Informatics Large Practical Report

Andrew Scollin - S1842899

December 2, 2020

0 Introduction

For the Informatics Large Practical we were tasked to program an autonomous drone which will collect readings from air quality sensors from around an urban geographical area as part of a research project to analyse urban air quality.

The main challenges of the project were :

- Processing input GeoJSON data from locally served data.
- Implementing a pathfinding algorithm to guide the drone between sensors.
- Finding a suitable order for the drone to visit each sensor for a given day.
- Formatting GeoJSON and text data for output.

The sensors to be visited and their readings are stored on a local web server. These 'maps' are test data and predictions for the sensors readings, in practice the drone would follow the path from one of these predicted maps.

Specification This project was to be made in Java 11 on a Maven framework. The project was to be packaged as a JAR file that would take the following arguments :

- The Day, Month and Year of the map to be tested.
- The starting coordinates of the Drone.
- A seed for the randomiser used in our project (for consistent results)
- The port at which the webserver is being hosted.

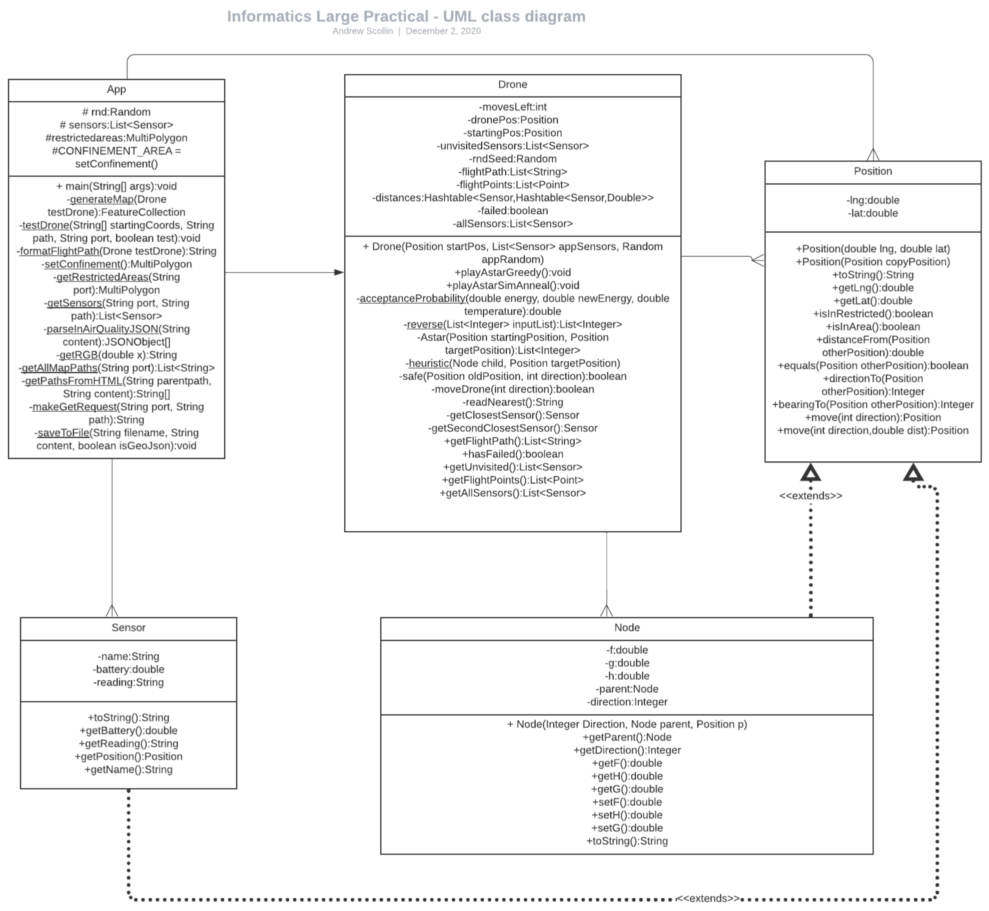
And would output two files :

- A GeoJSON file showing the sensors read as Point objects and the flightpath of the drone as a LineString object.
- A text file with each of the moves taken by the drone and the readings obtained each move.

