# Automatic Routing of UAVs for Wildlife and Habitat Monitoring

Mihir Bala
Rye High School, Rye, NY

## Abstract

Unmanned Aerial Vehicles (commonly referred to as “drones”) are becoming an important tool in wildlife conservation and habitat monitoring. They enable low cost surveillance of remote areas that are difficult to reach, at a scale that would otherwise be impossible with the typically meager resources of most conservation organizations. Unfortunately, their potential is currently limited by the fact that long range operations have to rely on autonomous flight when the drone extends beyond radio control range. The current process of manually specifying the route for autonomous flight is error prone especially in remote areas, because of challenging terrain variations, restricted airspaces and weather conditions along the route.

This paper describes a novel algorithm for automatically computing an efficient route that satisfies both static constraints (such as terrain, no-fly zones, etc.) and dynamic constraints (such as weather and wind data) between any two points on the surface of the Earth. The computed route is returned in a standard format that can be uploaded into any drone that supports GPS based autonomous flight - a capability that is already widely available in low-cost commercial drones. I have implemented this algorithm as a Web Service on the Google Cloud Platform, providing a flight routing service for drones analogous to the way Google Maps provides a map routing service for vehicles, that anybody in the world can use.

I hope that my work will increase the use of drones in wildlife conservation around the world, and ultimately improve the habitat and lives of wild creatures that have been my lifelong passion.

## Introduction

Unmanned Aerial Vehicles, also known as “drones” are becoming an important tool in wildlife conservation and environmental monitoring. Drones have helped perform tasks such as combating elephant poaching, monitoring birding sites, and collecting data about endangered habitats [24, 25, 26]. The range of commercial civilian drones used in these applications has been increasing rapidly, with some drones whose ranges now exceed 180 kilometers [1]. While these longer ranges allow drones to be more versatile, they also present new challenges.

Sight-guided radio control needs to be complemented with an autonomous flight capability when the drone goes beyond the radio controller’s range. Fully autonomous flight involving automatic terrain sensing and navigation is too cost prohibitive to include on most civilian drones. Due to this problem, drone operators have resorted to mission planner tools to define a pre-determined sequence of GPS waypoints (latitude, longitude, elevation tuples) for the drone to follow. This system allows operators to precisely control drones even when their range extends beyond a hand-held radio controller. However, today’s mission planner tools are very limited in their capabilities [2]. They are primarily a graphical user interface that allows the user to manually mark GPS waypoints on a map, with no mechanism to assist in the routing process. While this process works well within a small geographic range and relatively even terrain, it is tedious and error prone when the range increases and terrain variations can amplify the margin for error. Manual routing is an especially acute problem in wildlife conservation scenarios, that typically operate in very remote areas with challenging terrain and weather conditions. To address this issue, I have created a service that automatically finds the most efficient aerial route between two GPS coordinates on the surface of the Earth. This route can then be flown by any drone with the ability to fly a route of GPS/GPX coordinates.

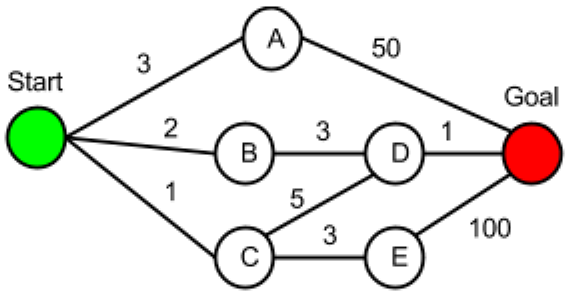
This paper describes my 2 year technical journey in chronological order, starting with a study of well known pathfinding algorithms, followed by a description of how I adapted one of these algorithms to deal with routing around both stationary and dynamic obstacles. Next follows a description of the system I designed and implemented on the Google Cloud Platform with experiments I performed to troubleshoot some performance and latency problems I encountered with my implementation. The paper ends with a novel optimization I developed and am currently implementing. I conclude with some thoughts on how I plan to extend my basic drone routing service by exploiting its situational awareness across multiple users operating drones in the same area.

## Overview of Pathfinding Algorithms

### Dijkstra's Algorithm

Dijkstra's Algorithm [3] is a classical path finding algorithm that uses breadth-first search to compute the shortest path between two nodes in a weighted directed graph, given a cost for each edge. The weighted directed graph can be used to model the routing constraints; for example a road map can be modeled as a graph whose nodes represent road intersections or other landmarks, and the edges represent the roads between these nodes (with their length as the associated edge cost). As each node is visited, the algorithm arranges them in a priority queue based on the node's distance to the start node. The next node is the node with the lowest distance from the start node. This process repeats until either there are no more nodes left that are reachable from the start node or the goal node has been reached.

Dijkstra's algorithm is best illustrated with an example. Figure 1 shows a graph with seven nodes labeled start, goal, A, B, C, D, and E with the start and goal node labeled in green and red respectively. Two sets of nodes, visited and unvisited, are created. The start node is first placed in the visited set. All other nodes are placed in the unvisited set with an initial estimated distance of undefined from the start node. The next node visited is the node with the smallest distance from the start node. Whenever the algorithm moves to the next node (the node with the smallest distance from the start on the frontier), the new current node is placed in the visited set and estimated distances for all neighboring nodes in the unvisited set are updated from undefined to their true value. The frontier nodes A, B, and C are put in a priority queue with C being at the top of the priority queue because it has the smallest distance from the start node. The next node would be B which has a smaller distance to the start (2) compared to A (3). Continuing with the process, the algorithm would visit neighbors of C; E and D get added with costs of 4 and 5 respectively to the queue. This step is repeated until the goal node is at the top of the priority queue, or all nodes in the unvisited set have an estimated distance of infinity. The lower panel of Figure 1 illustrates the steps.



