Informatics Large Practical - Coursework 2

Section 1 - Software Architecture

1.1 - Main Class Summary

I will briefly summarise the roles and responsibilities of a few important classes that make up the main bulk of the program. I'll also give justification for why I decided to use them.

Waypoint
- an *abstract* class representing a place that the drone can travel to/towards
- has a position, and nothing else
- extended by both Sensor and StartEndPoint

Sensor
- a Waypoint representing a pollution sensor
- has a pollution reading, battery level, and a what-3-words address
- created from the pollution sensor data we read from the web server

StartEndPoint
- a Waypoint representing the start/end point of the flight
- unlike Sensor, stores no additional data

These classes are a useful way of representing places the drone needs to visit. The abstract class Waypoint lets us use polymorphism to pass either a Sensor or a StartEndPoint to any methods that only need position information.

Drone
- can move in in a direction specified by a bearing
- can read and return the pollution readings from sensors in its range
- records the number of moves it has made so far

You can see that the drone is very simple and has no knowledge of its surroundings. All it knows is where it is, and how much battery it has left (the number of moves made). Much like in real life, a drone needs a pilot to fly it (whether that be a human or a computer program).

Pilot
- is assigned a drone, and is given a list of no-fly-zones and a drone confinement area
- is responsible for flying its assigned drone to each sensor in the route provided, while avoiding the no-fly-zones and staying in the drone confinement area
- is also responsible for recording and remembering information about the drone's flight

Separating the Drone and Pilot classes like this help keep cohesion high. Although the Pilot class has only two main responsibilities, there is a lot of work involved in flying the drone to each sensor. For example, determining which moves are legal and which aren't, or planning a route around any no-fly-zones.

To prevent the Pilot class from becoming a monolith, it contains two static nested classes - NoFlyZoneChecker and SearchBranch.

NoFlyZoneChecker
- a static nested class that, given a list of no-fly-zones and a drone confinement area, can determine whether a move is legal or not
- can also determine whether a move lands inside a no-fly-zone

With this class, legality checking is abstracted. The Pilot doesn't have to remember anything about where the no-fly-zones are or where the drone confinement area covers. All it needs to do is query its instance of NoFlyZoneChecker and it gets a yes or no answer.

SearchBranch

- a static nested class that represents a search branch
- can plan a route around any no-fly-zones the pilot encounters

If the pilot instantiates these branches and calls their evaluate() method, it gets a legal route around any no-fly-zone in its way. This abstracts away the process of pathfinding.

These classes make up most of the main loop of the program. The remaining classes mostly do work before or after the main flight loop. We will have a look at these in the next section.

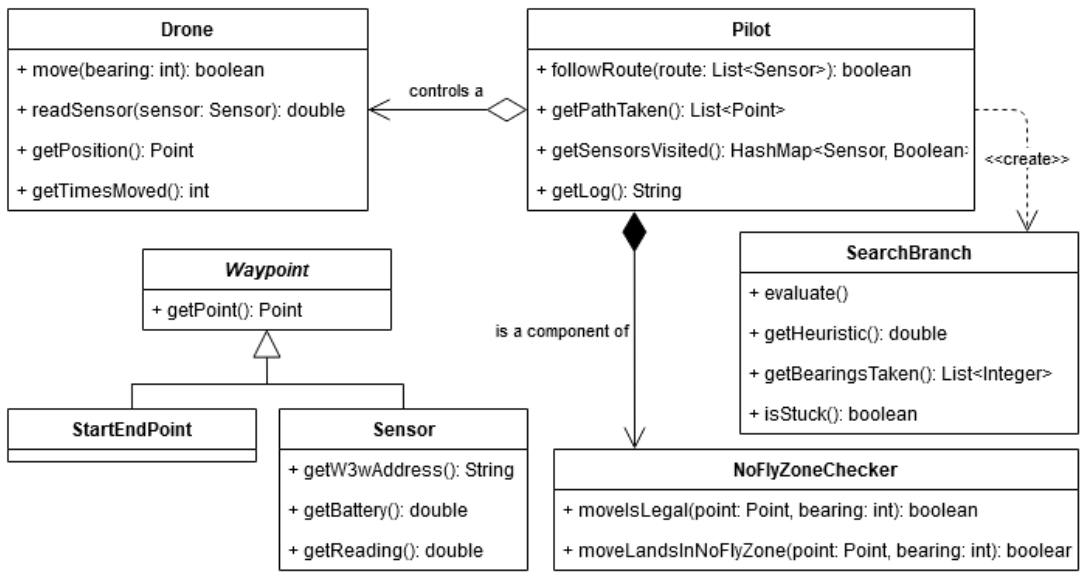


Figure 1.0: UML class diagram of the main classes

1.2 - Utility Classes

The following classes store no state information, have a private constructor, and all of their methods are static. Their main purpose is to group related code together.

FlightPlanner

- takes a starting point and a list of arbitrarily ordered sensors
- orders the list of sensors using the "greedy" travelling salesperson method
- runs the 2-opt path optimisation algorithm on the greedy sensor path

Optimising the route before giving it to the pilot means that it doesn't have to do any sort of long term planning. It also serves to reduce the length of the flight.

