Informatics Large Practical Report

Name: Teh Min Suan

Student id: s1817967

Software Architecture Description

There are a total of 7 classes present in the program.

1. App.class
2. DataReceiver.class
3. SensorReader.class
4. MapReader.class
5. PathFinder.class
6. Output.class
7. NearestNeighbour.class

App.class

This class is the main class of the program. It uses methods from DataReceiver, SensorReader and MapReader to receive,read and process data from the web server. Then, using the processed data as argument, it constructs a new PathFinder class to locate and move to the sensors. When all the sensors are visited, App.class uses methods from Output.class to generate a text file, recording the flight path of the drone and a geojson map.

DataReceiver.class

This is a helper class in which its methods are used by both SensorReader and MapReader. Basically, it receives data from webserver as a String object and return the object.

SensorReader.class

It reads the output of DataReceiver and process them into arrays. There are 2 class within SensorReader, which is, Sensor and SensorDetails. Sensor is a class to read data from a Json formatted

String. Location, battery and reading of sensors are kept within this class. Meanwhile, SensorDetails does the same thing except that it reads the coordinate of sensors. It stores them in a class called Coordinates which has 2 variable, lng storing longitude and lat storing latitude.

MapReader.class

MapReader does the same thing as SensorReader but for the no fly zone instead of sensors. A Json formatted String from one of the methods of DataReceiver are processed by this class to a List of GeoJson Polygon. Each polygon contains the coordinates of the vertex of a building. This list is then converted into a List containing a list of coordinates.

PathFinder.class

This class contains the algorithm of the drone's flying mechanism. First, it computes the distance between sensors and stores them in a table. Then, using the table, it calculates the order to visit sensors using methods from NearestNeighbour.class. Then, the drone visits the sensors according to the order and stores the flight path as coordinates in an array. If there are some sensors that somehow can't be reached, the sensors will be stored in an array for the use of outputting a text file at the end of execution of the program.

Output.class

This class is used for outputting a text file and a geojson map. With the data from SensorReader and a list of unvisited sensors generated by PathFinder, it generates a GeoJson map by turning the coordinates of sensor to a GeoJson features, each containing the

details of sensors as GeoJson String property. These includes location, size of marker, rgb string, color of marker and symbol of marker. Unvisited sensors won't have a marker symbol. Flight path is generated by turning the list of coordinates from PathFinder into a list of GeoJson Point. Then, the list of Point is converted into a GeoJson LineString, eventually to a GeoJson feature. This feature is then added into the GeoJson map generated previously. The map is then written into a file. The list of coordinates from PathFinder is also used to generate the text output file. Coordinates in the list is converted to a String. Then, the String is written into a file.

NearestNeighbour.class

This class is used for computing the order of visiting sensors. It uses repetitive randomized nearest neighbor(rnn) algorithm to compute the order. For each sensor, 3 nearest neighbors are set as options and one of them is randomly chosen. Then, the result is further optimized by 2-opt algorithm to get a better order. Depending on the number of iterations, nearest neighbor without randomization nor repetitive(nn) can produce a better result. But rnn can be better than nn most of the time if number of iterations is set as 100 or above. A default of 500 iterations is being used. It will return nn if nn is found to be better than rnn after 500 iterations. If iteration is set to be 0, it will return nn.

Class Documentation

App.class

Attribute:

Visibility and type	Name	Description
Private String	date	Representing the date inputted from command line
Private String	month	Representing the month inputted from command line
Private String	year	Representing the year inputted from command line
Private String	longitude	Representing the longitude inputted from command line
Private String	latitude	Representing the longitude inputted from command line
Private String	seed	Representing the seed inputted from command line
Private String	port	Representing the port inputted from command line

Method:

1. main(String[]):
    a. visibility: public
    b. return: none
    c. description: methods that execute the program

DataReceiver.class

Attribute:

1. HttpClient client:
    a. visibility: private
    b. type: HttpClient
    c. description: an object to receive data from the webserver

Method:

1. get(String):
    a. visibility: private

    b. return: String
    c. Description: Used by all the other methods in this class to receive data from webserver
2. no_fly_zones(String):
    a. visibility: public
    b. return: String
    c. Description: Reads a geojson file representing the no fly zones
3. air_quality(String, String, String, String):
    a. visibility: public
    b. return: String
    c. Description: Reads a json file containing the sensors reading, location and battery
4. sensor_detail(String, String):
    a. visibility: public
    b. return: String
    c. description: Reads a json file containing the coordinates of sensors