The drone can only read a sensor if within 0.0002 degrees. The drone is to move and then read a sensor - it cannot not read multiple sensors at once. It was also limited to 150 moves of 0.0003 degrees and was to avoid flying through no fly zones or outside the confinement area specified in the hosted data and specification. After completing a tour of all the sensors, the drone was also to return within 0.0003 degrees of the start point.

1 Software Architecture Description

Close consideration was made to the specifications outlined above when choosing the architecture of the Classes.


Informatics Large Practical - UML class diagram
Andrew Scollin | December 2, 2020
App
# rnd:Random
# sensors:List<Sensor>
#restrictedareas:MultiPolygon
#CONFINEMENT_AREA = setConfinement()
+ main(String[] args):void
-generateMap(Drone testDrone):FeatureCollection
-testDrone(String[] startingCoords, String path, String port, boolean test):void
-formatFlightPath(Drone testDrone):String
-setConfinement():MultiPolygon
-getRestrictedAreas(String port):MultiPolygon
-getSensors(String port, String path):List<Sensor>
-parseInAirQualityJSON(String content):JSONObject[]
-getRGB(double x):String
-getAllMapPaths(String port):List<String>
-getPathsFromHTML(String parentpath, String content):String[]
-makeGetRequest(String port, String path):String
-saveToFile(String filename, String content, boolean isGeoJson):void
Drone
-movesLeft:int
-dronePos:Position
-startingPos:Position
-unvisitedSensors:List<Sensor>
-rndSeed:Random
-flightPath:List<String>
-flightPoints:List<Point>
-distances:Hashtable<Sensor,Hashtable<Sensor,Double>>
-failed:boolean
-allSensors:List<Sensor>
+ Drone(Position startPos, List<Sensor> appSensors, Random appRandom)
+playAstarGreedy():void
+playAstarSimAnneal():void
-acceptanceProbability(double energy, double newEnergy, double temperature):double
-reverse(List<Integer> inputList):List<Integer>
-Astar(Position startingPosition, Position targetPosition):List<Integer>
-heuristic(Node child, Position targetPosition)
-safe(Position oldPosition, int direction):boolean
-moveDrone(int direction):boolean
-readNearest():String
-getClosestSensor():Sensor
-getSecondClosestSensor():Sensor
+getFlightPath():List<String>
+hasFailed():boolean
+getUnvisited():List<Sensor>
+getFlightPoints():List<Point>
+getAllSensors():List<Sensor>
Position
-lng:double
-lat:double
+Position(double lng, double lat)
+Position(Position copyPosition)
+toString():String
+getLng():double
+getLat():double
+isInRestricted():boolean
+isInArea():boolean
+distanceFrom(Position otherPosition):double
+equals(Position otherPosition):boolean
+directionTo(Position otherPosition):Integer
+bearingTo(Position otherPosition):Integer
+move(int direction):Position
+move(int direction,double dist):Position
<<extends>>
Sensor
-name:String
-battery:double
-reading:String
+toString():String
+getBattery():double
+getReading():String
+getPosition():Position
+getName():String
Node
-f:double
-g:double
-h:double
-parent:Node
-direction:Integer
+ Node(Integer Direction, Node parent, Position p)
+getParent():Node
+getDirection():Integer
+getF():double
+getH():double
+getG():double
+setF():double
+setH():double
+setG():double
+toString():String
<<extends>>

1.1 App

This is the main App. It is the entry point once our project is packaged into a JAR file and does all the necessary retrieval and parsing of data to set up the rest of our project. This class also holds the restricted areas and confinement zone attributes that are public to the rest of the package for use in Position, Sensor and Node.