FlightMap

- takes a map of Sensors → Booleans denoting which sensors have been visited by the drone, and a list of positions that the drone has occupied throughout the flight
- returns a map of the flight as a FeatureCollection object

PointUtils

- contains several useful methods for manipulating com.mapbox.geojson.Point objects
- also contains helper methods such as inRange and mod360 that are used by several classes

1.3 - The WebServer Class

The WebServer class is responsible for pulling data down from the web server and packaging it in a useful way. Its two public methods getSensors and getNoFlyZones return a list of Sensors and a list of Polygons respectively.

It also has one extra piece of functionality where it will try to recover from failed HTTP requests by retrying the request up to a set number of times. I have only ever had this happen once, but better to be safe than sorry!

1.4 - Error Handling

The program will quit using System.exit(1) and print an error message if it ever finds itself in an illegal state. This happens when:

- data cannot be retrieved from the web server for whatever reason
- the drone attempts to read a sensor that it is too far away from
- there was a problem creating or writing to the output files

1.5 - Other Architecture Details

Use of java.util.Optional
If any methods need to return either a value or no value, they will return an Optional object. This is far safer than returning null values and forces me to explicitly handle both empty and non-empty returns. Unfortunately, the syntax for using this can be quite verbose and unintuitive, but I thought the added safety was worth it.

Taking “readings” from sensors
Because we have access to every sensor’s location, what-3-words address, battery level, and pollution reading ahead of time, I didn’t think it was worth pretending to not know each pollution reading and copying it to a separate array for use later. Instead, each sensor gets marked as visited or unvisited by the pilot. The HashMap of each sensor with its visited status then gets sent to the FlightMap class, who directly accesses the sensors’ readings, battery levels, positions etc. to create the readings-DD-MM-YYYY.geojson file.

The Drone class still has a readSensor method for the sake of completeness, but its return value is not saved by the pilot. It does serve some purpose though; it will exit the program if the drone is told to read an out of range sensor.

1.6 - Program Flow

The App class contains the main method. It handles the following:

- instantiation of the Pilot, Drone, and WebServer
- the flow of data in and out of them
- processing the retrieved data and writing it to file
- printing useful information about what’s going on to the screen

I did make a UML sequence diagram showing this in more detail, but I can’t manage to fit it in. A look at the App main method should give you a good idea of what’s going on though.

Section 2 - Class Documentation

[Waypoint and its children are not listed, but they are “struct-like” classes, so there’s not much to say about them]

Drone

Fields
private Point position
private int timesMoved = 0
public static final int MAX_MOVES = 150
public static final double MOVE_DISTANCE = 0.0003
public static final double SENSOR_READ_DISTANCE = 0.0002
public static final double END_POINT_DISTANCE = 0.0003 The drone must be less than END_POINT_DISTANCE degrees from the place it started from to successfully complete the flight.
Methods
public Drone(Point startPosition) Creates a drone with the specified start position.
public boolean move(int bearing) Moves the drone with the provided bearing if it is not out of moves. Returns true if the move was successful, false otherwise.
public double readSensor(Sensor sensor) Returns the pollution reading of the provided sensor if it is in range, exits the program with System.exit(1) if it is not.
public Point getPosition()
public int getTimesMoved()

Pilot

Fields
private final Drone drone The pilot’s assigned drone.
private final NoFlyZoneChecker noFlyZoneChecker Lets the Pilot check whether a move is legal or not before sending it to the drone.
private final Queue<Integer> precomputedBearings When we encounter an obstruction, we compute a path around it and fill this queue with the bearings the drone needs to take to follow the path.
private final HashMap<Sensor, Boolean> sensorsVisited We mark every sensor as unvisited (false) before we start flying the drone. Every time the drone visits a sensor, we update sensorsVisited to reflect that. Used to create readings-*.geojson later after the flight.
private final List<Point> pathTaken List of points where the drone has been. Used to create readings-*.geojson later after the flight.
private final List<String> log Holds each log line that we later write to flightpath-*.txt.
Methods
public Pilot(Drone drone, List<Polygon> noFlyZones, BoundingBox droneConfinementArea) Creates a pilot with an assigned drone, and with specified restrictions.
public boolean followRoute(List<Sensor> route) Tries to fly the drone to each sensor in the route (in order), returning to its initial position

afterwards.
Returns true if the flight was successful (made it back in 150 moves, visited every sensor, didn't get stuck), false otherwise.
Uses - navigateTo, takeReading

private boolean navigateTo(Waypoint waypoint)
Tries to fly the drone to the specified waypoint.
Returns true if the drone makes it to the waypoint without running out of moves or getting stuck, false otherwise.
Uses - nextBearing, Drone::move, logMove

private Optional<Integer> nextBearing(Waypoint waypoint)
Determines the most appropriate bearing to take next.
If the precomputedBearings queue is non-empty, we know the drone is in the process of following a path around an obstruction (it hasn't "used up" all of the bearings in the path around the obstruction yet).
Returns the bearing at the front of the precomputedBearings queue if it is non-empty. Else, returns the bearing directly towards the waypoint if it is legal. Else, computes a path around the obstruction and returns the first bearing of the path.
Uses - mostDirectBearing, computePathAroundObstruction

private List<Integer> computePathAroundObstruction(Waypoint waypoint)
Tries to compute a legal path for the drone to take to avoid an obstruction.
The provided waypoint helps direct the search.
Returns the path (as a list of bearings) if successful, or an empty list if no legal path could be found.
Uses - SearchBranch

public String getLog()
Returns the concatenation of all log strings in the log list.
Result is written to flightpath-*.txt.

