PIZZA DRONE DELIVERY SYSTEM

REPORT

Author – B179983

INTRODUCTION

The idea behind the project is to program a drone to deliver pizzas to students at Appleton tower from a series of restaurants. The drone fetches all navigation and delivery related data from a REST server. The drone has a restricted battery capacity of 2000 moves for a day and the aim is to maximize the amount of orders you can deliver in a day. The drone needs to calculate the best possible path to maximise the orders. I employ a greedy algorithm (discussed later) to ensure that.

SOFTWARE ARCHITECTURE DESCRIPTION

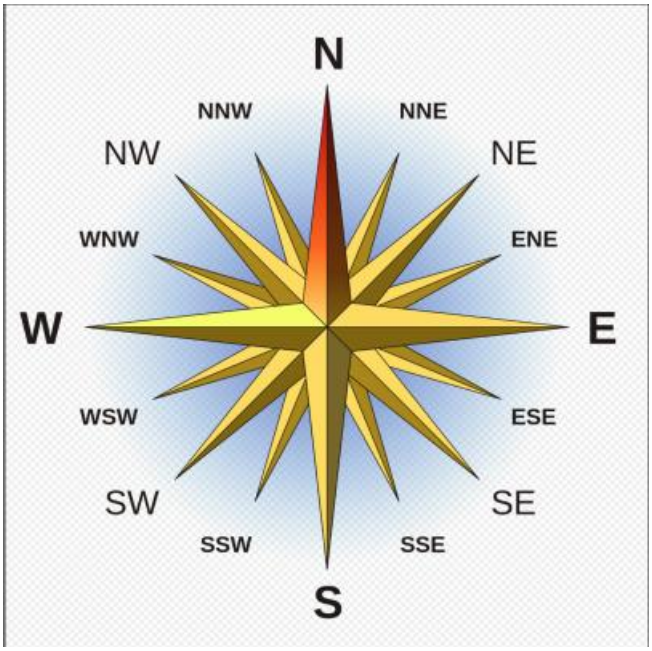
This section explains the reasoning behind each class.

1. **FetchResponse**
   The fetch response class is responsible for handling the networking and communication with the REST server. It accesses all the different content from the web server. This class plays a critical role in making all the data fetching unified. All methods in this class are kept static as an instance of this class isn't created. Only the methods are accessed inside the data storage class.
   Methods: All the following methods fetch data as a list of the respective object classes which are discussed later
   1.1 **getRestaurants** – fetches restaurant data as List of Restaurant Objects
   1.2 **getOrders** - fetches orders data as List of Order Objects
   1.3 **getNoFlyZones** - fetches No fly zones data as List of NoFlyZone Objects
   1.4 **getCentralArea** - fetches the central area coordinates as a List of LngLat Objects

2. **WorldDataStorage**
   This class is responsible for storing all the data required for the functioning of the drone under one roof. Presence of this class prevents recurrent calls to the REST server for data which can lead to a response loop. Every time this class is instantiated, it initializes by using the FetchResponse methods to get the data and then stores the data in private fields inside the class.
   Methods: The class has getters for all the data that is fetched. Apart from the getters, the following methods are implemented for convenience in other classes.
   2.1 **getOrdersByDate**
      Parameter – Date with the format YYYY-MM-DD as a string
      Function – the method filters out the orders on the given date from the list of all the orders in the webserver.
   2.2 getValidOrders
      Parameter – Date with the format YYYY-MM-DD as a string
      Functionality – This method filters out all the valid orders (validity criteria is described below) on the given date from the list of all orders fetched.

3. **Compass – ENUM CLASS**
   This ENUM class holds all the directions that the drone can move in. As per the coursework specification, the drone can only move in the 16 major compass directions (As shown in figure

below). Each direction is associated with an angle. This angle is used to compute the next position that the drone will reach if it takes the associated direction.



4. **OrderOutcome – ENUM CLASS**
   This ENUM class hold all the possible outcomes that an order can have based on the different validity criteria. This ENUM allows us to maintain information about whether an order is valid , delivered, valid but undelivered, or invalid due to any reason like wrong card number, date, cvv, wrong pizza combinations, wrong total cost etc.

