Spring 5-1-2014

Optimum Path Planning for an Impaired Aircraft

Suzannah Bailey

Optimum Path Planning for an Impaired Aircraft

A Capstone Project Submitted in Partial Fulfillment of the Requirements of the Renee Crown University Honors Program at Syracuse University

Suzannah Bailey
Candidate for BS and BA Degrees
and Renee Crown University Honors
May 2014

Honors Capstone Project in Aerospace Engineering

Capstone Project Advisor: _____
Professor John F. Dannenhoffer, III

Capstone Project Reader: _____
Professor Melissa Green

Honors Director: _____
Stephen Kuusisto, Director

Date: 23 April 2014

Abstract

Proportionally speaking, it is safer to travel by plane than any other form of transportation. However, in some parts of the world such as Africa, the lack of updated aircraft, instability within the region, and inexperience of flight crews contribute to a higher rate of aircraft incidents and accidents. This capstone combines elements from aerospace engineering, as well as international relations to create a program to mitigate these risks.

This new algorithm, the Bailey Algorithm, is very different from the commonly used Dijkstra Algorithm. Unlike Dijkstra, the Bailey Algorithm not only incorporates the distance traveled between cities, but it also applies costs at airports visited along the way. To effectively generate the best possible path, the Bailey Algorithm combines the Dijkstra Algorithm with an optimization method called Simulated Annealing.

To show the effectiveness and variety of the Bailey Algorithm, several scenarios were created, based on real incidents. These scenarios were then applied in a 600 mi^2 area in East Africa. Selecting this region allowed for variation in topography, and therefore more constraints to be used in defining scenarios.

To account for a variation of possible impairments, some scenarios dealt with mechanical malfunctions, such as one where cabin pressurization becomes a problem, restricting the plane from flying above 5,000 feet. Other scenarios depend on the way the plane interacts with the environment. For example, in one scenario, there is a leak of toxic chemicals, which means the plane cannot fly over National Parks or other protected areas.

Although this program was only exercised on a small number of airports in East Africa, the Bailey Algorithm is able to be modified for any region of airports around the globe. Due to scenarios being created that involve mechanical malfunctions, environmental impacts, and passenger health, the Bailey Algorithm has shown that it is applicable in a variety of situations. In addition, it is easily adaptable to more than the seven scenarios considered.

Table of Contents

Executive Summary

As an aerospace engineer and international relations dual major, it was important for me to pick a capstone that combined elements from both disciplines. Under the advisement and guidance of Prof. John F. Dannenhoffer, III this was accomplished. This capstone, entitled "Optimal Path Planning for an Impaired Aircraft," created a program to generate emergency action plans that would allow an aircrew to mitigate risks associated with potential impairments.

This capstone began in Spring 2013 with the official proposal. The objective was to create a new path-planning algorithm that, given a specific scenario, could plot a path to safety. In an effort to make sure the capstone stayed on track, weekly meetings were held with Dr. Dannenhoffer. Before each meeting, a summary was sent detailing the work that had been done since the last meeting. The capstone continued up through the Spring 2014 semester. At this point, it was turned into a presentable paper with the help of Professor Melissa Green, as the Reader.

This new algorithm, the Bailey Algorithm, is a significant extension of the commonly used Dijkstra Algorithm. The Dijkstra algorithm is one that is likely found in a standard GPS unit. It simply finds the shortest path from the origin to the destination.

Unlike Dijkstra, the Bailey Algorithm not only incorporates the distance traveled between cities, but it also applies costs at airports visited along the way. This is revolutionary because this means the Bailey Algorithm

takes into consideration the middle steps taken to get to the destination. To effectively generate the best possible path, the Bailey Algorithm combines the Dijkstra Algorithm with an optimization method called Simulated Annealing. Simulated Annealing is an approach to finding the minimum value of a given function. Applying it to the Bailey Algorithm, Simulated Annealing takes the initial and final airports and finds the path that has the lowest cost. This cost value is a combination of the distance traveled as well as the cost associated with visiting each city.

To show the effectiveness and broad applicability of the Bailey Algorithm, several scenarios were created, based on real incidents. Over a dozen aircraft incidents and accidents were surveyed to track down common impairments that could occur. From these, the seven most common were turned into scenarios. These scenarios were then applied in a 600 mi^2 area in East Africa. Selecting this region allowed for variation in topography, and therefore more constraints to be used in defining scenarios.

To account for a variation of possible impairments, some scenarios dealt with mechanical malfunctions, such as one where cabin pressurization becomes a problem, restricting the plane from flying above 5,000 feet. When this scenario was run, the Bailey Algorithm successfully generated the shortest path, while avoiding airports along the way that violated the elevation constraint.

Another scenario depends on the way the plane interacts with the environment. For example, in one scenario, there is a leak of toxic chemicals,

which means the plane cannot fly over National Parks or other protected areas. Once again, the Bailey Algorithm was able to find the optimum path while respecting the constraints.

A third scenario concerns with an ill passenger. Due to conflicts in the region, the passenger is unable to fly over the airspace of a specific country. However, they also need a hospital. The Bailey Algorithm was able to effectively find a path to take that finds hospitals while also avoiding Uganda, the forbidden country.

Although this program was only exercised on a small number of airports in East Africa, this report will demonstrate that the Bailey Algorithm is able to be modified for any region of airports around the globe. Due to scenarios being created that involve mechanical malfunctions, environmental impacts, and passenger health, the Bailey Algorithm has shown that it is applicable in a variety of situations. In addition, it is easily adaptable to more than the seven scenarios considered.

Acknowledgements

My sincerest thanks go out to Professor John F. Dannenhoffer, III. Without his guidance, support, and patience this project would have never taken flight. Thank you for knowing when to give me a push in the right direction, and when to sit back and let me make mistakes. Thank you also to Professor Melissa Green for taking the time to read my many drafts and respond to my numerous questions. This capstone reads much better due to your comments. Finally, thank you to my parents, for trying to understand what I was doing and constantly offering words of encouragement.

Chapter 1: Background Information

Introduction and Preliminary Work

This Capstone project, entitled "Optimum Path Planning for an Impaired Aircraft," encompasses both aerospace engineering and international relations. The goal was to create a path-planning algorithm that could take a specific impairment of an aircraft and generate an optimal path to safety.

In an effort to make the scope of the capstone manageable, airports needed to be selected in a relatively small region. To include an international aspect, this region was chosen to be in Africa. To pick a particular part of Africa, the prevalence of airports and airstrips was considered. In Figure 1, below, the yellow planes indicate larger airports, defined as having millions of visitors travelling through annually on major airlines.[1] The blue airplanes represent medium-size airports that have regular regional traffic.[2]

[1] "Airports in Africa." *Megginson Technologies, Ltd.* Updated 2009.
[2] Ibid.

Figure 1: Airports in Africa

The East Africa region was chosen because it offered variety in terms of mountains, large bodies of water, forests, and rebel activity. This variety would allow for very different scenarios to be used by the Bailey Algorithm to plot a path. Knowing this, the region shown in Figure 2 was selected.[3] In this figure, there are small pink planes as well. These planes represent airstrips that do not have regular service, the smallest of the three levels depicted.[4] This 620 square mile region included airports in Kenya, Uganda, Tanzania, Rwanda, Burundi, and the Democratic Republic of Congo.

[3] Ibid.
[4] Ibid.

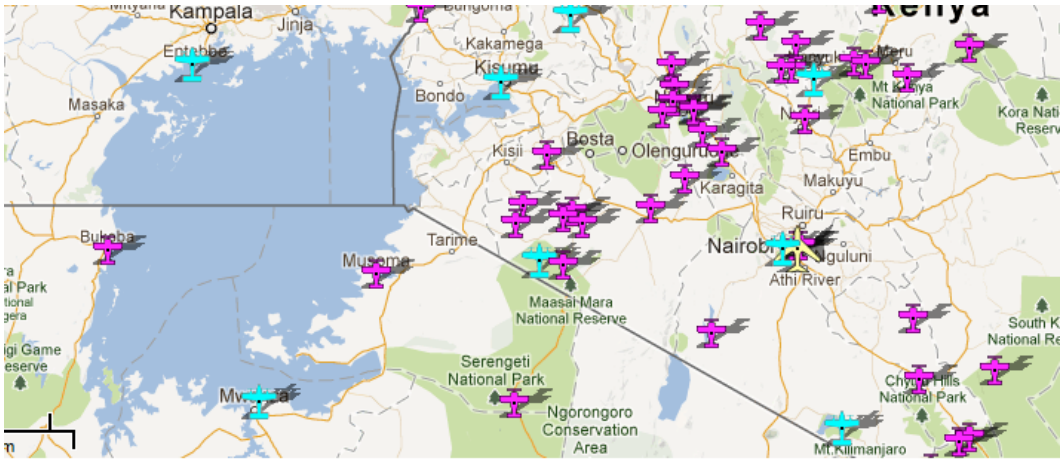

Figure 2: East African Airports

Having established the region and goal of the algorithm, it was time to research air accidents and incidents. After surveying dozens of incidents, two main trends became apparent:

- Common Plane: The DC-10 was involved in many air disasters. This can be attributed to its popularity and long lifespan.
- Common Causes: The three most common issues associated with disasters were: decompression or loss of pressure due to puncture of fuselage, loss of engine(s) or engine power, and fuel leaks.

The following sections in this chapter will explore the significance of runways, the importance of plane selection, the process of preparing the raw data, and the listing of the scenarios.

Runways

Before the Bailey Algorithm could be written, certain data needed to be collected. This included the location, elevation, direction, length, and surface of all the runways in the region. This information would be crucial when it came to selecting the "best" runway for an airplane to land safely on.

In an effort to have a large variety of airports, 30 different runways were chosen. A sample of the information collected is shown below. A full copy of the chart can be found in Appendix A.

			ELEVATION					LENGTH				
AIRPORT	CODE	LOCATION	ft	nmi	mi	m	COORDINATES	FEET	nmi	METERS	SURFACE	DIRECTION
Adjumani Airport	HUAJ	Adjumani, Uganda	2611	0.43	0.495	796	03°20'19"N, 31°46'08"E	3710	0.611	1130	Unpaved	09/27
Moyo Airport	OYG	Moyo, Uganda	3100	0.51	0.587	940	03°38'57"N, 31°45'54"E	4260	0.702	1300	Unpaved	02/20
Arua Airport	RUA	Arua, Uganda	3951	0.65	0.748	1204	03°02'50"N, 30°54'44"E	5600	0.922	1700	Unpaved	18/36
Gulu Airport	ULU	Gulu, Uganda	3510	0.58	0.665	1070	02°48'00"N, 32°16'30"E	10314	1.699	3144	Asphalt	17/35

Table 1: Runway Data

The first column is the airport name, followed by the code used to address it. The third column is the airport location. The next four columns are the elevation of the runway. Some of the information provided was in feet and some was in meters, meaning a conversion was necessary.[5] To remain consistent with typical aerospace units, the units of nautical miles were chosen. The column after the elevation shows the coordinates of the runway. The next three columns correspond to the length of the runway, in feet, nautical miles, and meters. The final columns are the surface and the orientation of the runway.

In terms of surface, there was a range of options. Some were paved, some were ice, and some were unpaved. The surface of the runways was

[5] "Airports in Kenya." *Air Broker Center International AB*. 2009.

necessary to know because it would affect the ground roll distance of the plane after landing. Based on the runway length, certain runways would not be possible for the plane to land on because there would not be enough space.

While recording all this information, the orientation of the runway was also noted. The orientation corresponds to the numbers printed on the ends of the runway, as shown in Figure 3:[6]

Figure 3: Runway Orientation

The numbers shown are the magnetic compass heading of the runway, ranging from 0 to 360 degrees, divided by 10 and rounded to the nearest integer. Using this convention, 0 degrees corresponds to due North. Each runway will have two numbers depending on which side of the runway the plane is entering or leaving. These numbers will always be 18 off from

[6] "Logan Plans to Add 600-Foot Runway Safety Area on Harbor Deck." *Boston Globe*, March 18, 2009.

each other, since they are 180 degrees apart.[7] Figure 4, below, shows this naming convention:[8]

Figure 4: Runway Orientation

[7] John Dannenhoffer, III, "Capstone Meeting: January 23." (Capstone Meeting, MAE 499: Honors Capstone Project, Syracuse, NY January 23, 2014).
[8] "Model Railroad Layouts: Airport Runways and Accessories." *Bakatronics LLC,* February 15, 2014.

Plane Information

The Bailey Algorithm is not dependent on one specific plane. Instead, it uses certain parameters such as the take off distance and cruise altitude to create viable scenarios. In Chapter 6, more specific aerodynamic characteristics will be discussed. However, in order for this algorithm to be as realistic as possible, a specific plane was chosen. This would allow characteristics of the plane to be used, such as stability, weight, fuel tank capacity and other variables that impact performance.

Knowing the region that was chosen, it was assumed that an older, more reliable and common plane would be more realistic. For this reason, the Cessna 172/182, Piper Cherokee, and DC-3 were all considered as the possible plane for the project.

There is often missionary work in the East African region selected. Based on research completed, the DC-3 is a plane that is commonly used for such work. Selecting the DC-3 includes additional benefits for the Bailey Algorithm as well. In the first place, the DC-3 requires a longer ground roll at landing than the Cessna or Piper. This will allow a scenario to be created that uses runway length as a constraint. Secondly, the DC-3 was built with an unpressurized cabin.[9] This allows a scenario to be created that includes an altitude restriction.

[9] "DC-3: The Genesis of a Legend." *DC-3/Dakota Historical Society, Inc.* March 26, 2014.

Data Preparation

Before the program can be run, the airport locations, as coordinate points, are imported from an Excel spreadsheet into MATLAB. The locations in the Excel sheet were obtained from researching airports and runways in the East African region. In the Excel sheet, the latitude and longitude were converted into coordinate points. To do this, the following conversion factors were used:

- There are approximately 69 miles between each degree of latitude. At the Equator, which is where most of the airports are located, the distance between each line of longitude is also 69 miles. As the lines of longitude approach the poles, the distance between each degree shrinks to zero.
- There are 60 minutes within each degree. Using the 69 miles as a base, this means each minute is approximately 1.15 miles apart.
- There are 60 seconds in each minute. Converting this into miles results in 0.019 miles per second.

Once these values were known, it was easy to convert the latitude and longitude into coordinate values. The coordinate values of the airports were found from summing the degrees, minutes, and seconds for each latitude and longitude measurement. In order to convert into nautical miles, the preferred unit for aerospace application, the sum was divided by 1.15. For simplicity, it was determined that the equator and 33° East should be the origin of the

graph. Once this was known, the airports could be graphed. Figure 5 shows the locations of the airports.

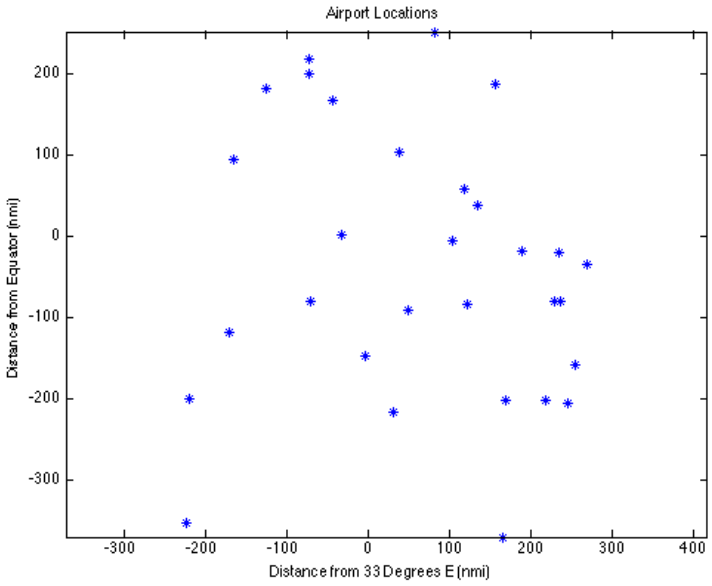

Figure 5: Airport Locations

In addition to graphing the airports, certain features were noted and graphed as well. In this case, hospitals, Lake Victoria, and Mountains were the notable features. They can be seen in Figure 6 below:

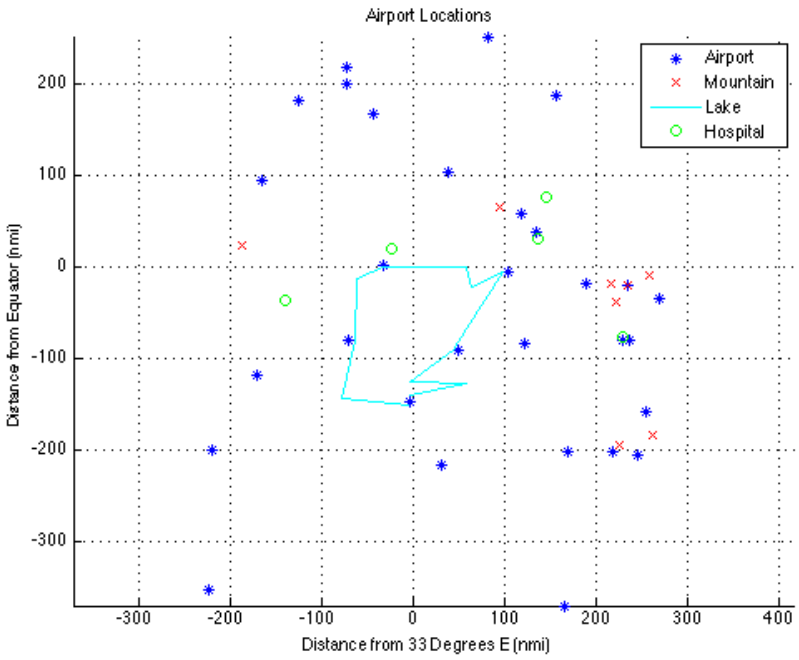

Figure 6: Notable Features

Knowing the configuration of airports and points of interest, seven scenarios were created. They are described in the section below.

Scenarios

1. While flying over an area inhabited by rebel forces, a barrage of bullets punctured the fuselage. Even thought the DC-3 was unpressurized, with an operating altitude of 10,000 feet, this caused some passengers to suffer from hypoxia (insufficient oxygen). In order to accommodate these passengers, the plane is unable to fly above 5,000 feet.

In this scenario, the penalty would be associated with cities, or nodes, that have an altitude greater than 5,000 feet. While in reality, planes can fly unpressurized up to an altitude of 12,500 feet, some individuals start to experience health problems due to lack of oxygen at altitudes as low as 8,000 feet.

In this particular case, the region selected is heavily mountainous. Some of the selected runways are at extreme altitudes that would prevent an impaired plane from landing, making this scenario realistic.

When flying, occasionally planes are restricted to specific altitudes. This scenario could be easily modified to account for that variation as well. For example, due to government regulations, a plane cannot fly lower than Z feet. This variability shows the importance of selecting an altitude restriction as a scenario.

In the MATLAB code, this scenario uses the X and Y locations of the airports as well as the elevation of each airport. Since this particular scenario

prohibits the plane from passing an airport that exceeds an elevation of 5,000 feet, it was also important to convert the elevation into nautical miles to remain consistent. Upon completing this conversion, it was apparent that the restriction prohibited the path from visiting an airport with an elevation over 0.82 nautical miles.

2. One of the flight attendants alerts the pilots that there is a passenger in desperate need of immediate medical attention. She is not sure what is wrong, but knows that the passenger needs the best medical facility that can be reached ASAP. In order to help the passenger, the pilot is given a list of high-level hospitals. S/he must select an airport close to one of these. However, the passenger is a former rebel, and therefore not allowed in Ugandan airspace. The pilot must land at the closest runway without crossing into Uganda.

