Instituto Superior de Engenharia do Porto

Projeto Integrador

Sprint 1

ESINF – Licenciatura em Engenharia Informática

Relatório

Turma 2DM e 2DN

Beatriz Santos - 1211178 (2DM)
Daniela Soares - 1211229 (2DM)
Duarte Ramalho – 1221806 (2DM)
Daniel Braga – 1200801 (2DN)
Rúben Silva – 1200546 (2DM)

Índice

US301

Construir a rede de distribuição através da informação fornecida nos ficheiros.

No método “read” começamos por ler a primeira linha do ficheiro: ”clientes-produtores_small.csv”, como esta é o cabeçalho ignoramos. De seguida dividimos os valores dos ficheiros pela virgula. Se for uma empresa cria um vértice como sendo uma “Company” se não, cria como sendo um “User” e guarda tudo numa lista.

No método “readDistances” começamos por ler a primeira linha do ficheiro, como esta é o cabeçalho ignoramos depois, dividimos os valores dos ficheiros pela virgula. De seguida, Para um “User” dentro da lista, se o seu código de Localização for igual ao primeiro campo referente ao código de localização do ficheiro: “distancias_small.csv”, vai fazer a mesma comparação para o segundo campo referente ao código de localização, se for igual adiciona o ramo com o seu respetivo peso.

Análise Complexidade

O método “public void read ()” (linhas 20 a 48) é determinístico pois só existe um caso. Sendo assim este método tem complexidade de O(n), pois das linhas 28 a 44 existe um ciclo “for” com complexidade O(n) e dentro desse ciclo existem operações de complexidade O(n) e O(1) como não são ciclos não se multiplica sendo assim a soma O(n+n+1)=O(2n+1)=O(n).

```java
public void read(String filename) {
    networkGraph = new MapGraph<>(false);

    try (BufferedReader input = new BufferedReader(new FileReader(filename))){
        String line = input.readLine();
        String[] header = line.split(",");//ler primeira linha e ignorar
        System.out.println(Arrays.toString(header));

        while((line = input.readLine()) != null){
            String separator = ",";
            String[] lineFields = line.split(separator);

            Localization localization = new Localization(lineFields[0],Float.valueOf(lineFields[1]),Float.valueOf(lineFields[2]));
            if(lineFields[3].contains("E")){
                Company company = new Company(localization,lineFields[3]);
                if(!networkGraph.validVertex(company)){
                    networkGraph.addVertex(company);
                }
            }else {
                User user = new User(localization,lineFields[3]);
                if(!networkGraph.validVertex(user)){
                    networkGraph.addVertex(user);
                }
            }
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

O método “readDistances()” (linhas 50 a 76) também é determinístico existindo apenas um caso. A complexidade deste método é de O(n^3) pois existe um ciclo “while” que tem complexidade O(n) das linhas 56 a 68 onde se encontram mais 2 ciclos desta vez ciclos “for” cada um com complexidade O(n), ou seja, a sua multiplicação dá complexidade O(n^3).

```java
50  public void readDistances(String filename){
51      try (BufferedReader input = new BufferedReader(new FileReader(filename))){
52          String line = input.readLine();
53          String[] header = line.split(",");//ler primeira linha e ignorar
54          System.out.println(Arrays.toString(header));
55
56          while((line = input.readLine()) != null){
57              String separator = ",";
58              String[] lineFields = line.split(separator);
59              for(User u : networkGraph.vertices()){
60                  if(u.getLocalization().getCodLoc().equalsIgnoreCase(lineFields[0])){
61                      for(User u1: networkGraph.vertices()){
62                          if(u1.getLocalization().getCodLoc().equalsIgnoreCase(lineFields[1])){
63                              networkGraph.addEdge(u,u1,Float.valueOf(lineFields[2]));
64                          }
65                      }
66                  }
67              }
68          }
69      } catch (IOException e) {
70          throw new RuntimeException(e);
71      }
72  }
73
    6 usages  DuarteRamalho
74  public Graph<User, Float> getNetworkGraph() {
75      return networkGraph;
76  }
```

US302

Verificar se o grafo é conexo