The app class has methods for making GET requests to the web server and for saving files to the working directory. The setup carried out by App involves getting the restricted areas, setting the confinement zone and reading all the sensors. It also has methods for formatting the data stored on the drone for the two necessary output files.

As described later in section 3 - Drone Algorithm - a two step method was implemented to cover edge cases where a Greedy approach was not sufficient. The App class is where this process is carried out. testDrone(..) will create a drone controller to be tested on Greedy A* and (if required) create a new drone controller to carry out our Simulated Annealing A*. The decision to not merge both algorithms into one on drone and to instead invoke them in App was made to make trouble shooting the algorithms easier and to reduce complexity working with the drone controllers attributes.

1.2 Drone

This is the Drone controller class. It is the key component in the project that runs the algorithms for traversing the Sensors. This class acts as if it was a drone itself, keeping track of : the moves left, starting position and a record of all drones to be visited/to be visited. The decision was made to have the Drone class not be dependant on the App that implements it, this makes it easy to test a small sample of sensors in testing or if for whatever reason there was not exactly 33 sensors to be visited. The Drone class still references the no fly zones directly this is because we assume there is only one no-fly zone file specified.

Acting within the constraints of what a the 'real' Drone could do this drone can only alter its Position by increments of 0.0003 degrees and can realise **all** of the given sensors throughout its flight so to find a suitable path to each of them.

The algorithms used to traverse the sensors are stored on the drone. Both use A* path finding to traverse from one sensor to another however the main algorithm uses a Greedy sensor order and the other a Simulated Annealing approach. This is talked about in detail in section 3 - Drone Algorithm.

1.3 Position

This is the Position class. This was made to represent positions within the confinement zone and to implement common methods for working with coordinates such as euclidean distance and direction calculations.

It has longitude and latitude attributes to represent coordinates on a GeoJSON map. The methods to determine whether these coordinates collide with the no fly zones or are inside the confinement area are dependant on the Apps confinement zone and restricted area attributes. This decision was made as there is only one file defined for no-fly zones and the confinement zone is hard coded therefore passing each of these MultiPolygon into each Position object wouldn't make sense and is more efficient to just reference them in the package.

The Position class also has the method for moving a certain distance in a certain direction. This makes the Drone class simpler as we therefore only need to worry about the directions we can take, not the calculations involved in moving.

1.3.1 Node

This is the Node Class that extends Position. This was made to work within the A* path finding algorithm coded into Drone. The decision to have it extend Position was made because the nodes in an A* path finding problem represent the positions that can be taken from the start to the target and useful methods exist in Position that can be used in drone such as the distance calculation.

It has the attributes of Position, g, h, f and parent attributes that are used within the general A* algorithm to compute an appropriate path. The path taken is easily backtracked as is the case with general A*, but in this implementation we also included a direction attribute that holds the direction taken from parent to child. This because a direction is the only metric used by our drone to follow the correct path so rather than returning each Nodes position and working out the direction between each we can trivially just take the directions to fly the path.

1.3.2 Sensor

This is the Sensor Class that extends Position. This was made to represent the Sensors to be visited by the drone. The decision to make this class extend Position was made as a Sensor is located at a specific and constant location and their direction towards and distance need to be easily calculated in our algorithm.

It has the attributes of position, a name representing the what3words string of the square that it belongs to, a reading and a battery. This data is read in from the maps and sensor folders on the WebServer by the App class where we instantiate all our Sensors.

Considerations There were a few things could've been done differently whilst keeping the structure relatively similar. The main considerations made during the software design process were :

- The drone class could reference the sensors and random seed directly from the App class. This could've been done in the final build as the software would only use all the map paths sensors anyway. This however was not changed to keep maintenance and future developments easier to test on a smaller set of sensors.

