ILP Coursework 2 Report

Angus Stewart (s1902147)

December 2020

Contents

1 Software Architecture

When reading the coursework specification, it was clear that there were two distinct components to this practical. The first being the **web server** and the second being the **drone**. For this reason, I decided to create two sub-packages within my `aqmaps` package to contain all the classes related to each component. My `App.java` puts the pieces together to form the final application (e.g. by dealing with the command-line inputs and calling the appropriate public methods of objects of the classes defined in the sub-packages).

The overall structure is shown in Figure 1. The links between the classes indicate a direct relationship (e.g an object of one class is an attribute of another, or a method of one class utilises an object of another).

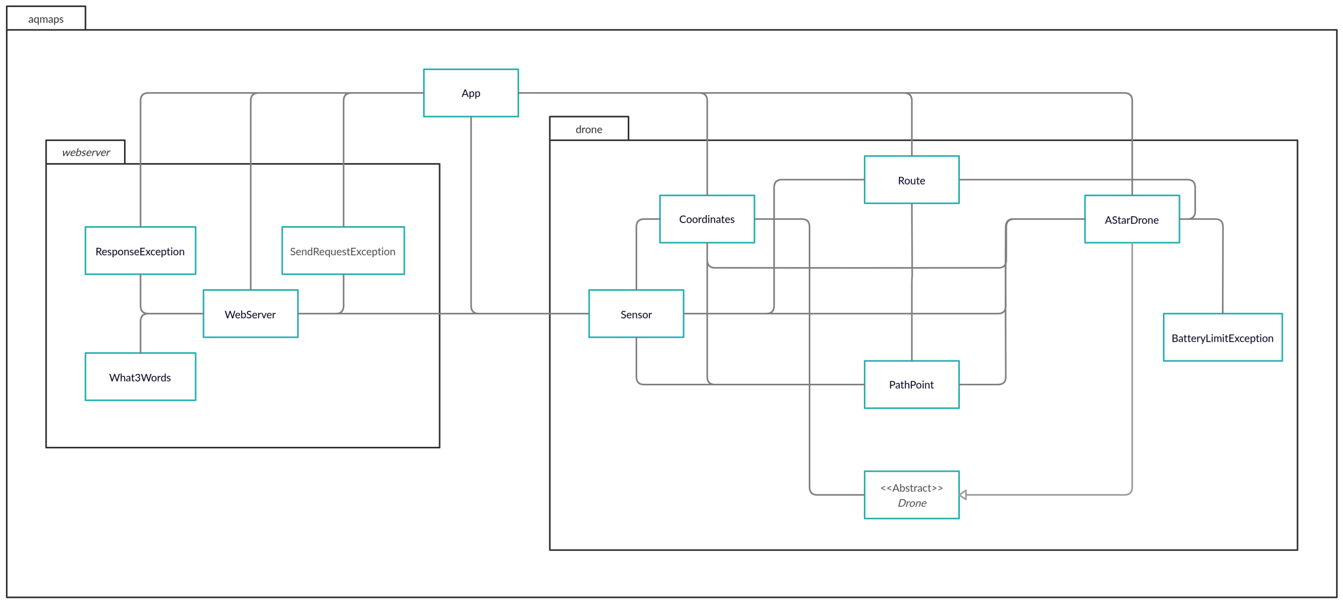

Figure 1: Class diagram of `aqmaps`

1.1 `webserver`

The web server sub-package deals exclusively with **obtaining information from the web server**. Because this is a very important part of the practical, it felt appropriate to dedicate an entire sub-package to this.

