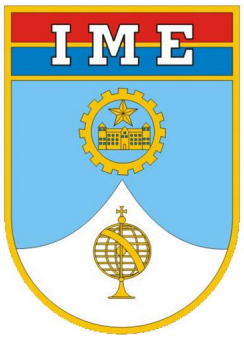


MINISTÉRIO DA DEFESA
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
(REAL ACADEMIA DE ARTILHARIA, FORTIFICAÇÃO
E DESENHO, 1792)

Laboratório de Programação II

Trabalho 1 - Caminho mínimo entre Aeroportos

Rafael Cavalcante **Timbó**

Rio de Janeiro, 2022

1 Introdução

O programa a seguir possui como finalidade determinar o trajeto mínimo entre quaisquer dois aeroportos internacionais localizados no Brasil, de modo que haja ao menos uma escala no trajeto. Devido à desigualdade triangular, segue de maneira direta que o caminho mínimo sempre corresponderá ao caminho com uma escala apenas. Assim, buscamos sempre um roteiro do tipo *"PARTIDA* $\rightarrow$ *ESCALA* $\rightarrow$ *CHEGADA"*.

Assim, a presente implementação do código do programa, feita em Java, possui 8 classes cujo funcionamento e aplicações serão explicadas individualmente, sendo elas:

- Aeroportos
- Trajetos
- Grafo
- Dijkstra
- Rotas
- Conexão
- Menu
- Principal

2 Aeroportos

A classe Aeroportos é responsável por criar os objetos que representarão os aeroportos internacionais brasileiros. Este tipo de objeto possuirá os seguintes atributos: String Nome, String Estado, double latitude, double longitude, inteiro índice, booleano explorado, lista de trajetos saídas.

Nome e estado serão utilizados para seleção do aeroporto e impressão na tela. Latitude e Longitude serão utilizados para Cálculo de distância entre aeroportos. índice é utilizado para facilitar a manipulação dos aeroportos nos algoritmos de Dijkstra e de seleção de aeroportos. Lista de saídas é utilizado para construção do Grafo e implementação do algoritmo de Dijkstra.

```java
package Principal;
import java.util.ArrayList;
19 usages
public class Aeroportos {
    //Atributos:
    3 usages
    private String nome;
    3 usages
    private String estado;
    3 usages
    private  double Lat;
    3 usages
    private  double Long;
    2 usages
    private final ArrayList<Trajetos> saidas;
    3 usages
    private  int indice;
    3 usages
    private boolean explorado;

    //Construtor:
    1 usage
    public Aeroportos() {
        Lat=0;
        Long=0;
        saidas = new ArrayList<>();
        nome = "";
        indice=-1;
        explorado = false;
        estado = "";
    }
```

Além dos atributos e do Construtor, esta classe possui métodos Getter and Setter:

```java
//Métodos Getter and Setter:
4 usages
public String getEstado() { return estado; }
1 usage
public void setEstado(String estado) { this.estado = estado; }
12 usages
public String getNome() { return nome; }
1 usage
public void setNome(String nome) { this.nome = nome; }
4 usages
public double getLat() { return Lat; }
1 usage
public void setLat(double lat) { Lat = lat; }
4 usages
public int getIndice() { return indice; }
1 usage
public void setIndice(int indice) { this.indice = indice; }
2 usages
public double getLong() { return Long; }
1 usage
public void setLong(double aLong) { Long = aLong; }
2 usages
public ArrayList<Trajetos> getSaidas() {return saidas;
}
2 usages
public boolean foiExplorado() {return explorado;}
1 usage
public void setExplorado(boolean explorado) { this.explorado = explorado; }
```

E também um método para calcular a distância entre dois aeroportos:

```
62      //Método par cálculo de distância entre aeroportos:
        6 usages
63  @   public double getDist(Aeroportos aero){
64
65          double dist = Math.pow(Math.sin((this.getLat() - aero.getLat()) / 2), 2);
66
67          dist+=Math.cos(this.getLat()) * Math.cos(aero.getLat()) * Math.pow(Math.sin((this.getLong() - aero.getLong()) / 2), 2)
68
69          dist = 2 * 6371 * Math.asin(Math.sqrt(dist));
70
71          return dist;
72      }
73  }
```

O cálculo da Distância é feito utilizando a fórmula de Haversine:

$$D = 2R \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta Lat}{2}\right) + \cos(Lat_1)\cos(Lat_2)\sin^2\left(\frac{\Delta Lon}{2}\right)}\right)$$