5. **LngLat – Record Class**
   This class is used to represent a point (or coordinate) on the map. This coordinates is used to keep track of the drone's position and the location of any landmarks and zones like the central area, no-fly zones, Appleton tower, and the restaurants. This class is made as a record class since the data in the class is immutable. Making it a record class removes the need to create getters and setter for all the fields. It also removes the need to modify methods if new fields are created. A LngLat object is instantiated by passing the longitude and latitude as double precision values for the parameters. This class also deserializes coordinates provided in the REST server.
   Fields – The class has a final field for the distance tolerance of the drone which is 0.00015.
   Methods: The following methods are implemented
   5.1 **distanceTo** – This method takes a LngLat object as parameter and calculates the Euclidean distance between the current LngLat object and the parameter object.
   5.2 **closeTo** – This method takes a LngLat object as parameter and returns a Boolean representing whether the current node is within the distance tolerance of the parameter node.
   5.3 **nextPosition** – This method takes a Compass direction enum as a parameter and calculates the next position that the drone will move to from the current LngLat in the given direction.

6. **Menu**
   Objects of this class represent one menu item in a restaurant. This class also deserializes the menu attribute associated with each restaurant being fetched from the 'restaurants' endpoint of the REST server. Each object of the menu class represents one menu item of the restaurant.

7. **Order**

This class deserializes and performs operations on orders fetched from the 'orders' endpoint of the REST server.

Fields – Apart from all the fields deserialized from the webserver, the class also has field for the outcome of the order as processed by the validity checks and delivery procedures of the drone. Since orders don't have the restaurant of the order as an attribute in the webserver, there is also a field for the restaurant for which the order belongs to. This field is set after the validity of the combination of pizzas is verified.

Methods :

7.1 **invalidCombination –** This method checks if the combination of pizzas associated with the order exists for a restaurant i.e., all the pizzas in the order must belong to a single restaurant.
Parameters - list of all restaurants and a list of the pizzas associated with the order
Functionality – The method loops through all restaurants and makes a method call to the **checkCombination** helper function for each. If the combination exists for a restaurant, it sets the order's restaurant field to that restaurant and returns True. Otherwise, it returns false.

7.2 **checkCombination –** this is the helper method for the invalidCombination method.
Parameters – A restaurant object and list of all pizzas associated with the order
Functionality – It fetches a map of all the menu items and their prices of the restaurant(the map is described in the restaurant class). It then checks if the keyset of the menu map (the keyset is a set of all the menu item names) contains all the pizzas associated with the order.

7.3 **getOrderedPizzasCost –** this method calculates the total cost of the ordered pizzas excluding the delivery cost. It fetches the cost from the menu map associated with the restaurant that the order belongs to.

7.4 **checkCreditCardNumber –** This method validates the card number associated with the order. As per the specifications, the card number has to have 16 numbers and no blank spaces. We then employ the Luhn's Algorithm to verify the number.

7.5 **checkCardDate –** this method validates the expiry date of the card. It first matches the input with a regex. We then parse the date as a LocalDate object and verifies that the card expires after the order date.

7.6 **checkCVV –** this method checks the validity of the CVV(should be of length 3 and not contain blank spaces)

7.7 **pizzaValidity –** this method checks whether the pizzas in the order even exist for any restaurant. We create a list of all the pizzas in all the restaurants and verify if that list contains all of the ordered pizzas.

7.8 **validTotalPrice –** this method verifies if the price associated with the order is equal to the calculated cost of the pizzas along with the 1-pound delivery fee.

7.9 **isOrderValid –** this method utilizes the above functions to set the outcome of the order and return Boolean value for the order validity.

8. **Restaurant**

This class is used to deserialize the restaurants associated with the drone delivery system which are fetched from the REST server. The class also has a method for making a HashMap of all menu items associated the restaurant and their price. This is done to make iterating through the items easier.

9. **NoFlyZone**
   This class is used to deserialize the NoFlyZone data that is fetched from the REST server. Since we are storing coordinates as LngLat objects, we need to convert the double precision coordinated fetched from the server to LngLat objects. The getCoordinates method is created to do that.