The sub-package contains 4 classes:

1. `WebServer`
2. `What3Words`
3. `SendRequestException`
4. `ResponseException`

The `WebServer` is the main class in this package and provides a layer of abstraction over all communication to the web server. It made sense to create a web server class because the web server is an standalone entity within our system. The `WebServer` class models this and provides some useful methods for retrieving data from the server. The other three classes support the main `WebServer` class. The `What3Words` class is used to store the JSON information obtained from the `words/` directory on the web server as a Java object. This is required by `gson`. We also use the `Sensor` object from the `drone` sub-package when obtaining sensor information from the web server. The reason that class was included in the `drone` sub-package and not this package was because it did not make much sense to store a class that represents a physical sensor in a package dedicated to dealing with a web server. It made more sense to include it in the drone package where the drone is modelled due to the close relationship that these two objects have in this practical.

The other two classes (`SendRequestException` and `ResponseException`) were created to abstract over the specific exceptions that can occur when communicating with the web server. The `SendRequestException` abstracts over the exceptions related to the creation and sending of the HTTP request (e.g. if an invalid URI was provided). The `ResponseException` abstracts over the exceptions relating to the HTTP reponse from the web server (e.g. if a 404 error was returned). This provides a cleaner interface for the user of the web server that hides away unnecessary complexity.

1.2 `drone`

This sub-package consists of a set of classes related to the drone's **internal representation of the world**. In other words, these classes together model the problem detailed in the specification of determining and presenting the flight path of a drone around a geographical area with no-fly-zones to collect readings from a set of sensors. This may seem like a lot of disparate ideas, but they are in fact very closely related.

There are 7 classes in this sub-package:

1. `Drone`
2. `AStarDrone`
3. `Sensor`
4. `Route`
5. `PathPoint`
6. `Coordinates`
7. `BatteryLimitException`

The `Drone` class models the **physical properties that affect the drone**. For example, it defines the method `move` which models how to drone moves in the physical world. `getDistance` represents the measure of distance between two points in the real world. It also has attributes that model the confinement area that constrain the drone as detailed in the specification and provides methods for modelling the no-fly-zones. All together this class models the relationship between the drone and the physical world as detailed in the specification. For this reason, the `Drone` class is abstract as instantiating it on its own is undefined. Instead this class is inherited by all drones to allow it to obey the physical contraints posed by the specification.

This brings us to the `AStarDrone` class which models the **behaviour of the drone**. It inherits from the `Drone` class to understand the physical world, then it uses its own methods to navigated within that world. This is where the drone control algorithm (detailed in section 2) is implemented. The class retrieves the initial starting conditions of the world (i.e starting position, sensors, and no-fly-zones) and calculates an appropriate route to visit all the sensors and return to the starting position.

In my original implementation, I only had one `GreedyDrone` class which contained the implementation of both `Drone` and `AStarDrone`. I then decided to create a new drone control algorithm based on the A* search algorithm as the greedy approach I was using was giving me some problems. It was when I was doing this that I noticed that some functions and attributes were being duplicated in both the `GreedyDrone` class and the then newly created `AStarDrone` class. This was the original motivation behind creating an abstract class that 'pulled out' this shared implementation. Upon closer inspection I realized that the code I was pulling out reflected the rules of the physical world in which the drone was operating under as described in specification (e.g distance of a single move of the drone, the distance measure, the representation of the confinement area, etc). This was a pleasant surprise.

The `Route` object is the third main class within this sub-package. This class arose due to a strong emphasis that the specification placed on the route of the drone as a GeoJSON map and as a flightpath in txt format. I felt that modelling the route itself was appropriate as the GeoJSON map and the flightpath were simply different representations of the same underlying 'object' i.e the route. This class provides convenient methods to obtain both the flightpath and the GeoJSON map of the route as `.txt` and `.geojson` files respectively.

The two classes `Coordinates` and `PathPoint` both represent important concepts that map to the real world. The `Coordinates` class represents the latitude and longitude values of a position on the map as described in the specification. This abstraction is incredibly useful for representing the position of objects such as the drone or the sensors. The `PathPoint` object represents a single movement of the drone from one position to another. This class was created to represent the concept outlined in the specification for each line of the flightpath text file. It is also used within the drone to represent the global route as a sequence of drone movements and the drone control algorithm as a way to represent the different possible movement options that the drone has at each particular step.

