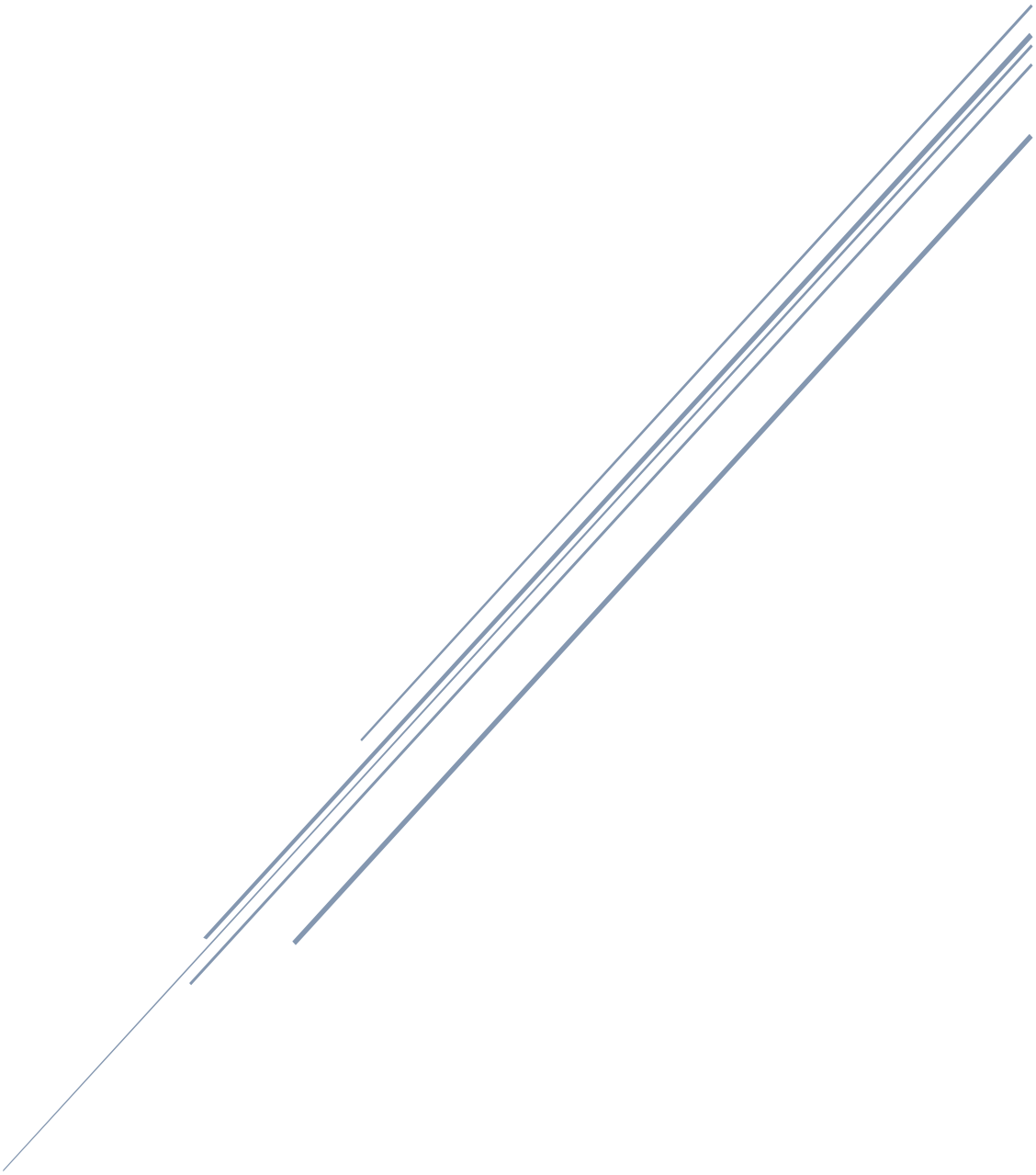
INFORMATICS LARGE PRACTICAL PROJECT REPORT

Guanwen WANG

S1824891



School of Informatics, University of Edinburgh

Informatics Large Practical

Table of Contents

1. Software architecture description

This application is to implement an autonomous drone which will collect readings from air quality sensors distributed around an urban geographical area as part of a (fictitious) research project to analyse urban air quality.[1] Therefore, to construct a coherent structure for our application we need classes which for each with separate functionality, obligation and objective. There are nine classes in the package for our application. The purpose and structure of the nine classes will be demonstrated in this section.