This scenario takes two constraints into consideration: location of hospitals and what country the plane is flying in. Unlike most constraints, the hospital constraint would provide a reward instead of a penalty. In a scenario that has a penalty, the Bailey Algorithm adds the penalty to the cost. However, for this scenario, the reward means the value for the penalty is instead subtracted, resulting in a lower cost.

Additionally, there is the cost associated with restricted airspace. Like the elevation restriction in the previous scenario, there is a cost penalty

associated with visiting a node within this restricted space. Since there are "no fly zones" set up around the world, this is a viable scenario. By specifying two constraints, the scenario is slightly more challenging to fulfill. This is a reflection of the complex problems facing international travel today.

In this particular scenario, if the airport was not in Uganda AND there was a hospital close by, then the penalty value was subtracted from the cost. To indicate whether or not an airport was in Uganda, logical values were used. When the data was collected, a value of "1" indicated that yes, the airport was in Uganda. A "0" indicated that it was not. This same convention was used to identify if there was a hospital nearby.

3. Unfortunately, the tubes containing the hydraulic fluid were not replaced when they should have been, and they sprung a large leak of toxic Skydrol hydraulic fluid. Unfortunately, this batch contained maximum levels of organophosphates, which are, according to the EPA, "highly acutely toxic to bees, wildlife, and humans."[10] In order to protect the environment, the plane cannot fly over national parks or protected areas.

Forests, bodies of water, and national parks are essential for the survival of many groups of people. Humans need food, water, and shelter to

[10] U.S. Department of Health and Human Services, "Toxicological Profile for Hydraulic Fluids," September 1997.

survive. However, toxic chemicals used with planes can cause serious devastation when leaked.

By leaking the toxic Skydrol fuel, real hydraulic fluid still used today, airlines can have a devastating effect on the environment, reflecting poorly on the airlines. Coupling this poor public image with the fines associated with polluting a national park and the airlines would want to be able to avoid protected areas. For this scenario, Lake Victoria and National Parks were chosen as the natural features that were considered "protected areas."

Like Scenario 2, this scenario depended on logical values. Airports located in or very close to National Parks or Lake Victoria were assigned a value of "1," in the Excel sheet. At this time, there are specific entry columns for specific natural features. This would show that the program could avoid the protected areas.

4. A flock of Goliath Herons sprung up suddenly. The pilot had enough time to react so that only the port engine was damaged. Unfortunately, it failed completely. Since the rate of climb for an aircraft is dependent on the difference between power available and power required, losing an engine would lower the climbing abilities of an aircraft, resulting in a lowered Rate of Climb. For simplicity, it is assumed that the plane can only climb to an airport that is at a maximum altitude 20% higher than the airport just visited.

Assuming that the DC-3 was not in the best shape, and therefore the reported Rate of Climb might no longer be applicable, it was assumed that the aircraft could only travel to an airport that was at an elevation less than 20% higher than the current airport. For simplicity's sake, this was independent of the distance between airports.

Over 40% of all bird strikes can result in engine damage.[11] This can constrain the ability of a plane to climb. This particular scenario could be modified for other mechanical problems that would also impact the rate of climb, such as thrust available or elevator motion.

Since the important quantity for this scenario is elevation, it was crucial to input the elevation for each corresponding airport. To calculate the cost associated with an impaired Rate of Climb, the following equation was used. If the value returned was greater than 1.2, then the constraint was violated. In the equation, "i" represents the current airport, and "i+1" is the next airport in the sequence.

$$\frac{Airport\ Elevation\ (i+1)}{Airport\ Elevation\ (i)}$$

5. Flying over Lake Victoria, the pilot notices she is almost out of fuel. She remembers asking for 600 gallons of fuel, so she is originally confused. However, she then remembers that it was a Tanzanian who refueled the plane. The Tanzanian accidentally did not look at the

[11] Roger Nicholson and William Reed, "Strategies for Prevention of Bird-Strike Events," *Aero Quarterly*, Quarter 3: 2011, 19.

units and instead put in 600 liters of fuel (~158 gallons). With no fuel, the plane is effectively turned into a glider. The pilot knows she has enough to make it to her destination, but she only wants to fly by runways of at least 5000 feet, enabling her to land safely at any airport along her way if necessary.

Without a consistent international unit system, it is entirely possible for mistakes of this magnitude to be made. However, just because the plane is out of fuel does not mean that a crash is inevitable. It is theoretically possible to glide a plane to a safe landing. To model this, the Bailey Algorithm assumed a runway length of 5,000 feet was the minimum distance for a safe landing. When coming in without power, there is no reverse thrust available to slow the plane. This means a longer runway distance is required.

Runway length is a serious concern for two reasons. First of all, when landing, the plane needs enough distance to slow down safely to protect the passengers. Second, once the plane lands at an airport, it does not sit there forever, it has to be able to take back off. In order to achieve takeoff, the plane must generate enough thrust to overcome the weight of the plane. The thrust is increased as the speed increases. In order for this to happen, the plane needs a long enough runway to build up enough speed.

This same scenario could be used when there is a complete loss of power. The cause for the impairment is not what matters, but how the plane reacts. As with scenario 1, the runway length was converted into nautical

miles and the imported into the MATLAB code. Unlike scenario 1, where there was a penalty for going over the constraint, this scenario has a penalty for going under the constraint. Since the length was chosen to be 5,000 feet, this translates to 0.82 nautical miles as the minimum runway length allowable.

6. While flying a special New Year's flight, a rogue firework exploded near the rudder of the plane, severing one of the 2 connections. Shrapnel from the firework got wedged in between the fuselage and the rudder, locking it into a right turn position, and overriding the safety mechanisms in place to prevent such a thing from occurring. With the rudder locked the plane is not capable of making left turns.

The Bailey algorithm looks at the node being visited and takes the cross product of the link used to get there and the one leaving. If the cross product is negative, the turn is considered "left," and "right" if the cross product is positive. By using the cross product, it does not matter where the plane started. Since the cross product will determine direction, it will work no matter if the plane is going from Airport 5 to Airport 25, or the other way around.

Initially, the thought was that links could be designated as either a "left" or "right" turn. Instead, it was decided that using cross products was more efficient. In this particular scenario, left turns are prohibited. However,

the program could be easily modified to prevent right turns or all turns, only favoring straight paths. Furthermore, this scenario counts all left turns as bad. In future versions, the code could be modified to allow slight turns, to see how the cost is affected.

7. In a rush to load the plane quickly, the ground crew neglected to properly tie down the cargo. As a result, during takeoff, items shifted moving the center of gravity to the aft of the plane, making the center of gravity aft of the stick-fixed neutral point. This leads to static instability, with the nose inclined above the fuselage, rotating the aircraft away from the equilibrium point.

Unfortunately, this is a serious unrecoverable issue. When the plane is stable, it has a center of gravity either forward of, or located at the stick fixed neutral point. However, by neglecting to properly secure cargo, the cargo can shift, therefore shifting the center of gravity.

When the center of gravity is behind the stick fixed neutral point, the plane is statically unstable. This means that the plane becomes too sensitive to handling by a pilot. Tragically, this scenario often leads to fatal consequences.

As this capstone progressed, it was discovered that to account for this scenario would take more time and resources than were available. For this reason, this scenario would be one to be considered as future work.

Chapter 2: Existing Path Planning Algorithms

Introduction

Path planning algorithms are more prevalent than most people would realize. They exist in mapping software and GPS units, but the concept behind them exists in many more aspects of life. For example, a first-year student will "map" out their college courses. Like a GPS unit, this takes into consideration where you started and where you want to end up. Think of each required class as a "node." Once a student completes a class, it is on to the next one. This is similar to how GPS units and other mapping programs work.

How the program determines which "node" to go to on the way to the final destination is where a specific algorithm is used. In the next sections, two common mapping algorithms, A* and Dijkstra will be explained. Both of these algorithms are commonly used for mapping, but both have drawbacks as well.

The A* Algorithm

The first mapping algorithm that will be discussed is the A* (pronounced "A-Star") algorithm. This algorithm operates in a 2-dimensional field. The basic idea of the program is that it takes a "start" location and an "end" location and fills in a grid between the two. The grid essentially consists of vertices (nodes), including the start and end "node," that make up all possible locations for a path to get from the start to the end.

Once the start location is known, the remaining "nodes" on the grid are split into "possible" or "impossible" nodes. In very simple terms, a node is "possible" if it is connected to the start node. From the list of "possible" nodes, the cost is calculated.

The goal of the A* algorithm is to find the path with the lowest cost. With the A* algorithm, the cost is calculated using a very basic formula: F = G + H.[12] In this case, the "G" term is the cost associated with moving from the current node to the next node.[13] This can be different based on the specific movement being made, direction traveled, or any other factor.

The "H" term is what defines the A* algorithm. The "H" term is a value associated with moving from the current square to the final square.[14] Essentially, this value is a guess, since the program does not know what path will be chosen. The "H" stands for "heuristic." A heuristic is a method used to

---

[12] Patrick Lester, "A* Pathfinding for Beginners," *Policy Almanac*, July 18, 2005.
[13] Ibid.
[14] Ibid.

improve problem solving, such as finding the best path. The value of the heuristic can change as the path is developed. For example, if there is a large blockage between the current node and the final destination, the heuristic might be very large.

Once the next node is chosen, the process of calculating the cost is repeated until the path is complete. Since the "G" value tends to remain constant, the value of the heuristic is the important value in the A* algorithm. This means that the heuristic can have an impact on what the final path is.

A high heuristic means short computational time, but not necessarily the shortest path.[15] If the heuristic has a value of zero, then the A* algorithm has essentially become the next algorithm mentioned, the Dijkstra algorithm.

[15] Ibid.

The Dijkstra Algorithm

As mentioned in the previous section, the Dijkstra algorithm is essentially the A* algorithm with a heuristic value of zero. In other words, Dijkstra simply looks for the lowest cost to get from one starting point to another "node."[16]

When Dijkstra starts, it recognizes a start and an end node. Assigning a value of zero to the current node (starting node), it assigns a value of infinity to all other nodes. From the starting node, Dijkstra calculates the cost to each available node as the distance to the next node added to the current node's value. If the new value for the unvisited node is less than the current value of that node, then the value is replaced to the lesser one and that node becomes the next one in the path. For example, if the start node is 0, and the distance to the Node 2 is 4, then Node 2 now has a value of 4, not infinity.

At each node, all possible connection costs are calculated. Once it calculates the shortest distance to the next node, it accepts the node and repeats the process.

This penalty value associated with a node is what sets the "Bailey" algorithm apart. As will be explained in the next section, this algorithm is able to assign a penalty function that accounts for the way in which the plane arrived at the node, something no other path planning algorithm has been able to do.

---

[16] "Dijkstra's Shortest Path Algorithm," Cornell University, accessed April 21, 2014.

Chapter 3: The Bailey Algorithm

Introduction

The Bailey algorithm is different from any other existing mapping program. Not only does it look at how one got to a specific node, but it also looks ahead to see where one is going. This is the biggest difference from the Dijkstra Algorithm, and what truly sets the Bailey Algorithm apart.

The Bailey algorithm does incorporate Dijkstra, as a method to establish an initial cost. Like Dijkstra, initially the cost for each link is calculated based on the distance between the nodes. However, based on a certain scenario, a specific penalty is applied to certain nodes, allowing the Bailey Algorithm to reject certain nodes that are too expensive to visit. By doing this, the Bailey algorithm finds the best path, not necessarily the shortest path.

Another way the Bailey algorithm is different from Dijkstra is that the Bailey algorithm incorporates Simulated Annealing as the method to calculate the best path. Simulated Annealing will be explained in the subsequent sections. Briefly summarized, Simulated Annealing is an optimization method used to find the "best" possible solution. What makes the Simulated Annealing program unique is that it allows solutions that initially do not appear to be the best option to be considered. The Bailey Algorithm uses a function to evaluate whether the new path is "not too much

worse." If it fulfills this requirement, then the new path will be accepted as a possible solution.

In the rest of this chapter, a flow chart diagram explaining the Bailey algorithm will be included and explained in detail. A complete copy of the code is contained in Appendix B1.

Exploring the Bailey Algorithm

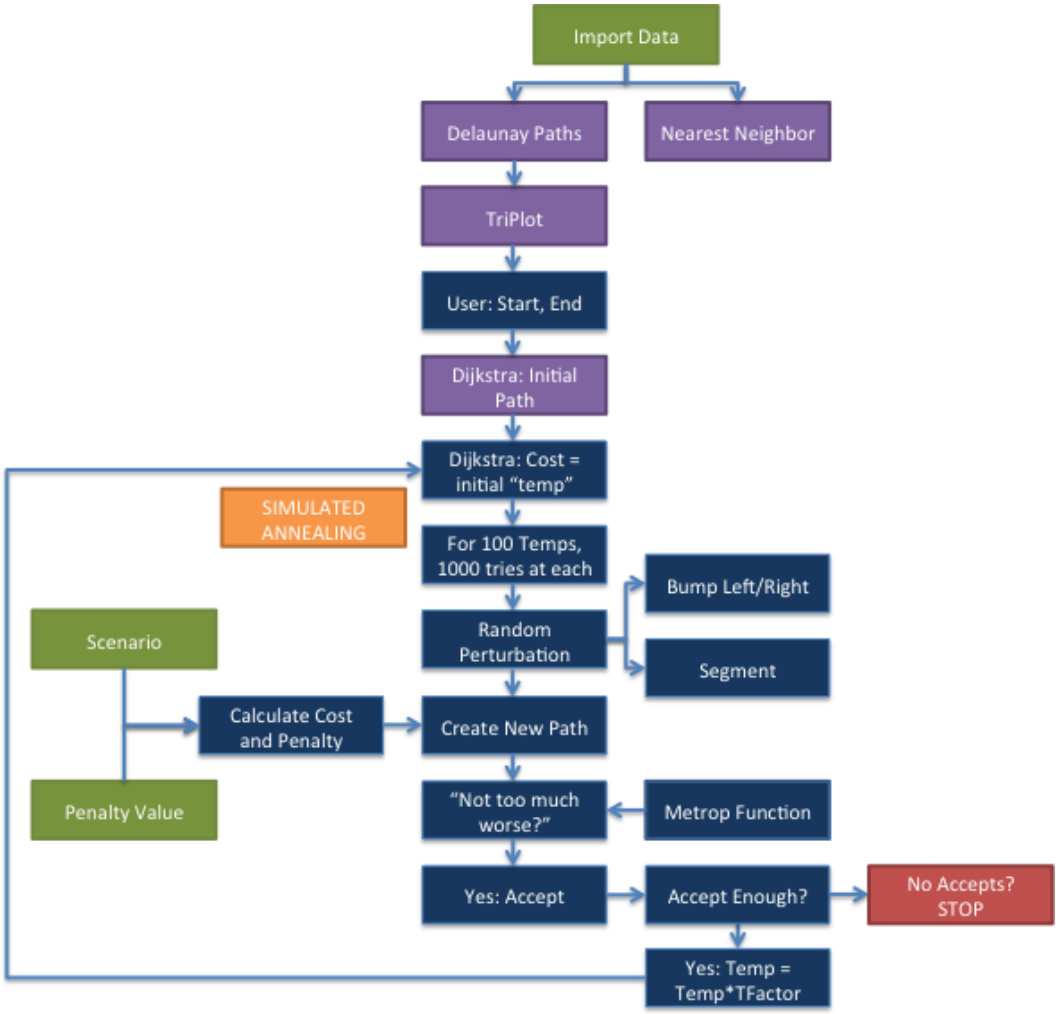

Figure 7: Flow Chart

The flow chart in Figure 7 shows how the Bailey Algorithm works. The purple boxes correspond to built-in MATLAB functions. For clarity, the start of the portion of the algorithm that uses Simulated Annealing is marked with the orange box. The green boxes indicate the values that will change based on the size of the region being used, the location of the airports, and the specific scenario being run. The red box indicates when the code is considered complete.

Once the initial data preparation had been completed and the airports were graphed, it was necessary to connect the airports. The links connecting the airports are what determine the baseline "cost" to go from one airport to another.

There are two possible methods for connecting the airport nodes: the nearest neighbor approach, or using the built-in MATLAB function Delaunay. The nearest neighbor method is very simple. A radius of R nautical miles is initially decided upon. Around each node, a circle is drawn corresponding to this radius. Any other node that falls within that circle is then connected to the centermost node. This process is repeated for all nodes. An example is shown below in Figure 8. This is for a radius of R = 100 nmi. Even though the radius selected was 100 nmi, there are still some airports that are not linked to any others.

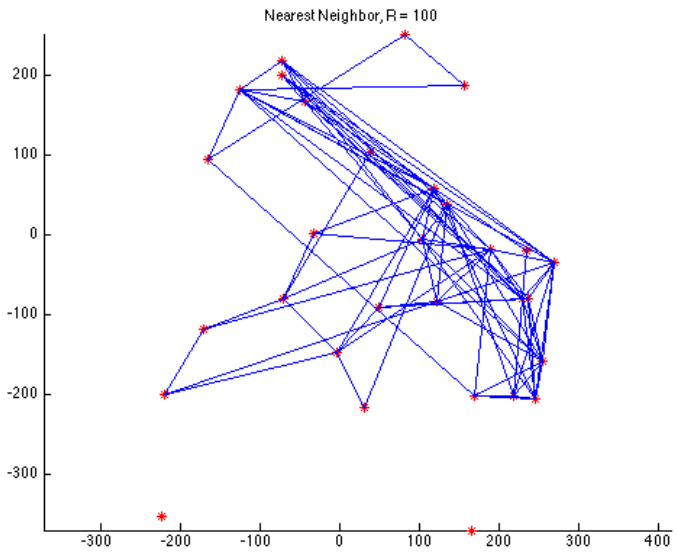

Figure 8: Nearest Neighbor Method

The Delaunay triangulation is slightly more complicated. All of the nodes are arranged such that a triangular shape can connect them. However, the triangle is not arbitrary. Once three nodes have been connected by a triangle, a circle is drawn around the points such that the three vertices of the triangle just touch the sides of the circle, making a circumcircle.

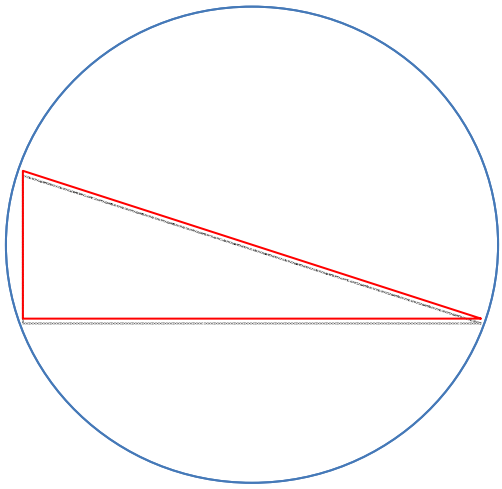

Figure 9: Triangle and Circumcircle

Each triangle is generated in an optimal way so that the minimum angle is maximized. Not only does this ensure that the triangles are as close to equilateral as possible, but it also means there are no other points within each circumcircle, making Delaunay unique.[17] Delaunay repeats the iterative triangle-making process until this condition is fulfilled.

The difference in approach between the two linking methods would result in different paths being drawn. Using the nearest neighbor method would require specifying a maximum distance for allowable links. This can result in many consequences, such as unreachable nodes if the link length is too short. A radius that is too large will allow all nodes to be linked, making this algorithm invalid. On the other hand, using the nearest neighbor method could shorten processing time, which is beneficial to a computer program.

