GROUP:

TANISH SHETTY

HARSH PANDITA

VIDHI VAIDYA

The Ship Wayfinder

Objective: Finding the most optimum path through a given set of countries and a given starting port in both fuel and cost.

Algorithm used:

-It is derived from the Dynamic Programming Approach of the Assembly Line Algorithm.

-To find the distance between two ports using coordinates, we made use of the Haversine formula.

Tools Used: Java and Swing(Java GUI).

Database Used: SQL database system using MySQL.

Structure of files:

main.java

mysql-driver: It is a .jar file to allow java to access MySQL

algo(package):

-Algorithm.java

gui(package):

-mainw.java

-test1.java

Database Table name – ports

Attributes: id, name, country, lat, longitude, cost

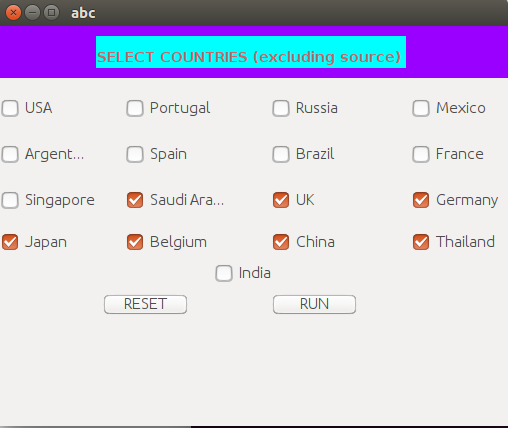
Class Composition:

Port – String name, double cost, double[] coordinates & distances.

Country – String name, double[] avgCoord, Port[] ports, double dist & amt.

Step-by-Step Explanation:

1.



The first window shows a series of checkboxes which correspond to the countries one can select when wanting to supply to said countries.

The Reset simply deselects those buttons and the Run button will take us to the next window as well as extracting all the names and making an **Array of Country Names.**

This array is then sent to the algo package – Algorithm.java to this function

public void getCountryArray(String[] country\_name){

        this.conn = null;

        this.state = null;

        this.result1 = null;

        this.countries = null;

        try{

            this.conn = DriverManager.getConnection(this.db, this.user, this.pass);

            this.state = this.conn.createStatement();

            this.countries = new Country[country\_name.length];

            double[] coordinates;

            for(int i = 0; i < country\_name.length; i++){

                try{

                    this.result1 = state.executeQuery("Select \* from ports where country = \"" + country\_name[i] + "\";");

                    int size = 0;

                    Map<Integer, Port> port\_map = new HashMap<Integer, Port>();

                    int j = 0;

                    while(this.result1.next()){

                        coordinates = Algorithm.getCoordinates(this.result1.getString("lat") + " " + this.result1.getString("longitude"));

                        this.px = new Port(Double.parseDouble(this.result1.getString("cost")), this.result1.getString("name"), coordinates);

                        port\_map.put(j, this.px);

                        j++;

                    }

                    size = j;

                    this.ports = new Port[size];

                    for(j = 0; j < size; j++){

                        this.ports[j] = port\_map.get(j);

                    }

                    this.countryx = new Country(country\_name[i], ports);

                    this.countries[i] = countryx;

                }catch(Exception e){System.out.println(e);}

                finally{

                    if(this.result1 != null){this.result1.close();}

                    this.result1 = null;

                }

            }

        }

        catch(Exception e){System.out.println(e);}

        finally{

            try{

                if(this.state != null){this.state.close();}

                if(this.conn != null){this.conn.close();}

            }catch(Exception e){}

            this.state = null;

            this.conn = null;

        }

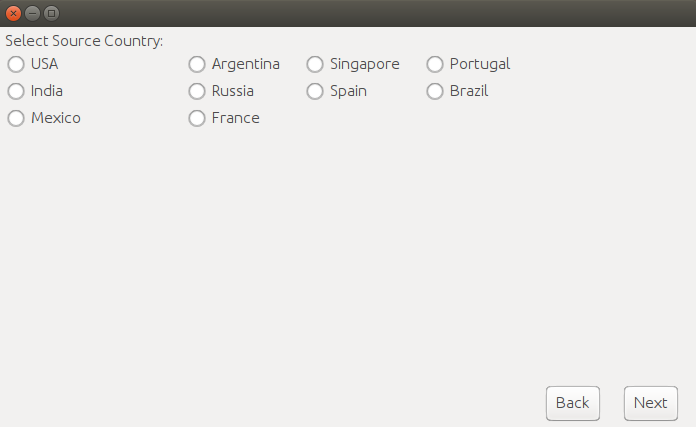
    }

**The above function in summary creates an array of countries (Objects of class Country) with all of its ports and coordinates which are taken from the database table “ports”.**

