

# Contents

<b>1</b>	<b>Software Architecture</b>	<b>1</b>
1.1	Client module . . . . .	1
1.2	Pathfinding module . . . . .	1
1.3	Simulation module . . . . .	2
1.4	Visualisation module . . . . .	3
<b>2</b>	<b>Class Documentation</b>	<b>6</b>
<b>3</b>	<b>Drone Control Algorithm</b>	<b>6</b>

# 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.

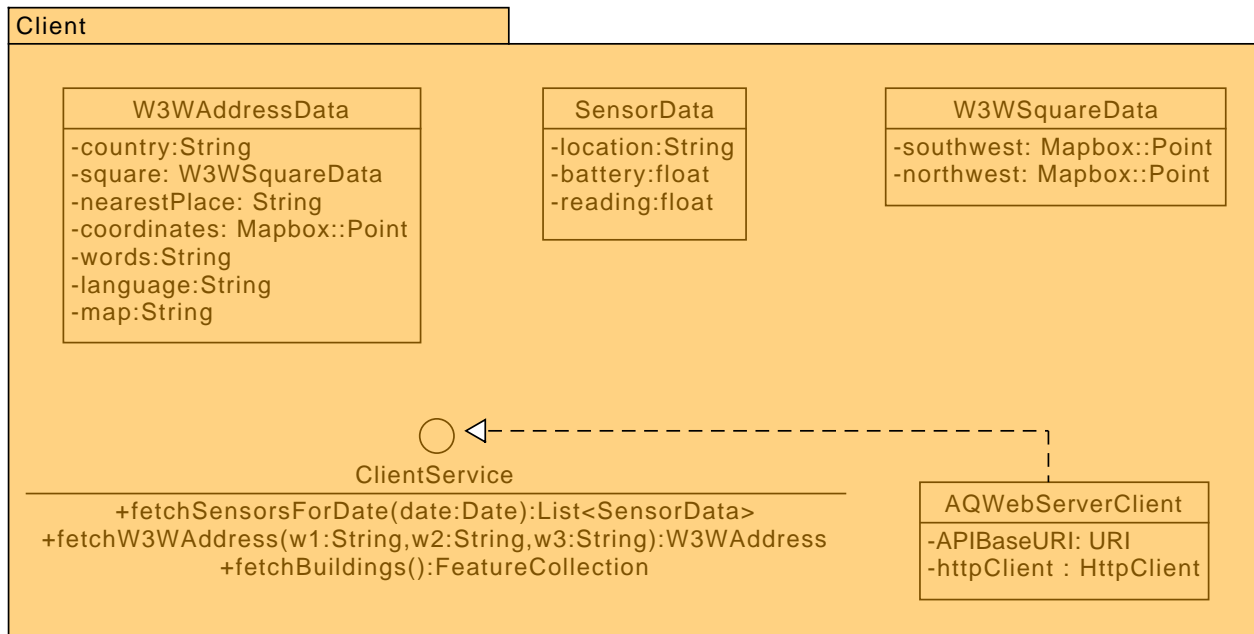


Figure 1: UML diagram of the client module

## 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 easily.

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. Pathfinding Algorithms 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 inversion were applied to reduce coupling and increase flexibility.

Figure 4 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 satisfying 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 abstract 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 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.

