ILP CW2

Yannik Nelson

November 30, 2020

Contents

1 Introduction

The project detailed in the coursework is to program an autonomous drone to collect readings from air quality sensors distributed around an urban geographical area as part of a research project to analyse urban air quality.
This software is to be created with the intention of passing it on to a team who will maintain and develop it further.

2 Software Architecture

As mentioned in the introduction, this software is intended to be passed on to another team and further developed, as such it is important the the structure of the software is simple, and easy to make substantial changes to. For this reason I have decided have a strong focus on dependency inversion in the architecture of this software. The structure of the system was designed to minimise coupling and maximise cohesion for this reason too.

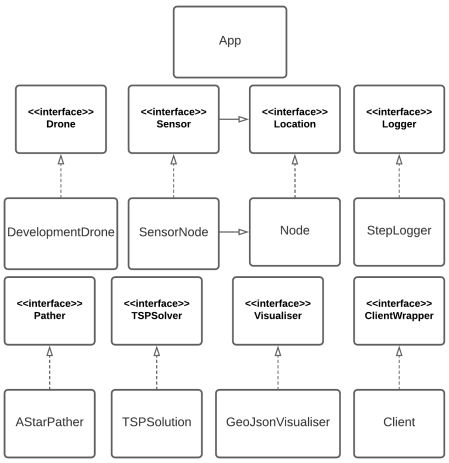
2.1 Dependency Inversion

Dependency Inversion is a principal intended to stress the decoupling of software modules. The principal states:

- High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

In order to achieve this I will be employing interface 'wrappers' around major components of the software to allow these modules to be more easily re-developed and replaced in the future.

2.2 Class Structure and Motivations



Here you can see the basic class structure I'll be using.

2.2.1 The App Class

The App class is present to simulate the drone receiving requests using the other classes to fulfill those requirests.

2.2.2 The Development Drone Class and Drone Interface

The Drone Interface is present in order to allow for multiple drone implementations

and the Development Drone class implements this interface for the drone detailed in the brief.

2.2.3 The Location Interface

I have a Location interface in order to abstract away the dependency on the mapbox library in the case the researchers decide to use another library in the future.

2.2.4 The Sensor Interface

The Sensor interface details the functions that will be required of any sensor representation. This mainly consists of getters and setters that supply required information about each sensor such as the reading and battery level. As can be seen from the diagram, the Sensor Interface extends the Location Interface, this is as we will want to be able to use Sensors as locations to reach.

2.2.5 The Node and SensorNode Classes

The Node and SensorNode classes implement the Location and Sensor interfaces respectively. As with the Sensor and Location interfaces, the SensorNode extends the Node class in order to maintain a consistent output for our locations.

2.2.6 The Client Class and ClientWrapper Interface

In the brief we are told that we will be getting our data from a webserver, but this is likely not going to be the case in a practical implementation of this project. As such the ClientWrapper interface defines the methods the drone will use to get data from sensors etc.., the Client class implements these methods for a webserver.

2.2.7 The AStarPather Class and Pather Interface

For my solution I intend on using an AStar implementation to find paths from one Location to another but in the future the researchers may want to use a different, more efficient algorithm or implementation of AStar, as such I have created a Pather Interface that will be used for any class that implements the path finding for the drone.

2.2.8 The TSPSolution Class and TSPSolver Interface

The drone will not only have to get from one Sensor to another, it will have to decide in what order in which to visit the Sensors in order to minimise the number of steps it takes. This is the Travelling Salesman Problem (TSP) and as such the drone will need to be able to solve the TSP for 34 destinations (as the drone will have to visit 33 Sensors and return to its starting position each day). There are many implementations of solutions for TSP and the researchers will likely have a preference as to which they would like to use, as such a TSPSolver interface was created in order to standardise and simplify the replacement of TSP implementations.

