2016

Autonomous Navigation with Obstacle Avoidance for Unmanned Aircraft Systems using MILP

James A. Devens

Follow this and additional works at: https://scholarscompass.vcu.edu/etd

Part of the Other Computer Engineering Commons, and the Other Electrical and Computer Engineering Commons

© The Author

Downloaded from
https://scholarscompass.vcu.edu/etd/4461

This Thesis is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.
AUTONOMOUS NAVIGATION WITH OBSTACLE AVOIDANCE FOR UNMANNED AIRCRAFT SYSTEMS USING MILP

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University

By

James A. Devens

Major Director: Dr. Robert H. Klenke,
Professor of Electrical and Computer Engineering, VIP Director

Virginia Commonwealth University
Richmond, Virginia
August 2016
Acknowledgements

First and foremost, I would like to thank my family for supporting me through my academic career and in life. Words cannot describe how appreciative I am to have the parents and family support structure I have witnessed throughout my life. I would next like to thank my advisor, Dr. Robert H. Klenke, for all his support, guidance, and patience with me. Furthermore, I would also like to thank Dr. Tim Bakker for his immense support and guidance throughout my research and in learning about the UAV laboratory. I have learned a tremendous amount from him and I appreciate his patience in answering the thousands of questions I had throughout my graduate school career. Also, I hope he enjoys the wooden flip-flops I made for him so he can take a break from wearing his wooden clogs.

I would also like to thank Matt Leccadito for convincing me to come to VCU and introducing me to the UAV laboratory. I will definitely miss the times we shared debugging programs, going to the gym, and talking about the good old VMI days. Lastly, I would like to thank the numerous graduate students and professors that have helped me throughout my graduate school career. I am greatly appreciative of the opportunity to have been a graduate student at VCU and work in the VCU UAV laboratory.
# Table of Contents

Acknowledgements ........................................................................................................ ii

List of Figures ...................................................................................................................... v

List of Tables ...................................................................................................................... viii

Acronyms .......................................................................................................................... ix

Table of Symbols ................................................................................................................ x

Abstract ............................................................................................................................ xi

Chapter 1 – Introduction ..................................................................................................... 1

1.1 UAV Overview .............................................................................................................. 1

1.2 Problem Statement ...................................................................................................... 3

1.3 Thesis Overview ......................................................................................................... 5

Chapter 2 – Background ..................................................................................................... 7

2.1 Chapter Overview ......................................................................................................... 7

2.2 Path-planning Algorithms ......................................................................................... 7

2.2.1 A-Star Algorithm ................................................................................................ 8

2.2.2 Exhaustive Search Algorithm ............................................................................. 11

2.3 Related Work ............................................................................................................. 14

2.3.1 Visibility Graphs .................................................................................................. 14

2.3.2 Mixed Integer Linear Programming .................................................................. 17

2.3.4 GNU Linear Programming Kit ........................................................................... 21

Chapter 3 – Implementation and Problem Formulation ....................................................... 24

3.1 Chapter Overview ....................................................................................................... 24

3.2 Visibility Graph Implementation .............................................................................. 24

3.3 Mixed Integer Linear Programming Problem Formulation ...................................... 29
Chapter 4 – Hardware and Software Platforms ................................................. 32

4.1 Chapter Overview ......................................................................................... 32
4.2 Unmanned Aerial Vehicle, Skyhunter EP ......................................................... 32
4.3 Ground Control Station .................................................................................. 34
4.4 RAMS Simulator .......................................................................................... 35
4.5 Flight Control System .................................................................................... 37
4.6 Mission Control System ................................................................................ 38

4.6.1 GLPK Software Implementation ................................................................. 45
4.6.2 VACS Communication Protocol ................................................................. 47
4.6.3 External MCS Microcontroller, Raspberry Pi 2 Model B ............................... 48

Chapter 5 – Testing and Results ......................................................................... 51

5.1 Chapter Overview .......................................................................................... 51
5.2 Simulation and Testing Process ...................................................................... 51
5.3 Simulation Results ......................................................................................... 54