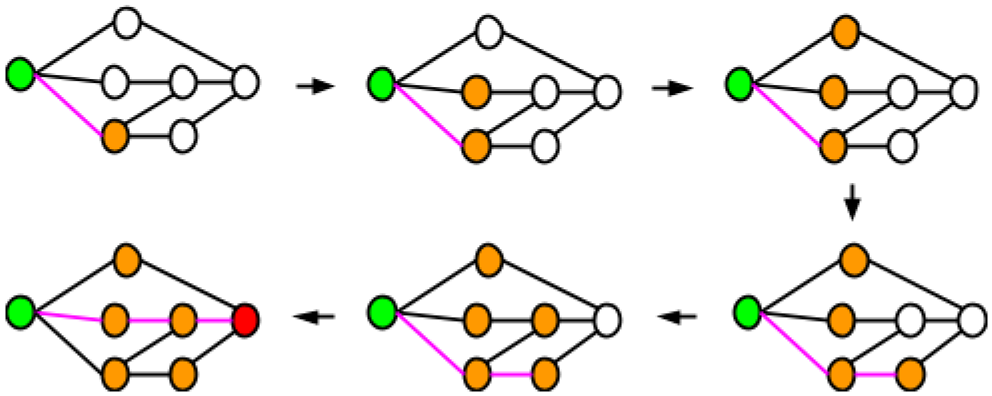


*Figure 1: Illustration of Dijkstra's algorithm. The figure illustrates a sequence of stages in the algorithm, where the orange nodes represent the ones that are visited, and pink edges reflect the shortest path among visited nodes at that stage.*

Dijkstra's Algorithm has a worst-case complexity of $O(E + Nlog(N))$ where $E$ is the number of edges and $N$ is the number of nodes in the graph. This is because every edge is visited once as shown in Figure 1 (*E)*, and the set of nodes in the priority queue have to maintained in a sorted order $Nlog(N))$. Thus the algorithm could visit every node in the graph to compute the shortest path. This happens because the algorithm focuses on the distance from the start node, meaning that every neighbor will be explored before making forward progress.

## A* Algorithm

The A* Algorithm [4, 5] is another path finding algorithm that uses an additional heuristic to estimate the shortest path between two nodes in a weighted directed graph given an edge cost. For each node examined, the algorithm arranges frontier nodes in a priority queue based on the sum of the cost of traveling from the start to the node (*g* value) *and* the estimated distance from the node to the goal which is based on a heuristic function (*h* value). The estimated cost for a node can be written in the form $d(n) = g(n) + h(n)$ where $d(n)$ is the estimated cost for node *n*, $g(n)$ is the cost of traveling from the start to the node, and $h(n)$ is the heuristic distance from the node to the goal. The next node considered is the node with the smallest estimated cost. The process continues until the goal has the lowest *g* value in the queue or until the queue is empty.

Figure 2 is the same graph that was used to demonstrate Dijkstra's Algorithm. For the purpose of explaining A* here, I make the simplifying assumption that *h(n)* is the known exact distance from the node *n* to the goal, but clearly, a heuristic function is in practice always an *estimate* of the distance to the goal. There are seven nodes: start, goal, A, B, C, D, and E with the start and goal labeled in green and red respectively. Two sets, visited and unvisited, are created. The start node is placed in the visited set and all other nodes are placed in the unvisited set. The frontier nodes A, B, and C are put in a priority queue based on the heuristic described above. The next node would be B because the sum of its *g* and *h* values (6) is smaller than that of A (53) and C (7). As shown in the figure, D (6) would still be above E (104) in the priority queue, C(7) or A (53). This process continues until the goal is reached or the priority queue is empty.

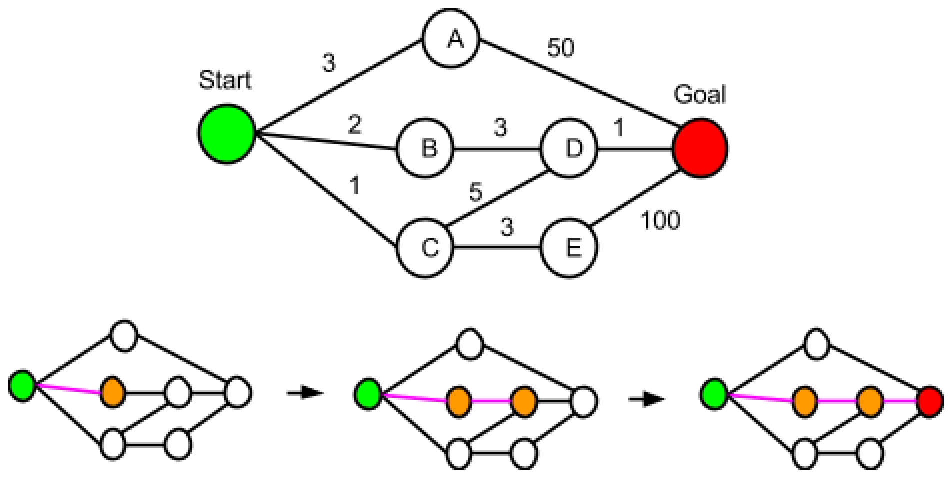


*Figure 2: Illustration of the A* algorithm.*

A* is guaranteed to find the shortest path if the heuristic function is *admissible*. That is, it should never overestimate the distance between a node and the goal. The heuristic function is always defined by the user for the specific problem that is being solved. In routing for example, the standard heuristic is Euclidean distance from a node to the goal. This is clearly an admissible heuristic so A* is always guaranteed to find the shortest path for routing.

With an admissible heuristic, A*'s complexity is $O(l)$ where $l$ is the length of the solution path. This can be clearly seen in the steps taken in Figure 2 where the path explored by A* is indeed the length of the solution path. This is because the heuristic will focus A* towards the solution as shown in the figure, and therefore will traverse the minimum amount of edges to reach the goal.

## Comparison of A* Algorithm and Dijkstra's Algorithm

The A* Algorithm and Dijkstra's Algorithm are both effective routing algorithms in context of global aerial routing. However, A* has a significant advantage in performance. This is due to A*'s inclusion of a heuristic function. Because the heuristic function factors in the distance of the current node to the goal, the algorithm is forced to search towards the goal and constantly make forward progress. The difference in performance can also be seen by comparing the algorithms' complexity. Dijkstra's complexity, $O(E + Nlog(N))$, is polynomial time in the number of edges and nodes, while the A* Algorithm's complexity equation, $O(l)$ is linear time, given an *admissible* heuristic function. As illustrated by Figures 1 and 2, the A* algorithm ends up visiting fewer nodes and makes faster progress towards the goal.

These characteristics make A* an ideal candidate for use in drone routing, because even relatively small geo-spatial grids (a grid whose nodes are latitude, longitude tuples) require a large number of nodes in order to model terrain variations with acceptable accuracy. This results in large graphs containing tens of thousands of nodes in practice, and the performance advantage of A* over Dijkstra will become significant. As a rough calculation, consider a geo-spatial grid that is 10 km x 10 km in size. If we want to model terrain variation with an accuracy of 100 m, this grid will have 100 x 100 = 10,000 nodes. Arbitrarily selecting diagonally opposite ends of this grid as the start and goal, and assuming there are no terrain obstacles, A* will only visit the nodes that are close to the diagonal path while Dijkstra will visit all 10,000 nodes.

