

Escola Superior de Tecnologia de Tomar

***"EuroTracks"* - Calculadora de Rotas**

Projeto de Estrutura de Dados e Algoritmos

Ângela Carolina da Silva Sebastião nº25946

José Luís Fernandes nº25979

Licenciatura em Engenharia Informática



RESUMO

Pretende-se com este trabalho criar uma ferramenta que auxilie um condutor de um automóvel elétrico com uma autonomia de 200km, a calcular uma rota entre duas cidades sem que chegue ao destino sem bateria. Assim é disponibilizado um ficheiro chamado worldcities.csv com o nome de 47.868 cidades de vários países e a sua localização no globo (latitudes e longitudes), como também o algoritmo de Dijkstra (soluciona o problema do caminho mais curto num grafo dirigido ou não dirigido com arestas de peso não negativo). Utilizamos Java como linguagem de programação e analisamos qual estrutura de dados teria a melhor complexidade temporal para este caso.

Palavras-chave: Java, Complexidade Temporal, Mapa, Cidades, Distância, Dijkstra.

ABSTRACT

The aim of this work is to create a tool that helps a driver of an electric car with a range of 200km, to calculate a route between two cities without reaching the destination without a battery. Thus, a file called worldcities.csv is made available with the name of 47,868 cities from various countries and their location on the globe (latitudes and longitudes), as well as the Dijkstra algorithm (solves the shortest path problem in a directed or undirected graph with edges of non-negative weight).

We used Java as a programming language and analyzed which data structure would have the best time complexity for this case.

Keywords: Java, Time Complexity , Map, Cities, Distance, Dijkstra.

ÍNDICE

Resumo.....	2
Abstract	3
Índice.....	4
Fundamentação Teórica	5
Implementação	8
Testes e Resultados	29
Conclusão	30
Referências.....	31

FUNDAMENTAÇÃO TEÓRICA

Árvores **Adelson-Velskii** e **Landis**:

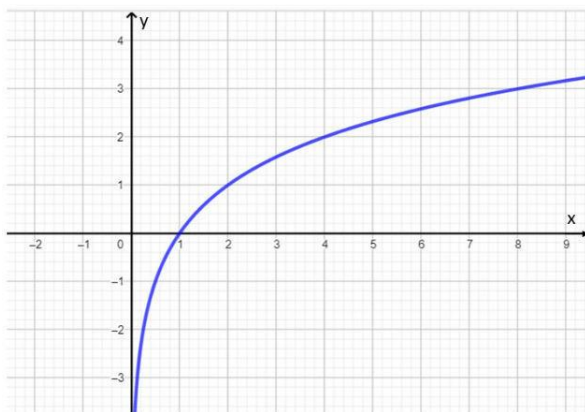
Uma árvore AVL é uma árvore de **pesquisa binária** cujos algoritmos de inserção e remoção fazem a árvore ficar sempre balanceada, ou seja não haver mais do que um nível de diferença entre a sub-árvore esquerda e a sub-árvore direita.

AVL TREE – Complexidade Temporal:

Inserção $O(\log n)$ - Remoção $O(\log n)$ - Pesquisa $O(\log n)$

Como uma complexidade $O(\log n)$ é bastante boa decidimos utilizar a AVLTree para a pesquisa das cidades num grafo.

À medida que a quantidade de dados aumenta , o tempo é proporcional a essa quantidade logo o tempo de procura será sempre mais ou menos o mesmo mesmo que tenhamos uma grande quantidade de cidades o que nos permitiu adicionar mais cidades ao nosso trabalho.



Algoritmo de Dijkstra:

O algoritmo de Dijkstra é um algoritmo de caminho mínimo usado em grafos. Ele é utilizado para encontrar o caminho mais curto entre dois vértices (neste caso, cidades), onde os pesos são associados às arestas (neste caso, distância entre as cidades).

O algoritmo funciona através da construção de uma árvore de caminho mínimo. Ele começa com um vértice inicial e, em seguida, explora todos os seus vizinhos, atualizando o custo para alcançar cada um deles. Em seguida, o algoritmo seleciona o vértice com o custo mais baixo e repete o processo. Dessa forma, ele constrói uma árvore de caminho mínimo que representa o caminho mais curto para cada vértice no grafo em relação ao vértice inicial.

Manual de Utilizador da nossa GUI:



País de partida: ComboBox para seleccionar o país da cidade em que quer começar a viagem.

Cidade de Partida: ComboBox para seleccionar cidade em que quer começar a rota (não consegue seleccionar a cidade se não tiver seleccionado o país primeiro).

País de Destino: ComboBox para selecionar o país da cidade em que quer terminar a viagem.

Cidade de Destino: ComboBox para selecionar a cidade em que quer terminar a viagem

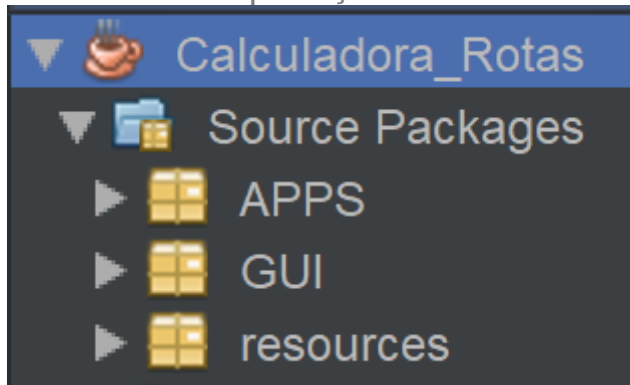
(não consegue selecionar a cidade se não tiver selecionado o país primeiro).

