Emily Curran, 2049 2496, Computational Thinking (MH603)

Explanation of TSP algorithm used

For this project I used a random nearest neighbour algorithm combined with two opt to produce a route between the orders which minimizes the total delay time. This method of generating a route with random nearest neighbour and then applying two opt is carried for as many times possible for the given time limit and then the best result generated is outputted to the screen.

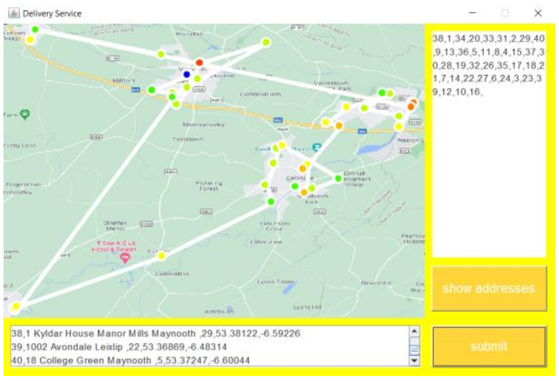
Random Nearest Neighbour:

The basic nearest neighbour algorithm is quite simple. Starting from the first order inputted, the algorithm will iterate through each of the orders the user inputs and connect each order to the closest available. In doing so, the goal is to create a linked list between the orders, where the route the delivery car will take to deliver the pizzas is equivalent to traversing the linked list.

The first is that the algorithm may not connect an order to another order if doing so will create a closed loop. The second is that if an order a has been connected to an order b, no other order may connect to order b.

In addition, I've implemented some randomness into comparing distances between orders so that the algorithm doesn't produce entirely predictable results. This is done by multiplying the distances by some random number in the range of 1 and 2.5 before comparing with the best distance so far. This means that the same algorithm can be run multiple times, giving slightly different outputs each time, increasing the likelihood that one of the outputs will be very efficient.

The result of the nearest neighbour algorithm with no added randomness can be seen across. Clearly it could do with further optimization, but it gives a result of 344 mins and 13 seconds.



```
38,1,34,20,33,31,2,29,40
,9,13,36,5,11,8,4,15,37,3
0,28,19,32,26,35,17,18,2
1,7,14,22,27,6,24,3,23,3
9,12,10,16,
```

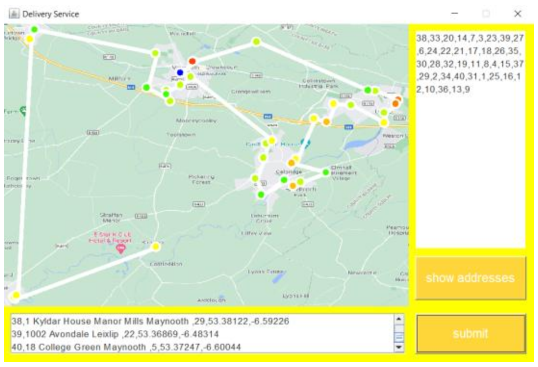
```
38,1 Kyldar House Manor Mills Maynooth ,29,53.38122,-6.59226
39,1002 Avondale Leixlip ,22,53.36869,-6.48314
40,18 College Green Maynooth ,5,53.37247,-6.60044
```

Two-opt:

Two-opt works to further optimize the results generated using the nearest neighbour algorithm by "uncrossing" any sequence orders which produce a path which crosses over itself. In the typical travelling salesman problem, where the goal is only to reduce travel distance and not delay time, having paths that cross will never be an optimal one, however, this version of two opt uses the delay time to measure if one route is more efficient than the other, and therefore it is possible that a path which crosses may be faster than one that does. However, the relationship between travel distance and delay time is so closely related in this instance that the effect is much the same.

This two opt algorithm works by exploiting the fact that uncrossing two orders is the same as reversing the order of the connections between them. The algorithm will check every valid combination of swaps possible, and if it finds that the delay time is reduced after swapping two orders it will make this swapped route the new route that the delivery person should take. This process is repeated until no further improvement is made and the best route is returned.

The effect of implementing this algorithm can be seen across. The delay time in this picture is just 205 minutes and 20 seconds, as compared with the 344 minutes and 13 seconds seen above.