5.3.1 Visibility Graph Correctness ...................................................................... 54
5.3.2 Path-planning Algorithm Performance Independent of MCS ......................... 61
5.3.3 Internal MCS Simulations ........................................................................... 66
5.3.4 External MCS Simulations .......................................................................... 71

5.4 Real-world Results ......................................................................................... 74

Chapter 6 – Conclusions and Future Work ......................................................... 80

6.1 Conclusions .................................................................................................. 80
6.2 Future Work .................................................................................................. 81

Bibliography ........................................................................................................ 83
List of Figures

Figure 1. Example of obstacle buffer............................................................ 4
Figure 2. Grid based versus straight-line distance A-Star graphs [10].................. 9
Figure 3. Exhaustive Search multi-branch tree structure.................................. 12
Figure 4. Visibility of an obstacle field from vertex 4........................................ 16
Figure 5. Grid based obstacle field .................................................................. 19
Figure 6. Obstacle field considering obstacle vertices........................................... 20
Figure 7. Demonstrates an ear of a concave polygon......................................... 26
Figure 8. Flow constraints representation.......................................................... 31
Figure 9. Skyhunter EP used for real-world flight-testing...................................... 33
Figure 10. High-level system architecture of the UAS used in flight-testing......... 34
Figure 11. GCS obstacle avoidance being executed............................................. 35
Figure 12. RAMS simulator featuring internal MCS.......................................... 36
Figure 13. RAMS simulator featuring external MCS.......................................... 36
Figure 14. Aries v1.1 (top side on left and bottom side on right)......................... 38
Figure 15. MCS software architecture modified for path-planning algorithms...... 40
Figure 16. MILP path-planning algorithm software architecture......................... 42
Figure 17. Visibility Graph algorithms software architecture............................... 43
Figure 19. XML code snippet of obstacle field................................................... 48
Figure 20. Methods for applying obstacles to the GCS map............................... 52
Figure 21. Right-click display for applying obstacle............................................ 52
Figure 22. Example of solved flight-path ........................................................... 53
Figure 23. Obstacle avoidance for a triangle ...................................................... 55
Figure 24. Obstacle avoidance for a simple convex polygon ................................................. 56
Figure 25. Obstacle avoidance for a simple concave polygon (example 1) .................... 57
Figure 26. Obstacle avoidance for a simple concave polygon (example 2) .................... 57
Figure 27. Obstacle avoidance for tunnel (example 1) .................................................. 58
Figure 28. Obstacle avoidance for tunnel (example 2) .................................................. 59
Figure 29. Obstacle avoidance for random obstacle field (example 1) ......................... 60
Figure 30. Obstacle avoidance for random obstacle field (example 2) ......................... 60
Figure 31. Test case 1, 4 obstacles, total distance = 386.35 meters ......................... 61
Figure 32. Test case 2, 9 obstacles, total distance = 485.28 meters ......................... 62
Figure 33. Test case 3, 16 obstacles, total distance = 1087.88 meters .................... 62
Figure 34. Test case 4, 25 obstacles, total distance = 1353.50 meters .................... 63
Figure 35. Test case 5, 64 obstacles, total distance = 2173.14 meters .................... 63
Figure 36. Test case 6, 100 obstacles, total distance = 2506.18 meters ................... 64
Figure 37. Internal MCS test case 1, 4 obstacles, total cost = 11839.48 meters .......... 68
Figure 38. Internal MCS test case 2, 9 obstacles, total cost = 7059.86 meters .......... 68
Figure 39. Internal MCS test case 3, 16 obstacles, total cost = 16180.42 meters ...... 70
Figure 40. Internal MCS test case 4, 25 obstacles, total cost = 2274.12 meters ...... 70
Figure 41. External MCS test case 1, 4 obstacles, total cost = 12113.95 meters ....... 72
Figure 42. External MCS test case 2, 9 obstacles, total cost = 7392.66 meters ....... 72
Figure 43. External MCS test case 3, 16 obstacles, total cost = 15969.34 meters ...... 74
Figure 44. Real-world flight testing, test case 1 (left), test case 2 (right)............... 75
Figure 45. Real-world flight testing, test case 3 (left), test case 4 (right)............... 76
Figure 46. Real-world flight testing, test case 5 (left), test case 6 (right)............... 77
Figure 47. Real-world flight testing, test case 7 ................................................................. 78
List of Tables

