Licenciatura em Engenharia Informática



**Estruturas de Informação**

**1ª Parte**

Ano Letivo 2019/2020

Breno Pacheco – 1180005

João Ferreira – 1181436



Outubro 2020

Índice

[Diagrama de Classes 3](#_Toc21275118)

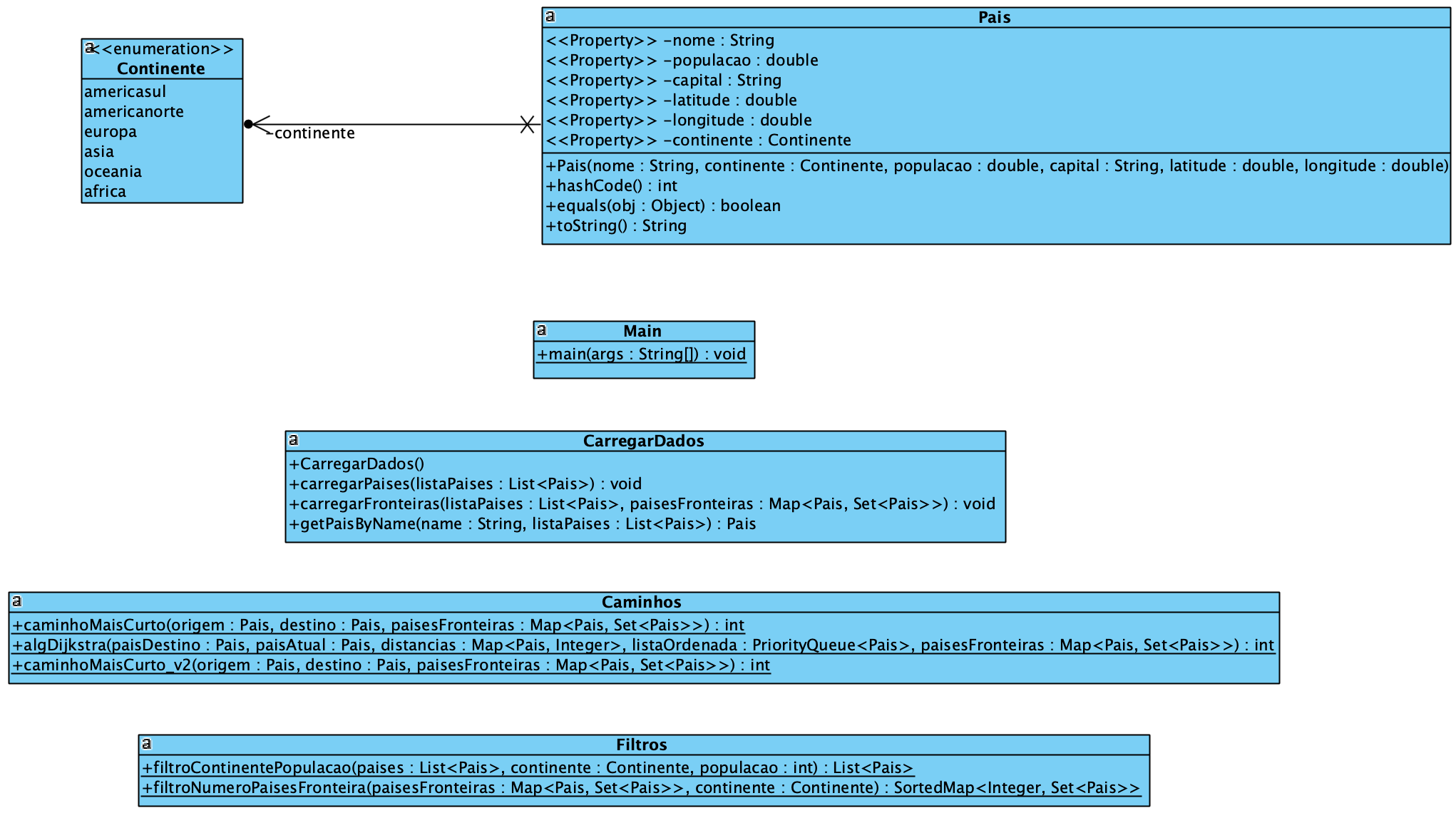
[Carregamento dos dados presentes em ficheiro 4](#_Toc21275119)