**Also, within the window, an Object of class Algorithm called data here has been initialised and all of the relevant values are stored there.**

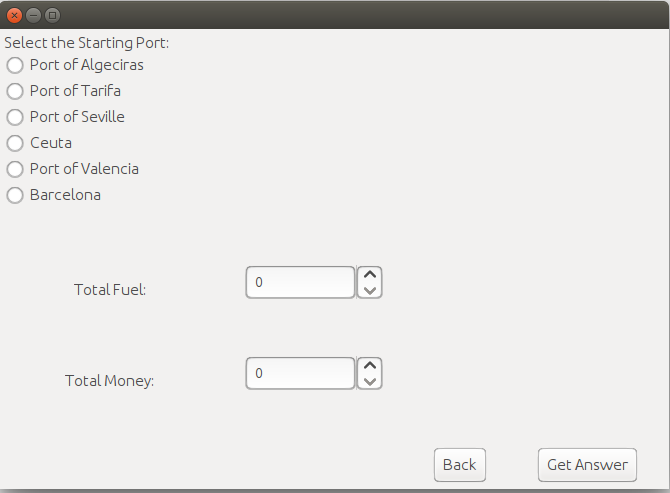
**It transfers this object from one window to another through the constructor as a parameter.**

2.

The Second window pops up and asks you to select the country the ship starts from. After selecting the appropriate radiobutton, the name of said country is extracted and all of the ports from that country are acquired. Let’s say we select Spain.

(Step 3 – next page)

3.



We select any one of the given ports, then we access all the information about the said port from the database and initialise an object of class Port called starting\_port which will be an attribute, therefore to access we would ask for **data.starting\_port**.

We also ask for the total fuel (in terms of mileage) and total money.

These two details help us in determining the rarer and thus the more important resource.

These are meant to get the values for the formula:

cost = ratio1 \* distance + ratio2 \* amt

ratio1 = total\_money / (total\_fuel + total\_money)

ratio2 = total\_fuel / (total\_fuel + total\_money)

If either total fuel or total money is zero, the equal importance is assumed and ratio1 = ratio2 = 0.5

On Clicking the Get Answer button, a set of functions are activated.

**Algorithm.countryOrder(Country[] countries, Port starting\_port)** – It starts by finding the average coordinates of each country, then using the Haversine formula, it finds the distance between the starting\_port and said countries, and it rearranges them such that the next country is the country which is closest to the current one.

Eg. Old Order – USA, Argentina, Thailand; starting port = Mumbai

NewOrder – Thailand, USA, Argentina

**Algorithm.getDistances(Country[] countries, Port starting\_port)-** This function/method initialises the distance between each port of each country and each port in the adjacent country.

**Algorithm.getPath(Country[] countries, Port starting\_port)-** This function/method is the one that carries out the Assembly Line Algorithm of Dynamic Programming. With the help of a cost function, it finds the “cost” to go from the current port to one of the ports in the adjacent country. The function getCost(distance, amt) implements the above cost function with ratio1 and ratio2 already initialised.

The following is its function definition:

public static int[] getPath(Country[] countries, Port starting\_point){

        int[] path = new int[countries.length];

        tot\_cost = 0;

        Port temp = starting\_point;

        int index = -1;

        for(int i = 0; i < countries.length; i++){

            index = temp.getNextPort(countries[i]);

            path[i] = index;

            if(i == 0){

                countries[i].dist = temp.distances[index];

                countries[i].amt = countries[i].ports[index].cost;

            }

            else{

                countries[i].dist = countries[i - 1].dist + temp.distances[index];

                countries[i].amt = countries[i - 1].amt + countries[i].ports[index].cost;

            }

            tot\_cost += Algorithm.getCost(temp.distances[index], countries[i].ports[index].cost);

            temp = countries[i].ports[index];

        }

        return path;

    }

It updates the total fuel and money spent by obtaining the current amount + the amount in the previous country. **Therefore each country keeps an account of the fuel and amount spent to reach that country.**

It returns an array of integers called path, which refers to the port index number for each country i.