3 Trajetos

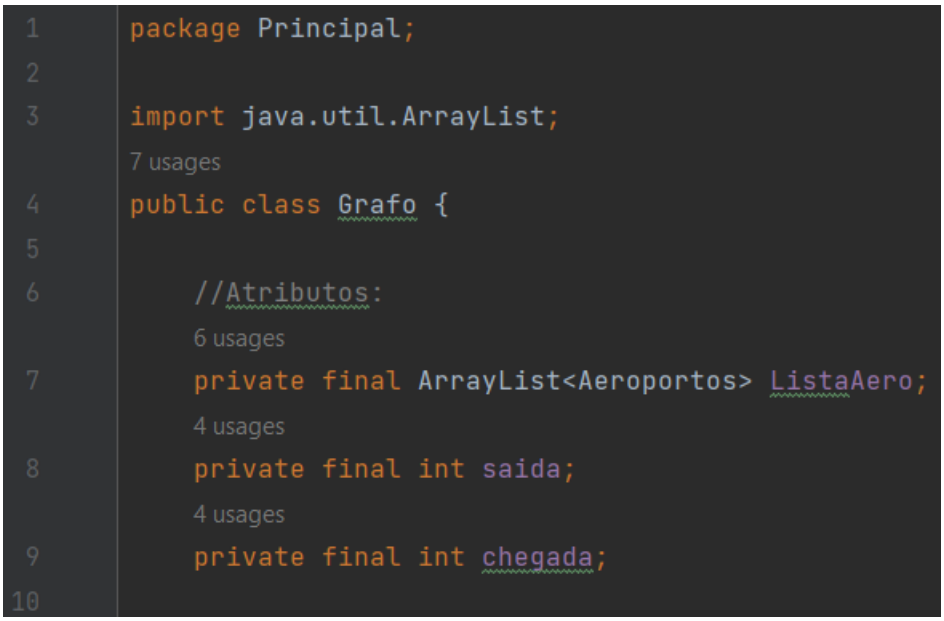
Esta classe cria objetos que comporão a lista de Saídas de um aeroporto. Ou seja, se existe o trajeto entre os aeroportos *Aero 1* e *Aero 2*, existirá, por exemplo, um Trajeto *Traj 12* na lista de saída de *Aero 1*, em que o aeroporto instanciado no *Traj 12* será o *Aero 2*.

```java
package Principal;

public class Trajetos {

    //Atributos:
    2 usages
    private final Aeroportos chegada;

    //Construtor:
    1 usage
    public Trajetos(Aeroportos chegada) {
        this.chegada = chegada;
    }

    //Método Getter:
    6 usages
    public Aeroportos getChegada() { return chegada; }

}
```

Além disso, podemos ter acesso ao aeroporto de chegada de um trajeto pelo método Getter.

4 Grafo

Classe que gera o objeto Grafo, que possui como atributos uma lista de Aeroportos e os aeroportos de saída e chegada. Assim, seu construtor cria as arestas (Trajetos), entre os vértices (Aeroportos), de modo que todos os aeroportos estejam interligados, exceto os aeroportos de saída e chegada, para que o algoritmo de Dijkstra, quando executado, retorne obrigatoriamente uma escala.



```
package Principal;

import java.util.ArrayList;
7 usages
public class Grafo {

    //Atributos:
    6 usages
    private final ArrayList<Aeroportos> ListaAero;
    4 usages
    private final int saida;
    4 usages
    private final int chegada;

```

```
    //Construtor:
    1 usage
    public Grafo(ArrayList<Aeroportos> listaAero, int sai, int cheg) {
        ListaAero = listaAero;
        saida = sai;
        chegada = cheg;

        //Para cada aeroporto na lista de aeroportos, cria-se a lista de trajetos (Todos se interligam, exceto a origem ao destino)

        for (int i = 0; i < ListaAero.size(); i++) {

            for (int j = 0; j < ListaAero.size(); j++) {
                if (i != j) {
                    if((i==saida && j==chegada)||(i==chegada && j==saida)){
                        continue;
                    }
                    Aeroportos aero1 = ListaAero.get(i);
                    Aeroportos aero2 = ListaAero.get(j);
                    aero1.getSaidas().add(new Trajetos(aero2));
                }
            }
        }
    }

    //Métodos Getter:
    7 usages
    public int getSaida() { return saida; }
    7 usages
    public int getChegada() { return chegada; }
    25 usages
    public ArrayList<Aeroportos> getListaAero() { return ListaAero; }
}
```

5 Dijkstra

Classe dedicada à execução do algoritmo de Dijkstra. Possui como atributos um Grafo, uma lista chamada *Vertice* e uma lista chamada *Caminho*, em que ambos terão o mesmo tamanho da lista de aeroportos do Grafo. O construtor inicializa essas listas com -1 e *infinito*, respectivamente, para todos os elementos dessas listas, exceto para o índice igual ao do aeroporto de saída. Neste, as listas recebem, respectivamente, o próprio índice do aeroporto correspondente e 0.

