Contents

1 Software Architecture

The solution is divided into four main components:

- The **client** - responsible for interfacing with the server or other methods of data input
- The **pathfinding** module - handling graph search and pathing between two points in any space
- The **simulation** module - handling the route selection and the actual "drone control" with the use of pathfinding
- The **visualisation** module - responsible for generating visualisations of the output from the simulation module

Most of the inter-module design choices explained below, are fueled by the dependency inversion principle and the rest by good general OOP practice. Due to the size of the solution, a lot of the simpler classes are missed out of the diagrams. The descriptions below will revolve mostly around the key decisions in the architecture and their benefits.

1.1 Client module

Figure 1 shows the structure of the client module classes, it contains Data classes which hold the intermediate, validated and standardised data. These intermediate classes detach the implementation from the exact shape input data. Since any number of formats can map to these classes, the solution is not closely related to the input data - this is very desirable.

The ClientService interface is used to actually create the data in the form of Data classes from the input data given. In the case of AQmaps the data is given via http server and so an appropriate implementing class will need to know the base URI/location of the API where sensor/map/w3w data is stored, and have access to the Http protocol.

Any other classes which will "feed" into the rest of the system, but are constructed from input data, will be declared in this module and implement the appropriate external service interfaces.

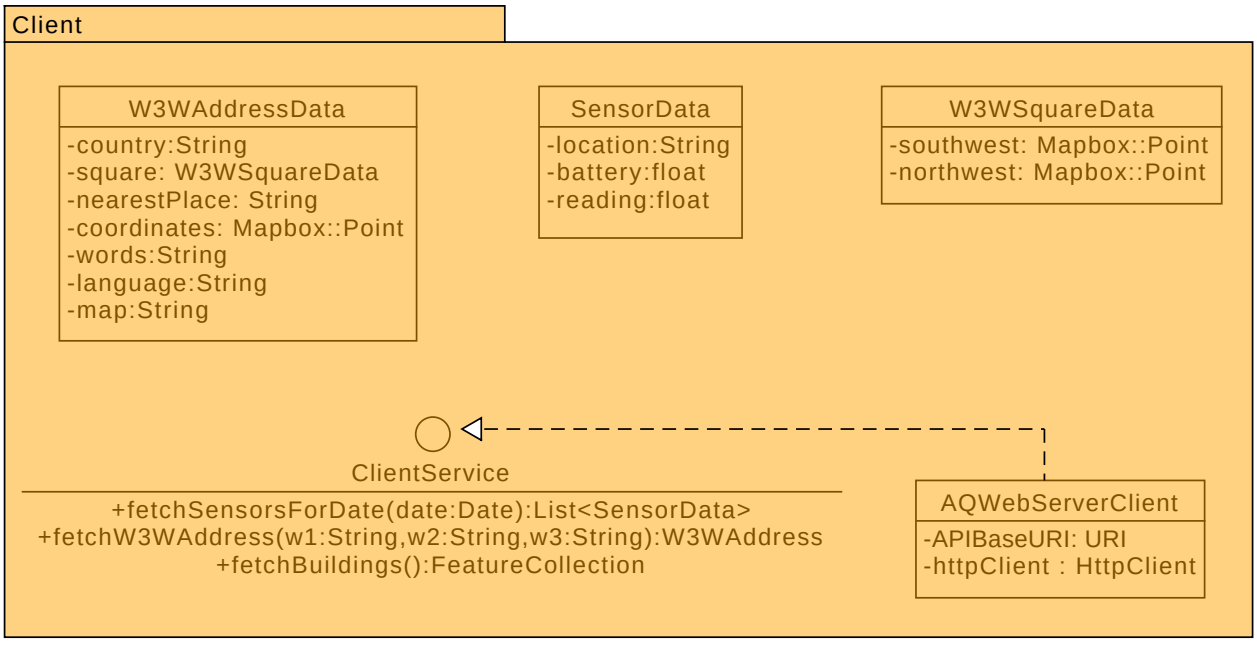

Figure 1: UML diagram of the client module

1.2 Pathfinding module

We separate the problem of pathfinding completely from the AQmap problem. This allows a broader range of techniques to be applied much more easilly.

Figure 3 shows the most important classes from the pathfinding module. The basis of this module is formed by the SearchNode and PathfindingAlgorithm abstract classes as well as the Graph interface.

The PathfindingAlgorithm class defines an abstract findPath method for finding a path to a single goal, leaving the exact way this is done to the concrete implementation. It is marked as abstract since we will always need to

apply pathfinding to multiple goals as well, and so this class should deal with that, since this would otherwise always have to be done by the consumer.