- Position could be an interface instead of super class for Node and Sensor. This decision could be implemented quite easily, however it wouldn't lead to much change as there is no need for Sensor or Node to have any further sub classes. For our task at hand this choice was made because it is simple to visualise a Sensor or Node as a Position when working with coordinates in this way, so this did not require multiple implementations of interfaces.
- Generation of the readings map and flight path text file could be kept on the Drone class instead of App. This was actually a change made **after** implementing the Drone class. The data was originally generated and formatted in the Drone class but it was deemed more simple and realistic for the App to retrieve the 'raw' data that the drone would keep a record of after the flight, and format it then than to have a public method in Drone that could be invoked whenever.

2 Class Documentation

2.1 App

2.2 Drone

2.3 Position

2.4 Sensor - extends Position

2.5 Node - extends Position

NB:

- isInArea() and isInRestricted() found in Position (and subsequently its sub classes Node and Sensor) reference variables in the App class directly. This means that they must be declared in main before a they are invoked.
- For the methods of this project to access the WebServer data we assume that the WebServer will have the same sub folder schema and use the same software as provided in the specification.

- Class Documentation

2.1 App

Return Type	Function Name and input arguments
-	main(String[] args)
FeatureCollection	generateMap(Drone testDrone)
-	testDrone(String[] startingCoords, String path, String port, boolean test)
String	formatFlightPath(Drone testDrone)
MultiPolygon	getRestrictedAreas(String port)
List<Sensor>	getSensors(String port, String path)
JSONObject[]	parseinAirQualityJSON(String content)
String	getRGB(double x)
List<String>	getAllMapPaths(String port)
String[]	getPathsFromHTML(String parentpath, String content)
String	makeGetRequest(String port, String path)
-	saveToFile(String filename, String content, boolean isGeoJson)

2.1.1 public static void main(String[] args)

This is the entry point for the packaged JAR file. The input arguments are : the day, month and year of the map you are wanting to test, the starting latitude and longitude of the drone, a seed for the random number generator and the port of the web server that is hosting the data we are testing. This method will run the algorithm to compute the drones moves for the data it is passed and save the necessary files to the working directory.

2.1.2 private static FeatureCollection generateMap(Drone testDrone)

This function will generate the output FeatureCollection that will be converted to JSON for saving. The input argument is the drone with the map that we are wanting to generate. This is invoked in out testDrone function.

2.1.3 private static void testDrone(String[] startingCoords, String path, String port, boolean test)

This function is for running the algorithm specified in section 3, given a path to a map. The input arguments are : The starting coordinates of the drone we are testing, the path to the map we are wanting to test, the port where the data is being served locally and a boolean to specify whether we are running a test or not it attempt to run the algorithms and to save the necessary output files with the names matching the map date we are testing. If we are running a test specified by the boolean 'test', the output files will be named "testfp.txt" and "testr.geojson" to make it easier for testing individual maps once at a time.

2.1.4 private static String formatFlightPath(Drone testDrone)

This function will format the flight path attribute of the Drone it is passed. This is used in the testDrone method for formatting the flightpath text output file.

2.1.5 private static MultiPolygon getRestrictedAreas(String port)

This function will retrieve the no fly zones specified on the webserver. It takes the port at which the webserver is being hosted locally.

2.1.6 private static List<Sensor> getSensors(String port, String path)

This function will return the sensors the drone is required to visit for a certain map which it is passed. The input arguments are the port the server is running at and the path of the 'map' we are wanting to retrieve the Sensors for.

2.1.7 private static JSONObject[] parseinAirQualityJSON(String content)

This method is used in getSensors to parse in the air-quality-data.json file read in from the webserver. The input arguments are the content from the get request and the output are each of the JSON objects for each sensor represented in an array of org.json.JSONObject for easy property manipulation.

2.1.8 private static String getRGB(double x)

This function returns the RGB code for a specified sensor reading. It's input arguments are the sensor reading x and it outputs the RGB code used in the GeoJSON rgb-string as a String.