## D* Algorithm

D* [6] is an incremental pathfinding algorithm, that finds the fastest route between two nodes on a graph without relying on edge weights. This algorithm is useful in situations where the graph can only be partially specified, such as the surface of a remote planet (D* is the algorithm used to route the Mars Rover [7]). D* is not ideal for our aerial drone routing problem because we already have sufficient terrain and weather data for the surface of the Earth to fully model it as a weighted graph.

## Bidirectional Search

Bidirectional Search [8] is an alternate approach to the conventional shortest path finding algorithm. As the name implies, Bidirectional search invokes A* from both the start and end node on a weighted directed graph given edge cost. When the routes for each individual A* search meet, the algorithm connects the two routes into one route. However, there is no major performance increase and therefore no advantage over the conventional A* Algorithm.

## Applying Pathfinding Algorithms to Aerial Routing

There are many challenges in applying these pathfinding algorithms to aerial routing. For instance, the problem of aerial routing is three dimensional (latitude, longitude, elevation) while most path finding algorithms work on a two dimensional graph. To account for this, the terrain can be modeled as a grid spaced using increments of latitude and longitude lines where each cell is a node with a cost determined at that location. This allows the pathfinding algorithm to traverse the graph normally yet still work in three dimensions. In addition, there are many new obstacles to consider that are not present in terrestrial routing including local elevation, wind, and weather. In this section, I will explain how I modified the A* algorithm to address these new factors.

Like terrestrial routing, obstacles in aerial routing can be also modeled by cost. Terrain variations can be modeled using elevation data to represent cost. Thus, the cost to go from A to an adjacent node B on the graph can be elevation(B) - elevation(A). In the algorithm, this cost can be used as the weight of the edge (A, B). Restricted airspace such as no-fly zones can be modeled using a fixed additional cost associated with all edges that lead into such a zone. Weather conditions such as rainfall forecasts can also be modeled in a similar way. For example, if rainfall is expected near node B, an additional fixed cost can be added to the edge (A, B).

Elevation, rainfall, and restricted airspace are examples of navigation constraints that can be modeled as a statically computed cost component associated with every edge because they are global effects that have the same effect over large portions of a drone's route. But constraints like wind have much more local effects on the flight path of the drone, and its effects can vary depending on a drone's heading relative to wind direction. The cost of getting to an adjacent node needs to reflect the drone's current heading relative to the wind, as illustrated by the simple case of Figure 3, where the drone's heading is either exactly opposite or exactly aligned with the wind heading.

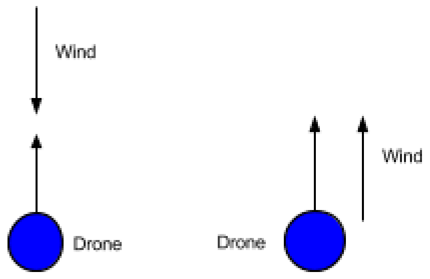


*Figure 3: Drone flight direction relative to wind direction.*

To translate this to the A* algorithm, the cost of a neighboring node must reflect a computation of the angle between the drone heading (if it moves to that node from its current point) and wind direction.  While a drone in one area can be affected negatively by the wind, when it reaches its next position, the wind might benefit the drone as shown in Figure 4.

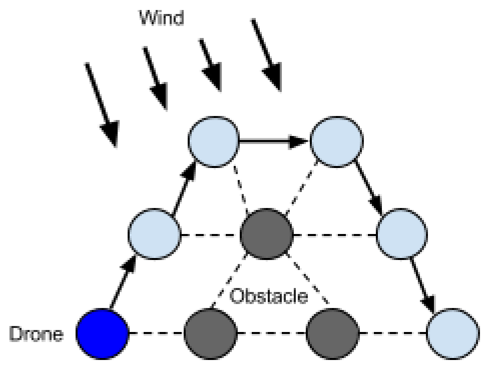


*Figure 4: To the left of the obstacle, the drone is flying against the wind and is therefore negatively affected.  To the right, the drone is traveling with the wind and is therefore positively affected.*

To handle such location and direction specific constraints, my solution was to add a dynamically computed cost factor based on the relative heading of the drone with respect to the wind, when computing the next frontier of neighbors in the A* algorithm (see Figure 5).

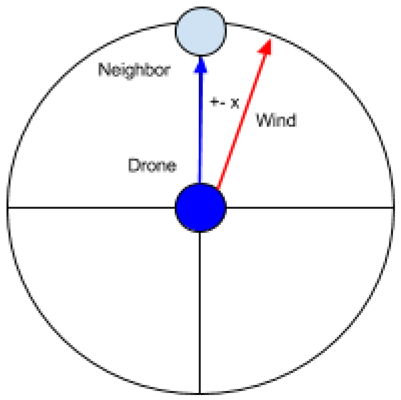


*Figure 5: A visual representation of how the cost of wind is computed in my A* algorithm.*

If the difference between the heading of the wind and the heading of the drone is $\pm x$ (where $x$ is a small constant) degrees then it will be a low cost to travel between the current location and the neighbor location.  If not, the difference in heading will be multiplied by a constant depending on the difference between $\theta$ (the drone's bearing relative to true North) and $\theta'$ (the wind's bearing relative to true North).  This can be modeled in the following equation where $Q$ is cost, $a$ is a constant, and $\theta$ and $\theta'$ represent the bearing (relative to true North) of the drone and the wind respectively:

$$Q = a|\theta - \theta'|$$

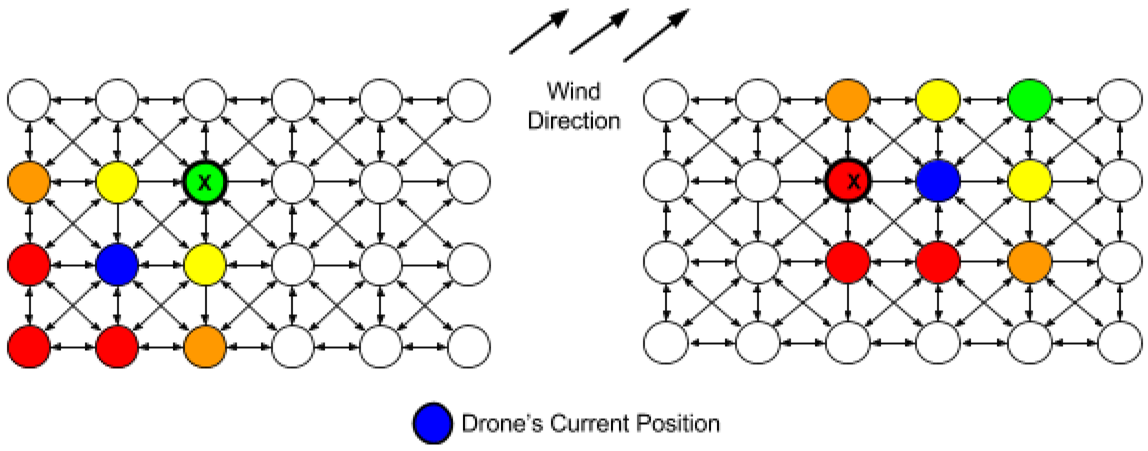


*Figure 6: How wind-related costs change with the drone's position.*

With the cost computed, the graph can be populated with the cost of wind relative to the drone's current position. Every time the drone changes position, the cost of the graph will change because of the cost of wind relative to the drone's new position might be different. Figure 6 below visually illustrates how the relative costs are set based on drone position, when the wind direction is fixed. The figure shows the immediate neighbors of the drone's current position color coded as a heatmap gradient ranging from green (low cost) through red (high cost). The wind relative costs are shown for two different drone positions relative to a given node labeled X.

Given that many factors influence drone routing, one needs to combine the factors to determine a drone's route. The proposed mechanism to achieve this is to incorporate a linear weighted combination of the different factors into the cost of each node in the pathfinding algorithm. That is, different factors such as elevation, precipitation and wind speed are combined to build a weighted estimate of the cost of each neighboring node. The general form of weighted combination will be an equation of the form shown below, where three major obstacles (elevation, precipitation, and wind) are combined to compute the cost of a single node in the graph:

$$Q = aE + bP + cW$$

In the general form, $Q$ is the final cost while $a$, $b$, and $c$ are all constants or weights that reflect that importance of each factor in drone routing, and $E$, $P$, and $W$ represent the actual elevation, actual level of precipitation, and relative cost of wind respectively. Elevation, precipitation, and wind data will be computed from public domain APIs (e.g. elevation data from the Google Elevation API [9] and precipitation and wind data from Open Weather Map [10]).

Note that the cost of wind, $W$, can change for the same location depending on what the drone's current location is, as explained earlier and illustrated in Figure 6. These constants are responsible for representing how significant an obstacle is for a given drone. For example, if the next location in a drone's path is a mountain, then the value of $aE$ will be scaled up to be so large that the A* algorithm will never consider it as a possible location to move through.

The constants $a$, $b$, and $c$ can be determined based on a given drone model's characteristics and will take into account the weight, power, maximum elevation, and other key features. This allows every drone to be unique in its cost calculations which therefore prevents drones from being pushed beyond their limits or not being used to their full potential. The $c$ constant for wind can be determined through wind testing of the drone in a controlled environment. This equation can be extended in a straightforward manner to include additional terms required to model other natural or man-made constraints that can impact the computed route. A future enhancement can provide a database of recommended values for the constants $a$, $b$, and $c$ for different commercially available drone models. These can be used as the default values in computing a drone's route, unless overridden by the user.

One final consideration when applying pathfinding algorithms to aerial routing is the curvature of the Earth. Because our grid is spaced using constant increments of latitude and longitude coordinates, this could introduce error as range increases. This is because the actual distance in meters between two successive coordinates along the longitude (or great circle) is larger near the poles than near the equator. To correct for this, grid spacing can be determined by the Haversine formula [27], a formula that converts a distance in meters to a distance in latitude and longitude degrees corresponding to where the points are on the Earth's surface. Using the Haversine Formula, it is possible to space the graph evenly while the space between longitude lines changes.

## System Design and Implementation

As stated in the previous section, aerial routing depends on cost modeling of elevation and weather. Fortunately much of the data for these factors are available as APIs on the Web. In order to exploit this fact, I implemented the aerial drone routing algorithm as a Web Service on the Google Cloud Platform, enabling anybody in the world to use it from a Web browser or programmatically via an API. My implementation itself relies on other publicly available Web Services to obtain up-to-date static information such as terrain data and dynamic information such as wind speed data. Several Web Services provide data about latitude, longitude and elevation across much of the earth, such as the the Google Maps API, and Google Elevation Data API [9]. Dynamic weather and wind speed data is available via Web Services like NOAA [11] and Open Weather Map [10]. No-fly zone (restricted air spaces) data is available from Mapbox [15]. A demo of my service is available at: www.neon-griffin-788.appspot.com.

The current implementation of my Web Service models static terrain information and works as follows. The user accesses the Web Service by going to its public URL from a web browser. A web form is presented through which the user first enters information about the starting latitude and longitude coordinates and range of the drone. Instead of manually providing the latitude and longitude of the start position, the user may also use the displayed Google Maps mashup (implemented as a Javascript front-end program) to search for and navigate to the starting point visually. In this case, my Web Service will get the latitude and longitude of the starting point automatically from Google Maps. Once the starting point is known, it is displayed on the Google Maps mashup as a circle whose center is the starting GPS coordinate of the drone. The radius of the circle is the drone range input by the user. Figure 7 shows a screenshot of the interface, showing the circle drawn on top of a location somewhere in the Rocky Mountains.