Each SearchNode is used to represent a part of the frontier of a graph search algorithm i.e. a **state** in the search space - in the pathfinding case, a path. These are made abstract to allow for the use of generics in such a way that each parent node is of the specific node type needed and also because nodes are expected to be annotated with problem specific data. It is this generic parameter which propagates to the rest of the classes.

Due to the fact we are mostly dealing with multiple goal pathfinding, each node should also hold a deque of goals which can be reached from it - take as an example the problem where you start at a position with 2 goals, no movement is necessary and both are reached.

Pathfinding goals are represented by their own class, alternatively we could simply use coordinates as goals, but this would force the consumer of this functionality to always have to match the coordinates of the path returned to the goals that it needed achieved, as opposed to simply looking through the goals achieved by each node and comparing them via reference to the deque of goals in order.

Finally the Graph interface defines the transition function from any searchNode to all its neighbours. PathfindingAlgorithms always accept the graph as a parameter as it forms the domain of each specific problem. Separating the domain from the algorithm completely makes the algorithms much more flexible and easier to unit test.

Other interfaces shown include the SpatialHash and PathfindingHeuristic, those will be used by implementing algorithms and are defined as interfaces to again further separate the components of algorithms from the domain. This allows us to further change the behaviour and performance of algorithms, by swapping out their components to suit (Composition over Inheritance).

1.3 Simulation module

The simulation module is responsible for: planning the route (using TSP solvers), applying pathfinding to find the detailed path required for collection, and setting the read status in sensors. The name stems from the fact that our sensor data collection is only simulated. This module directly interfaces with the pathfinding module as expected. Principles of dependency inverersion were applied to reduce coupling and increase flexibility.

Figure 9 shows the most important classes of the simulation module. The biggest architectural choice here is the splitting of the drone into components.
This means that we keep the specific pathfinding and route planning behaviour separate from the behaviour of the drone/data collector itself. Since it is not the job of the path or route planner to "read" any sensors, this will be the task of the drone in addition to perhaps applying some strategy to the produced path in case it is not satysfying enough (maybe re-routing from a point with a different collection order).

Another big choice is the fact that this module does not know anything about the input data, it simply defines a contract for the sensor class, letting the consumers of this module deal with data conversion.

The path planner component itself is further composed of the PathfindingAlgorithm and DistanceMatrix which means it only concerns itself with the task of translating a path of individual points it receives from the PathfindingAlgorithm to a path composed of path segments enforcing the move pattern of the drone.

The PathSegment class is necessary since it is a major requirement that the collector must move in a specific pattern, this class can be used to enforce such a contract.

On the other hand CollectionOrderPlanners are only defined by their implementation and the choice of DistanceMatrix as well as the set of route optimisers. This choice decouples the major TSP solving strategy from the distance measure that it uses.

Both the PathPlanner and CollectionOrderPlanner interfaces have a base abstract class implementation since:

- Collection order planners will always apply the set of given optimisers to the final route.
- They will also have to always set up the distance matrix with the given sensors
- Path planners will always have to apply the given pathfinding algorithm before actually performing conversion to path segments
- We also use these abstract bases to enforce problem specific rules, such as the maximum number of moves, this leaves the possibility of straying away from those base classes if the problem changes drastically in the future.

This module also implements a specific Graph and SearchNode specific to the problem at hand. Here the ConstrainedTreeGraph produces nodes satisfying the angle, move length and obstacle constraints, in effect all logic to do with checking whether the collector is hitting an obstacle, or if the move length is valid etc.. is contained in the graph itself. This is very desirable and intentional. The addition of the direction field to the DirectedSearchNode further emphasizes the angle requirements, and allows for the construction of PathSegments from search nodes without calculating angles twice.

Another important class here is the abstract Sensor class. This is made abstract for two reasons:

- The concrete implementation of the sensor should be done in another module, since it is more convenient for example to closely link it to the input data for convenience. So according to the principle of dependency inversion we do not provide a concrete implementation in this module.

- The sensor cannot show any reading unless it is actually read, this can be enforced by this abstact class using the setHasBeenRead method regardless of the concrete implementation/constructor.

1.4 Visualisation module

This module only needs to interface with the output classes of the simulation module, i.e. the PathSegment and Sensor classes but other than that, it is completely decoupled from the shape of the initial input data of the system. This is very desirable

Figure 2 shows the UML diagram of the most important classes in the visualisation module.
Classes implementing the SensorCollectionVisualiser interface generate geojson visualisations of sensor data collections. These have access to both the flight path and the sensors and hence their readings. The AQMapGenerator is the problem specific example of implementation of such a visualiser.

Using an interface like like this allows for the swapping out of visualisers at will whenever new requirements arise or current ones change.