To help determine which method to use, a short program entitled "Method Test" was written. This code can be found in Appendix B2. The Method Test code took the airport locations used in this capstone and calculated the distance to travel along all the links. The results are shown below in Figure 10.

[17] "Delaunay Triangulation." *MathWorks Inc.* 2014.

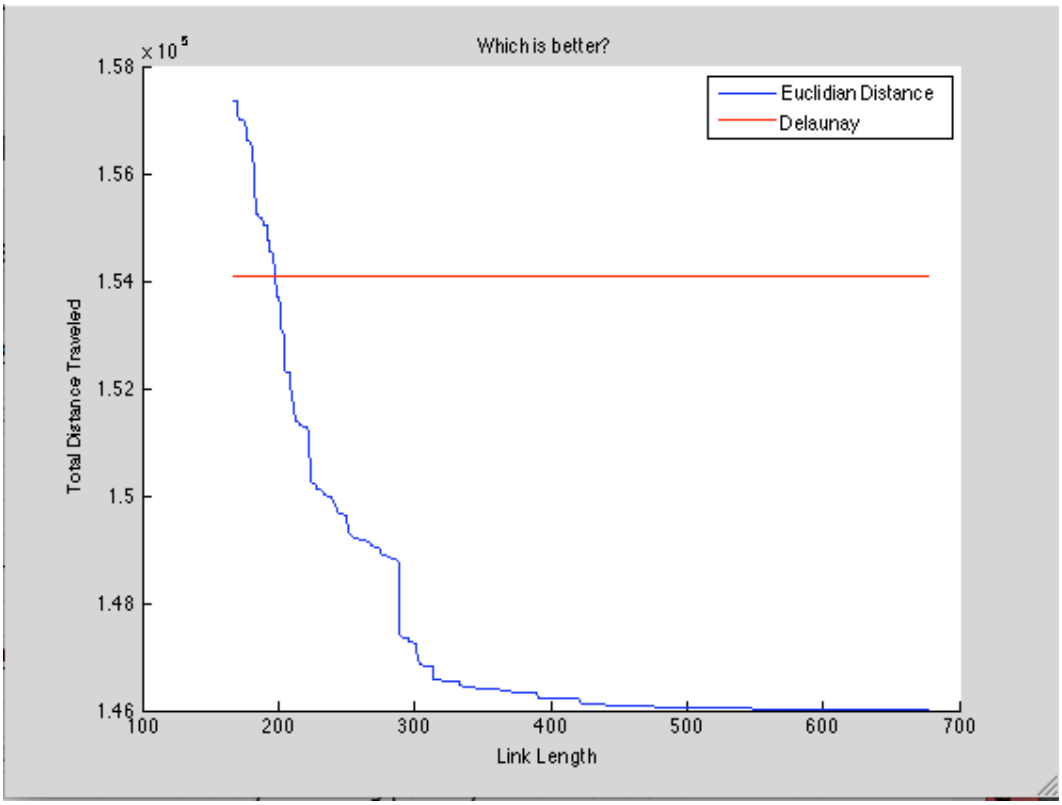

Figure 10: Delaunay Compared to Nearest Neighbor

In Figure 10, the X-axis represents the distance between the nodes in nautical miles. The Y-axis represents the total distance traveled between all of the links, also in nautical miles. The blue line represents the distance traveled using the nearest neighbor method. Clearly, as the link length increases, the total distance needed to visit all nodes decreases. The red line corresponds to the Delaunay triangulation. Since the Delaunay triangulation is independent of the link length, this value remains constant throughout the experiment.

The intersection between the two lines occurs at a total distance of approximately $1.54 * 10^5$ nmi. When comparing this to the range for the total

distances generated from the nearest neighbor approach, it is clear that Delaunay is less than 20% from either extreme value. This supported the decision to use Delaunay.

Upon completion of the Delaunay links, Figure 11 was generated:

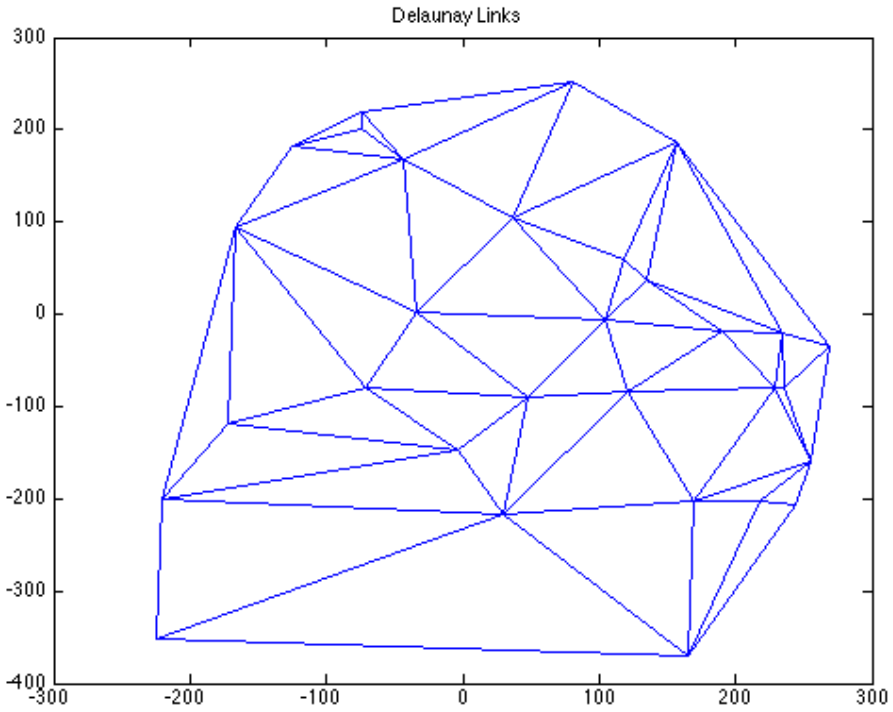

Figure 11: Delaunay Paths

From here, it is now possible for the user to determine which airport to "start" and "end" from. For this program, the user clicks on the desired start and end nodes, as seen in Figure 12. The starting airport is designated with a green dot, and the red dot indicates the final airport. The title of the graph shows which nodes are the first and final of the path.

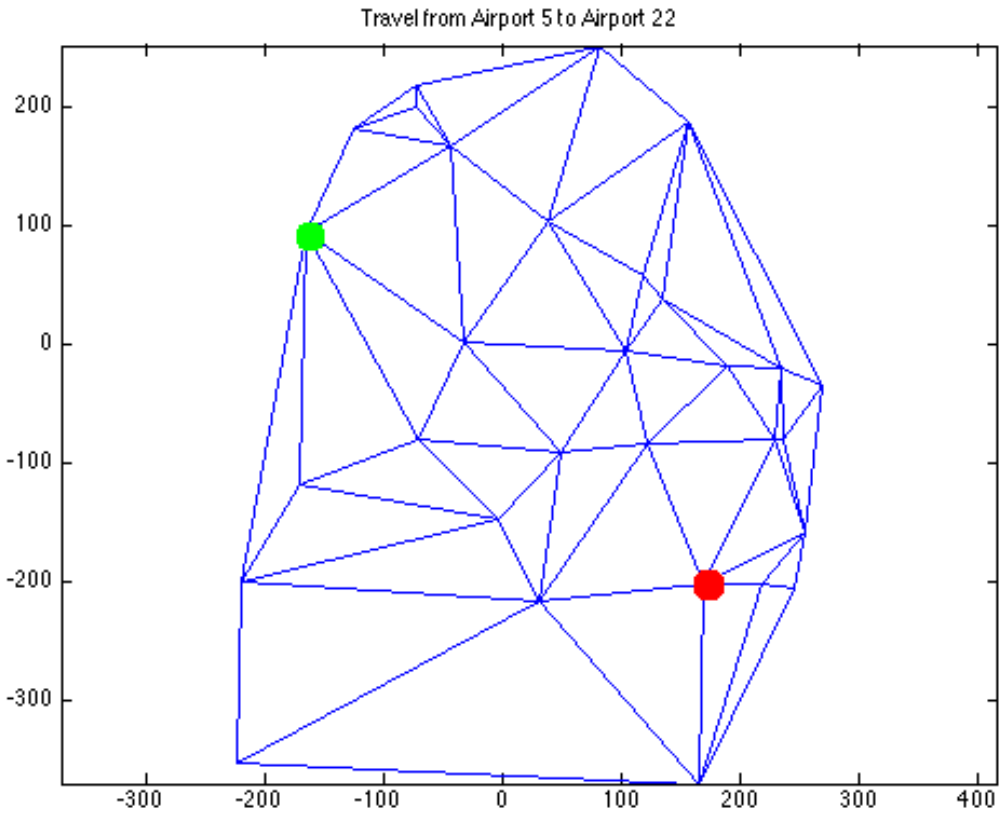

Figure 12: Start and End Node

Initially, the cost is calculated as the same cost to run Dijkstra. That is, the distance of each link being traveled. When this occurs, the cost is added to the title of the graph. For the actual cost to be calculated, the user needs to identify a particular scenario for the code to run. Once the user identifies the scenario, the Bailey Algorithm begins to process the paths. Each time the path cost is generated, the Algorithm applies a penalty value if necessary. The penalty remained the same for each scenario. It was based on the average of the vertical and horizontal spread of the data. If the scenario requires a penalty, then the value is added to the cost. However, if the scenario required a reward, the cost was subtracted from the cost.

For this example, Scenario 1, with a penalty for exceeding a certain elevation, will be shown. Running the same case as the example above, it is possible to see the initial cost:

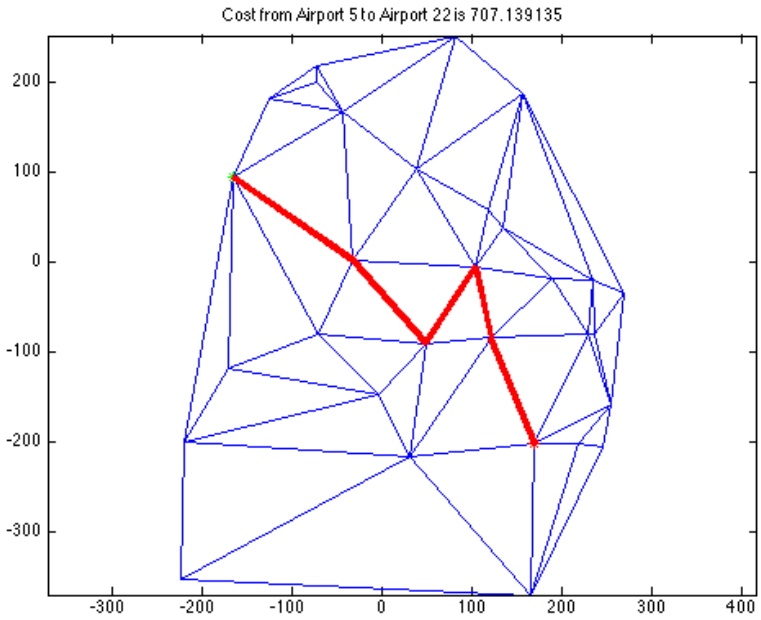

Figure 13: Initial Path

Once the initial cost is known, it is now possible to incorporate the Simulated Annealing into the code. Simulated Annealing is an optimization method that is used to the "global minimum of a function."[18] In this case, the minimum of the function is the path with the lowest cost to go from the starting city to the ending city. Simulated Annealing was established based on the metal annealing process.[19] In the annealing process, metals are heated and cooled repeatedly in an effort to make them more ductile, more homogenous, and more workable. With every heating and cooling cycle, the

[18] Jasbir Arora, *Introduction to Optimum Design*, (Boston: Elsevier, 2012), 630.
[19] Ibid.

temperature used to heat the metal is lowered. Likewise, Simulated Annealing works by establishing an initial "temperature" and then "cooling" it off slowly. In other words, large changes can be made initially at the high starting temperature. As the temperature is lowered, smaller changes are accepted. For this example, the initial cost generated by Dijkstra is 707. Thus, the initial temperature is 707.

The program runs for 1000 tries at this initial temperature. The way Simulated Annealing works is that it randomly selects a link to perturb. With each iteration, a random link is randomly bumped either left or right. Figure 14, below, shows a possible perturbation. The green arrows indicate the path is being bumped left and the orange arrows indicate a bump right.

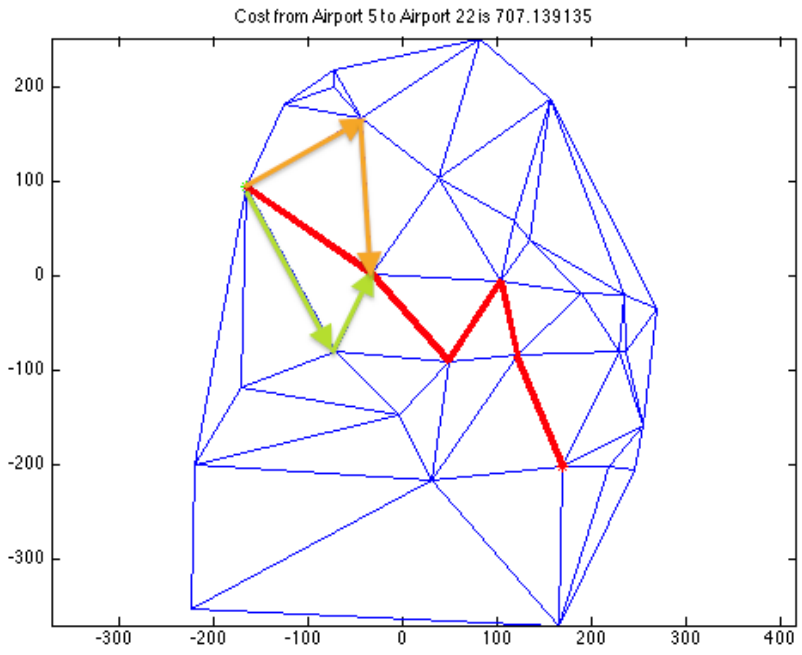

Figure 14: Path Perturbation

Once the segment has been bumped, a new path is created. This can be seen in Figure 15. In this case, the orange path is the new path.

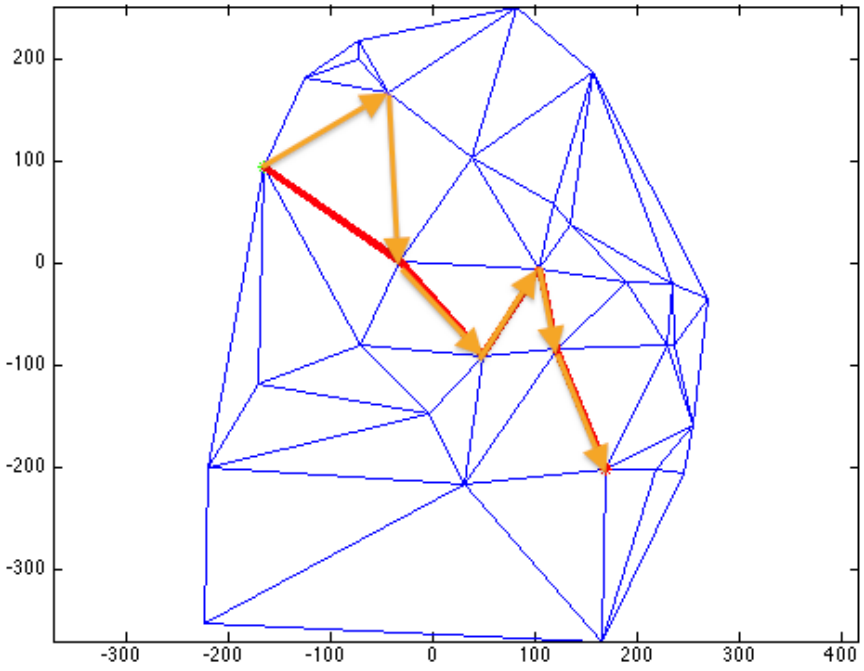

Figure 15: New Path

After generating the new path, the cost is recalculated. In order to recalculate the cost, it is first necessary to know the sum of the link distance. This provides the baseline cost. Added to this is the penalty value. The penalty is a function of area covered by the graph, it will remain constant for any scenario. Throughout this capstone, the penalty value was calculated based on the spread of the coordinate points. The distance was calculated between the extremes in both the vertical and horizontal directions, and then averaged. Once the average was known, it was then divided by 5 to provide the penalty value. However, when and how often the penalty is applied depends on the specific scenario being called. Table 2 briefly describes what penalty each scenario is associated with:

Scenario	Constraint Penalized
1	Elevation
2	Country Airspace, Hospital Proximity
3	Natural Features
4	Rate of Climb
5	Runway Length
6	Stick Fixed Neutral Point Location
7	Turn Direction

Table 2: Scenarios and Penalties

Clearly, enacting each scenario between the same initial and final cities will result in very different paths. Once the new path has been completely generated, it is time to either accept or reject it.

Acceptance of the path is done using the "Metrop" function. This function is based on the one provided in *Numerical Recipes in C.*[20] Essentially, Metrop looks for a path cost that is "not too much worse" than the previous path. To determine if this is true, Metrop looks at two possible equations:

$$old\ cost - new\ cost\ < 0$$

$$random\ number\ from\ 0-1\ < e^{\frac{-(old\ cost - new\ cost)}{temperature}}$$

In these equations, the "old cost" is the cost from the previous iteration, and the "new cost" is the cost for the current iteration. The "temperature" is determined based on the iteration. If the cost difference is deemed "not too much worse" then the path adjustment is accepted. After each iteration, the number of acceptances are recorded. At the end of each of 1000 tries worth of temperatures, if there are enough accepted paths then

[20] William T. Vetterling et al., *Numerical Recipes in C: The Art of Scientific Computing*, (New York: Cambridge University Press, 1992), 351.

the temperature is reduced by 10% and the 1000 tries are repeated for the new temperature. This process will continue for 100 iterations of temperature, or until there are no more accepted perturbations, whichever comes first.

At the end of the program, the best path will be shown, along with the cost. Continuing the example from Scenario 1, the following graph represents the best path to from Airport 5 to Airport 22:

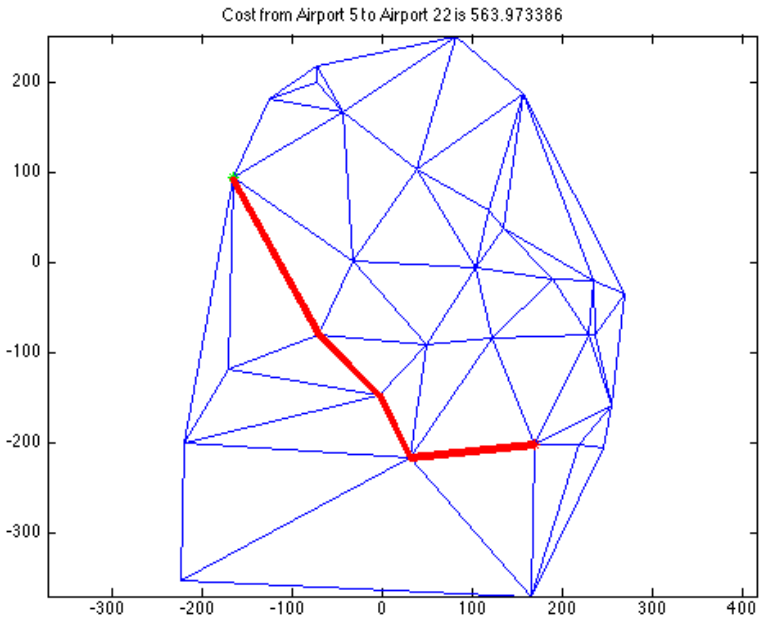

Figure 16: Best Path

Clearly, the cost has gone down, demonstrating that Simulated Annealing works. In the next chapter, an example will be done that shows the program can find an acceptable path, accounting for any penalties that may occur.