Table 1. Adjacency matrix for obstacle field including source and target .................. 16
Table 2. Popular and well-known open source MIP/LP solvers................................. 22
Table 3. Popular and well-known commercial MIP/LP solvers .................................. 22
Table 4. Sorted adjacency list without cost (left), with cost (right).............................. 44
Table 5. VACS packet format ................................................................................. 47
Table 6. Raspberry pi 2 Model B specifications ...................................................... 49
Table 7. Raspberry Pi 2 Model B Connectors ......................................................... 49
Table 8. Execution time in seconds relative to the number of obstacles for each algorithm .................................................................................................................. 65
Table 9. Total cost in meters relative to the number of obstacles for each algorithm ..... 66
Table 10. Execution time relative to the visibility graph algorithm ......................... 66
Table 11. Internal MCS Simulation performance data ............................................. 71
Table 12. External MCS simulation performance data .......................................... 74
Table 13. Real-world testing performance data ...................................................... 79
Acronyms

i. UAV – Unmanned Aerial Vehicle

ii. UAS – Unmanned Aerial Systems

iii. MILP – Mixed Integer Linear Program

iv. GCS – Ground Control Station

v. FCS – Flight Control System

vi. MCS – Mission Control System

vii. GUI – Graphical User Interface

viii. UPS – United Parcel Service

ix. GLPK – GNU Linear Programming Kit

x. OOP – Object Oriented Programming

xi. LOS – Line Of Sight

xii. ILP – Integer Linear Program

xiii. UDP – User Datagram Protocol

xiv. XML – Extensible Markup Language

xv. API – Application Program Interface
### Table of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_T$</td>
<td>Triangular obstacles</td>
</tr>
<tr>
<td>$O_{cx}$</td>
<td>Convex simple polygon obstacles</td>
</tr>
<tr>
<td>$O_{cv}$</td>
<td>Concave simple polygon obstacles</td>
</tr>
<tr>
<td>$O$</td>
<td>Set of all obstacles</td>
</tr>
<tr>
<td>$V$</td>
<td>Set of obstacle vertices</td>
</tr>
<tr>
<td>$E$</td>
<td>Set of obstacle edges</td>
</tr>
<tr>
<td>$T_\Delta$</td>
<td>Set of triangles</td>
</tr>
<tr>
<td>$G$</td>
<td>Undirected graph or mesh</td>
</tr>
<tr>
<td>$A$</td>
<td>Set of arcs</td>
</tr>
<tr>
<td>$S$</td>
<td>Source node</td>
</tr>
<tr>
<td>$T$</td>
<td>Target node</td>
</tr>
<tr>
<td>$C_{ES}$</td>
<td>Set of solutions for Exhaustive Search algorithm</td>
</tr>
</tbody>
</table>
Abstract

Autonomous coordination among multiple aerial vehicles to ensure a collision free airspace is a critical aspect of today’s airspace. With the rise of Unmanned Aerial Vehicles (UAVs) in the military and commercial sectors, obstacle avoidance in a densely populated airspace is necessary. This thesis investigates finding optimal or near-optimal trajectories in real-time for aircraft in complex airspaces containing a large number of obstacles. The solution for the trajectories is described as a linear program subject to mixed integer constraints, known as a Mixed Integer Linear Program (MILP). The resulting MILP problem is solved in real time using a well-known, public domain MILP solver. In addition, an Exhaustive, Breadth-First Search algorithm was implemented and is used for comparison in terms of execution time and flight path optimality. The Exhaustive Search algorithm is comprised of a multi-branch tree structure that iterates through all possible flight paths from source to target. The MILP solution was implemented in both PC based and embedded system environments. The embedded system environment was implemented on an onboard processor to develop trajectories for each individual aircraft in real time.
Chapter 1 – Introduction