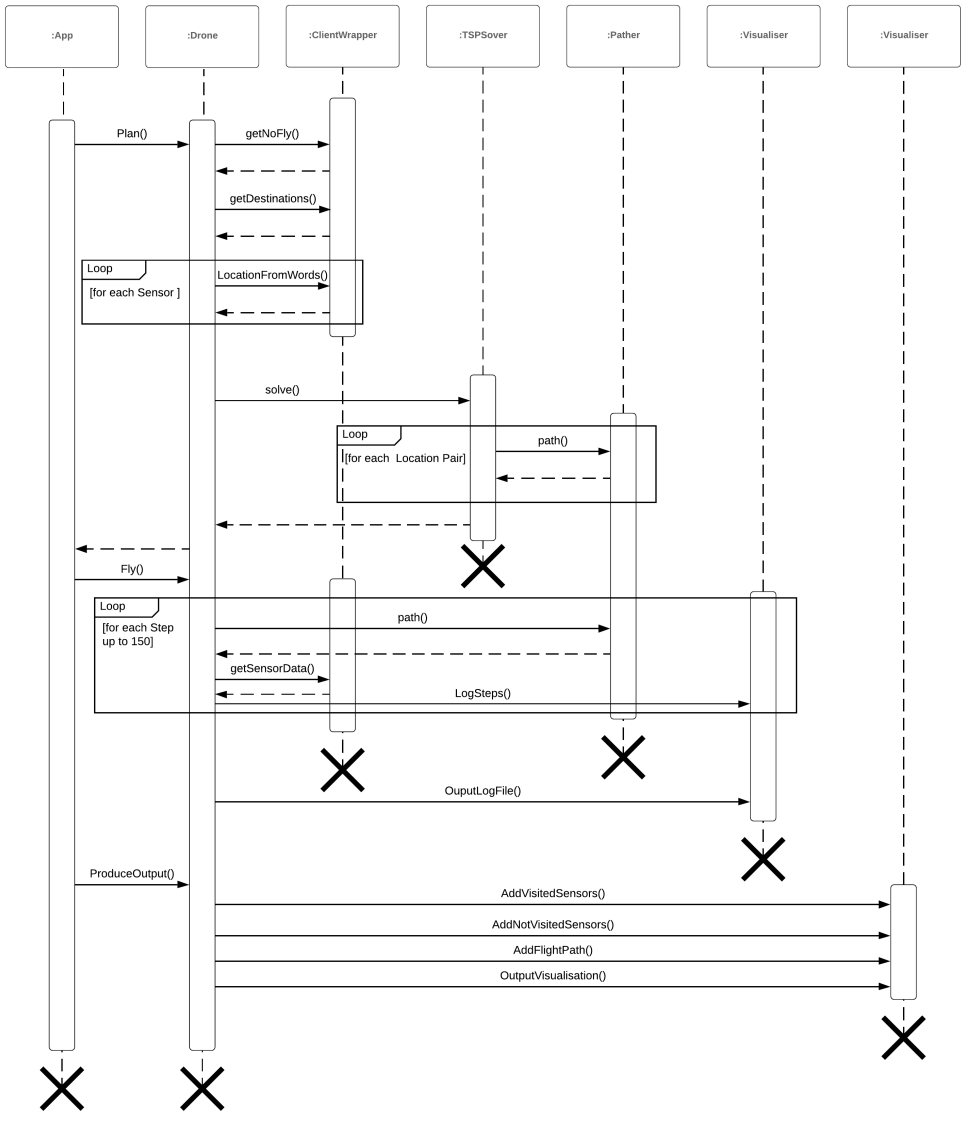
2.2.9 The GeoJsonVisualiser Class and Visualiser Interface

The brief requires the system to output a GeoJson file contining Points for all of the Sensors and a Line for the path the drone took flying to all of them. This requirement is fulfilled by the GeoJsonVisiualiser class but the visualisation requirements may change in the future or better visualisation libraries may come into existence and as such this functionality is abstracted by the Visualiser Interface.

2.2.10 The StepLogger Class and Logger Interface

The brief requires the system to output a text file detailing each step the drone took in a specified format, this functionality is handled by the StepLogger Class by the format may change and as such this functionality is abstracted by the Logger interface for easier replacement.