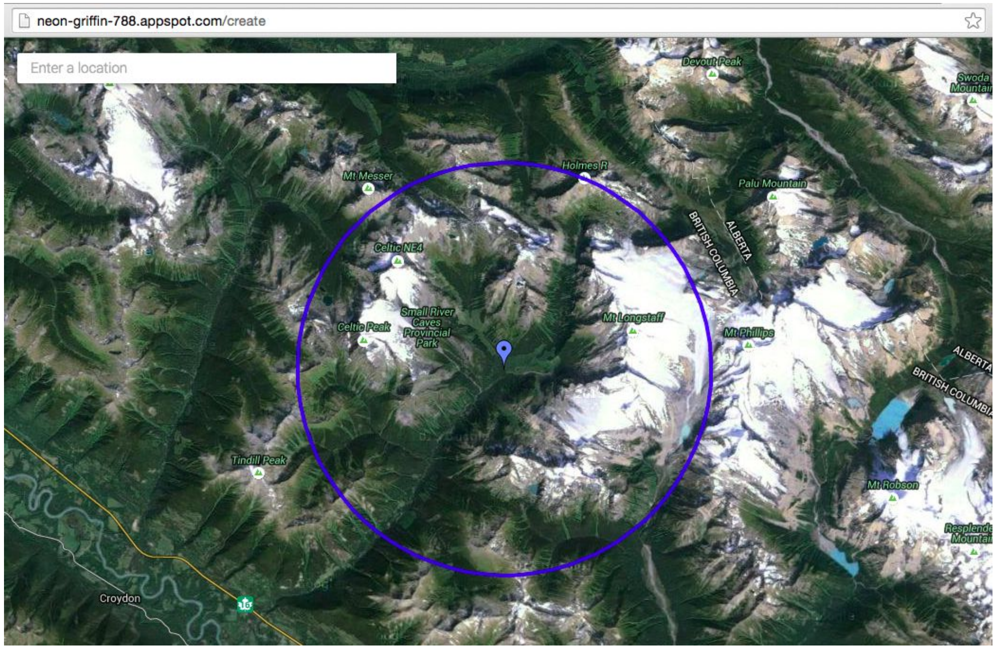


*Figure 7: Screenshot of the drone routing service, showing the chosen starting point as the center of a circle whose radius is the drone's flying range.*

The user can then click anywhere within the circle to select a destination point - clicks outside the drone range circle will have no effect. When the destination point is selected, the Javascript front-end program sends the source, destination, and range information to a back-end program written in the Python programming language, and executing as a Google App Engine application on the Google Cloud Platform. This back-end program implements my adapted version of the A* algorithm, and computes a route in 3-dimensional space from start to destination as an array of GPS coordinates. Each GPS coordinate is a 3 tuple of latitude, longitude and elevation. This array is returned to the Javascript front-end program, which then displays it as a path that is overlaid on the Google Maps interface, as shown in Figure 8.

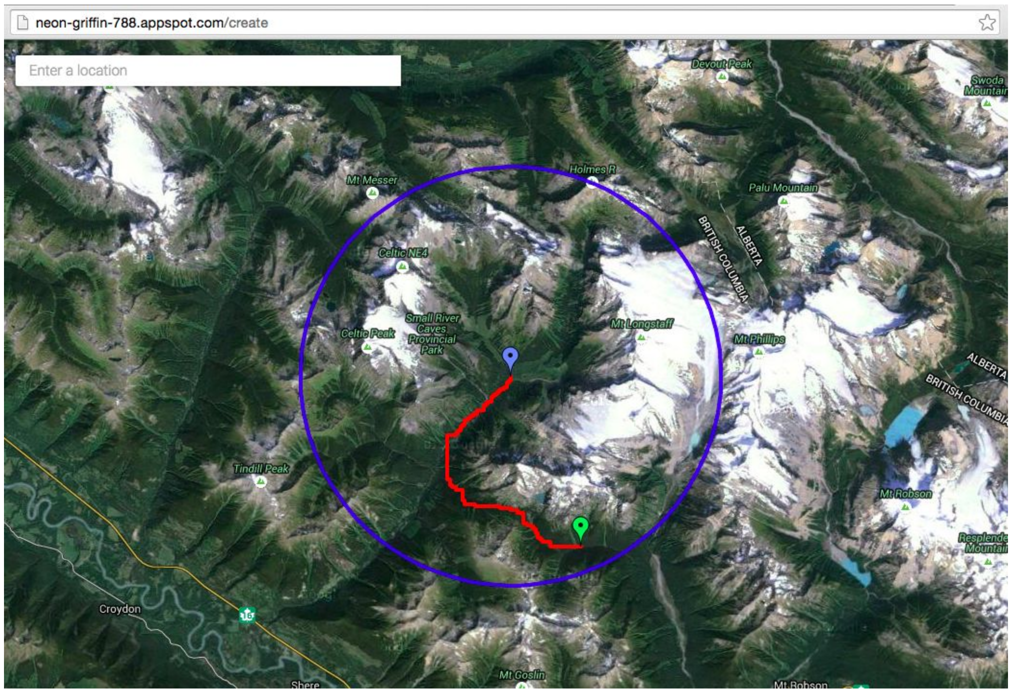


*Figure 8: Screenshot of the drone routing service, showing the computed route (in red) between a user selected start point and destination point. Note how the route avoids high elevation areas by following the valley when possible.*

The user can optionally download this route as a GPS Exchange Format (GPX) file [12] to the local computer. Most commercially available drones with GPS flight support allow uploading of flight plans specified as GPX files. Uploading of the GPX file to the drone's on-board computer is typically accomplished using a USB connector between the user's computer and the drone. This completes the end-to-end scenario, where the user interacts with my drone routing Web Service to retrieve a route, and uploads it to the drone in preparation for autonomous flight. The user may also simulate the drone flight using Google Earth, which allows the GPS route to be "played" in a 3-dimensional terrain map of the Earth. This provides the user with additional assurance that the route successfully navigates around obstacles as promised. Although the demonstration shown here is a one-way route from the starting point to a single destination, the underlying implementation is capable of handling a multi-destination route, possibly returning to the starting point. Figure 9 shows the architecture block diagram of my Web Service implementation.

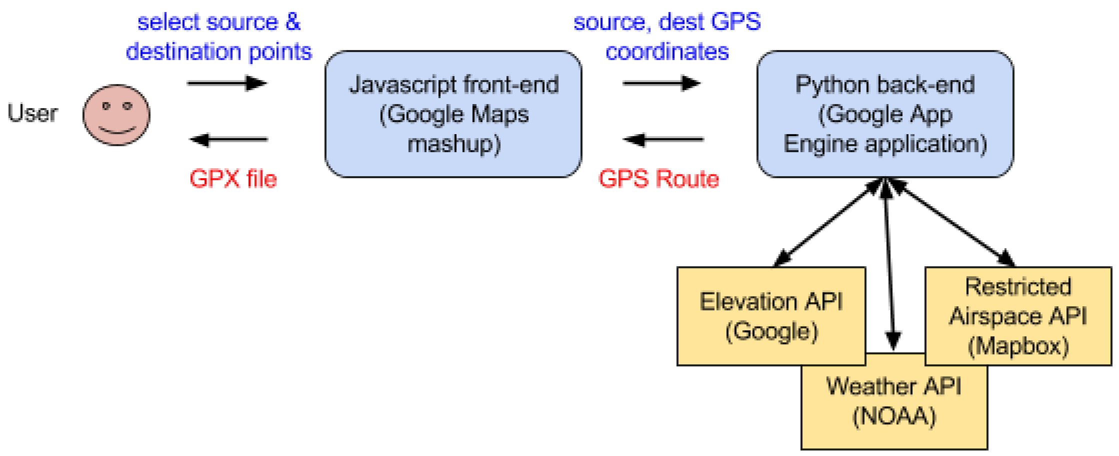

User
select source & destination points
GPX file
Javascript front-end (Google Maps mashup)
source, dest GPS coordinates
GPS Route
Python back-end (Google App Engine application)
Elevation API (Google)
Weather API (NOAA)
Restricted Airspace API (Mapbox)