public HashMap<Sensor, Boolean> getSensorsVisited()

public List<Point> getPathTaken()

private void takeReading(Sensor sensor)
Calls drone.readSensor(sensor) and marks the sensor as visited if in range.

private void logMove(Point previousPosition, int bearing, Point newPosition, String w3wAddress)
Creates a new log entry (String) and adds it to log.

private static Optional<Integer> mostDirectBearing(Point point, Waypoint waypoint, NoFlyZoneChecker noFlyZoneChecker)
Helper method that returns the bearing directly towards the waypoint if the resulting move would be legal.
If the bearing overshoots the waypoint and hits a no-fly-zone, tries to correct it and returns the result. If that fails, returns nothing.
This method is static so that it can be used by both the pilot and any instance of SearchBranch.

NoFlyZoneChecker

Fields

private BoundingBox droneConfinementArea

private HashMap<Polygon, Polygon> boundariesWithNoFlyZones = new HashMap<>()
Map with no-fly-zones as the values and their bounding boxes as the keys.

Methods

public NoFlyZoneChecker(List<Polygon> noFlyZones, BoundingBox droneConfinementArea)
Creates a NoFlyZoneChecker object that checks the legality of moves against the provided no-fly-zones and drone confinement area.

public boolean moveIsLegal(Point point, int bearing)
Returns true if the move starting at the specified point, moving in the direction of the specified bearing would be legal. False otherwise.
Uses - pointStrictlyInsideBoundingBox, lineIntersectsPolygon

public boolean moveLandsInNoFlyZone(Point point, int bearing)
Returns true if the move starting at the specified point, moving in the direction of the specified bearing terminates inside a no-fly-zone.
Uses - TurfJoins.inside

private static boolean pointStrictlyInsideBoundingBox(Point point, BoundingBox bound)

private static boolean lineIntersectsPolygon(Point start, Point end, Polygon poly)
Returns true if the line segment defined by the points start and end intersects with any of the line segments that make up poly.
Uses - vectorsOppositeSidesOfLine

private void calculateBoundaries(List<Polygon> noFlyZones)
Calculates the rectangle that fully encloses each noFlyZone. These are then stored in boundariesWithNoFlyZones, with the enclosing rectangle as the key, and the noFlyZone as the value. This information is used to reduce the number of no-fly-zones that have to be checked when checking if a move is legal.

private static Point toVector(Point start, Point end)
The points start and end define a line segment. This line segment is then moved so that start now lies on the origin. The point end can then represent a 2D vector from the origin.

private static double cross(Point vectorA, Point vectorB)
Returns the (Z component of) the cross product of vectorA and vectorB. Because both vectors lie on the xy-plane, the cross product points directly up or down on the Z-axis.

private static boolean vectorsOppositeSidesOfLine(Point vectorA, Point vectorB, Point lineVector)
Returns true if vectorA and vectorB are on opposite sides of the line defined by lineVector. The naming is a bit confusing here; lineVector defines a line, not a line segment. The gradient of lineVector is what's important here.
Uses - cross

SearchBranch

Fields

private Point branchHead
Current head of the search branch.

private List<Integer> bearingsTaken
Directions the branch has chosen at each expansion.

private final Waypoint goal
The target waypoint for the branch. Helps determine when to stop searching.

private NoFlyZoneChecker noFlyZoneChecker

private final boolean clockwise
True if the branch explores clockwise, false if anti-clockwise.

private boolean stuck
Branch gets marked as stuck if it cannot be expanded any further.

Methods

public SearchBranch(Point startPoint, Waypoint goal, boolean clockwise, NoFlyZoneChecker noFlyZoneChecker)

public void evaluate()
Repeatedly expands the branch until it finishes or gets stuck.

private void expand()
Tries to expand the search branch by one move (updating branchHead and bearingsTaken).
If clockwise == true, will expand the branch in a clockwise direction around the obstruction and vice versa. Marks the branch as stuck if the branch could not be expanded.
Uses - bearingScan

private Optional<Integer> bearingScan(int scanFrom, int scanTo, int step)
Checks the legality of the moves with bearings in the range scanFrom-scanTo (step is the interval). Returns the first bearing it finds that is legal, nothing if no legal bearing was found.

private boolean isFinished()
Returns true if the branch has found a legal path around the obstruction, false otherwise.

private int backtrackBearing()
Returns the bearing the branch last took - 180.

public double getHeuristic()
Returns the length of the branch + the euclidean distance to the goal.

public List<Integer> getBearingsTaken() Returns a list of bearings taken by the search branch.
public boolean isStuck() Returns whether the branch was marked as stuck when evaluating it.