[Listagem dos países com mais do que N milhões de habitantes 5](#_Toc21275120)

[Listagem, por continente, dos países consoante o número de fronteiras 5](#_Toc21275121)

[Caminho mais curto entre dois países 5](#_Toc21275122)

# Diagrama de Classes



# Carregamento dos dados presentes em ficheiro

Para o carregamento dos dados presentes nos ficheiros optou-se por primeiro carregar os dados presentes no ficheiro com as informações relativas aos países para uma ArrayList composta por instâncias de Pais. Sendo os atributos dessa instância o nome do Pais, a instância do Continente a que pertence, a sua população, capital e as coordenadas de longitude e latitude. Para tal foi usado o método que se segue:

public void carregarPaises(List listaPaises) {

ClassLoader classloader = Thread.currentThread().getContextClassLoader();

InputStream is = classloader.getResourceAsStream("paises.txt");

Scanner scan = new Scanner(is);

while (scan.hasNextLine()) {

String[] dados = scan.nextLine().trim().split(",");

dados[1] = dados[1].trim();

Pais pais = new Pais(dados[0].trim(),

Continente.valueOf(dados[1]),

Double.parseDouble(dados[2].trim()),

dados[3].trim(),

Double.parseDouble(dados[4].trim()),

Double.parseDouble(dados[5].trim()));

listaPaises.add(pais);

}

}

Para o carregamento das fronteiras, foi criado um HashMap em que as chaves eram as instâncias de Pais previamente criadas e os seus valores são um HashSet composto pelas instâncias de Pais que fazem fronteira com a chave. Tendo o método criado para isto o seguinte:

public void carregarFronteiras(List<Pais> listaPaises, Map<Pais, Set<Pais>> paisesFronteiras) {

for (Pais p : listaPaises) {

ClassLoader classloader = Thread.currentThread().getContextClassLoader();

InputStream is = classloader.getResourceAsStream("fronteiras.txt");

Scanner scan = new Scanner(is);

Set<Pais> fronteiras = new HashSet<>();

while (scan.hasNextLine()) {

String[] dados = scan.nextLine().trim().split(",");

dados[0] = dados[0].trim();

dados[1] = dados[1].trim();

if (p.getNome().equals(dados[0]) || p.getNome().equals(dados[1])) {

Pais p1 = getPaisByName(dados[0], listaPaises);

Pais p2 = getPaisByName(dados[1], listaPaises);

if (p1 != p) {

fronteiras.add(p1);

} else {

fronteiras.add(p2);

}

}

}

paisesFronteiras.put(p, fronteiras);

}

}

# Listagem dos países com mais do que N milhões de habitantes

O exercício propõe a devolução de uma lista com países de determinado continente com mais de N milhões de habitantes, fornecidos como argumentos à uma função. É exigido que a lista retornada forneça os países ordenados pelo número de habitantes. Dessa forma, a solução adotada incluiu realizar uma iteração por todos os países carregados. Se um país cumprir com os critérios de número de habitantes e continente, este é adicionado a um Set na forma de árvore binária (implementado pelo Java Collections como TreeSet), que realiza o ordenamento natural do elemento adicionado. É fornecido um Comparator para ordenamento do TreeSet pelo número de habitantes do país. No fim da iteração os elementos do TreeSet são convertidos em uma lista simples e retornados.

public static List<Pais> filtroContinentePopulacao(List<Pais> paises, Continente continente, int populacao) {