*Figure 9: Architecture diagram of my drone routing Web Service. The blue boxes represent software that I wrote, and the yellow boxes represent external Web Service APIs that my software uses to get elevation, weather, and no-fly zone data to determine the cost function. Note that the current implementation only considers elevations in its cost.*

## Experiments

When I got the entire Web Service to work end-to-end, I was initially elated, but that soon turned to dismay as I discovered that the latency of my drone routing service appeared to get substantially worse as the distance from start to goal increased. To better understand the reasons for this, I performed a series of tests to assess completion time of various aspects of the system.

### A* Admissible Heuristic Experiment

**Hypothesis:** A* completion time increases exponentially as distance from the start to the goal increases because the heuristic is not admissible. This could account for my performance issues.
**Experiment:** Time my A* implementation while traversing a slowly increasing line of equal cost nodes.
**Procedure:**

1. Create a 20 kilometer north to south straight line of nodes with even spacing and equal cost. I chose a north-south direction for the computation because the spacing between latitude lines is independent of location on the earth. For this particular experiment, I used 100 meter spacing.
2. Set the southernmost node as the start node and the node immediately to the north of the start node as the goal node.
3. Time A* between the start and the goal node.
4. Move the goal node 100 meters north to the next node.
5. Repeat steps 3 and 4 until the goal node is the northernmost node in the line.

**Result:** The results shown in Figure 10 demonstrate the average running time for A* over 3 runs, as I gradually increased the distance to the goal. My hypothesis was incorrect. A* completion time increases linearly in relation to distance. This clearly proves that my A* implementation indeed uses an admissible heuristic, and is not the cause of my latency issues. A graph of the experimental results is shown below.

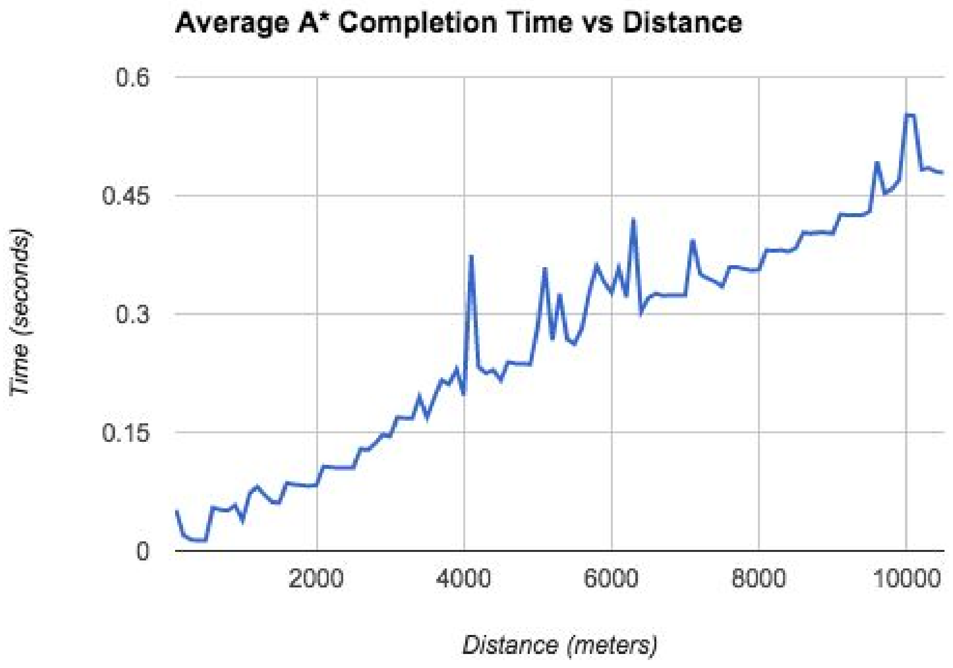


*Figure 10: The results of the average time taken to find the goal over 3 runs*

## API Request Time Experiment

**Hypothesis:** The completion time for each Google Elevation API request (which is used to determine the elevation cost component for frontier nodes visited in the A* algorithm) is the source of my latency issues.

**Experiment:** Time each invocation of the Google Elevation API from my A* implementation to get the average response time.

**Procedure:**

1. Choose a start and goal node within 10 kilometers of each other.
2. Invoke A* with the start and goal node.
3. Time each Google Elevation API request.

**Result:** My hypothesis was correct. Figure 11 shows the variability in time taken to make a single API call to the Google Elevation API. The API took an average of 0.38 seconds for each call.  Because the number of API calls is directly proportional to the area of the search, the amount of API calls increases with distance which causes the high latency initially observed. A 10 Km distance with elevation data retrieved every 50 meters, results in at *least* 2,000 API calls, which would take more than 1 minute. This is a long time for a user to wait.

