Informatics Large Practical: Report

Samo Novák
(s1865783)

Semester 1, 2020/2021

Contents

1 Software architecture description **2**
1.1 Functional classes . . . 2
1.2 Data classes . . . 2
1.2.1 Miscellaneous . . . 3
1.3 External libraries . . . 3

2 Class documentation **3**
2.1 AirQualityMapsApp . . . 3
2.2 WebClient . . . 3
2.3 PathPlanner . . . 4
2.4 ObstacleEvader . . . 4
2.5 DroneController . . . 5
2.6 W3WDetails . . . 6
2.7 SensorReading . . . 6
2.8 Obstacle . . . 7
2.9 Move . . . 7
2.10 RotationDirection . . . 7
2.11 ColourSymbols . . . 8
2.12 Pair<Left, Right> . . . 8
2.13 WebClientException . . . 8

3 Drone control algorithm **9**
3.1 Path planning . . . 9
3.1.1 Generating the sequence . . . 10
3.1.2 Optimizing the sequence . . . 10
3.1.3 Preparing the plan for execution . . . 11
3.2 Executing the path plan . . . 11
3.3 Computing where to go . . . 12
3.4 Handling Nearby Sensors . . . 12
3.5 Example paths . . . 12

References **13**

1 Software architecture description

This section describes the architecture of the project in terms of classes and their interaction. The classes can be in general split into two categories: functional classes that perform computation and control the drone, and data classes which act as data storage.

1.1 Functional classes

The main class of the project is **AirQualityMapsApp** (section 2.1). This is used for running the program - it takes the inputs, instantiates other classes and calls the appropriate method. It is the ultimate controller of the program's execution.

The first class used by **AirQualityMapsApp** is the class **WebClient** (section 2.2). This performs all access to the web server and provides methods to load the different types of content needed. It was chosen to separate all of the workings of the web interface from the rest of the program.

The computations for the drone itself are split between different phases:

- *planning* – this constitutes creating a high-level sequence of waypoints as guidance for the drone, which is an abstract representation of the drone's objectives,
- *control/simulation* – during which the drone takes the high-level flight plan an executes it, taking into account contraints on its motion created by the specification (individual steps are of length 0.0003 degrees; allowed directions are multiples of 10 degrees, counterclockwise from the axis of positive longitude), as well as the no-fly zones and the boundaries of the confinement area. In the future development, this could also include corrections for wind and other environemntal influences.

The planning phase is performed by the **PathPlanner** class (section 2.3), which creates a sequence of waypoints, taking into account the locations of sensors and their distance from each other, as well as the obstacles, i.e. no-fly zones defined around certain buildings.

The **PathPlanner** works together with the **ObstacleEvader** class (section 2.4), which provides support for geometrical computation: it helps with finding intersections of possible plan segments with the no-fly zones. This also takes care of the confinement area, which is seen as a no-fly zone extending outwards from the boundaries.

The other phase, control (and so far simulation, as this project merely simulates a possible future drone), is embodied by the class **DroneController** (section 2.5). This class is responsible for controlling the flight of the drone, following the waypoints generated by **PathPlanner**. It interacts with the **ObstacleEvader** as well, making sure each step is valid, i.e. does not intersect a no-fly zone, and by extension that it does not cross the boundaries of the confinement.

1.2 Data classes

These classes serve as storage and processing of data. In particular, classes **W3WDetails** and **SensorReading** are used for deserializing JSON data from the web server, which is a requirement of the library *GSON* used to process JSON files.

W3WDetails (section 2.6) holds the translation of a *What 3 Words* address into a location in terms of longitude and latitude.

The sensor locations, along with their readings (for the purpose of this project, which simulates the sensors) are held in **SensorReading** (section 2.7).

Each no-fly zone, including the space outside of confinement, is represented as a polygon describing its external boundary (i.e. there are no holes in the zones - these would be unreachable anyway). This is ultimately encoded as a list of points. However, since these are very common in the code, in order to improve readability and make intentions clear, a new class **Obstacle** (section 2.8), which contains those points, is used to represent the no-fly zones.

There is a class **Move** (section 2.9), which represents a move that the drone can make. It is beneficial to have all the data about a move in one place, because it simplifies the storage of data and its serialization into output files.

Another abstraction, in this case an *enum*, is **RotationDirection** (section 2.10), which encodes the direction of rotation (in the usual sense of positive being counterclockwise, with axis taken to go upwards). This would be easily encoded with just numbers, but the enum abstraction makes the usage more clear in terms of semantics, and also limits the allowed values.

The class **ColoursSymbols** (section 2.11) is responsible for generating properties of markers – their colour and symbol – based on the data read from sensors. Similarly to previous classes, it is used to that everything that has to do with colours and symbols of markers is in one place.

