INFR09051------Informatics Large Practical

Written Report

Name: Liu Rongxing

UUN: s1810054

1. Software Architecture Description

2. Class documentation

3.Drone Control Algorithm

4. References

1. Software Architecture Description

My software consists of eight classes:

① **Sensor:** This class stores the information of a sensor, including its word3address, longitude-latitude coordinate, battery and reading. It has no other methods apart from simple getters and setters. I think this class is necessary because we must be able to get the accurate information about any sensors anytime.

② **SensorCollection:** This class stores the list of sensors to be visited on a given day. It is closely related to **Sensor**. It acquires data from the server and uses the data to instantiate **Sensor** one by one. **SensorCollection** will help calculate flight paths and the generate the final text file and geojson file.

③ **NoFlyZone:** This class stores the four no-fly-zones by acquiring relevant data from the server. It also has a method of checking for intrusion to the no-fly-zones given a start point and an end point. This class is necessary as it helps monitor the flight paths of the drone.

④ **Sequence:** This class finds a sequence of sensors to be visited of which the total distance travelled is minimum. It consists of two heuristics. It is necessary as it is part of the algorithms for determining the flight paths. Since I broke down the algorithms into two parts---one for deciding the sequence of sensors generally and the other for designing a specific path from one sensor to another, I used this class alone for deciding the sequence.

⑤ **AStarPoint:** Since I am using the A* search algorithm, instead of using the class **Point** in the mapbox package, I created this class to store the information of a point separately. Besides the essential values of longitude and latitude of the point, I also included other important attributes of the point such as its neighboring points and its f, g values. All the information will be used in the A* search algorithm later.

⑥ **ComparePoints:** This class implements the comparator and compares the two **AStarPoint** by comparing their **f** value. As I am using a PriorityQueue in the A* search algorithm, this implementation will help constructing the PriorityQueue.

⑦ **AStarSearch:** This class is used for finding a specific path from one location to another using the A* search algorithm, which is the core of the entire software.

⑧ **Generator:** This class gathers the results obtained from other classes together, synthesize them for the final generation of both the text and the geojson files. For example, it firstly finds the list of sensors to be visited, then find the sequence of visiting the sensors and creates a path for each pair of adjacent sensors. Finally, it outputs the list of points travelled in the entire journey, the direction turned at each move and the nearest sensor (could be null) after each move.

⑨ **App:** This class is for the final execution of the software. It extends the class **Generator**,

which means it inherits all the data in **Generator**. It will then use all these data to generate the text file and geojson file as required.

2. Class documentation

All global variables are in blue. All parameters are in orange.

2.1 Sensor

(1) Inheritance or Interface: None

(2) Variables:

Name	Visibility	Type	Description
word3address	private	String	This is the What3Words address of the sensor.
reading	private	double	This is the reading of the sensor.
location	private	Point	This is the location of the sensor in terms of its longitude and latitude.

(3) Constructor:

public Sensor (String address, double reading, Point location)
Description: Set the value of word3address, reading and location accordingly
Parameters: address: What3Words address of the sensor reading: reading of the sensor location: coordinate location of the sensor

(4) Methods:

public Point getLocation ()
Description: Get the coordinate location of the sensor
Returns: location

public double getReading ()
Description: Get the reading of the sensor
Returns: reading

Public String getAddress ()
Description:

Get the What3Words address of the sensor
Returns: word3address

2.2 SensorCollection

(1) Inheritance or Interface: None

(2) Variables:

Name	Visibility	Type	Description
sensors	private	List<Sensor>	Sensors to be visited on a given date
client	private	HttpClient	Used to send requests and retrieve their responses

(3) Constructor:

public SensorCollection (int port, int yy, int mm, int dd)
Description: Connect to the server and navigate to the maps json file with the given date. Then navigate to the detailed json file with the corresponding What3Words address for each sensor to get the coordinate location of the sensor. Store all the necessary information for each sensor and put all the sensors to the list. If the battery is below 10%, the reading should not be taken, so just record as -1.
Parameters: port: port to be connected to on the server yy: given year mm: given month dd: given day
Throws: IOException, InterruptedException