2.3 Basic Sequence Diagram



The above sequence Diagram details the expected interaction between objects in the planning an flying phases of the system. To reduce clutter I have excluded the calls to get location data from Location and Sensor instances (in my implementation Node and SensorNode instances). I have also excluded how the TSPSolver and Pather produce thier results internally as it is not integral to the architecture of this software as well as how the Logger is used to ensure the number of steps output stays below or equal to 150.

3 Class Documentation

The class diagram to the left gives a more detailed view of the classes used in my implementation of this project.

In the diagram I have detailed the required functions of each interface and the attributes and extra functions each of my classes use.

For my implementation I will require an implementation of the Comparator interface that can order potential paths based on their heuristic value, this is called AStarNodeComparison. I excluded this from the basic class diagram as it's specific to my implementation and not required for the general case of this problem.

In the following sections I shall detail the functions of each interfaces' and class's functions.

.pdf"
.png"
.jpg"
.mps"
.jpeg"
.jbig2"
.jb2"
.PDF"
.PNG"
.JPG"
.JPEG"
.JBIG2"
.JB2"
.eps"
[width=1.15ar

3.1 Interfaces

3.1.1 Location

This interface defines the functionality required by any representation of a location on the map. The functions are longitude() and latitude() and they return the longitude and latitude values respectively at Double precision.

3.1.2 Sensor

This interface extends the Location interface detailing the functionality required by any representation of a Sensor. The functions added to the Location functionality are: getWhat3Words() which returns the what3words coordinate of the Sensor as a String; getBattery() which returns the battery level of the Sensor with Double precision; and getReading() which returns the reading from the sensor as a String, String is used as this value received from the actual sensor can be "null" or "NaN".

3.1.3 ClientWrapper

The ClientWrapper Interface defines the minimum functionality of a client module of the drone, this is the module that communicates with 'the outside world' and gets required data. This functionality includes: getNoFly() which gets and returns a list of the areas the drone is not allowed to fly over*; getDestinations() which gets the list of sensors to be visited on the passed in date converting the recieved data into the internal Sensor representation; LocationFromWords() which will get the location data for the passed in what3words string and return the internal location representation for that location; and getSensorData() which takes in a Sensor the drone believes it's in range of and return a Sensor object containing the readings and battery level.

*Each no-fly-zone is represented as a list of Locations corresponding the vertexes of the zone's perimeter, this list is ordered so that each location-vertex is preceded and followed by the previous and next in the perimeter respectively.

3.1.4 Pather

This defines the expected functionality from any pathing modules the drone uses. setNoFLyZones() takes in and saves the areas to be avoided (in the form described above). setBounds() takes in and saves max and min values (with Double Precision) for the allowed longitude and latitude defining the bounds of the area in which the drone is allowed to fly. setStepSize() takes in and saves the length each step the drone takes must be (with Double precision). findDistance() takes in two Location and returns the distance between them (with Double precision). Finally path() which takes in a starting Location, an end Location and a tolerance (Double precision) and attempts to find a path starting at the starting Location and ending within the tolerance of the end Location while remaining within the bounds and avoiding the no-fly-zones.

3.1.5 TSPSolver

As mentioned before, the drone will require the ability to solve the Travelling Salesman Problem in order to minimise the steps it requires to travel to every sensor. The TSPSolver interface defines the function format that will be used to interact with any TSP Solution: solve() takes in a list of sensors to be visited and a star Location and returns the of Sensors in the order they should be visited.

3.1.6 Logger

The Drone is required to Log and save the details of each step it takes. This interface details the required functionality of any module that provides this logging. LogSteps() takes in a list of Locations paired with Integers which represent Locations the drone visited and the angle the drone flew at to reach that Location and a Sensor the Drone attempted to read at the end of that series of steps, this list is expected to be the path from one sensor to the next, not the complete path. OutputLogFile() which takes a path (or filename) and saves the log file to that location(/file).

3.1.7 Visualiser

The Drone is also required to output a visualisation of its activities including all of the sensors it visited, all the sensors it didn't visit and the path it flew. This Interface details the required functionality of any module that performs this visualisation: AddVisited-Sensors() takes in a list of Sensors that the drone has visited and adds them to the visualisation; AddNotVisitedSensors() takes in a list of Sensors that the drone failed to visit and adds them to the visualisation; AddFlightPath() takes in a list of Locations that the drone visited, in the order it visited it and adds it to the visualisation; and finally OutputVisualisation() which takes a path (or filename) and saves the visualisation to that location(/file).