Finally, the `BatteryLimitException` class is used to represent the event when a route calculated by the drone control algorithm is longer than the battery of the drone allows. By raising this exception the control algorithm can react and create a contingency plan (e.g by returning to the starting position). This class is not essential but proved to be convenient when implementing the control algorithm.

2 Drone Control Algorithm

The specification document describes a modified version of the well known travelling salesman problem (TSP) where we look to visit each sensor and return to the starting position. Of course, it is not exactly the same problem as the path to each sensor is not a single link but rather a series of restricted movements. Hence, the combination of the travelling salesman problem being NP-complete and the movements of the drone made the complexity of the problem very apparent from the beginning.

Knowing that finding an algorithm that would solve the entire path planning problem optimally would not be feasible, I broke the problem into two sub-problems. The first is the order in which the drone visits the sensors and the second is finding the sequence of valid moves to get from one coordinate position to another.

With these two components, we can find the entire route:

1. Get the next **unvisited** sensor given the current position of the drone
2. Find the route to the sensor
   (a) If enough battery, move the drone along the route and mark the sensor as visited
   (b) Otherwise, skip to step 4
3. Go to step 1 until all sensors have been visited
4. Find the route back to the start given the current position. If not enough battery to complete the entire route, follow the route until we run out of battery to return as close to the start as possible.

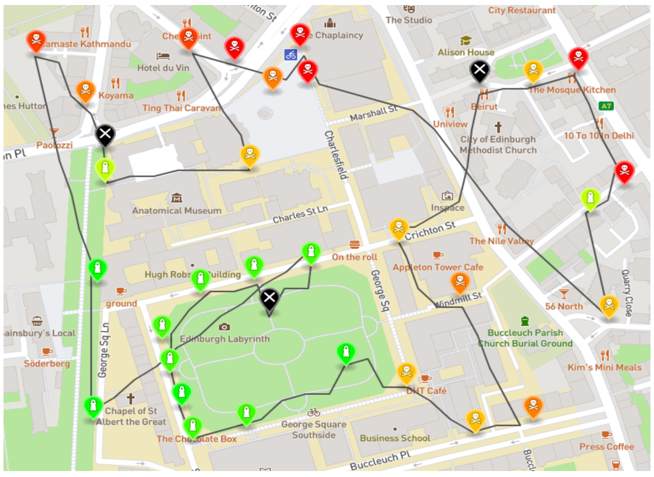
Figure 2: 25th of October 2021

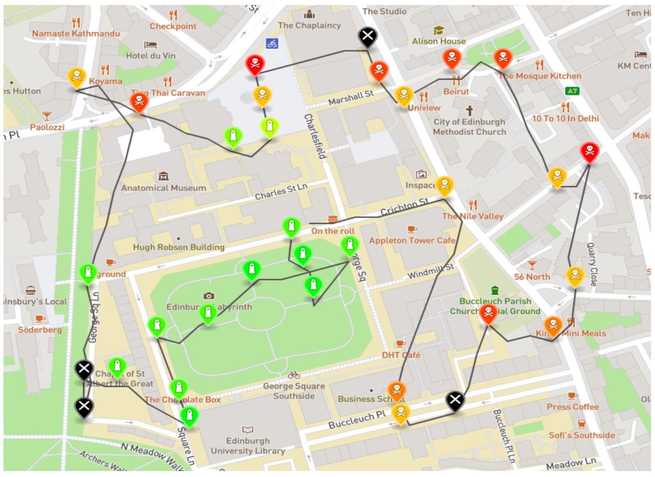
Figure 3: 26th of December 2021

2.1 Ordering of the Sensors

The first problem is determining the order in which the drone should visit the sensors. To keep things simple, I chose a greedy approach, and this is what the drone uses in the implementation. The algorithm calculates the straight-line distance between a given coordinate position (e.g. current position of the drone) and the coordinates of a list of sensors (e.g. list of unvisited sensors) and returns the closest sensor.

