INFORMATICS
LARGE PRACTICAL

Coursework 2 Report

Peong Wei Zhen

S1852034

TABLE OF CONTENT

Software Architecture Description

This software is basically run with a main class called App, with series of sub-classes called Drone , _tour , Position, and txt. While class Sensor and Words are used to interact with the webserver to get data needed.

A series of sub-classes are presented in the package. The reasoning of each sub-classes is specified as follows:

Sub-class	Reasoning
Sensor	This class is used as an object type to store the attributes of each sensor from the webserver for specific day.
Words	This class is used as an object type to store the attributes of each sensor which is represented in What3Words from the webserver for specific day. Each of 33 What3words from the Sensor classes are passed to the webserver to get the exact coordinates of the sensors located.
Drone	The one of the most important sub-class in this implementation. This class stores the Drone attributes such as current coordinates and angle to the next sensor. Drone class has some useful function too which helps to decide how the drone moves in the map.
_tour	_tour class is used as an object type to implement the algorithms for this project.
Position	Position class has two attributes *lng* and lat, which represents the coordinates of every moves that the done has moved.
txt	For each simulation, two files should be outputted (one Geo-json file and one text file). A class is created to store all the attributes needed for the text file.

❖

Class Documentation

<table>
<tr><th colspan="2">App</th></tr>
<tr>
<td>Description:
▪ The class that contains the main method. When the jar file is run, the main method inside this class is executed.
▪ Contains most of the important functions including reading and writing files, algorithms of the implementation and categorization of the reading of sensors.</td>
<td>Field:
❖ client : HttpClient
➢ Http Client to deal with the webserver.
❖ lnglat : double[ ][ ]
➢ Coordinates of each sensor in type of double.
❖ fl : ArrayList<Feature>
➢ Store the list of Feature to be converted to Geo-json output file.
❖ fcSensors : FeatureCollection
➢ FeatureCollection to be converted to Geoj-json output file.
❖ category : String [ ]
➢ Reading of each sensor from the Sensor class in String form.
➢ “null” and “NaN” means no reading when low battery.
❖ categorydouble : double [ ]
➢ Reading from category which converted to type double.
➢ “null” ,“NaN” and reading for low battery stored as -1.
❖ properties : String[ ][ ]
➢ Features of each sensor for Geo-json output file which contains “location”, “rgb-string”/”marker-color”, and “marker-symbol”.
❖ polygon0 : double[ ][ ]
➢ Appleton Tower.
❖ polygon1 : double[ ][ ]
➢ David Hume Tower.
❖ polygon2 : double[ ][ ]
➢ Main Library.
❖ polygon3 : double[ ][ ]
➢ Informatics Forum.
❖ initial : double[ ]
➢ Starting coordinates of the drone.
❖ city1 : double[ ]
➢ The order of sensor to be visited before passing the algorithm.</td>
</tr>
</table>

	❖ city2 : double[ ] ➢ The order of sensor to be visited after passing the algorithm. ❖ drone : Drone ➢ Drone. ❖ visit : int [ ] ➢ Check if the sensor is visited. ➢ 1 for yes,0 for no. ❖ dronepath : ArrayList<Position> ➢ Stores the coordinates for each move of the drone. ❖ sensorname : String[ ] ➢ Name of the sensor in the order before passing algorithm. ❖ visitname : String[ ] ➢ Name of the sensor in the order after passing algorithm. ❖ txtstring : ArrayList<txt> ➢ List of txt storing the output for generating output text file. ❖ tour : _tour ➢ Object _tour for swapping algorithm. ❖ newTour : _tour ➢ Object _tour for swapping algorithm.