2.1.9 private static List<String> getAllMapPaths(String port)

This function is not used in the main execution of this project but is useful for testing purposes. It polls the webserver hosted at the port specified in the input arguments to return all sub folders paths in maps/ as a List of String. This can then for example : be written to a csv file then looped over in the command line to test the correctness of this program as a JAR file.

2.1.10 private static String[] getPathsFromHTML(String parentpath, String content)

This function is for use in the getAllMapPaths function. It takes in a parent path representing the directory we are polling and then the HTML content as a String that was returned from the GET request of this path. It returns an array of string where each element is the complete path of the folders specified inside the parent path.

2.1.11 private static String makeGetRequest(String port, String path) throws IOException

This function will throw an IO exception if the URL connection is not valid. This method will take a the port of the webserver hosted locally and the path that is needing a get request made to it, and will carry out the get request and return the content that is recieved.

2.1.12 private static void saveToFile(String filename, String content, boolean isGeoJson)

This function will save and/or overwrite a file with the content specified as a string in the input arguments. The filename you are trying to save is specified in the input arguments, if this file exists already and has the required permissions it will be overwritten if not a new file with the filename will be made. The boolean input is to specify whether the input content is GeoJSON, that is : a GeoJSON object that has been represented by .toJson(), This boolean if asserted will format the JSON correctly for saving to an external file.

- Class Documentation

2.2 Drone

Constructor :	Drone(Position startPos, List<Sensor> appSensors, Random appRandom)

Methods :

Return Type	**Function Name and input arguments**
-	playAstarGreedy()
double	acceptanceProbability(double energy, double newEnergy, double temperature)
-	playAstarSimAnneal()
List<Integer>	reverse(List<Integer> inputList)
List<Integer>	Astar(Position startingPos, Position targetPosition)
double	heuristic(Node child, Position targetPosition)
boolean	safe(Position oldPosition, int direction)
List<String>	getFlightPath()
boolean	moveDrone(int direction)
String	readNearest()
Sensor	getClosestSensor()
Sensor	getSecondClosestSensor()
boolean	hasFailed()
int	getMoves()
List<Sensor>	getUnvisited()
List<Point>	getFlightPoints()
List<Sensor>	getAllSensors()

2.2.1 public Drone(Position startPos, List<Sensor> appSensors, Random appRandom)

Constructor class for the Drone controller, Takes the starting position of drone, A list of sensors to be toured and the Random object for generating consistent random values. Returns the Drone object with initial values for attributes respecting those in the input arguments.

2.2.2 public void playAstarGreedy()

This methodn attempts the Greedy ordering and the Astar pathfinding algorithm on the drone and will update the drone to have followed the tour generated by Greedy using A* pathfinding.

2.2.3 private static double acceptanceProbability(double energy, double newEnergy, double temperature)

This method returns the acceptance probability used in the simulated annealing algorithm, will return 1 if the swapped tour value is strictly greater and the temperature weighted difference of values otherwise. The input arguments are : the current permutations tour value, the swapped tour value and the current temperature.

2.2.4 public void playAstarSimAnneal()

This method attempts the Simulated annealing ordering and the Astar pathfinding algorithm on the drone and will update the drone to have followed the tour generated by Simulated Annealing using A* pathfinding.

2.2.5 public List<Integer> reverse(List<Integer> inputList)

This method returns a reversed list of integers. It takes a list of integers that will be returned in reversed order. Used in the A* algorithm.

2.2.6 private List<Integer>Astar(Position startingPos, Position targetPosition)

This method returns a list of directions to be taken for the path between one position and another using the A* algorithm. Takes a starting position and target position.

2.2.7 private double heuristic(Node child, Position targetPosition)

Returns the heuristic for weighting A* nodes H values in the A* pathfinding algorithm. Takes the child node and the target position. Configured to the Manhattan heuristic with D = 3.

2.2.8 private boolean safe(Position oldPosition,int direction)

This method returns a boolean specifying whether a move is legal or not for the drone given a starting position of the drone 'oldPosition'. Takes the old position and the direction to move in. This method should be used to test a direction being passed to moveDrone before executing it.