1.1 UAV Overview

This Thesis presents the research done in the field of autonomous Unmanned Aircraft Systems (UAS) and, in particular, path-planning algorithms for maneuvering around obstacles and their implementation for an Unmanned Aerial Vehicle (UAV). The scale and scope of the current UAV revolution, especially within the civilian sector, has caught many by surprise in recent years. In fact, the Federal Aviation Administration (FAA) forecasts projections for growth in small, unmanned aircraft purchases, from 1.9 million in 2016 to as many as 4.3 million by 2020 [1]. Although these forecast predictions are for small scale UAV use within commercial sector, larger scale UAV research and development for military or government use is expected to grow for the numerous advantages they provide. The primary one being the elimination of the need for a pilot(s) or crew in conventional manned aircraft. UAVs drastically reduce maintenance and flight costs associated with much larger manned aircraft. These advantages of reducing cost and increasing efficiency have attracted numerous commercial aircraft companies, along with the United States government, to investigate the possibility of fully autonomous flight. In fact, the United States government and its military have increased their budgeting for UAV research and development programs, which are ultimately designed to decrease the number of manned aircraft and costs for pilot training. The United States Air Force (USAF) is so convinced UAVs are the way of the future that they make up the fastest growing aerial segment within the USAF.
Similar to the USAF vision for UAVs, commercial sectors including real-estate, construction and engineering, utilities and energy, technology, media and entertainment, agriculture, delivery, and internet are at the forefront of exploring the potential of UAV applications. For instance, within the retail delivery sector, companies such as Amazon and UPS are researching same-day UAV package delivery. Furthermore, UAVs are currently being used for aerial photography, land surveying, fire detection and early containment, law enforcement, search and rescue operations, and air traffic control support to name a few. However, with the rise in commercial and defense use of UAVs, research and development of flight control algorithms are needed to maintain the safety of the airspace and on the ground. Thus, fully and partially autonomous path-planning or navigation algorithms to avoid obstacles are an essential component for the future of UAVs.

Currently, all UAVs are controlled via a ground control station either through semi-autonomous, guided behavior or complete manual control. The ground control station for a US military Predator drone consists of two trained operators in which one person is responsible for flight and the second operator is responsible for sensor data and monitoring [2]. These individual responsibilities are often very taxing, requiring multiple teams of pilots working shifts during a particular mission [2]. Although the current system is quite functional, the system can be greatly improved through the development of advanced autonomous flight algorithms. Reducing the stress and responsibilities of the operators allows for increased operator situational awareness and the ability for efficient handling of the UAV(s), especially in a combat environment. Similarly, reducing the stresses on commercial operators will increase the effectiveness of the UAV
control, but also increase the safety of the civilian sector airspace. With the dramatic increase in the small commercial drone population as previously referenced, obstacle avoidance is a necessity. Static obstacles such as buildings and geographic barriers, along with dynamic obstacles such as manned aircraft or other UAVs, are all prime examples of obstacles that could pose a threat.

1.2 Problem Statement

The objective of this thesis is to investigate an optimal guidance and obstacle avoidance algorithm using Mixed Integer Linear Programming (MILP). Essentially, the UAV will be modeled as a moving point mass within a 2-dimensional geographic grid. Although this is considered a simplified model of aircraft dynamics, it is proven to be an accurate approximation of the aircraft dynamics in a three dimensional airspace given a fixed altitude [3][4]. The algorithm focuses on obstacle avoidance pertaining to static obstacles, which could possibly include buildings, restricted airspace, or geographic barriers (mountains, trees, etc.). Static obstacles are represented as simple concave and/or convex polygons, containing possible waypoints at each corner of the polygon. Having described the general problem, it is important to state key assumptions defining the problem statement.

First, it is assumed all obstacles contain a perimeter buffer between the physical obstacle and the characteristic obstacle entered into the GCS. An example of this obstacle buffer is shown in Figure 1.
It is also assumed the Ground Control Station (GCS) operators have prior knowledge of the static obstacle geographic locations, represented in latitude and longitude. Furthermore, planar motion is only considered for UAV navigation from waypoint to waypoint; meaning the aircraft is only represented in a two dimensional airspace where the UAV can only fly around obstacles and cannot fly over or under them.

