DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Relatório do Projeto de ESINF

Turma_2DE
1170500 _ Hugo Frias
1180730 _ Vera Pinto

Professor
Nuno Malheiro - NFM

Unidade Curricular
ESINF

isep Instituto Superior de Engenharia do Porto

Exercício 1

No primeiro exercício fizemos load dos dados dos ficheiros txt para o grafo. Neste trabalho, adicionamos um atributo extra “int cor” aos países, que vai ser usado no exercício 2. Cada país no ficheiro países.txt é então inserido como um vértice do grafo, enquanto que do ficheiro fronteiras.txt se extrai as edges do mesmo, em que o weight da edge é a distância entre os dois países.

```
public void loadPaises() throws FileNotFoundException, IOException {

    Set<Country> scountries = new HashSet<>();
    File directory = new File("./");
    String path = directory.getAbsolutePath();
    path = path.substring(0, path.length() - 1) + "\\src\\main\\java\\com\\mycompany\\esinf\\de\\resources\\";
    String fileNamePais = path + "paises.txt";
    List<String> list = Files.lines(Paths.get(fileNamePais)).collect(Collectors.toList());
    for (String list1 : list) {
        if (list1.length() > 0) {
            String temp[] = list1.split(",");
            Country country = new Country(-1, temp[0].trim(), temp[1].trim(), Double.parseDouble(temp[2]), temp[3].trim(),
                    Double.parseDouble(temp[4]), Double.parseDouble(temp[5]));
            g.insertVertex(country);
            scountries.add(country);

        }
    }

    String fileNameFronteira = path + "fronteiras.txt";
    List<String> listF = Files.lines(Paths.get(fileNameFronteira)).collect(Collectors.toList());
    for (String list2 : listF) {
        if (list2.length() > 0) {
            String temp2[] = list2.split(",");
            Country a = getCountry(temp2[0].trim(), scountries);
            Country b = getCountry(temp2[1].trim(), scountries);
            g.insertEdge(a, b, null, distance(a.getLatitude(), a.getLongitude(), b.getLatitude(), b.getLongitude()));
        }
    }

}
```

Exercício 2

No segundo exercício fizemos o colouring do grafo. Neste método nós preenchemos um vetor “resultado“ e também o atributo dos países “cor” com números inteiros, números esses que tinham de ser diferentes para os países fronteira do país em questão.

Este método possui um vetor “resultado” que possui o status da cor de cada vertex do grafo, sendo que -1 significa que o vértice está por colorir. De seguida criamos um vetor disponível, que nos indica se as cores nas posições do vetor estão disponíveis ou não (true or false) e implementamos um loop que para cada pais fronteira do país “vértice”, vai verificar o status das suas cores e vai então alterar o vetor disponível de modo a sabermos que cores estarão disponíveis para o país “vértice”. De seguida, procura-se a primeira cor disponível no vetor resultados, e pinta-se o país com essa cor. Voltamos a colocar o vetor “disponível” com todas as cores disponíveis, e passa-se para a próxima iteração, fazendo isto sucessivamente.

```java
public void colouring() {
        //array com o status de cada vertex (-1 caso estejam por colorir)
        int[] resultado = new int[g.numVertices()];
        //por predefinição, todos os vetores estão por colorir
        Arrays.fill(resultado, -1);
        resultado[0]=0;
        getCountryByKey(0).setCor(0);
        // array temporario para guardar as cores disponiveis.
        //Os valores ficam falsos caso algum dos vetores adjacentes
        //do vetor em questão esteja colorido
        boolean disponivel[] = new boolean[g.numVertices()];

        // Inicialmente, todas as cores estão disponiveis
        Arrays.fill(disponivel, true);

        // ciclo para atribuir cores aos vertices

        for(Country vertice : g.vertices()){
            if (vertice != getCountryByKey(0)){
               Iterable<Country> vAdjs = (Iterable<Country>) g.adjVertices(vertice);

                //verifica as cores dos vertices adjacentes e mete-as a falso
            for (Country vertex : vAdjs) {

                int j = g.getKey(vertex);
                if (resultado[j] != -1) {
                    disponivel[resultado[j]] = false;
                }
            }
                //percorre o vetor das cores até encontrar uma disponivel
                int cor;
                for (cor = 0; cor < g.numVertices(); cor++) {
                    if (disponivel[cor]) {
                        break;
                    }
                }
                //preenche o vertex com a tal cor
                vertice.setCor(cor);
                resultado[g.getKey(vertice)] = cor;

                // Reset aos valores para a proxima iteração
                Arrays.fill(disponivel, true);
            }
        }
```