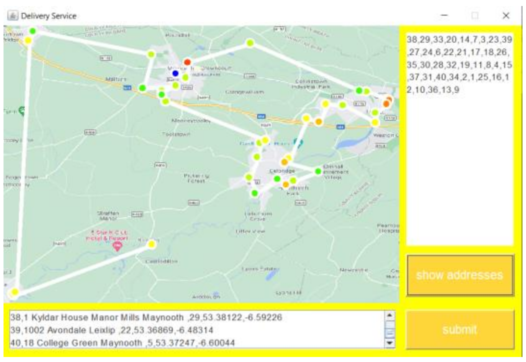
```
38,33,20,14,7,3,23,39,27
,6,24,22,21,17,18,26,35,
30,28,32,19,11,8,4,15,37
,29,2,34,40,31,1,25,16,1
2,10,36,13,9
```

```
38,1 Kyldar House Manor Mills Maynooth ,29,53.38122,-6.59226
39,1002 Avondale Leixlip ,22,53.36869,-6.48314
40,18 College Green Maynooth ,5,53.37247,-6.60044
```

Putting it all together:

Finally, these two algorithms are placed in a while loop which will test as many routes as it can generate it the 9.5 seconds permitted and return the best one. Using the sample data of 40 orders, the algorithm tests ~10,000 possible routes before outputting the result to the screen.

Adding in randomness and searching for 9.5 seconds gives an improved result, although only slightly. The delay time is reduced from 205 minutes and 20 seconds to 201 minutes and 55 seconds.



```
38,29,33,20,14,7,3,23,39
,27,24,6,22,21,17,18,26,
35,30,28,32,19,11,8,4,15
,37,31,40,34,2,1,25,16,1
2,10,36,13,9
```

```
38,1 Kyldar House Manor Mills Maynooth ,29,53.38122,-6.59226
39,1002 Avondale Leixlip ,22,53.36869,-6.48314
40,18 College Green Maynooth ,5,53.37247,-6.60044
```

Code structure (objects, classes, graphics etc)

The code can be broken down into 2 main sections, each with its own relevant classes. The first is the graphics section, which includes the Window, Globe, Node and Runner classes and the second is the pathfinding section which includes the PathFinder class. Divided between the two of these categories the Order class which is used by both the pathfinding and the graphics section of the code.

Graphics

As mentioned above, four classes are primarily responsible for handling the graphics the user sees.

1) The Window Class

   This class consists of a JFrame imported from javax.swing and contains all of the components which are displayed to the user, including the panels, text fields, buttons and canvases which are available for the user to interact with.

2) The Globe Class

   This class holds all the data relating to the canvas, the biggest rectangle on the top left of the screen which displays the idle animation. This class primarily operates in two modes, running the start animation or displaying the results of the pathfinding section. When the Boolean runningBigMap is true, Globe will display a start animation which represents a random node traversal on network resembling the continents of Eurasia and Africa. This is done by calling on the Node and Runner classes. Alternatively, when runningBigMap is false, Globe will display the route generated by the pathfinding section, displayed over a map which corresponds to each location. The switch from one mode to the other is triggered when the submit button is pressed.

3) Node and Runner

   These two classes are both responsible for the start animation played when app is first run. The Node class represents the nodes on the network of the start animation and contains information about the node's location on the screen in addition to a list of other nodes it is connected to. The runner class represents the dots which move between these nodes at random and contains methods for determining their own velocities and rendering to the screen.

4) Order

   The Order class works in a similar way to the node class, however, it stores additional information like the address, wait time, gps coordinates and number of the order. Once the screenLocation attribute of each order object is populated, the object can then be rendered to the screen using those points.

Pathfinding

Finally, the pathfinding portion of the program is almost entirely contained within the PathFinding class. This class contains all the methods used to parse the user data, create the order objects, calculate gps distance, assign the nearest neighbours, implement two-opt and to calculate the delay produced by any given route. In addition, this class also contains methods for creating the strings which are displayed in the right panel and the address field.

Learning outcomes

1) **How to use documentation**– Prior to attempting this project, I had never built anything using javax.swing or java.awt. As a result, I needed to read the documentation provided for these libraries and implement the packages appropriately.
2) **How to use a more Object-Oriented approach** – This is the largest project I have coded so far and, as a result, I found I needed to be more intentional with how I structured my code. I found it was necessary and, in fact, much easier to use classes to represent things like each order or, for example, all the methods related to a single task in classes rather than in poorly organized methods. As a result, I now feel much more comfortable using object oriented patterns in how I structure code.
3) **How to debug** – Throughout this problem I encountered problems interfacing with the various methods and classes included in the javax.swing and java.awt libraries. I found I got much better at using error codes and sites like stackoverflow.com to find solutions from users who had similar problems to my own.