1.1 App

This class acts as a controller for the whole application. It contains the main method which is the entry for our programme. This class handles the input arguments, runs the simulation and outputs files.

1.2 Play

This class has several objectives to achieve. It downloads the data of all sensors and the data of four no-fly buildings from the webserver for our application. This class achieves to connect all sensors that the drone read in the flight path by lines for a specified date and sets the colour and symbol of sensors for a specified reading. This class also converts a list of features into a feature collection. In addition, this class involves a method that converts the flight path from a list of arrays of strings into a string. Lastly, this class has methods to write the text and geo-json files as required.

1.3 Drone

The fundamental to our application is the autonomous drone which finds the closest sensor and moves to that sensor in the drone confinement area. It is achieved in the Drone class. This class provides methods for the drone to locate where the closest sensor is and to move to where the sensor is. The class also checks if the number of moves to get to all the sensors around the confinement area exceeds the maximum limit and checks if the drone is close enough to receive readings from sensor and to move back to the original location.

1.4 Sensor

This class is created for selecting the right colour and symbol in the form of strings for a given input of reading and battery. Therefore, the features from the geo-json could be easily accessed.

1.5 FindPosition

This class is where the Greedy algorithm is applied. It contains the calculation of the Euclidean distance between two points, the destination points for the drone to move and the angle between the current point of the drone and the potential point of the drone to move. There are three approaches for acquiring the angle for which our application will

decide which of the approaches will be applied based on which has the smallest number of moves.

1.6 FlightPath

This class consists of the process for our application to make a decision on which of the approaches will be applied by comparing the number of moves for each approach. This class also involves three methods that return the flight path corresponded to the three approaches respectively. Besides, and lastly, this class has a method to get the flight path from the last point to the original start point.

1.7 NoFlyZone

This class treats the no-fly buildings as four polygons and gets the coordinates of the points of the polygons. There are several Boolean functions involved in this class to check if the drone flies into the no-fly buildings.

1.8 Properties

This class contains several static final variables for our application to use. The variables will not be updated during the algorithm runs.

1.9 W3W

This class simply represents a What3Word.

2. Class documentation

Remark: Getters and setters are not mentioned in this section.

2.1 App

The *App* class contains the main method which is the entry for our application. It deals with the input arguments and invokes functions from other classes to output the txt and geo-json file.

Fields summary
public static String air_quality_data_jsonURL A URL string for downloading data from the webserver with specified date, month, year and port. It is updated in the main method.
public static String no_fly_zone_jsonURL A URL string for downloading data from the webserver with the specified port. It is updated in the main method.

Methods summary
public static void main(String[] args) Reads and parses the input arguments to update the two URL strings and calls functions from other classes to produce a geo-json file that shows the flight path on a map and a text file which consists of the flight path in strings.

2.2 Play

The *Play* class represents the play site for our application. It consists of functions to retrieve sensor information from the webserver and stores the information into a 2D array called "database". It also provides methods to acquire information of the no-fly buildings information from the webserver. Besides, this class contains several methods that interact with the Geo-JSON document to plot the flight path of the drone and output the path into a text and geo-json file.

Fields summary
private String day A string that represents the day of this play site.
private String month A string that represents the month of this play site.
private String year A string that represents the year of this play site.
private String latitude A string that represents the latitude of the starting point of this play site.
private String longitude A string that represents the longitude of the starting point of this play site.
private String seed A string that represents random seed number of this play site.
public final String[][] database A 2D array of strings that consists of information of all the sensors. It has size 33 x 5. Each row represents a sensor, whereas each column represents the location of the sensor, the latitude of the sensor, the longitude of the sensor, the reading of the sensor and the battery of the sensor.

Constructor summary
public Play (String day, String month, String year, String latitude, String longitude, String seed, String port) Initialises the object with a given date, latitude, longitude, seed number and port number.