Exercício 3

No exercício 3, foi-nos pedido para encontrar o shortestPath entre dois países. Para a resolução deste exercício utilizamos os algoritmos cujo pseudocódigo está presente nos powerpoints das aulas teóricas, e que foram desenvolvidos ao longo das aulas práticas. Apenas adaptamos os algoritmos para receberem objetos do tipo países.

```java
public static double shortestPath(Country cOrig, Country cDest, LinkedList<String> capitaisPassadas) {
        if (!g.validVertex(cOrig) || !g.validVertex(cDest)) {
            return 0;
        }
        int nVerts = g.numVertices();
        boolean[] visited = new boolean[nVerts];
        int[] pathKeys = new int[nVerts];
        double[] dist = new double[nVerts];
        Country[] vertices = (Country[]) g.allkeyVerts();
        for (int i = 0; i < nVerts; i++) {
            dist[i] = Double.MAX_VALUE;
            pathKeys[i] = -1;
        }
        shortestPathLength(cOrig, vertices, visited, pathKeys, dist);


        double lengthPath = dist[g.getKey(cDest)];

        if (lengthPath != Double.MAX_VALUE) {
            getPath(cOrig, cDest, vertices, pathKeys, capitaisPassadas);
            return lengthPath;
        }
        return 0;
    }
```

```java
private static void shortestPathLength(Country cOrig, Country[] vertices, boolean[] visited, int[] pathKeys, double[] dist) {
    Country aux = null;
    int key = 0;
    for (Country c : vertices) {
        if (c.getCapital().equalsIgnoreCase(cOrig.getCapital())) {
            key = g.getKey(c);
            aux = c;
        }
    }
    dist[key] = 0;
    while (key != -1) {
        visited[key] = true;
        for (Country c1 : g.adjVertices(aux)) {
            int adjKey = g.getKey(c1);
            if (!visited[adjKey] && dist[adjKey] > dist[key] + g.getEdge(aux, c1).getWeight()) {
                dist[adjKey] = dist[key] + g.getEdge(aux, c1).getWeight();
                pathKeys[adjKey] = key;
            }
        }
        key = getVertMinDist(dist, visited);
        for (Country vert : g.vertices()) {
            if (g.getKey(vert) == key) {
                aux = vert;
            }
        }
    }
}

private static int getVertMinDist(double[] dist, boolean[] visited) {
    int key = 0;
    double min = dist[0];
    for (key = 0; key < visited.length; key++) {
        if (!visited[key] && dist[key] < min) {
            min = dist[key];
            return key;
        }
    }
    return -1;
}
```

```java
private static void getPath(Country cOrig, Country cDest, Country[] vertices, int[] pathKeys, LinkedList<String> capitaisPassadas) {
    if(!cOrig.equals(cDest)) {
        capitaisPassadas.push(cDest.getCapital());
        int vKey = g.getKey(cDest);
        int prevVKey = pathKeys[vKey];
        cDest = vertices[prevVKey];
        getPath(cOrig, cDest, vertices, pathKeys, capitaisPassadas);
    } else {
        capitaisPassadas.push(cOrig.getCapital());
    }
}
```

Exercício 4

No exercício 4 foi-nos pedido algo parecido com o 3, mas só que desta vez temos de encontrar o shortestPath entre a capital origem e destino, passando por certas capitais pelo meio. Criamos alguns métodos novos para este exercício, como o convertStringListToCountries onde transformava-mos a linkedlist de strings original para uma linkedList que possui-se os países referentes ás strings da lista original, de modo a facilitar o código mais á frente.