(4) Methods:

public JsonElement getJson (String path)
Description: Acquiring the data on the website specified by path by sending http requests.
Parameters: path: The address of the website.
Throws: IOException, InterruptedException

Public List<Sensor> getSensors ()
Description: Get the list of sensors to be visited on a given date
Returns: sensors

2.3 NoFlyZone

(1) Inheritance or Interface: None

(2) Variables:

Name	Visibility	Type	Description
zones	private	List<Polygon>	Polygons of the four no-fly-zones
client	private	HttpClient	Used to send requests and retrieve their responses

(3) Constructor:

public NoFlyZone (int port**)**
Description: Connect to the server and obtain the no-fly-zones json file. Get the four polygons and put them to a list.
Parameters: port: Port to be connected to.
Throws: IOException, InterruptedException

(4) Methods:

public boolean checkIntersect (Point start**, Point** end**, Point** p1**, Point** p2**)**
Description: Check whether the line segment joining start and end intersects with the line segment joining p1 and p2
Parameters: start: coordinate of one end of the first line segment end: coordinate of the other end of the first line segment p1: coordinate of one end of the second line segment p2: coordinate of the other end of the second line segment
Returns: true if two line segments do not intersect, false otherwise

public boolean checkNoFlyZone (AStarPoint starta**, AStarPoint** enda**)**
Description: Apply **checkIntersect** for the line segment joining starta and enda against all sides of all four polygons
Parameters: starta: coordinate of one end of the first line segment enda: coordinate of the other end of the first line segment

Returns:
true if the line segment does not intersect with any side of the four polygons, false otherwise

public List<Polygon> getZones ()
Description:
Get the list of four no-fly-zone polygons
Returns: zones

2.4 Sequence

(1) Inheritance or Interface: None

(2) Variables:

Name	Visibility	Type	Description
coordinates	private	List<Point>	Coordinates of sensors
n	private	int	Number of sensors
distance	private	double[][]	Distance between sensors
perm	private	int[]	Optimal permutation of sensors

(3) Constructor:

public Sequence (List<Point> points**)**
Description:
Load points and calculate the optimal permutation of points
Parameters:
points: coordinates of sensors

(4) Methods:

public void setDistance ()
Description:
Calculate the distance between each pair of points in coordinates, and store the distance between $i \; and \; j$ in distance[i][j]

public int mod (int i**, int** mod**)**
Description:
Prevent occurrence of negative number by converting it to its modulo.
Parameters:
i: the number we are converting mod: the modulus we are taking
Returns:
the modulus of i under mod. If i is negative, keep adding mod until it becomes positive.

public boolean trySwap (int i)
Description: Try to swap the sequence of the i^{th} point and the $(i+1)\%n^{th}$ point in the permutation.
Parameters: i: index of the point to be swapped
Returns: true if the swap gives a shorter total distance and apply the swap, false otherwise

public void reverse (int start, int end)
Description: Reverse the order of elements from start to end in current permutation
Parameters: start: the start position of the part of the permutation to be reversed end: the last position of the part of the permutation to be reversed

public boolean tryReverse (int start, int end)
Description: Try to reverse the order of elements from start to end
Parameters: start: the start position. end: the last position
Returns: true if the reversal gives a shorter total distance and apply the reversal, false otherwise

public void SwapHeuristic ()
Description: Apply **trySwap** repeatedly, if we find a shorter path while going through a loop, we will start a new loop until no shorter path is found

public void ReverseHeuristic ()
Description: Apply **tryReverse** repeatedly, if we find a shorter path while going through a loop, we will start a new loop until no shorter path is found

public int[] getPerm ()
Description: Get the optimal permutation of sensors to be visited
Returns: perm

2.5 AStarPoint

(1) Inheritance or Interface: None

(2) Variables:

Name	Visibility	Type	Description
theta	private	double	Base angle drone turns in
longitude	private	double	Longitude of the point
latitude	private	double	Latitude of the point
f	private	double	f value of the point
g	private	double	g value of the point
previous	private	double	Preceding point to form the shortest path

(3) Constructor:

public AStarPoint (double lng**, double** lat**)**
Description: Set the longitude and latitude of the point
Parameters: lng: longitude of the point, lat: latitude of the point

(4) Methods:

public List<AStarPoint> getNeighbors ()
Description: Create a list of the 36 neighbors of the point
Returns: a list of neighbors

2.6 ComparePoints

(1) Inheritance or Interface:

This class implements the interface Comparator<AStarPoint>.