Methods summary

public String[][]getDatabase(String day, String month, String year, String latitude, String longitude, String seed, String port) Returns a 2D arrays of strings which stores all the data of 33 sensors with a given input data, latitude and longitude of the starting point, seed number and port number.
public String getNoFlyZone() Returns a Json string that contains information of the no-fly buildings.
public List<Feature> getFeatures(List<Point> linePoints, String[][] database) Return a list of features that stores the features of all sensors.
public FeatureCollection getFeatureCollection(List<Point> linePoints, String[][] database, List<Feature> features) Returns the feature collection of the features on the map in order to output a Geo-JSON file.
public List<Point> getLinePoints(String day, String month, String year, String latitude, String longitude, String seed, String port, List<String[]> finalFlightPath) Returns a list of point in the drone's flight path with a given date, latitude and longitude of the starting point, seed number and port number.
public String convertStr(List<String[]> paths) Returns a string converted from the input list of arrays of strings.
public void toJson(FeatureCollection fc, String day, String month, String year) Outputs the Geo-JSON file with the given input feature collection and date.
public void wirteTxtFile(String str, String day, String month, String year) Outputs the text file with the given input date.

2.3 Drone

The *Drone* class represents a drone. It contains methods that moves the drone to a certain point and locates where the closest sensor is.

Fields summary
private double longitude A double represents the longitude of the drone's current location.
private double latitude A double represents the latitude of the drone's current location.

Constructor summary
public Drone(double longitude, double latitude) Initialises the object with a given longitude and latitude.

Methods summary
public Point Move(double drone_lng, double drone_lat, int angle) Returns a Point which is the location after the drone moves with a given longitude, latitude and an angle.
public static boolean isEnd(int moves)

Returns a boolean to check if the drone's flight path has exceeded 150 moves.
public String getClosestSensor(double drone_lng, double drone_lat, List<String> W3WList, String[][] database) Return a W3W string that refers to the closest sensor in the database.
public static int getRow(String[][] database, String w3w) Returns an integer which is a row in the database. Specifically, a row represents a sensor in the database.
public String closeBy(double drone_lng, double drone_lat, List<String> W3WList, String[][] database) Returns a W3W string that refers to the closest sensor in the database if the drone's location is close enough (< 0.0002) to this sensor, returns "Sensor not in range" otherwise.
public boolean checkRange(double drone_lng, double drone_lat, double start_lng, double start_lat) Returns a boolean to check if the drone's location is close enough (< 0.0002) to move to the starting point. True for yes, false for no.

2.4 Sensor

The *sensor* class represent a sensor and contains some information of a sensor.

Fields summary
private String location A string represents the location of a sensor in the form of What3Words.
private String battery A string represents the battery of a sensor.
private String reading A string represents the air quality reading of a sensor.

Constructor summary
public Sensor(String location, String battery, String reading) Initialises the object with a given location, battery and reading. The properties are in String because location is a What3Words string and the rest of two parameters are stored as strings as well in the sensor database.

Methods summary
public String getColor(String reading, String battery) Returns a rgb-string which corresponds to the interval of which the air-quality reading is in. Returns rgb-strings for low battery or not visited as well.
getSymbol(String reading, String battery) Returns a symbol in string which corresponds to the interval of which the air-quality reading is in. Returns symbol strings for low battery or not visited as well.

2.5 FindPosition

The *FindPosition* class is a class only containing static functions in which the main algorithm of our application is involved. Methods that return the next location of the drone with a certain angle are contained in this class. There are three methods to obtain the angle in which the drone would fly. The approach that returns the best angle which causes the shortest number of moves in the drone's flight path will be applied for our application.

Fields summary
public static List<Point> path A list of points that stores the point visited in the drone's flight path.