3.1.8 Drone

Finally there's the Drone interface which dictates the activities the drone itself will perform: Plan() the functionality of planning the route and returns said route; Fly() flying the route previously planned returning nothing; and ProduceOutput() which will return nothing but save the log and visualisation for the flight.

3.2 Classes

3.2.1 Node

This is my implementation of the Location interface, Initially this stored a MapBox Point and the functions simply acted as wrappers, but this added a lot of overhead and slowed the process down significantly, as such this class now simply stores the longitude and latitude values and outputs them when requested.

3.2.2 SensorNode

This is the implementation of the Sensor interface, it extends the Node class and performs similar operatiosn for the Sensor data.

3.2.3 Client

The Brief of the project stated that all the data would be stored on a web-server, as such the client module that the drone needs to use in development is an HttpClient, this class holds that client and performs the necessary operations to get the required data from the server. At instantiation, the Client expects the port number of the webserver.

3.2.4 AStarPather

For my pathing solution I decided I would be using the AStar algorithm. I also implemented some optimisations for object avoidance which will be detailed in the Algorithm Section of this Report

3.2.5 TSPSolution

My TSPSolver implementation TSPSoltuion is a combination of the Ant Colony Optimisation and 2-Opt Heuristic. Both of these implementation require knowledge of each possible connection between destinations and the length of those connections. As such two helper functions have been created that both of these algorithms use, buildConnectionMatrix() which uses the Pather to estimate the number of steps required to get from one Sensor to another (also including the starting Location in the form of a Sensor) taking in a list of Sensors and saving the resulting matrix returning nothing; getCost() which takes in a potential ordering of Sensors and evaluates the expected step cost.
These functions are used by ACOTSP() which implements the Ant Colony Optimisation for the passed in list of destinations (sensors) assuming the connection matrix contains at least those sensors and returns the optimum order it found; and Two_Opt() which implements the 2-opt heuristic algorithm using the tryReverse() fuction to reverse sub-sections of the list for readability.

3.2.6 StepLogger

The StepLogger class implements the Logger interface. For my implementation, before adding a sensor read to the log, I ensure the drone is within range of said sensor, this requries the get distance function from the Pather implementation and as such the StepLogger requires a Pather instance in its constructor.

3.2.7 GeoJsonVisualiser

The GeoJsonVisuliser is the implementation of the Visualiser interface for outputting a GeoJson visualisation. It does this using the MapBox library. Two helper functions are used getVisitedSensorFeature() and getNotVisitedSensorFeature() which both take in a Sensor object and return a MapBox Feature consisting of a Point with the appropriate properties for if the sensor had been visited or not.

3.2.8 DevelopmentDrone

The DevelopmentDrone is the implementation of the Drone interface used for this development. It directly implements the interface with no extra functions needed, though it's constructor takes in a Pather object, a TSPSolver object, a Logger object, a Visualiser object, a ClientWrapper object and details about its start Location and permitted fly area. The Constructor was made this way to facilitate easy of testing with mock objects and to make swapping out interface implementations easy as you need only provide a different object to the drone.

3.2.9 App

This is the main class of the project. The main of this class is used to instantiate the instance of all the required objects (bar the Locations and Sensors) and calling the Drone's Plan, Fly and ProduceOutput() functions.

4 Drone Control Algorithm

4.1 AStar Pathing Algorithm

For pathing and object avoidance I decided to use the AStar algorithm as it has a simple implementation and should give the optimal result in a short time. My AStar implementation works as such Each branch consists of a list of pair of Locations and Integers,

Algorithm 1 path(Location Start, Location End, Double tollerance):

```
1: create an empty list of branches to search, call this list branches
2: create the first branch consisting of the Start and add it to the list
3: create a list of visited Locations containing the start Location
4: while True do
5:    current = the first branch in branches
6:    if current ends within the tollerance of End then
7:       return current
8:    end if
9:    available = list of all next possible Locations from current
10:   remove current from branches
11:   for Each Location l in available do
12:      if l is not in visited then
13:         create a copy of current, place l as the end, add this branch to branches
14:      end if
15:   end for
16:   sort branches by the heuristic value of each branch
17:   add the last Location from current in visited
18: end while
```

Location is a Location visited by the drone, the Integer is the angle the drone flew at to reach that Location, conceptually we shall ignore the Integer for AStar.