Methods:

i. ReadFile(String[ ] args)
   - Return the String of *response.body()* from the maps folder in webserver.

ii. fromJsonSensor(String x)
   - Return an ArrayList of object type Sensor from a String from the *response.body().*

iii. ReadCoordinates(String x, String[] args)
   - Return the String of *response.body()* for the specify What3Words from the words folder in webserber.

iv. fromJsonWords(String x)
   - Return a Words object from a String from the *response.body().*

v. ReadGeojson(String[]args)
   - Return the String of *response.body()* from the buildings folder in webserver.

vi. fromGeojson (String x)
   - Passing the coordinates of the points of each no-fly-zone to corresponding polygon0, polygon1, polygon2, polygon3.

vii. Category(ArrayList<Sensor> lst)
   - Passing and categorize the readings of the sensors to the array *properties.*

viii. setpath()
   - Copy the coordinates from array l*nglat* to array *city1* with adding the starting coordinates of the drone at the beginning and the end of the array *city1.*

ix. TwoOptSwap(int i, int k)
   - Swapping for TwoOpt algorithm.

x. TwoOpt()
   - TwoOpt algorithm.

xi. setvisitname()
   - Passing the name of sensors in What3Words form to *visitname*.

xii. dronepath(double[ ]initial)
   - Implement how the drone fly from initial until the end.

xiii. SensorMap(double[ ][ ] lnglat)
   - Generate the features of the sensors and save in the FeatureCollection *fcSensors*.

xiv. CreateFile(String[ ] args)
   - Create a specify Geo-json file.

xv. WriteFile(String[ ] args)
   - Write the Geo-json file.

xvi. CreateFiletxt(String[ ] args)
   - Create a specify Text file.

xvii. WriteFiletxt(String[ ] args)
   - Write the Text file.

xviii. main(String[] args)
   - Main function.

Sensor

Description:
- Each Sensor object corresponds to an air-quality-sensor on the map.
- Initially, the sensors are store as Features in Json File, making updating the properties a difficult process.
- The Sensor class simplifies this process.

Fields:
- location : String
  - What3Word which representing the location of the sensor.
- battery : double
  - Real value expressing the percentage battery charge between 0.0 and 100.0.
- reading : String
  - Character String represent a real value between 0.0 and 256.0.
  - “NaN” and “null” for reading of low battery.

Methods:
- None.

Words

Description:

- Each Words object corresponds to the What3Words of an air-quality-sensor on the map.
- Initially, the features are store as Features in Json File, making updating the properties a difficult process.
- The Words class simplifies this process.
- Only coordinate is used in Words class.

Fields:

- country : String
  - Not used
- coordinates : coordinate
  - Has two attributes *lng* and *lat* representing the coordinates of the sensor.
- square : square
  - Not used.
- nearestPlace : String
  - Not Used.
- words : String
  - Not Used.
- language : String
  - Not Used.
- map : String
  - Not Used.

Methods:

- None.

Drone

Description:

- The class representing a drone in this project.
- Drone class contains all the necessary attributes of a drone and also the function needed for a drone to move in the flight path in order to collect the sensors' readings.

Fields:

- initial : double [ ]
  - Initial coordinates of a drone for starting a flight path.
- current : double [ ]
  - Current coordinates which the drone located.
- direction : int
  - Angle in multiples of 10 degrees that a drone move.
- direction1 : int
  - Previous direction used for comparison
- polygon0 : double [ ] [ ]
  - Coordinates of a no-fly-zone.
- polygon1 : double [ ] [ ]
  - Coordinates of a no-fly-zone.
- polygon2 : double [ ] [ ]
  - Coordinates of a no-fly-zone.

	❖ polygon3 : double [ ] [ ] ➢ Coordinates of a no-fly-zone.

Methods:

i. double[ ] getInitial()
   - Return initial.

ii. setInitial (double[] initial)
   - Set the initial coordinates of the drone.

iii. getCurrent ()
   - Return current.

iv. setCurrent (double [ ] current)
   - Set the current coordinates of the drone.

v. getDirection()
   - Return direction.

vi. setDirection(int direction)
   - Set the direction that the drone moves.

vii. DistnextSensor(double [ ] current, double [ ] nextSensor)
   - Return the Euclidean distance between current and nextSensor in double type.

viii. getnext(int angle, double [ ] current)
   - Return the coordinates of the drone after a move in double [ ] type.

ix. int getAngle(double[ ] current, double[ ] nextSensor)
   - Return the angle of the drone from current to next sensor which to be collected readings

x. onSegment(MapPoint p, MapPoint q, MapPoint r)
   - Given three colinear MapPoints p, q, r, the function checks if MapPoint q lies on line segment 'pr' and return in Boolean.