Methods summary
public static double euclideanDistance(double lng0, double lat0, double lng1, double lat1) Returns the Euclidean Distance between two points.
public static Point newPosition(double drone_lng, double drone_lat, int angle) Returns a new point for the drone from the current point with an angle.
public static int getOneAngle(double drone_lng, double drone_lat, double sensor_lng, double sensor_lat) Returns one angle between the drone and a sensor. The angle returned is a multiple of ten degrees.
public static int getAngleAntiClockwise(double drone_lng, double drone_lat, double sensor_lng, double sensor_lat) Returns an angle between the drone and a sensor. The angle will be added 10 degrees each time the drone touches the edges of the no-fly zones.
public static int getAngleClockwise(double drone_lng, double drone_lat, double sensor_lng, double sensor_lat) Returns an angle between the drone and a sensor. The angle will be deducted 10 degrees each time the drone touches the edges of the no-fly zones.
public static int getMinAngle(double drone_lng, double drone_lat, double sensor_lng, double sensor_lat) Returns an angle which results the least distance between the drone and a sensor in all 36 directions.
public static List<Integer> getAngleList(double drone_lng, double drone_lat, double sensor_lng, double sensor_lat) Returns a list of all possible angles between the drone and a sensor in all 36 directions.
public static boolean checkInPath(double lng, double lat) Returns a boolean that checks if the flight path contains a point that has been reached by the drone. True for it contains the point, false for not.
public static boolean inArea(double longitude, double latitude) Returns a boolean that checks whether this point reached by the drone is in the drone confinement area. True for in the area, false for not in.

2.6 FlighPath

The *FlightPath* class conducts comparison of the number of moves returned by the three approaches and decides which of them to apply.

Fields summary
public List<String[]> finalFlightPath A list of arrays of strings that stores the points visited in the drone's flight path. It contains each move of the drone in terms of the longitude and latitude of the drone before the move, the direction it chose to move, the longitude and latitude of the drone after the move, and the location of any sensor that is connected to after the move, or null otherwise. [1]

Constructor summary
public FlightPath(String date, String month, String year, String latitude, String longitude, String seed, String port) Initialises the variable finalFlightPath. It selects the path with the lease number of moves from three generated paths.

Methods summary
public List<String[]>getFlightPath(String day, String month, String year, String latitude, String longitude, String seed, String port) Returns the drone's path as a list of arrays of strings based on the method getMinAngle in the FindPosition class.
public List<String[]> getFlightPathAntiClockwise(String day, String month, String year, String latitude, String longitude, String seed, String port) Returns the drone's path as a list of arrays of strings based on the method getAngleAntiClockwise in the FindPosition class.
public List<String[]>getFlightPathClockwise(String day, String month, String year, String latitude, String longitude, String seed, String port) Returns the drone's path as a list of arrays of strings based on the method getAngleClockwise in the FindPosition class.
public List<String[]>getFinalPath(List<String[]> formerPath, double lng, double lat, double start_lng, double start_lat) Returns the drone's final flight path concatenated by FlightPath and the path for the drone to return to where it starts flying.

2.7 NoFlyZone

The *NoFlyZone* class obtains no-fly buildings and treats them as four polygons. There are several Boolean functions that check whether the drone's next location is inside the no-fly buildings.

Fields summary
private String day A string that represents the day of the specified play site.
private String month A string that represents the month of the specified play site.
private String year A string that represents the year of the specified play site.
private String latitude A string that represents the latitude of the starting point of the specified play site.
private String longitude A string that represents the longitude of the starting point of the specified play site.
private String seed A string that represents random seed number of the specified play site.
private String port A string that represents port number of the specified play site.
private static double[] poly0lats An array of doubles that stores the latitudes of the points that form the first polygon.
private static double[] poly0lngs An array of doubles that stores the longitude of the points that form the first polygon.
private static double[] poly1lats An array of doubles that stores the latitudes of the points that form the second polygon.
private static double[] poly1lngs An array of doubles that stores the longitudes of the points that form the second polygon.
private static double[] poly2lats An array of doubles that stores the latitudes of the points that form the third polygon.
private static double[] poly2lngs An array of doubles that stores the longitudes of the points that form the third polygon.
private static double[] poly3lats An array of doubles that stores the latitudes of the points that form the fourth polygon.
private static double[] poly3lngs An array of doubles that stores the longitudes of the points that form the fourth polygon.
Play play A play site with specified date, latitude, longitude, seed number and port number.

Constructor summary
public NoFlyZone() Initialises the variables poly0lats, poly0lngs, poly1lats, poly1lngs, poly2lats, poly2lngs, poly3lngs, poly3lats.