Botão calcular: Irá calcular a viagem mais curta entre as cidades que escolheu

Mapa: No mapa vão estar representadas (por pontos) as cidades que pass ana viagem mais curta, da inicial à final.

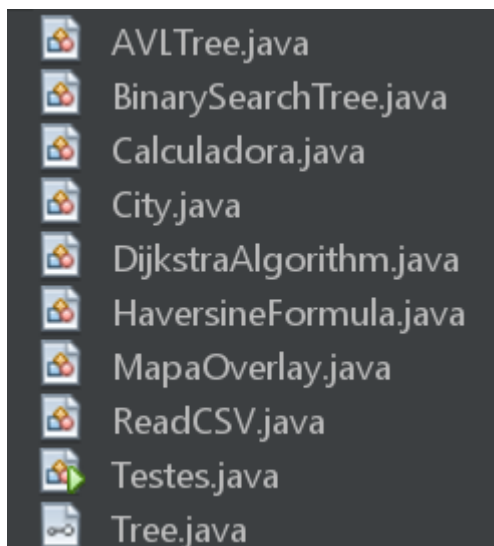
IMPLEMENTAÇÃO:

Estrutura da aplicação:



Nós temos três “packages” divididos em “APPS” (classes que utilizámos para, por exemplo ,calcular as distâncias, criar o grafo das cidades, etc...), “GUI” (contém a nossa GUI apresentada no Manual de Utilizador da nossa GUI), “resources”(contém imagens e ficheiros utilizados na codificação do problema).

APPS:



AVLTree.java:

AVLTree herda a classe BinarySearchTree que sucessivamente implementa a interface Tree. Todos eles estudados e codificados nas aulas, no entanto adicionámos alguns métodos tais como:

```
public int countVertices() {
    return countVertices(source);
}

private int countVertices(Nodo nodo) {
    if (nodo == null) {
        return 0;
    }
    return 1 + countVertices(nodo.left) + countVertices(nodo.rigth);
}
```

countVertices(Nodo nodo): decidimos fazer este método para colocar no construtor do algoritmo de Dijkstra para que ele receba o número de vertices. O método conta recursivamente os vértices da

sub-árvore esquerda e da sub-árvore direita, adicionando 1 para que conte com a raíz.

```
public ArrayList<City> getCities() {
    ArrayList<City> cities = new ArrayList<>();
    AlphabeticOrder(source, cities);
    return cities;
}
```

getCities (): decidimos fazer este método para retornar todas as cidades e as suas informações em apenas um ArrayList o que vai ajudar a implementar na GUI e o uso na classe DijkstraAlgorithm.

```
public int findCityIndex(String City) {
    ArrayList<City> cities = getCities();
    for (int i = 0; i < cities.size(); i++) {
        if (cities.get(i).getName().equals(City)) {
            return i;
        }
    }
    return -1;
}
```

findCityIndex (String City): decidimos fazer este método para retribuir o index dentro do ArrayList das cidades.

```
public ArrayList<City> getCitiesByCountry(String Country) {
    ArrayList<City> citiesByCountry = new ArrayList<>();

    ArrayList<City> cities = getCities();
    for (City city : cities) {
        if (city.getCountry().equalsIgnoreCase(Country)) {
            citiesByCountry.add(city);
        }
    }

    return citiesByCountry;
}
```

getCitiesByCountry (String Country): decidimos fazer este método para retribuir as cidades de dado país.

```
private void AlfabeticOrder(Nodo node, ArrayList<City> cities) {
    if (node != null) {
        AlfabeticOrder(node.left, cities);
        cities.add(node.data);
        AlfabeticOrder(node.right, cities);
    }
}
```

AlfabeticOrder (): decidimos fazer este método para retornar as cidades de ordem alfabética.

DijkstraAlgorithm.java:

Nós nesta classe adicionamos várias adaptações uma delas foi "loadGraph", "saveGraph", "buildGraph" para ler, criar e guardar o grafo que vai ser utilizado para os caminhos entre as cidades.

```
public void saveGraph() {
    try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("graph.ser"))) {
        out.writeObject(graph);
        System.out.println("Graph saved successfully.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
public void loadGraph() {
    File graphFile = new File("graph.ser");
    if (graphFile.exists()) {
        try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(graphFile))) {
            graph = (double[][] in.readObject());
            System.out.println("Graph loaded successfully.");
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    } else {
        System.out.println("Graph file not found. Calculating the graph.");
        buildGraph();
        saveGraph(); // Save the graph for future use
    }
}
```

```

public void buildGraph() {
    // Assuming the AVL tree has been populated with cities
    ArrayList<City> cities = avlTree.getCities(); // Get all cities from AVL tree
    if (cities.isEmpty()) {
        csv.ReadCountryFile("EUROPE");
        avlTree = csv.getAvlTree();
        cities = avlTree.getCities();
    }

    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (i != j) {
                City city1 = cities.get(i);
                City city2 = cities.get(j);

                double lat1 = Double.parseDouble(city1.getLatitude());
                double lon1 = Double.parseDouble(city1.getLongitude());
                double lat2 = Double.parseDouble(city2.getLatitude());
                double lon2 = Double.parseDouble(city2.getLongitude());

                // Set the distance between two cities using Haversine
                double distance = haversine.Distance(lon1, lat1, lon2, lat2);
                if (distance <= 200 && distance != 0) {
                    graph[i][j] = distance;
                    graph[j][i] = distance;
                }
            }
        }
    }
}

```

No loadGraph(), verificamos se graph.ser existe, se existir, lemos esse ficheiro, se não, criamos o ficheiro e depois guardamos.