FlightPlanner

Methods
public static List<Sensor> greedyPath(Point start, List<Sensor> sensors) Returns a path (ordered list of Sensors) generated using the greedy TSP algorithm. Uses - closestSensor
public static List<Sensor> twoOptPath(Point start, List<Sensor> sensors) Optimises and returns the provided path (ordered list of Sensors) using the 2-opt path optimisation algorithm. Uses - reversalImprovesPath, modifiedPath
private static boolean reversalImprovesPath(List<Waypoint> path, int i, int j) Returns true if reversing the sub-path i-j (inclusive) decreases the overall length of the path, false otherwise.
private static List<Waypoint> modifiedPath(List<Waypoint> path, int i, int j) Returns the provided path with the sub-path i-j (inclusive) reversed.
private static Sensor closestSensor(Point point, List<Sensor> sensors) Returns the sensor closest to point in the provided list of sensors.

WebServer

Fields
private static WebServer singletonInstance Stores the only existing instance of WebServer.
private final String serverURL Full url of the web server excluding the port.
private final String port The port that we connect to the web server on.
private final HttpClient client = HttpClient.newHttpClient() Object that is responsible for sending the HTTP requests.
private final int MAX_HTTP_REQUEST_ATTEMPTS = 10 Maximum number of times that an HTTP request will be retried.
Methods
public static WebServer getInstanceWithConfig(String serverURL, String port) Returns the singleton instance of WebServer and sets its serverURL and port fields.
public List<Polygon> getNoFlyZones() throws UnexpectedHTTPResponseException Returns a list of no-fly-zones as polygons defined in /buildings/no-fly-zones.geojson. Uses - getResourceAsString
public List<Sensor> getSensorData(String day, String month, String year) throws UnexpectedHTTPResponseException Returns a list of Sensor objects created from the contents of the relevant /maps/YYYY/MM/DD/air-quality-data.json file. Uses - getResourceAsString, getCoordinateFromWhat3WordsAddress
private Point getCoordinateFromWhat3WordsAddress(String w3wAddress) throws UnexpectedHTTPResponseException Returns the Point that the provided what-3-words address corresponds to. Uses - getResourceAsString
private String getResourceAsString(String pageUrl) throws UnexpectedHTTPResponseException Returns the contents of a specified file on the web server as a string. Throws an exception if the web server does not return an HTTP status code of 200 Will also retry the request if it is interrupted

FlightMap

Fields
private static final String[] HUES = {"#00ff00",...,"#ff0000"} Colours from green to red.
private static final String GREY = "#aaaaaa"
private static final String BLACK = "#000000"
Methods
public static FeatureCollection generateFromFlightData(List<Point> flightpath, HashMap<Sensor, Boolean> sensorsAndVisitedStatus) Returns the flight map as a FeatureCollection given the path of the drone and the map of Sensors and their corresponding reports. Uses - createMarkerFeatures
private static List<Feature> createMarkerFeatures(HashMap<Sensor, SensorReport> sensorReports) Returns a list of Features (with appropriate properties) defining each marker on the map. Uses - pollutionToColour
private static String pollutionToColour(double reading) Returns the marker colour corresponding to reading.

PointUtils

Methods
public static boolean inRange(Point point, Waypoint waypoint) Returns true if point is in range of waypoint. If waypoint is a Sensor, this range is defined by Drone.SENSOR_READ_DISTANCE. If it's a StartEndPoint, it's defined by Drone.END_POINT_DISTANCE. Uses - distanceBetween
public static double distanceBetween(Point pointA, Point pointB) Returns the euclidean distance between pointA and pointB.
public static int bearingFromTo(Point point, Waypoint waypoint) Returns the bearing of the line from point to waypoint rounded to the nearest 10. Uses - mod360
public static Point moveDestination(Point point, int bearing) Returns the point you would arrive at if moving Drone.MOVE_DISTANCE degrees from the specified point with the specified bearing.
public static int mod360(int bearing) Returns bearing mod 360. Java's modulo operator % actually produces the wrong answer! For whatever reason, -60 % 360 evaluates to -60, not 300. We instead use Math.floorMod

Section 3 - Drone Control Algorithm

3.1 - Path Optimisation

As mentioned previously, the 2-opt path optimisation algorithm is run on the list of sensors returned from the web server before the flight. The FlightPlanner class that does the path optimisation does not have any knowledge of the no-fly-zones. This means that certain configurations of sensors and no-fly-zones will yield a bad path, but with the no-fly-zones used in this project, there doesn't seem to be many problems.

3.2 - Main Drone Loop

My drone control algorithm is quite simple. I wasn't aiming for an extremely efficient algorithm when writing it, but I'm quite impressed with the performance given its simplicity. **Figure 3.0** below gives a high level overview.