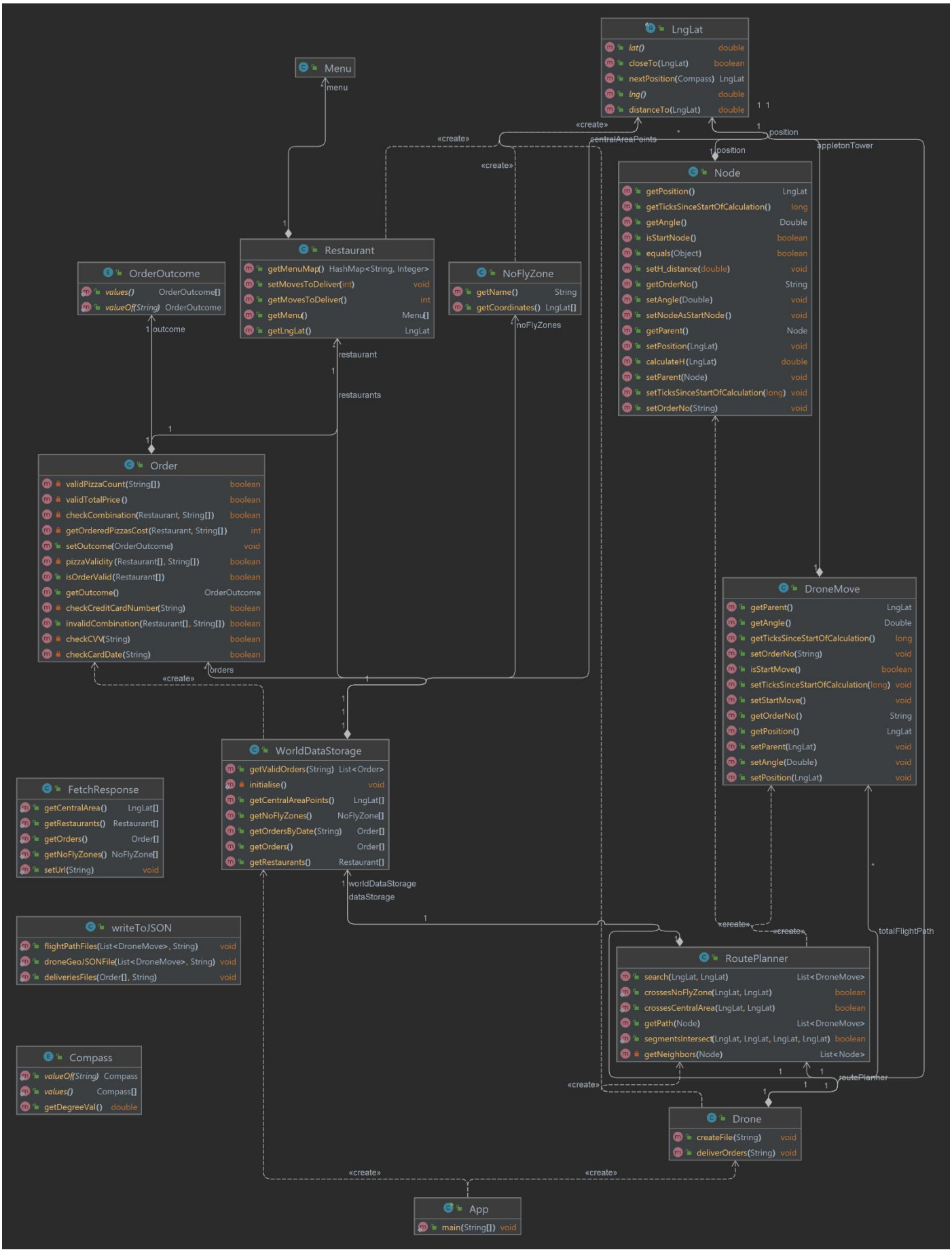
10. **Node**
    This class is used to store information about the nodes traversed in the path finding algorithm. The node represents a move of the drone. This class has fields for storing the position associated with the node as a LngLat object. It also stores the parent node, the angle that the drone would move from the parent node to the current node, the heuristic value, and the ticks elapsed since the start of the pathfinding algorithm. We also override the equals method in order to use the heuristic value in a priority queue for path finding.
11. **RoutePlanner**
    This class is used to implement the greedy path finding algorithm. This class has methods to support the algorithm. Implementation and working of this class is discussed in the next section.