SensorReader.class

Class:

Public	Sensor	String, String, double	Containing the location, battery and readings of sensors
Public	SensorDetails	Coordinates	Containing the coordinate of sensors
Public	Coordinates	double, double	Containing the longitude and latitude of sensor

Method:

1. read_aq_data(String, String, String, String):
    a. visibility: public
    b. return: ArrayList<Sensor>
    c. description: Convert sensors in Json formatted String to a list of Sensor
2. read_coordinates(ArrayList<Sensor>, String):
    a. visibility: public
    b. return: ArrayList<SensorDetails>
    c. description: Get a list of SensorDetails from a list of sensor
3. read_sensor_details(String):

    a. visibility: private
    b. return: SensorDetails
    c. description: Get a SensorDetails class using the location of sensor

MapReader.class

Methods:

1. nf_buildings(String):
    a. visibility: private
    b. return: List<Polygon>
    c. description: Get a list of Polygon, each containing coordinates of vertex of buildings
2. get_coordinates()Polygon:
    a. visibility: private
    b. return: ArrayList<List<Double>>
    c. description: Get the coordinate of building as a list of double with size 2 from a Polygon object
3. nf_buildings_coordinates(String):
    a. visibility: public
    b. return: ArrayList<ArrayList<List<Double>>>
    c. description: Convert a list of Polygon to an array containing a list of coordinates where each coordinate is representing as a list of double with size 2

PathFinder:

Attribute:

1. xorigin, yorigin:
    a. visibility: private
    b. type: final double
    c. description: initial longitude and latitude of drone
2. lng, lat:
    a. visibility: private
    b. type: double
    c. description: current longitude and latitude ofdrone
3. sensor_coordinate:
    a. visibility: private
    b. type: ArrayList<SensorDetails>
    c. description: list of coordinates of sensors

4. buildings:
    a. visibility: private
    b. type: ArrayList<ArrayList<List<Double>>>
    c. description: list of list of coordinates of buildings in no fly zones
5. fly_distance:
    a. visibility: private
    b. type: double
    c. description: length of a move of drone
6. range:
    a. visibility: private
    b. type: double
    c. description: range of sensor to be read
7. moves:
    a. visibility: public
    b. type: int
    c. description: remaining available moves of drone
8. unvisited:
    a. visibility: public
    b. type: ArrayList<Integer>
    c. description: unvisited sensors after the whole journey in terms of their number
9. read:
    a. visibility: public
    b. type: HashMap<Integer, Integer>
    c. description: key indicates the number of moves and value indicates the number of sensor in which drone is reading after doing this move.

Method:

1. fly():
    a. visibility: public
    b. return: ArrayList<ArrayList<Double>>
    c. description: Visit all sensors according to the order calculated and return flight path as an list of coordinates and angle.
2. fly_to(double, double, ArrayList<ArrayList<Double>>):
    a. visibility: private
    b. return: none

    c. description: a helper function for fly() to move to a destination specified as argument, add the flight path to the input array.
3. distance():
    a. visibility: private
    b. return: double[][]
    c. description: Return a table which element (i, j) is the distance between sensor i and sensor j.
4. cal_dist(double, double, double, double, int):
    a. visibility: private
    b. return: double
    c. description: calculate the distance between 2 points recursively with a iteration limit of 3.
5. alternate_path(ArrayList<Integer>):
    a. visibility: private
    b. return: double[]
    c. description: Given the buildings that has crossed in an array, find the biggest and smallest longitude and latitude.
6. lowest_cost(double[], double, double, double, double, int):
    a. visibility: private
    b. return: double
    c. description: Given the corners of a rectangular, calculate the distance between point0 and one of the corners + point1 and the corner. If straight line between point1 and the corner crosses no fly zone, calculate the distance between the corner and one other corner + the other corner and point1.
7. crossing_buildings(double, double, double, double):
    a. visibility: private
    b. return: ArrayList<Integer>
    c. description: Check if a straight line from point0 to point1 crosses no fly zone
8. crossed_boundary(8 doubles):
    a. visibility: private
    b. return: Boolean
    c. description: Check if a straight line crosses the boundary of a building. Boundary of building is represented as a linear equation.
9. line_equation(double, double, double, double):
    a. visibility: private

    b. return: double[]
    c. description: Construct a linear equation given 2 points.
10. Euclidean(double,double,double,double):
    a. visibility: private
    b. return: double
    c. description: find the Euclidean distance between 2 points

Output.class

Attribute:

1. color_table:
    a. visibility: private
    b. type: String[]
    c. description: an array storing color of marker for sensor.
2. symbol:
    a. visibility: private
    b. type: String[]
    c. description: an array storing symbol of marker for sensor.