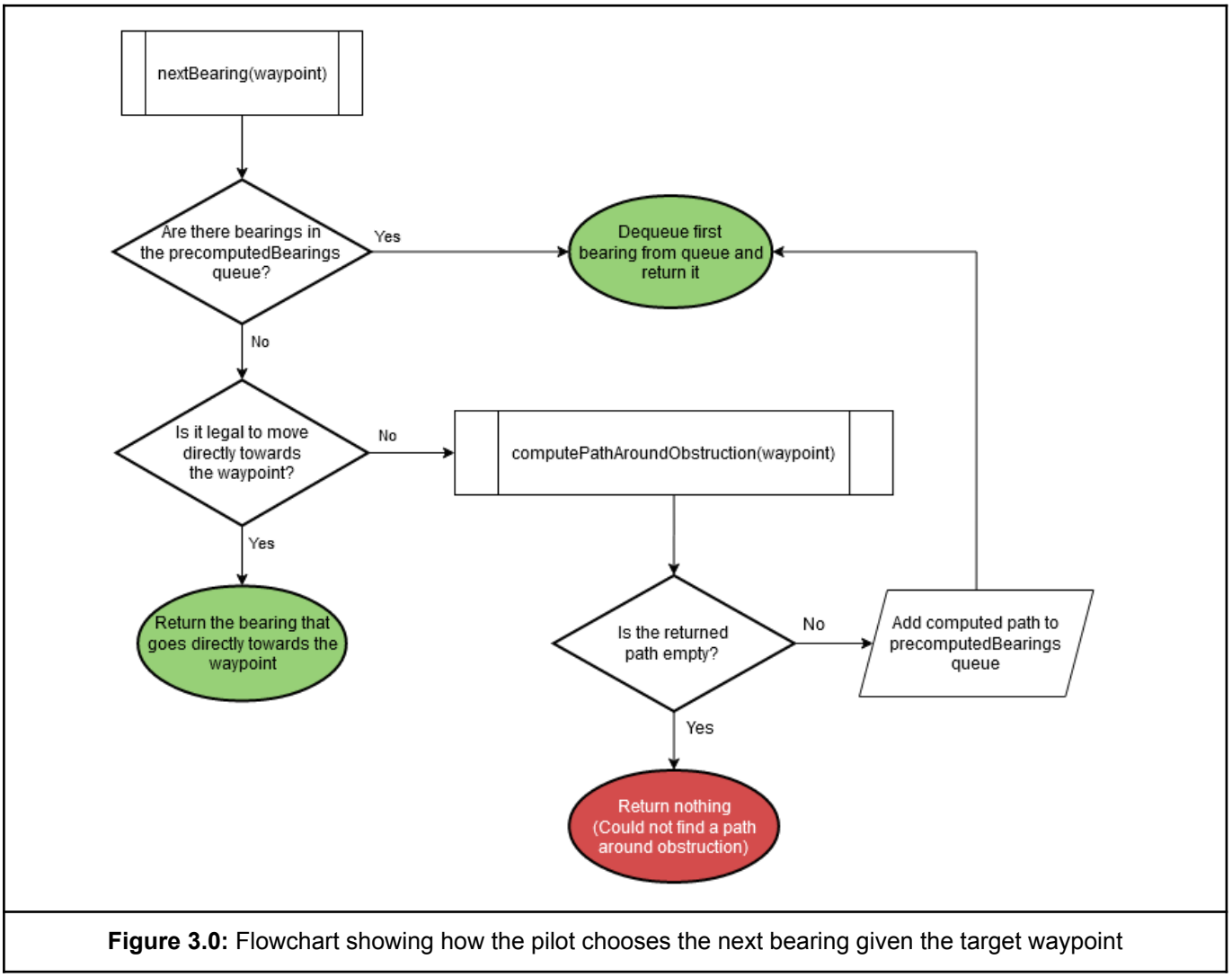


Figure 3.0: Flowchart showing how the pilot chooses the next bearing given the target waypoint

The following sections will go into more detail about each subprocess, but to summarise:

- The drone will go directly towards the target waypoint until it reaches it, or encounters an obstruction.
- If it encounters an obstruction, the pilot computes a route around it and follows that route until it is exhausted, at which point it will have cleared the obstruction.

3.3 - Determining the Most Direct Move

As you can see in **Figure 3.0**, we check the legality of the move directly towards the target waypoint (to the nearest 10 degrees) and take it if it's legal. We also check for the edge case where the pilot "overshoots" the waypoint.

This can happen when the drone is *just* out of range of a waypoint that is situated near the edge of a no-fly-zone. Examples of this are shown below (**Figure 3.1 / 3.2**). We can try to correct this by checking the legality of the adjacent moves (clockwise and anti-clockwise).