xi. orientation(MapPoint p, MapPoint q, MapPoint r)
   - To find orientation of ordered triplet (p, q, r).
   - The function returns following values
   - 0 --> p, q and r are colinear
   - 1 --> Clockwise
   - 2 --> Counterclockwise

xii. doIntersect (MapPoint p1, MapPoint q1, MapPoint p2, MapPoint q2)
   - The function that returns true if line segment 'p1q1' and 'p2q2' intersect.

xiii. isIntersect (MapPoint polygon[], int n, MapPoint p)
   - Check if the path intersect the polygon.

xiv. isInside (MapPoint polygon[], int n, MapPoint p)
   - Returns true if the MapPoint p lies inside the polygon[ ] with n vertices.

xv. isInside_nfz (double[] current)
   - Check if drone is inside any no-fly-zones.

xvi. putframe()
   - Generate the drone confinement area.

xvii. move(double[ ] crr, double[ ] nextSensor)
   - Drone moves one move and updates current coordinates.

_tour	
Descriptions: ▪ This class is created to carry out the TwoOpt algorithm in an easier way.	Fields: ❖ City : double[ ] [ ] ➢ The coordinates which a drone needs to go in order to collect readings and back to initial points. ❖ tourDistance : int ➢ Total Euclidean distance of the whole flight path connecting all the sensors.
Methods: i. GetCity (int x) ▪ Return City with index x. ii. SetCity(int x, double[] city) ▪ Set the City with index x to city. iii. TourSize() ▪ Return the length of *City*. iv. Euclidean (double[ ] current, double [ ] nextSensor) ▪ Return the Euclidean distance between current coordinates and nextSensor coordinates. v. TourDistance() ▪ Total amount of distance needed to have a closed loop in this tour.	

Position	
Descriptions: ▪ This class is used to store the longitude and latitude of each move of the drone	Fields: ❖ lng : double ➢ longitude ❖ lat : double ➢ latitude
Methods: ▪ None.	

<table>
<tr><th colspan="2">txt</th></tr>
<tr><td>Descriptions:
▪ Class used to store attributes which need to be used to generate text output file.</td><td>Fields:
❖ count : int
➢ Count of moves made by drone.
❖ crr : double [ ]
➢ Current coordinates of the drone.
❖ angle : int
➢ Direction of drone moves.
❖ nxt : double [ ]
➢ Next Coordinates of the drone.
❖ sensor : String
➢ Name of sensor in What3Words if visited in specify move.</td></tr>
<tr><td colspan="2">Methods:
i. getCount()
▪ Return count.
ii. setCount(int count)
▪ Set the count to specific interger.
iii. getCrr()
▪ Return crr.
iv. setCrr(double[ ] crr)
▪ Set crr to the specific coordinate.
v. getAngle()
▪ Return angle.
vi. setAngle(int angle)
▪ Set angle to specific direction.
vii. getNxt()
▪ Return nxt.
viii. setNxt(double[ ] nxt)
▪ Set nxt to specific coordinate.
ix. getSensor()
▪ Return sensor.
x. setSensor(String sensor)
▪ Set the sensor to specific String.</td></tr>
</table>

Drone Control Algorithm

Aim : To collect as many sensors as possible and back to original location within 150 moves

Note: The fields of class are in bold and italic

Strategy:

1. For all the coordinates of those 33 sensors available on the map , save them in an array with size of 35 called ***city1*** with an initial coordinate at the starting and ending of the flight path.

2. At first, I try to get the shortest path connecting all 33 sensors without considering the no-fly-zones on the map.

3. In this project, a famous algorithm called 2-Opt algorithm is used. This is a simple local search algorithm for solving the traveling salesman problem. In this algorithm, the processes are basically split into two parts. Here we just call them “swap” and “replace”.