Chapter 4: Demonstration of the Bailey Algorithm

Overview

The Bailey Algorithm inputs the data from Excel into MATLAB so that it can select the appropriate values for each scenario. In Table 3, an excerpt from the Excel Sheet, it is clear to see the X and Y location of each airport with respect to the predetermined origin. The elevation is recorded in nautical miles. The fourth column uses logical values to designate the presence of a hospital. A value of zero means there is not a nearby hospital, and a value of one indicates there is a hospital close to that particular airport. This same identification convention is used to determine whether a particular airport is close to a natural feature. The runway length is the sixth column. For consistency, it is also in terms of nautical miles. The final column indicates the country the airport is in. To account for Scenario 2, this column also uses logical values to show whether or not the airport is in Uganda.

X Location	Y Location	Elevation	Hospital	Nat'l Feat.	RW Length	Uganda
-73.8678	200.3139	0.42971577	0	0	0.61058809	1
-74.1078	218.9417	0.5101949	0	0	0.70110654	1
-125.273	182.8261	0.65025163	0	0	0.9216424	1

Table 3: Constraint Variables and Values

In the following sections, each scenario will be briefly reintroduced, followed by the specific way the scenario affects the code. A series of graphs reflecting the path progression the scenario makes will be presented.

Scenario 1

This scenario describes an elevation constraint. A penalty is assessed when the plane passes through an airport at an elevation above 5,000 feet. In this particular Algorithm, mountains that are located between the nodes were not considered, but they could be added in during future work.

To show that the Bailey Algorithm is capable of accounting for an elevation constraint, the user chose the start node as 4 and the final node as 29. These two particular nodes are linked through node 10 and 11, both of which violate the constraint. By selecting these as an example, it is possible to see the evolution from a path with violations to one that adapts.

Path	Node
Start Node	4
Node of Violation	10, 11
Final Node	29

Table 4: Scenario 1

Once the user identifies the start and final nodes, the Bailey Algorithm starts running. In Figure 17, the initial path is shown to be: 4-8-9-30-11-29. For all scenarios, the initial cost is generated using Dijkstra's Algorithm within the Bailey Algorithm. However, the Bailey Algorithm generates the first path. This means that violations can occur. Unfortunately, this violates the constraint at both Airport 11 and Airport 29. These violations are shown by the yellow dots.

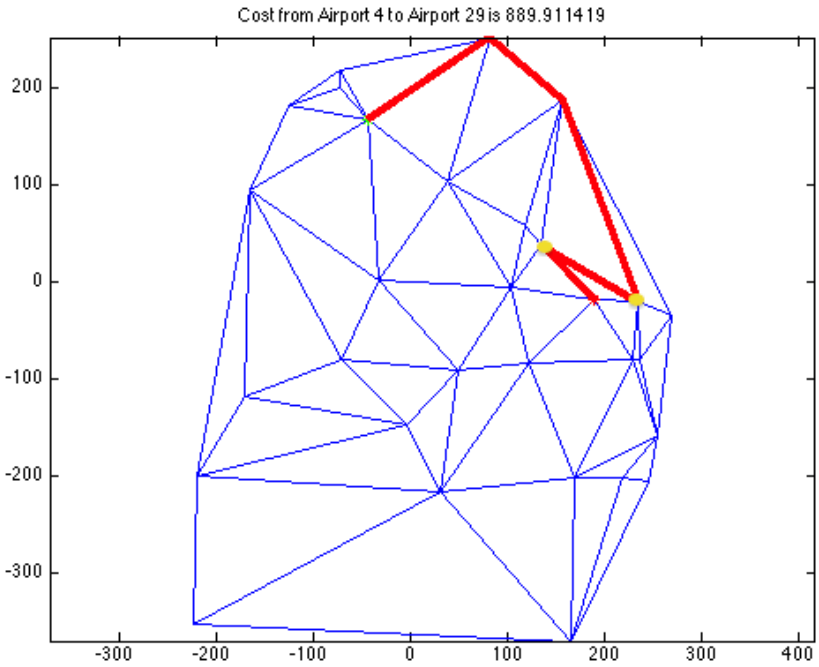

Figure 17: Scenario 1, Initial Path

To be able to show the intermediate steps, a "Pause" command was inserted when this Scenario was run. By doing this, the Bailey Algorithm paused after each graph was generated before continuing to run. This showed each new possible path slowly enough for the graphic images to be captured, like the one presented in Figure 18, where the path is 4-7-5-17-26-19-27-24-25-15-30-29. Clearly, there are still violations occurring. However, this shows Simulated Annealing's approach of accepting paths that are not "too much worse" before moving on.

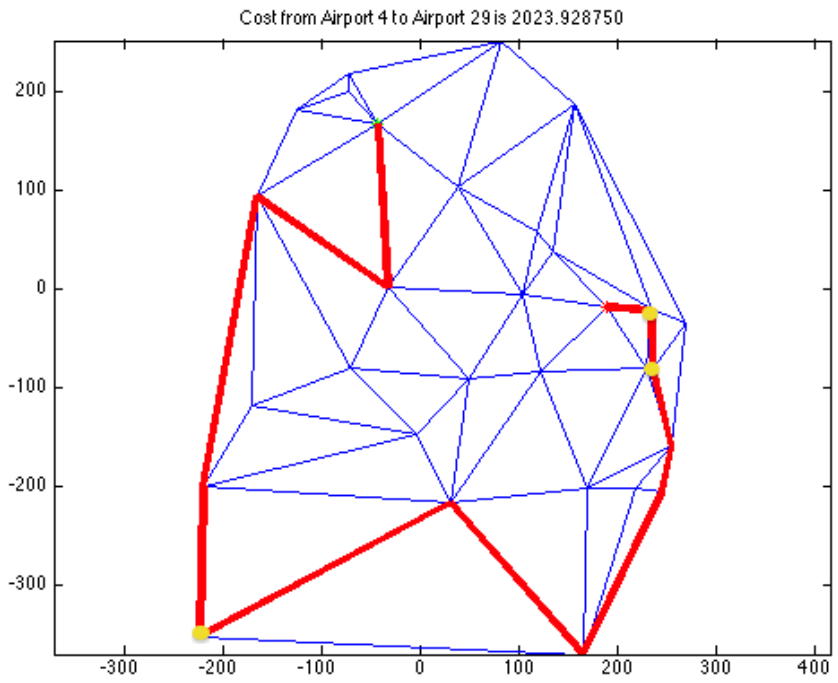

Figure 18: Scenario 1, Intermediate Path

Allowing the code to run to completion settles on the best path. The code was considered complete when enough temperature iterations had been run to prevent changes in the path. Not only does Figure 19 not have any constraint violations, but it also has a short distance resulting in a low cost.

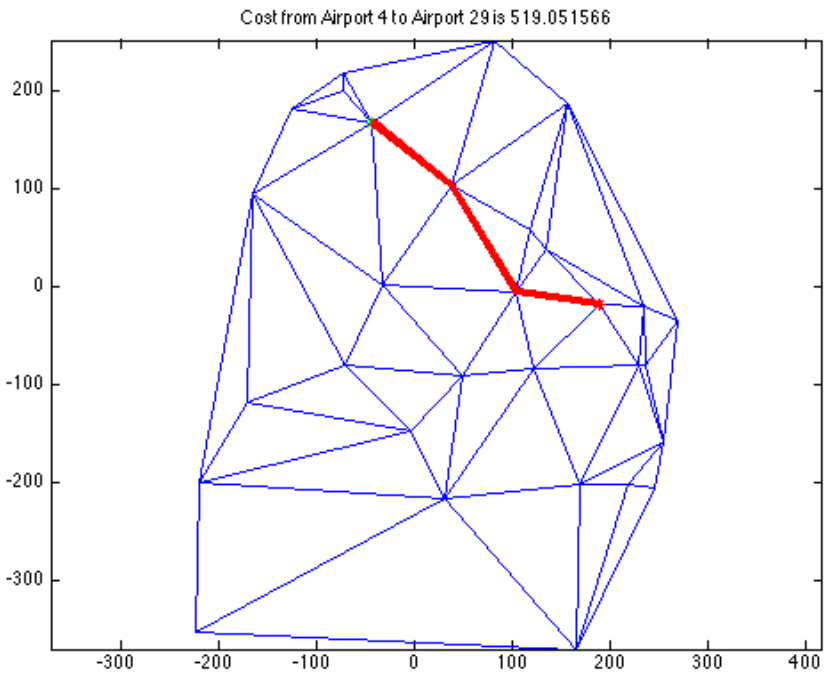

Figure 19: Scenario 1, Final Path

Scenario 2

In this scenario, there are two constraints: there must be an airport nearby, and the plane cannot cross through Uganda. Unlike the other scenarios, this one provides a reward instead of a penalty for passing through airports that satisfy both constraints. At this time, "no fly zones" that occurred between airports were not considered, but will be discussed in Chapter 6.

As with Scenario 1, the user selected the following start and end nodes, knowing the "Nodes of Violation" were likely to be part of the initial path.

Path	Node
Start Node	16
Nodes of Violation	1, 2, 3, 4, 6, 7
Final Node	9

Table 5: Scenario 2

To demonstrate this scenario, the path was charted from Airport 16 to Airport 9. Figure 20, below, shows the initial path as well as the airports in violation. With the initial path, the airports visited are 16-5-3-2-4-7-12-10-6-9. Of those visited, 3, 2, 4, 7, and 6 are in Uganda. Looking at the cost, as shown on the top of the graph, shows this was clearly an expensive path to take. The high expense comes from going through airports in Uganda.

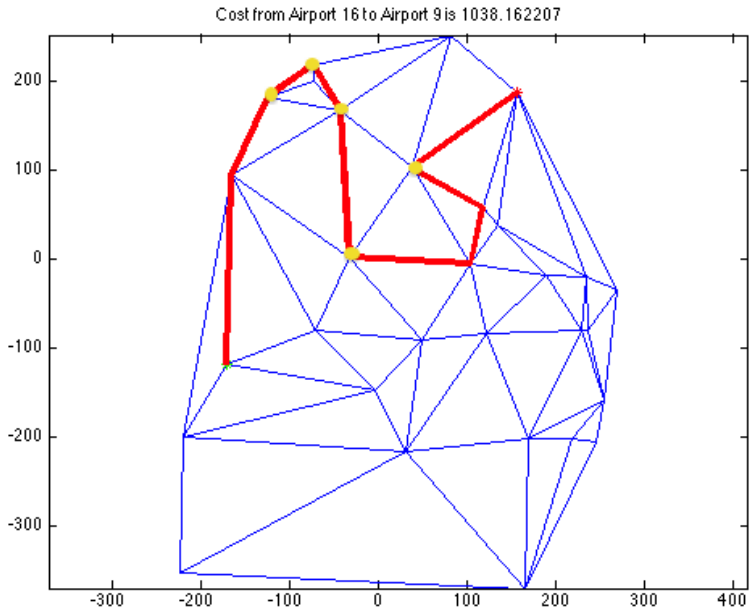

Figure 20: Scenario 2, Initial Path

As with Scenario 1, a pause command was inserted to allow for the graphs to appear slowly. Once the number of temperature iterations had become larger than 10, meaning the code had been running for a while, the following path was generated:

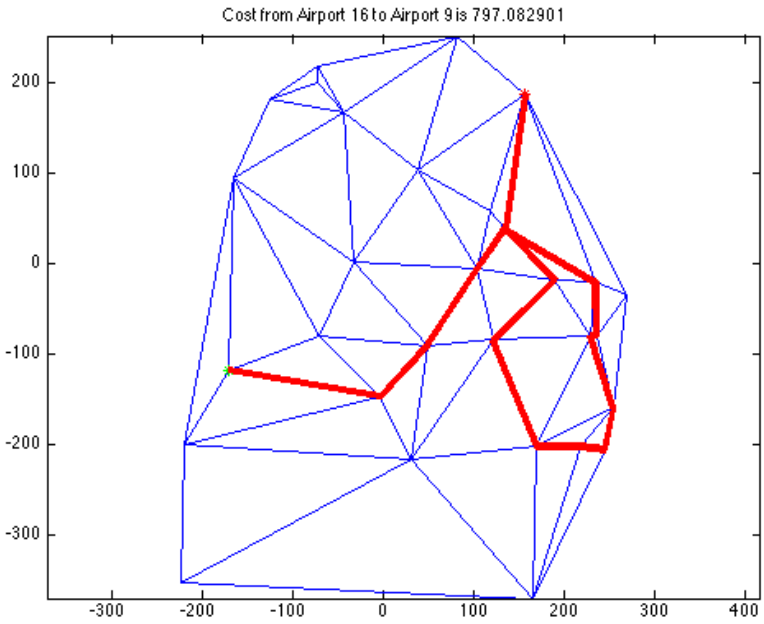

Figure 21: Scenario 2, Intermediate Path

In this solution, there are no airports in Uganda, but, this is a very expensive path in terms of distance traveled. The path shown is 16-18-20-12-11-30-15-25-24-23-22-13-29-11-9. As a viewer, it is clear to see that eliminating the loop would greatly shorten the distance traveled. As the Bailey Algorithm finished running and the temperature value decreased, the Algorithm removed the loop. The final path, going from 16-21-18-20-12-11-9, is shown in Figure 22.

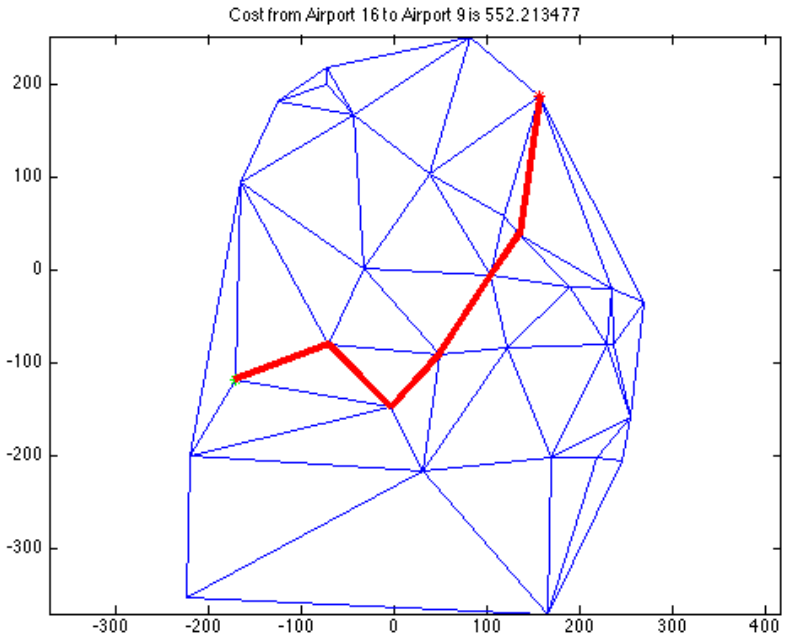

Figure 22: Scenario 2, Final Path

Scenario 3

In this scenario, the plane is penalized for flying over national parks or wildlife areas. Knowing which nodes would result in a violation, it was possible for the user to select a start and end node that would make a path with a high likelihood of containing a node of violation:

Path	Node
Start Node	9
Nodes of Violation	7, 12, 17, 19, 21, 22, 29
Final Node	26

Table 6: Scenario 3

In the initial path, 9-30-15-14-13-29-11-9-6-12-7-4-1-3-5-17-18-19-26, there were numerous violations. Some, such as 12, 7 and 18, were due to close proximity to Lake Victoria. The rest were National Parks.

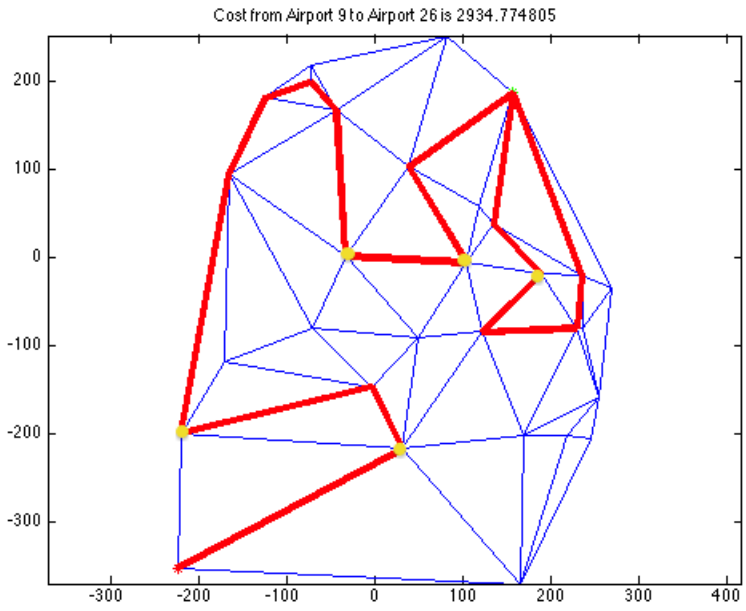

Figure 23: Scenario 3, Initial Path

As with the previous scenarios, once the temperature had been changed over 10 iterations, the Bailey Algorithm found a path that had been

updated to have a shorter distance traveled, but still had the same number of violations. Figure 24 shows this intermediate solution. The number of nodes of violation coupled with the loop the path takes shows that there are still improvements that can be made. The intermediate path contains the following airports: 9-6-7-5-4-6-12-29-13-14-22-27-19-26, as found by the Bailey Algorithm.

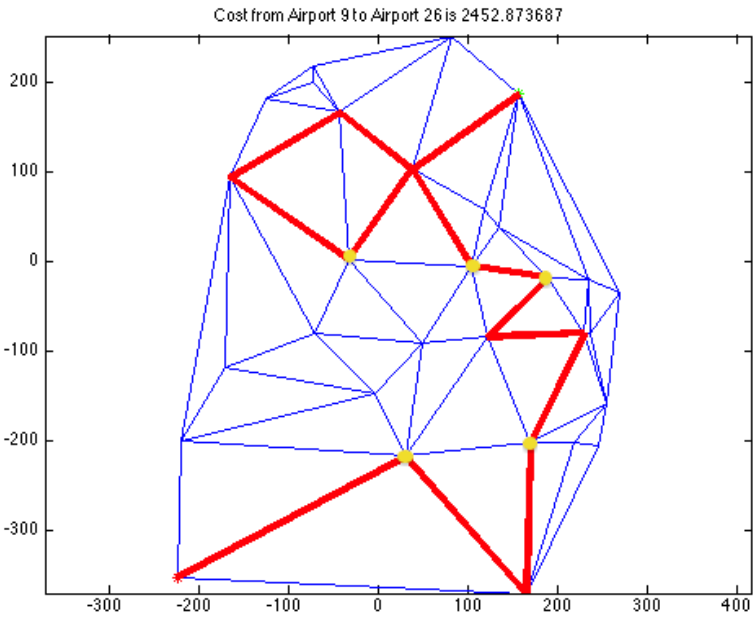

Figure 24: Scenario 3, Intermediate Path

As the Bailey Algorithm finished running, it found the "best" solution. This path, going from 9-30-15-25-24-27-26 does not contain any violations. It also is the shortest path, in terms of distance traveled.