1.2.1 Miscellaneous

The code base contains two more classes. These are separate in a way – though they are used to handle data, they are used more as a convenience.

The first of these is class **Pair<Left, Right>** (section 2.12), where **Left** and **Right** are arbitrary types. This represents an arbitrary pair (2-tuple) of possibly different data types, and is designed to mimic the behaviour of languages where creation of arbitrary tuples is commonplace. I use it to return multiple values from a single function call, where the two values are related (belong together in some way) and it would be inconvenient to create a new class for this purpose.

The other class in this section relates to **WebClient** and it is the **WebClientException** (section 2.13), which unifies all the reasonably expectable exceptions from the **WebClient**. The reason for using such class is that, while there are many possible things that can go wrong when trying to load data from a web server, these should all be ultimately handled the same way: the program should terminate, because it is impossible to do anything of value without having the data from the server. Therefore, this single unified exception is provided, so that the program knows that an error happened in the web server.

1.3 External libraries

The project includes two external libraries: **GSON**, Google's libary for processing JSON in general, and **GeoJSON**, MapBox's library for handling GeoJSON (JSON containing geographical data) in particular.

2 Class documentation

2.1 AirQualityMapsApp

The main class of the project. It has the following methods:

Methods

- *public static* void **main**(String[] args)
  The main function of the program. Handles the terminal arguments which are: day, month, year, latitude, longitude, randomness seed (not used in the program), server port. This is the part of the codebase which controls all the rest.
- *private static* String **formatDateDMY**(int day, int month, int year)
  A helper function used to format the date as `DD-MM-YYYY`.

2.2 WebClient

Web client used for loading (and parsing) relevant data from the HTTP server. It contains the following local properties and methods:

Properties

- *private final* String **host** `= "http://localhost";`
  The connection host. It has the value *localhost*, as the program is connecting to a local server.
- *private final* int **port**
  Web server port.
- *private* HttpClient **client**
  The client instance. Instantiated once with and then used throughout.

Methods

- *public* **WebClient**(int port) – constructor
- *private* String **load**(String path)
  The internal method for loading data as a **String** from the server.
  On error, it throws **WebClientException** (section 2.13).

- *public* ArrayList<SensorReading> **loadSensorList**(int year, int month, int day)
  Load and parse the list of sensors for a particular day, which is returned as a list of **SensorReading** (section 2.7). Uses the next function to translate *What 3 Words* addresses.
  Also throws **WebClientException**.

- *private* Point **loadPointFromWords**(String words)
  Translate an address in *What 3 Words* into a **Point** (a class from *MapBox GeoJSON* library).
  Throws **WebClientException**.

- *public* FeatureCollection **loadNoFlyZones**()
  Loads no-fly zones, parses them using *MapBox GeoJSON*.
  Throws **WebClientException**.

2.3 PathPlanner

The class responsible for high-level planning of waypoints for the drone to follow.

Properties

- *private final* ArrayList<SensorReading> **sensorList**
  Contains the list of sensors as **SensorReading**.

- *private final* int **NUMBER_OF_SENSORS**
  The number of sensors. This is used in the path-finding and optimization methods of the Planner.

- *private* double[][] **distances**
  The distance matrix of the undirected weighted graph containing as vertices the locations of sensors and also the initial drone location.
  Has the property that distances$[i][j] \equiv$ distances$[j][i]\, \forall i, j$ (i.e. is *symmetric*). (See section 3.1)

Methods

- *public* **PathPlanner**(ObstacleEvader evader, ArrayList<SensorReading> sensorList) – constructor
  Apart from storing values, it also precalculates the distance matrix for later use.

- *private* double **distance**(Point x, Point y)
  Computes the Euclidean distance between the two points.

- *public* ArrayList<Point> **findPath**(double startLatitude, double startLongitude)
  The function for finding waypoints to guide the drone. Calculates the distance matrix entries for the drone initial position, and generates the sequence of waypoints using **solveTSP** (see below).

- *private* ArrayList<Point> **solveTSP**()
  Generates the sequence of sensors to be visited in an order that is heuristically efficient. It works by recasting the problem as a *Travelling Salesperson Problem* (TSP), where the vertices of the graph are sensor locations. The initial TSP circuit is generated by **nearestInsert** and then optimized by **flipSwap** (see both below). It **guarantees** that the drone's initial location is the first element of the sequence.

- *private* ArrayList<Point> **nearestInsert**()
  Generates a TSP circuit using the *Nearest Insert* heuristic. (See Algorithm 1)

- *private* void **flipSwap**(ArrayList<Point> sequence)
  Optimize (in place) the given TSP circuit (sequence of vertices) using the *Flip + Swap* heuristic. (See Algorithm 2)

2.4 ObstacleEvader

This class is used to handle the geometry of obstacles (the no-fly zones) - finding whether lines (e.g. flight path segments or drone moves) intersect any of them; and to help the drone with their evasion.