No buildGraph(), vamos buscar as latitudes e longitudes do ArrayList dado pela AVLTree, caso esse ArrayList esteja vazio o ficheiro com as cidades será lido e as informações são colocadas na AVLTree, e depois são calculadas as distâncias e adicionadas ao grafo caso cumpram o requisito.

No saveGraph(), guardamos o grafo no ficheiro.

```

private String printPath(double[] distances, int[] parent,
    int source, int destination) {
    int crawl = destination;
    double totalDistance = distances[destination];

    ArrayList<String> cidades_de_passagem = new ArrayList<>();
    ArrayList<Integer> distancias = new ArrayList<>();

    while (parent[crawl] != -1) {
        int parentCity = parent[crawl];
        City cidadePartida = avlTree.getCities().get(parentCity); // Get the City object

        // Get the latitude and longitude for both cities from the AVL tree
        double latFrom = Double.parseDouble(cidadePartida.getLatitude());
        double lonFrom = Double.parseDouble(cidadePartida.getLongitude());
        City cidadeChegada = avlTree.getCities().get(crawl); // Get the destination City
        double latTo = Double.parseDouble(cidadeChegada.getLatitude());
        double lonTo = Double.parseDouble(cidadeChegada.getLongitude());

        // Calculate the distance between the cities using the Haversine formula
        double distanceBetweenCities = haversine.Distance(lonFrom, latFrom, lonTo, latTo);

        cidades_de_passagem.add(cidadePartida.getName());
        distancias.add((int) distanceBetweenCities);
        crawl = parent[crawl];
    }
}

```

```

        distancias.add((int) distanceBetweenCities);
        crawl = parent[crawl];
    }

    StringBuilder result = new StringBuilder();
    result.append("Caminho mínimo entre ").append(avlTree.getCities().get(source).getName())
        .append(" e ").append(avlTree.getCities().get(destination).getName()).append(": \n");

    for (int i = distancias.size() - 1; i >= 0; i--) {
        result.append("→").append(cidades_de_passagem.get(i)).append("(").append(distancias.get(i))
            .append("km").append(" \n");
    }

    result.append("→").append(avlTree.getCities().get(destination).getName());
    result.append("\nCusto total: ").append((int) totalDistance).append(" km");

    String resultString = result.toString();
    System.out.println(resultString);
    this.cities = cidades_de_passagem;
    return resultString;
}

```

Atualizámos também o `printPath()`, para que imprimisse na consola as cidades que precisamos de passar para chegar ao destino (`ArrayList cidades_de_passagem`), as distâncias entre as cidades e o custo total (`totalDistance`).

```
public String[] getCities(){
    return this.cities.toArray(new String[0]);
}
```

Criámos um `getCities()` para ir buscar as cidades pelas quais passamos, que estão no `ArrayList` que criámos inicialmente na classe (`cities`).

```
public String dijkstra(int source, int destination) {

    loadGraph(); // Load the precomputed graph

    double[] distances = new double[V];
    boolean[] visited = new boolean[V];
    int[] parent = new int[V];

    for (int i = 0; i < V; i++) {
        distances[i] = Integer.MAX_VALUE;
        visited[i] = false;
        parent[i] = -1;
    }
    distances[source] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minimumDistance(distances, visited);
        visited[u] = true;

        for (int v = 0; v < V; v++) {
            if (!visited[v] && graph[u][v] != 0 && distances[u] != Integer.MAX_VALUE
                && distances[u] + graph[u][v] < distances[v]) {
                distances[v] = distances[u] + graph[u][v];
                parent[v] = u;
            }
        }
    }
    return printPath(distances, parent, source, destination);
}

}
```

E atualizámos o `dijkstra` para que carregue o grafo já criado e que faça o algoritmo de dijkstra passo a passo e no final chame o `printPath`(parâmetros necessários).

HaversineFormula.java:

```
public class HaversineFormula {  
  
    double Distance(double long1, double lat1, double long2, double lat2) {  
        // Convert to radians  
        long1 = Math.toRadians(long1);  
        lat1 = Math.toRadians(lat1);  
        long2 = Math.toRadians(long2);  
        lat2 = Math.toRadians(lat2);  
  
        // Haversine formula  
        double dLat = lat2 - lat1;  
        double dLon = long2 - long1;  
  
        double sinDLat = Math.sin(dLat / 2);  
        double sinDLon = Math.sin(dLon / 2);  
        double Form1 = sinDLat * sinDLat + Math.cos(lat1) * Math.cos(lat2) * sinDLon * sinDLon;  
  
        double radius_of_earth = 6378.1; // Earth's radius in kilometers  
        double distance = 2 * radius_of_earth * Math.asin(Math.sqrt(Form1));  
        return distance;  
    }  
}
```

Criamos uma classe chamada HaversineFormula, que calcula a distância em linha reta entre duas cidades utilizando as suas longitudes e latitudes.