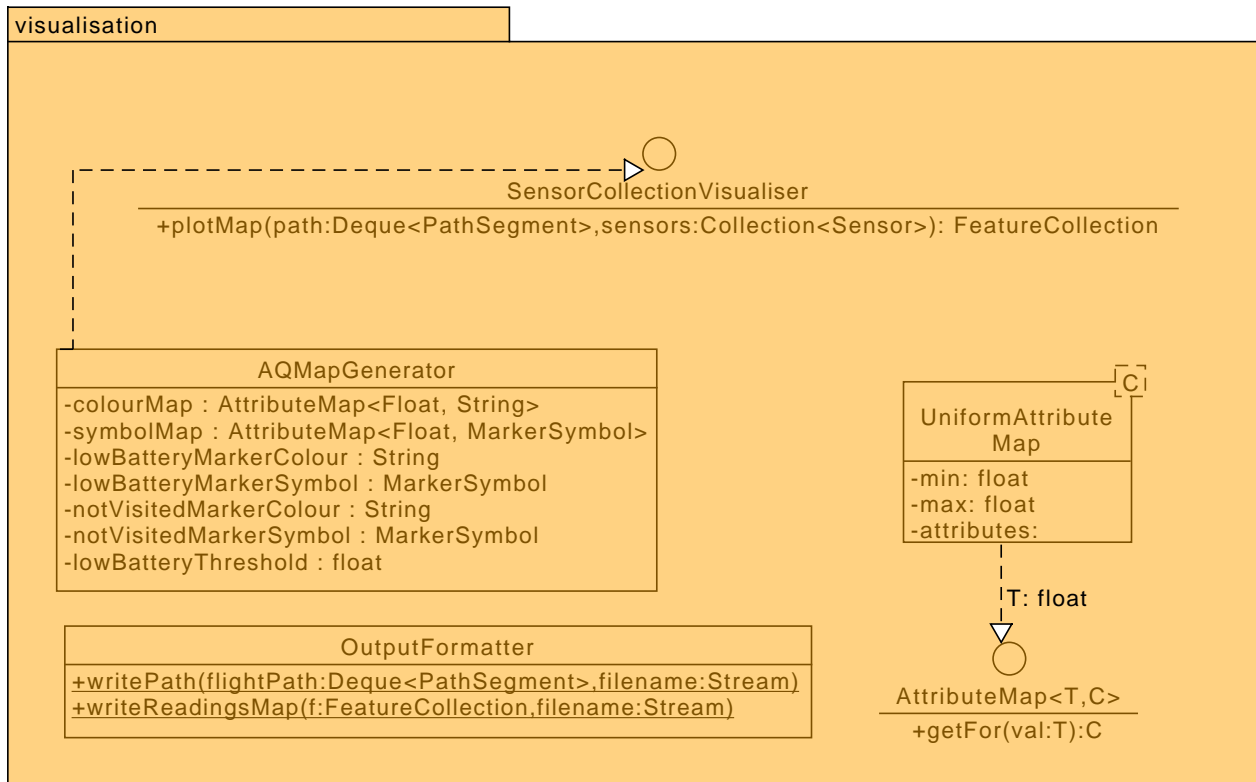


Figure 2: UML diagram of the Visualisation module

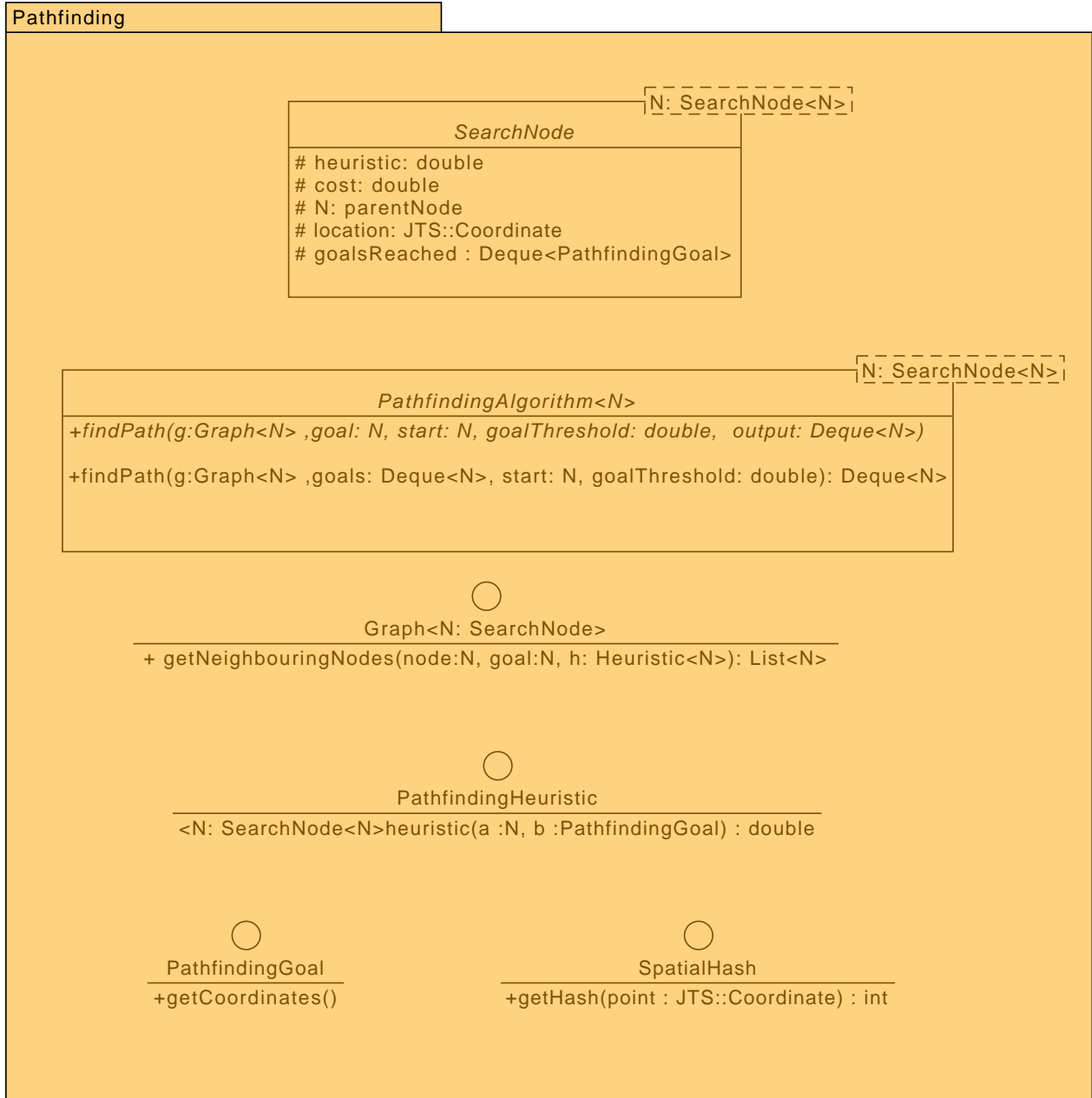


Figure 3: UML diagram of the pathfinding module