Properties

- *private final* double **LAT_MAX, LAT_MIN, LON_MAX, LON_MIN**
  Boundaries of the confinement area.
- *private* ArrayList<Obstacle> **noFlyZones**
  Array containing no-fly zones as lists of Points
- *private* HashMap<Obstacle, Point> **averages**
  A hash map assigning to each no-fly zone the average of its points, meant to be the zone's centre in a way.

Methods

- *public* **ObstacleEvader**(FeatureCollection noFlyZones) – constructor
  During initiation, it also unpacks the no-fly zones from a list of **Feature** to a list of **Obstacle**s. It adds the confinement area to this list as well, as all space outside the confinement can be seen as another no-fly zone. For each zone, it computes the centre (see **averages** above).
- *private* Point **getAveragePoint**(List<Point> points)
  Compute the average point. Does not count the last point into this average, because that is always the same as the first (due to the points coming from a closed polygons).
- *public* boolean **crossesAnyObstacles**(Point a1, Point a2)
  Checks if any of the no-fly zones are crossed by a line segment from a1 to a2 (denoted a1 — a2).
- *public* Obstacle **nearestCrossedObstacle**(Point a1, Point a2)
  Find the obstacle intersected by the line segment a1 — a2. If multiple obstacles are found, returns the one that is nearest to a1.
- *private* ArrayList<Obstacle> **allCrossedObstacles**(Point a1, Point a2)
  Finds all the zones crossed by this line.
- *private* boolean **crossesOneObstacle**(Point a1, Point a2, Obstacle obstacle)
  Checks if the line segment a1 — a2 crosses the obstacle.
- *private* ArrayList<Pair<Point, Integer>> **obstacleIntersections**(Point a1, Point a2, Obstacle obstacle)
  Finds all the intersections of line segment a1 — a2 with the obstacle. It returns a list of **Pair**s, where *left* is the point of intersection and *right* is the index i of the vertex of the polygon, such that a1 — a2 intersects the line segment i — $((i+1) \mod n)$, where $i, (i+1) \mod n$ are points at respective indices of the obstacle and n is the number of points of the obstacle.
- *private* Point **intersection**(Point a1, Point a2, Point b1, Point b2)
  Finds the point of intersection of the lines a1 — a2 and b1 — b2.
- *public* RotationDirection **chooseEvasionDirection**(Obstacle obstacle, Point origin, double angle)
  Decide in which direction the drone should go to avoid this obstacle. (See 3.3)

2.5 DroneController

The class responsible for controlling (simulating) the drone operations, collecting data from sensors and exporting data into the output files.

Properties

- *private final* double **MOVE_LENGTH** = 0.0003
  Length of allowed drone moves.
- *private final* double **SENSOR_READ_MAX_DISTANCE** = 0.0002
  Maximum distance over which the drone is able to connect to a sensor. It is not inclusive, meaning the actual distance has to be strictly less.
- *private final* int **MAX_BATTERY** = 150
  Maximum (and initial) battery charge.

- *private final* ArrayList<SensorReading> **sensors, visitedSensors**
  List of all sensors on the map. (For **SensorReading** see 2.7)

- *private* ArrayList<SensorReading> **visitedSensors**
  Those sensors that have already been visited.
- *private* int **battery** and *private* double **latitude, longitude**
  The state variables of the drone. Battery is initialized to **MAX_BATTERY**.
- *private* ArrayList<Move> **trajectory**
  The record of all the moved made by drone. (for **Move** see 2.9)
- *private* ObstacleEvader **evader**
  Used for avoiding hitting no-fly zones.

Methods

- *public* **DroneController**(ArrayList<SensorReading> sensors, ObstacleEvader evader) – constructor
- *public* void **executePathPlan**(ArrayList<Point> waypoints)
  Takes the high-level waypoints and navigate the drone along them, updating its internal state at every point and recording moves in **trajectory**. (See 5)
- *private* Point **computeMove**(double angle)
  Return the location where the drone would arrive after executing a move in specified angle from its current location.
- *private* Obstacle **moveIntersectsObstacle**(Point end)
  Return the nearest no-fly zone intersected by a move from the current location to end.
- *private* ArrayList<SensorReading> **getUnvisitedSensorsInRange**()
  Gets the sensors that are in the drone's current range, could be read and have not been read yet.
- *private* double **droneDistance**(SensorReading sensor *OR* Point pt)
  Computes the distance to the sensor or point from the current location of the drone.
- *public* void **serializeTrajectory**(String flightpathFilename, String readingsMapFilename)
  Generates the flight-path and map files with specified filenames.
- *private* Feature **createSensorMarker**(SensorReading sensor, boolean visited)
  Creates the marker for the specified sensor, taking into account whether thedrone was visited. This function uses **ColoursSymbols** (section 2.11).