(2) Variables: None

(3) Constructor: None

(4) Methods:

public int compare (AStarPoint a**, AStarPoint** b**)**
Description: Order AStarPoint by comparing their f values, which will be used in PriorityQueue.
Parameters: a: first AStarPoint to be compared b: second AStarPoint to be compared
Returns: -1 if f values of a is smaller than f values of b, 0 if equal, 1 otherwise

2.7 AStarSearch

(1) Inheritance or Interface: None

(2) Variables:

Name	Visibility	Type	Description
noflyzones	private	NoFlyZone	Used for checking no-fly-zones when finding the path
cutoff	private	int	Quota of the PriorityQueue

(3) Constructor:

public AStarSearch (int port**)**
Description: Set up the no-fly zones
Parameters: port: port to be connected to
Throws: IOException, InterruptedException

(4) Methods:

public double euclidDistance (AStarPoint p1**, AStarPoint** p2**)**
Description: Find the Euclidean distance between two points
Parameters: p1: the first AStarPoint p2: the second AStarPoint
Returns: the Euclidean distance between p1 and p2

public boolean inRange (AStarPoint p**, AStarPoint** goal**)**
Description: Check whether the point is within 0.0002 degree of the goal
Parameters: p: the current point we are at goal: our target position
Returns: true if p is within 0.0002 degree of goal, false otherwise

public List<Point> findPath (AStarPoint start**, AStarPoint** end**)**
Description: Find a path from start to end using A* search algorithm
Parameters: start: starting position end: target position
Returns: a list of points to be passed by from start to end

public List<Point> constructPath (AStarPoint p**)**
Description: Find the path all the way till point p
Parameters: p: the last point in the path
Returns: a list of points as a path from some starting point to p

2.8 Generator

(1) Inheritance or Interface: None

(2) Variables:

Name	Visibility	Type	Description
flightpaths	protected	List<Point>	List of points that the drone passes for the entire journey
directions	protected	List<Integer>	Angle the drone flies in each time
sensorspath	protected	List<Sensor>	Sensor visited in the order determined by "perm"
total_sensorspath	protected	List<Sensor>	Nearest sensor after each move
n	protected	int	Number of sensors

(3) Constructor:

public Generator (int port**, int** yy**, int** mm**, int** dd**, Point** start**)**
Description: Acquire information of the sensors on the server. Apply the two heuristics to get the optimal permutation of sensors. Find the nearest sensor to start with. Apply A* search algorithm to find the entire path. Find the directions turned in each move.
Parameters: port: port to be connected to yy: given year mm: given month dd: given day start: starting position of the drone
Throws: IOException, InterruptedException

(4) Methods:

public double euclidDistance (AStarPoint p1**, AStarPoint** p2**)**
Description: Find the Euclidean distance between two points
Parameters: p1: the first AStarPoint p2: the second AStarPoint
Returns: the Euclidean distance between p1 and p2

public void generatePaths (int port**, Point** start**)**
Description: Compute flightpaths where the drone passes through sensors one by one and returns to its starting position. Compute total_sensorspath at the same time
Parameters: port: port to be connected to start: starting position of the drone
Throws: IOException, InterruptedException

public void setDirections (List<Point> path)
Description: Calculate the angle the drone flies in. Add the angles to directions
Parameters: path: list of points as the drone's path (flightpaths)

2.9 App

(1) Inheritance or Interface:

This class is a subclass of the parent class **Generator**

(2) Variables:

Same as **Generator**

(3) Constructor:

Same as **Generator**

(4) Methods:

public String getColor (double n)
Description: Get the color corresponding to the reading,expressed as a rgb-string
Parameters: n: reading taken by the drone
Returns: the corresponding rgb-string of the reading

public String getSymbol (double n)
Description: Get the marker-symbol corresponding to the reading
Parameters: n: the reading taken by the drone
Returns: the corresponding marker-symbol of the reading

public void generateText (int yy, int mm, int dd)
Description: Generate the required text file on the given date
Parameters: yy: given year mm: given month dd: given day

public void generateJson (int yy, int mm, int dd)
Description: Generate the required geojson file on the given date
Parameters: yy: given year mm: given month dd: given day

3.Drone Control Algorithm

3.1 Explanation

Instead of trying to find the optimal path directly, which would be extremely complicated, I broke down the algorithms into two parts---one for deciding the sequence of sensors generally and the other for designing a specific path from one sensor to another.

The first part looks like a **Travelling Salesman Problem**. There are mainly two ways of solving this problem. The first way is the greedy approach, which is to choose the nearest sensor each time, travel to that sensor and repeat the process. This method is easier to understand and execute but its result might not be ideal. Hence, I used another method which is the combination of a few heuristics that usually gives a shorter path than greedy approach. (**Reference** ①)

In the first of these heuristics (called the "Swap Heuristic"), we explore the effect of repeatedly swapping the order in which a pair of adjacent cities are visited, as long as this swap improves (reduces) the cost of the overall tour. As a result, we would reach a **local minimum**--this does not imply that we will have a global **optimum**

Let $v_1, v_2, \cdots, v_n$ denote all the vertices (with each vertex representing a sensor) and let $a_1, a_2, \cdots, a_n$ denote the permutation of the vertices, with $a_i \in \{1,2, \cdots n\}$.

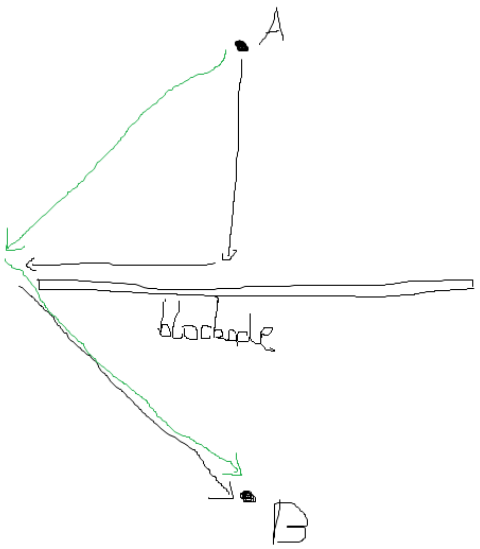
We first set the permutation as the default permutation with simply $a_i = i$. Then for each value of $i\ from\ 1\ to\ n$, we would try to swap the order of $a_i\ and\ a_{(i+1)\%n}$ and check whether the total distance travelled becomes smaller. If the value is smaller we would swap $a_i\ and\ a_{(i+1)\%n}$,otherwise we keep the original ordering and move on. **Notice that we are not doing this for only one loop.** Whenever there is an update of the ordering while going through a loop, we would start a new loop after that until we can see no improvement from these swaps.

The TwoOpt Heuristic is another heuristic which repeatedly makes "local adjustments" until there is no improvement from doing these. In this case the alterations are more significant than the swaps - this method repeatedly nominates a contiguous sequence of cities on the current tour, and proposes that these be visited in the reverse order, if that would reduce the overall cost of the tour. For each pair of value $(i, j)\ with\ 1 \leq j \leq n-1\ and\ 1 \leq i \leq j$, we would consider the effect of reversing the segment from $a_i\ to\ a_j$ (in other words, to change $a_i\ to\ a_j, a_{i+1}\ to\ a_{j-1}\ and\ so\ on$. If this reversal will make the total distance travelled (strictly) shorter, we would commit to it, otherwise we move on. **Again, we are not doing this for only one loop of $i\ and\ j$.** Whenever there is a reversal committed while going through a loop, we would start a new loop until we can see no further improvement.