```java
private static <V,E> void connectComps (Graph<V,E> g, ArrayList<LinkedList<V>> ccs, boolean[] visited){

    int vKey;
    LinkedList<V> conComp = new LinkedList<>();

    for (V vert : g.vertices()){
        vKey = g.key(vert);
        visited[vKey] = false;
    }

    for (V vert : g.vertices()){
        vKey = g.key(vert);
        if (!visited[vKey]) {                       //in some previous BFS
            conComp = BreadthFirstSearch(g, vert);  //discovers precisely V's
            ccs.add(conComp);                        //connected component
        }
        assert conComp != null;
        for (V v : conComp) {
            vKey = g.key(v);
            visited[vKey] = true;
        }
    }
}
```

Primeiro certificamo-nos de que todos os vértices de g ainda não foram visitados, através de um ciclo for. De seguida vamos percorrer novamente todos os vértices de g. Caso o vértice em análise ainda não tenha sido visitado, através do *BreadthFirstSearch* vamos guardar numa LinkedList todos os vértices alcançáveis através do vértice de origem. Adicionamos a lista obtida ao ArrayList css e mudamos o estado dos vértices que estavam na LinkedList para visitados. Desta forma, no css vão estar todos os conjuntos de vértices que estão ligados entre sim.

```java
//se existir algum vértice que não faz parte do arrayList conComp, então o grafo não é conexo

public static <V, E> boolean connectComps(Graph<V, E> g) {

    ArrayList<LinkedList<V>> ccs = new ArrayList<>();
    boolean[] visited = new boolean[g.numVertices()];

    connectComps(g, ccs, visited);

    /*se o primeiro elemento (LinkedList) de css tiver o número de elementos igual
    ao número de vértices de g então o grafo é conexo*/
    return ccs.get(0).size() == g.numVertices();

}
```

Assim, pegando no ArrayList css, caso a primeira LinkedList tenha o mesmo número de vértices do grafo, então significa que todos os vértices estão conectados e, consequentemente, que o grafo é conexo, sendo retornado true. Caso contrário, o método retorna false.

Análise Complexidade

O algoritmo *connectComps* é determinístico e tem complexidade de $O(V^3)$ ou $O(n^3)$.

Número mínimo de ligações

Para descobrir o número mínimo de ligações foi utilizado o método shortestPaths da classe Algotithms.

```java
/**
 * Shortest-path between a vertex and all other vertices
 *
 * @param g     graph
 * @param vOrig start vertex
 * @param ce    comparator between elements of type E
 * @param sum   sum two elements of type E
 * @param zero  neutral element of the sum in elements of type E
 * @param paths returns all the minimum paths
 * @param dists returns the corresponding minimum distances
 * @return if vOrig exists in the graph true, false otherwise
 */
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;
    }

    dists.clear();
    paths.clear();

    int numVerts = g.numVertices();
```

O shortestPaths guarda num ArrayList de LinkedList todos os caminhos mínimos entre o vértice enviado por parâmetro e todos os outros vértices do grafo.

Na classe DistributionNetWork temos o método numeroLigacoesMin. Este método vai descobrir qual é o caminho de maior comprimento que será o diâmetro do grafo.

```java
//US302
public int numeroLigacoesMin (Graph<User, Float> g){
    int diâmetro = 0;
    //LinkedList<User> pathRes = null;
    for (User v : g.vertices()){
        ArrayList<LinkedList<User>> paths = new ArrayList<>();
        ArrayList<Float> dists = new ArrayList<>();
        Algorithms.shortestPaths(g, v, Float::compare, Float::sum, zero: 0f, paths, dists);

        for (LinkedList<User> path : paths) {
            if (path.size() > diâmetro) {
                diâmetro = path.size()-1;
                //pathRes=path;
            }
        }
    }
    //System.out.println(pathRes);
    return diâmetro;
}
```

Análise Complexidade

- O método shortestPaths é determinístico e tem complexidade O(V*E).

O método numeroLigacoesMin é também determinístico e tem complexidade $O(n^3)$.

US303

Definir os hubs da rede de distribuição.