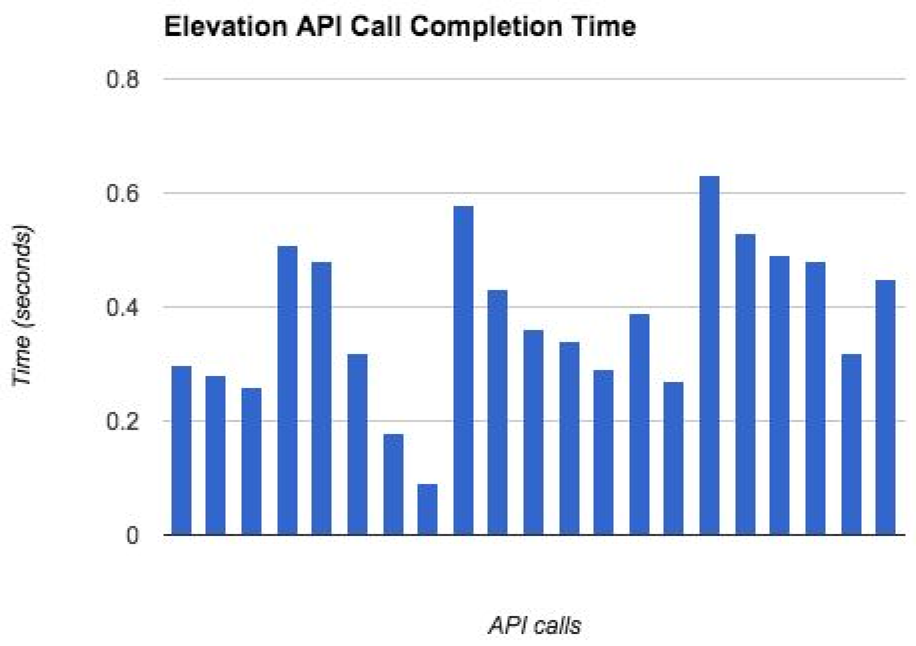


*Figure 11: The time per API call and variability in the time to complete the call*

## Improving The Current Design

To address the performance problem described in Figure 11, I came up with a new design that should eliminate large portions of the region from being represented in the graph.  The trouble with my old design was that I had modeled every single point in the region in the graph, and assigned it a cost.  As a result, the path finding algorithm has two undesirable traits: (a) it is biased towards areas of low elevation, and (b) it requires making too many API calls.

The algorithm's bias towards areas of low elevation (e.g. valleys) is quite clear in Figure 8.  The reason for this is my method for assigning cost to nodes.  Currently, elevation determines cost, meaning that areas of low elevation will always be lower cost than areas of high elevation.

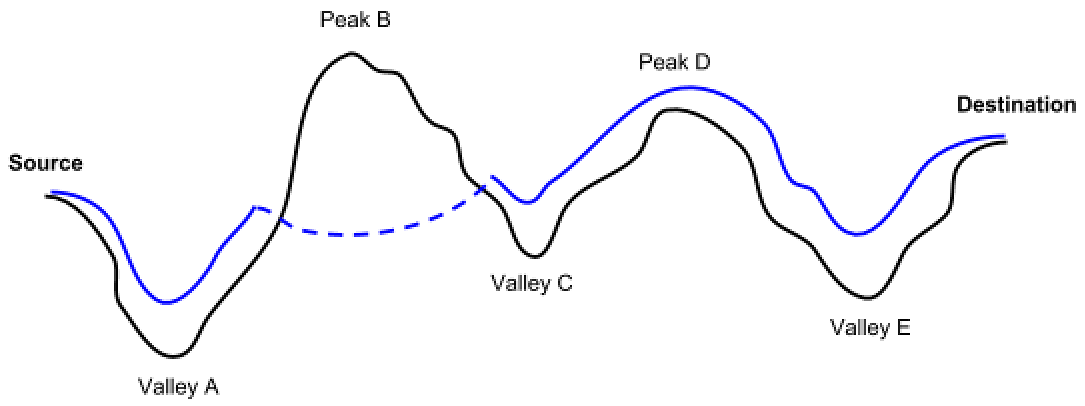


*Figure 12: The current implementation of our routing service. As shown in Valleys A, C, and E, the route dips into each valley causing the drone to fly an unnecessarily longer flight.*

My insight is that the most efficient aerial route between two points in a three dimensional space with no obstacles is a straight line route. When obstacles are introduced into the space, the route must avoid the obstacles but still generally follow the original straight line route. Given the example in Figure 12, we could compute a more efficient route by applying the concept stated above. The first step is to draw a line between the source and the destination as shown in Figure 13:

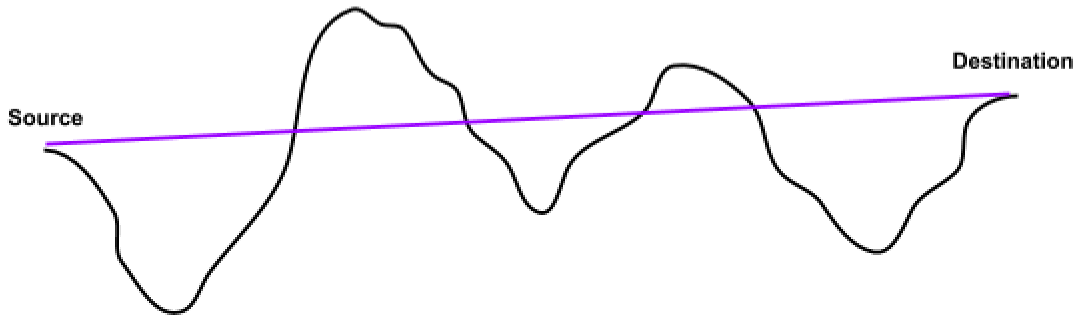


*Figure 13: The straight line route between the source and the destination.*

Next, we can isolate the areas of elevation that are greater than the elevation of the straight line route as shown in Figure 13. Because all other terrain obstacles are below the straight line route and therefore not blocking the route, they are irrelevant to the computation of the route.

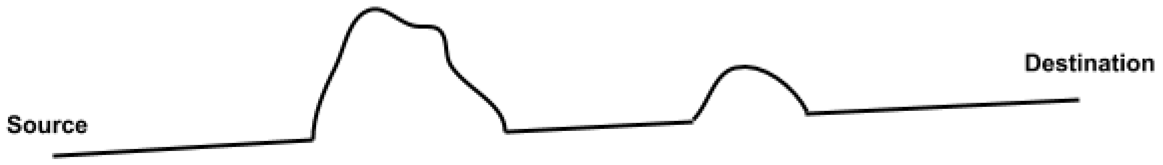


*Figure 14: The elevations greater than the route elevation are isolated.*

The A* Algorithm can now be invoked *only* around the elevated islands as shown in Figure 14. This can return a far more efficient route than the original route (compare the route in Figure 15 with the one my service currently computes in Figure 12).