ReadCSV.java:

Esta classe é uma das mais importantes no nosso código porque é onde lemos o ficheiro worldcities.csv, e criamos o ficheiro EUROPE.txt com apenas países da europa.

```
public ReadCSV() {
    if (cities.isEmpty()) { // Only read the file once
        String filePath = "worldcities.csv";
        String line;

        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            while ((line = br.readLine()) != null) {
                String[] values = line.split(",");
                if (values.length >= 4) {
                    cities.add(values[1].replace("\"", "").trim());
                    latitudes.add(values[2].replace("\"", "").trim());
                    longitudes.add(values[3].replace("\"", "").trim());
                    pais.add(values[4].replace("\"", "").trim());
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Com isto, conseguimos separar o worldcities.csv em "cities", "longitudes", "latitudes" e "pais". Como estas categorias estão separadas no ficheiro por virgulas foi fácil conseguir separar a informação.

Posteriormente num método chamado writeCitiesByCountry() é onde selecionamos que países queremos guardar no novo ficheiro EUROPE.txt, aqui está um exemplo de como fomos buscar dois países da europa:


```
ArrayList<Integer> index_paises = new ArrayList<Integer>();

int Austria = findIndexCountry("Austria");
int Belgium = findIndexCountry("Belgium");
```

```
String fileName = "EUROPE.txt";
File file = new File(fileName);

try {
    // Check if the file exists and skip writing if it does
    if (file.exists()) {
        System.out.println("File already exists. Skipping writing.");
        return; // Exit the method if the file exists
    } else {
        file.createNewFile(); // Create the file if it doesn't exist
    }

    // Write data to the file
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(file))) {
        for (int i = 0; i < index_paises.size(); i++) {
            for (int j = 1; j < cities.size(); j++) {
                if (pais.get(index_paises.get(i)).equals(pais.get(j))) {
                    writer.write("\"" + cities.get(j) + "\",");
                    writer.write("\"" + latitudes.get(j) + "\",");
                    writer.write("\"" + longitudes.get(j) + "\",");
                    writer.write("\"" + pais.get(j) + "\"");
                    writer.newLine();
                }
            }
        }

    } catch (IOException e) {
        System.err.println("Error writing file for countries");
        e.printStackTrace();
    }

} catch (IOException e) {
    System.err.println("Error checking/creating file");
    e.printStackTrace();
}
```

Assim, guardamos estes novos valores que escolhemos(cidade, latitudes, longitudes e pais) no novo ficheiro separados também por virgulas.

```

public void ReadCountryFile(String Country) {

    String fileName = Country + ".txt";
    String line;

    try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
        while ((line = br.readLine()) != null) {

            String[] values = line.split(",");

            if (values.length == 4) {
                String name = values[0].replace("\"", "").trim();
                String lat = values[1].replace("\"", "").trim();
                String log = values[2].replace("\"", "").trim();
                String country = values[3].replace("\"", "").trim();

                City city = new City(name, lat, log, country);
                avlTree.add(city); // Insert city into the AVL Tree
            }

        }
    } catch (IOException e) {
        e.printStackTrace();
    }

}

```

Criámos um método chamado ReadCountryFile() que vai buscar

apenas os dados que estão no EUROPE.txt e não no worldcities.csv, que vai adicionar objetos "city" à AVLTree com as informações do nome da cidade, da sua latitude, da sua longitude e qual o seu país.

Nesta classe temos vários getters e mais um método que nós criámos :

```

public int findIndexCountry(String country) {
    int index = -1;
    for (int i = 0; i < pais.size(); i++) {
        if (pais.get(i).equals(country)) {
            index = i;
            break; // Once found, exit the loop
        }
    }
    return index; // Returns -1 if the country is not found
}

```

Criámos este método para nos ajudar na criação do ficheiro com as cidades da Europa.

Este é o contrutor da classe "City" que é onde ficaram armazenadas as informações de cada cidade.

```
public City(String name, String latitude, String longitude, String country) {  
    this.name = name;  
    this.latitude = latitude;  
    this.longitude = longitude;  
    this.country = country;  
}
```

MapaOverlay.java:

Para que na nossa GUI apareça os pontos das respetivas cidades que passamos para chegar ao destino tivemos de criar uma

classe para que conseguíssemos desenhar no ecrã, por cima da imagem.

A imagem utilizada como mapa na GUI teve de ser esta em específico pois, com ela, vem uma fórmula para transformar latitudes e longitudes em píxeis e assim saber onde desenhar os pontos no mapa.