```
Map<Company,Float> map = new HashMap<>();
//US303
3 usages  Daniel Braga +1 *
public Map<Company, Float> networkHubs(int n){
    float sum;
    ArrayList<Float> dists = new ArrayList<>();

    for(User u : networkGraph.vertices()){
        sum=0;
        if(u.getClass().equals(Company.class)){
            Algorithms.shortestPaths(networkGraph,u,Float::compare,Float::sum, zero: 0F,new ArrayList<>(),dists);
            for(Float f : dists){
                sum+=f;
            }
            map.put((Company) u,sum/(networkGraph.numVertices()-1));
        }
    }
    map=map.entrySet().stream().sorted(Map.Entry.comparingByValue()).limit(n).collect(Collectors.toMap
            (Map.Entry::getKey, Map.Entry::getValue, (e1, e2)->e1, LinkedHashMap::new));

    for(Company c : map.keySet()){
        c.setHubTrue();
    }

    return map;
}
```

Para definir os hubs da rede de distribuição, é criado um *map* que vai guardar todas as empresas com a sua respetiva distância média a todos os clientes e produtores e iniciada uma variável *sum* que vai acumular o valor da distância média para cada empresa, valor este que está guardado na lista *dists*.

Inicialmente vão ser percorridos todos os utilizadores no grafo e, quando for encontrada uma empresa, será invocado o método *shortestPaths()* que retornará a distância da empresa a todos os outros vértices na lista *dists*. Todos os valores na lista serão somados e guardados no mapa juntamente com a empresa.

Por fim o *map* será cortado para apenas manter as empresas que são relevantes e nessas empresas serão definidos os hubs.

Análise Complexidade

O algoritmo é determinístico e tem complexidade O(V^3*E).

US304

Para cada cliente (particular ou empresa) determinar o hub mais próximo

```java
/** US304
 * Determine the nearest hub for each client (particular or empresarial)
 * @return clientNearestHub a map where the keys are the clients and the values are the nearest hub
 */
public Map<String, String> getClientsNearestHub (){
    Map<String, String> clientNearestHub = new HashMap<>();

    Map<String, Float> lengthPaths = new HashMap<>();

    for (User client : networkGraph.vertices()){
        for (Company hub : map.keySet()){
            Float lenPath = Algorithms.shortestPath(networkGraph,client,hub,Float::compare,Float::sum, 0F,new LinkedList<>());
            lengthPaths.put(hub.getCodUser(), lenPath);
        }
        Stream<Map.Entry<String, Float>> sorted = lengthPaths.entrySet().stream().sorted(Map.Entry.comparingByValue());
        clientNearestHub.put(client.getCodUser(), sorted.findFirst().get().getKey());
        lengthPaths.clear();
    }
    clientNearestHub=clientNearestHub.entrySet().stream().sorted(Map.Entry.comparingByKey()).collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue, (e1, e2)->e1,
                                                                                                                        LinkedHashMap::new));

    return clientNearestHub;
}
```

No método *getClientNearestHub()* em primeiro lugar é declarado o Map *clientNearestHub,* o qual irá ser retornado, tem como *key* o código de cada User, ou seja, cada cliente e os *values* são o código de cada *Company*, hub. Também é declarado o Map *lengthPaths* que tem como chave o código de cada *Company*, hub, e o tamanho do caminho de cada cliente a cada hub. A seguir são percorridos cada cliente(quer seja particular ou empresa), ou seja, cada vértice do grafo e para cada cliente, através do algoritmo de *Dijkstra*, *shortestPat(),* é determinado o tamanho do caminho do cliente a cada hub sendo, em cada iteração do segundo ciclo for ,adicionado ao lenthPath o hub e o tamanho do caminho.