O valor armazenado no $caminho[i]$ corresponde ao menor caminho para se chegar ao aeroporto i a partir do aeroporto de saída. O valor vai sendo atualizado à medida em que o algoritmo vai rodando.

De maneira semelhante, valor armazenado no $vertice[i]$ corresponde ao índice do aeroporto visitado imediatamente antes de se chegar ao aeroporto i, de modo que tenhamos o menor caminho possível.

```java
package Principal;

import java.util.ArrayList;
1 usage
public class Dijkstra {
    //Atributos:
    10 usages
    private final Grafo grafo;
    5 usages
    private final ArrayList<Integer> vertice;
    9 usages
    private final ArrayList<Double> caminho;

    //Construtor:
    1 usage
    public Dijkstra(Grafo grafo) {
        double infinito = Double.POSITIVE_INFINITY;
        this.grafo = grafo;
        this.caminho = new ArrayList<>();
        this.vertice = new ArrayList<>();

        for (int i=0;i<grafo.getListaAero().size();i++){
            if(i==grafo.getSaida()){
                caminho.add(0.0);
                vertice.add(i);
            }
            else {
                caminho.add(infinito);
                vertice.add(-1);
            }
        }
    }
```

Por construção, como todos os vértices do grafo são interligados, pela desigualdade triangular é de se prever que todos os valores de $vertice[i]$ serão iguais ao índice do aeroporto de partida, exceto para o aeroporto de chegada, após a execução do algoritmo. para este, $vertice[c]$ corresponde ao índice do aeroporto de escala. Do mesmo modo, o caminho mínimo para qualquer aeroporto é o voo direto. Como isso não é possível apenas para o aeroporto de chegada, para este, teremos a soma das distâncias percorridas nas rotas do aeroporto de partida para o de escala com a do aeroporto de escala para o aeroporto de chegada.

Agora, para executar o algoritmo, seguimos o seguinte loop até que todos os aeroportos sejam marcados como visitados: Primeiramente, verifica-se o aeroporto não visitado com menor valor de caminho. Este será o próximo aeroporto ser visitado. Marcamos esse aeroporto como visitado (setando a variável booleana *explorado* dele como *true*). Em seguida, fazemos o teste para determinar se o caminho do aeroporto que está sendo visitado, somado com a distância desse aeroporto à um outro aeroporto A_j não visitado é menor que o $caminho[j]$. Caso seja, encontramos uma rota melhor para o aeroporto j do que a anterior. Assim, atualizamos o valor de $caminho[j]$ e também o valor de $vertice[j]$ para o índice do aeroporto que está sendo visitado. Ao final desse processo, teremos as listas *caminho* e *vertice* preenchidas com os dados que precisamos.

```java
// Método que executa o algoritmo de Dijkstra e calcula a Escala:
1 usage
public String Escala(){
        double infinito = Double.POSITIVE_INFINITY;
        int posicaoMin=0;

        for(int i=0;i<grafo.getListaAero().size();i++) {

            // Verifica o menor caminho não percorrido
            double min = infinito;

            for (int j = 0; j < grafo.getListaAero().size(); j++) {
                if ((!grafo.getListaAero().get(j).foiExplorado()) && caminho.get(j)<min) {
                    min = caminho.get(j);
                    posicaoMin = j;
                }
            }
            //Marca o aeroporto a ser trabalhado como visitado:
            grafo.getListaAero().get(posicaoMin).setExplorado(true);

            //atribui o valor do caminho, caso seja menor que o existente.
            for(Trajetos traj:grafo.getListaAero().get(posicaoMin).getSaidas()){
                if((!traj.getChegada().foiExplorado()) &&
                        (caminho.get(posicaoMin)+grafo.getListaAero().get(posicaoMin).getDist(traj.getChegada())<caminho.get(traj.getChegada().getIndice()))){

                    caminho.set(traj.getChegada().getIndice(),caminho.get(posicaoMin)+grafo.getListaAero().get(posicaoMin).getDist(traj.getChegada()));
                    vertice.set(traj.getChegada().getIndice() , posicaoMin);
                }
            }
        }
```

Por fim, o que nos interessa é o aeroporto de escala entre os aeroportos de saída e chegada. Assim, queremos saber o nome do aeroporto cujo índice está alocado em *vertice*[*c*]. Assim, invocamos o método *getNome* da classe Aeroportos para este aeroporto e retornamos essa String.

```
58          //Retorna a String com o nome do aeroporto correspondente à escala que minimiza o trajeto.
59
60          return grafo.getListaAero().get(vertice.get(grafo.getChegada())).getNome();
61      }
62  }
63
```