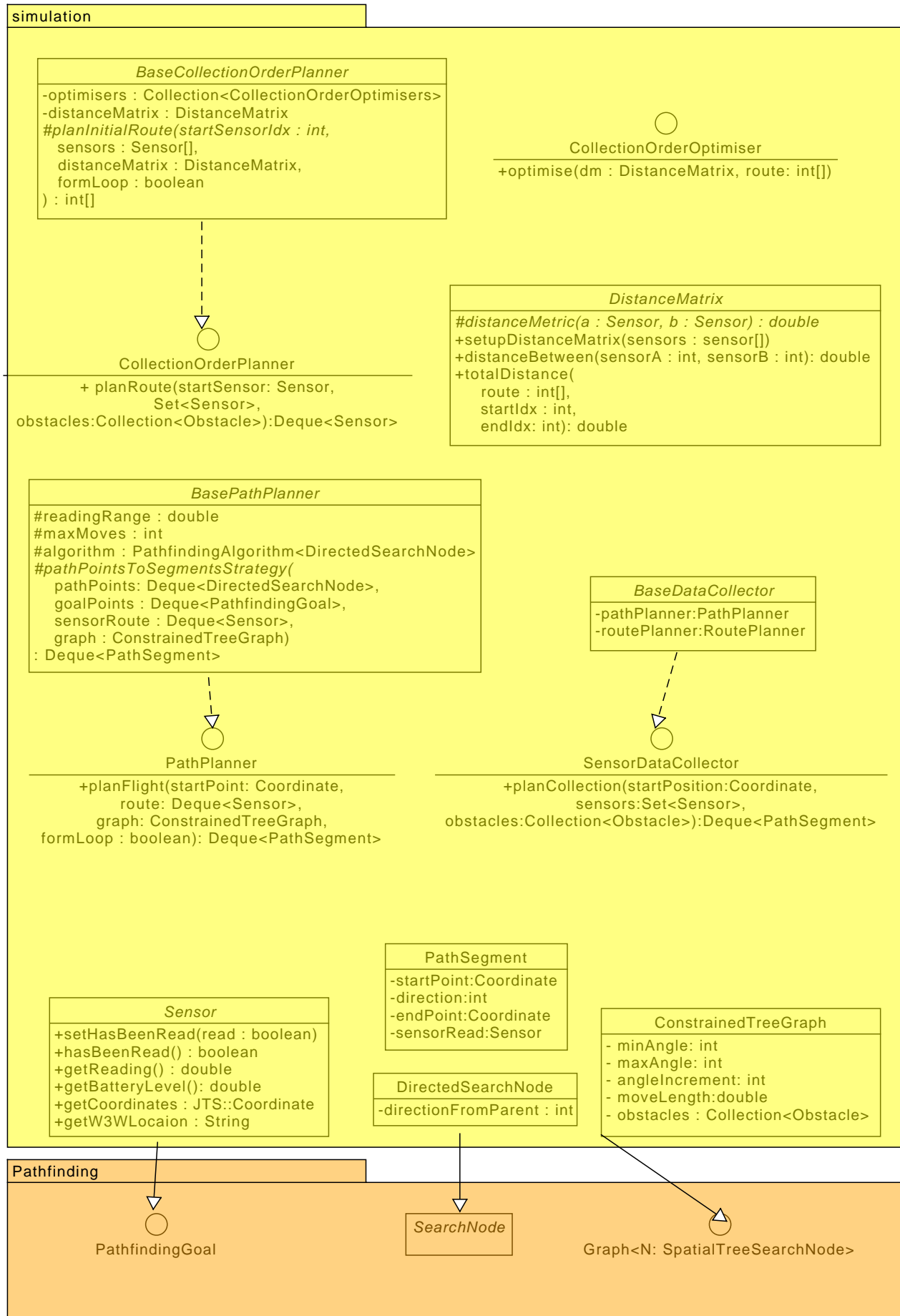


Figure 4: UML diagram of the Simulation module

**2 Class Documentation**

**3 Drone Control Algorithm**