Appendix (just paste your code with comments)

```
import java.util.Random;

import java.lang.Math;

import java.util.HashMap;

import java.awt.Font;

import java.awt.Color;

import java.awt.Panel;

import java.awt.Point;

import java.awt.Cursor;

import java.awt.Button;

import java.awt.Canvas;

import java.awt.Graphics2D;

import java.awt.BasicStroke;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.image.BufferStrategy;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JTextArea;

import javax.swing.*;

import java.awt.geom.Point2D;

import java.awt.geom.AffineTransform;

import java.io.*;

import java.util.*;

import java.awt.*;

import java.awt.image.*;

import javax.imageio.ImageIO;



  public class tsp{

  private static final int WIN_WIDTH = 800;
private static final int WIN_HEIGHT = 540;
private static Window window;
public static void main(String[] args) {
    window = new Window(WIN_WIDTH, WIN_HEIGHT, "Delivery Service");
    // ~60 FPS game loop
    // Using a separate Thread to display the window object on the screen
    // for simple games or animations like this one, just using
    // the main thread is also fine
    // gameloop();
    new Thread() {
        @Override
        public void run() {
            gameloop();
        }
    }
    .start();
}

static void gameloop() {
    long lastTime = System.currentTimeMillis();
    double amountOfTicks = 60.0;
    double ms = 1000f / amountOfTicks;
    double delta = 0;
    while (true) {
        long now = System.currentTimeMillis();
        delta += (now - lastTime) / ms;
        lastTime = now;
        while (delta >= 1) {
            // rendering and updating the Window object
            window.display();
            delta--;
            }
        }
    }
}

    class Window {
    // Main Components of the window
    JFrame windowFrame;
    // Inner components that are used in this window
    JPanel rightPanel, bottomPanel, addressPanel;
    JTextArea taOutput, taInput, taAddresses;
    JScrollPane addressScroll, inputScroll, outputScroll;
    Button btnSubmit, btnNextInput,btnShowAddresses;
    Font font;

    // the animated map
    Point topLeft = new Point(0,0);
    Point area = new Point(600,420);
    Globe globe;
    int count = 0;

    Window(int width, int height, String title) {
        windowFrame = new JFrame();
        windowFrame.setBounds(0, 0, width, height);
        windowFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        windowFrame.setCursor(new Cursor(Cursor.HAND_CURSOR));
        windowFrame.setTitle(title);
        windowFrame.setResizable(false);
        windowFrame.setLayout(null);

        createRightPanel();
        createBottomPanel();
        createAddressPanel();
        globe = new Globe(topLeft.x,topLeft.y, area.x, area.y);

        windowFrame.add(rightPanel);
        windowFrame.add(bottomPanel);
        windowFrame.add(addressPanel);
        windowFrame.add(globe.canvas);

        windowFrame.add(bottomPanel);
        windowFrame.setVisible(true);
    }

    void display() {
        // TIP: Always render first, update later !!!
        globe.render();
        globe.update();
    }
    private void createBottomPanel() {
        bottomPanel = new JPanel();
        bottomPanel.setBounds(0, 420, 800, 82);
        bottomPanel.setBackground(new Color(255,255,0));
        bottomPanel.setLayout(null);

        taInput = new JTextArea("",0,0);
        taInput.setFont(new Font("SansSerif", Font.PLAIN, 14));
        taInput.setBackground(Color.WHITE);
        taInput.setVisible(true);
        taInput.setLineWrap(true);

        inputScroll = new JScrollPane(taInput);
```

```
inputScroll.setBounds(10, 10, 585, 60);
inputScroll.setVerticalScrollBarPolicy(
ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED);

btnSubmit = new Button("submit");
btnSubmit.setBounds(610, 10, 165, 60);
btnSubmit.setBackground(new Color(255, 214, 57));
btnSubmit.setForeground(Color.WHITE);
btnSubmit.setFocusable(true);
btnSubmit.setFont(new Font("SansSerif", Font.PLAIN, 18));
btnSubmit.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    if(Globe.runningBigMap){
      String dataString = taInput.getText();
      globe.runningBigMap = false;
      globe.initialiaseOrders(dataString);
      rightPanel.add(btnShowAddresses);
      btnShowAddresses.setVisible(true);
      rightPanel.add(outputScroll);
      taOutput.setText(PathFinder.createTextRoute());
    }
  }
});

// creating the bottom panel and adding child components to it
bottomPanel.add(btnSubmit);
bottomPanel.add(inputScroll);

// if setVisible is set to false, the object won't appear on the screen
// it's best practice to call setVisible method after everything about
```

```
    // the UI component has been set. e.g. X, Y, width, height, Color etc.
    bottomPanel.setVisible(true);