6 Rotas

Classe que possui como atributos os nomes de dois aeroportos, o de saída e o de chegada, para facilitar a busca por um caminho no Banco de Dados, para posterior inclusão ao se fazer uma busca.

```java
package Principal;

public class Rotas {
    //Atributos:
    2 usages
    private String saida;
    2 usages
    private String chegada;

    //Métodos Getter e Setter:
    1 usage
    public String getSaida() { return saida; }
    1 usage
    public void setSaida(String saida) { this.saida = saida; }
    1 usage
    public String getChegada() { return chegada; }
    1 usage
    public void setChegada(String chegada) { this.chegada = chegada; }
}
```

7 Conexão

Classe que estabelece a conexão entre o Banco de Dados SQL e o Java, com métodos para busca de aeroportos no Banco de Dados e também para inclusão de Rotas neste banco de Dados.

```java
public class Conexao {

    //Método que importa a lista de aeroportos do BD e retorna uma ArrayList de objetos do tipo Aeroportos:
    1 usage
    public ArrayList<Aeroportos> ImportarAeroportosSQL() throws SQLException, ClassNotFoundException {

        ArrayList<Aeroportos> listaAeroportos = new ArrayList<>();
        //Estabelece a Conexão com o BD:
        Class.forName( className: "com.mysql.cj.jdbc.Driver");
        Connection connection = DriverManager.getConnection( url: "jdbc:mysql://localhost/airportdata?user=root&password=123456");
        ResultSet resultSet = connection.createStatement().executeQuery( sql: "SELECT * from aeroportos");

        try {
            while(resultSet.next()){
                double conv= Math.PI/180;
                Aeroportos A = new Aeroportos();

                A.setIndice(resultSet.getInt( columnLabel: "id"));
                A.setNome(resultSet.getString( columnLabel: "Nome"));
                A.setLat(conv * resultSet.getDouble( columnLabel: "Lat"));
                A.setLong(conv * resultSet.getDouble( columnLabel: "Long"));
                A.setEstado(resultSet.getString( columnLabel: "Estado"));

                listaAeroportos.add(A);
            }
            resultSet.close();
        } catch (SQLException e) {System.out.println("ERRO");}
        return listaAeroportos;
    }
```

A conexão é estabelecida nas três linhas seguintes ao comentário de conexão. Em seguida, no *Try*, temos a busca no Banco de Dados pelos dados do aeroportos, que são instanciados em um Aeroporto *A* e, em seguida, adicionado em uma lista de Aeroportos.

Agora, no método de exportação, da mesma maneira do método anterior, criamos primeiramente a conexão entre o Java e o Banco de dados. Em seguida, cria-se uma lista de rotas, para que busquemos no Banco de Dados as rotas já adicionadas, para facilitar a verificação se uma nova rota deve ser adicionada no BD ou não, evitando assim rotas duplicadas no banco.

Esse método recebe como parâmetros o Grafo que contém os aeroportos de partida e chegada e também uma String, correspondente à escala retornada no algoritmo de Dijkstra, executado no método *Executar*, na classe *Menu*.

```java
    //Método que exporta a rota Partida  -> Escala -> Chegada para o BD, caso esta já não esteja contida no BD:
    1 usage
    public void ExportarRotasSQL(Grafo G,String str) throws SQLException, ClassNotFoundException {

        PreparedStatement pstm;

        //Cria a Conexão:

        Class.forName( className: "com.mysql.cj.jdbc.Driver");
        Connection connection = DriverManager.getConnection( url: "jdbc:mysql://localhost/airportdata?user=root&password=123456");
        ResultSet resultSet = connection.createStatement().executeQuery( sql: "SELECT * from rotas");

        ArrayList<Rotas> rotas = new ArrayList<>();

        //Cria uma lista de Rotas, contendo Partida e Chegada com as Rotas já armazenadas no BD:

        while(resultSet.next()){
            Rotas R = new Rotas();

            R.setSaida(resultSet.getString( columnLabel: "Saida"));
            R.setChegada(resultSet.getString( columnLabel: "Chegada"));
            rotas.add(R);
        }

        String sql = "insert into rotas (Saída, Chegada, Escala) values (?,?,?...";
```