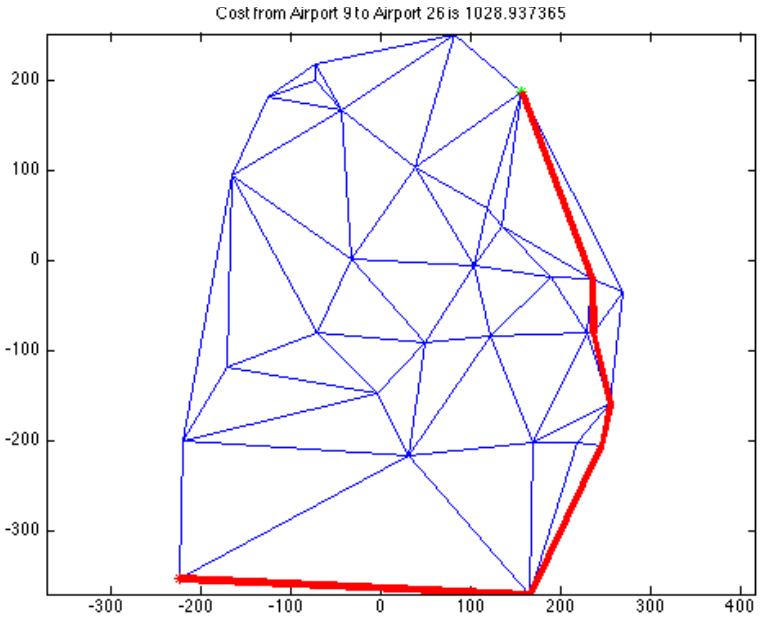

Figure 25: Scenario 3, Final Path

Scenario 4

In this scenario, one of the engines is damaged. This means that the plane is unable to ascend as quickly as it would normally. For this reason, the airports between the starting and ending node must not exceed 1.2 times the altitude of the airport before.

The following path was generated based on the user-defined start and end node:

Path	Node
Start Node	9
Node of Violation	LOTS
Final Node	19

Table 7: Scenario 4

Initially, this generated the following path. The links marked with an "X" are the ones that violate the constraint. For simplicity, a small chart follows Figure 26 to clearly show which links were violated. The violated links are highlighted in red. These two graphics reveal that there are four violations.

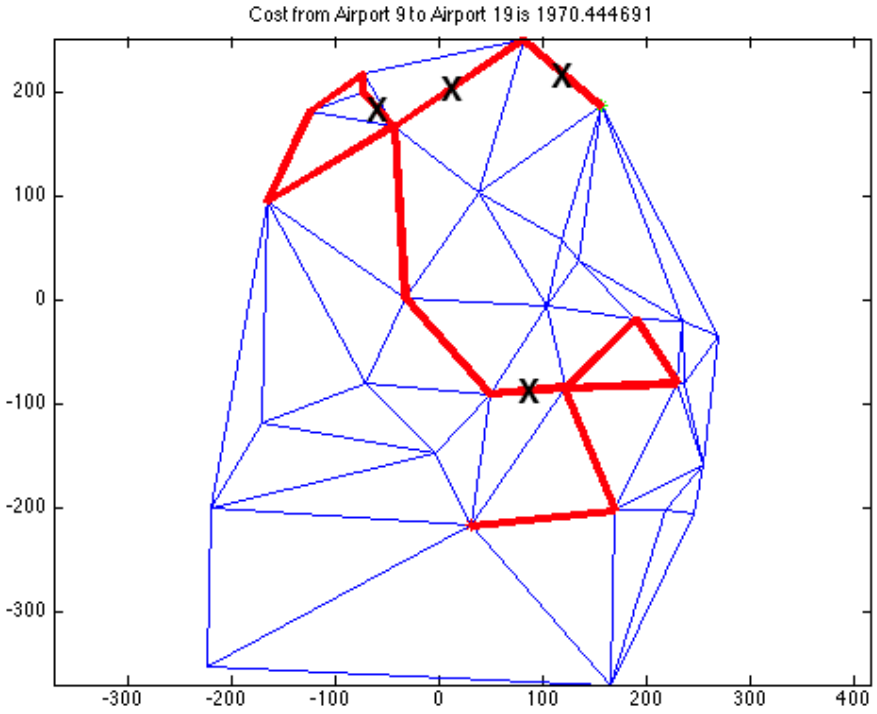

Figure 26: Scenario 4, Initial Path

Node	Percent Change in Altitude
9-8	1.23
8-4	1.6
4-5	1.15
5-3	0.98
3-2	0.78
2-1	0.84
1-4	1.30
4-7	1.08
7-20	1.00
20-13	1.48
13-14	0.99
14-29	1.11
29-13	0.90
13-22	0.74
22-19	0.91

Table 8: Scenario 4, Initial Links Violated

Similarly, there is an improvement between the initial path and the intermediate path. However, there are still three links that violate the constraint.

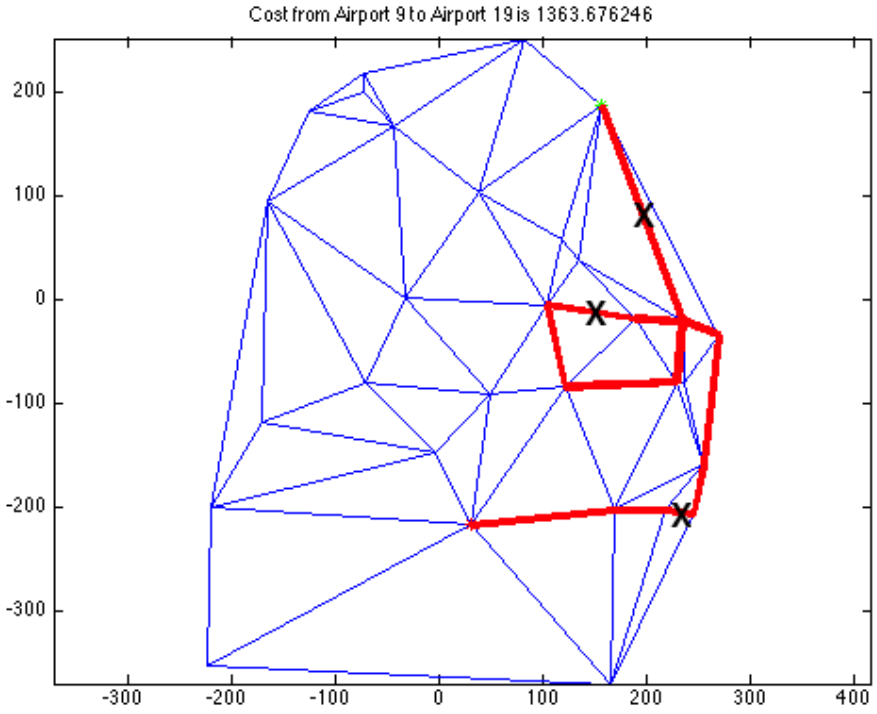

Figure 27: Scenario 4, Intermediate Path

Node	Percent Change in Altitude
9-30	3.40
30-14	0.95
14-13	1.00
13-12	0.67
12-29	1.60
29-30	0.94
30-28	0.71
28-25	0.91
25-24	0.78
24-23	1.55
23-22	0.91
22-19	0.92

Table 9: Scenario 4, Intermediate Links Violated

At the conclusion of the code, there is still a constraint that is violated. Even though this is the case, the cost is still extremely low. To fix this, a stronger penalty could be applied to the violation value.

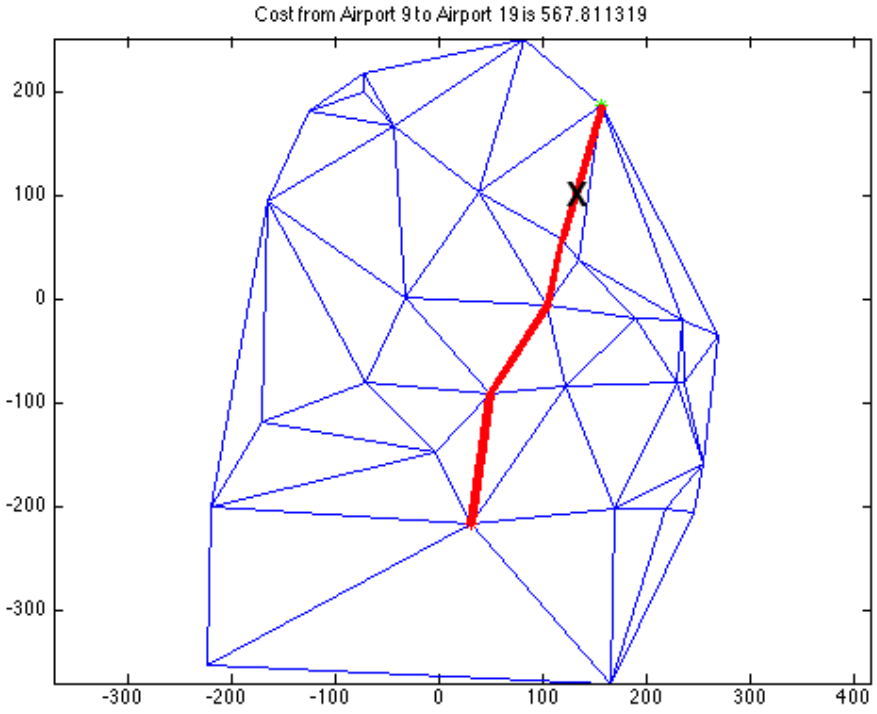

Figure 28: Scenario 4, Final Path

Node	Percent Change in Altitude
9-10	3.54
10-12	0.63
12-20	0.99
20-19	1.00

Table 10: Scenario 4, Final Links Violated

Scenario 5

In this scenario, the runway length provides the constraint that the Bailey Algorithm takes into consideration. In particular, the pilot must fly a route that contains intermediate nodes of at least 5,000 feet in order to land safely if (s)he cannot make it to the final destination.

The user selected the start and final node. Based on the initial path, it was apparent there were numerous nodes that violated this constraint along the path. The following path was chosen to demonstrate the effectiveness of the code:

Path	Node
Start Node	1
Node of Violation	2, 6, 9, 10, 13
Final Node	11

Table 11: Scenario 5

In this case, the initial path was 1-4-8-6-9-11. This caused a violation of airport 6 and 9, as seen with the yellow dots in Figure 29.

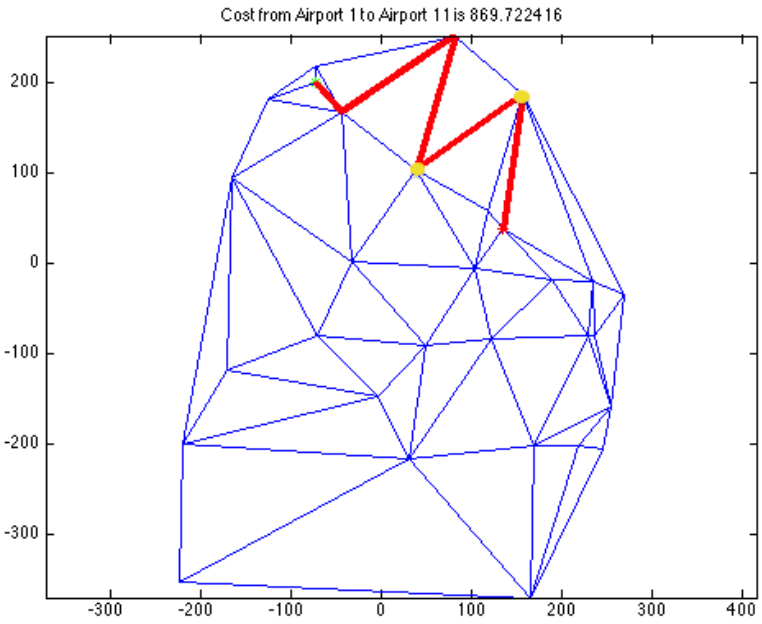

Figure 29: Scenario 5, Initial Path

As with the other scenarios, the path underwent many modifications and perturbations. Approximately halfway through the temperature iterations, the following path was generated before being rejected. This path was an improvement over the initial, with only one violation:

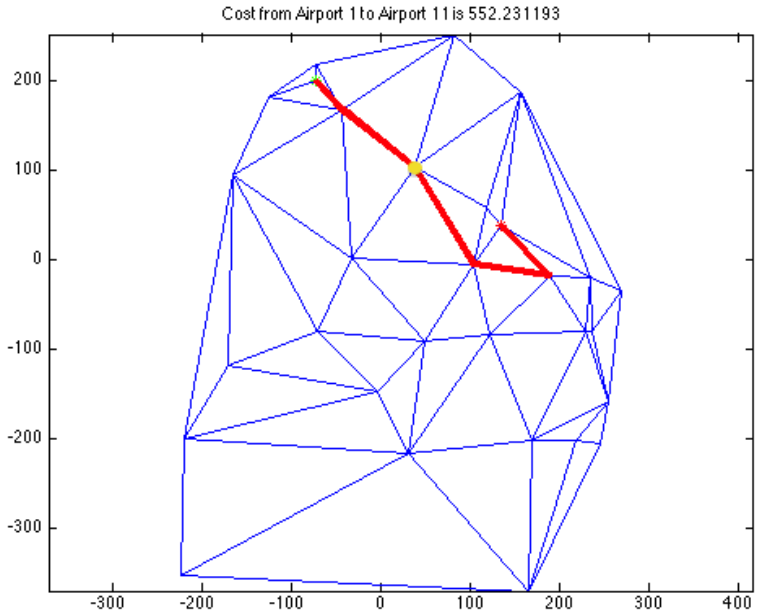

Figure 30: Scenario 5, Intermediate Path

Looking at the final path, we see that the overall cost did not decrease by very much. This is because to avoid the penalty associated with the city, the path had to extend a little longer.

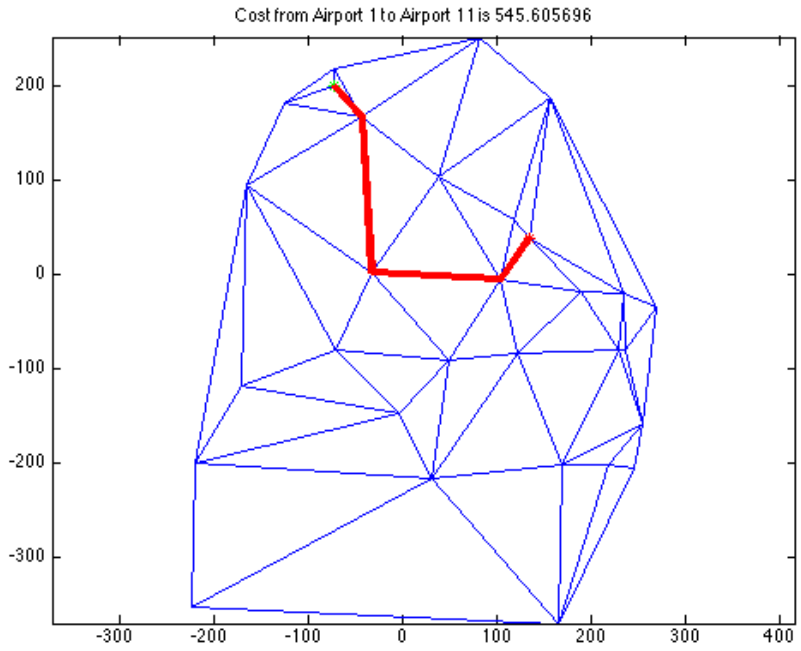

Figure 31: Scenario 5, Final Path

Scenario 6

In this scenario, the plane is impaired in its turning ability. After being hit by debris, the rudder jammed, causing the plane to be unable to turn left.

Initially, the path contained numerous left turns, resulting in a very high initial cost. As with Scenario 1, the user selected the following start and end nodes, knowing the "Nodes of Violation" were likely to be part of the initial path.

Path	Node
Start Node	5
Nodes of Violation	LOTS
Final Node	24

Table 12: Scenario 7

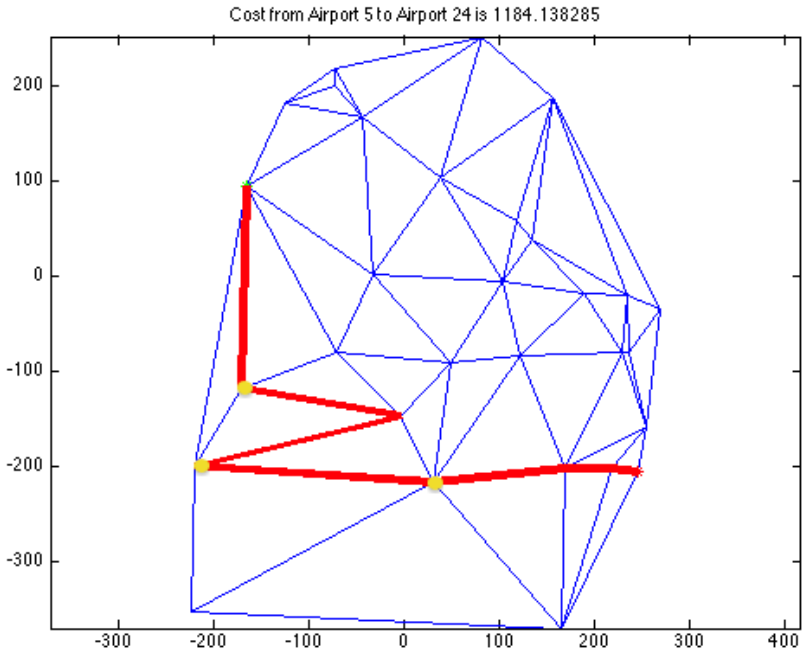

Figure 32: Scenario 7, Initial Path

Once the program had been running for a while, the intermediate path was created. As with some of the other examples, this was an example where a path was suggested as it was "not too much worse." In this example, it is

clear that the cost is so high due to the high contribution from the penalty function – it makes up nearly 1/3 of the total cost.

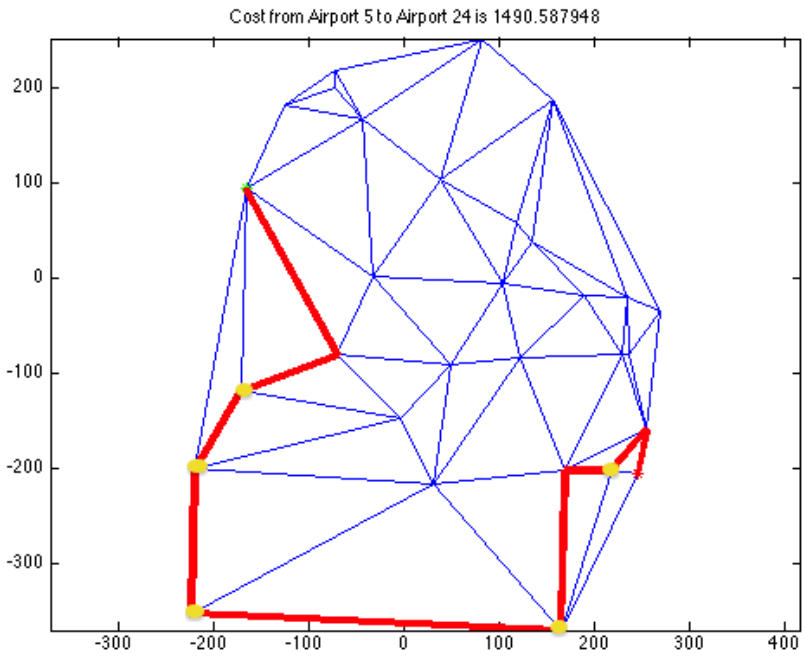

Figure 33: Scenario 7, Intermediate Path

At the final path, there are still two left turns. Like the intermediate step, they account for approximately of the total cost. In a future version of this code, this could be corrected by adding a more severe penalty. Having said that, only one of the turns is an extreme turn. It is possible that the slight left was the result of the turn being made that was almost 360 degrees to the right, resulting in a slight left.

