das User Stories #

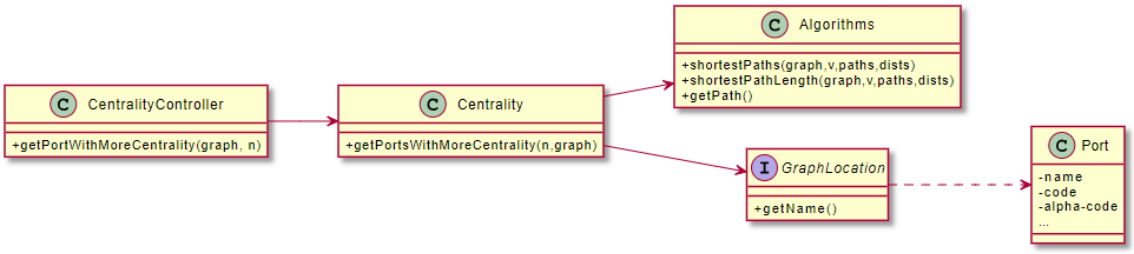
US 401 – Jorge Ferreira 1201564

US 402 – Rafael Leite 1201566

US 403 – Rui Pina 1201568

- **US401**

Class Diagram



Na classe CentralityController, é possível instanciar o método getPortsWithMoreCentrality, que tem como paramêtro um grafo, neste caso, o grafo criado na US301, e o número de Ports mais centrais a retornar. Este método retorna uma List com os n Ports desejados.

O método getPortsWithMoreCentrality da classe Centrality, é onde está o código central que permite resolver a esta user storie. Este método utiliza também o ShortestPaths, ShortestPathLenght e o getPath, da classe Algorithms, implementados neste sprint.

É possível verificar o funcionamento da US no teste do CentralityController.

- **getPortsWithMoreCentrality**

Inicialmente, os vértices do grafo são percorridos, e para cada um, é calculado o shortestPaths que retorna um ArrayList<Double> *dists*, com as distâncias mínimas correspondentes para cada vértice. Depois somamos todas as distâncias, mas, se um dos valores do *dists* for infinito (Double.Max_Value), não o consideramos, pois, significa que não existe distância mínima entre o vértice percorrido e outro vértice do grafo. No fim cada Port e a soma são colocados num Map.

```
public static List <Port> portsWithMoreCentrality(Graph<GraphLocation, Double> graph, int n) {
Map<GraphLocation, Double> network = new HashMap<>();
ArrayList<LinkedList<GraphLocation>> paths = new ArrayList<>();
ArrayList<Double> dists = new ArrayList<>();

for (GraphLocation port : graph.vertices()) {
    double sum = 0;
    Algorithms.shortestPaths(graph, port, paths, dists);
    for (int i = 0; i < dists.size(); i++) {
        if (dists.get(i) != Double.MAX_VALUE) {
            sum = sum + dists.get(i);
        }
    }
        network.put(port, sum / graph.numVertices());
}
```

Depois ordenamos o Mapa por ordem descendente de value, de forma que o primeiro elemento seja o mais central.

Por fim, retiramos os n ports mais centrais e retornamos um ArrayList com os mesmos.

```
    //sort network by descending value based on https://www.baeldung.com/java-hashmap-sort
    LinkedHashMap<GraphLocation,Double> sortedNetwork =  network.entrySet().stream()
            .sorted(Collections.reverseOrder(Map.Entry.comparingByValue()))
            .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,
                    (entry1, entry2) -> entry2, LinkedHashMap::new));


    LinkedHashMap<Port, Double> nPorts = new LinkedHashMap<>();
    int cnt = 0;

    for (Object key : sortedNetwork.keySet()) {
        if (key.getClass().equals(Port.class)) {
            if (cnt == n) {
                break;
            }
            nPorts.put((Port) key, sortedNetwork.get(key));
            cnt++;
        }
    }

    return new ArrayList<>(nPorts.keySet());
}
```