The OutputFormatter class deals with writing the visualisations and flight paths to a file, these are made static as changes in output format are assumed to be very few in the future, and should such changes be required, new methods can be added to the formatter.

The usage of the AttributeMap interface allows for a lot of flexibility in the way the AQMapGenerator assigns colors and symbols, and should this behaviour need to be changed, it'd be very easy to do.

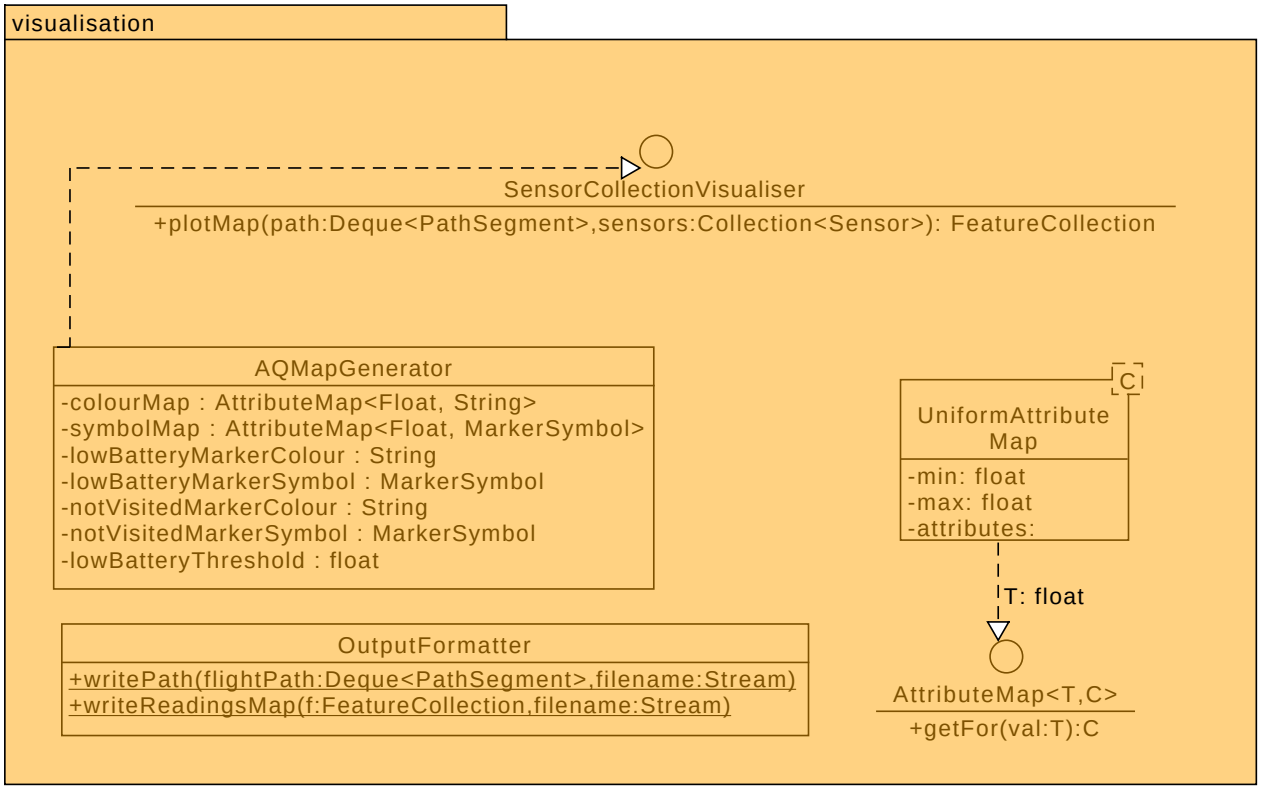

Figure 2: UML diagram of the Visualisation module

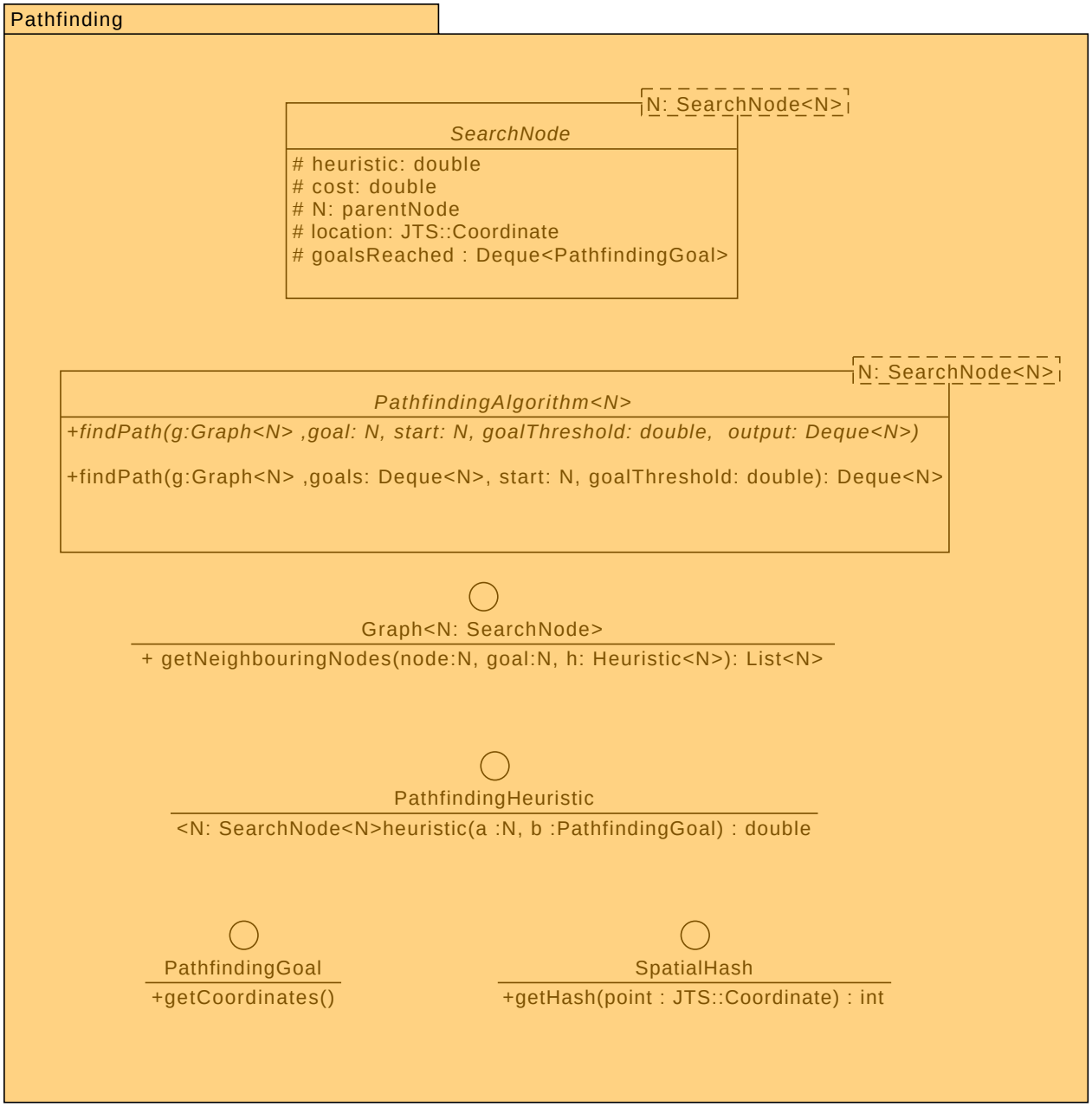

Figure 3: UML diagram of the pathfinding module

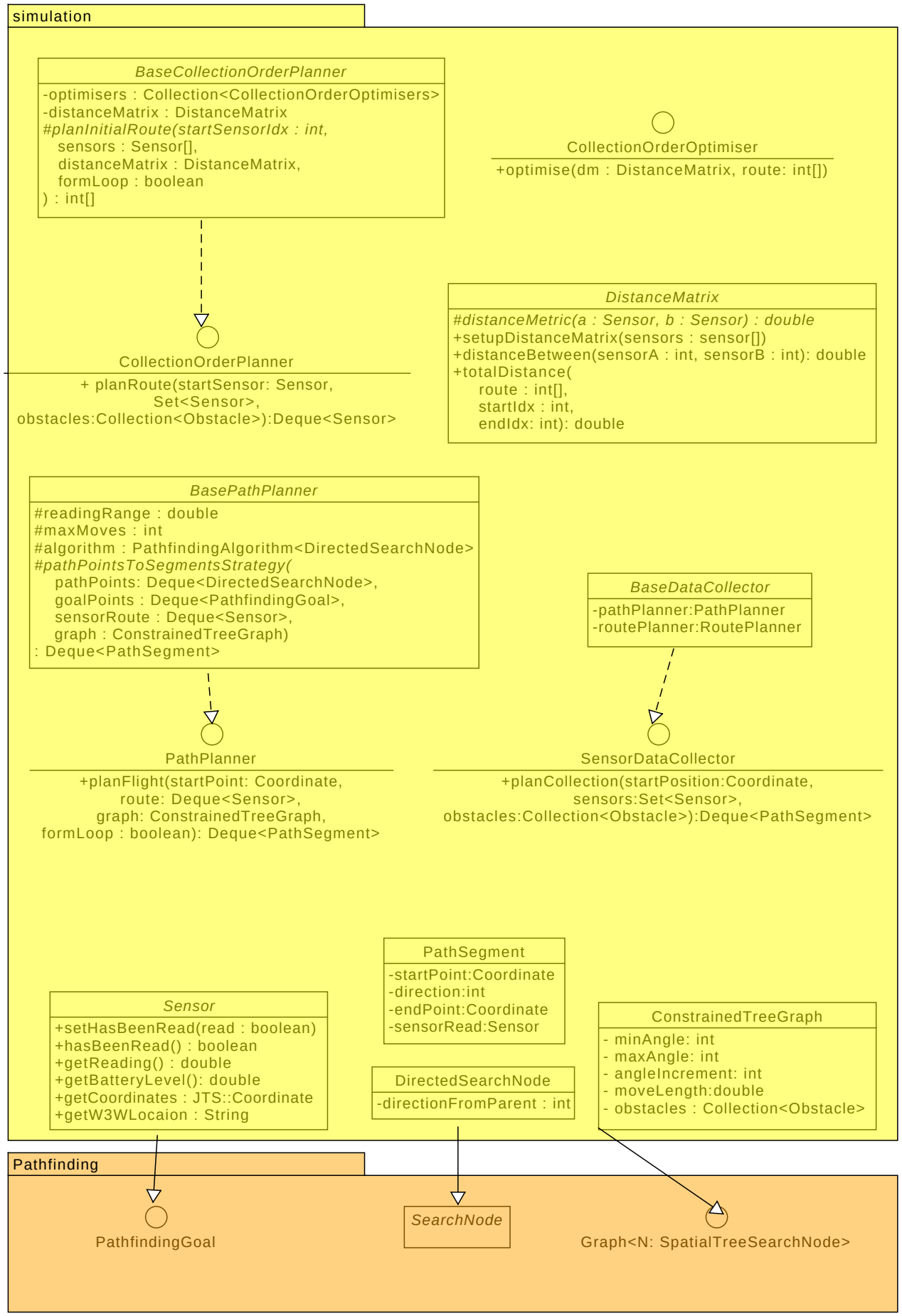

Figure 4: UML diagram of the Simulation module

2 Drone Control Algorithm

2.1 TSP Solver

```
Algorithm NearestInsertion:
    Let R be the sensor nearest to the starting point
    Let U ← unvisited sensors
    Let T ← [R] be the current tour (implicitly looping)

    While U ≠ ∅:
        R ← arg min_{s∈R} distanceToTour(s)
        i ← arg min_{i∈I(T)} insertionCost(i,R)
        T ← T with R inserted at i