É feita então a verificação se o trajeto inserido (saida/chegada) já está contido no banco de dados por meio de um loop passando por todas as rotas adicionadas à lista *Rotas* e utilizando um booleano auxiliar *estaNoBD*, setado como falso e tornando verdadeiro caso o caminho já pertença ao banco. Assim, se estiver no BD, é exibida uma mensagem informado e, caso contrário, o trajeto é adicionado ao banco e é exibida uma mensagem informando que a inclusão foi realizada com sucesso.

```java
        //teste se o deslocamento já está no BD:

        boolean estaNoBD = false;

        for(Rotas R: rotas){

            if((R.getSaida().equals(G.getListaAero().get(G.getSaida()).getNome())) && (R.getChegada().equals(G.getListaAero().get(G.getChegada()).getNome()))){
                estaNoBD = true;
                break;
            }
        }
        //Caso teste dê falso, adiciona ao BD:
        if (!estaNoBD){

            pstm = connection.prepareStatement(sql);

            pstm.setString( parameterIndex: 1,G.getListaAero().get(G.getSaida()).getNome());
            pstm.setString( parameterIndex: 2,G.getListaAero().get(G.getChegada()).getNome());
            pstm.setString( parameterIndex: 3,str);

            pstm.execute();
            pstm.close();

            System.out.println("Caminho adicionado ao Banco de Dados!");
        }
        else System.out.println("Caminho já está no Banco de Dados!");
    }
}
```

8 Menu

Esta classe é responsável por estabelecer a comunicação do usuário com o programa, perguntando a ele quais aeroportos devem ser selecionados como partida e chegada, além de chamar o método *Escala* da classe *Dijkstra*, para retornar o caminho mínimo desejado. Além disso, essa classe também é a responsável por chamar o método *ExportarRotasSQL*, da classe *Conexão*, para adicionar o trajeto ao BD, se for o caso.

A classe possui como atributos duas listas, uma de Aeroportos e uma de Strings, que conterá os estados que possuem aeroportos internacionais contidos na lista de Aeroportos.

O método construtor importa do banco de dados os aeroportos e os adiciona à lista de aeroportos da classe, por meio do método *ImportarAeroportosSQL* da classe *Conexão*. Logo após, o método preenche a lista de estados com os diferentes estados acessando o método *getEstado* de cada aeroporto contido na lista, caso este não tenha sido inserido anteriormente. Por fim, é feito um *sort* nessa lista para que ela fique em ordem alfabética.

```java
package Principal;

import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;
1 usage
public class Menu {

    //Atributos:
    7 usages
    private final ArrayList<Aeroportos> listaAeroportos;

    9 usages
    private final ArrayList<String> listaEstados;

    //Construtor:
    1 usage
    public Menu() throws SQLException, ClassNotFoundException {

        listaAeroportos = new Conexao().ImportarAeroportosSQL();

        listaEstados = new ArrayList<>();
        for(Aeroportos aero:listaAeroportos){
            if(!listaEstados.contains(aero.getEstado())){
                listaEstados.add(aero.getEstado());
            }
        }
        Collections.sort(listaEstados);
    }
```

A classe possui o método *selecioarAeroporto*, que imprime na tela a lista de estados disponíveis e logo em seguida pede para o usuário inserir o número correspondente ao estado desejado na lista. Esse inteiro é alocado por meio de um Scanner na variável local *Estado*. Assim, logo em seguida, é imprimida na tela uma lista com os aeroportos presentes naquele estado, por verificação direta na lista de Aeroportos (se o aeroporto possui estado igual ao selecionado pelo usuário, o nome do aeroporto é impresso na tela).

```java
//Método que Pergunta ao usuário o Aeroporto que ele deseja selecionar e retorna o índice do aeroporto:
4 usages
public int selecionarAeroporto(){

    int index=-1;

    //Imprime a lista de Estados:

    for(int i=1;i<=listaEstados.size();i++){
        System.out.println(i + " - " + listaEstados.get(i-1));
    }

    //Pergunta ao usuário o estado do aeroporto a ser selecionado:

    System.out.print("\nSelecione o Estado do Aeroporto: ");

    Scanner sc1 = new Scanner(System.in);
    Scanner sc2 = new Scanner(System.in);

    int Estado = sc1.nextInt()-1;
    String aerop;
```

```java
    //Imprime lista de Aeroportos no Estado Selecionado:

    System.out.println("\nAeroportos em "+ listaEstados.get(Estado) + " :\n");

    int contador = 1;
    for(Aeroportos aero:listaAeroportos){
        if(aero.getEstado().equals(listaEstados.get(Estado))){
            System.out.println( contador + " - " + aero.getNome());
            contador++;
        }
    }

    //Pergunta ao Usuário o Aeroporto a ser selecionado:

    System.out.print("\nSelecione o Aeroporto (Nome): ");
    aerop = sc2.nextLine();
    for (Aeroportos A : listaAeroportos) {
        if ((A.getNome().equals(aerop)) && (A.getEstado().equals(listaEstados.get(Estado))) ) {
            index = A.getIndice();
        }
    }

    //Retorna o índice do aeroporto selecionado:
    return index;
}
```