2.6 W3WDetails

Class used to deserialize *What 3 Words* address translations. It has no methods.

Properties

- String **words** – the *What 3 Words* address
- CoordinatesObject **coordinates** – the coordinates
  CoordinatesObject is a local class that contains two doubles: **lng** (longitude) and **lat** (latitude). This is to reflect the format given in the files from web server.

2.7 SensorReading

This class represents a sensor and its readings. It is used to deserialize sensor data from JSON.

Properties

- String **location** and double **lat**, **lon**
  Location as *What 3 Words* and also GPS.
- double **battery** and String **reading**
  Data from the sensor - its battery and reading of air pollution.

Methods

- *public* Point **toPoint**() – convert to *MapBox GeoJSON* Point

2.8 Obstacle

This class represents a no-fly zone, which is encoded as a list of points defining the zone's external boundaries. The class **Obstacle** is therefore a **subclass of**, and acts as an alias for: **ArrayList<Point>** (where **Point** is provided by *MapBox GeoJSON* library). Functionally, it behaves the same, but I chose to use it to improve readability. Apart from having all the properties and methods of **ArrayList<Point>**, it also has:

Methods

- *public static* Obstacle **fromList**(List<Point> list)
  This is a static function that creates an instance of **Obstacle** from any kind of list containing points. It is needed when extracting a list of points from *MapBox*'s class **Polygon**.

2.9 Move

This class is a representation of the drone's move. It is used to record trajectory and collected data.

Properties

- *private final* Point **original, next**
  Start and end point of the drone's step.
- *private final* int **direction**
  Direction in degrees.
- *private final* SensorReading **sensor**
  The recorded sensor readings at the end of the move.

Methods

- *public* **Move**(Point original, int direction, Point next, SensorReading sensor) – constructor
- getters for **direction**, **sensor**, **original**, **next**
- *public* String **serialize**(int moveNumber)
  This function serializes the **Move** in the format required by the specification for the *flightpath* file. Argument moveNumber is the number of the move, i.e. the first part of the serialized string, and it is there because the **Move** does not know when it happened.

2.10 RotationDirection

An enumerated type that represents a direction of rotation. The convention used is the usual one, where positive rotation means going counterclockwise about some axis. This enum provides a nice semantic abstraction, in contrary to using just an integer value.

Values

- *None* – corresponds to no rotation and is encoded by 0
- *Positive* – positive rotation (counterclockwise), encoded by $+1$
- *Negative* – negative rotation (clockwise), encoded by -1

Properties

- *private* int **value**
  The internal storage of the value, which can be one of the above

Methods

- **RotationDirection**(int value) – constructor
- *public* int **getValue**()
  Getter for **value**
- *public* boolean **equal**(RotationDirection direction)
  Checks two objects of type **RotationDirection** for equality.

2.11 ColourSymbols

This is a class used to decide the properties of sensor markers. It contains two enums, implemented this way them because they provide a nice abstraction of the relevant values.

- *public* enum **Colour**
  Values: *Green, MediumGreen, LightGreen, LimeGreen, Gold, Oragne, RedOrange, Red, Black, Gray*
- *public* enum **Symbol**
  Values: *Lighthouse, Danger, Cross, None*

Each enum has a property *private* String **value**, that stores internally the string that the enumerated value corresponds to: in the case of **Colour** these are the hexadecimal colour codes; and in the case of **Symbol** these are symbol names. Each enum also has the following functions:

Methods of Colour and Symbol enums

- **Colour**(String colour) and **Symbol**(String symbolString) – constructors
- *public static* {Colour *OR* Symbol} **getFromPollution**(double pollution)
  Choose the appropriate enumerated value depending on the value of pollution. These functions assume a valid pollution value, i.e. $0 \leq \text{pollution} < 256$.
- *public* String **getValue**() – getter for **value**

Methods of ColourSymbols

- *public static* Pair<Colour, Symbol> **getColourSymbol**(String reading, double battery)
  Decides both the **Colour** and **Symbol** based on sensor readings. (For **Pair<Left, Right>** see 2.12).
- *public static* Pair<Colour, Symbol> **getNotVisited**()
  Returns the properties of an unvisited sensor. To make it less expensive, it only returns a preexisting property *private final static* Pair<Colour, Symbol> **notVisited**.

2.12 Pair<Left, Right>

A generic class representing a pair (2-tuple) of arbitrary values and possibly different types. It mimicks the behaviour of 2-tuples in languages where they are prevalent - it is an immutable data type intended simply as storage. This construct is quite convenient for example when a function needs to return two related values of different types.