        U ← U − {R}

    Return T

Function distanceToTour(s):
    minimum ← ∞
    For t in T:
        If dist(s,t) < minimum:
            minimum ← dist(s,t)
    Return minimum

Function insertionCost(i,s):
    N ← T with s inserted at i
    Return euclidian length of N
```

Figure 5: Nearest Insertion TSP solver

The more general part of the problem is selecting the collection order of the sensors on any given day. Assuming the drone already has the data for the appropriate collection day, it must decide on a visiting order of the sensors.

For this part many techniques were tried, and experimentation proved that applying the Nearest Insertion heuristic together with a euclidian distance matrix was an adequate solution. At each step we chose the sensor which is closest to one of the sensors present in the tour already - the first sensor is the one nearest to the starting point -, and insert it in a way which minimises the cost. If U is the set of unvisited sensors and T is set of sensors in the tour, at each step we pick the sensor s such that:

$$\exists s, \exists k \in T, \forall o \in U. distance(s,k) < distance(o,k) \tag{1}$$

and insert it into the tour between consecutive sensors $i, j \in T$ such that the cost of the tour is minimized.
Following this initial pass, 2-opt optimisations are applied to the route to remove crossings in the graph and further reduce the cost. The 2-opt algorithm in each pass checks if reversing any sub-segment of all possible sub-segments of a full tour reduces it's cost, and if so keeps the reversal. The passes are repetead until improvements fall under a threshold of 0.00003 degrees

This alrogithm turns out to be pareto-optimal [1] among a family of cutting-edge algorithms for problem sizes of between 30-50 vertices. Pareto-optimality means that this algorithm was either finding the best path compared to the other algorithms, or was finding one the quickest. It also happens that this algorithm is just a variation of Prim's algorithm as in this process we find the minimum spanning tree of the implicit graph, therefore Nearest insertion is actually a 2-approximation algorithm [2] of the optimal solution.