4. The “swap” part has the following implementation as following:

```
procedure 2optSwap(route, i, k) {
    1. take route[0] to route[i-1] and add them in order to new_route
    2. take route[i] to route[k] and add them in reverse order to
new_route
    3. take route[k+1] to end and add them in order to new_route
    return new_route;
}
```

5. Here is an example of the above with arbitrary input:

   - Example route: A → B → C → D → E → F → G → H → A
   - Example parameters: i = 4, k = 7 (starting index 1)
   - Contents of new_route by step:
     1. **(A → B → C)**
     2. A → B → C → **(G → F → E → D)**
     3. A → B → C → G → F → E → D → **(H → A)**

6. The “replace” part, which is also the complete 2-opt swap making use of the “swap” mechanism.

```
repeat until no improvement is made {
    start_again:
    best_distance = calculateTotalDistance(existing_route)
    for (i = 1; i <= number of nodes eligible to be swapped - 1; i++)
{
        for (k = i + 1; k <= number of nodes eligible to be swapped;
k++) {
            new_route = 2optSwap(existing_route, i, k)
            new_distance = calculateTotalDistance(new_route)
            if (new_distance < best_distance) {
                existing_route = new_route
                best_distance = new_distance
                goto start_again
            }
        }
    }
}
```

7. Since we need to start and end at the ***initial***, we do not put the first and last element of ***city1*** array into the “swap” mechanism. This algorithm will continue to swap until the end of the loop and replace the ***city1*** whenever a shorter path is counted. The total length of the path is calculated based on an addition of Euclidean Distance between each two nodes. The formula of Euclidean Distance is as shown below:

distance between (x_1, y_1) and (x_2, y_2) is just

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

8. After getting the order of sensors to be visited, the drone can start moving. First from the starting point and before every move, the drone will check the ***city1*** array to get the coordinates of the sensor in turn to be visited. Thus, a function called ***getAngle*** is created to find the nearest multiples of 10 degrees to the coordinates of next sensor to be visited. We first get the exact angle between two points, then divides it by 10.0 , round off to an integer and multiplies 10 back. For example, if the angle is 74.3, it will return a 70.

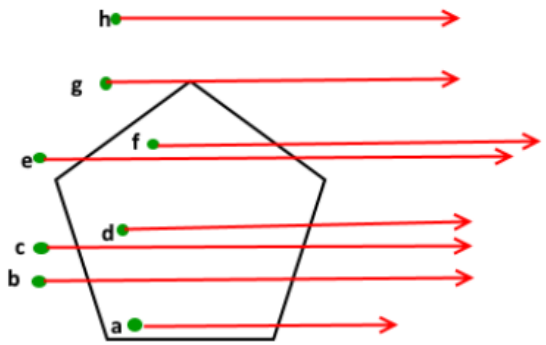
9. After having the angle, moves the drone at the particular angle and a distance of 0.0003 and get a coordinate that the drone will be reaching by simple trigonometry calculation with longitude is equal to *(distance)*(sin(angle in radian)) + current longitude* and latitude is equal to *(distance)*(cos(angle in radian)) + current latitude*

10. Then we need to check if the points fall in an “illegal” area. We thus create another function called ***isInside*** determine whether the point is inside the no-fly-zones. The idea used in this function is as shown below:

```
1) Draw a horizontal line to the right of each point and extend it to infinity

1) Count the number of times the line intersects with polygon edges.

2) A point is inside the polygon if either count of intersections is odd or
   point lies on an edge of polygon.  If none of the conditions is true, then
   point lies outside.
```