SortedSet<Pais> setPaises = new TreeSet<Pais>(new Comparator<Pais>() {

@Override

public int compare(Pais p1, Pais p2) {

return Double.compare(p1.getPopulacao(), p2.getPopulacao());

}

});

Iterator<Pais> it = paises.iterator();

while (it.hasNext()) {

Pais pais = it.next();

if (pais.getContinente().equals(continente) && (pais.getPopulacao() >= populacao))

setPaises.add(pais);

}

return new ArrayList<Pais>(setPaises);

}

O método proposto tem ordem de complexidade temporal O(n\*log n), que representa a complexidade da adição de um elemento à lista vezes o número de elementos. Dessa forma, apresenta vantagem em complexidade se comparada ao uso de listas simples, sendo ainda de fácil aplicação para solução do problema.

# Listagem, por continente, dos países consoante o número de fronteiras

O exercício requer um método que retorne todos os países com um número N de fronteiras agrupados e relacionados à este número segundo o formato [N] -> [país A, país B...]. Como argumento do método é fornecido também o continente ao qual os países considerados devem pertencer. Adicionalmente, é exigido que os elementos da estrutura estejam ordenados de forma decrescente pelo número de fronteiras N. Dessa forma, propõe-se que a estrutura retornada seja mapa (hash) cuja chave é o número de fronteiras e o valor um Set de países.

De forma a respeitar a ordem dos elementos, o mapa escolhido é da classe TreeMap, que utiliza a interface SortedMap (implementada pelo Java Collections como árvore binária), e que realiza ordenamento natural pela chave do mapa por defeito durante adição dos elementos. É definido que o ordenamento dos elementos seja feito em ordem reversa das chaves do mapa (i.e. 5, 4, 2 …).

A solução consiste em realizar uma iteração sobre os países carregados. Se o país pertencer ao continente, a dimensão do Set de fronteiras relativo a esse país é obtida, e o país é adicionado ao SortedMap utilizando o número de fronteiras como chave.

public static SortedMap<Integer, Set<Pais>> filtroNumeroPaisesFronteira(Map<Pais, Set<Pais>> paisesFronteiras, Continente continente) {

SortedMap<Integer, Set<Pais>> map = new TreeMap<>(Collections.reverseOrder());

paisesFronteiras.forEach((pais, setFronteiras) -> {

if (continente.equals(pais.getContinente())) {

int nFronteiras = setFronteiras.size();

if (!map.containsKey(nFronteiras)) {

map.put(nFronteiras, new HashSet<Pais>());

}

map.get(nFronteiras).add(pais);

}

});

return map;

}

A solução é simples devida à forma como foram carregados os dados, pois é possível aceder ao Set de fronteiras de um país utilizando-o como chave. Além disso, o algoritmo é de boa eficiência, uma vez que possui complexidade temporal O(n\*log n), semelhante ao exercício anterior.

# Caminho mais curto entre dois países

O problema de se determinar o número mínimo de fronteiras a atravessar para chegar de um país origem a um país destino poderá ser visto como um problema de grafos. Sendo o problema traduzido por um problema de caminho mínimo num grafo.

Para a resolução deste problema optou-se por se implementar o algoritmo de Dijkstra. Este algoritmo segue os seguintes passos:

1. Atribuir 0 à distância do país de origem e infinito à distância dos restantes países;
2. Enquanto não tiver sido visitado o país de destino. Segue-se o seguinte procedimento:
   1. Visitar o país que apresenta a menor distância;
   2. Atualizar a distância dos países vizinhos para o novo valor menor, caso exista;
   3. Retirar o país onde se encontra atualmente da lista. E seguir para o país seguinte repetindo os passos anteriores.
3. Quando for atingido o país destino é retornada a sua distância.

A fim de se analisar possíveis vantagens/desvantagens do uso de recursividade para a resolução deste problema. O algoritmo foi implementado de forma iterativa e de forma recursiva.