Properties

- *public final* Left **left**
- *public final* Right **right**

It is worth noting that both of these are constants and are directly accessible without the need of a getter function – this is a deliberate due to **Pair** being a purely storage data type.

Methods

- *public* **Pair**(Left left, Right right) – constructor
- *public* Pair<Left, Right> **clone**()
  Function that returns a shallow clone of the original **Pair**. This means it creates a new **Pair** instance and assigns **left** and **right** to it respectively – however it does not clone the internal values.

2.13 WebClientException

This is a unifying exception class for multiple different exceptions that can occur when attempting to connect to the web server and load files from there. It is thrown by **WebClient** (section 2.2).

Properties

- *private static final* long **serialVersionUID**
  Serial number required by Java. No specific significance for our project.
- *private final* String **comment**
  Error message or comment.
- *private final* Exception **originalException**
  Used to store the original exception that caused the **WebClient** to thrown **WebClientException**. This is used for printing stack stace.

Both **comment** and **originalException** properties can be *null*, but then the exception conveys little useful information. Conventionally, **comment** should not be *null*.

Methods

- *public* **WebClientException**(String comment, Exception originalException) – constructor
- *public* void **printStackTrace**()
  This function prints the **comment** followed by the original stack trace that caused the exception (so the stack trace of **originalException**) if they exist. The values are printed to standard error output.

3 Drone control algorithm

As noted at the start, there are two phases of the drone control:

- *planning* (handled by **PathPlanner** (section 2.3))– creating a high-level sequence of waypoints that the drone will later use for navigation, and
- *control/simulation* (handled by **DroneController** (section 2.5)) – executing this high-level plan, taking into account specific contraints on the drone's motion, such as no-fly zones.

This allows for efficient planning of overall order of sensors to visit, where the specific actions of the drone will be determined during the flight. This is beneficial also for the future development, when the drone might need to take into account wind and possibly other phenomena.

Both of these phases are are described in the following sections:

3.1 Path planning

In this first phase, the problem of navigation at hand can be reduced to an instance of the *Travelling Salesperson Problem*. First, some definitions:

We have a *complete undirected graph* $G = (V, E)$, where V is the set of vertices and $E = V \times V$ is the set of edges, $N = |V|$ being the number of vertices. There is a weight function $w : E \to \mathbb{R}_{\geq 0}$ which assigns to each edge a real, nonzero weight; and which is symmetric, i.e. $\forall (u, v) \in E.\ w(u, v) = w(v, u)$.

The set V of vertices contains the sensors (mainly the points where they are located) and the location where the drone starts. In the program, these vertices are represented as integers, such that a vertex $i \in V$ corresponds to the following

- the sensor at index i in **PathPlanner.sensorList** if $i < N - 1$; where indexing starts at 0,
- the starting location of the drone if $i = N - 1$ (which is the maximum i).

The graph is complete (meaning all edges exist, $E = V \times V$), because the drone can conceivably travel between any two vertices.

The weight function is represented by the distance matrix **PathPlanner.distances** (which will be also denoted by $\mathbf{D}$, with components denoted $\mathbf{D}[i, j]$ for good readability). This weight function/distance matrix satisfies the following properties $\forall i, j \in V$:

$$w(i, j) = \mathbf{D}[i, j] = \mathbf{PathPlanner.distances}[i][j] \geq 0 \qquad \mathbf{D}[i, j] = \mathbf{D}[j, i] \qquad \mathbf{D}[i, i] = 0$$

The components of $\mathbf{D}$ are not just defined by the Euclidean distance, however. Here, the **ObstacleEvader** (section 2.4) comes into play and the distance matrix will depend on whether the particular edge intersects a no-fly zone:

$$\forall i \neq j.\ \mathbf{D}[i, j] = \begin{cases} \text{EuclideanDistance}(i, j) & \text{line between } i, j \text{ does not intersect obstacles} \\ d^{\uparrow} & \text{otherwise} \end{cases}$$

where $d^{\uparrow}$ is some arbitrary number larger than any possible Euclidean distance within the graph G (e.g. 1 degree). This means that a direct line between two consecutive waypoints may intersect a no-fly zone, and then the drone will have to handle the evasion itself; but this situation is disfavoured in comparison to waypoints that have an easy straight-line connection without intersecting obstacles.

A *TSP tour* in graph G is a *Hamiltonian cycle*, i.e. a path through the graph that visits each vertex exactly once, with the exception of the initial vertex, where the path has to return in the end. This can be represented by a sequence of vertices $\mathcal{S}$ that has to contain each vertex exactly once, hence being a permutation of V (it is understood that the path will loop back to the start after the last vertex). This sequence $\mathcal{S}$ is represented as a list in the program.