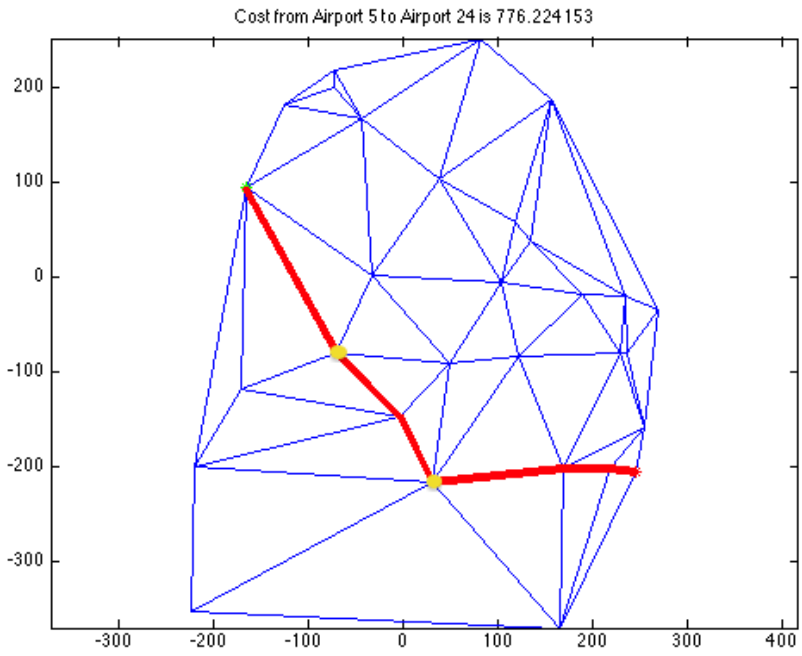

Figure 34: Scenario 7, Final Path

Chapter 5: Conclusions

As was explained in earlier sections, the main aspect that differentiates the Bailey algorithm from all other mapping algorithms is the ability to assign a penalty value to the intermediate nodes. By incorporating this penalty, the Bailey algorithm sets itself apart from the shortest path algorithms. Other algorithms, like those in GPS units, only look ahead. They do not take into consideration how one got to their current location. Some GPS units are able to account for traffic that might occur along the way, showing that there is some existing software similar to the Bailey Algorithm.

This very fact is what makes the Bailey algorithm matter in real life. This algorithm allows for paths to be charted that satisfy specific constraints at all points along the path. For example, if the plane needs a runway of a certain length in order to land safely, the route it flies should be populated with airports along the way that it could land at, just in case it cannot make it to the final destination.

Works for Any Arrangement of Cities

This program is just a model. It will work for any airplane, almost any scenario (See Scenario 7), and any collection of cities.

To show that the code will work for any arrangement of cities, the random number generator in MATLAB was used, generating X and Y coordinates for "Airports" as well as a random assortment of elevations. The resulting graph, including the Delaunay links, is shown in Figure 35:

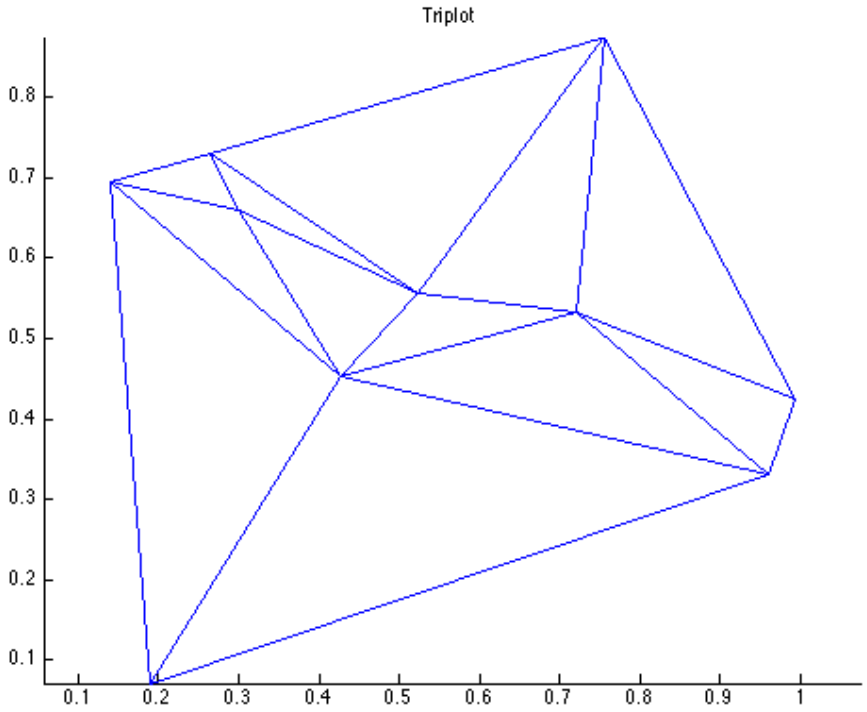

Figure 35: Random City Arrangement

These cities were run through Scenario 1, a complete table of values used is included in Appendix A4. As with the earlier scenarios, an initial path was generated:

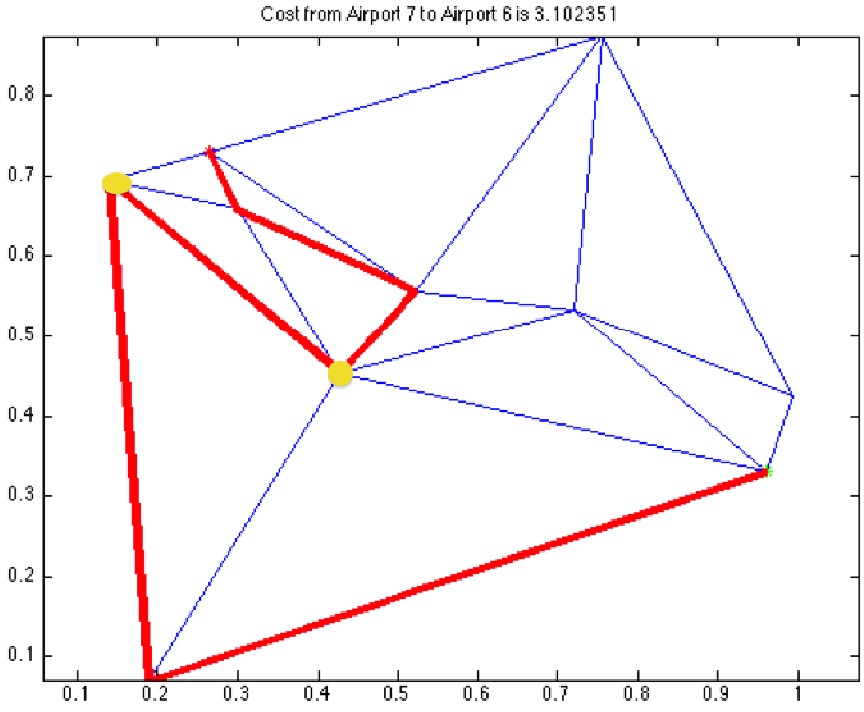

Figure 36: Initial Path

This initial path has two violations: first at node 10 and then node 8. As the code runs, it eliminates one of the violations, resulting in the intermediate graph shown in Figure 37, below.

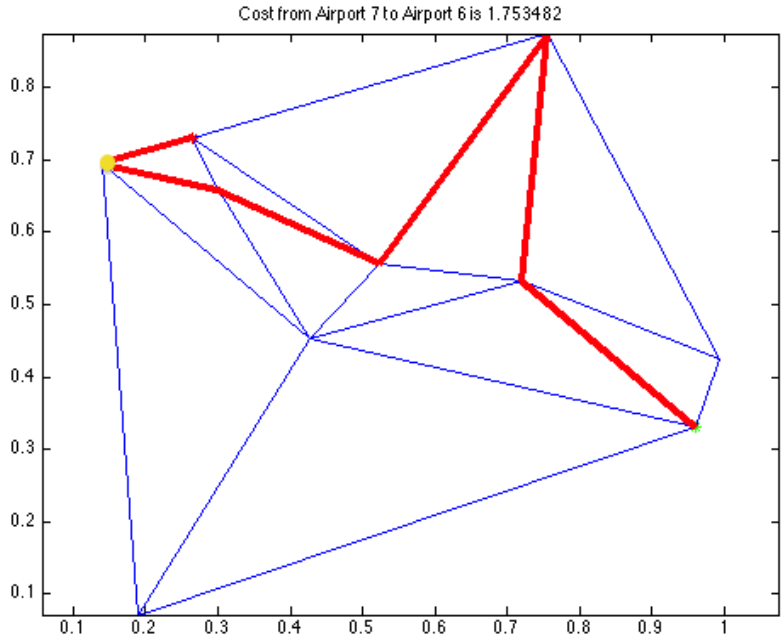

Figure 37: Intermediate Step

However, there is still the violation. This is corrected by the time the code finishes. The final path solution is displayed below in Figure 38:

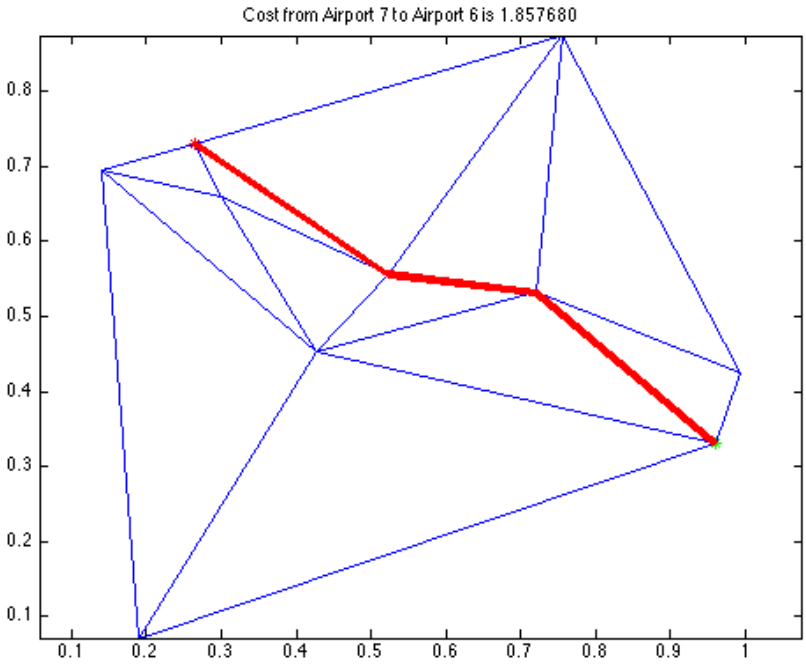

Figure 38: Final Path

This shows that this code is easily adaptable for different city arrangements. Not only is the origin changed, but the number of cities used is also different. In addition, the scale being tested is dramatically different.

Simulated Annealing is a Robust Optimization Method

While this project produces viable results, how can one be sure the results are consistent? By determining how robust the Simulated Annealing method is, this question can be answered. If the Simulated Annealing method is robust, it means that the optimizer is a good one that can stand up to scrutiny. A weak optimizer will report many different answers for the same scenario. It is possible for Simulated Annealing to return different values for different runs. As mentioned earlier, the Annealer randomly perturbs a link for each iteration. This randomness can result in slightly different costs being produced for the same start and end node.

To demonstrate how robust the Simulated Annealer is, the code was run ten times for Scenario 1, from airport 28 to airport 17. The complete data table is in Appendix A4. The result is shown in Figure 39. In this image, each run is graphed against the score it generated. There are two values for the final cost that were repeated multiple times: 662 and 915. This shows that the method of simulated annealing can often find the local minimum, but occasionally struggles to find the global minimum. Since the cost accounts for the distance traveled as well as the penalty applied at each node, it is possible for slightly different paths to be generated. Additionally, if a change is not made initially, the scaling factor that determines what changes are allowed might be reduced, resulting in a slightly higher cost path.

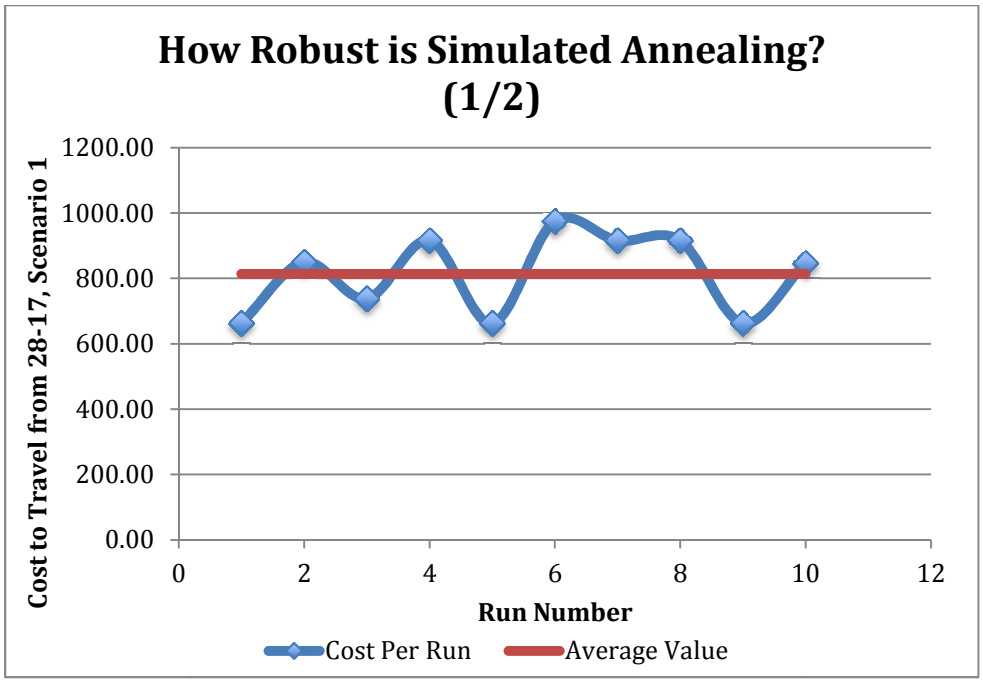

Figure 39: Simulated Annealing Robust Test (1/2)

Referring to Appendix A4 for the full table, the standard deviation is revealed to be approximately 121 nmi. Since the average is calculated to be around 800 nmi, a standard deviation of 121 nmi is only about 15% of the data spread. Relatively speaking, this is not too large of a data spread. While this shows that the simulated Annealer is not perfect, it also shows that the Simulated Annealing method is acceptable for the majority of the time.

Furthermore, closely examining Figure 39 reveals that there are two local minima that the Simulated Annealer focused on. The value of 662 and 915 both occurred multiple times. This shows that the annealer is settling on a solution, but has not quite reached it yet. Adjusting the temperature scale factor could allow for the annealer to settle on a more consistent number.

To provide another set of data, Scenario 5 was tested as well. For this test, the start node was selected to be 17 and the final node was 8. This

would force the code to avoid nodes 21 and 6, as they both would violate the constraint of runway length.

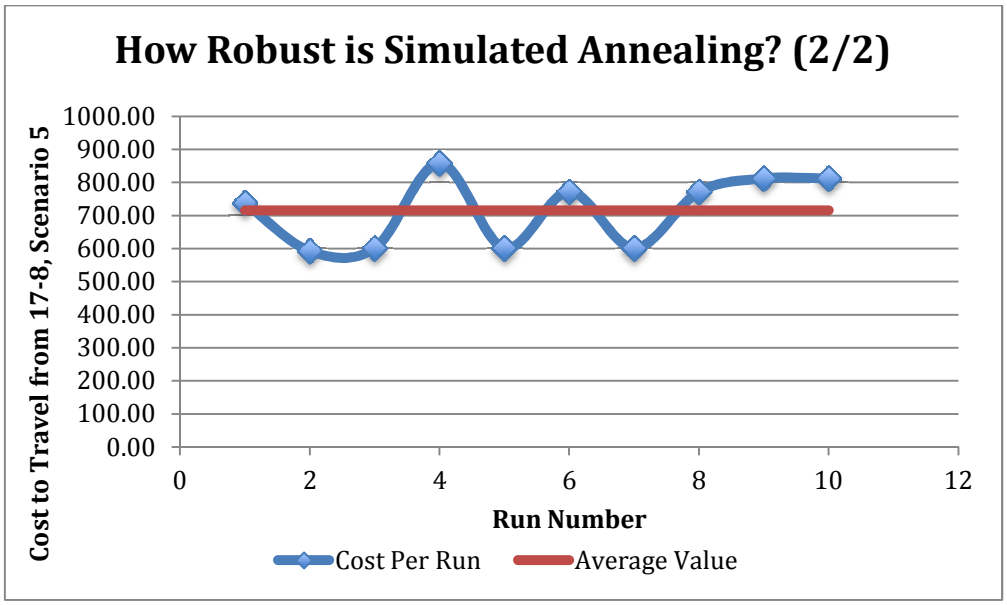

Figure 40: Simulated Annealing Robust Test (2/2)

In this test, the standard deviation was calculated to be about 106 nmi, again showing that the Simulated Annealer is robust. Like the results from Scenario 1, the Simulated Annealing optimizer found the same local minima multiple times. As mentioned previously, the results could be improved by adjusting the temperature scale factor. Based on these two conclusions, it can be assumed that there would be similar results for the other scenarios.

Penalty Value is Acceptable

When calculating the penalty value, it was important not to hard code in a value. If a set value was hard coded in, the results would be very different if all the airports were located between zero and one, compared to ones that might go from zero to one hundred.

To calculate the penalty value, the average of the vertical distance and horizontal distance covered by the data was taken. This number was then divided by 5, to allow for situations to be "not too much worse." This resulted in a penalty value of about 111 for each scenario. This made the penalty value a function of the data spread, allowing it to be transferred to any set of data.

To test the significance of the penalty value, five different values were chosen. Initially, it was thought that the values tested would go up to ten, but upon running the code, it became apparent that a value larger than 3 resulted in the Simulated Annealing deciding all paths were too expensive, making the first guess always the accepted path.

Penalty Values Tested
0.2
0.5
1
2
3

Table 13: Coefficients of Penalty Values Tested

The code was run numerous times at each penalty value. The cost obtained was then averaged out and graphed against the penalty value. To

show the effect of cost, the same two scenarios were run: Scenario 1 from 28-17 and Scenario 5 from 17-8. It was interesting to note that when the penalty value reached a certain point, it was too high for the Simulated Annealer to consider. This resulted in the Annealing process only being completed for one temperature iteration.