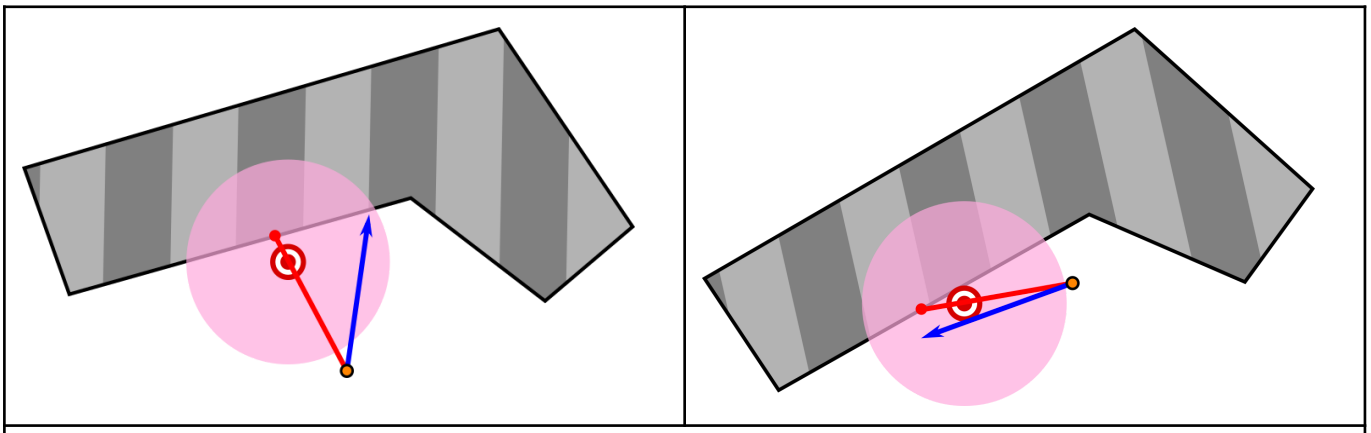


Figure 3.1 / 3.2:
Illegal moves (**red**) being corrected (**blue**).
The range of the sensor is shown in **pink**.

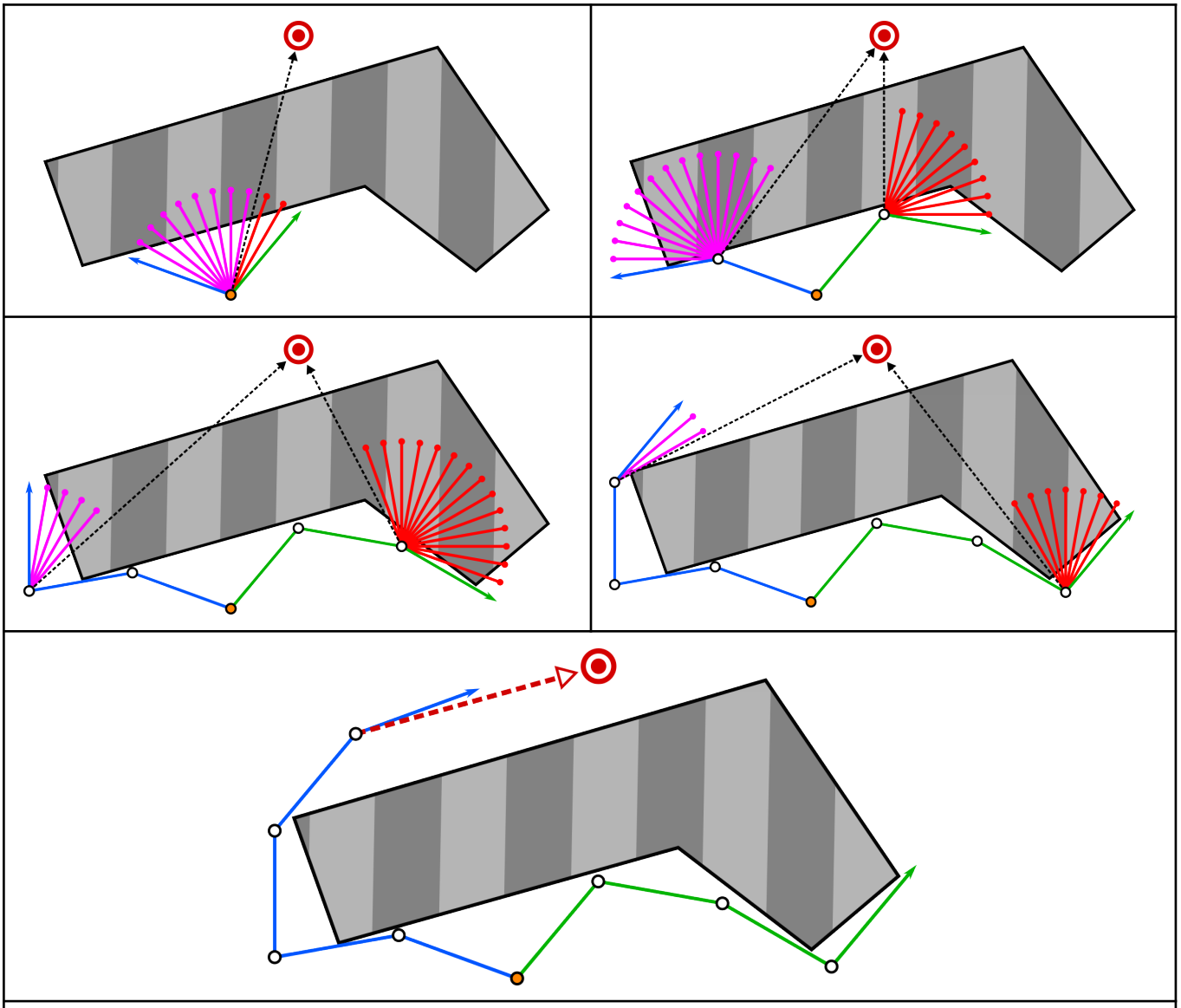
We catch this edge case by checking if a move lands in range of the target waypoint, but also lands *inside* a no-fly-zone.

If the move directly towards the target waypoint is illegal and can't be corrected using the above method, we have encountered an obstruction and need to find a path around it.