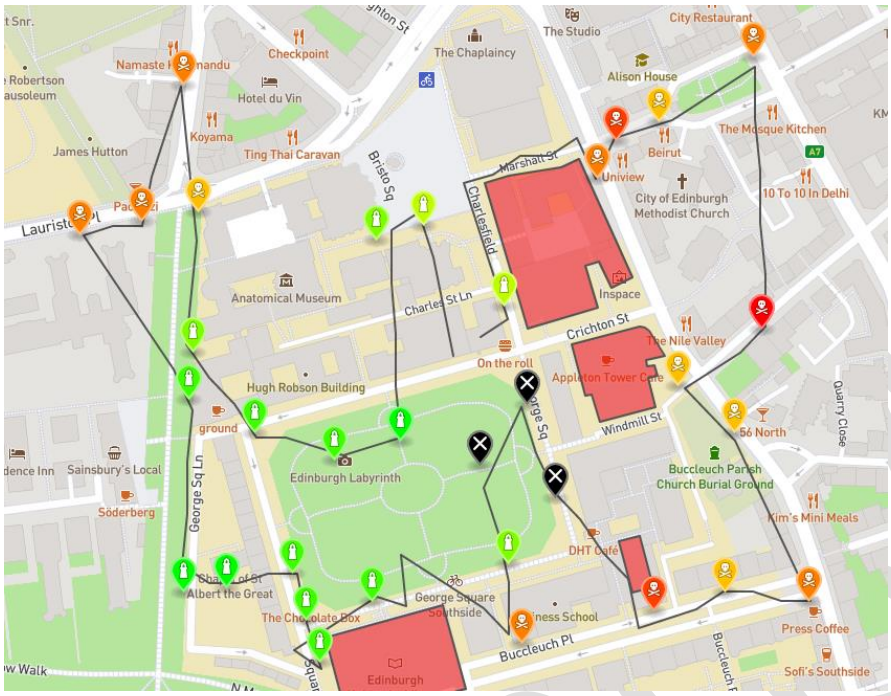


11. If the next point is inside the polygon, it changes it angle by subtracting 30 degrees. As the coming ***direction*** will always be available for the drone to moves out, I put a conditional statement when the ***direction*** of next move of the drone is opposite of the previous move, ***direction*** will be subtracted by 70 degrees which tested to be one of the best combination to escape from some “dead” angle.

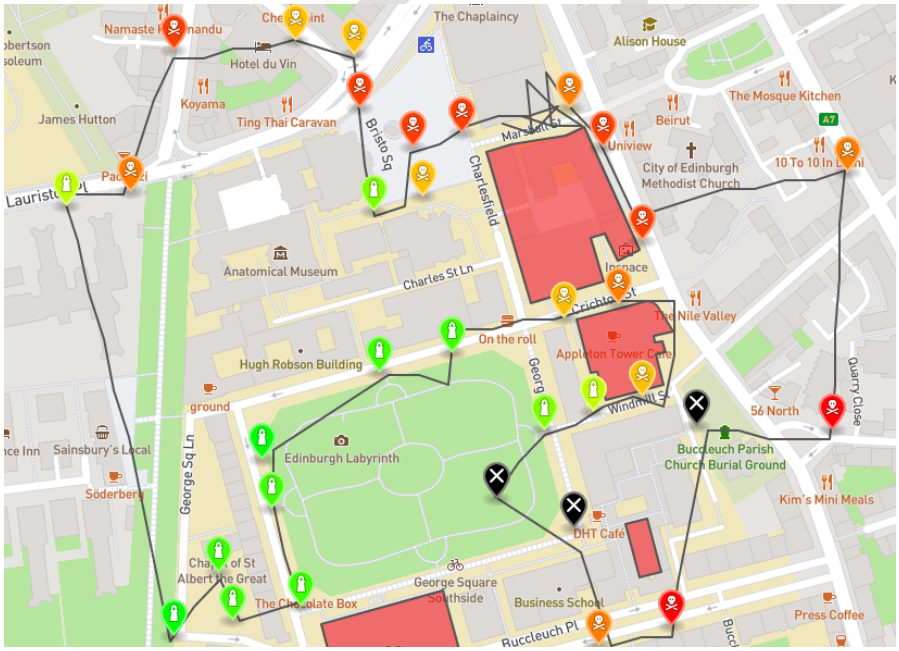
12. Another case to be considered is even if the next coordinate is outside of the no-fly-zones, the path connecting two nodes might still intersect the edges of no-fly-zones. For this case, I put an if statement so that if the line connecting the current and next move intersecting the edges, this angle will not be used too.

13. The last thing to be considered is whether the drone is in the drone confinement area. I just use the same ***isInside*** function to make sure the drone will not fly to the outside of the confinement area.

14. When the drone reach a point which its Euclidean distance with the target sensor less than 0.0002 degrees, the sensor is marked visited and the target sensor will be updated to the next one until we reach the end which has a Euclidean distance with the ***initial*** less than 0.0003degrees.



This drone's flight path shown on the 09/04/2020. We can notice that when the angle is blocked by the red no-fly-zones, the drone turns its direction and continues its journey and finally reaches back to the initial point with a nice and smooth path.



This drone's flight path shown on the 09/04/2021. We can notice that the drone got stuck for some moves at the top right corner of the map. This might because the drone adjusted itself for a few times to escape from the "dead" angle to get to the next sensor.