This algorithm in itself does not have any object avoidance in it. I produced the object avoidance in the function that gets next possible locations:

Algorithm 2 Reachable(Location Current, List⟨*Location*⟩ Visited):

```
1: nextPoints = and Empty list of pairs of Locations and Integers
2: OuterLoop:
3: for i from 0 to 355 increasing by 5 (We run through every angle the drone can move
   at (every multiple of 5)) do
4:    long = Longitude after from moving the stepsize from current Location at angle i
5:    if long is out of bounds then
6:       continue OuterLoop
7:    end if
8:    lat = Latitude after from moving the stepsize from current Location at angle i
9:    if lat is out of bounds then
10:      continue OuterLoop
11:   end if
12:   newLoc = Location made from long and lat
13:   for Each Visited Node v do
14:      if newLoc is within a step of v then
15:         continue OuterLoop
16:      end if
17:   end for
18:   for Each no-fly-zone bounding box b do
19:      if The line segment from the current Location to newLoc intersects the bounding
         box then
20:         for Each line segment l of the no-fly-zone perimeter do
21:            if The line segment from the current Location to newLoc intersects l then
22:               continue OuterLoop
23:            end if
24:         end for
25:      end if
26:   end for
27:   Add the Pair of newLoc and i to next Points
28: end for
29: return nextPoints
```

This algorithm finds all the possible next Locations a drone could visit from a given Location, and removes any that are close to a Location already visited and any Locations that would require the drone to enter an area in which it is not allowed.

The code for checking the intersection of two line segments was taken from here:
`https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/`

4.2 Travelling Salesman

4.2.1 Connection Matrix Creation

Both of my TSP algorithms require a look up table from one destination to another containing the expected cost to travel from the first the the second, here is the algorithm I used to create said table: This function creates a mapping from a combination of two

Algorithm 3 buildConnectionMatrix(List⟨*Sensor*⟩ list):

```
1: connectionsMatrix = HashMap Indexed by Sensors holding a HashMap Indexed by
   Sensors holding Integers
2: for Each Sensor to in list s do
3:    connectionsMatrix.put(s, new HashMap Indexed by Sensors holding Integers)
4: end for
5: for Each Sensor in list s1 do
6:    for Each Sensor in list s2 do
7:       if s1 == s2 then
8:          connectionsMatrix.get(s1).put(s2, 0)
9:       else
10:         connectionsMatrix.get(s1).put(s2, pather.path(s1, s2, 0.0002).size())
11:      end if
12:   end for
13: end for
```

Sensors s1 and s2 to an Integer equal to the estimated number of step to reach s2 from s1 (+1 to allow for inconsistencies in tolerances and starting positions).
To keep track of the progress of producing this matrix i have used the ProgressBar library: `https://tongfei.me/progressbar/`

4.2.2 Path Evaluation

Both of the algorithms also need to be able to measure the cost of a produced route: This

Algorithm 4 getCost(List⟨*Sensor*⟩ list):

```
total = 0
for Each index in list starting at the second element do
   total += connectionMatrix.get(list.get(index-1).get(list.get(index))
end for
total += connectionMatrix.get(list.get(last index).get(list.get(0))
return total
```

sums the lengths of every connection in the route (including the return to the starting Sensor).