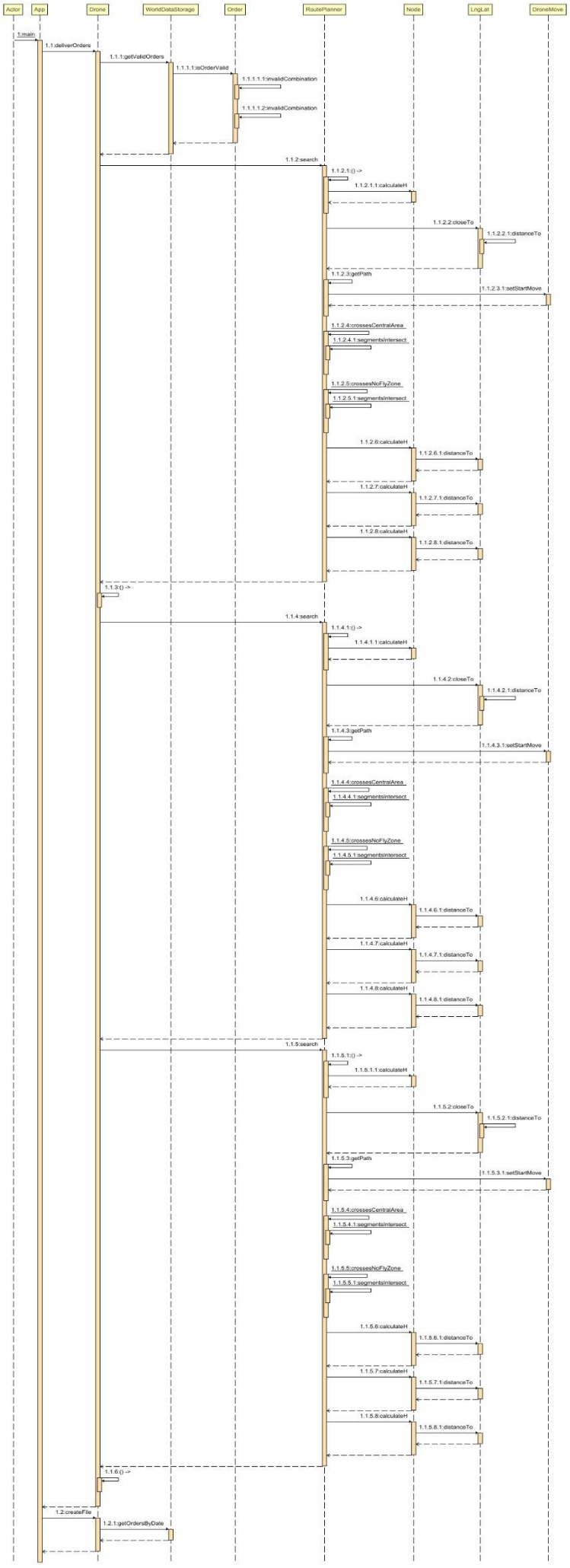
12. **Drone**
    This class is used to represent the actual movement and deliveries made by the drone. This class has the deliverOrders method which utilizes the RoutePlanner output to make deliveries. It also keeps track of the number of steps the drone has made. The method initially sorts all the orders in a day by the number of moves required to deliver the order. Since we want to maximise the number of orders delivered in a day, the orders which require less time to deliver are prioritized. The method also returns the total flight path of the drone in a day.

13. **DroneMove**
    This class is used to represent the moves made by the drone. A move is associated with a node traversed in the path finding algorithm. A DroneMove object has field for the order number of the order that was delivered in that path, the position coordinates of the move as a LngLat object, the parent coordinates as a LngLat object, the angle that the drone would move when travelling from the parent to the current position, the ticks since start of the calculation, and a Boolean representing whether the move is a starting move(in which case the parent will be null).

14. **writeToJSON**
    This class is used to create the JSON and GeoJSON files of the deliveries and flightpath of the drone.