The optimal trajectory is taken with respect to the distance from source to destination. In addition, the optimization of trajectory is accomplished with the assumption that there is an initial position, final position, and possible flyable waypoints in between. It is assumed the GCS operator has this information prior to conducting the flight mission. The UAV is required to navigate from the initial state to the final state using the known flyable waypoints while achieving obstacle avoidance. The cost function is also implemented over a limited range due to the limitations of the guidance algorithm in that, a fixed horizon is considered. This allows for a simplified extension of
a planar geographic grid as opposed to a receding horizon approach where the curvature of the earth is taken into account.

In current path planning and autonomous flight algorithms, there are significant advantages in creating an embedded system contained within the aircraft. One major advantage is a compact, hardwired system where hardware and software is centrally located within the plane. Furthermore, it creates a simpler and user-friendlier environment at the GCS level. However, there are major disadvantages and complexities including processing power, memory capacity, and hardware size and weight constraints, which all need to be considered. For this reason, most MILP implementations resort to an implementation at the GCS level primarily due to memory and processing power. In this thesis a compact, low cost microcontroller is used to satisfy size and weight limitations of the aircraft along with the necessary memory and processing power needed to fulfill MILP optimization in real-time on an embedded system.

1.3 Thesis Overview

This thesis provides a practical solution for finding near-optimal flight paths from source to target while maintaining obstacle avoidance. In the following chapters the design, development, and testing of a MILP guidance algorithm with obstacle avoidance are presented. Chapter 2 provides background information in regards to fundamental path-planning algorithms and related work. The related work section consists of background information as it pertains to the MILP based path-planning solution.
Chapter 3 introduces the problem formulation and implementation. This chapter investigates the visibility graph formulations used to model the connectivity of the obstacle field along with the mathematics of MILP problem formulation. Chapter 4 describes the hardware and software platforms used for the simulations and testing done throughout the progression of this thesis. Furthermore, the connectivity of the different hardware and software systems is explained, along with the characteristics and specifications of the embedded microcontroller. The next two chapters focus on the results and conclusions of this thesis.

Chapter 5 describes the different simulations conducted as it pertains to algorithm execution time and the optimality of the flight-path trajectory. Additionally, an implementation of an Exhaustive, Breadth-First Search algorithm is used for comparison in simulation. The simulations consist of multiple scenarios and configuration environments. Environments include stand-alone algorithm and internal Mission Control System (MCS) simulations, and external MCS testing using a Raspberry Pi 2 Model B microcontroller. These simulation environments ultimately lead to real-world flight-testing, in which the results and data from flight-testing are included in Chapter 5.

The final chapter, Chapter 6, concludes the work achieved in the design, development, and testing of an autonomous MILP based path-planning solution with obstacle avoidance. Future work to incorporate dynamic obstacles within a collaborative UAV network is included. Furthermore, incorporation of the aircraft flight dynamics in the MILP problem is suggested.
Chapter 2 – Background

2.1 Chapter Overview

This chapter provides background information leading to the design and development of a UAV guidance and obstacle avoidance algorithm using Mixed Integer Linear Programming (MILP). Section 2.2 introduces fundamental path-planning algorithms including the A-Star algorithm along with Dijkstra’s algorithm for finding the shortest path. In addition, an Exhaustive Search algorithm, also considered as a Breadth-first Search, is discussed. It is important to note the Exhaustive Search algorithm was implemented as a test bench for the MILP based path-planning algorithm, which will later be discussed. Section 2.3 introduces related work as it pertains to the MILP path-planning algorithm created in this thesis. Visibility graphs, MILP, and the MILP solver, GNU Linear Programming Kit (GLPK) are all discussed in this section.