    taInput.setVisible(true);
    btnSubmit.setVisible(true);
}
private void createRightPanel() {
    rightPanel = new JPanel();
    rightPanel.setBackground(new Color(255,255,0));
    rightPanel.setBounds(600, 0, 200, 420);
    rightPanel.setLayout(null);

    taOutput = new JTextArea("",0,0);
    taOutput.setFont(new Font("SansSerif", Font.PLAIN, 14));
    taOutput.setBackground(Color.WHITE);
    taOutput.setLineWrap(true);

    outputScroll = new JScrollPane(taOutput);
    outputScroll.setBounds(10, 10, 165, 325);
    outputScroll.setVerticalScrollBarPolicy(
    ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED);

    btnShowAddresses = new Button("show addresses");
    btnShowAddresses.setBounds(10, 345, 165, 65);
    btnShowAddresses.setBackground(new Color(255, 214, 57));
    btnShowAddresses.setForeground(Color.WHITE);
    btnShowAddresses.setFocusable(true);
    btnShowAddresses.setFont(new Font("SansSerif", Font.PLAIN, 18));
    btnShowAddresses.addActionListener(new ActionListener() {
    // when the button is clicked, this method runs
    @Override
    public void actionPerformed(ActionEvent e) {
        taAddresses.setText(PathFinder.createAddressList());
        if(addressPanel.isVisible()==true){
            addressPanel.setVisible(false);
        }else{
            addressPanel.setVisible(true);
        }
    }
});

    // creates the panel and adds all the above inner components
    rightPanel.setVisible(true);
}
private void createAddressPanel() {
    addressPanel = new JPanel();
    addressPanel.setBackground(Color.WHITE);
    addressPanel.setBounds(0, 0, 600, 420);
    addressPanel.setLayout(null);

    taAddresses = new JTextArea("",0,0);
    taAddresses.setFont(new Font("SansSerif", Font.PLAIN, 14));
    taAddresses.setVisible(true);
    taAddresses.setLineWrap(true);
    taAddresses.setBackground(Color.WHITE);

    addressScroll = new JScrollPane(taAddresses);
    addressScroll.setBounds(0,0,600,420);
    addressScroll.setVerticalScrollBarPolicy(
    ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);

    // creates the panel and adds all the above inner components
    addressPanel.add(addressScroll);
```

```
        taAddresses.setVisible(true);
        addressPanel.setVisible(false);
    }
}
class Node {
    int nVertex = 0;
    int index;
    Point location;
    int radius = 15;

    Node(int index, int x_pos, int y_pos){
        this.index = index;
        location = new Point(x_pos,y_pos);
    }
    void render(Graphics2D gfx) {
        gfx.setColor(new Color(255, 214, 57));
        gfx.fillOval(location.x, location.y, radius, radius);
    }
}
class Globe {
    /*
    represents the top left corner display. Depending on the value of runningBigMap
    will display either idle animation or solution. Methods with "big" in the
    title will be used in displaying default animation and "small" refers to
    the graphics for the solution.
    */
    Canvas canvas;
    private Graphics2D gfx;
    private BufferStrategy bfs;
    private Point area;
```

```
static boolean runningBigMap = true;

//attributes for bigMap
static Node[] nodes = new Node[31];
static int[][] graph = new int[31][31];
Runner[] runners = new Runner[6];

//attibutes for smallMap
static Order[] orders;
double scaling;
double xDisplacement;
double yDisplacement;

//attributes for the image
BufferedImage img;
Point2D.Double imgTopLeft = new Point2D.Double(-6.71261, 53.41318);
Point2D.Double imgBottomRight = new Point2D.Double(-6.45509, 53.28700);
AffineTransform at;
Boolean canDraw = false;

Globe (int x1, int y1, int x2, int y2) {
   area = new Point(x2, y2); // relative to (0, 0)
   Point topLeft = new Point(x1, y1);
   canvas = new Canvas();
   canvas.setSize(area.x, area.y);
   canvas.setLocation(new Point(topLeft.x, topLeft.y));
   canvas.setBounds(topLeft.x, topLeft.y, area.x, area.y);
   canvas.setBackground(new Color(191,255,0));
   canvas.setVisible(true);

   createBigGraph(graph);
```

```
    placeBigNodes(nodes);
    createBigRunners();
}
static Node randomBigNode(Node currentNode){
    Node randNode = new Node(0,0,0);
    int index = currentNode.index;

    int countConnected = 0;
    for(int j=0; j<graph[index].length; j++){
        if(graph[index][j]==1){countConnected++;}
    }


    int randInt = (int) (Math.random()*countConnected);
    int counter = 0;
    int i = 0;
    while(counter<=randInt){
        if(graph[index][i]==1){
            randNode = nodes[i];
            counter ++;
        }
        i++;
    }
    return randNode;
}
void render() {
    bfs = canvas.getBufferStrategy();
    // this null checking is VERY important because when
    // the game first starts, bfs will be null !!
    if (bfs == null) {
        int buffers = 0x2;
        canvas.createBufferStrategy(buffers);