UML DIAGRAM


LngLat
lat() double
closeTo(LngLat) boolean
nextPosition(Compass) LngLat
lng() double
distanceTo(LngLat) double
Menu
menu
«create»
centralAreaPoints
position
appletonTower
Node
getPosition() LngLat
getTicksSinceStartOfCalculation() long
getAngle() Double
isStartNode() boolean
equals(Object) boolean
setH_distance(double) void
getOrderNo() String
setAngle(Double) void
setNodeAsStartNode() void
getParent() Node
setPosition(LngLat) void
calculateH(LngLat) double
setParent(Node) void
setTicksSinceStartOfCalculation(long) void
setOrderNo(String) void
Restaurant
getMenuMap() HashMap<String, Integer>
setMovesToDeliver(int) void
getMovesToDeliver() int
getMenu() Menu[]
getLngLat() LngLat
OrderOutcome
values() OrderOutcome[]
valueOf(String) OrderOutcome
NoFlyZone
getName() String
getCoordinates() LngLat[]
outcome
restaurant
restaurants
noFlyZones
Order
validPizzaCount(String[]) boolean
validTotalPrice() boolean
checkCombination(Restaurant, String[]) boolean
getOrderedPizzasCost(Restaurant, String[]) int
setOutcome(OrderOutcome) void
pizzaValidity(Restaurant[], String[]) boolean
isOrderValid(Restaurant[]) boolean
getOutcome() OrderOutcome
checkCreditCardNumber(String) boolean
invalidCombination(Restaurant[], String[]) boolean
checkCVV(String) boolean
checkCardDate(String) boolean
orders
DroneMove
getParent() LngLat
getAngle() Double
getTicksSinceStartOfCalculation() long
setOrderNo(String) void
isStartMove() boolean
setTicksSinceStartOfCalculation(long) void
setStartMove() void
getOrderNo() String
getPosition() LngLat
setParent(LngLat) void
setAngle(Double) void
setPosition(LngLat) void
WorldDataStorage
getValidOrders(String) List<Order>
initialise() void
getCentralAreaPoints() LngLat[]
getNoFlyZones() NoFlyZone[]
getOrdersByDate(String) Order[]
getOrders() Order[]
getRestaurants() Restaurant[]
worldDataStorage
dataStorage
FetchResponse
getCentralArea() LngLat[]
getRestaurants() Restaurant[]
getOrders() Order[]
getNoFlyZones() NoFlyZone[]
setUrl(String) void
writeToJSON
flightPathFiles(List<DroneMove>, String) void
droneGeoJSONFile(List<DroneMove>, String) void
deliveriesFiles(Order[], String) void
totalFlightPath
RoutePlanner
search(LngLat, LngLat) List<DroneMove>
crossesNoFlyZone(LngLat, LngLat) boolean
crossesCentralArea(LngLat, LngLat) boolean
getPath(Node) List<DroneMove>
segmentsIntersect(LngLat, LngLat, LngLat, LngLat) boolean
getNeighbors(Node) List<Node>
routePlanner
Compass
valueOf(String) Compass
values() Compass[]
getDegreeVal() double
Drone
createFile(String) void
deliverOrders(String) void
App
main(String[]) void



Sequence Diagram

DRONE CONTROL ALGORITHM

The aim of the drone is to deliver as many orders as possible within 2000 moves. Additionally, the drone can only move in one of the 16 major compass directions and each move is of 0.00015 degrees. There are no fly zones that the drone must avoid. The drone should also cross the central area only once when picking up or delivering the order.

The **RoutePlanner** class is responsible for the calculation of the flight path. The **search** method in the class is used to do all the computation. My algorithm is a greedy algorithm. The heuristic I used is the Euclidean distance from the current point to a certain end point. Each point that will be put in the drone's path is represented by objects of the Node class. The node class is used to keep track of parents, the coordinate associated with that point, the heuristic value and the ticks elapsed.