3.1.1 Generating the sequence

In the implementation, the sequence $\mathcal{S}$ is generated using the *Nearest Insert heuristic.*[1] The specific algorithm used is described below:

Algorithm 1: Nearest Insert

Input: an undirected complete graph $G = (V, E)$, where $|V| = N$
Input: a distance matrix $\mathbf{D} = (\mathbb{R}_{\geq 0})^{N \times N}$
Result: a permutation $\mathcal{S}$ of V (represented as list)

$\mathcal{S} \leftarrow [i, j]$, where $(i, j) = \operatorname{argmin}_{(i,j) \in V \times V} \mathbf{D}[i, j]$; /* the two nearest vertices */
$U \leftarrow \{x \in V \mid x \notin \mathcal{S}\}$; /* set of unused vertices */
while *there are unused vertices, i.e.* $U \neq \varnothing$ **do**
  $(i, u) \leftarrow \operatorname{argmin}_{(i,v):\, i \in \{0,\ldots,|\mathcal{S}|-1\},\, v \in U} \mathbf{D}[\mathcal{S}[i], v]$; /* find a vertex $u \in U$ that is nearest to some vertex $\mathcal{S}[i] \in \mathcal{S}$ */
  $d_- \leftarrow \mathbf{D}\left[u, \mathcal{S}\left[(i-1) \mod |\mathcal{S}|\right]\right]$; /* distance between u and the vertex before $\mathcal{S}[i]$ in $\mathcal{S}$ */
  $d_+ \leftarrow \mathbf{D}\left[u, \mathcal{S}\left[(i+1) \mod |\mathcal{S}|\right]\right]$; /* distance between u and the vertex after $\mathcal{S}[i]$ in $\mathcal{S}$ */
  if $d_- < d_+$ **then**
    Insert u into $\mathcal{S}$ at position i ; /* before $\mathcal{S}[i]$ in $\mathcal{S}$ */
  else
    Insert u into $\mathcal{S}$ at position $((i+1) \mod |\mathcal{S}|)$; /* after $\mathcal{S}[i]$ in $\mathcal{S}$ */
  Remove u from U.
return $\mathcal{S}$;

3.1.2 Optimizing the sequence

Afterwards, when $\mathcal{S}$ is generated, it is optimized further by a heuristic *Flip + Swap*. This is a simple algorithm for *local optimization* that in each iteration checks whether flipping the other of two consecutive vertices in $\mathcal{S}$ produces a shorter overall cycle length and then it tries to swap the endpoints of two different edges. This particular implementation was chosen because it provides reasonably good results in testing.

Algorithm 2: Flip + Swap

Input: an undirected complete graph $G = (V, E)$, where $N = |V|$
Input: distance matrix $\mathbf{D}$
Input: a permutation $\mathcal{S}$ of V as list
Result: an optimized permutation $\mathcal{S}$ (in place)

for $i \leftarrow 0 \ldots N+1$ **do**
  for $i \leftarrow 0 \ldots N+1$ **do**
    if $i = j$ **then**
      Skip this iteration.
    $V_i \leftarrow$ subsequence $\{\mathcal{S}[i+k \mod N] \mid k = -1, 0, 1, 2\}$; /* subsequence around index i */
    Flip(i, V_i) ; /* See Algorithm 3 */
    Swap$(i+1 \mod N, j+1 \mod N)$; /* See Algorithm 4 */
return $\mathcal{S}$

Where Flip and Swap are defined as follows:

Algorithm 3: Flip

Input: index i, subsequence V_i of vertices from $\mathcal{S}$
Result: (in place) optimised $\mathcal{S}$

$\ell_0 \leftarrow \text{length}(V_i) = \sum_{k=0}^{2} \mathbf{D}\left[V_i[k], V_i[k+1]\right]$; /* path length of subsequence V_i in G */
$V_i^\star \leftarrow [V_i[0], V_i[2], V_i[1], V_i[3]]$; /* subsequence with the middle vertices flipped */
$\ell^\star \leftarrow \text{length}(V_i^\star) = \sum_{k=0}^{2} \mathbf{D}\left[V_i^\star[k], V_i^\star[k+1]\right]$; /* path length of the flipped subsequence */
if $\ell^\star < \ell_0$ **then**
  $\mathcal{S}[i] \leftarrow V_i^\star[0]$; /* Perform the same flip on the original $\mathcal{S}$ */
  $\mathcal{S}[i+1 \mod N] \leftarrow V_i^\star[1]$