```java
/** Shortest-path between two vertices
 *
 * @param g graph
 * @param vOrig origin vertex
 * @param vDest destination vertex
 * @param ce comparator between elements of type E
 * @param sum sum two elements of type E
 * @param zero neutral element of the sum in elements of type E
 * @param shortPath returns the vertices which make the shortest path
 * @return if vertices exist in the graph and are connected, true, false otherwise
 */
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) {

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

    shortPath.clear();
    int numVerts = g.numVertices();
    boolean[]visited = new boolean[numVerts];
    /unchecked/
    V [] pathKeys = (V[]) new Object[numVerts];
    /unchecked/
    E [] dist = (E[]) new Object[numVerts];
    for (int i =0; i< numVerts; i++){
        dist[i]=null;
        pathKeys[i]=null;
    }
    shortestPathDijkstra(g, vOrig, ce, sum, zero, visited, pathKeys, dist);
```

No final deste segundo ciclo, o *map* anterior é ordenado pelos *values* com o auxilio do método *sorted()*, por ordem ascendente, para se puder obter o caminho de menor tamanho. Então, antes da próxima iteração do primeiro ciclo, o código do cliente é adicionado ao *clientNearestHub* assim como o hub cujo tamanho àquele cliente é o menor que corresponde ao primeiro elemento daquele *map*. Também são removidos todos os elementos do *lengthPaths* através do clear() porque as chaves nos *maps* não podem ser repetidas, assim garantimos que para cada cliente a distância a cada hub é verdadeira. No fim é retornado para cliente o hub mais próximo através do *clientNearestHub,* o qual está ordenado pelas keys devido ao método sorted().

Análise Complexidade

Este algoritmo é determinístico e tem complexidade de O(V^3 E^2 $log(V)$).

US305

Algoritmo de Kruskall

```java
public static <V,E> Graph<V,E> kruskall (Graph<V,E> g) {
    if (!connectComps(g)) {
        return null;
    }

    Graph<V, E> mst = new MapGraph<>( directed: false);

    ArrayList<Edge<V, E>> lstEdges = new ArrayList<>();
    LinkedList<V> connectedVerts;

    V vOrig;
    V vDst;

    for (V vert : g.vertices()) {
        mst.addVertex(vert);
    }

    for (Edge<V, E> edge : g.edges()) {
        lstEdges.add(edge);
    }
```

```java
    Comparator<Edge<V, E>> c = new Comparator<Edge<V, E>>() {
        @Override
        public int compare(Edge<V, E> o1, Edge<V, E> o2) {
            if (o1.getWeight().hashCode() < o2.getWeight().hashCode()) return -1;
            else if (o1.getWeight().hashCode() > o2.getWeight().hashCode()) return +1;
            else return 0;
        }
    };

    lstEdges.sort(c);        // in ascending order of weight

    for (Edge<V, E> e : lstEdges) {
        vOrig = e.getVOrig();
        vDst = e.getVDest();
        connectedVerts = DepthFirstSearch(mst, vOrig);
        assert connectedVerts != null;
        if (!connectedVerts.contains(vDst))
            mst.addEdge(vOrig, vDst, e.getWeight());
    }

    return mst;

}
```

Primeiro verificamos se o grafo é conexo. Se não for então o grafo retornado será nulo. Se for vamos então calcular o caminho mínimo.

Começamos por adicionar todos os vértices do grafo passado por parâmetro “g” ao novo grafo (mst). De seguida vamos guardar todos as arestas de “g” num ArrayList para que posteriormente o possamos ordenar por ordem crescente de distâncias. Para o ordenar vamos usar o sort e um compare.

Vamos depois percorrer todos as arestas ordenadas através de um ciclo for. Vamos buscar o vértice de origem e o de destino da aresta. Através do DepthFirstSearch iremos obter todas as ligações que o vértice de origem já tem registada no grafo mst. Se o connectedVerts não contiver ainda o vDest vamos então adicionar a aresta ao grafo mst.

Depois de percorrermos todo o ciclo for teremos o grafo com o caminho de menor custo pronto para retornar.

Análise Complexidade

Este algoritmo é determinístico e tem complexidade de $O(V \times E^2)$ ou $O(n^3)$.