Methods:

1. Initialize_color():
    a. visibility: private
    b. return: none
    c. description: set up color_table
2. txt_file(ArrayList<ArrayList<Double>>, 5 Strings, HashMap<Integer, Integer>, ArrayList<Sensor>)
    a. visibility: public
    b. return: none
    c. description: Output a text file containing the flight path of drone.
3. draw_path(ArrayList<ArrayList<Double>>, 3 Strings)
    a. visibility: public
    b. return: String
    c. description: Output a Json formatted map with flight path given a map.
4. write_map(4 Strings)
    a. visibility: public
    b. return: none
    c. description: output a GeoJson file given input Strings

5. generate_map(ArrayList<Sensor>, ArrayList<SensorDetails>, ArrayList<Integer>)
    a. visibility: public
    b. return: String
    c. description: output a Json formatted String given list of sensors, list of sensors coordinates, list of unvisited sensors.

NearestNeighbour.class

Attribute:

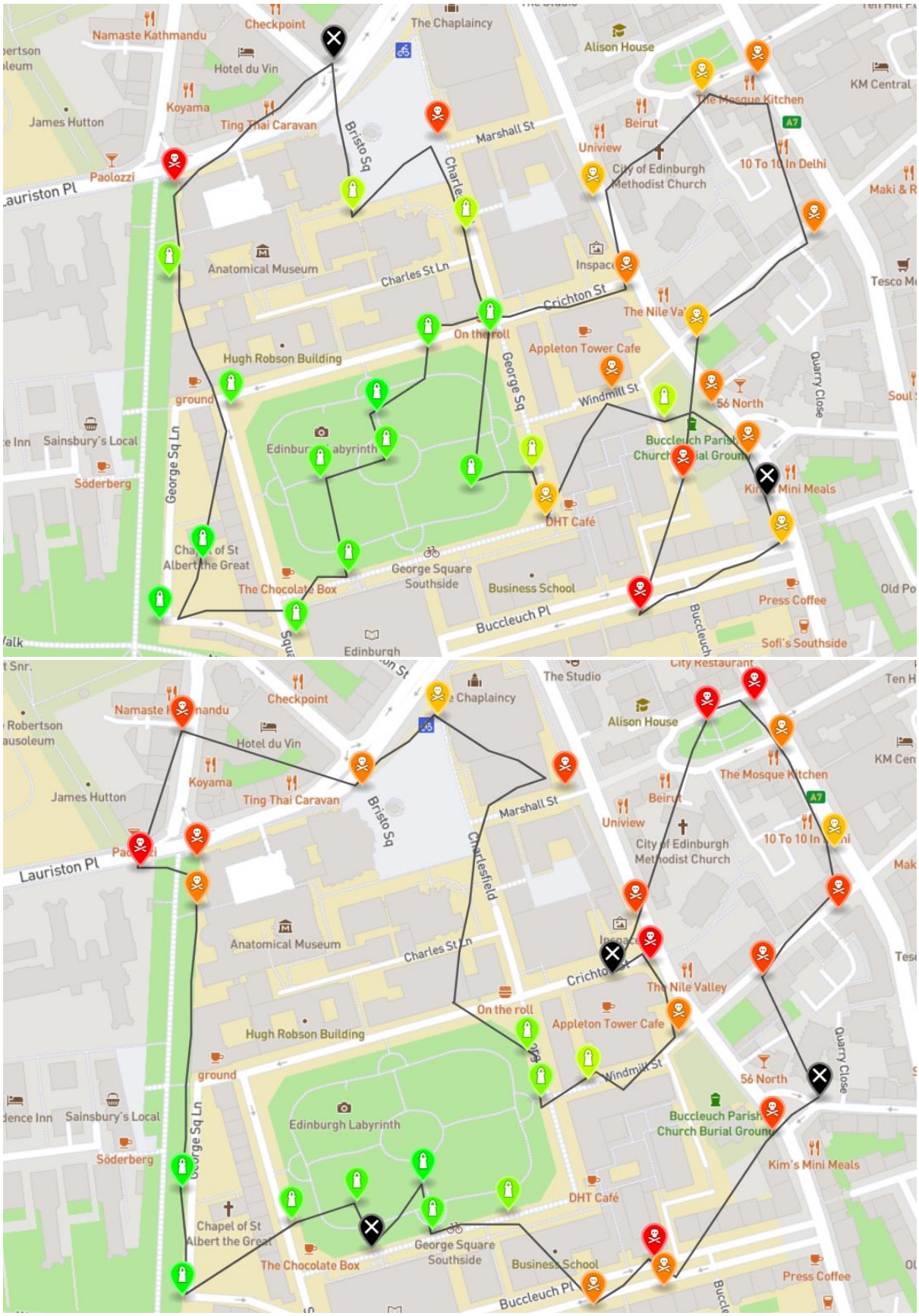
1. distance:
    a. visibility: private
    b. type: double[][]
    c. description: table of distance between sensors, element (i, j) = distance between sensor i and sensor j. 0 indicates origin.
2. n:
    a. visibility: private
    b. type: int
    c. description: number of sensors + 1 (origin)