Terminology note
From now on, if I mention the "most direct bearing", I mean the bearing that points directly towards the target waypoint. If I say "most direct move", I mean the move that goes directly towards the target waypoint, or if that overshoots, the corrected move (hence the name *most* direct move).

3.4 - Searching for a Legal Path Around a No-fly-zone

When the pilot encounters a no-fly-zone, it creates two SearchBranch objects; one that explores clockwise around the no-fly-zone and another that explores anti-clockwise. The branches stay as tight to the no-fly-zone as possible as they expand. See **Figures 3.3-3.7** below for a visual explanation of how the branches expand.



The most direct move is legal. The branch is finished searching.

Figures 3.3-3.7 (left to right, top to bottom):
Shows the expansion of the **clockwise** and **anti-clockwise** branches, with the start shown in **orange**.
The dashed line shows the line (bearing) from the branch's head to the target waypoint.
Illegal moves (**pink** and **red**) terminate with a circle.
Legal moves terminate with an arrowhead.

Note - the branches do not expand simultaneously as shown here. The clockwise branch is fully explored and then the anti-clockwise branch is fully explored. We pick the one with the shorter heuristic (length of branch + euclidean distance from branch head to waypoint).

The branches are expanded by “scanning” either clockwise or anti-clockwise from the most direct bearing until a legal move is found. The branch then updates its head using this legal move. If the scan doesn’t find a legal move before the one that makes the branch backtrack, the search terminates and the branch is marked as stuck. This should only happen if the branch explores down a tight concave path and cannot turn back without backtracking.

3.5 - Concluding the Search

After every expansion of a branch, we check whether the branch has “finished”.
We consider the branch finished and stop the search if either:

- the most direct move from branch’s head is legal (it has “cleared” the no-fly-zone, shown in **Figure 3.7**)
- the branch’s head is in range of the target waypoint (the drone is range, but doesn’t have line of sight)

The first rule has one exception; because the most direct bearing is rounded to the nearest 10, the event shown in **Figure 3.8** can sometimes occur, described below:

The branch expands and then checks whether the direct move is legal. Internally, we get the bearing from the branch’s head to the target waypoint (97.15°, shown in **blue**) and round it up to 100°. It tests whether the direct move with bearing 100° is legal, and finds it is not.
Since the direct move is illegal, the branch has not cleared the no-fly-zone and proceeds to expand again.

This time, the bearing towards the branch is 94.60°, and gets rounded *down* to 90°. To the SearchBranch, it seems it has cleared the no-fly-zone as the most direct move is legal, but this is not the case.
We can see that the branch is just backtracking to where it last was.
We already established that the direct move from this point was not legal.

For this reason, a branch is only considered finished if the most direct move doesn’t make the branch backtrack.

There is one exception to this exception.
This very rarely happens, but in the following circumstances (shown in **Figure 3.9**) we allow the search to conclude by backtracking:

1. The drone starts in range of a sensor.
2. It tries going towards it and overshoots, hitting a no-fly-zone.
3. The pilot fails to correct the overshoot and starts computing a path around the no-fly-zone.
4. The SearchBranch scans for legal moves and finds the first legal move (**green**), which happens to travel in the opposite direction of the most direct move.

In this particular case, we allow the branch to finish by heading directly backwards.

I noticed this when my drone started in the corner between the library’s no-fly-zone and the drone confinement area, while also being in range and just to the left of the sensor shown in the 09/04/2020 flight map figure. Without this check, the branch would just continuously explore west along the confinement area’s edge until it hit the south-west corner, at which point it explored north and then returned to the waypoint as the most direct bearing changed.

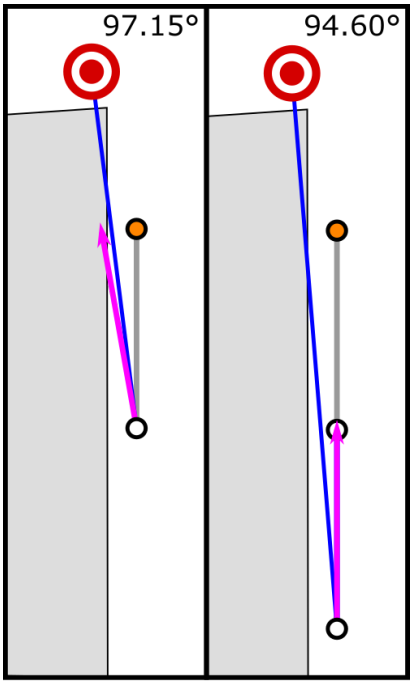


Figure 3.8:
Problem that can occur when rounding the most direct bearing to the nearest 10

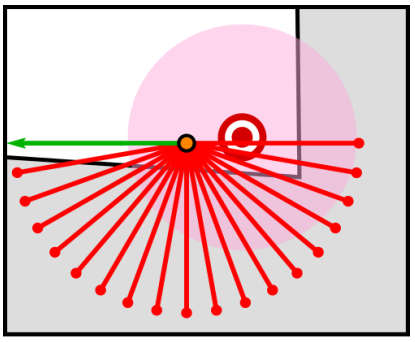


Figure 3.9:
Problem that can occur when the drone starts in range of a waypoint and overshoots it

3.6 - No-fly-zone Checking

The legality of every move is checked with a NoFlyZone object. When we check a move, we test to see if the line segment defined by the start and end points of the move intersects with any of the line segments that make up the no-fly-zones.