        return;
    }
    // casting the return type (Graphics) to Graphics2D just to be safe
    gfx = (Graphics2D) bfs.getDrawGraphics();
    // calling all render methods
    update();
    canvas.update(gfx);
    if(runningBigMap){
        renderBigLines(gfx);


        for(int i=0; i<nodes.length; i++){
            nodes[i].render(gfx);
        }
        for(int i=0; i<runners.length; i++){
            runners[i].render(gfx);
        }
    }else{
    if(canDraw == true){
    gfx.drawImage(img, at, null);
    }
    if(orders != null){
    renderSmallLines(gfx);
            for(int i=1; i<=orders.length; i++){
                orders[i%(orders.length)].render(gfx);
            }
        }
    }

    bfs.show(); // displays all contents on the screen
    gfx.dispose(); // resets the gfx object for the next frame
```

```
}
void update() {
  //use this to update the positions of the moving runners
  if(runningBigMap){
    for(int i=0; i<runners.length; i++){
      runners[i].update();
    }
  }
}
void createBigGraph(int[][] graph){
  //creates the network between the nodes
  for(int i = 0; i<30; i++){
    for(int j = 0; j<30; j++){
      graph[i][j] = 0;
    }
  }
  String input =  "0:1&1:0,2,3&2:1,3&3:1,2,4,5&4:3,5&5:3,4,6,15&"+
  "6:5,7,8&7:6,8&8:6,7,9,10,14&9:8,10&10:8,9,11,13&11:10,12,13&"+
  "12:11,13&13:10,11,12,14,15&14:8,13,15&15:5,13,14,16,17&16:15&"+
  "17:15,18,19,28&18:17,28&19:17,20&20:19,21,26,27,28&21:20,22,23&"+
  "22:21,23,25&23:21,22,24&24:23,25,26&25:22,24&26:20,24,27&"+
  "27:20,26,28,30&28:17,18,20,27,29&29:28,30&30:27,29";

  String[] arr = input.split("&");
  for(int i=0; i<arr.length; i++){
    String[] nodeConnections = arr[i].split(":");
    int primeNode = Integer.parseInt(nodeConnections[0]);
    String[] secondaryNodes = nodeConnections[1].split(",");
    for(int j=0; j<secondaryNodes.length; j++){
      if(primeNode==Integer.parseInt(secondaryNodes[j])){
      }
```

```
            graph[primeNode][Integer.parseInt(secondaryNodes[j])] = 1;
        }
    }
}
    void placeBigNodes(Node[] nodes){
        //create and place the nodes for the default animation
    int[][] nodePosits = {
        {160,90},{197,55},{235,85},{253,55},{285,25},{335,90},{341,37},{397,30},
        {422,90},{478,60},{491,115},{516,160},{478,200},{435,160},{391,130},
        {303,155},{322,195},{210,160},{141,135},{253,220},{185,230},{272,255},
        {247,330},{210,295},{172,335},{203,375},{147,285},{128,230},{147,185},
        {97,165},{85,220}
    };

    for(int i=0; i<31; i++){
        nodes[i] = new Node(i, nodePosits[i][0], nodePosits[i][1]);
    }
}
    void createBigRunners(){
    for(int i=0; i<runners.length; i++){
        runners[i] = new Runner(nodes[i*5]);
    }
}
void renderBigLines(Graphics2D gfx){
    for(int i=0; i<31; i++){
        for(int j=30; j>=0; j--){
            if(graph[i][j]==1){
                int offset = 8;
                int x1 = nodes[i].location.x +offset;
                int y1 = nodes[i].location.y +offset-2;
                int x2 = nodes[j].location.x +offset;
                int y2 = nodes[j].location.y +offset-2;
                gfx.setColor(Color.WHITE);
                gfx.setStroke(new BasicStroke(6));
                gfx.drawLine(x1,y1,x2,y2);
            }
        }
    }
}
void initialiaseOrders(String dataString){
//create the orders, solve them and set their screen location
    orders = PathFinder.createPath(this, dataString);
    if(orders != null){
        setScreenLocations(orders);
    }
}
void setScreenLocations(Order[] orders){
    //assume gps coordinate approximate a flat plane
    //gps coordinate need to be scaled appropiately to fit on the
    //screen.

    double biggestX = Integer.MIN_VALUE;
    double smallestX = Integer.MAX_VALUE;
    double biggestY = Integer.MIN_VALUE;
    double smallestY = Integer.MAX_VALUE;

    for(int i=0; i<orders.length; i++){
        Point2D.Double point = orders[i].location;
        if(point.x > biggestX){biggestX = point.x;}
        if(point.x < smallestX){smallestX = point.x;}
        if(point.y > biggestY){biggestY = point.y;}
        if(point.y < smallestY){smallestY = point.y;}
        }



        double deltaX = biggestX - smallestX;

        double deltaY = biggestY - smallestY;


        double widthScreen = area.x;

        double heightScreen = area.y;


        double yScale = (heightScreen*0.95)/deltaY;   //95% of screen

    double xScale = (widthScreen*0.95)/deltaX;


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

            Point2D.Double point = orders[i].location;

            double xPos = ((point.x-smallestX)*(xScale))+(widthScreen*0.025)-6;//leave a 5 percent
margin

            double yPos = heightScreen-((point.y-smallestY)*(yScale))-

            (heightScreen*0.025)-6;