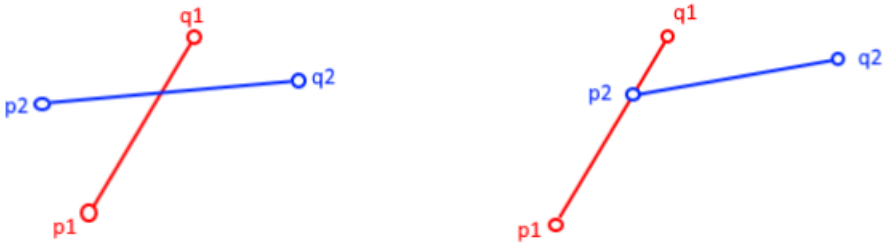
We use the **getNeighbors** method to fetch all possible neighbors for a node. The methods loops through all the 16 compass directions and utilizes the **nextPosition**() method of a LngLat instance, which takes a compass direction as parameter, to obtain the next point that is 0.00015 degrees i.e., one drone move, away in that direction.

Our greedy algorithm favours those nodes which reduce the overall cost of traversing to the end point. Neighbors which take the drone closer to the end point are prioritized and evaluated first.

There are constraints for the validity of the drone path. The drone path will be valid if each individual move is valid. Before adding a move to the path, we check for the validity of the move by making sure that moving from the previous point to the point currently in consideration in a straight line doesn't violate the No-fly zone and central area conditions.

The checks for avoiding no fly zones, not entering or leaving the central more than once when picking up or delivering the order, and returning back to Appleton Tower before the moves run out are as follows:

1. **Avoiding no-fly zones**
   The drone path will be invalid if it crosses any of the no-fly zone. We check this by verifying that a straight line from the current node to the neighbor being evaluated doesn't intersect any of the edges of the no-fly zones. We utilize the Line2D package of Java to check for the intersection of the line joining the nodes and the no-fly zone edges. Every time a neighbor of the current node is fetched, we pass the neighbor and the current node to the **crossesNoFlyZone** method. This method loops through all the no-fly zones and returns True if the line joining the nodes crosses any of the no-fly zone edges.
   If we were to check that the neighbor was IN the no-fly zone, the path would fail in the case where the current point and neighbor point were both outside the no fly zone but the drone's straight line move would have crossed the no fly zone.
   The Line2D package checks for intersections in the case of the two line segments crossing each other as well as that of the endpoint of one line segment lying on the other line segment as shown in the figure below.

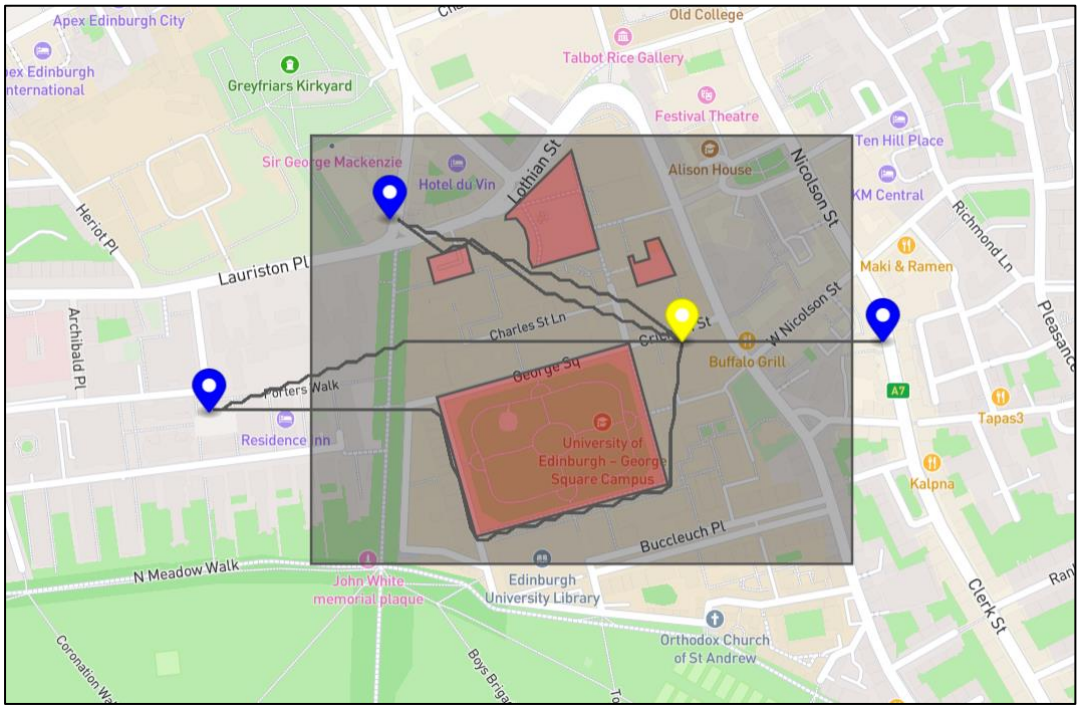


The algorithm will ignore any neighbors which would cross the no-fly zones and move onto the evaluation of the next.