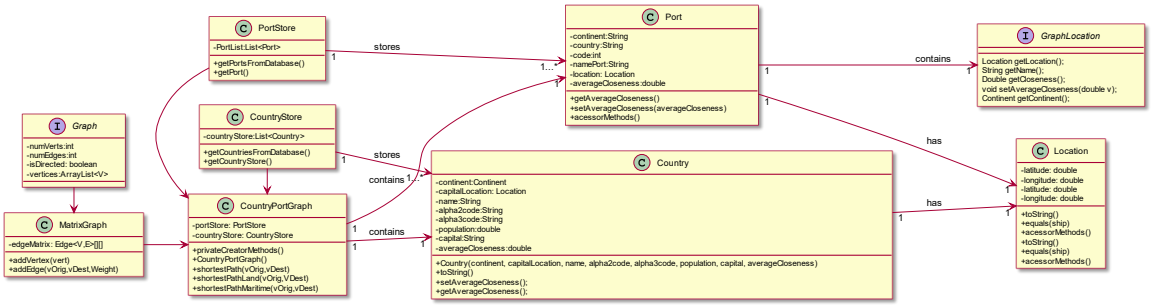
Relembro que os métodos shortestPaths, shortestPathLength e getPath da classe Algorithms também foram implementados e usados nesta US.

```
public static <V, E> boolean shortestPaths(Graph<V, E> g, V vOrig,
                    ArrayList<LinkedList<V>> paths, ArrayList<Double> dists) {
    if (!g.validVertex(vOrig)) return false;

    int nverts = g.numVertices();
    boolean[] visited = new boolean[nverts];
    /unchecked/
    V[] pathKeys = (V[]) new Object[nverts];
    Double[] dist = new Double[nverts];

    for (int i = 0; i < nverts; i++) {
        dist[i] = Double.MAX_VALUE;
        pathKeys[i] = null;
    }

    shortestPathLength(g, vOrig, visited, pathKeys, dist);
    dists.clear();
    paths.clear();

    for (int i = 0; i < nverts; i++) {
        paths.add(null);
        dists.add(Double.MAX_VALUE);
    }

    for (V vDst : g.vertices()) {
        int i = g.key(vDst);
        if (dist[i] != Double.MAX_VALUE) {
            LinkedList<V> shortPath = new LinkedList<>();
            getPath(g, vOrig, vDst, pathKeys, shortPath);
            paths.set(i, shortPath);
            dists.set(i, dist[i]);
        }
    }
    return true;
}
```

Complexidade Temporal da US: O (V x E x V x E) (*V -> vértices E -> ligações)*

- **US402**

Class Diagram



O class diagram é quase exatamente igual ao da US301 e só difere na adição de três métodos à classe CountryPortGraph (shortestPath,shortestPathLand,shortestPathMaritime), que vão dar o caminho mais pequeno de um vértice ao outro.

shortestPath

```
public List<String> shortestPath(Graph<GraphLocation, Double> g, GraphLocation vOrig, GraphLocation vDest) {
    int keyCopy = g.key(vDest);
    int size = g.numVertices();
    boolean[] visited = new boolean[size];
    int[] path = new int[size];
    double[] dist = new double[size];
    for (int i = 0; i < size; i++) {
        dist[i] = Double.MAX_VALUE;
        path[i] = -1;
        visited[i] = false;
    }
    dist[g.key(vOrig)] = 0;
    while (g.key(vOrig) != 0) {
        visited[g.key(vOrig)] = true;
        for (GraphLocation vAdj : g.adjVertices(vOrig)) {
            Edge<GraphLocation, Double> edge = g.edge(vOrig, vAdj);
            if (!visited[g.key(vAdj)] && dist[g.key(vAdj)] > dist[g.key(vOrig)] + edge.getWeight()) {
                dist[g.key(vAdj)] = dist[g.key(vOrig)] + edge.getWeight();
                path[g.key(vAdj)] = g.key(vOrig);
            }
        }
        vOrig = getVertMinDist(g,dist, visited);
    }
    List<Integer> shortestPathList= new ArrayList<>();
    shortestPathList.add(keyCopy);
    int keyToBeAdded=g.key(vDest);
    while (keyToBeAdded!=-1){
        keyToBeAdded=path[g.key(vDest)];
        shortestPathList.add(keyToBeAdded);
        vDest=g.vertex(keyToBeAdded);
    }
    shortestPathList.remove( index: shortestPathList.size()-1);
    Collections.reverse(shortestPathList);
    List<String > GraphLocationPathList= new ArrayList<>();
    for (int i=0;i<shortestPathList.size();i++){
        GraphLocationPathList.add(g.vertex(shortestPathList.get(i)).getName());
    }

    return GraphLocationPathList;
```


Este método realiza o algoritmo de Dijkstra e deposi percorre um array com o path do vértice final até ao inicial e retorna uma lista de Strings com os mesmos

shortestPathMaritime

```
if (vOrig instanceof Country|| vDest instanceof Country){
    List<String> stringList= new ArrayList<>();
    stringList.add("Countries should not be used.");
    return stringList;
}
```