2.2 Generating neighbour nodes

All the other sections of the algorithm make use of the "neighbour" function, i.e. the function which generates neighbour nodes for any node representing a point on the map.

This is done using the Bounding Volume Hierarchy data structure. The structure allows for logarithmic time lookups of possibly colliding obstacles (with any shape). We do this by creating a binary tree whose nodes are defined by an Axis Aligned Bounding Volume enveloping all the obstacles present in the leaf nodes underneath the node. The root node then envelops all the obstacles present in the hierarchy.

When creating the tree we find the axis along which the difference between the extremal coordinates of the AABB's is the largest, i.e. the "longest axis". We then pick a splitting point on the axis and partition the shapes into the left and right sub trees according to which side of the splitting point they're on.

With this setup, the structure can tell us which obstacles are possibly colliding with any given shape by checking for collisions (cheaply) with the AABB's and only returning those leafs whose AABB's were collided with (possibly from both subtrees).

2.3 Pathfinding

```
Algorithm A*:
    Let R ← starting node
    let G ← goal coordinates
    Let O ← {R} // open set
    Let V ← {} // approximately visited

    While O ≠ ∅:
        R ← arg min_{o∈O} Fvalue(o)
        O ← O − R

        If isNearGoal(R):
            R.goalsReached ← R.goalsReached + G
            Return reconstructPath(R) // using parent references

        // generate neighbour nodes with appropriate costs and parent set to R
        // nodes colliding with obstacles or outside of boundary are excluded
        // this is done via Bounding Volume Hierarchies
        N ← neighbours(R)

        For n in N:

            hash ← cantorHash(n.x,n.y)

            If hash in V:
                Skip n

            V ← V + hash
            O ← O + n

    // no path found
    Return []

Function Fvalue(n):
    Return dist(n,G)*1.5 + cost(n)

Function cantorHash(x,y):
    Let gridWidth ← 1/75 * 0.0003
    Let gridX,gridY ← coordinates of center of drone confinement area
    nx,ny ← (x − gridX)/gridWidth,(y − gridY)/gridWidth
    nx,ny ← makePositive(⌊nx⌋),makePositive(⌊ny⌋)

    return ⌊(((0.5 * (nx + ny)) * (nx + ny + 1)) + ny)⌋;

Function makePositive(n):
    If n ≥ 0:
        Return 2n
    Otherwise:
        Return −2n − 1
```

Figure 6: Custom A* pathfinding algorithm

Pathfinding between any two points on the plane was carried out using the A* tree search algorithm with the euclidian distance heuristic. The algorithm is just a modified breadth-first search, where the search nodes are

picked in order of least f value which is defined as:

$$f(n) = h(n) + c(n) \tag{2}$$

with $c(n)$ is the cost of reaching the node n from the start state, and $h(n)$ is the approximation of the cost of reaching the goal state from the node n. Assuming that the node in the search frontier with the smallest f value is always expanded and that the heuristic is admissible A* will return the optimal path.

A naive application of A* could not work however due to floating-point error causing the expansion of a lot of poor nodes and the sheer amount of nodes present in the frontier at any time. The straight-line-distance heuristic was relaxed and the new definition was the following:

$$h(n) = distance(n, g) * 1.5 \tag{3}$$

This makes nodes closer to the goal receive much smaller heuristic values and so more direct paths to the goal are expanded first. This means that the returned paths are no longer optimal as the heuristic is no longer admissible, but since most paths would not be obstucted by buildings - this change does not actually impact the length of the paths all that much.

Another change was the addition of a spatial hash to try and prune nodes which were approximately visited, by finding out their coordinates relative to a grid centered around the center of the boundary (with integer width square size), and hashing those coordinates using a modified cantor pairing allowing for negative values

2.4 Path segmenting

Once we have a path of individual points, we need to convert it to a path of path segments with information about which sensor is read at each segment. We can do this by "sliding" a window over the points and always looking at two points at a time in order. We call the points at each position of the windo P and N, where P is the point before N. The pathfinding algorithm also attaches a deque of sensors reached by each P/N. Multiple conversion problems arise when naively pairing up the points to create segments:

- P or N may reach multiple sensors which is invalid, each P-N segment must only allow for one reading
- The list of points might be empty if no path is found
- P might reach a sensor which is illegal, only N's in each segment are allowed to read sensors (since an N point always occurs at the end of a move)

The empty list of points can be prevented by simply returning no path segments, the rest of the problems can be defeated if we apply a number of rules in order (portrayed in Figure 7):