4.2.3 Ant Colonly Optimisation TSP Solution

The first part of my TSP solution is to run the Ant Colony Optimisation. This works by simulating a series of ants traversing the connections of the 'graph' and laying down pheromone as they go, with their decision guided by connection lengths and pheromone strengths:

Algorithm 5 ACOTSP(List⟨*Sensor*⟩ list):

```
1: pheromone = copy of connectionMatrix with 1 at every Mapping
2: ants = empty List of Lists of Sensors
3: best = copy of list bestCost = getCost(best)
4: for t from 0 to 99 do
5:    for k from 0 to the size of list do
6:       possibleNext = copy of list
7:       ant = new empty List of Sensors
8:       pick a random Sensor from possibleNext and add it to ant and remove it from
         possibleNext
9:       while possibleNext is not empty do
10:         pick a next Sensor from possibleNext with probability based off of the length
            of the connection to that sensor and the pheromone strength on that same
            connection (equation bellow).
11:         Add the chosen Sensor to ant and remove it from possibleNext
12:      end while
13:      Add ant to ants
14:   end for
15:   Apply evaporation to the pheromone Matrix by multiplying every element in it by
      1-evap
16:   for every ant in ants do
17:      currentCost = getCost(ant)
18:      if currentCost < bestCost then
19:         best = ant
20:         bestCost = currentCost
21:      end if
22:      for each connection the ant travlled down do
23:         update that connection's pheromone strength by adding 1/currentCost
24:      end for
25:   end for
26:   return best
27: end for
```

$$prob = \frac{p_{pn,c}^{a} + \frac{1}{d_{pn,c}}^{b}}{\sum_{next} p_{next,c}^{a} + \frac{1}{d_{next,c}}^{b}}$$

The probability of moving to Sensor pn from Sensor c is the strength of the pheromone on that connection + 1\the length of that connection (raised to the power of b), all divided by the sum of that calculation for every possible next Sensor. This equation allows the

ants to take into account both the length of a connection and the pheromone on it for influencing the paths they take, raising the pheromone and inverse of the distance to powers a and b respectively allow us to weight how important each factor should be when deciding where to go next.

4.2.4 2-Opt Heuristic TSP Solution

The 2-Opt Heuristic algorithm works by reversing every subsection of a route and seeing if that action improved the cost of the route. If the cost improved by reversing any subsection, then this process is run again until no improvement is found by reversing a subsection of the route. Her is the pseudocode for my implementation: The Algorithm for

Algorithm 6 Two_Opt(List⟨*Sensor*⟩ list)

```
better = True
while better do
  better = False
  for Integer j from 1 to the size of the list do
    for Integer i from 0 to j do
      better = tryReverse(list, i, j)
    end for
  end for
end while
```

Algorithm 7 TryReverse(List⟨*Sensor*⟩ list, Integer i, Integer j)

```
Intial = getCost(list)
while better do
  better = False
  reverse sublist of list from i to j in place
  if getCost(list) Initial Intial then
    return True
  end if
  reverse sublist of list from i to j in place again (to undo the change)
  return False
end while
```

the Ant Colony Optimisation and the idea to run the Two-Opt heuristic on the rest was from: `https://www.sciencedirect.com/science/article/pii/S1002007108002736#fd1`

5 Examples

Bellow are two examples of paths produced by this system:

Path From 15/01/2021

Path From 15/06/2021

Both of these examples used the same random seed and starting position:

seed	latitude	longitude
5678	55.944	-3.1878

The first example (15/01/2021) took 15 seconds and 85 steps to complete. The second example (15/06/2021) took 29 seconds and 76 steps to complete.

6 Additional Dependencies

I shall now list all of the sources mentioned and dependencies used in this project:

6.1 Sources

- Line segment intersection detection:
  `https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/`
- Ant Colonly Optimisation algorithm and source idea to also use 2-Opt:
  `https://www.sciencedirect.com/science/article/pii/S1002007108002736#fd1`
- The algorithms for the AStar Search and 2-Opt algorithms were learned in the Reasoning And Agents course.

6.2 Dependencies

- MapBox used for outputting the GeoJson visualisation
  `https://docs.mapbox.com/android/java/overview/`
- Gson used to aid the Json parsing of Sensors `https://github.com/google/gson`
- javatuples used to pair locations in a path with the angle traveled to reach them
  `https://www.javatuples.org/index.html`
- progressbar used to display progress creating the connectionsMatrix
  `https://tongfei.me/progressbar/`
- Mockito used to aid in testing `https://site.mockito.org/`