2.2.9 public List<String> getFlightPath()

This method is used to return the List of flight path lines saved by drone during its flight. Returns the Drones flightPath attribute.

2.2.10 private boolean moveDrone(int direction)

This method moves the drone and reads a sensor, it returns true if the move was successful and false otherwise. Takes the direction in which the drone moves and assumes that the direction passed is safe for the drones current position.

2.2.11 private String readNearest()

This method returns the name of the closest sensor read by the drone and 'reads it' by taking it off the unvisited list. This method is used in the moveDrone method.

2.2.12 private Sensor getClosestSensor()

This method returns the closest sensor not yet visited by the drone based off of the drones current position. Used in the Greedy algorithm.

2.2.13 private Sensor getSecondClosestSensor()

This method is used for returning the second closest sensor not yet visited by the drone based off of the drones current position. Used in the Greedy algorithm.

2.2.14 public boolean hasFailed()

This method returns the failed attribute of the drone. If true the Drone was not successful in the flight otherwise it was.

2.2.15 public int getMoves()

This method returns the moves attribute of the drone specifying the moves the drone has left.

2.2.16 public List<Sensor> getUnvisited()

This method returns the unvisited sensors attribute of the drone, it specifies the sensors the drone is yet to / was supposed to visit.

2.2.17 public List<Point> getFlightPoints()

This method returns the flight points attribute of the drone. Each point represents the drones position at the start and after each move.

2.2.18 public List<Sensor> getAllSensors()

This method returns the all sensors attribute of the drone. This specifies all the sensors the drone was tasked to visit when instantiated.

- Class Documentation

2.3 Position

Constructor :	Position(double lng, double lat)
Constructor :	Position(Position copyPosition)

Methods :

Return Type	**Function Name and input arguments**
String	toString()
double	getLng()
double	getLat()
Boolean	isInRestricted()
Boolean	isInArea()
double	distanceFrom(Position otherPosition)
boolean	equals(Position otherPosition)
Integer	directionTo(Position otherPosition)
Integer	bearingTo(Position otherPosition)
Position	move(int direction)
Position	move(int direction, double dist)

2.3.1 public Position(double lng, double lat)

Constructor method for new Position, takes two doubles representing the longitude and latitude. Returns a Position object instantiated with the correct attributes.

2.3.2 public Position(Position copyPosition)

constructor for position that takes another position that is to be 'copied'. Returns a position identical in latitude and longitude to the position that was passed into it.

2.3.3 public String toString()

This method is used for representing a position as a string displaying the latitude and longitude.

2.3.4 public double getLng()

This method returns the longitude of the Position.

2.3.5 public double getLat()

This method returns the latitude of the Position.

2.3.6 public Boolean isInRestricted()

This method returns whether the position is inside the restricted no fly zones or not. References the Apps restricted areas attribute.

2.3.7 public Boolean isInArea()

method that returns whether the position is in the drone confinement zone. References the Apps confinement zone attribute.

2.3.8 public double distanceFrom(Position otherPosition)

This method calculates the distance between this position and another position that is passed as an argument. Returns a double value with the distance in degrees.

2.3.9 public boolean equals(Position otherPosition)

This function returns whether or not a positions longitude and latitude is equal to another positions longitude and latitude. Takes the other position as an argument.

2.3.10 public Integer directionTo(Position otherPosition)

This method calculates the direction from this position toward another position. The input argument is the other position we are getting the direction towards. Returns an integer specifying the direction from 0 to 35 with 0 being 0 degrees in our calculations.

2.3.11 public Integer bearingTo(Position otherPosition)

This method calculates the bearing taken for the drones flightpath text file output. The input argument is the other position used for calculating the bearing towards the old position to the drones new position. For this method East is 0deg, North is 90deg, West is 180deg and South is 270deg.