No método findShortestPathPassingByCapitals começamos por verificar se já passamos por todas as capitais que tínhamos que passar. Em caso afirmativo, então vamos calcular o shortestPath entre

o pais origem e o pais destino, e retornar a distancia entre os mesmos (caso isto seja numa iteração futura, iremos retornar essa distância, mais a soma das distâncias entre as capitais passadas). De seguida iremos ao método orderDistancias, onde iremos ver qual a capital pela qual temos que passar é que tem o shortestPath mais próximo da capital origem. O método orderDistancias irá retornar uma lista ordenada pelas distancias, e depois num ciclo for no método findShortestPathPassingByCapitals iremos encontrar a que índex é que corresponde a menor distancia, para irmos buscar o próximo pais origem. Depois voltamos a chamar a função, até a verificação inicial ser afirmativa.

```
public static double shortestPathPassingByCapitals(String capitalOrig, String capitalDest, LinkedList<String> capitaisAPassar) {
    double distanciaTotal = 0;
    Country cOrig = getCountryByCapital(capitalOrig);
    Country cDest = getCountryByCapital(capitalDest);
    LinkedList<Country> paisesAPassar = convertStringListToCountries(capitaisAPassar);
    return findShortestPathPassingByCapitals(cOrig,cDest,paisesAPassar,distanciaTotal);
}
private static LinkedList<Country> convertStringListToCountries(LinkedList<String> path) {
    LinkedList<Country> finalPath = new LinkedList<>();
    for (String s : path){
        for (Country c : g.vertices()){
         if(c.getCapital().equalsIgnoreCase(s)){
            finalPath.push(c);
            break;
        }
    }
    }
    return finalPath;
}
public static double findShortestPathPassingByCapitals(Country cOrig, Country cDest, LinkedList<Country> paisesAPassar, double distanciaTotal) {

    ArrayList<Double> distanciasSorted = new ArrayList<>();
    LinkedList<String> pathFinal = new LinkedList<>();
    if (paisesAPassar.isEmpty()) {
        double distAux = shortestPath(cOrig, cDest, pathFinal);
        if (distAux == 0) {
            return 0;
        } else {
            return distanciaTotal + distAux;
        }
    }
    ArrayList<Double> distanciasIndex = orderDistancias(paisesAPassar, cOrig, distanciasSorted);
    for (int i = 0; i < distanciasIndex.size(); i++) {
        if (distanciasSorted.get(0).equals(distanciasIndex.get(i))) {
            distanciaTotal = distanciaTotal + distanciasSorted.get(0);
            cOrig = paisesAPassar.get(i);
            paisesAPassar.remove(i);
            break;
        }
    }
    return findShortestPathPassingByCapitals(cOrig,cDest,paisesAPassar,distanciaTotal);
}
```

```
public static ArrayList orderDistancias(LinkedList<Country> paisesAPassar, Country cOrig, ArrayList<Double> distanciasSorted) {
    double dist;
    LinkedList<String> pathAux = new LinkedList<>();
    ArrayList<Double> distanciasIndex = new ArrayList<>();
    for (int i = 0; i < paisesAPassar.size(); i++) {
        dist = shortestPath(cOrig, paisesAPassar.get(i), pathAux);
        distanciasIndex.add(dist);
        distanciasSorted.add(dist);
        pathAux.clear();
    }
    Collections.sort(distanciasSorted);
    return distanciasIndex;
}
```