This “path” is saved in an attribute called path. Therefore to call it, we would call for **data.path**. Together with the countries array, alomg with the path array, we are able to obtain the ports which are used to move in the optimum path.

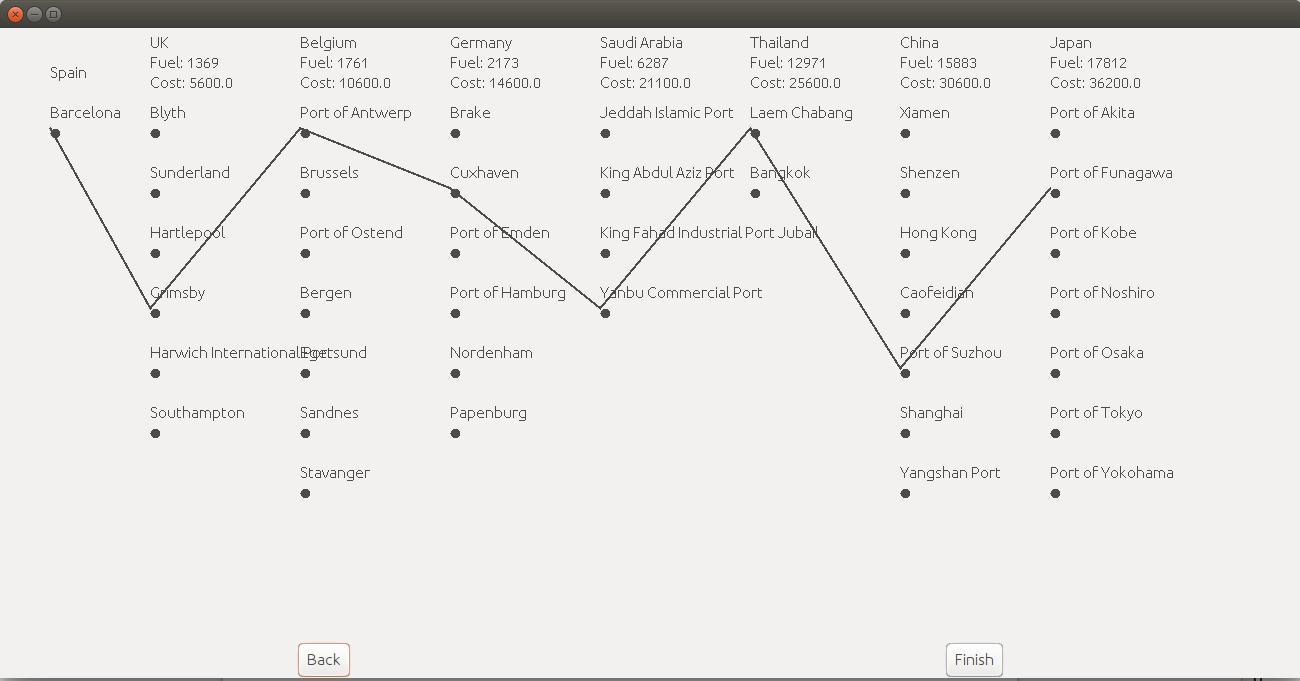
Thus, this final information is used and presented in the final window.

In the below example, we pick Barcelona as the source port.

Total fuel – 46,373

Total Money - 56776

4.



The above window show the optimum solution in the node-edge graph form.

Using the information from data.countries and data.path, we select certain usable local variables to extract that information to be used and then present the above.

For the above, we use Graphics2D for the above.

If g is the Graphics2D object of the given panel

We use the fillOval method to create the nodes.

We use the drawLine method to draw the edges between the nodes.

Above the graph, the name of the country the path leads to, along with the amount of fuel and money spent to get to that point are displayed there.

All the back buttons lead to the previous JPanel or Jframe. The finish button closes the window and effectively terminates the program.