Após a impressão da lista de Aeroportos pertencentes àquele estado, é perguntado ao usuário qual aeroporto ele deseja selecionar. Assim, por meio de mais um Scanner, atribuímos à variável local *aerop* o nome do aeroporto desejado pelo usuário. É feita assim mais uma busca na lista de Aeroportos. Caso seja encontrado na lista um aeroporto com o mesmo nome do aeroporto desejado pelo usuário, é retornado pela função o índice desse aeroporto. Caso não seja encontrado, a função retorna -1.

```
73      //Método que seleciona os Aeroportos de partida e chegada, calcula a Rota e armazena no BD:
        1 usage
74      public void executar() throws SQLException, ClassNotFoundException {
75          //Escolhe os Aeroportos de saída e chegada:
76          int c, s;
77
78          System.out.println("\nSelecione o Aeroporto de Saida: \n");
79
80          s = selecionarAeroporto();
81
82          while(s==-1){
83              System.out.println("\nAeroporto inválido! Tente novamente.\n");
84
85              System.out.println("Selecione o Aeroporto de Saida: \n");
86
87              s = selecionarAeroporto();
88          }
89
90          System.out.println("\nxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\n");
91
92          System.out.println("\nSelecione o Aeroporto de Chegada:\n");
93
94          c = selecionarAeroporto();
95
96          while(c==-1){
97              System.out.println("\nAeroporto inválido! Tente novamente.\n");
98              System.out.println("Selecione o Aeroporto de Chegada:\n");
99              c = selecionarAeroporto();
100         }
```

Agora, o método Executar tem o objetivo de, de fato, se comunicar com o usuário, chamando o método *selecionarAeroporto* para determinar o aeroporto de saída e o aeroporto de chegada, verificando se é inserido pelo usuáro um aeroporto válido (com a utilização dos loops while($s == -1$) e while($c == -1$)). Em seguida, é feita a verificação se os aeroportos de saída e chegada são diferentes. Se forem iguais, é exibida uma mensagem de erro e o método se encerra. Caso sejam distintos, é construído um Grafo com a lista de aeroportos e os índices dos aeroportos de saída e chegada, para se determinar a escala, armazenada em uma variável local *escala*, que é utilizada em seguida para imprimir na tela a rota ótima e para se chamar o método *ExportarRotaSQL*, da Classe *Conexão*.

```
        if(s==c){
            System.out.println("O aeroporto de saída deve ser diferente do aeroporto de chegada!");
        }

        //Caso os aeroportos selecionados sejam válidos, calcula a Escala e imprime a rota na tela:
        else{
            System.out.println("\n\nA menor rota, considerando uma escala, é:\n");
            System.out.println("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\n");

            Grafo G = new Grafo(listaAeroportos,s,c);

            String escala = new Dijkstra(G).Escala();

            System.out.println("\t\t" + listaAeroportos.get(s).getNome() + " -> " + escala + " -> " + listaAeroportos.get(c).getNome());

            System.out.println("\nxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\n");

            //Armazena o trajeto no Banco de Dados:
            Conexao C = new Conexao();
            C.ExportarRotasSQL(G,escala);
        }
    }
}
```

9 Principal

Classe para execução do *main*. Dentro do *main*, cria-se um objeto da classe Menu e chama-se o método *Executar* para rodar o programa.

```java
package Principal;

import java.sql.SQLException;

public class Principal {

    public static void main(String[] args) throws SQLException, ClassNotFoundException {

        new Menu().executar();
    }
}
```

10 Bando de Dados(SQL)

O BD que foi utilizado possui duas tabelas, uma com os aeroportos internacionais e seus dados e outra com as rotas já pesquisadas pelo usuário, que estão dispostas da seguinte maneira:

	id	Nome	Lat	Long	Estado
1	0	CZS	-7.598349	-72.77343	Acre
2	1	RBR	-9.86634	-67.89739	Acre
3	2	MCZ	-9.511861	-35.793182	Alagoas
4	3	MCP	0.0500081	-51.06711	Amapá
5	4	MAO	-3.0358474	-60.046326	Amazonas
6	5	TBT	-4.25038	-69.93939	Amazonas
7	6	SSA	-12.911098	-38.33124	Bahia
8	7	FOR	-3.7771554	-38.533096	Ceará
9	8	BSB	-15.869737	-47.917236	Distrito Federal
10	9	VIX	-20.257648	-40.283535	Espírito Santo
11	10	GYN	-16.632303	-49.216347	Goiás
12	11	SLZ	-2.5849926	-44.23488	Maranhão
13	12	CGB	-15.653079	-56.117268	Mato Grosso
14	13	CGR	-20.468693	-54.67412	Mato Grosso do Sul
15	14	CMG	-19.013786	-57.66343	Mato Grosso do Sul
16	15	PMG	-22.549639	-55.702614	Mato Grosso do Sul
17	16	CNF	-19.634098	-43.965397	Minas Gerais
18	17	BEL	-1.3820615	-48.477524	Pará
19	18	STM	-2.4248295	-54.786243	Pará
20	19	JPA	-7.146009	-34.948776	Paraíba
21	20	CWB	-25.532713	-49.17248	Paraná
22	21	IGU	-25.59771	-54.488514	Paraná
23	22	REC	-8.125932	-34.924015	Pernambuco
24	23	PHB	-2.895248	-41.729797	Piauí
25	24	THE	-5.063463	-42.82119	Piauí
26	25	GIG	-22.805265	-43.25663	Rio De Janeiro
27	26	CFB	-22.925629	-42.079235	Rio De Janeiro
28	27	NAT	-5.7679067	-35.364033	Rio Grande do Norte
29	28	POA	-29.993473	-51.17538	Rio Grande do Sul
30	29	BGX	-31.389591	-54.11207	Rio Grande do Sul
31	30	PET	-31.715822	-52.33069	Rio Grande do Sul
32	31	URG	-29.783995	-57.0357	Rio Grande do Sul
33	32	PVH	-8.71417	-63.89832	Rondônia
34	33	BVB	2.8419237	-60.69259	Roraima
35	34	FLN	-27.670118	-48.545967	Santa Catarina
36	35	NVT	-26.880535	-48.649086	Santa Catarina
37	36	GRU	-23.430573	-46.47304	Sao Paulo
38	37	VCP	-23.008205	-47.13757	Sao Paulo
39	38	QSC	-21.87696	-47.90555	Sao Paulo
40	39	AJU	-10.98509	-37.07262	Sergipe