```
public static int[] LatLonToPixels(String latStr, String lonStr, int mapWidth, int mapHeight) {
    try {
        // Convert input strings to double
        double latitude = Double.parseDouble(latStr);
        double longitude = Double.parseDouble(lonStr);

        // Validate latitude and longitude
        if (latitude > 85.0511 || latitude < -85.0511) {
            throw new IllegalArgumentException("Latitude must be between -85.0511 and 85.0511 degrees.");
        }
        if (longitude > 180 || longitude < -180) {
            throw new IllegalArgumentException("Longitude must be between -180 and 180 degrees.");
        }

        // Longitude to X (linear mapping)
        int x = (int) (((longitude + 180) / 360) * mapWidth);

        // Latitude to Y (using Mercator formula)
        double latRad = Math.toRadians(latitude); // Convert latitude to radians
        double mercatorY = Math.Log(Math.tan(Math.PI / 4 + latRad / 2));
        int y = (int) ((1 - mercatorY / Math.PI) * (mapHeight / 2)) + 17;

        return new int[]{x, y};
    } catch (NumberFormatException e) {
        throw new IllegalArgumentException("Invalid latitude or longitude format. Must be a valid number.", e);
    }
}
```

Recebemos as latitudes, longitudes, largura do mapa e comprimento do mapa, e confirmamos se as latitudes e longitudes estão corretas. Se estiverem, convertemos para píxeis.

Retornamos esses valores e desenhamos os pontos laranjas com o método `draw(Graphics g)`:

```
public void draw(Graphics g) {

    for (int i = 0; i < cities.size(); i++) {
        City city = cities.get(i);
        int[] cityPixels = LatLonToPixels(city.getLatitude(), city.getLongitude(), mapWidth, mapHeight);

        // Draw city as a red circle
        g.setColor(Color.ORANGE);
        g.fillOval(cityPixels[0] + xOffset, cityPixels[1], 7, 7);

        // Draw connections between consecutive cities
    }
}
```

Para que a GUI calculasse os novos valores a partir dos escolhidos manualmente nas ComboBoxes, criámos uma classe

chamada Calculadora com um método chamado `GUI_Response()`:

```

private String[] cities;

public String GUI_Response(String cidade_partida, String cidade_destino, AVLTree avlTree) {

    int source = avlTree.findCityIndex(cidade_partida);
    int destination = avlTree.findCityIndex(cidade_destino);

    ArrayList<City> city = avlTree.getCities();

    DijkstraAlgorithm dj = new DijkstraAlgorithm(city.size());
    String result = dj.dijkstra(source, destination);

    this.cities = dj.getCities();
    return result;
}

public String[] getCities() {
    return this.cities;
}

```

Calculamos o caminho baseado nos parâmetros que enviâmos, e retornamos o resultado para a GUI.

Na GUI nós criámos estas variáveis:

```
private MapaOverlay overlay;
private ReadCSV csv;
private AVLTree avlTree;
private ArrayList<String> cidades_partida;
private ArrayList<String> países_partida;
private ArrayList<String> cidades_destino;
private ArrayList<String> países_destino;
private City SelectedCityPartida;
private City SelectedCityDestino;
private String SelectedCountryPartida;
private String SelectedCountryDestino;
```

E também pelo construtor, criamos o ficheiro, lemos o ficheiro e guardamos a AVLTree com as informações, e carregamos as cidades e países.

```
public GUI() {
    csv = new ReadCSV();
    csv.writeCitiesByCountry();
    csv.ReadCountryFile("EUROPE");
    avlTree = csv.getAvlTree();

    // Initialize city and country lists
    loadCityAndCountryLists();

    // Initialize GUI components
    initComponents();
}
```

Para as comboBoxes criámos métodos de actionPerformed para cada uma delas:

```

private void cidadePartidaActionPerformed(java.awt.event.ActionEvent evt) {
    String cidadeNome = cidadePartida.getSelectedItem().toString();
    SelectedCityPartida = avlTree.getCities().get(avlTree.findCityIndex(cidadeNome));
}

private void cidadeDestinoActionPerformed(java.awt.event.ActionEvent evt) {
    String cidadeNome = cidadeDestino.getSelectedItem().toString();
    SelectedCityDestino = avlTree.getCities().get(avlTree.findCityIndex(cidadeNome));
}

```

Para as cidades, nós só tínhamos de ler a cidade escolhida na ComboBox e procurar por ela no ficheiro. Criámos então, um método que em que atualizamos

SelectedCityPartida/SelectedCityDestino com os novos valores da cidade selecionada.

```

private void paisPartidaActionPerformed(java.awt.event.ActionEvent evt) {
    updateCitiesByCountry(paisPartida, cidadePartida, this.cidades_partida);
}

private void paisDestinoActionPerformed(java.awt.event.ActionEvent evt) {
    updateCitiesByCountry(paisDestino, cidadeDestino, this.cidades_destino);
}

```

Para os países, tínhamos de ler o país selecionado na ComboBox e atualizar a ComboBox das cidades (respetiva para cada situação, partida ou destino) para que apresentassem apenas cidades daquele país.

```
private void updateCitiesByCountry(JComboBox<String> countryComboBox, JComboBox<String> cityComboBox, ArrayList<String> citiesList) {
    String selectedCountry = countryComboBox.getSelectedItem().toString();
    ArrayList<City> citiesByCountry = avlTree.getCitiesByCountry(selectedCountry);

    citiesList.clear();
    for (City city : citiesByCountry) {
        citiesList.add(city.getName());
    }

    DefaultComboBoxModel<String> model = new DefaultComboBoxModel<>(citiesList.toArray(new String[0]));
    cityComboBox.setModel(model);
    cityComboBox.revalidate();
    cityComboBox.repaint();
}
```

Fizemos então este método que recebe as informações das comboBoxes dos países e atualiza então as comboBoxes das cidades.

Para atualizar as JTextArea onde vai aparecer o caminho desejado, criámos um método onde , lemos o ficheiro EUROPE.txt, calculamos

o resultado com Calculadora.GUI_Response, guardamos as cidades e as adicionamos num citiesList com nome , latitude e longitude, que

depois é usado para desenhar os pontos no mapa.