Para a forma recursiva, foram implementados dois métodos, sendo o primeiro método o seguinte:

public static int caminhoMaisCurto(Pais origem, Pais destino, Map<Pais, Set<Pais>> paisesFronteiras) {

Set<Pais> lista = new HashSet(paisesFronteiras.keySet());

Map<Pais, Integer> distancias = new HashMap<>();

PriorityQueue<Pais> listaOrdenada = new PriorityQueue<>(new Comparator<Pais>() {

@Override

public int compare(Pais p1, Pais p2) {

if (distancias.get(p1) > distancias.get(p2)) {

return 1;

} else if (distancias.get(p1) < distancias.get(p2)) {

return -1;

} else {

return 0;

}

}

});

for (Pais pais : lista) {

if (pais.equals(origem)) {

distancias.put(pais, 0);

} else {

distancias.put(pais, Integer.MAX\_VALUE);

}

listaOrdenada.add(pais);

}

return algDijkstra(destino, listaOrdenada.poll(), distancias, listaOrdenada, paisesFronteiras);

}

E o segundo método o seguinte:

public static int algDijkstra(Pais paisDestino, Pais paisAtual, Map<Pais, Integer> distancias, PriorityQueue<Pais> listaOrdenada, Map<Pais, Set<Pais>> paisesFronteiras){

if (paisAtual.equals(paisDestino)) {

if (distancias.get(paisDestino) == Integer.MAX\_VALUE) {

return -1;

} else {

return distancias.get(paisDestino);

}

} else {

Set<Pais> paisesVizinhos = paisesFronteiras.get(paisAtual);

for (Pais p : paisesVizinhos) {

if (distancias.get(p) > (distancias.get(paisAtual) + 1)) {

listaOrdenada.remove(p);

int temp = (distancias.get(paisAtual) + 1);

distancias.put(p, temp);

listaOrdenada.add(p);

}

}

Pais paisSeguinte = (Pais) listaOrdenada.poll();

return algDijkstra(paisDestino, (Pais) paisSeguinte, distancias, listaOrdenada, paisesFronteiras);

}

}

Enquanto que a implementação da forma iterativa do algoritmo foi implementada num único método:

public static int caminhoMaisCurto\_v2(Pais origem, Pais destino, Map<Pais, Set<Pais>> paisesFronteiras) {

Map<Pais, Integer> distancias = new HashMap<>();

for (Pais pais : paisesFronteiras.keySet()) {

distancias.put(pais, Integer.MAX\_VALUE);

}

distancias.put(origem, 0);

PriorityQueue<Pais> listaPrioritara = new PriorityQueue<>(new Comparator<Pais>() {

@Override

public int compare(Pais p1, Pais p2) {

return Integer.compare(distancias.get(p1), distancias.get(p2));

}

});

listaPrioritara.add(origem);

while (!listaPrioritara.isEmpty()) {

Pais pais = listaPrioritara.poll();

Set<Pais> vizinhos = paisesFronteiras.get(pais);

for (Pais vizinho : vizinhos) {

Integer peso = 1; // distancia entre países tem peso sempre 1

if (distancias.get(vizinho) > distancias.get(pais) + peso) {

distancias.put(vizinho, distancias.get(pais) + peso);

listaPrioritara.add(vizinho);

}

}

}

if (distancias.get(destino) == Integer.MAX\_VALUE) {

return -1;

}

return (distancias.get(destino));

}

Ao se analisar o tempo de execução de cada uma das formas concluiu-se que a forma recursiva (tempo médio de execução = 5.83 ms) foi cerca de 3.43 vezes mais lenta do que a forma iterativa (tempo médio de execução = 1.70 ms). Isto pode-se dever ao facto da implementação utilizada para que a recursividade fosse implementada. Pelo que, caso a complexidade da implementação seja reduzida, é possível que os tempos de execução adotem valores mais próximos.