2.3.12 public Position move(int direction)

This method returns the position resulting if this position was to move in a certain direction 0.0003 degrees. The input argument is the direction in degrees/10.

2.3.13 public Position move(int direction, double dist)

This method returns the position resulting if this position was to move in a certain direction a variable distance. The input argument are the direction in degrees/10 and double with the value of the distance to move in degrees.

- Class Documentation

2.4 Sensor

Constructor :	Sensor(String name, Position sensorPosition, double battery, String reading)

Methods : List below excluding methods inherited from Positon

Return Type	**Function Name and input arguments**
String	toString()
double	getBattery()
String	getReading()
Position	getPosition()
String	getName()

2.4.1 Sensor(String name, Position sensorPosition, double battery, String reading)

Constructor method for a Sensor. Takes the sensors name (A what three words string), A position object representing the objects position this is passed as the super class constructor, a double representing the battery of the sensor and a string representing the reading of the sensor.

2.4.2 public String toString()

This method overrides that of the Position toString(). This method returns a string representation of the Sensor showing the name, location and battery.

2.4.3 public double getBattery()

This method returns the battery value of the sensor.

2.4.4 public String getReading()

This method returns the reading the sensor is giving.

2.4.5 public Position getPosition()

This method returns the Position object of the sensor.

2.4.6 public String getName()

This method returns the name of the sensor.

- Class Documentation

2.5 Node

Constructor :	Node(Integer direction, Node Parent, Position p)

Methods : List below excluding methods inherited from Positon

Return Type	**Function Name and input arguments**
Node	getParent()
Integer	getDirection()
double	getF()
-	setF()
double	getH()
-	setH()
double	getG()
-	setG()
String	toString()

2.5.1 public Node(Integer Direction, Node Parent, Position p)

Constructor for Node, the input arguments are the direction taken to get the node from the parent node - this is the degree direction / 10 - it takes the parent of the node, and the position of the node. It returns a Node object instantiated with the correct attributes specified in the arguments.

2.5.2 public Node getParent()

This method returns the parent node of this node.

2.5.3 public Integer getDirection()

This method returns the direction of the move taken from the parent to get to this node.

2.5.4 public double getF()

This method returns the F value of this node.

2.5.5 public void setF(double newf)

This method sets the F value of this node to a new value specified in the arguments. The input is a double with the new F value.

2.5.6 public double getH()

This method returns the H value of this node.

2.5.7 public void setH(double newh)

This method sets the H value of this node to a new value specified in the arguments. The input is a double with the new H value.

2.5.8 public double getG()

This method returns the G value of this node.

2.5.9 public void setG(double newg)

This method sets the G value of this node to a new value specified in the arguments. The input is a double with the new G value.

2.5.10 public String toString()

This method overrides the positions toString(). It returns a string representation of this node displaying the F value and direction taken to get to it.

3 Drone Algorithm

The Drones Algorithm can be broken down into two parts :

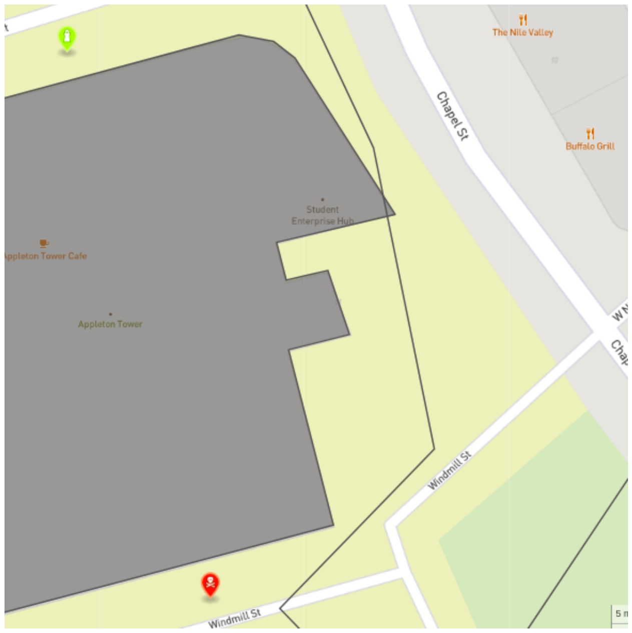
1. The order in which the Sensors are visited - The Sensor Order
2. The method for traversing the drone between these Sensors - Traversing the sensors