Although this algorithm is simple and runs very quickly (linear in the number of sensors), it is unlikely to produce an optimal sequence of sensors to visit. We can see this with Figure 2 where the drone has to take long routes across the map to reach sensors that it missed. In addition, it does not take into account the no-fly-zones of the map and how this affects the distances between sensors. But given the TSP nature of this problem, obtaining good solutions requires significant effort in implementing complex algorithms. Hence, I believe this greedy algorithm is a good starting point due to its balance of simplicity and performance.

It is worth noting that the drone only looks for the next sensor and does not pre-compute the order of all sensors. This decision was taken because the drone does not necessarily reach the exact coordinates of the target sensor and hence pre-computing everything from the beginning would be inaccurate.

2.2 Routing to the Next Position

Given the current position of the drone and the target position (e.g coordinates of the next sensor or of the starting position), we must calculate a valid route between the two points. Note that we do not need to end up **at** the target position, just within a certain distance from it (e.g 0.0002 for sensors).

This can be thought of as a tree search problem. The tree represents all possible sequences of moves (represented by the edges of the tree) that the drone is allowed to make, and the nodes represents coordinate positions with the root being the current position of the drone. Note that the branching factor for this tree is 36 (though it may be less if the movement of the drone is restricted e.g due to being near no-fly-zones).

In my implementation, I use a modified A* search algorithm to find the path down the tree to a node that is within range of our target position. The algorithm consists of two lists: `closedPathPoints`, which stores all the nodes that we have visited (expanded) and `openPathPoints`, which stores all the nodes that we have yet to visit but whose parents have been visited.

1. We begin with only the root node in `openPathPoints`. Remove this node from the list and 'expand' it to generate all **valid** children nodes [1]. These child nodes represent positions the drone can move to in one step that does not violate any of the rules (e.g regarding no-fly-zones).
   (a) Each child node gets assigned a `distanceScore` (among other attributes such as `prev`) which is the sum of the travelled distance from the root to the current node and the straight-line distance from the current node to the target position.
   (b) We then select 5 of the child nodes with the smallest `distanceScore`. We do this to reduce the time complexity of the algorithm as using all child nodes would cause the algorithm to run slowly.
2. Add the 5 nodes to `openPathPoints` and add the root to `closedPathPoints`
3. Select the node from `openPathPoints` with the lowest `distanceScore`. If in range of the target position, skip to step 6, otherwise expand it as per step 1.
4. Before adding each of the 5 nodes to `openPathPoints`, we do two checks:
   (a) If their coordinates match with a node already in `openPathPoints`, we look to see if it has a smaller `distanceScore`. If it does, we add the new node and remove the old one. Otherwise, we discard the new node.
   (b) If the node was not added to `openPathPoints` or discarded in part (a), check to see if the coordinates match with a node already in `closedPathPoints`. If it does, discard the node.
5. Add the expanded node (from step 3) to `closedPathPoints` and go to step 3

[1] To check if a child node is valid, we take the line segment formed by the parent node and the child node and see if it intersects (using the `java.awt.geom`) with an edge of the drone confinement area or no-fly-zones. If it does, it is invalid.

6. Reconstruct the route from the final node from step 3 by following `prev` (which points to the node's parent) until we reach the root.

Notice that this algorithm will always produce a route of at least length 1 because it always expands the root node before checking if any of the nodes are within range of the target position. This is required as per the specification. However, because we do not need to move before finishing, when calculating the route back to the start we check before step 1 whether the root is within range of the target position.

This algorithm has the advantage that it can handle no-fly-zones without much modification (we just don't generate the child nodes that cross the no-fly-zones). This contrasts with my original algorithm that used a greedy approach which would get 'stuck' oscillating between two nodes if the target position was on the other side of a no-fly-zone. This is because unlike the A* search algorithm, the greedy algorithm did not take into account the distance from the root node. However, the time complexity of the A* algorithm is not as good (hence step 1.b) so the time the algorithm takes to find routes over larger distances become much more noticeable which puts the scalability of this algorithm into question.