Methods summary
public boolean checkIntersection(double px1, double py1, double px2, double py2, double px3, double py3, double px4, double py4)

Returns a boolean that checks whether two line segments formed by the eight input coordinates intersect. True for intersected, false for not intersected. Remark: this method is acknowledged as coming from the answers posted on StackOverflow [2].
checkFlyZone(double start_lng, double start_lat, double lng, double lat, double[] poly_lng, double[] poly_lat) Returns a boolean that checks if the new point reached by the drone is inside the no-fly zones. True for not in the zone, false for in the zone.
public boolean check0(double start_lng, double start_lat, double lng, double lat) Returns a boolean that checks if the new point is in the no-fly zones for the first building. True for not in the zone, false for in the zone.
public boolean check1(double start_lng, double start_lat, double lng, double lat) Returns a boolean that checks if the new point is in the no-fly zones for the second building. True for not in the zone, false for in the zone.
public boolean check2(double start_lng, double start_lat, double lng, double lat) Returns a boolean that checks if the new point is in the no-fly zones for the third building. True for not in the zone, false for in the zone.
public boolean check3(double start_lng, double start_lat, double lng, double lat) Returns a boolean that checks if the new point is in the no-fly zones for the fourth building. True for not in the zone, false for in the zone.
public boolean checkFlyCondition(double drone_lng, double drone_lat, double new_lng, double new_lat) Return a boolean that checks if the drone is under the fly condition. True for not in the zone and within the confinement area, false for in the no-fly zone and not in the confinement area.
public boolean checkTouchedNoFlyZone(double drone_lng, double drone_lat, double new_lng, double new_lat) Returns a boolean that checks if the drone touches the margin of the four no-fly zones. True for touched the margin, false for not touched.

2.8 Properties

The *Properties* class contains static final variables for our application to use. The variables will not be updated during the application runs.

Fields summary
private static final double BOTTOM A double that represents the bottom latitude of the drone confinement area.
private static final double LEFT A double that represents the left longitude of the drone confinement area.
private static final double RIGHT A double that represents the right longitude of the drone confinement area.
private static final double MAX_DISTANCE

A double that represents the maximum distance for a drone to receive readings from a sensor.
private static final doublw EACH_MOVE_DISTANCE A double that represents the distance for a drone to move each time.
private static final int MAX_MOVES A double that represents the maximum number of moves in the flight path for a drone.
private static final int NUM_SENSORS A double that represents the total number of sensors to be read each day.
private static final int MAX_DEGREE A double that represents the theoretical maximum degree of an angle for a drone to fly.

2.9 W3W

The *W3W* class contains multiple sub classes to give the What3Words information corresponding to a What3Words address. The most important field in this record for our purposes is the coordinates field which allows us to associate a What3Words ad- dress with a specific longitude ("lng") and longitude ("lat"). In this project, longitudes will always be negative (close to −3) and latitudes will always be positive (close to 56) because Edinburgh is located at 3 degrees West and 56 degrees North. [1]

Fields summary
String country A string that represents which country the What3Words address in.
public static class Square A sub class from class W3W that includes two sub classes “Southwest” and “Northeast”. The two sub classes have the same fields.
public static class Southwest A sub class from class Square that has two fields.
public static class Northeast A sub class from class Square that has two fields.
double lng A double that represents the longitude of the specified address.
double lat A double that represents the latitude of the specified address.
String nearestPlace A string that represents the nearest place to the given coordinates.
public static class Coordinates A sub class from class W3W that has two fields.
double lng A double that represents the longitude of the coordinates.
double lat A double that represents the latitude of the coordinates.
String words A string that represents the specified address.
String language

A string that represents the which language is used to describe the specified address.
String map A string that represents the map of the specified address. The format of map is https://w3w.co/words.