Este método é igual ao shortestPath, com a diferença que não aceita países como vértices de entrada e também não os têm em conta na escolha de um caminho.

```
for (GraphLocation vAdj : g.adjVertices(vOrig)) {
    if (vAdj instanceof Port || vAdj.equals(vDest)) {
```

shortestPathLand

```
for (GraphLocation vAdj : g.adjVertices(vOrig)) {
    if (vAdj instanceof Country || vAdj.equals(vDest)) {
```

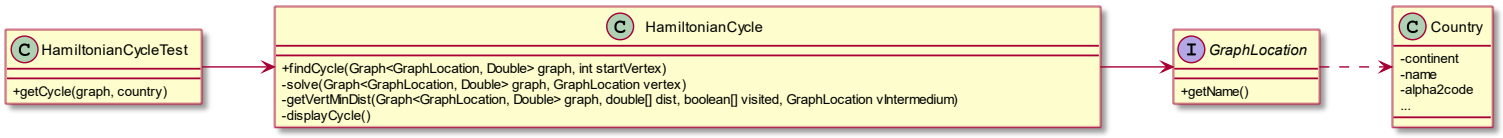
Este método é igual aos outros, mas tem apenas em contas os vértices que são paises e os vértices iniciais ou finais caso sejam portos.

Complexidade temporal:

O(nArestas*lognVertices)(complexidade do algoritmo de Dijkstra)

- **US403**

Class Diagram



Na classe HamiltonianCycleTest, é possível instanciar o método findCycle, que tem como paramêtro um grafo, neste caso, o grafo criado na US301, e o country que onde irá ser efetuado o Hamilton cycle. No fim este método irá invocar o método displayCycle que irá apresentar no ecrã o resultado obtido.

O método solve da classe HamiltonianCycle, é onde está o código central que permite resolver a esta user story. Este método utiliza também o getVertMinDist implementado neste sprint.

É possível verificar o funcionamento da US no teste do HamiltonianCycleTest.

- **Análise ao problema**

Como o problema do circuito hamiltoniano é um problema NP-Completo, ou seja não existe um algoritmo que garanta a melhor solução ao problema, nós não podemos garantir que a solução chegada neste problema seja a melhor.

Para resolver o problema utilizamos uma variação do Nearest Neighbour que por norma é utilizado para resolver o problema do caixeiro-viajante. De facto, como usamos um algoritmo guloso podemos concluir que provavelmente haveriam melhores algoritmos para resolver o problema, nomeadamente o Branch & Bound e o A*.

Porém apesar de estes algoritmos ganharem em termos de melhor solução, perdem na simplicidade do algoritmo. Ambos algoritmos foram tentados com a nossa implementação não tendo chegado a nenhum resultado. Puramente por razões de tempo resolvemos por adoptar uma variação de Nearest Neighbour, já que é o mais simples e rápidos de implementar.

- **findCycle**

Inicialmente, os vértices do grafo são percorridos, para retirar dele os vértices que são da classe Port, já que é impossível fazer um Hamilton cycle com estes. Após isso adicionamos o vértice inicial do ciclo ao cyclo (que é uma stack) e chamamos o método solve que irá resolver o problema.

```
public void findCycle(Graph<GraphLocation, Double> graph, int startVertex) {
    Graph<GraphLocation, Double> g = graph.clone();
    for (GraphLocation vertice: matrixGraph.vertices()) {
        if(vertice instanceof Port) {
            g.removeVertex(vertice);
        }
    }
    g.removeVertex(graph.vertex( key: 38));
    //add starting vertex to the list
    start = startVertex;
    cycle.push(g.vertex(startVertex));

    //start searching the path
    solve(g, g.vertex(start));

}
```

- **solve**

Este método pode ser dividido em 3 momentos lógicos:

1º momento lógico

```
private void solve(Graph<GraphLocation, Double> graph, GraphLocation vertex) {
    //Base condition: if the vertex is the start vertex
    //and all nodes have been visited (start vertex twice)

    if (graph.key(vertex) == start && cycle.size() == N + 1 && count != 0) {
        hasCycle = true;
    }

    count++;
    GraphLocation vIntermedium = vertex;

int nbr;
if (count == 1) {
    size = graph.numVertices();
    visited = new boolean[size];
    dist = new double[size];
    for (int i = 0; i < size; i++) {
        dist[i] = Double.MAX_VALUE;
        visited[i] = false;
    }
}

dist[graph.key(vertex)] = 0;
```

Neste pedaço de código iremos ver primeiramente se o já encontramos o vértice final. Se sim damos assign a true ao hasCycle, se não continuamos e se o count for 1 vamos basicamente inicializar todas as variáveis com o size adequado e iremos dar assign ao número máximo possível de representar do double, porque iremos comparar as distancias entre os vértices relativamente com este.