It is also worth mentioning here, that although **the algorithm is the same**, in the actual implementation we use `PathPoint` objects, which represent the movement of the drone and hence would correspond to the edges of the tree, rather than nodes as described above. However, `PathPoint` stores the nodes in `endPos` (and its parent in `startPos`) and attributes such as `distanceScore` correspond to the position of `endPos` and so the implementation details of the algorithm is very similar.

3 Class Documentation

3.1 `uk.ac.ed.inf.aqmaps`

`App`

This class encapsulates the application as described in the specification.

`public static void main(String[] args)`

This method is the main entry point into the application.

It is responsible for pulling in the different components defined in the `drone` and `webserver` sub-packages to provide the expected top-level behaviour of the application (i.e. process the command-line inputs, get information from the web server, calculate the route of the drone and save this information in the two output files).

Parameters:

`args`: command-line arguments as defined in the specification

3.2 `uk.ac.ed.inf.aqmaps.webserver`

`WebServer`

This class encapsulates methods relating to communication with the web server. The class attributes are `client` which stores the `HttpClient` that sends the HTTP get requests (this is static because we want to prevent multiple instances of the client being created), `noFlyZonePath` which stores the file path to the `geojson` file for the no-fly-zones, `protocol` which stores the protocol to use in the URI (e.g. http), `host` which stores the host name of the web server and `port` which stores the port number to access.

`public class WebServer(String protocol, String host, String port)`

Creates and initializes a `WebServer` object that sends requests to the specified web server.

Parameters:

`protocol`: the protocol to use to send requests (e.g `http`)

`host`: the host name of the server to send requests to (e.g `localhost`)

`port`: the port number to connect to (e.g 80)

`public ArrayList<Sensor> getSensorsWithCoordinates(String year, String month, String day)`

This method returns a list of `Sensor` objects for the specified date **with** coordinate information pre-populated. This is a convenience function built on top of `getSensors` and `getWhat3WordsDetails`. This method will send multiple HTTP requests, one for the relevant `air-quality-data.json` file and one for each sensor to get its coordinate position.

Parameters:

`year`: year of the desired date

`month`: month of the desired date

`day`: day of the desired date

Throws:

`SendRequestException`: if another exception occurs while creating or sending the request

`ResponseException`: if the response object is `null` or the response code is not 200

`public ArrayList<Sensor> getSensors(String year, String month, String day)`

This method returns a list of `Sensor` objects **without** coordinate information (i.e the `coordinates` attribute of each object is `null`). Hence, this method only sends a single request to the server to obtain the `air-quality-data.json` file for the specified date.

Parameters:

`year`: year of the desired date

`month`: month of the desired date

`day`: day of the desired date

Throws:

`SendRequestException`: if another exception occurs while creating or sending the request

`ResponseException`: if the response object is `null` or the response code is not 200

`public What3Words getWhat3WordsDetails(String location)`

This method returns a `What3Words` object for the specified What3Words string. This is useful as the resulting object contains the coordinate position of the sensor at the What3Words location. Only one request to the server is made to get the `details.json` file of the given location string.

Parameters:

`location`: the What3Words location (e.g. “`hurt.green.filer`”)

Throws:

`SendRequestException`: if another exception occurs while creating or sending the request

`ResponseException`: if the response object is `null` or the response code is not 200

`public FeatureCollection getNoFlyZones()`

This method returns the no-fly-zone information stored in `buildings/no-fly-zones.geojson` on the web server as a `FeatureCollection`.

Throws:

- `SendRequestException`: if another exception occurs while creating or sending the request
- `ResponseException`: if the response object is `null` or the response code is not 200

`private HttpResponse<String> sendRequest(String filePath)`

This method is a utility function that sends a HTTP get request to the web server asking for the specified file and returns the `HttpResponse<String>` object if the status code is 200, otherwise an exception is thrown.