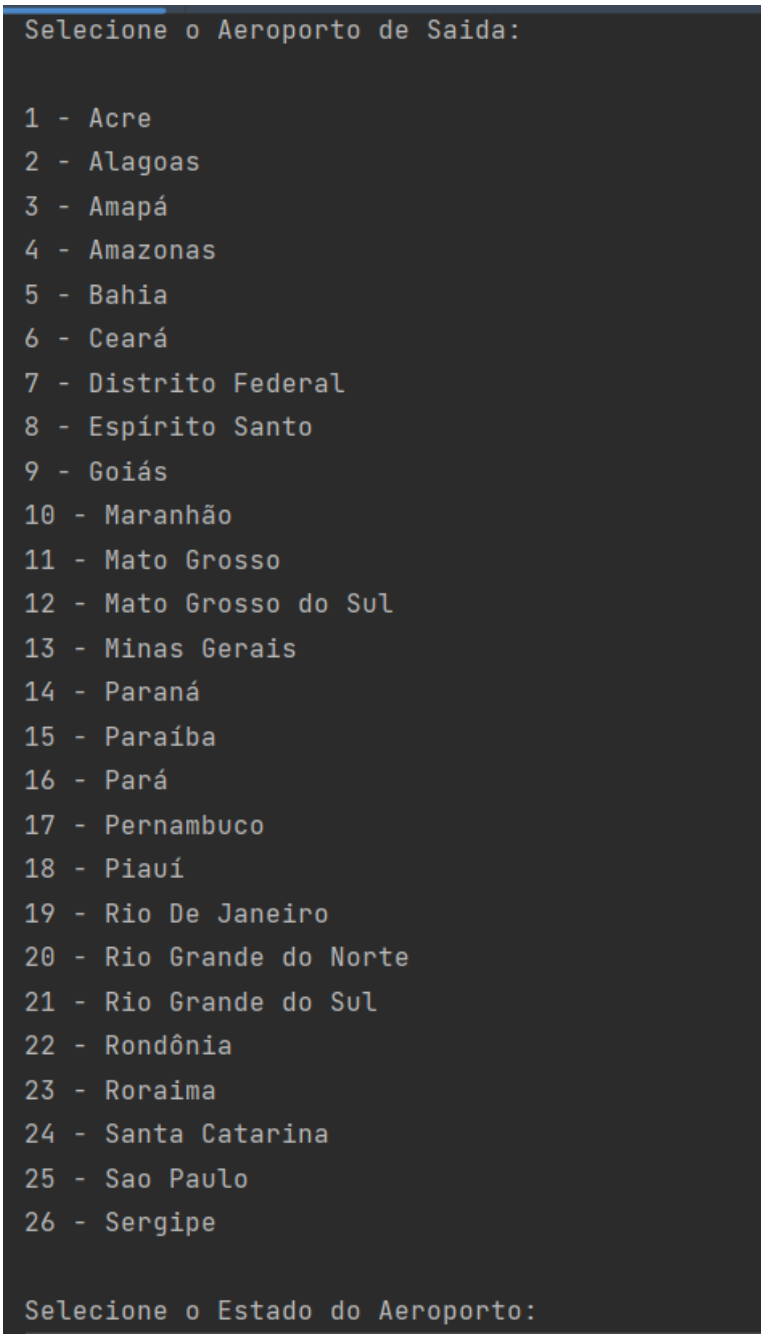
WHERE ORDER BY Índice

	Índice ▲ 1	Saída	Escala	Chegada
1	1	FOR	REC	MCZ
2	2	GIG	VCP	GRU
3	3	GIG	FLN	POA
4	4	FOR	CNF	GIG
5	5	POA	NVT	CWB

11 Programa Rodando

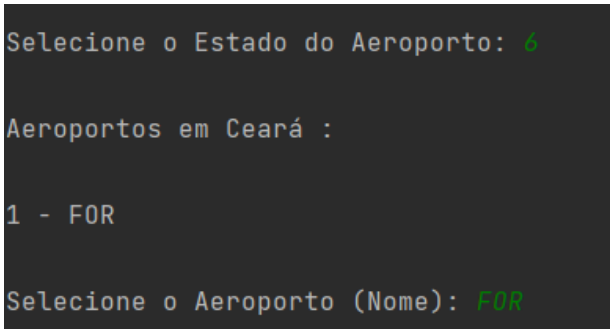
Por fim, será mostrado aqui o funcionamento do programa, passo a passo, ao adicionarmos uma nova rota ao Banco de Dados:

1) Lista de estados para aeroporto de saída:

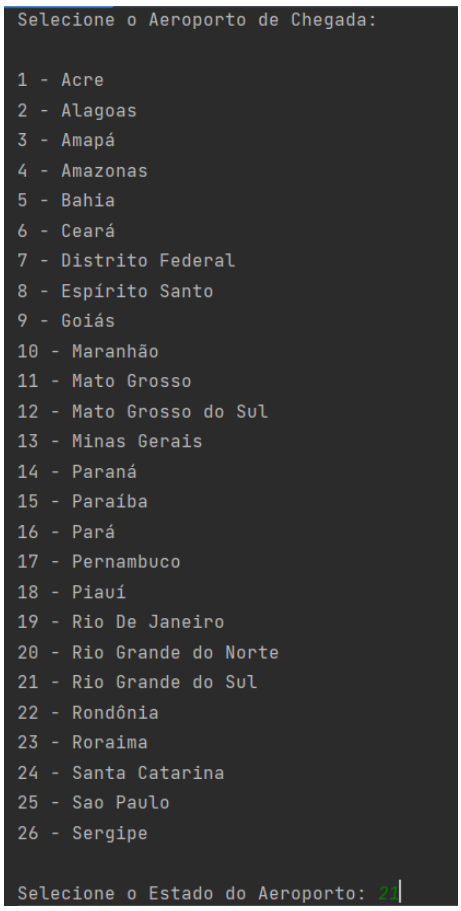


2) Seleção do estado:

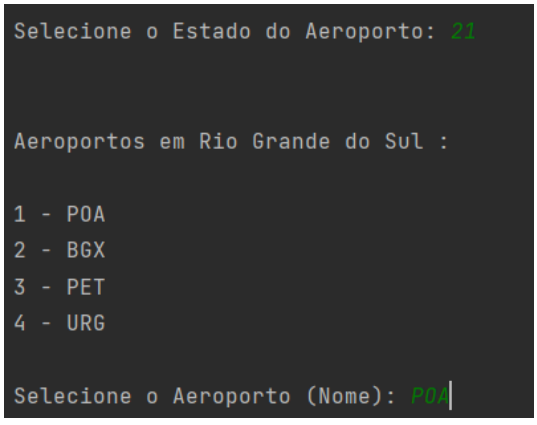
3) Seleção do aeroporto de saída:



4) Seleção de estado para aeroporto de chegada:



5) Seleção do Aeroporto de chegada:



6) Impressão da Rota:

```
A menor rota, considerando uma escala, é:

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

        FOR -> VCP -> POA

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Caminho adicionado ao Banco de Dados!

Process finished with exit code 0
```

7) Atualização do Banco de Dados:

	Índice	Saída	Escala	Chegada
1	1	FOR	REC	MCZ
2	2	GIG	VCP	GRU
3	3	GIG	FLN	POA
4	4	FOR	CNF	GIG
5	5	POA	NVT	CWB
6	6	FOR	VCP	POA