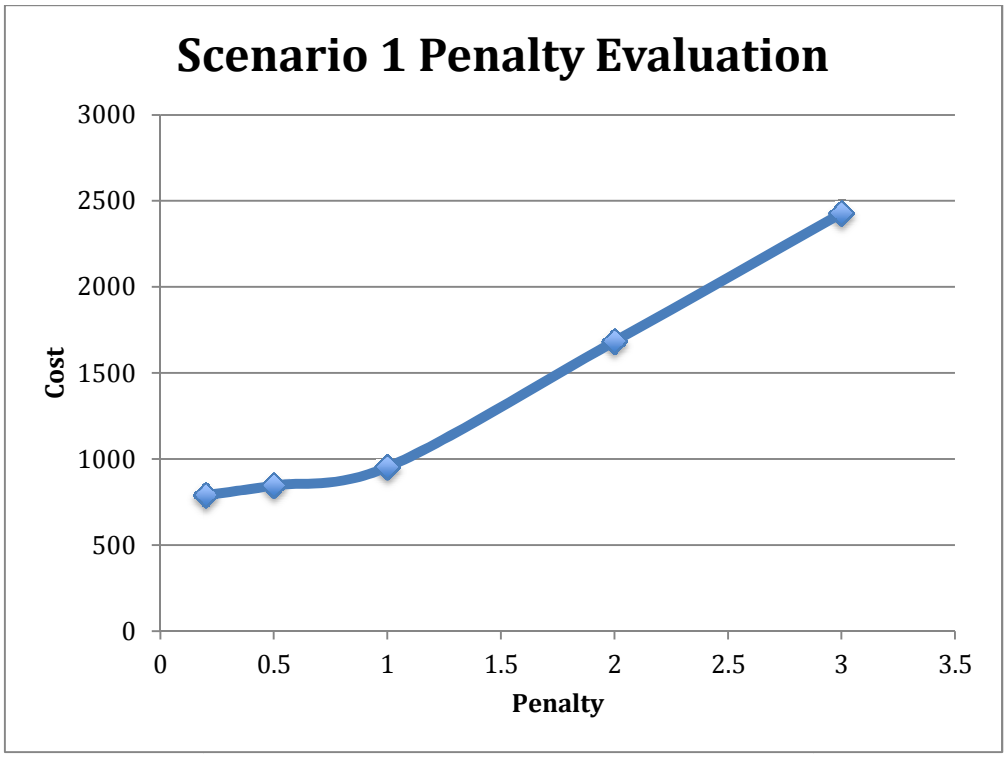

Figure 41: Penalty Value for Scenario 1

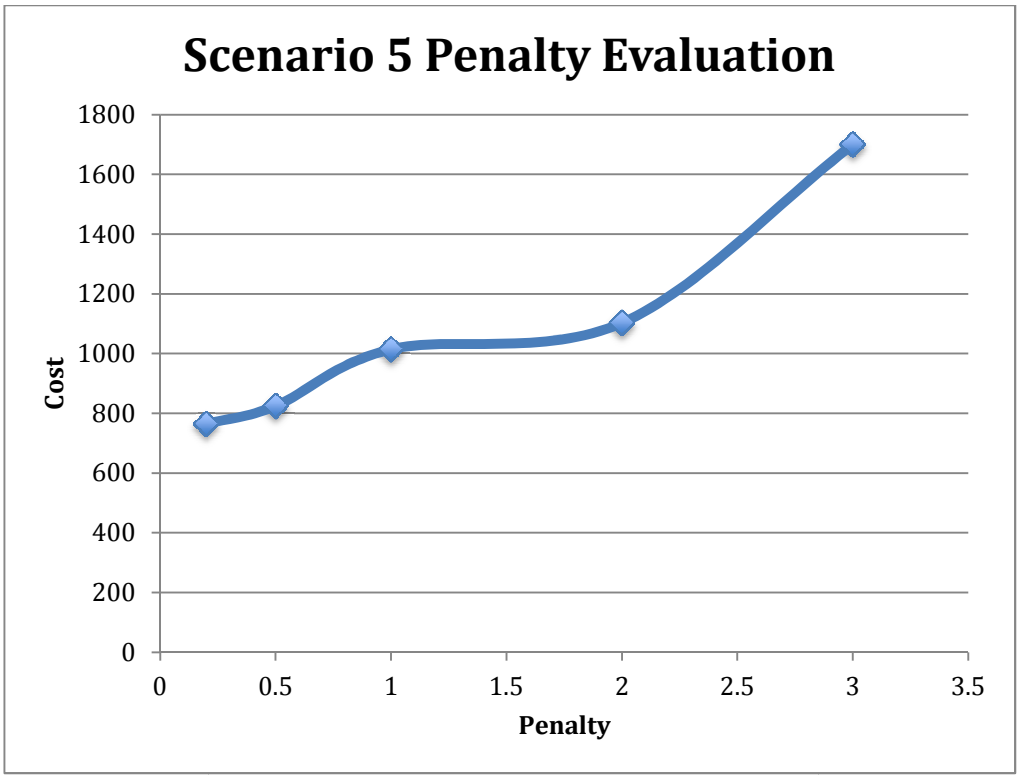

Figure 42: Penalty Value for Scenario 5

Chapter 6: Future Work

It became apparent that this could be extended much more than the work that was completed. Alterations and extensions could be made to the scenarios, more features could be included, and the code could be modified to be more accommodating.

With regards to specific scenarios, in the future, modifications could be made to account for Scenario 6, the change in the Stick Fixed Neutral Point, and center of gravity. When the SFNP shifts, there are serious consequences. At this point, there is not a qualitative way to account for this shift, and therefore it is unable to be viable at this time.

A scenario could also be added to account for runway direction for towered airports. This could provide the plane with a penalty for taking off and landing a specific way, based on the wind or the direction the plane is unable to turn. Furthermore, the code could be altered to account for flight patterns at each airport. This modification could also be adapted to include a constraint based on runway surface.

In the future, the code could be modified to incorporate aerodynamic characteristics of a specific plane. For example, a scenario could be added that would account for a malfunction with the Rate of Climb ability. This would require knowing the cruise speed for the plane to fly and take into consideration the distance between nodes.

More features could be added to the data MATLAB inputs. This could include mountains located in the middle of links, rather than right at an

airport or “no fly zones” that would prohibit flying in a certain area. Similarly, it could be made so that links are prohibited from crossing over Lake Victoria. Both of these could be done by dropping the respective links from Dijkstra, but it would provide more of a challenge, and make the program more generic, to have MATLAB identify the paths as problematic and remove them.

Furthermore, the code could be adapted to include a “severity” of the penalty. For example, turns that only go “a little left” are not penalized as heavily as sharp, 90 degree left turns. It could also allow for a severity in the other scenarios as well. Perhaps for Scenario 3, it is worse to fly over Lake Victoria than it is to fly over a National Park. This could also be applied to the scenario with hospitals. The closer the airport is to the hospital, the less severe the penalty is.

The last piece of notable future work would be to apply this code to completely different scenarios. For example, it could be incorporated into GPS units for charting the best path home in rush hour traffic, knowing certain stops have to be made along the way. Similarly, the Bailey Algorithm could be used for delivery vehicles to find the best way to deliver food or packages. It could also be used for everyday activities like completing scavenger hunts, or charting out what classes to take when.

Sources Cited and Consulted

"Airports in Africa." *Megginson Technologies, Ltd.* Updated 2009. http://www.ourairports.com/continents/AF/#lat=6.315298538330033,lon=17.578125,zoom=2,type=Map,airport=DGAA,continent=AF.

"Airports in Kenya." *Air Broker Center International AB*. Updated 2009. http://www.aircraft-charter-world.com/airports/africa/kenya.htm.

Arora, Jasbir. *Introduction to Optimum Design*. Boston: Elsevier, 2012.

Cornell University, "Dijkstra's Shortest Path Algorithm." Accessed April 21, 2014. http://www.cs.cornell.edu/courses/cs312/2002sp/lectures/lec20/lec20.htm.

Dannenhoffer, III, John. "Capstone Meeting: January, 23." Capstone Meeting, MAE 499: Honors Capstone Project from Syracuse University, Syracuse, NY, January 23, 2014.

"DC-3: The Genesis of a Legend." *DC-3/Dakota Historical Society, Inc.* Accessed March 26, 2014. http://www.dc3history.org/dc3.htm.

"Delaunay Triangulation." *MathWorks Inc.* 2014. http://www.mathworks.com/help/matlab/math/delaunay-triangulation.html.

Lester, Patrick. "A* Pathfinding for Beginners." *Policy Almanac*. July 18, 2005. http://www.policyalmanac.org/games/aStarTutorial.htm.

"Logan Plans to Add 600-Foot Runway Safety Area on Harbor Deck." *Boston Globe*, March 18, 2009. Accessed February 15, 2014. http://www.boston.com/news/local/breaking_news/2009/03/logan_plans_to.html.

"Model Railroad Layouts: Airport Runways and Accessories." *Bakatronics LLC.* Accessed February 15, 2014. http://www.bakatronics.com/shop/category.aspx?catid=117.

Nicholson, Roger and William Reed, "Strategies for Prevention of Bird-Strike Events," *Aero Quarterly*, Quarter 3: 2011.

U.S. Department of Health and Human Services. "Toxicological Profile for Hydraulic Fluids." Accessed April 22, 2014. http://www.atsdr.cdc.gov/ToxProfiles/tp99.pdf.

Vetterling, William T., William H. Press, Saul A. Teukolsky, and Brian Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* New York: Cambridge University Press, 1992.

Wachman, Monica. "What is the Altitude of a Plane in Flight?" *Travel Tips*. Accessed April 21, 2014. http://traveltips.usatoday.com/altitude-plane-flight-100359.html.

Appendix A—Excel Spreadsheets

Appendix A1: Runway Information

			ELEVATION					LENGTH				
AIRPORT	CODE	LOCATION	FEET	nmi	MILES	METERS	COORDINATES	FEET	nmi	METERS	SURFACE	DIRECTION
Adjumani Airport	HUAJ	Adjumani, Uganda	2611	0.43	0.495	796	03°20'19"N, 31°46'08"E	3710	0.611	1130	Unpaved	09/27
Moyo Airport	OYG	Moyo, Uganda	3100	0.51	0.587	940	03°38'57"N, 31°45'54"E	4260	0.702	1300	Unpaved	02/20
Arua Airport	RUA	Arua, Uganda	3951	0.65	0.748	1204	03°02'50"N, 30°54'44"E	5600	0.922	1700	Unpaved	18/36
Gulu Airport	ULU	Gulu, Uganda	3510	0.58	0.665	1070	02°48'00"N, 32°16'30"E	10314	1.699	3144	Asphalt	17/35
Bunia Airport	BUX	Orientale, Congo	4045	0.67	0.766	1233	01°33'57"N, 30°13'15"E	6070	1	1850	Ice	14/32**
Soroti Airport	SRT	Soroti, Uganda	3641	0.6	0.69	1110	01°43'30"N, 33°37'16"E	2525	0.416	770	Tarmac	09/27
Entebbe International Airport	EBB	Kampala, Uganda	3782	0.62	0.716	1153	00°02'41"N, 32°26'35"E	12000	1.976	3658	Asphalt	17/35
Lokichoggio Airport	LKG	Lokichoggio, Kenya	2116	0.35	0.401	645	04°12'18"N, 34°20'42"E	6194	1.02	1888	Asphalt	03/21**
Lodwar Airport	LOK	Lodwar, Kenya	1715	0.28	0.325	523	03°07'20"N, 35°36'36"E	3281	0.54	1000	Asphalt	09/27
Kitale Airport	KTL	Kitale Kenya	6070	1	1.15	1850	00°58'30"N, 34°57'36"E	4757	0.783	1450	Asphalt	22/04
Eldoret International Airport	EDL	Eldoret, Kenya	7050	1.16	1.335	2150	00°24'16"N, 35°14'20"E	11400	1.877	3475	Asphalt	08/26
Kisumu Airport	KIS	Kisumu, Kenya	3796	0.63	0.719	1157	00°05'10"S, 34°43'44"E	10826	1.783	3300	Asphalt	06/24
Mara Serena Lodge Airstrip	MRE	Masai Mara, Kenya	5600	0.92	1.061	1707	01°24'18"S, 35°00'36"E	2700	0.445	820	Unpaved	04/22**
Nairobi Wilson Airport	WIL	Nairobi, Kenya	5546	0.91	1.05	1690	01°19'12"S, 36°48'54"E	5052	0.832	1540	Asphalt	14/32
Jomo Kenyatta Int'l Airport	NBO	Nairobi, Kenya	5327	0.88	1.009	1624	01°19'09"S, 36°55'39"E	13507	2.224	4117	Asphalt	06/24
Kigali International Airport	KGL	Kigali, Rwanda	4891	0.81	0.926	1491	01°57'59"S, 30°07'59"E	11482	1.891	3500	Paved	10/28
Bujumbura International Airport	BJM	Bujumbura, Burundi	2582	0.43	0.489	787	03°19'26"S, 29°19'07"E	11811	1.945	3600	Asphalt	17/35
Mwanza Airport	MWZ	Mwanza, Tanzania	3763	0.62	0.713	1147	02°26'40"S, 32°55'57"E	10827	1.783	3300	Asphalt	12/30
Shinyanga Airport	SHY	Shinyanga, Tanzania	3800	0.63	0.72	1158	03°36'34"S, 33°30'15"E	6562	1.081	2000	Gravel	11/29
Musoma Airport	MUZ	Musoma, Tanzania	3783	0.62	0.716	1153	01°30'10"S, 33°48'08"E	5248	0.864	1600	Grass	18/36
Bukoba Airport	BKZ	Bukoba, Tanzania	3766	0.62	0.713	1148	01°19'56"S, 31°, 49'16"E	4921	0.81	1500	Gravel	13/31
Lake Manyara Airport	LKY	Lake Manyara Nat'l Park, Tanzania	4150	0.68	0.786	1265	03°22'33"S, 35°49'06"E	4003	0.659	1220	Asphalt	12/30
Arusha Airport	ARK	Arusha, Tanzania	4550	0.75	0.862	1387	03°22'00"S, 36°37'19"E	5315	0.875	1620	Asphalt	09/27
Kilimanjaro International Airport	JRO	Hai District, Tanzania	2932	0.48	0.555	894	03°25'46"S, 37°04'28"E	11811	1.945	600	Asphalt	09/27
Amboseli Airport	ASV	Amboseli National Park, Kenya	3757	0.62	0.712	1145	02°38'32"S, 37°15'00"E	3284	0.541	1001	Asphalt	15/33**
Kalemie Airport	FMI	Katanga, Congo	5741	0.95	1.087	1750	05°52'32"S, 29°15'00"E	5741	0.945	1750	Asphalt	06/24
Dodoma Airport	DOD	Dodoma, Tanzania	3637	0.6	0.689	1109	06°10'13"S, 35°44'58"E	6699	1.103	2042	Asphalt	10/28
Embu Airport	HKEM	Embu, Kenya	4150	0.68	0.786	1265	00°34'08"S, 37°29'32"E	2953	0.486	900	Asphalt	08/26**
Nakuru Airport	NUU	Nakuru, Kenya	6200	1.02	1.174	1900	00°18'00"S, 36°09'36"E	5607	0.923	1709	Asphalt	16/34**
Nyeti Airport	NYE	Nyeri, Kenya	5830	0.96	1.104	1777	00°20'24"S, 36°54'36"E	4050	0.667	1230	Unpaved	02/20**

Appendix A2: Complete Data

ID	Latitude	Longitude	LatDeg	LatMin	LatSec	SUM (mi)	DIST (nmi)	LongDeg	LongMin	LongSec	SUM (mi)	DIST (nmi)	Ele (m)	Elev (ft)	Length (m)	Length (ft)
A	03°20'19"N	31°46'08"E	207	23	0.36	230.4	200.314	2139	52.9	0.152	-84.95	-73.868	796	2611	1130	3710
A	03°38'57"N	31°45'54"E	207	43.7	1.08	251.8	218.942	2139	51.75	1.026	-85.22	-74.108	940	3100	1300	4260
A	03°02'50"N	30°54'44"E	207	2.3	0.95	210.3	182.826	2070	62.1	0.836	-144.1	-125.273	1204	3951	1700	5600
A	02°48'00"N	32°16'30"E	138	55.2	0	193.2	168.000	2208	18.4	0.57	-50.03	-43.504	1070	3510	3144	10314
A	01°33'57"N	30°13'15"E	69	38	1.08	108	93.942	2070	14.95	0.285	-191.8	-166.752	1233	4045	1850	6070
A	01°43'30"N	33°37'16"E	69	49.5	0.57	119	103.496	2277	42.55	0.304	42.854	37.264	1110	3641	1900	6100
A	01°43'31"N	33°37'17"E	69	49.5	0.59	119	103.512	2277	42.55	0.323	42.873	37.281	1110	3641	770	2525
A	00°02'41"N	32°26'35"E	0	2.3	0.78	3.079	2.677	2208	29.9	0.665	-38.43	-33.422	1153	3782	3658	12000
A	00°02'42"N	32°26'36"E	0	2.3	0.8	3.098	2.694	2208	29.9	0.684	-38.42	-33.405	1153	3782	2408	7900
A	04°12'18"N	34°20'42"E	276	13.8	0.34	290.1	252.297	2346	23	0.798	92.798	80.694	645	2116	1888	6194
A	03°07'20"N	35°36'36"E	207	8.05	0.38	215.4	187.330	2415	41.4	0.684	180.08	156.595	523	1715	1000	3281
A	00°58'30"N	34°57'36"E	0	66.7	0.57	67.27	58.496	2346	65.55	0.684	135.23	117.595	1850	6070	1450	4757
A	00°24'16"N	35°14'20"E	0	27.6	16	43.6	37.913	2415	16.1	0.38	154.48	134.330	2150	7050	3475	11400
A	00°05'10"S	34°43'44"E	0	-5.8	-0.2	-5.94	-5.165	2346	49.45	0.836	119.29	103.727	1157	3796	3300	10826
A	01°24'18"S	35°00'36"E	-69	-28	-0.3	-96.94	-84.297	2415	0	0.684	138.68	120.595	1707	5600	820	2700
A	01°19'12"S	36°48'54"E	-69	-22	-0.2	-91.08	-79.198	2484	55.2	1.026	263.23	228.892	1690	5546	1462	4798
A	01°19'13"S	36°48'55"E	-69	-22	-0.2	-91.1	-79.215	2484	55.2	1.045	263.25	228.909	1690	5546	1540	5052
A	01°19'09"S	36°55'39"E	-69	-22	-0.2	-91.02	-79.149	2484	63.25	0.741	270.99	235.644	1624	5327	4117	13507
A	01°57'59"S	30°07'59"E	-69	-66	-1.1	-135.7	-117.975	2070	8.05	1.121	-197.8	-172.025	1491	4891	3500	11482
A	03°19'26"S	29°19'07"E	-207	-22	-0.5	-229.3	-199.430	2001	21.85	0.133	-254	-220.884	787	2582	3600	11811
A	02°26'40"S	32°55'57"E	-138	-30	-0.8	-168.7	-146.661	2208	63.25	1.083	-4.667	-4.058	1147	3763	3300	10827
A	03°36'34"S	33°30'15"E	-207	-41	-0.6	-249	-216.562	2277	34.5	0.285	34.785	30.248	1158	3800	2000	6562
A	01°30'10"S	33°48'08"E	-69	-35	-0.2	-103.7	-90.165	2277	55.2	0.152	55.352	48.132	1153	3783	1600	5248
A	01°19'56"S	31°49'16"E	-69	-22	-1.1	-91.91	-79.925	2139	56.35	0.304	-81.35	-70.736	1148	3766	1500	4921
A	03°22'33"S	35°49'06"E	-207	-25	-0.6	-232.9	-202.545	2415	56.35	0.114	194.46	169.099	1265	4150	1220	4003
A	03°22'00"S	36°37'19"E	-207	-25	0	-232.3	-202.000	2484	42.55	0.361	249.91	217.314	1387	4550	1620	5315
A	03°25'46"S	37°04'28"E	-207	-29	-0.9	-236.6	-205.760	2553	4.6	0.532	281.13	244.463	894	2932	600	11811
A	02°38'32"S	37°15'00"E	-138	-44	-0.6	-182.3	-158.529	2553	17.25	0	293.25	255.000	1145	3757	1001	3284
A	05°52'32"S	29°15'00"E	-345	-60	-0.6	-405.4	-352.529	2001	17.25	0	-258.8	-225.000	1750	5741	1750	5741
A	06°10'13"S	35°44'58"E	-414	-12	-0.2	-425.7	-370.215	2415	50.6	1.102	189.7	164.958	1109	3637	2042	6699
A	00°34'08"S	37°29'32"E	0	-39	-0.2	-39.25	-34.132	2553	33.35	0.608	309.96	269.529	1265	4150	900	2953
A	00°18'00"S	36°09'36"E	0	-21	0	-20.7	-18.000	2484	10.35	0.684	218.03	189.595	1900	6200	1709	5607
A	00°20'24"S	36°54'36"E	0	-23	-0.5	-23.46	-20.397	2484	62.1	0.684	269.78	234.595	1777	5830	1230	4050
M	03°04'33"S	37°21'09"E	-3	-4	-33	-212.2	-184.545	37	21	9	300.32	261.149	5895	19341		
M	00°09'07"S	37°18'30"E	0	-9	-7	-10.48	-9.116	37	18	30	297.27	258.496	5199	17058		
M	00°23'08"N	29°52'21"E	0	23	8	26.6	23.132	29	52	21	-215.8	-187.653	5109	16763		
M	01°05'59"N	34°34'26"E	1	5	59	75.87	65.975	34	34	26	108.59	94.430	4302	14115		
M	03°14'39"S	36°45'00"E	-3	-14	-39	-223.8	-194.644	36	45	0	258.75	225.000	4566	14981		
M	00°37'37"S	36°42'28"E	0	-37	-37	-43.25	-37.611	36	42	28	255.83	222.463	3902	12802		
M	00°18'40"S	36°36'58"E	0	-18	-40	-21.46	-18.661	36	36	58	249.5	216.958	4001	13127		
M	01°16'10"N	35°25'50"E	1	16	10	87.59	76.165	35	25	50	167.7	145.826	3530	11582		
H	01°15'41"S	36°49'27"E	-1	-15	-41	-87.03	-75.677	36	49	27	263.86	229.446				
H	00°30'43"N	35°16'49"E	0	30	43	35.32	30.710	35	16	49	157.33	136.810				
H	00°36'59"S	30°39'32"E	0	-36	-59	-42.52	-36.975	30	39	32	-161.5	-140.471				
H	00°20'17"N	32°34'34"E	0	19	47	22.74	19.777	32	36	22	-27.18	-23.637				
H	00°19'47"N	32°36'22"E	-2	-31	-14	-173.9	-151.231	32	54	33	-6.273	-5.455				
L	MU	MU	-2	-24	-30	-166.2	-144.496	31	40	30	-91.43	-79.504				
L	MU	MU	MU	MU	MU	MU	-79.925	MU	MU	MU	MU	-65.000				
L	MU	MU	0	-12	-30	-14.37	-12.496	31	58	30	-70.73	-61.504				
L	MU	MU	MU	MU	MU	MU	0.000	MU	MU	MU	MU	-33.422				
L	MU	MU	0	0	45	0.855	0.743	33	58	30	67.27	58.496				
L	MU	MU	0	-21	-5	-24.25	-21.083	34	3	8	72.602	63.132				
L	MU	MU	MU	MU	MU	MU	-5.165	MU	MU	MU	MU	98.000				
L	MU	MU	-1	-6	-40	-76.66	-66.661	33	58	0	66.7	58.000				
L	MU	MU	MU	MU	MU	MU	-90.165	MU	MU	MU	MU	44.000				
L	MU	MU	-2	-4	-45	-143.5	-124.743	32	55	0	-5.75	-5.000				
L	MU	MU	-2	-6	-45	-145.8	-126.743	33	58	30	67.27	58.496				
L	MU	MU	MU	MU	MU	MU	-140.000	MU	MU	MU	MU	-4.058				
L	MU	MU	-2	-24	-30	-166.2	-144.496	31	40	30	-91.43	-79.504				