- While P attains a goal, create a proxy segment to any neighbouring node, and one back, assign the first sensor attained by P to the newest back-facing segment's N node
- If N attains no goals, see if the next segment's P node attains any sensors, if so "steal" one away from it
- while N attains more than one sensor reduce the number of attained sensors by creating proxy segments as above for each
- If N only attains one sensor or none, create P-N path segment as normal

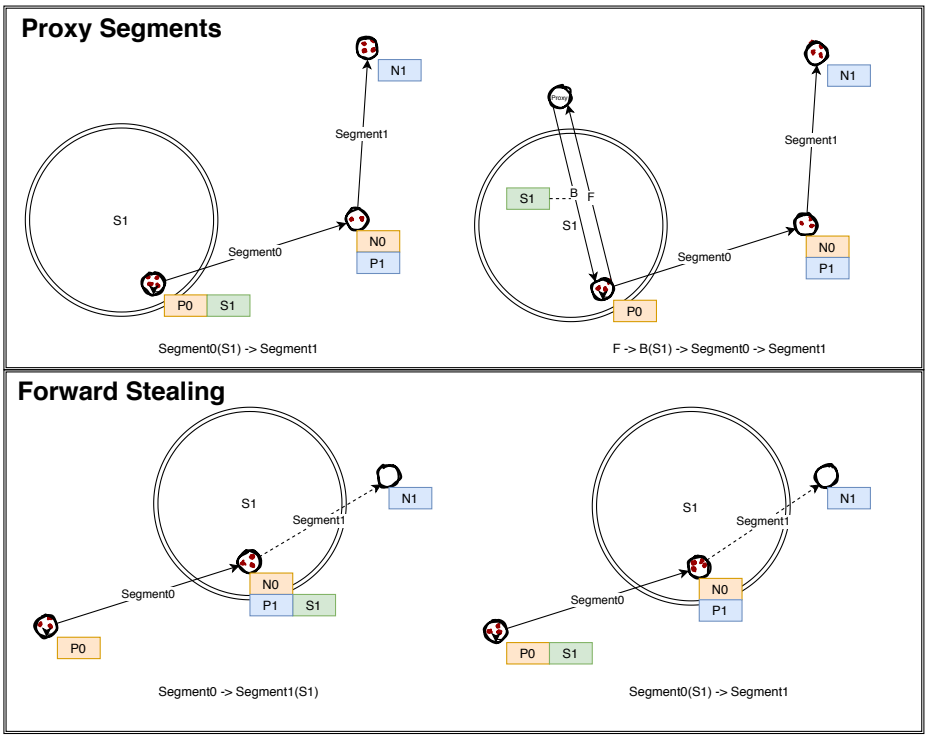

Figure 7: Segmenting behaviours

2.5 Examples

The algorithm was tested on over 35000 configurations over the data provided. The hardest and easiest collection days' geojson visualisations are shown below (with the algorithm set to optimal parameters as given above)

Figure 8: geojson.io rendering of hardest collection at day 9-2-2020 with a starting point of -3.19087,55.945778, with 111 moves

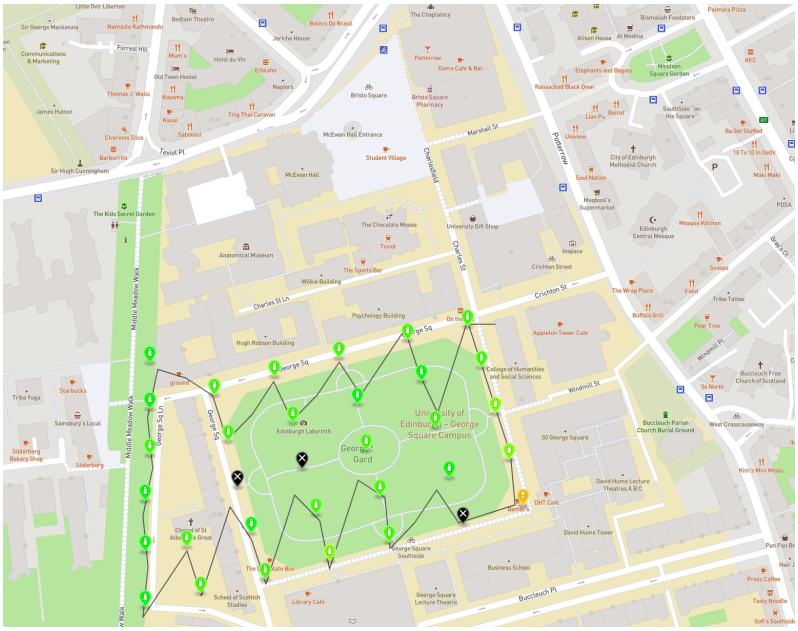

Figure 9: geojson.io rendering of easiest collection at day 2-1-2021 with a starting point of -3.1878,55.9444, with 46 moves

The artifacts of path segmenting are evident in the harder day, where the drone wanders near closely spaced sensors on the bottom right it creates a horizontal proxy segment. But overall the algorithm manages very well, never carrying out the collection in more than 111 moves and averaging at 90 moves ± 6.7. Execution times averaged at 103ms.