In conclusion, we start with a default permutation of vertices (sensors), apply Swap Heuristic followed by TwoOpt Heuristic to the permutation and we would get a sequence of sensors to be travelled.

Now that we have our sequence, we need to travel from one sensor to another. Usually we could just travel in the straight line, which is the fastest way. However, in the context there

are some no-fly zones that our drone is not supposed to cross.



As can be seen from the figure above, suppose we are going to travel from A to B. Normally we would start travelling in the direction towards B which is shown by the black arrows and we would succeed if there is no blockade between A and B. However, when we realize that there is actually a blockade, we would have to take a detour and the total distance travelled will definitely be longer than the green ones which is determined in advance.

Hence, I decided to use the A* search algorithm. (**Reference** ②) The main idea of using A* search algorithm is that of all the points we can travel to each time, we find a point $\boldsymbol{v}$ with lowest $\boldsymbol{f}$ $\boldsymbol{where}$ $\boldsymbol{f = g + h}$. Here g is the cost of travelling from the starting point to $\boldsymbol{v}$ and $\boldsymbol{h}$ is the estimated cost of travelling from $\boldsymbol{v}$ to the goal. In this context, I used the Euclidean Distance as the metric to determine h. We repeat the process until reaching the end.

We firstly create a list which stores the potential points as our candidate. Then we find the point **(**call it **current)** with the smallest $\boldsymbol{f}$. We can use a **PriorityQueue** to store those candidate points because we just need to call the method **poll()** to get that specific point, which is very efficient. After having found **current**, we would check through each point (call it **neighbor**) of its neighbors with the condition that the drone does not intrude any no-fly-zones when flying from **current** to **neighbor**. We set $\boldsymbol{g}$ of **neighbor** as $\boldsymbol{g\ of\ current + 0.0003}$; then calculate $\boldsymbol{f}$ of **neighbor**; set **current** as the point immediately preceding the neighbor on the cheapest path from the start; and add **neighbor** into the candidate list. The process is repeated until we have reached the target point or there is no more point in the list to be evaluated. Sometimes during the computation, we might happen to have too many points in the **PriorityQueue,** which slows down the computation drastically. To

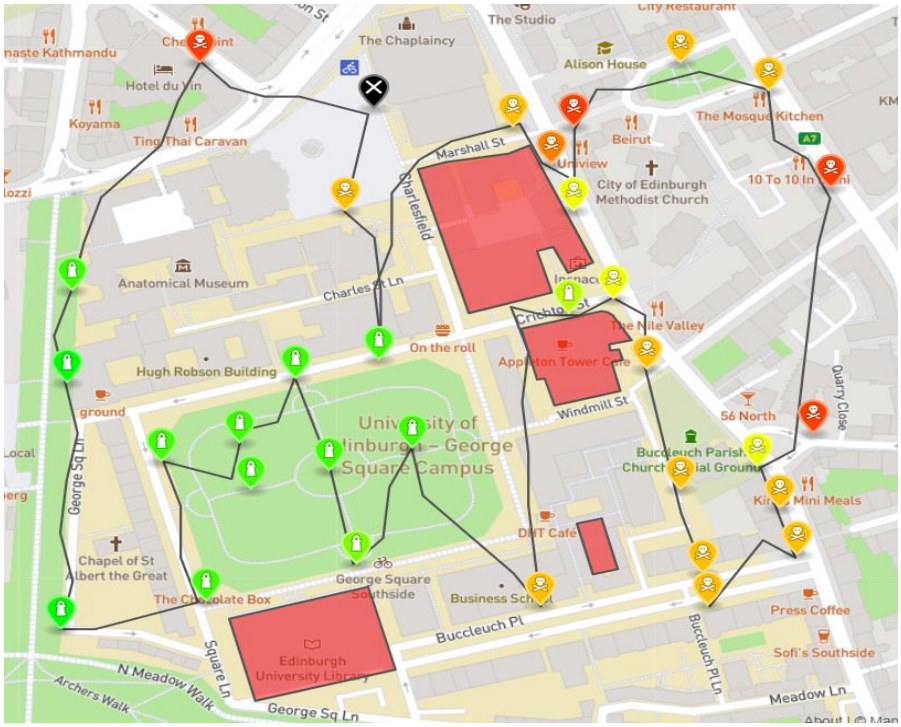
prevent this, we could set a quota for the **PriorityQueue.** If the size of the **PriorityQueue** exceeds the quota, we will take the point with lowest f as our new starting point, record the path from the starting point to that point, start a new loop and finally concatenate the paths we obtained along the way together.