```
public void atualizarTexto() {
    Calculadora calc = new Calculadora();
    csv = new ReadCSV();
    csv.ReadCountryFile("EUROPE");

    // Calculate result based on selected cities
    String result = calc.GUI_Response(
        cidadePartida.getSelectedItem().toString(),
        cidadeDestino.getSelectedItem().toString(),
        avlTree
    );

    String[] cities = calc.getCities();
    ArrayList<City> avl_cities = avlTree.getCities();
    ArrayList<City> citiesList = new ArrayList<>();

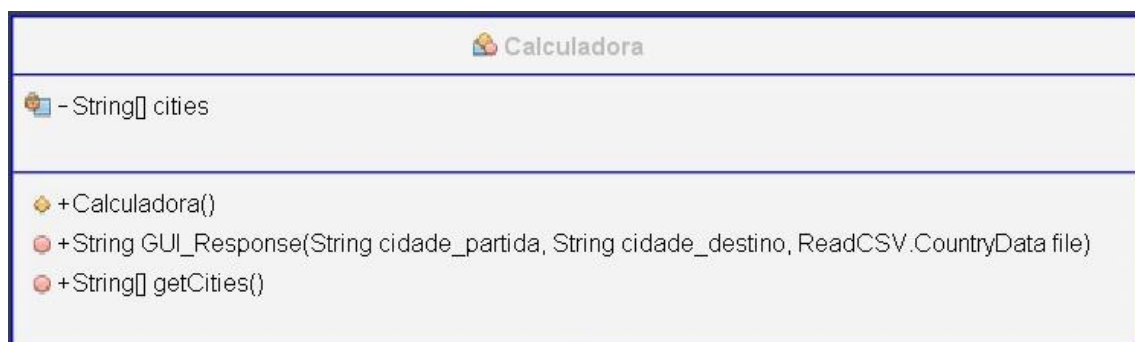
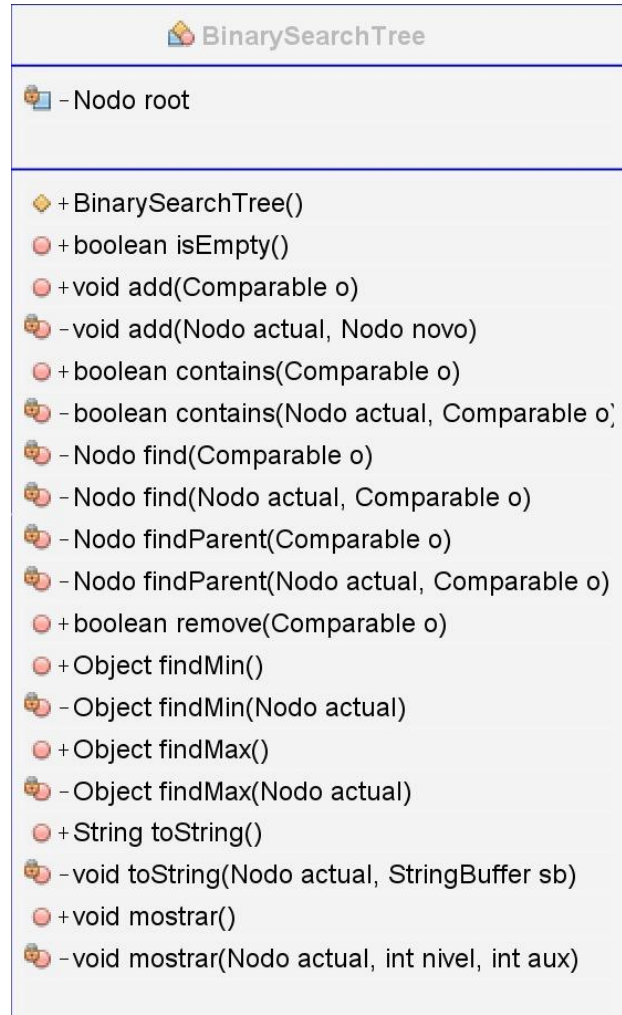
    for (int i = 0; i < cities.length; i++) {
        String city = cities[i];
        int cityIndex = avlTree.findCityIndex(city);
        citiesList.add(new City(city, avl_cities.get(cityIndex).getLatitude(), avl_cities.get(cityIndex).getLongitude(), ""));
    }

    this.overlay = new MapaOverlay(citiesList);







    jTextArea2.setWrapStyleWord(true);
    jTextArea2.setLineWrap(true);
    jTextArea2.setEditable(false);
    jTextArea2.setText(result); // Update the text area