3 Class Documentation

3.1 Quicklinks

aqmaps
aqmaps.client
aqmaps.client.data
aqmaps.pathfinding
aqmaps.pathfinding.goals
aqmaps.pathfinding.hashing
aqmaps.pathfinding.hashing.heuristics
aqmaps.simulation
aqmaps.simulation.collection
aqmaps.simulation.planning
aqmaps.simulation.planning.collectionOrder
aqmaps.simulation.planning.collectionOrder.optimisers
aqmaps.simulation.planning.path
aqmaps.utilities
aqmaps.visualisation

3.2 Documentation

3.2.1 aqmaps

App Application which organises the collection of air quality data and visualises it

- static void **main**(String[] args) - launches the program with the given arguments.
  args - an array of string arguments the first 7 must correspond (in order) to:
  - day of collection
  - month of colleciton
  - year of collection
  - starting latitude
  - starting longitude
  - random seed
  - port number of the web server

3.2.2 aqmaps.client

AQSensor An implementation of the Sensor interface for consumption in the simulation module. Conveniently can be formed from sensor and address data corresponding to the data received via API client.

Implements **Sensor**, **PathfindingGoal**

• **AQSensor**(SensorData sensorData, W3WAddressData w3WAddressData)	• Construct new AQ sensor from sensor data representing the sensor's readings and battery information, and with the address data containing the w3w address words and coordinates of the sensor
• String **hasBeenRead**()	• returns true if this sensor has been read
• String **setHaveBeenRead**()	• Sets the sensor "read" state to the given argument
• float **getBatteryLevel**()	• returns the battery level at this sensor provided it has been read
• jts::Coordinate **getCoordinates**()	• returns the long,lat coordinates of the sensor
• float **getReading**()	• returns the reading at this sensor provided it has been read
• String **getW3WLocation**()	• returns the 3 word address (w3w) of this sensor

AQWebServerClient The web server client communicates with a server which contains the air quality information and retrieves/parses/validates it

Implements **ClientService**

• **AQWebServerClient**(HttpClient client, URI APIBaseURI)	• Construct a new web server client served via the given http client and the api uri
• mapbox::FeatureCollection **fetchBuildings**	• Retrieves the feature collection of all the no-fly zones available from the server
• List<SensorData> **fetchSensorsForDate**(LocalDate date)	• Retrieves the data of all sensors to be collected on the given date
• W3WAddressData **fetchW3WAddress**(String wordAddress)	• Retrieves the detailed data on the given what-3-words address

3.2.3 aqmaps.client.data

SensorData Object representing the reading, battery and location data of a sensor

• **SensorData**(String W3WLocation, float battery, float reading)	• Constructs a new sensor data object
• getters for all attributes	• -

W3WAddressData An object containing information about a what-3-word square.

• **W3WAddressData**(String country, W3WSquareData square, String nearestPlace, mapbox::Point coordinates, String words, String language, String map)	• Constructs a new W3WAddress data object
• getters for all attributes	• -

W3WSquareData Object representing information about a what-3-words square's coordinates.

• **W3WSquareData**(mapbox::Point southwest, mapbox::Point northeast)	• Constructs a new W3WSquare data object
• getters for all attributes	• -

3.2.4 aqmaps.pathfinding

3.2.5 aqmaps.pathfinding.goals

3.2.6 aqmaps.pathfinding.hashing

3.2.7 aqmaps.pathfinding.heuristics

3.2.8 aqmaps.simulation

3.2.9 aqmaps.simulation.collection

3.2.10 aqmaps.simulation.planning

3.2.11 aqmaps.simulation.planning.collectionOrder

3.2.12 aqmaps.simulation.planning.collectionOrder.optimisers

3.2.13 aqmaps.simulation.planning.path

3.2.14 aqmaps.utilities

3.2.15 aqmaps.visualisation

References

[1] The Metric Travelling Salesman Problem: The Experiment on Pareto-optimal Algorithms, Sergey Avdoshin, E.N.Beresneva 2017.

[2] https://aswani.ieor.berkeley.edu/teaching/FA13/151/lecture_notes/ieor151_lec17.pdf, The Traveling Salesman Problem Professor Z. Max Shen