Exercício 5

No exercicio 5 foi-nos pedido o maior circuito possivel sem passar por capitais já visitadas. Nós fizemos 2 versões: uma que calcula o maior circuito possivel com os países do ficheiro.txt e um com o maior circuito para um determinado país. Primeiro nós adicionamos o pais origem á lista de paises do maior circuito e criamos um vetor boolean para registar as keys dos países já visitados. No método findMaiorCircucito nós começamos por verificar se o país origem ainda tem paises fronteiras por visitar. Caso não tenha, terminamos o método e retornamos a lista com os países até então. Caso hajam países ainda por visitar, vamos então ver dos países que ainda não foram visitados, qual deles é que é o país que se encontra mais perto do pais origem. Adicionamos esse país á lista, e fazemos dele o próximo país origem. Fazemos este processo até não termos mais países por visitar, e nesse caso, “fechamos” a lista de países visitados e vamos a essa lista procurar o último país visitado que seja fronteira com o país inicial, e vamos removendo países os países (a contar do fim) até chegarmos a essa ultima fronteira

```java
public static LinkedList<Country> findMaiorCircuito(Country cOrig, boolean[] visited, LinkedList<Country> maiorCircuito){
    Iterable fronteiras =  g.adjVertices(cOrig);
    double minDist = Double.MAX_VALUE;
    Country nextCountry = null;
    int count = 0;
    int count2 = 0;
    for( Country cAdj : g.adjVertices(cOrig)){
        count2++;
        if (visited[g.getKey(cAdj)] == false){
            count++;
            break;
        }
    }
    if(count2>0 && count!=0){
    for(Country cAdj : g.adjVertices(cOrig)){
        double distance = distance(cOrig.getLatitude(), cOrig.getLongitude(), cAdj.getLatitude(), cAdj.getLongitude());
        if(distance<minDist && visited[g.getKey(cAdj)]==false){
            minDist = distance;
            nextCountry = cAdj;
        }
    }
    visited[g.getKey(nextCountry)] = true;
    maiorCircuito.add(nextCountry);
    return findMaiorCircuito(nextCountry, visited, maiorCircuito);
    } else{
        return maiorCircuito;
    }
}
private static LinkedList<Country> encontrarUltimaFronteira(LinkedList<Country> maiorCircuitoAux, Country cOrig) {
    if(maiorCircuitoAux.size()>0){
    Country aux = maiorCircuitoAux.getLast();
   if(isFronteira(cOrig, aux)){
       return maiorCircuitoAux;
   } else{
       maiorCircuitoAux.remove(maiorCircuitoAux.getLast());
       return encontrarUltimaFronteira(maiorCircuitoAux, cOrig);
   }
    }
    return null;
```

```java
 public static LinkedList maiorCircuito(){
     LinkedList<Country> maiorCircuito = new LinkedList<>();
     Country[] paises = g.allkeyVerts();
     int maior = 0;
     for (Country c : paises){
         LinkedList<Country> maiorCircuitoAux = findMaiorCircuitoDeUmPais(c);
         if(maiorCircuitoAux!=null){
         if (maiorCircuitoAux.size()>maior){
             maiorCircuito = maiorCircuitoAux;
             maior = maiorCircuitoAux.size();
         }
         }
     }
     return maiorCircuito;

 }
 public static LinkedList<Country> findMaiorCircuitoDeUmPais(Country cOrig){
     LinkedList<Country> maiorCircuito = new LinkedList<>();
     boolean[] visited = new boolean[g.numVertices()];
     maiorCircuito.add(cOrig);
     visited[g.getKey(cOrig)] = true;
     LinkedList<Country> maiorCircuitoAux = findMaiorCircuito(cOrig, visited, maiorCircuito);
     if(maiorCircuitoAux.size()>0){
        maiorCircuito = encontrarUltimaFronteira(maiorCircuitoAux, cOrig);
     }
     return maiorCircuito;
 }
```

Diagrama de Classes