    jPanel3.revalidate(); // Revalidate the panel
    jPanel3.repaint();
    this.repaint(); // Repaint the entire GUI
}
```


DIAGRAMAS DE CLASSES:



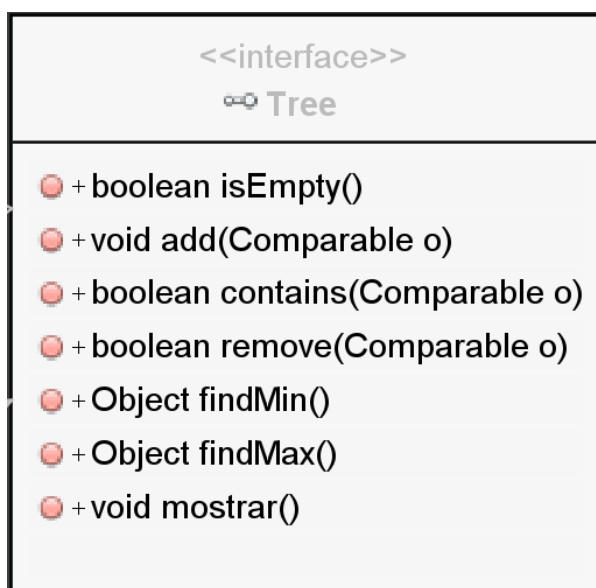
DijkstraAlgorithm

 - int V
 - ArrayList<String> cities
 - double[][] graph
 ~ReadCSV csv
 - AVLTree avlTree
 ~HaversineFormula haversine

 +DijkstraAlgorithm(int vertices)
 - int minimumDistance(double[] distances, boolean[] visited)
 - String printPath(double[] distances, int[] parent, int source, int destination)
 + String[] getCities()
 + void saveGraph()
 + void loadGraph()
 + void buildGraph()
 + String dijkstra(int source, int destination)

HaversineFormula

 ~double Distance(double long1, double lat1, double long2, double lat2)



 GUI

-  -MapaOverlay overlay
-  -ReadCSV csv
-  -AVLTree avlTree
-  -ArrayList<String> cidades_partida
-  -ArrayList<String> paises_partida
-  -ArrayList<String> cidades_destino
-  -ArrayList<String> paises_destino
-  -City SelectedCityPartida
-  -City SelectedCityDestino
-  -String SelectedCountryPartida
-  -String SelectedCountryDestino
-  -javax.swing.JComboBox<String> cidadeDestino
-  -javax.swing.JComboBox<String> cidadePartida
-  -javax.swing.JButton jButton1
-  -javax.swing.JComboBox<String> jComboBox1
-  -javax.swing.JComboBox<String> jComboBox2
-  -javax.swing.JInternalFrame jInternalFrame1
-  -javax.swing.JLabel jLabel1
-  -javax.swing.JLabel jLabel2
-  -javax.swing.JLabel jLabel3
-  -javax.swing.JLabel jLabel4
-  -javax.swing.JLabel jLabel5
-  -javax.swing.JLabel jLabel7
-  -javax.swing.JLabel jLabel8
-  -javax.swing.JPanel jPanel1
-  -javax.swing.JPanel jPanel3
-  -javax.swing.JScrollPane jScrollPane1
-  -javax.swing.JScrollPane jScrollPane2
-  -javax.swing.JTextArea jTextArea1
-  -javax.swing.JTextArea jTextArea2
-  -javax.swing.JComboBox<String> paisDestino
-  -javax.swing.JComboBox<String> paisPartida

```

+ GUI()
-void loadCityAndCountryLists()
+void paint(Graphics g)
+void atualizarTexto()
// <editor-fold defaultstate="collapsed" desc="Generated Code"> //GEN-BEGIN: initComponents void initComponents()
-void cidadePartidaActionPerformed(java.awt.event.ActionEvent evt)
-void cidadeDestinoActionPerformed(java.awt.event.ActionEvent evt)
-void paisPartidaActionPerformed(java.awt.event.ActionEvent evt)
-void paisDestinoActionPerformed(java.awt.event.ActionEvent evt)
-void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
-void updateCitiesByCountry(JComboBox<String> countryComboBox, JComboBox<String> cityComboBox, ArrayList<String> citiesList)
+static void main(String args)
    
```

TESTES E RESULTADOS:

Durante a criação do trabalho passámos por duas fases distintas.

1. Tentámos fazer o trabalho utilizando apenas ArrayLists, e apenas usámos as cidades de Portugal e o resultado era :

```
run:
Shortest path from Tomar to Porto: Tomar ->
Total cost: 169
BUILD SUCCESSFUL (total time: 17 seconds)
```

Só para irmos de Tomar para o Porto demorávamos 17 segundos, apenas a ler o caminho.

2. Decidimos implementar a AVLTree no nosso trabalho devido à sua complexidade temporal e os resultados foram:

Para a criação do ficheiro "EUROPA.txt" e do grafo :

```
Graph file not found. Calculating the graph.
Graph saved successfully.
Caminho minimo entre Porto e Tomar:
?Porto(174km)
?Tomar
Custo total: 174 km
BUILD SUCCESSFUL (total time: 17 seconds)
```

Apenas a criação do grafo:
Podemos concluir que o trabalho
De criação do ficheiro .txt não
Impacta severamente o tempo

```
File already exists. Skipping writing.
Graph file not found. Calculating the graph.
Graph saved successfully.
Caminho minimo entre Porto e Tomar:
?Porto(174km)
?Tomar
Custo total: 174 km
BUILD SUCCESSFUL (total time: 17 seconds)
```

No entanto se apenas lermos o grafo...