Parameters:

- `filepath`: the path to the desired resource on the web server

Throws:

- `SendRequestException`: if another exception occurs while creating or sending the request
- `ResponseException`: if the response object is `null` or the response code is not 200

`What3Words`

This class encapsulates all the information for a particular What3Words location string stored on the web server as described in the specification. It has no methods, just used to encapsulate the information.

`SendRequestException`

This class provides a layer of abstraction over all errors that relate to the creation and sending of a HTTP request to the target web server.

`public SendRequestException(String message)`

Creates and initializes a `SendRequestException` with the provided message.

Parameters:

- `message`: the message to be shown when the exception is thrown

`ResponseException`

This class provides a layer of abstraction over all errors that relate to the HTTP response (or lack thereof) from the target web server.

`public ResponseException(String message)`

Creates and initializes a `ResponseException` with the provided message.

Parameters:

- `message`: the message to be shown when the exception is thrown

3.3 `uk.ac.ed.inf.aqmaps.drone`

Drone

This is an **abstract class** which encapsulates methods and attributes related to the physical characteristics that affect the behaviour of the drone. It includes attributes that define the confinement area as a set of boundaries or line segments (using the `java.awt.geom` package) as well as other constants defined in the specification (i.e max number of moves, the maximum range of a sensor reading, maximum allowable distance from start position when returning and distance of each move of the drone). It also stores some non-static attributes such as `battery` which keeps track of the battery level of the drone, `boundaryLines` which stores all the line segments which the drone cannot intersect, and `noFlyZones` which stores the no-fly-zones.

`Drone(FeatureCollection noFlyZones)`

Initializes the set of line segments which the drone is not allowed to intersect while flying.

Parameters:

`noFlyZones`: the set of no-fly-zones the drone is not allowed to cross while flying

`protected boolean isMoveValid(Coordinates start, Coordinates end)`

Checks to see if the move from `start` to `end` enters a no-fly-zone or leaves the confinement area by looking to see if the move intersects any of the boundaries in `boundaryLines`.

Parameters:

`start`: the coordinate position before the move

`end`: the coordinate position after the move

`protected static ArrayList<Line2D> getBoundaryLines(FeatureCollection noFlyZones)`

Returns an array of line segments that describe the boundaries of the no-fly-zones.

Parameters:

`noFlyZones`: the set of no-fly-zones the drone is not allowed to cross while flying

`protected static Coordinates move(Coordinates currentPos, int direction)`

Returns the position of the drone if it were to move in the given direction from the given position.

Parameters:

`currentPos`: current coordinate position of the drone

`direction`: direction the drone is to move

`protected static double getDistance(Coordinates start, Coordinates end)`

Returns the distance, in degrees, between the two specified coordinate positions.

Parameters:

`start`: first coordinate position

`end`: second coordinate position

AStarDrone

This class encapsulates the behaviour of the drone (i.e the drone control algorithm). It extends the `Drone` abstract class to obtain the physical constraints. The class attributes are `notVisited`, which keeps track of the sensors the drone has not yet visited, `route` which stores a list of `PathPoint` objects that gradually form the route of the drone, `startPos` which remembers the starting position of the drone so that it can be returned to and `currentPos` which keeps track of the current position of the drone. There is also the static attribute `LIMIT_BRANCHING_FACTOR` which is explained in Section 2.

`public AStarDrone(ArrayList<Sensor> sensors, Coordinates startPos, FeatureCollection noFlyZones)`

Creates and initializes the `AStarDrone` object, then calculates the route using `calculateRoute()`.

Parameters:

`sensors`: the list of sensors the drone should visit on its route

`startPos`: the starting position of the drone

`noFlyZones`: the no-fly-zones the drone must avoid

`public Route getRoute()`

Returns a `Route` object that encapsulates the route of the drone.

`private void calculateRoute()`

Calculates the route using the drone control algorithm described in Section 2.