So that we don't have to check every single no-fly-zone line segment for every single move, we only check for collisions with no-fly-zones where:

- the start or end point of the move are inside the no-fly-zone's bounding box
- the line segment defined by the move intersects with the edges of the no-fly-zone's bounding box (but doesn't enter or leave it)

The actual intersection check is done using the cross product. The cross product is helpful here because for two vectors $(\overline{u},\ \overline{v}),\ \ \overline{u} \times \overline{v}$ will have a positive magnitude if $\overline{u}$ is on the right of $\overline{v}$, and a negative magnitude if $\overline{u}$ is on the left of $\overline{v}$.

It should be noted that the cross product is not defined in two dimensions. We are technically solving for the z-component of the cross product. The longitudes are on the x-axis and the latitudes are on the y-axis.

By using this useful property of the cross product, we can easily check if two lines intersect.

Figure 3.10 below shows an example. $\overline{PQ}$ is a vector defined by one line segment of a no-fly-zone. $\overline{SE}$ is a vector defined by the start and end points of a move.

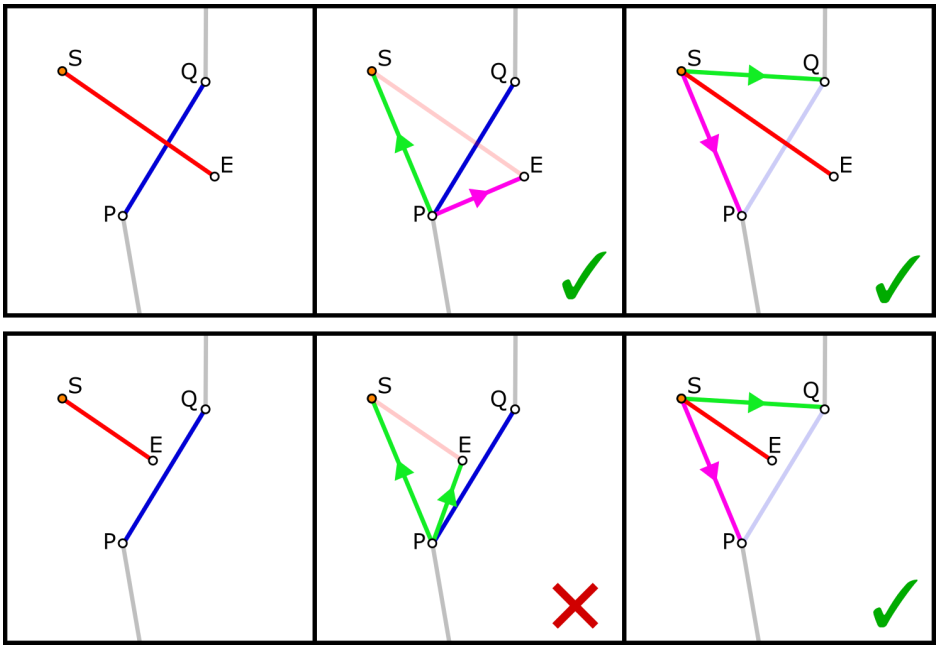


Figure 3.10:
Explanatory diagram showing what checks pass/fail when line segments do/do not intersect.
Vectors on the left of the relevant lines are **green**, and vectors on the right are **pink**.
A check only passes when both vectors are on different sides of the relevant line.

Hopefully you get the idea from **Figure 3.10**, but if not, see **bibliography entry 2** for a helpful web page with more information on checking line segment intersection. The "bounding boxes" I refer to have no relation to the line segment bounding boxes Martin Thoma mentions on his blog page. I don't use these in my implementation.

3.7 - Flight Maps

04/04/2020 (90/150 moves)

09/04/2020 (88/150 moves)

Bibliography

1. 3Blue1Brown (Grant Sanderson)
   “Cross products | Essence of linear algebra, Chapter 10”
   YouTube, 1st September 2016
   Accessed around 10th October
   https://youtu.be/eu6i7WJeinw

2. Martin Thoma
   “How to check if two line segments intersect”
   martin-thoma.com, 21st February 2013
   Accessed around 10th October
   https://martin-thoma.com/how-to-check-if-two-line-segments-intersect/

3. Mapbox Java SDK
   Accessed around 20th September
   https://docs.mapbox.com/android/java/overview/

4. Wikipedia contributors
   “2-opt”
   Wikipedia, 2nd September 2020
   Accessed around 10th October
   https://en.wikipedia.org/w/index.php?title=2-opt&oldid=976312461

5. Unknown author
   “2-opt move”
   TSP Basics blog, 3rd March 2017
   Accessed around 30th September
   http://tsp-basics.blogspot.com/2017/03/2-opt-move.html

6. Unknown author
   “2-opt: basic algorithm”
   TSP Basics blog, 4th March 2017
   Accessed around 30th September
   http://tsp-basics.blogspot.com/2017/03/2-opt-basic-iterative-algorithm.html