Algorithm 4: Swap

Input: indices $i, j; i \neq j$
Result: (in place) optimised $\mathcal{S}$

$\ell_{\mathcal{S}} \leftarrow \text{length}(\mathcal{S}) = \sum_{k=0}^{N} \mathbf{D}[\mathcal{S}[k], \mathcal{S}[k+1 \mod N]]$; /* original path length of entire $\mathcal{S}$ */
$\mathcal{T} \leftarrow \mathcal{S}$; /* trial sequence, starts as copy of $\mathcal{S}$ */
$\mathcal{T}[i] \leftarrow \mathcal{S}[j]$; /* swap the two vertices specified */
$\mathcal{T}[j] \leftarrow \mathcal{S}[i]$
$\ell_{\mathcal{T}} \leftarrow \text{length}(\mathcal{T}) = \sum_{k=0}^{N} \mathbf{D}[\mathcal{T}[k], \mathcal{T}[k+1 \mod N]]$; /* path length of entire $\mathcal{S}$ */
if $\ell_{\mathcal{T}} < \ell_{\mathcal{S}}$ **then**
  $\mathcal{S}[i] \leftarrow \mathcal{T}[i]$; /* Perform the same swap on the original $\mathcal{S}$ */
  $\mathcal{S}[j] \leftarrow \mathcal{T}[j]$

Note that in *Flip + Swap* (Algorithm 2), both *Flip* and *Swap* are done in each iteration of the inner loop, even though *Flip* only acts on the index i. This has proved to produce heuristically quite good results in testing.

3.1.3 Preparing the plan for execution

Before the plan is ready to be executed, the sequence $\mathcal{S}$ is rotated so that the initial location of the drone is the first element – this means navigation will start there (as it should, since that is where the drone will be at the start), and also the drone will attempt to return to this location after passing all sensors.

In this context, rotation means shifting the elements of $\mathcal{S}$ cyclically until the desired situation is achieved. More explicitly, if ρ denotes rotation by a single element and ρ^k by $k \in \mathbb{Z}^+$ elements:

$$\rho \cdot \mathcal{S} = \left[\mathcal{S}[1], \mathcal{S}[2], \ldots, \mathcal{S}[N-1], \mathcal{S}[0]\right]$$
$$\rho^k \cdot \mathcal{S} = \left[\mathcal{S}[k], \mathcal{S}[k+1], \ldots, \mathcal{S}[0], \ldots, \mathcal{S}[k-1]\right]$$

After the rotation, the abstract sequence $\mathcal{S}$ representing the order of sensors to visit is transformed into a list of **waypoints**: $\mathcal{W}$. This means vertices in $\mathcal{S}$ are translated into their corresponding points in the 2D Euclidean space. In this more concrete plan, the initial point is now exlicitly added at the end of $\mathcal{W}$ as well.

3.2 Executing the path plan

When executing the path plan $\mathcal{W}$ generated beforehand, the drone has to store its state variables (battery value, location) and keep them up to date. While flying, it generates the trajectory, which stores information about what moves were taken and what their potential sensor readings were.

In the previous section, the algorithms for planning are essentially solving a purely mathematical problem, which means expressing them in pseudocode with all the details is the best way to present them. However, the algorithm used to control the operation of the drone is a more complex one, and a more abstract description of the overall process is more suitable.

Algorithm 5: Execute Path Plan

Input: list of waypoints $\mathcal{W}$

1 Initialize state variables: *battery* ← MAX_BATTERY, drone's *location* ← $\mathcal{W}[0]$
2 Initialize *target* (point where the drone will try to navigate)
3 Initialize control variables: *evasionDirection* ← None; *needToReturn* ← false
4 *waypointIndex* ← 1 ; `/* the index of waypoint to navigate to */`
5 **while** *waypointIndex* < $|\mathcal{W}|$ **do**
6 *target* ← Choose Target(*needToReturn*) ; `/* see section 3.4 */`
7 **if** *drone is within sensor reading distance of target* **then**
 `/* the sensor at target (if any) was already read in the previous iteration */`
8 Increment *waypointIndex*
9 go to next iteration of **while** (line 5)
10 *targetAngle* ← angle between current *location* and *target*
11 *newLocation, moveAngle, evasionDirection* ← Compute Move (*location, targetAngle*) ; `/* which takes obstacles into account, see section 3.3 */`
12 Update drone state: decrement *battery*, *location* ← *nextLocation*
13 *sensor, needToReturn* ← Handle Nearby Sensors() ; `/* see section 3.4 */`
14 Append *Move(location, moveAngle, newLocation, sensor)* into the *trajectory*, and mark *sensor* as visited
15 **if** *battery* = 0 **then**
16 Stop iterating (break the **while**)
17 Return the *trajectory*

The Algorithm 5 describes the overall operation of the drone. It references multiple different sub-algorithms, which will be described in the following sections.

3.3 Computing where to go

Since there are no-fly zones, it is not entirely trivial to compute where the drone should go based on where it wants to go. The intention is to move in the direction specified *targetAngle*. First, this has to be aligned to the closest multiple of 10 degrees. Then, obstacles are checked to see if any would be crossed by a move in this direction.