`private Sensor getClosestSensor(Coordinates currentPos, ArrayList<Sensor> sensors)`

Returns the closest sensor from the given sensors to the given position using the distance measure defined in `Drone`. If two sensors are equidistant, the sensor closer to the start of the list is returned.

Parameters:

`currentPos`: the current position from which to find the closest sensor

`sensors`: the list sensors from which the closest sensor is found

`private ArrayList<PathPoint> getRouteToSensor(Coordinates currentPos, Sensor sensor)`

Returns the route from the given position to the given sensor. This method always returns a route of length greater than or equal to one. Internally uses the `getRoute` method to calculate the route.

Parameters:

`currentPos`: the starting position from which to calculate the route

`sensor`: the target sensor the route must reach (within a certain tolerance)

Throws:

BatteryLimitException: if the length of the route is more than the drone's battery level.

`private ArrayList<PathPoint> getRouteToStart(Coordinates currentPos)`

Returns the route from the given position to the original starting position of the drone. This may return a route of length zero if the given position is within acceptable range of the original starting position. Internally uses the `getRoute` method to calculate the route.

Parameters:

`currentPos`: the starting position from which to calculate the route

`private ArrayList<PathPoint> getRoute(Coordinates currentPos, Coordinates target, double maxOffset)`

Calculates and returns the route from the given position to within `maxOffset` of the target position. `maxOffset` allows us to utilize the same method to calculate the route to sensors or back to the starting position. It will always return a route of length 1 or more. This method uses the algorithm described in Section 2 to find the route.

Parameters:

`currentPos`: the starting position from which to calculate the route

`target`: the target position the route should aim to

`maxOffset`: the maximum allowable distance from the target position within which the route is considered to have reached the target `private PathPoint getNextPathPoint(ArrayList<PathPoint> nextPathPoints)`

Returns the `PathPoint` object with the lowest `distanceScore`. In other words, it returns the `PathPoint` object that takes us closest to the target destination. The measure of distance is dependent on the drone control algorithm (see Section 2 for details on the current definition).

Parameters:

`nextPathPoints`: the list of `PathPoint` objects to choose from

`private ArrayList<PathPoint> generatePathPoints(PathPoint currentPathPoint, Coordinates target, int limit)`

Creates and returns a list of `PathPoint` objects representing the potential next move of the drone in the route to `target`. `limit` restricts the size of the returned array to help with time complexity.

Parameters:

`currentPathPoint`: the `PathPoint` object representing the latest move of the drone

`target`: the coordinates of the target position

`limit`: places an upper-bound on the number of `PathPoint` objects to return

Route

This class encapsulates the route of the drone and provides convenient methods to transform and save the route as different formats. Its class attributes are `dronePath` which stores the route taken by the drone, `skippedSensors` which stores the sensors not visited by the drone (important as the markers are different for these sensors), `visitedSensors` which stores the sensors visited by the drone, `noFlyZones` which stores the no-fly-zones and `map` which stores the GeoJSON map generated by the `buildMap` method.

`public Route(ArrayList<PathPoint> dronePath, ArrayList<Sensor> skippedSensors, FeatureCollection noFlyZones)`

Creates and initializes the `Route` object. Gets the list of visited sensors from `dronePath` for convenience as it simplifies the code when creating the GeoJSON map.

Parameters:

`dronePath`: the route taken by the drone

`skippedSensors`: the list of sensors not visited by the drone on its route

`noFlyZones`: the list of no-fly-zones

`public void saveMap(String fileName)`

Saves the created GeoJSON map to a file of the given name. Note that `buildMap` must be run first, or a `RuntimeException` is thrown. This is because the map has some customizable features that the user needs to decide on (i.e displaying the no-fly-zones or not).

Parameters:

`fileName`: name of the file to be created

Throws:

`IOException`: if I/O error occurs (thrown by `FileWriter`)

`public void saveRoute(String fileName)`

Saves the route in a text file in the format given by the specification.