When we reach the target point, we keep acquiring its preceding points and in the end we will have a list of points, which is exactly the path we are looking for.

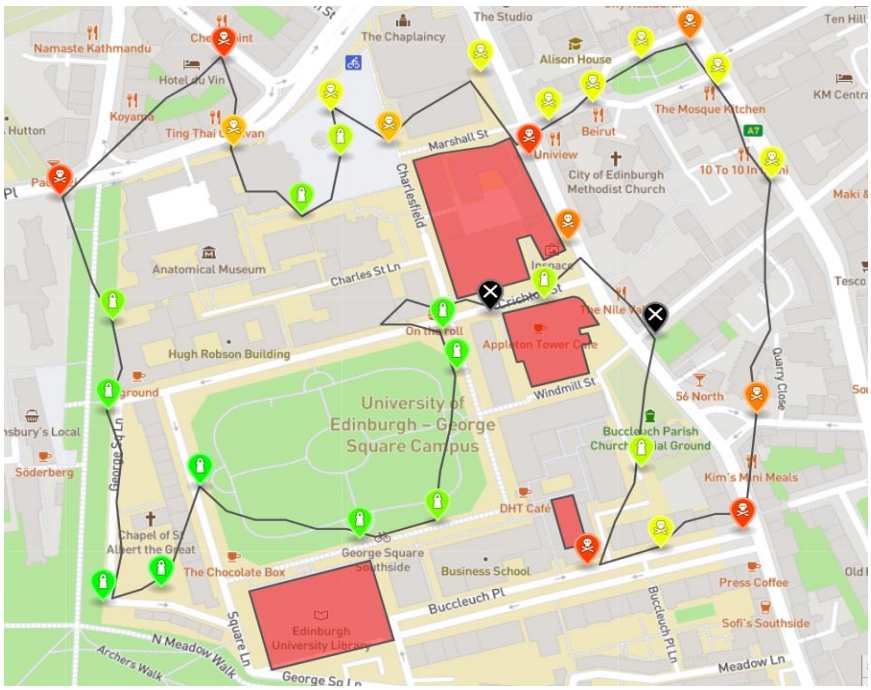
The drone is designed to fly a fixed amount of distance 0.0003 degree each time, so it is extremely difficult to make it fly to the exact point. Instead, since the drone can take reading as long as it is within 0.0002 degree from the sensor, we would consider a point reaching the goal when it is within 0.0002 degree from the goal. This would be applied when travelling from one sensor to another as well as returning to the starting point.

3.2 Graphical Illustration

This figure below shows the flight paths and markers on 2020-01-01. As can be seen from the figure, the path is very efficient as there is hardly any “back and forth” path. The drone also does not intrude any no-fly zones (sometimes it flies along the edge, but never enters).



This figure below shows the flight paths and markers on 2020-02-02. As can be seen from the figure, the path takes advantage of the fact that the drone can take reading anywhere within 0.0002 degree around the sensor. Hence, it seems that the drone is never close to the sensor “laws.merit.bleak”, but it came within 0.0002 degree around the sensor.



4. References

① Course INFR08026, Introduction to Algorithms and Data Structures, Coursework problem sheet 3, March 2020

②Introduction to A*. (n.d.). Retrieved December 03, 2020, from http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html