2ºMomento Lógico

```
while (graph.key(vIntermedium) != 0) {
    if(!foundCircuit) {

        visited[graph.key(vIntermedium)] = true;
    for (GraphLocation vAdj : graph.adjVertices(vIntermedium)) {
        Edge<GraphLocation, Double> edge = graph.edge(vIntermedium, vAdj);
        if (!visited[graph.key(vAdj)] && dist[graph.key(vAdj)] > dist[graph.key(vIntermedium)] + edge.getWeight()) {

            dist[graph.key(vAdj)] = 0;
            dist[graph.key(vAdj)] = edge.getWeight();
            //visit and add vertex to the cycle

        }

    }
```

Iremos passar para um loop em que apenas irá caso um cycle ainda não tenha sido encontrado. Caso não tenha iremos então dar assign ao vértice que estamos neste momento, que posto num exemplo em que estávamos a fazer um cycle a começar na Itália e tínhamos que o cycle até ao momento era [Itália, Eslóvenia, Croácia], o vIntermedium seria a Croácia. Iremos dar assign a true ao visited na posição do vIntermedium e iremos ver os vértices adjacentes a ele. Após isso vamos ver desses vértices aqueles que ainda não foram visitados e iremos calcular a distância dele até os vértices adjacentes que satisfazem essa condição.

3º Momento Lógico

```
vIntermedium = getVertMinDist(graph, dist, visited, vIntermedium);

nbr = graph.key(vIntermedium);
visited[nbr] = true;
cycle.push(graph.vertex(nbr));

//Go to the neighbor vertex to find the cycle
solve(graph, graph.vertex(nbr));
//Backtrack
if (graph.adjVertices(vIntermedium).contains(graph.vertex(start))) {

    hasCycle = true;
    cycle.push(graph.vertex(start));
    //output the cycle
    displayCycle();
    foundCircuit = true;
}

visited[nbr] = false;
cycle.pop();
if(foundCircuit) {
    break;
}
```

Iremos chamar o getVertMinDist, que como o nome indica iremos buscar o vértice de menor distância entre todos os vértices adjacentes ainda não visitados do vértice anterior. Após isso o novo vIntermedium será esse vértice de menor distância. Damos assign a que já foi visitado o vIntermedium e introduzimo-lo no cycle. Após isso iremos chamar recursivamente o método solve para continuarmos a descobrir o cycle.

Se o vIntermedium conter o vértice inicial como adjacente iremos terminar o ciclo, porque já encontramos o resultado pretendido. Com isso apenas damos push ao ciclo do vértice inicial e damos display ao circuito.

Caso nós nos encontremos num “dead end” iremos ter que dar backtrack até um sítio onde seja possível continuar para um vértice que ainda não tenha sido visitado. Para isso, retiramos o vértice que está no topo da stack e procuramos se temos vértices que ainda não tenham sido visitados. Caso não tenha fazemos isso sucessivamente até encontrarmos um.

Caso tenhamos encontrado o ciclo damos break ao ciclo, já que não é justificável continuarmos à procura de ciclos se já encontramos o pretendido.

- **getVertMinDist**

```
private GraphLocation getVertMinDist(Graph<GraphLocation, Double> graph, double[] dist, boolean[] visited, GraphLocation vIntermedium) {
    int copy = 0;
    double minDist = Double.MAX_VALUE;
    for (int i = 0; i < visited.length; i++) {
        if (!visited[i] && dist[i] != Double.MAX_VALUE && graph.adjVertices(vIntermedium).contains(graph.vertex(i))) {
            if (dist[i] < minDist) {
                copy = i;
                minDist = dist[i];
            }

        }
    }
    return graph.vertices().get(copy);
}
```

De entre todos os vértices adjacentes do vértice atual iremos ver qual deles tem a menor distância ao vértice atual e retornamos-lho.

- **getVertMinDist**

```
private GraphLocation getVertMinDist(Graph<GraphLocation, Double> graph, double[] dist, boolean[] visited, GraphLocation vIntermedium) {
    int copy = 0;
    double minDist = Double.MAX_VALUE;
    for (int i = 0; i < visited.length; i++) {
        if (!visited[i] && dist[i] != Double.MAX_VALUE && graph.adjVertices(vIntermedium).contains(graph.vertex(i))) {
            if (dist[i] < minDist) {
                copy = i;
                minDist = dist[i];
            }

        }
    }
    return graph.vertices().get(copy);
}
```

Apenas iremos pegar no ciclo e daremos display ao ciclo encontrado.

Complexidade Temporal da: O (V x E x V x E) (*V -> vértices E -> ligações)*