Parameters:

`fileName`: name of the file to be created

Throws:

`IOException`: if I/O error occurs (thrown by `FileWriter`) `public void buildMap(boolean showNoFlyZones)`

Creates the GeoJSON map which is a `FeatureCollection` that includes the flightpath as a `Feature` of type `LineString`, the sensors as `Features` of type `Point` and optionally the no-fly-zones as `Features` of type `Polygon`.

Parameters:

`showNoFlyZones`: whether the no-fly-zones should be included in the GeoJSON map

`private ArrayList<Sensor> getVisitedSensorsFromDronePath()`

A utility function that returns the list of visited sensors from `dronePath` (set in the constructor).

`private Feature createPathFeature()`

Creates and returns a `Feature` of type `LineString` from the `dronePath`. The `LineString` is a series of `Point` objects representing the coordinate positions along the route.

`private Feature createSensorMarker(Sensor sensor, boolean visited)`

Creates and returns the given sensor as a `Feature` of type `Point`. This includes adding various `Feature` properties relating to the symbol and color of the `Feature` as per the specification. `sensor` must have all its attributes populated, otherwise the returned `Feature` will not be useful.

Parameters:

`sensor`: the sensor for which to create the `Feature` for

`visited`: indicates whether the sensor was visited or skipped

`private String getHexColor(Sensor sensor, boolean visited)`

Returns the color, in hexadecimal, determined by the sensor information and whether the sensor was visited or not (as per the specification).

Parameters:

`sensor`: the sensor information used to determine the returned color

`visited`: indicates whether the sensor was visited or skipped

`private String getMarkerSymbol(Sensor sensor, boolean visited)`

Returns the marker symbol (as defined by the specification) determined by the sensor information and whether the sensor was visited or not.

Parameters:

`sensor`: the sensor information used to determine the returned marker symbol

`visited`: indicates whether the sensor was visited or skipped

`Sensor`

This class encapsulates all the information relating to a sensor (i.e. What3Words `location`, `battery` value, sensor `reading`, and `coordinates` position of the sensor) into a single object.

PathPoint

This class encapsulates all the information relating to a single move of the drone (i.e position before the move as `startPos`, position after the move as `endPos`, the `direction` of the move and the `sensor` connected after the move). It also contains three other attributes which are used by the drone control algorithm and has no meaning outside this context. `distanceTravelled` shows the distance travelled by the drone from the start of the sub-route to `endPos` (hence includes the movement in this `PathPoint` object). `distanceScore` is the same as described in Section 2. `prev` points to the previous `PathPoint` object in the sub-route.

`public String toString()`

Returns the attributes as a string that matches the format of a line in the flightpath txt file.

Coordinates

This class encapsulates the latitude (`lat`) and longitude (`lng`) of a coordinate position in a single object.

`public String toString()`

Returns a string representation of the object that fits the format of the flightpath txt file.

`public boolean equals(Object obj)`

Returns true if the given object is of type `Coordinates` and has the same latitude and longitude values as determined by `Double.compare()`.

Parameters:
`obj`: object to be compared with

BatteryLimitException

This class represents the event where the route to a sensor generated by the drone control algorithm is longer than what the battery of the drone allows.

`public BatteryLimitException(String message)`

Creates and initializes a `BatteryLimitException` with the provided message.

Parameters:
`message`: a message to be shown when the exception is thrown

4 References

The following sources were used either to provide inspiration for my implementation or to directly aid in it (e.g. through example code or pseudocode):

1. Wikipedia - Motion planning
2. Wikipedia - Any-angle path planning
3. Wikipedia - A* search algorithm
4. Wikipedia - Best-first search

5. Stackoverflow - How to compare two java objects

The following sources were used to help with the report either for the figures or for the formatting:

1. geojson.io
2. Creately
3. Wikipedia - Class diagram
4. Tex StackExchange - LaTeX figures side by side
5. Tex StackExchange - Remove ugly borders around clickable cross-references and hyperlinks