Appendix A3: Constraints

X Location	Y Location	Elevation	Hospital	Nat'l Feat.	RW Length	Uganda
-73.8678	200.3139	0.42971577	0	0	0.61058809	1
-74.1078	218.9417	0.5101949	0	0	0.70110654	1
-125.273	182.8261	0.65025163	0	0	0.9216424	1
-43.5043	168	0.57767229	0	0	1.69746781	1
-166.7522	93.9417	0.66572206	0	0	0.99899453	0
37.2809	103.5122	0.59923214	0	0	0.41556198	1
-33.4217	2.6774	0.62243778	1	1	1.974948	1
80.6939	252.2974	0.34824916	0	0	1.01940233	0
156.5948	187.3304	0.28225299	0	0	0.5399837	0
117.5948	58.4957	0.99899453	0	0	0.7829023	0
134.3304	37.913	1.16028195	1	0	1.8762006	0
103.727	-5.1652	0.62474188	0	1	1.78173225	0
120.5948	-84.2974	0.9216424	0	0	0.4443633	0
228.9087	-79.2148	0.91275513	1	0	0.83145311	0
235.6443	-79.1487	0.87671233	0	0	2.22296855	0
-172.0252	-117.9748	0.80495589	0	0	1.88969608	0
-220.8843	-199.4296	0.42494298	0	1	1.94384257	0
-4.0583	-146.6609	0.61931078	1	0	1.78189683	0
30.2478	-216.5617	0.6254002	0	1	1.0799674	0
48.1322	-90.1652	0.62260236	0	0	0.86371059	0
-70.7357	-79.9252	0.61980451	1	1	0.80989326	0
169.0991	-202.5452	0.68300285	0	1	0.65880974	0
217.3139	-202	0.74883445	0	0	0.87473739	0
244.4626	-205.76	0.48254563	0	0	1.94384257	0
255	-158.5287	0.6183233	0	0	0.54047744	0
-225	-352.5287	0.94484804	0	0	0.94484804	0
164.9583	-370.2148	0.59857382	0	0	1.10251472	0
269.5287	-34.1322	0.68300285	0	0	0.48600179	0
189.5948	-18	1.0203898	0	1	0.92279445	0
234.5948	-20.3965	0.95949557	0	0	0.66654495	0

Appendix A4: Conclusion Supporting Tables

Random City Information

X	Y	Elevation (nmi)
0.755	0.8745	385.1
0.9927	0.4238	250.2
0.1908	0.0716	527.8
0.72	0.5318	188.9
0.2991	0.6591	620.1
0.265	0.7312	924.7
0.9591	0.3312	67.2
0.4275	0.4538	947
0.5221	0.5568	4.3
0.1406	0.6942	899.2

Simulated Annealing Robust Test (Scenario 1)

RUN	COST	Mean	STDev
1	662.34	813.96	121.86
2	848.78		
3	737.55		
4	915.38		
5	662.34		
6	974.54		
7	915.38		
8	915.38		
9	662.34		
10	845.54		

Simulated Annealing Robust Test (Scenario 5)

RUN	COST	Mean	STDev
1	737.74	715.80	105.61
2	592.21		
3	600.86		
4	857.90		
5	600.86		
6	771.60		
7	600.86		
8	771.60		
9	812.17		
10	812.17		

Appendix B—MATLAB Code

Appendix B1: The Bailey Algorithm

```
function BumpOut(scenario)
%Feed in airport locations to MATLAB. Points are distance in nmi
%from the origin, located at 33 degrees East (longitude, X) and
the equator (latitude, Y).

clc;
clf;

Data      = xlsread('BumpOutData.xls');
AirportDat= [Data(:, :)];
AirportX  = AirportDat(:, 1);         %Airport X coordinate (nmi)
AirportY  = AirportDat(:, 2           %Airport Y coordinate (nmi)


%This variable will be used throughout the code.
num       = length(AirportX);

%Establish an annealing schedule
tfactor   = 0.9;

%-----------------------------------------------------------------
---------
figure(1)
hold on
tri = delaunay(AirportX,AirportY);
triplot(tri,AirportX,AirportY);
axis equal
hold off
title('Triplot')

%-----------------------------------------------------------------
%Combining with Dijkstra

V = [AirportX,AirportY];
I = delaunay(AirportX,AirportY);
J = I(:,[2 3 1]); E = [I(:) J(:)];

E = [E; fliplr(E)];

ibeg = 100;

if (ibeg < 1 || ibeg > length(AirportX))
    fprintf(1, 'Click on Airport to start from\n');
    fprintf(1, 'Click on Airport to end with\n');

    [x, y] = ginput(2);

    xbeg   = x(1);
    ybeg   = y(1);
    xend   = x(2);
```

```
    yend   = y(2);

    ibeg = 1;
    iend = 1;
    sbeg = (AirportX(ibeg)-xbeg)^2 + (AirportY(ibeg)-ybeg)^2;
    send = (AirportX(iend)-xend)^2 + (AirportY(iend)-yend)^2;
    for i = 2 : length(AirportX)
        stest1 = (AirportX(i)-xbeg)^2 + (AirportY(i)-ybeg)^2;
        stest2 = (AirportX(i)-xend)^2 + (AirportY(i)-yend)^2;
        if (stest1 < sbeg)
            sbeg = stest1;
            ibeg = i;
        end % if
        if (stest2 < send)
            send = stest2;
            iend = i;
        end
    end % for i
end % if

[costs,paths] = dijkstra(V,E,ibeg);

PATHS = paths{iend};
tri = delaunay(AirportX, AirportY);

oldprice = 1000;

CDIJ  = 0;                          %Cost of dijkstra
for i = 1 : length(PATHS)-1
    CDIJ = CDIJ + sqrt(((AirportX(PATHS(i)) -
AirportX(PATHS(i+1))) ^ 2) + ...
        (AirportY(PATHS(i)) - AirportY(PATHS(i+1))) ^ 2);
end

temp = -CDIJ / log(0.            %Equation from "metrop" function

for itemp = 1 : 100
    itemp
    nsuccess = 0;
    nfail    = 0;

    figure(2)
    triplot(tri, AirportX, AirportY);
    title(sprintf('Travel from Airport %d to Airport %d', ibeg,
                  iend))
    axis equal

    hold on
    plot(AirportX(ibeg), AirportY(ibeg), 'g*')
    plot(AirportX(iend), AirportY(iend), 'r*')

    for itry = 1 : 1000

        point = rand(1);
        port  = point * num;

        if port >= 0.5
            x1    = AirportX(round(port));
            y1    = AirportY(round(port));
```

```
            else
                x1    = AirportX(1);
                y1    = AirportY(1);
            end

            click = rand(1);
            ibest = 0;
            dbest = Inf;

            for i = 1 : length(PATHS)-1
                dtest = sqrt((x1 - (AirportX(PATHS(i)) +
                              AirportX(PATHS(i+1)))/2)^2 ...
                            +(y1 - (AirportY(PATHS(i)) +
                              AirportY(PATHS(i+1)))/2)^2);
                if (dtest < dbest)
                    ibest = i;
                    dbest = dtest;
                end % if
            end % for i

            if (click >= 0.5)
                Links = [I,      I(:,1),    I(:,2)]; %Bump Right
            else
                Links = [I(:,2), I(:,1), fliplr(I)]; %Bump Left
            end % if

            [NEWPATH, price, okay] = Bump(ibest, Links, PATHS,
                              AirportDat, oldprice, temp, scenario);
            if okay == 1
                PATHS = NEWPATH;
                oldprice = price;
            else nfail = nfail + 1;
            end % if

            if okay == 1
                nsuccess = nsuccess + 1;
            end % if

            if (nsuccess > 100)
                break
            end % if

        end % for itry

        plot(AirportX(PATHS), AirportY(PATHS), 'r-', 'LineWidth', 4)
        title(sprintf('Cost from Airport %d to Airport %d is %f',
                      ibeg, iend, price))
        hold off
        pause

        if (nsuccess == 0)
            break
        end % if
        nsuccess
        nfail
        temp = temp * tfactor

    end % for itemp
```

```
function [NEWPATH, price, okay] = Bump(ibest, Links, PATHS,
AirportDat,...
    oldprice, temp, scenario)

shape = size(Links);

for j = 1:shape(1)
    for jj = 1:3
        jjj = jj + 2;
        if (Links(j, jj) == PATHS(ibest)
            && Links(j,jjj) == PATHS(ibest+1))
            NEWPATH = [PATHS(1:ibest), Links(j,jj+1),
                       PATHS(ibest+1:end)];
            for ii = length(NEWPATH) - 2 : -1 : 1
                if NEWPATH(ii) == NEWPATH(ii+2)
                    if ii+3 <= length(NEWPATH)
                        NEWPATH = [NEWPATH(1:ii),
                                   NEWPATH(ii+3:end)];
                        ii = ii - 1;
                    else
                        NEWPATH = [NEWPATH(1:ii)];
                        break
                    end %if
                end
            end
            price = costfun(NEWPATH, AirportDat, scenario);
            okay  = metrop(price, temp, oldprice);
            return
        end %if
    end %for jj
end %for j

% by default, return input path
NEWPATH = PATHS;

price = costfun(NEWPATH, AirportDat, scenario);
okay  = false;

%-------------------------------------------------------------------
---------
function [cost] = costfun(NEWPATH, AirportDat, scenario)

AirportX = AirportDat(:, 1);
AirportY = AirportDat(:, 2);
AirportE = AirportDat(:, 3);      %Airport Elevation    (nmi)
AirportH = AirportDat(:, 4);      %Hospital Close By (true/false)
AirportN = AirportDat(:, 5);      %Lake/Natural Feature
AirportL = AirportDat(:, 6);      %Length of Runway (nmi)
AirportU = AirportDat(:, 7);      %In Uganda? (true/false)

%Establish an array of coefficients to be used in the "cost"
%function. Order of array is: Right Turn, Left Turn, Elevation,
%Hospital Proximity, Natural Features, Runway Length

cost = 0;

%Establish a baseline value (in this case the average of the
%maximum latitudinal and longitudinal distance). This will be
```

```
%used in the penalty function.

MaxY    = max(AirportY) + (-1*min(AirportY));    %Maximum Y dist
MaxX    = max(AirportX) + (-1*min(AirportX));    %Maximum X dist
AvgD    = (MaxY+MaxX) / 2;                       %Average Dist
Penalty = AvgD * (2/10);                         %Penalty value


for i = 1:length(NEWPATH) - 1
    cost = cost + sqrt(((AirportX(NEWPATH(i)) -
                         AirportX(NEWPATH(i+1))) ^ 2) + ...
                        (AirportY(NEWPATH(i)) -
                         AirportY(NEWPATH(i+1))) ^ 2);
end %for


if     scenario == 1          %Penalty for Elevation Constriction
    for i = 2 : length(NEWPATH) - 1
        if AirportE(NEWPATH(i)) > 0.82
            cost = cost + Penalty;
        end %if Elevation
    end %for i

elseif scenario == 2          %Reward for Hospital
    for i = 2 : length(NEWPATH)
        if AirportH(NEWPATH(i)) == 1 && AirportU(NEWPATH(i)) == 0
            cost = cost - Penalty;
        end %if Hospital, no Uganda
    end %for

elseif scenario == 3           %Nat'l Parks Constriction
    for i = 2 : length(NEWPATH) - 1
        if AirportN(NEWPATH(i)) == 1
            cost = cost + Penalty;
        end %if Natural Features
    end %for

elseif scenario == 4           %Climb Constriction
    for i = 1 : length(NEWPATH) - 1
        if 1/((AirportE(NEWPATH(i))) / (AirportE(NEWPATH(i+1))))
            >= 1.2
            cost = cost + Penalty;
        end %if climb rate
    end %for

elseif scenario == 5            %Penalty for Runway Length
    for i = 2 : length(NEWPATH) - 1
        if (AirportL(NEWPATH(i))) < 0.82
            cost = cost + Penalty;
        end %if runway length
    end %for

elseif scenario == 6              %Penalty for left turn
    for i = 1 : length(NEWPATH) - 1
        if (AirportX(NEWPATH(i))   * AirportY(NEWPATH(i+1))) -...
           (AirportX(NEWPATH(i+1)) * AirportY(NEWPATH(i))) < 0
            cost = cost + Penalty;
        end %if left turn
    end %for
```

```
    cost;
end


%-------------------------------------------------------------------
function [okay] = metrop(price, temp, oldprice)

pd  = price - oldprice;

if pd < 0
    okay = true;
elseif rand(1) < exp(-pd/temp)
    okay = true;
else
    okay = false;
end
```

Appendix B2: Method Test Code

```
%Test to see how different Delaunay and Dijkstra are

%Feed in airport locations to MATLAB. Points are distance in nmi
%from the origin, located at 33 degrees East (longitude, X) and
%the equator (latitude, Y).

clc;
clf;
clear;

%Import the data
DataSet   = xlsread('Data.xls');

%Set up desired airports
AirportX  = DataSet(1:33, 10);
AirportY  = DataSet(1:33, 5);

%Create ending point based on number of airports
num       = length(AirportX);

%Set up Pairs.
Pairs     = [];
a = 1:length(AirportX);
b = 1:length(AirportY);
for i = 1:num
    for j = 1:num
        Pairs = [Pairs; a(i), b(j)];
    end
end

%Create array of airport coordinates
V = [AirportX,AirportY];

%-------------------------------------------------------------
%CALCULATE THE COST OF EUCLIDIAN DISTANCE

%Establish starting and ending cities
EStart = Pairs(:,1);
EEnd   = Pairs(:,2);

%Calculate the distance in between each city
EE = [];
for j = 1:length(Pairs)
    ee = sqrt((AirportX(EStart(j)) - AirportX(EEnd(j)))^2 + ...
        (AirportY(EStart(j)) - AirportY(EEnd(j)))^2);
    EE = [EE, ee];
end

%Sort lengths so the longest link is at the end
EE = EE';
Ee = [Pairs, EE];
E3  = sortrows(Ee, 3);

%Perform calculation, removing the longest link with each
%iteration
numpairs = (length(Pairs) - 1) /2;
```

```
Dst   = [];
LLngh = [];
for i = 1:numpairs
    i
        [costs,paths] = dijkstra(V,E3);
        if (max(isinf(costs)) > 0)
            break
        end % if
        Total = sum(costs)/2;
        total = sum(Total');
        Link  = E3(end, end);
        LLngh = [LLngh, Link];
        Dst = [Dst, total];
        E3 = E3(1:end-2, :);
end
% end

Dst = Dst';

%-----------------------------------------------------------------------
%CALCULATE THE COST OF DELAUNAY
I = delaunay(AirportX, AirportY);
J = I(:,[2 3 1]); E = [I(:) J(:)];

E = [E; fliplr(E)];

DelDst = [];

[costsD,pathsD] = dijkstra(V,E);
for ii = 1:length(costsD)
    for jj = 1:length(costsD)
        if costsD(ii, jj) == Inf;
            costsD(ii, jj) = 0;
        else
            costsD(ii, jj) == costsD(ii, jj);
        end %if
        DelDst = [DelDst; costsD(ii, jj)];
    end
end

DTotal = sum(DelDst)/2;
Dtotal = sum(DTotal')
% DelDst = [DelDst, Dtotal];
% DelTot = sum(DelDst)/2

hold on
plot(LLngh, Dst)
plot(LLngh, Dtotal*ones(size(LLngh)), '-r')
    xlabel('Link Length')
    ylabel('Total Distance Traveled')
    title('Which is better?')
    legend('Euclidian Distance', 'Delaunay')
```