3. Drone control algorithm

The main objective for our application is to program a drone to fly and collect the sensor readings of air quality. The drone will fly within a specified confinement region and avoid four no-fly buildings in its flight path. There will be 33 sensors for a drone to read on a given date. The drone will start searching the closest sensor from a location around the middle of the confinement area and return to this location after the drone reaches all of the 33 sensors. The drone can only retrieve the readings from sensors provided that it is within 0.0002 degrees in distance of the air quality sensor. The drone's flight path cannot exceed 150 moves and each move must be exactly 0.0003 degrees in distance. The drone can only fly in an angle which is a multiple of 10 degrees. There are three approaches to obtain the best and appropriate angle between the drone and the sensor in our application. The application applies a Greedy algorithm to find the nearest sensor for the drone. The Greedy algorithm is slightly modified to avoid passing through the no-fly zones for the drone. The strategy of our application will be illustrated in this section.

3.1 Description of drone strategy

The distance metric for the drone to find the distance to the closest sensor is Euclidean distance.
The Greedy algorithm is then applied which leads the drone to move from its current location to a location that is within the range of the sensor.
After locating the nearest sensor, the application determines an angle for the drone to fly at to avoid flying through the no-fly buildings.
After the route and the angle is determined, the drone is executed to move. The number of moves is kept up to date. The drone stops execution if the number of moves in the drone's flight path exceeds the specified maximum number of moves (150).
The drone continues the process above until all the sensors are reached by the drone in its flight path.
Then, the drone returns where it starts after the it connects all the sensors.

3.2 Description of angle strategy

As aforementioned, there are three strategies on finding the direction for a drone to move in our application. There are the obstacles in the drone confinement area and the drone is not allowed to fly in an arbitrary direction, thus, applying a single method would cause loss of efficiency for our application. The fewer the number of moves required to visit all the sensors, the better the quality of the drone. Therefore, applying three approaches of

obtaining the angle and then comparing the number of moves in each of the flight path would result in the smallest possible number of moves for our application.

The first approach is called *getMinAngle*. It follows the concept of the Greedy algorithm. It checks all of the 36 possible directions and acquires the angle which causes the least distance between the drone's current location and a sensor.
The second approach is called *getAngleAntiClockwise*. Its first step is to find the angle between the drone and a sensor. Nonetheless, every time the drone encounters the no-fly buildings, the angle increases 10 degrees to update the direction until there is no no-fly zones in front of the drone.
The third approach is called *getAngleClockwise*. It's similar to the second approach but with one difference. Every time the drone encounters the no-fly buildings, the angle reduces 10 degrees instead to update the direction until there is no no-fly zones in front of the drone.

The reason why three methods are applied instead of one is that for the first approach, it probably has the best algorithm to get the least number of moves in the drone's flight path while there are no obstacles. But the drone would sometimes fly within a certain area just beside the edge of a no-fly zone using the first approach. One instance is the output flight path on 02/02/2020, which is displayed in Figure 3.2.1 in the following. In contrast, Figure 3.2.2 displays the flight path of the drone produced using the third approach. It presents a clear path that connects all the sensors and returns the starting location.


Figures 3.2.1, 3.2.2: The drone's flight path maps from the first approach (left) and the third approach (right) on 02/02/2020.

4. References

[1] Stephen Gilmore, Paul Jackson. (2020). Informatics Large Practical © School of Informatics, University of Edinburgh. Available at: https://learn-eu-central-1-prod-fleet01-xythos.content.blackboardcdn.com/5d1b15b77a8ac/9695945?X-Blackboard-Expiration=1607072400000&X-Blackboard-Signature=wnjZRvrzKYKrmGCwVccyUjWCH0I5Ptt7srQ%2BZVJ9f%2BU%3D&X-Blackboard-Client-Id=301835&response-cache-control=private%2C%20max-age%3D21600&response-

content-disposition=inline%3B%20filename%2A%3DUTF-8%27%27ilp-coursework-v1.0.2.pdf&response-content-type=application%2Fpdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=20201204T030000Z&X-Amz-SignedHeaders=host&X-Amz-Expires=21600&X-Amz-Credential=AKIAZH6WM4PL5M5HI5WH%2F20201204%2Feu-central-1%2Fs3%2Faws4_request&X-Amz-Signature=cd0c03d370c8324ff03a9ec7345e125be17f31ce2a629b3252acfc0e904a6884

[2] Martijn Pieters. (2019). StackOverflow. Available at: https://stackoverflow.com/questions/3838329/how-can-i-check-if-two-segments-intersect