        orders[i].screenLocation = new Point((int) xPos, (int) yPos);

        }

        Point2D.Double newTopLeft = new Point2D.Double(smallestX-(deltaX*0.025),
biggestY+(deltaY*0.025));

        Point2D.Double  newBottomRight = new Point2D.Double(biggestX+(deltaX*0.025), smallestY-
(deltaY*0.025));


        sizeImage(newTopLeft, newBottomRight);

    }

    void sizeImage(Point2D.Double newTopLeft, Point2D.Double newBottomRight){

        //crop and stretch the image to fit under the order postions

        try{

            File file = new File("map.png");

            img = ImageIO.read(file);
    }catch(IOException e){
        System.out.println("couldn't get the file");
        e.printStackTrace();
    return;
    }

    double xDelta = (imgBottomRight.x - imgTopLeft.x);
    double yDelta = (imgTopLeft.y - imgBottomRight.y);

    double width = img.getWidth();
    double height = img.getHeight();

    int tlxPixel =(int) (width*(Math.abs(imgTopLeft.x-newTopLeft.x)/xDelta));
    int tlyPixel =(int) (height- (height*((Math.abs(imgBottomRight.y-newTopLeft.y)/yDelta))));
    int brxPixel =(int) (width*(Math.abs(imgTopLeft.x-newBottomRight.x) /xDelta));
    int bryPixel =(int) (height-((height*(Math.abs(imgBottomRight.y-newBottomRight.y))/yDelta)));

try{
      img = img.getSubimage(tlxPixel, tlyPixel, brxPixel-tlxPixel, bryPixel-tlyPixel);
}
catch(RasterFormatException e){
    System.out.println("points don't fit on the background");
    return;
}

canDraw = true;
width = img.getWidth();
height = img.getHeight();
at = new AffineTransform(area.x/width, 0, 0, area.y/height, 0, 0);
}
void renderSmallLines(Graphics2D gf){
        //render the route between the points
        int[] route = PathFinder.bestRoute;
        Order currentOrder = orders[PathFinder.bestRoute[0]];
        for(int i=0; i<orders.length-1; i++){
            int offset = 8;
            if(currentOrder != null && currentOrder.nextOrder != null){
                int x1 = currentOrder.screenLocation.x +offset;
                int y1 = currentOrder.screenLocation.y +offset - 2;
                currentOrder = currentOrder.nextOrder;
                int x2 = currentOrder.screenLocation.x +offset;
                int y2 = currentOrder.screenLocation.y +offset-2;
                gfx.setColor(Color.WHITE);
                gfx.setStroke(new BasicStroke(6));
                gfx.drawLine(x1,y1,x2,y2);
            }
        }
    }
    public void reset(){
        orders = null;
        runningBigMap = true;
    }
}

class Runner {
    //class for moving objects on the screen
    double xVelocity, yVelocity;
    double xPos, yPos;
    double speed = 0.4;
    static int radius= 14;
    Point location;
    Node currentNode;
Node targetNode;


Runner(Node startNode) {
   //intialise the runner
   currentNode = startNode;
   targetNode = Globe.randomBigNode(currentNode);
   location = new Point(currentNode.location.x, currentNode.location.y);
   xPos = location.x;
   yPos = location.y;
   setVelocityNode();
   }
   public void setVelocityNode(){
   double xDelta = targetNode.location.x-currentNode.location.x;
   double yDelta = targetNode.location.y-currentNode.location.y;
   double angle = Math.atan(yDelta/xDelta);


   if(xDelta>0){
      xVelocity = speed*Math.cos(angle);
      yVelocity = speed*Math.sin(angle);
   }else{
      xVelocity = -speed*Math.cos(angle);
      yVelocity = -speed*Math.sin(angle);
   }
}
void update(){
   //update its positon
   if(hasReachedNode()){
      currentNode = targetNode;
      targetNode = Globe.randomBigNode(currentNode);
      setVelocityNode();
   }
```

```
    xPos += xVelocity; //Storing xPos as a double for more control
    location.x = (int) xPos;

    yPos += yVelocity;
    location.y = (int) yPos;
}
void render(Graphics2D gfx){
    int x = location.x;
    int y = location.y;
    gfx.setColor(new Color(255, 214, 57));
    gfx.fillOval (x, y, radius, radius);
}
private boolean hasReachedNode(){
    //method to check if the runner has reached a node

    //handling x and y positons seperately in case one is reached
    //faster than the other. Mulitplying both sides by velocity to
    //reduce the number of if else statements needed.
    if(xVelocity*location.x>=xVelocity*targetNode.location.x){
        xVelocity = 0;
    }
    if(yVelocity*location.y>=yVelocity*targetNode.location.y){
        yVelocity = 0;
    }

    if(xVelocity == 0 && yVelocity == 0){
        return true;
    }
    return false;
}
```

```
}
class PathFinder{
  //class to find the optimal solution between houses
  static Order[] orders;
  static int[] bestRoute; //int array to represent how orders are connected
  static String bestRouteText; //store the route for printing to screen
  static double bestDelay; //to be replaced
  //with lower values