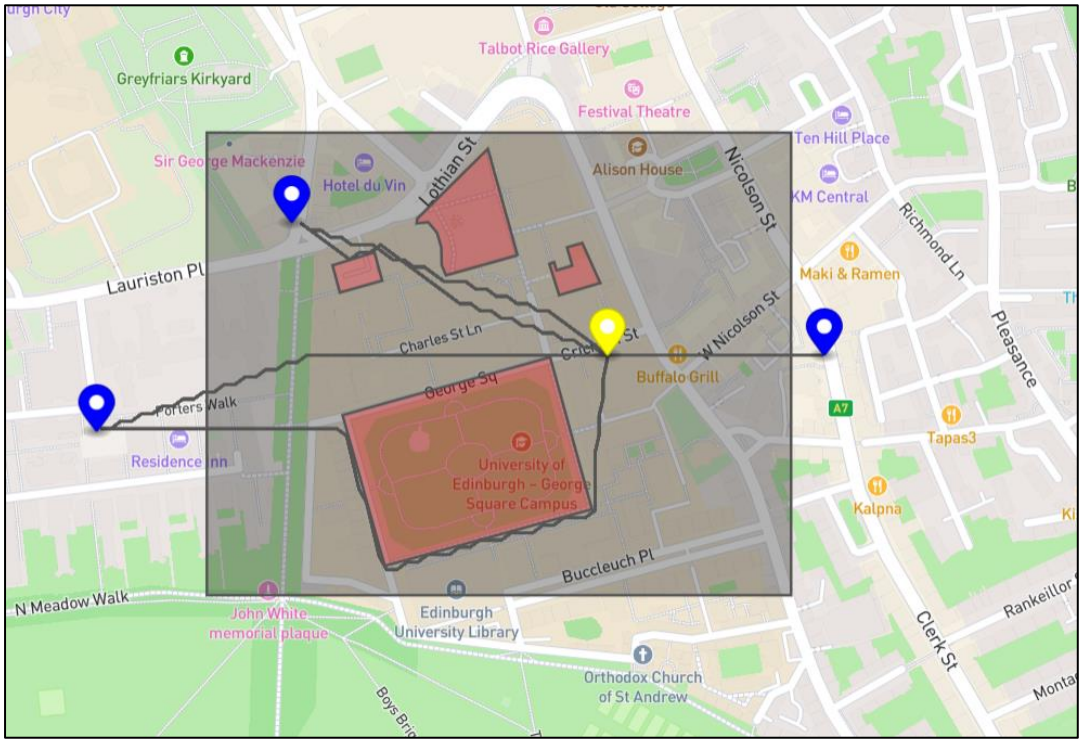
2. **Crossing the central area only once when picking up or delivering orders**
   Similar to the check for no-fly zones, the algorithm will check if the drone's straight line move would intersect any of the central area edges. The primary **search** method has a Boolean field which holds the status of whether the central area has been crossed already. The Boolean is set to true the first time a line joining two subsequent nodes crosses the central area boundary. If any subsequent nodes cross the boundary, they are ignored.

3. **Returning to Appleton Tower before moves finish**
   Orders are processed one after the other in the Drone class. Since the total path for the order is an aggregate of the path from Appleton tower to the restaurant and the path from the restaurant to Appleton tower, the drone's final position after each delivery is automatically close to(in a 0.00015 degree radius) of Appleton. After an order is delivered, the drone will only deliver the next order if the number of moves required to deliver that order are less than or equal to the drone's remaining move. This functionality implicitly ensures that the final position of the Drone is Appleton Tower.

DRONE FLIGHTPATH IMAGES



Flight Path for 2023-01-01



Flight Path for 2023-04-15