The idea is that when the desired move intersects an obstacle, the *evasionDirection*= ± 1 is chosen (in **ObstacleEvader** (section 2.4)) and then the drone will rotate its intended angle (aligned to allowed angles) in steps of ± 10 degrees (where the $\pm$ is determined by *evasionDirection*) until a move in the direction of the new angle could happen. When found, this becomes *moveAngle*, and the *newLocation* is calculated, going from the drone's current *location* in the direction defined by *moveAngle*.

The value *evasionDirection* is stored for next iterations of the **while** loop, because this move is a part of obstacle evasion – i.e. the drone may have not cleared the obstacle just yet. This is to keep the evasion going in one direction, otherwise the drone would calculate a new *evasionDirection* after every, and if this were different, it would end up moving back and forth in the same place.

The calculation of *evasionDirection* is depends on the *targetAngle* and an angle φ that encodes the direction from drone's *location* towards the *central (average)* point of the obstacle being evaded:

$$evasionDirection = \begin{cases} -1 & if\, \varphi > targetAngle \\ +1 & \text{otherwise} \end{cases}$$

3.4 Handling Nearby Sensors

To find a sensor that needs to be read, the algorithm iterates over all sensors and prepares a list of those that are in range of drone's *location* and have NOT been marked as read yet.

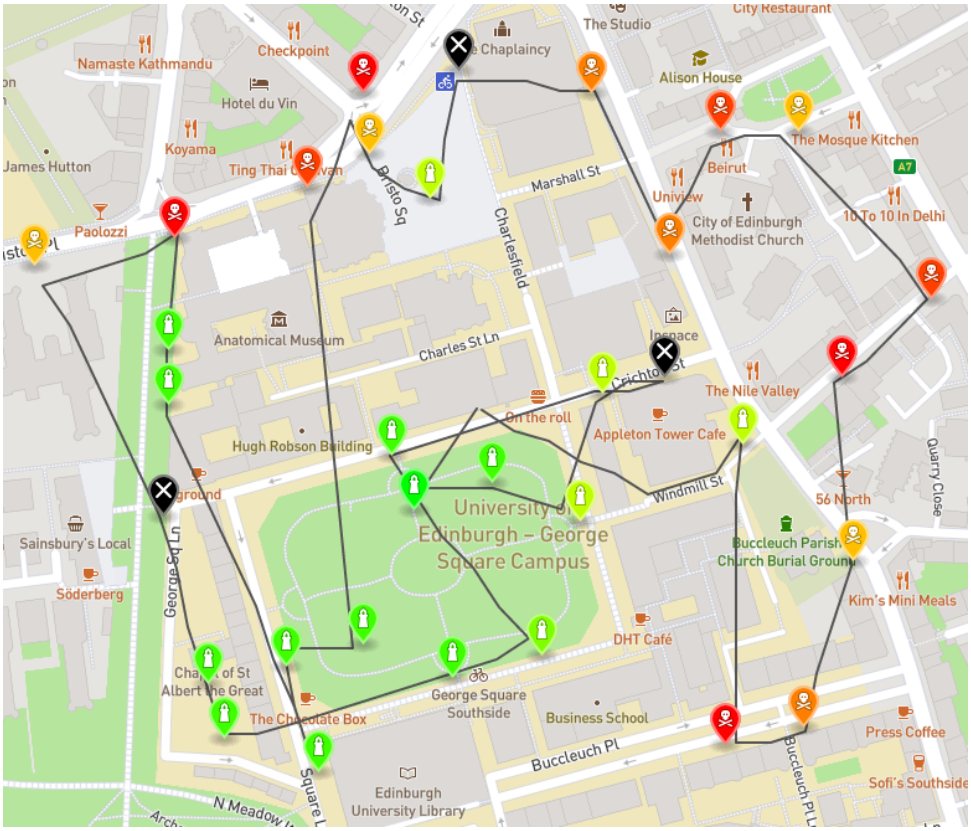
In the event that multiple sensors are found, one of them is read (so it will become the value *sensor* returned by this procedure) and the state variable *needToReturn* is set to *true*. This influences the choice of *target*: it tells the drone that it needs to take a step back towards to previous location in order to find that sensor again. If there are no more unread sensors, the value *needToReturn* is set back to *false*.

3.5 Example paths

Here I present some examples from trial runs of the drone simulation. These are screenshots taken from the maps generated using the *geojson.io* web page. The first example was generated for **01/01/2020**, starting from

location (55.944425, −3.188396). Screenshow is below:

The second picture is from a drone run on **05/05/2020**, starting again from the same location (55.944425, −3.188396). The screenshow is below:



References

[1] L. Weru, "11 animated algorithms for the traveling salesman problem," 2019. https://stemlounge.com/animated-algorithms-for-the-traveling-salesman-problem/, Accessed 2020-11-30.