  public static Order[] createPath(Globe globe, String dataString){
    long startTime = System.currentTimeMillis();
    long elapsedTime = 0L;

    createOrders(dataString);

    double randomness = 1.5;

    nearNeighbour(orders, 0);
    int[] testRoute = createRoute(orders);
    double testDelay = delayTime(testRoute);
    bestRoute = twoOpt(testRoute, testDelay); //optomise testRoute with twoOpt
    bestDelay = delayTime(testRoute);
    int iterations = 0;



    while (elapsedTime < 9.5*1000) { //test for the best route while the program has
      //been running for less than 9 seconds
      nearNeighbour(orders, randomness);
      testRoute = createRoute(orders);
      testDelay = delayTime(testRoute);
      testRoute = twoOpt(testRoute, testDelay); //optomise testRoute with twoOpt
```

```
        testDelay = delayTime(testRoute); //get optomised delayTime
        if(testDelay<bestDelay){
            bestDelay = testDelay;
            bestRoute = testRoute;
        }
        elapsedTime = (new Date()).getTime() - startTime;
    }

    //update connections between orders to reflect changes to the best route
    bestDelay = delayTime(bestRoute);
    updateOrders(bestRoute);
    bestRouteText = createTextRoute();

    //get positions to render the orders
    globe.setScreenLocations(orders);

    return orders;
}
public static void createOrders(String dataString){
    dataString = "0, 3 Mill St Maynooth, 0, 53.381178, -6.592988 \n" + dataString;
    String[] dataRows = dataString.split("\n");
    orders = new Order[dataRows.length];

    for(int i=0; i<dataRows.length; i++){
        String[] data = dataRows[i].split(",");
        int orderId = Integer.parseInt(data[0].trim());
        String address = data[1].trim();
        int minutes = Integer.parseInt(data[2].trim());
        double xPos = Double.parseDouble(data[4].trim());
        double yPos = Double.parseDouble(data[3].trim());
        orders[i] = new Order(orderId,address,minutes,xPos,yPos);
    }
    for(int i=0; i<orders.length; i++){
        orders[i].fillDistances(orders);
    }
}
public static double delayTime(int[] route){
    // method to calculate how much delay a given route will generate


    double totalDist = 0; //60kmh speed so no need to convert to minutes
    double totalDelay = 0;


    for(int i=0; i<route.length-1; i++){
        totalDist += orders[route[i]].distances[route[i+1]];
        if(orders[route[i+1]].wait+totalDist > 30){// only wait time
            //greater than 30 minutes considered
            totalDelay += (orders[route[i+1]].wait+totalDist)-30; //add delay after thirty minutes
        }
    }
    return totalDelay;
}
public static void nearNeighbour(Order[] orders, double randomness){
    //connect every order to the nearest unconnected order
    for(int i=0; i<orders.length; i++){ // reset values for connected to
        orders[i].connectedTo = false;
        orders[i].nextOrder = null;
    }
    for(int i=0; i<orders.length; i++){
        double smallestDistance = Integer.MAX_VALUE;
        Order next = orders[i];
        for(int j=0; j<orders.length; j++){
            if(orders[j].connectedTo == false && //two orders don't point to the same order
            orders[j].nextOrder != orders[i]){ // two orders don't point to each other
              double distance = orders[i].distances[j];
              distance = distance*(1+(Math.random()*randomness));
              if(distance<smallestDistance &&
              i!=j && //don't connect an order to itself
              (!isClosed(orders[j], orders[i])
              || i == orders.length-1)){//don't connect orders if they close a loop
                smallestDistance = distance;
                orders[i].nextOrder = orders[j];
                next = orders[j];
              }
            }
          }
          next.connectedTo = true; // no other orders can point to this order
        }
      }
      public static boolean isClosed(Order start, Order end){
        //method to check if connecting orders will create a closed loop
        Order current = start;
        while(current.nextOrder != null){
          if(current.nextOrder == end){
            return true;
          }
          current = current.nextOrder;
        }
        return false;
      }
      public static int[] twoOpt(int[] route, double delay){
        double bestDel = delay;
        int numOrders = route.length;
        boolean improved = true;
```

```
    int[] newRoute = deepCopy(route);
    while(improved){
        improved = false;
        for(int i=1; i<numOrders-1; i++){ //start from 1, don't move start order
            for(int j=i+1; j<numOrders; j++){
                int[] newestRoute = reverse(i, j, newRoute);
                double newDelay = delayTime(newestRoute);
                if (newDelay<bestDel){