Method:

1. nn():
    a. visibility: public
    b. return: int[]
    c. description: nearest neighbor with 2-opt, return order to visit sensors
2. rnn(int, int)
    a. visibility: public
    b. return: int[]
    c. description: repetitive randomized nearest neighbor with 2-opt, return order to visit sensors
3. twoOpt(int[])
    a. visibility: private
    b. return: int[]
    c. description: 2-opt algorithm to further optimize the result of nearest neighbors
4. cal(int[])
    a. visibility: private

b. return: double
c. description: calculate the total cost of visiting all sensors according to the input order.

Drone Control Algorithm



Before finding the order to visit the sensors, my algorithm find the distance between sensors using a specific algorithm which is:

1. Check if the straight line crossing the destination and current location intercept with any of the no fly zone.
2. If no, distance is the Euclidean distance between these 2 points
3. Otherwise, find out the biggest and smallest of longitude and latitude of the no fly zone.
4. Move to one of the corners, then repeat from step 1.
5. Only the path with shortest distance will be chosen.

The algorithm to check if a straight line crosses no fly zone is:

1. Compute the linear equation of boundaries of no fly zone using the formula: y = mx + c where y is latitude, x is longitude, m is gradient and c is a constant.
2. Check the interception of 2 lines is on the line between the vertex of no fly zone and on the line between current location and destination.
3. If no, return an array with size 0
4. Otherwise, return an array in which its element is the number of buildings in no fly zone

After finding the distance table, my algorithm computes the order to visit each sensor by using repetitive randomized nearest neighbor with 2-opt.

1. Starting with initial location, compute 3 nearest neighbor of current sensor (or location).
2. Randomly choose one of them as the next sensor to visit.
3. Iterate until there are 2 sensors left.
4. Randomly choose one of them as the next sensor to visit.
5. The remaining sensor will be the last sensor to visit before flying back to initial location.
6. The result is the further optimized by 2-opt.

7. For each sensor, swap the position of the sensor with all the other sensors.
8. Compute the total distance to travel all sensors after each swap.
9. Undo the swap if the distance is bigger than distance before the swap.
10. After doing 2-opt, repeat this whole procedure again from step 1.
11. Repeat this for n (default is 500) times, output the order with smallest distance.
12. Finally, compute the order once again using 1 nearest neighbor with 2-opt.
13. If 1-nn has smaller distance, return its result. Otherwise, return 3-nn.

After computing the order, my drone now starts to visit the sensors/

1. Iterate according to the order computed above and start visiting each sensor.
2. Compute the angle between current location and destination using inverse tangent and coordinates.
3. Find possible location after a move using sine, cosine, the length of a move and the angle calculated.
4. Increase the angle by a multiple of 10 starting from 0 to 350.
5. If the move is valid as in the move won't result in crossing the boundary of no fly zone, find the move that results in smallest distance between next location and destination. (method to determine crossing the boundary of no fly zone is same as the one when calculating distance)
6. Repeat from step 1 until the Euclidean distance between current location and destination is smaller than 0.0002.