The Sensor Order Surprisingly enough, a Greedy implementation of this path generation works very well in the test data given to us. To begin the drone will choose the closest sensor then use A* to path find towards it, then once at the sensor and has read the sensor it will find the next closest sensor from its new position. There are some edge cases to be accounted for due to the fact that the drone cannot read and then read again without moving. So we implemented the functionality that if the Drone is close enough it will trivially move in any safe direction and then back again to read the Sensor or if its not close enough it will then trivially move towards it and read or if this move is not legal it will just fly to the second closest sensor or the starting position if there is no other sensors left to read.

Although the scope of our test data works perfectly well with Greedy from the research conducted when inspecting this problem it became clear that Greedy algorithms would be quite easily stumped when it comes to sparse sensor positions. This is why in the method devised : if greedy is unsuccessful, an attempt to find a path using simulated annealing will be made. We define a tour to be the order in which sensors are visited and the value of a tour to be the total distance between these sensors in the given order (our implementation was kept simple to optimise running time, so it doesn't account for the no fly zones in our ordering). Simulated Annealing is a probabilistic method of optimisation that - in our implementation - relies on randomly swapping two sensors to be visited in a tour and checking if there is an improvement in their tour value, in which case we pass the swapped order to be our new current order, or (depending on the decreasing temperature related to the difference in tour values) we pass a less favourable order to be our new current order. This helps the optimisation escape local dips in the values that can occur in our search for optimal solutions. We then simply move from each sensor to the next in our ordering, removing them from the order as we read them. This works however it faces the same challenges in edge cases. We tackle them similarly to the Greedy approach.

Traversing the Sensors This was easy enough to implement, once we have our path order all we have to do is path find between the two points : the drones current position and the position of the sensor. This was done using the A star search algorithm[1] using the Manhattan heuristic[3]. A* search is similar to Dijkstra's[2] in that we construct a graph of nodes representing positions, the nodes neighboring the current node are worked through until we reach our target position. A node will be considered if it has a lower f score which in our case depends on the number of moves taken and our heuristic for distance. Our heuristic optimises for the fewest number of 'squares' we can fit between our node and the target. This was necessary in our algorithm as there exist 36 possible neighbors for any node we are checking, if all of them were added to the open list from using a more optimal heuristic (such as the distance from a node to the target), the time of computation would be exponentially big.

The problem of determining whether a move is safe or not was a challenge in our traversing algorithm. This was a challenge as there are cases in which the Drone could fly completely through a building or clip the corner of a building like so ..



The method first used for determining the safety of a move was to check each point a certain distance along the line drawn from the move. The distance between each point was.
$\lceil 0.0003/\text{closest distance in no fly zone} \rceil$ to give a bound for how many points to check. Clearly however the case of clipping a corner could still happen as we cannot check every point along the line. In the end it was decided that a move was legal if the line drawn from the move intersected any of the lines of the outer perimeter of a restricted area. This was done using the Java.awt.geom package for 2D lines which was possible to use as we assume we are working on a 2D grid. This however may only work under the assumption that we are not working with restricted areas that have holes them, although more testing would need to be done to confirm this.

References

[1] P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968, doi: 10.1109/TSSC.1968.300136.

[2] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269–271.

[3] Wikipedia contributors. (2020, June 3). Taxicab geometry. In Wikipedia, The Free Encyclopedia. Retrieved 17:59, December 1, 2020, from en.wikipedia.org/w/index.php?title=Taxicab_geometryoldid=960454083