                    bestDel = newDelay;
                    newRoute = newestRoute;
                    improved = true;
                }
            }
        }
    }
    return newRoute;
}
public static int[] deepCopy(int[] array){
    int[] newArray = new int[array.length];
    for(int i=0; i<array.length; i++){
        newArray[i] = array[i];
    }
    return newArray;
 }
public static double gpsDistance(Point2D.Double firstPoint,
Point2D.Double secondPoint){
    //use the haversine formula to calculate the distance between points
    double lat1 = firstPoint.y*(Math.PI/180);
    double lon1 = firstPoint.x*(Math.PI/180);
    double lat2 = secondPoint.y*(Math.PI/180);
    double lon2 = secondPoint.x*(Math.PI/180);
    double deltaLat = lat2-lat1;
    double deltaLon = lon2-lon1;

    double a = (Math.sin(deltaLat/2)*Math.sin(deltaLat/2))+
    (Math.cos(lat1)*Math.cos(lat2)*Math.sin(deltaLon/2)*Math.sin(deltaLon/2));
    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));

    return 6371*c;
}
public static int[] createRoute(Order[] orderArr){
    //create an integer array from the connection between nodes
    //representing the route to be taken
    int[] newRoute = new int[orderArr.length];
    Order currentOrder = orderArr[0];
    for(int i=0; i<orderArr.length; i++){
        newRoute[i] = currentOrder.orderNumber;
        currentOrder = currentOrder.nextOrder;
    }
    return newRoute;
}
public static Order[] updateOrders(int[] route){
    //update order connectons based on integer array of route
    for(int i=0; i<route.length-1; i++){
        orders[route[i]].nextOrder = orders[route[i+1]];
    }
    orders[route[route.length-1]].nextOrder = null;
    return orders;
}
public static int[] reverse(int i, int j, int[] route){
    //elements from one element to another
    int size = route.length;
```

```
        int[] newRoute = new int[route.length];
        for(int a=0; a<i; a++){ //copy up until i
            newRoute[a] = route[a];
        }
        for(int b=0; b<=j && (b+i)<route.length; b++){//reverse from i to j
            if((j-b)<route.length){
                newRoute[b+i] = route[j-b];
            }
        }
        for(int c=j+1; c<route.length; c++){ //copy remaining
            newRoute[c] = route[c];
        }
        return newRoute;
    }
    public static String createTextRoute(){
        //create the string which is displayed in the right panel
        String textRoute = "";
        for(int i=0; i<bestRoute.length; i++){
            textRoute = textRoute+orders[bestRoute[i]].orderNumber+",";
        }
        textRoute = textRoute.substring(2, textRoute.length()-3);
        return textRoute;
    }
    public static String createAddressList(){
        //create the string which is displayed in the address panel
        String addressList = "";
        addressList = "Delay Time: "+(int) bestDelay+" mins  "+(int) (((bestDelay%1)*60)+1)+" secs   \n
\n";


        for(int i=1; i<bestRoute.length; i++){
            addressList = addressList +(i)+") "+orders[bestRoute[i]].address+"\n";
        }
        return addressList;
    }
}
class Order{
    //class to represent each order and its properties
    int orderNumber;
    String address;
    int wait;
    Point2D.Double location; //gps coordinates in the real world
    int radius = 10;
    Point screenLocation; //for rendering to the screen
    Order nextOrder;
    boolean connectedTo; //true if another order has this order as
    //its nextNode;
    double[] distances;

    Order(int orderId, String houseAddress, int minutes, double xPos,
        double yPos){
        orderNumber = orderId;
        address = houseAddress;
        wait = minutes;
        location = new Point2D.Double (xPos, yPos);
    }
    void render(Graphics2D gfx) {
        if(screenLocation != null){
            gfx.setColor(Color.WHITE);
            gfx.fillOval(screenLocation.x-3, screenLocation.y-3, (int) radius+6,
            (int) radius+6);

            if(wait<6){gfx.setColor(new Color(64,255,0));}
            else if(wait<11){gfx.setColor(new Color(191,255,0));}
            else if(wait<16){gfx.setColor(new Color(255,255,0));}
            else if(wait<21){gfx.setColor(new Color(255,191,0));}
            else if(wait<26){gfx.setColor(new Color(255,128,0));}
            else{gfx.setColor(new Color(255,64,0));}
            if(orderNumber == 0){gfx.setColor(Color.BLUE);}
            gfx.fillOval(screenLocation.x, screenLocation.y, radius, radius);
        }
    }
    void fillDistances(Order[] orders){
        //fill the distance from this node to every other node
        distances = new double[orders.length];
        for(int i=0; i<orders.length; i++){
            distances[i] = PathFinder.gpsDistance(this.location, orders[i].location);
        }
    }
}
```