```
File already exists. Skipping writing.  
Graph loaded successfully.  
Caminho minimo entre Porto e Tomar:  
?Porto(174km)  
?Tomar  
Custo total: 174 km  
BUILD SUCCESSFUL (total time: 2 seconds)
```

Demoramos 2 segundos para ir de Tomar para o Porto.

CONCLUSÃO:

ipt.pt



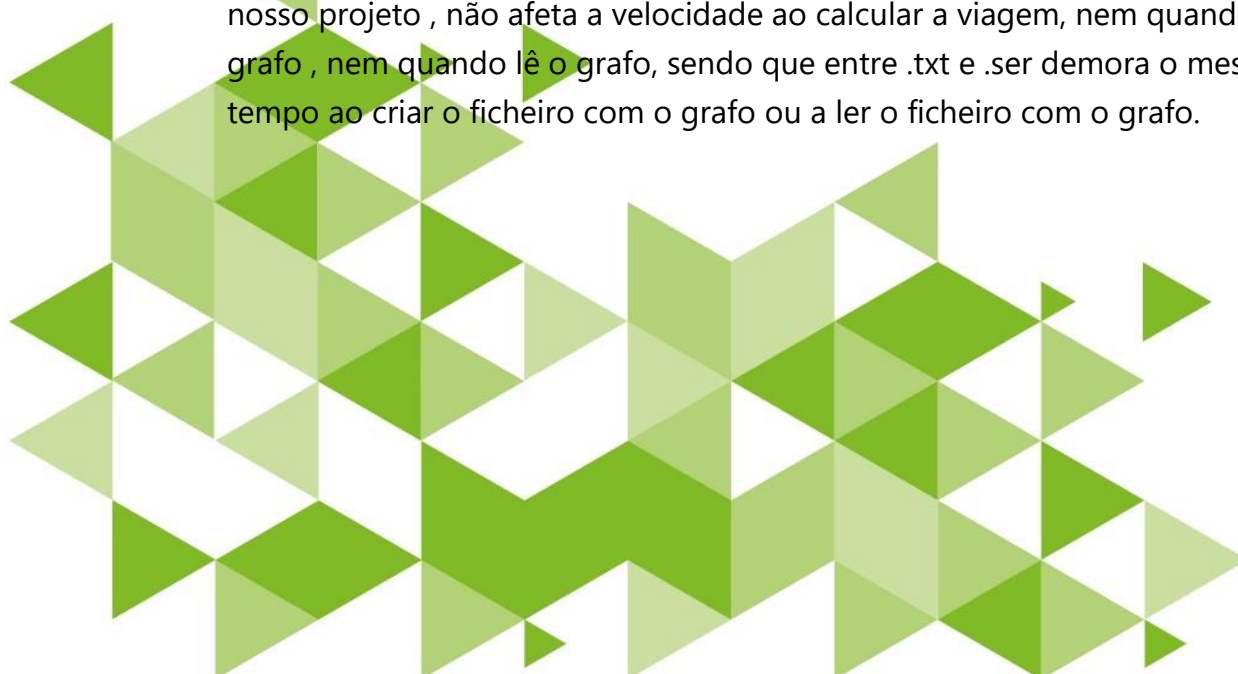
Com este trabalho aprendemos a melhorar a nossa utilização de estruturas de dados de maneira a gerenciar os dados de forma mais eficiente.

Também aprendemos a pensar de forma construtiva e intuitiva de maneira a encontrar caminhos mais eficientes do que outros.

Aprendemos a criar ficheiros serial.

Melhoramos o nosso conhecimento em ficheiros, como por exemplo, ler do ficheiro certas informações separadas por caracteres.

Criámos um ficheiro com um objeto "graph" que representa as distâncias entre as cidades, e como podemos ver, o tipo de ficheiro em que fica guardado, no nosso projeto, não afeta a velocidade ao calcular a viagem, nem quando cria o grafo, nem quando lê o grafo, sendo que entre .txt e .ser demora o mesmo tempo ao criar o ficheiro com o grafo ou a ler o ficheiro com o grafo.



REFERÊNCIAS:

<https://ezgif.com/>

-Usado para o tratamento de imagens.

<https://www.todamateria.com.br/paises-da-europa/>

-Usado para confirmar os países da europa.

<https://stackoverflow.com/questions/13171791/how-can-i-print-the-distance-between-2-cities-given-their-latitude-and-longitud>

-Um exemplo de implementação da fórmula de Haversine.

<https://www.geeksforgeeks.org/haversine-formula-to-find-distance-between-two-points-on-a-sphere/>

-Mais um exemplo de implementação da fórmula de Haversine.

<https://www.vcalc.com/wiki/vcalc/haversine-distance>

-Usado para verificar se os cálculos de Haversine estavam corretos.

<https://www.calcmaps.com/pt/map-distance/>

-Usado para verificar se a distância das contas estava de acordo com a distância nesse mapa 2D de forma a sabermos se estávamos a retirar e a usar as informações do ficheiro de forma correta.

<https://stackoverflow.com/questions/45026140/java-code-to-split-csv-file-into-different-csv-files-and-extracting-a-single-col>

-Exemplo de como separar a informação de um ficheiro .csv .

<https://www.baeldung.com/java-csv-file-array>

-Exemplo de como colocar as informações de um ficheiro .csv em um ArrayList.