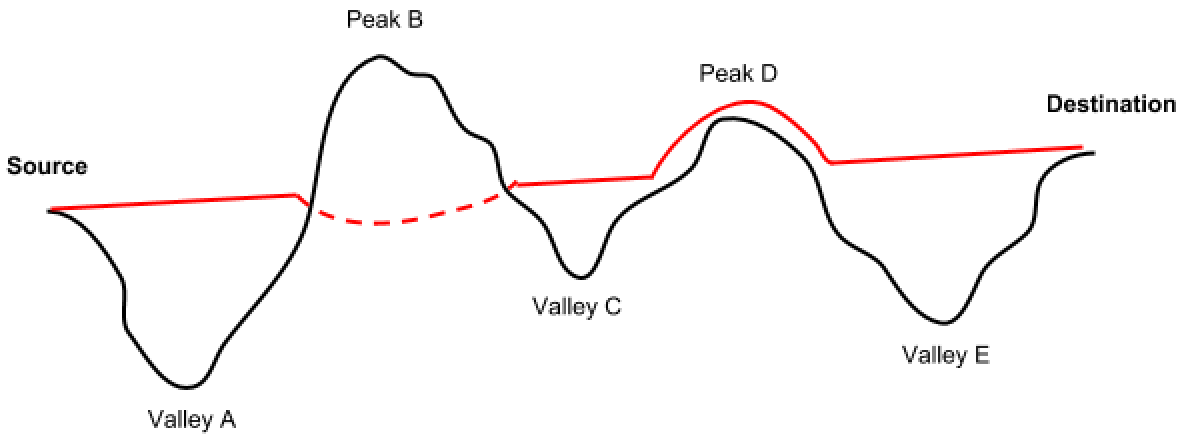


*Figure 15: The routing algorithm is now biased towards a straight line path because elevations greater than the straight line route elevation were isolated. This results in a more efficient path.*

I am currently implementing the method demonstrated above to improve my routing service. Two main improvements I seek are increased performance and greater accuracy. Hopefully, because my new method disregards much of a given graph, it will create fewer API calls (which as demonstrated in the Experiments section, are extremely inefficient).

## Conclusion and Future Work

In addition to improving the Web Service by implementing wind information, I also have future plans for my idea. As of now, my service focuses on routing drones efficiently by avoiding terrain obstacles. The cost modelling for these obstacles is constant; that is, regardless of the drone flown by the user, the cost remains unchanged. However, this is a fallacious assumption because all drones have different characteristics. For instance, a heavier drone may have more difficulty ascending than a lighter drone. To account for this, cost has to be custom modelled for each type of drone. One way to accomplish this is to test every drone in a set of standardized experiments (wind tunnel experiments, speed experiments etc.) to determine how it performs.  This information could be stored in a massive database that could easily be accessed by my Web Service and used to accurately model cost for every type of drone.

I believe my Web Service could change how drones are controlled in future.  Instead of tedious, manual routing, drones could fly effortlessly between specified destinations delivering goods, mapping terrain, or even monitoring endangered species.  Not only does my Web Service allow easier routing but it can also coordinate drone swarms more fluidly.  For instance, if a delivery company wanted fifty drones to deliver packages to homes across the country, it would simply have to map out the destinations and the drones could be automatically routed to each home.  This is the power of automatic routing and I trust that it soon will become the standard for drone routing.

A future extension of this Web Service will also provide a capability similar to what the Federal Aviation Authority (FAA) does for civilian aircraft.  As more and more customers use the same Web Service to route drones, the Web Service gains situational awareness of what drone is flying in a particular area.  Specifically, this means that the Web Service could use elevation data, dynamic weather data from NOAA, and paths for other nearby drones in its drone routing computations.  Further, it can be used to predict where a drone may have fallen if it fails to reach its destination. This is useful for recovering drones lost during flight.

My next goal is to work with a conservation team that operates drones in remote areas for wildlife and habitat monitoring, to field test the routes that my Web Service computes. I hope to learn valuable lessons about how to tune my algorithm for their use cases. It will also help me improve the user experience.

## References

1. The Corsica project: flying a drone from Nice to Corsica over the Mediterranean sea. http://wiki.paparazziuav.org/wiki/Corsica.
2. APM Mission Planner: http://planner.ardupilot.com/.
3. Dijkstra, E. W. Dijkstra's single source shortest path algorithm: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.
4. A* pathfinding algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm.
5. Stanford University. Amit Patel's A* blog: http://theory.stanford.edu/~amitp/GameProgramming/.
6. D* incremental pathfinding algorithm. https://en.wikipedia.org/wiki/D*.

7. Mars autonomy project: path planning. https://www.frc.ri.cmu.edu/projects/mars/dstar.html.
8. Bidirectional search algorithm. https://en.wikipedia.org/wiki/Bidirectional_search.
9. Google Elevation API: https://developers.google.com/maps/documentation/elevation/.
10. Open Weather Map wind and precipitation API: http://openweathermap.org/api.
11. National Oceanic and Atmospheric Administration (NOAA) Web Services: https://www.ncdc.noaa.gov/cdo-web/webservices.
12. GPS Exchange Format. https://en.wikipedia.org/wiki/GPS_Exchange_Format.
13. Google App Engine. https://cloud.google.com/appengine/docs.
14. Google Cloud Platform. https://cloud.google.com/.
15. Mapbox no-fly zone data. https://www.mapbox.com/blog/dont-fly-here/.
16. Source code for my Drone Routing Web Service: https://github.com/mihirbala/dronemap/tree/master/webservice.
17. DJI Ground Station: GPS waypoint routing. http://www.dji.com/product/ipad-ground-station.
18. Drone Buddy. https://itunes.apple.com/us/app/drone-buddy-fly-uav-safe-weather/id992303145?mt=8
19. Ravenscroft, D.L. Computing flight plans for UAVs while routing around obstacles having spatial and temporal dimensions. US Patent 8082102. https://www.google.com/patents/US8082102?dq=routing+routing+uav&cl=en
20. Huss, R.E. and Pumar, M.A. Method for determining terrain following and terrain avoidance trajectories. US Patent 5706011 A. https://www.google.com/patents/US5706011.
21. Kugelmass, B. Unmanned aerial vehicle and methods for controlling same. US Patent 9075415 B2. https://www.google.com/patents/US9075415.
22. Koh, L. P. and Wich, S. A. 2012. Dawn of drone ecology: low-cost autonomous aerial vehicles for conservation. Tropical Conservation Science Vol. 5(2):121-132.
23. Amazon Delivery Drones: http://www.cbsnews.com/news/amazon-unveils-futuristic-plan-delivery-by-drone/
24. Conservation Drones: http://conservationdrones.org/.
25. Audubon Society Magazine, July 2014. Drones take off as tool for wildlife conservation.
26. Wildlife Conservation UAV Challenge. http://www.wcuavc.com/.
27. Haversine formula. https://en.wikipedia.org/wiki/Haversine_formula.