2.2 Path-planning Algorithms

Path planning with obstacle avoidance is a fundamental aspect of autonomous vehicle navigation. Regardless of whether the autonomous vehicle is a land, air, or water based, the trajectory methods are very similar. Nonetheless, there are key differences in path planning for UAVs versus land or water based robots. Guidance for UAVs typically involve non-trivial dynamics, three-dimensional environments, disturbed operating conditions, and high levels of uncertainty [5]. A survey of motion planning algorithms conducted in [5] lists several algorithms suited for these more complex environments. The algorithms listed include: roadmap methods, probabilistic approaches, state-space
sampling methods, mathematical programming, and potential field methods to name a few. For the work described in this thesis, the complexities in terms of aircraft dynamics are considered negligible. Furthermore, a two-dimensional geospatial airspace is considered without change relative to the third dimension altitude. The preliminary algorithm research conducted in this section is based on these assumptions. In addition, the path-planning algorithms described are well suited to the use of a visibility graph to model the obstacle field connectivity. Section 2.3.1 describes visibility graphs in depth.

2.2.1 A-Star Algorithm

One of the most common path-planning algorithms used in autonomous vehicle navigation, game theory, or general navigation from source to destination is known as the A-Star or the A* algorithm discovered in 1968 [6]. The A* algorithm is a modification to Dijkstra’s algorithm [7], in that a heuristic function $h(n)$ is used to estimate the lowest cost of the path from source $S$ to target $T$ through node $n$. For each node in the current search, the total search cost $f(n) = g(n) + h(n)$ where $g(n)$ is the cost from the source to node $n$ and $h(n)$ is the estimated (heuristic) cost from node $n$ to the target $T$. The node in the search with the lowest cost, $f(n)$ is selected as the next node to explore in the search for the shortest path. If the heuristic function is considered to be zero then the A* algorithm actually implements Dijkstra’s algorithm. A restriction on the A* heuristic $h$ function must not overestimate the cost to reach the target; that is, it must be what is termed admissible [8][9]. The $h$ function must always equal the exact distance from source to target from all vertices $v_i \in V$. The heuristic function plays a very important role in terms of time complexity, completeness, and optimality of the algorithm [9]. For
instance, if the heuristic cost function is solved at every time step as the vehicle moves versus pre-solving the heuristic at the start of the algorithm, it may be detrimental to performance. Typically, there are two different heuristic function approaches to the shortest path problem.

The first one consists of a straight-line distance heuristic from all vertices to the target. That is, the heuristic cost $h(n)$ for each node $n$ is the simple straight-line distance from node $n$ to the target $T$. The second approach consists of a grid-based system of nodes or vertices, in that the cumulative line-nodes heuristic sums up the cost values of grid nodes that are adjacent to every grid node and the target node. This method requires an iterative process that looks at adjacent nodes, until the target is determined an adjacent node [9]. Both methods of modeling the shortest path problem are pictorially shown in Figure 2.

![Figure 2. Grid based versus straight-line distance A-Star graphs [10]](image)

For the purposes of this thesis, the first method described would be applicable due the visibility graph formulation, which will be discussed in Section 2.3.1. Although, there are numerous implementations of the A-Star algorithm, an example is described In
Algorithm 1, which is known as the Minimum Weight Node First Principle A-Star algorithm [11].

Algorithm 1 Minimum weight first principle A-Star algorithm
given an open list of nodes, initially containing the source node
given a closed list of nodes, initially empty
do consider the node with the lowest f value
if considered node is target then
    target found, exit algorithm
else
    move current node to closed list and consider neighbor nodes
    for each currently explored neighbor node do
        if this neighbor is in closed list and current g value is lower then
            update the neighbor with the new, lower g value
            change the neighbor’s parent to our current node
        else if neighbor is in open list and current g value is lower then
            update the neighbor with the new, lower g value
            change the neighbor’s parent to our current node
        else
            this neighbor is not in either open or closed list
            add the neighbor to the open list and set its g value
        end if
    end for
end if
while target isn’t reached

The computational complexity of the A-Star algorithm, along with Dijkstra’s algorithm, is considered a quadratic-time algorithm $O(n^2)$ [9][12]. Specifically, the complexity function $T(n) = 2(n - 1)^2 \in O(n^2)$, where $n$ is the number of vertices of the graph data structure. As previously mentioned, the only difference between the two algorithms is the addition of a heuristic function $h(n)$ which greatly reduces the search from source to target for most applications compared to Dijkstra’s algorithm.
2.2.2 Exhaustive Search Algorithm

An alternate algorithm used for solving the path-planning problem is an implementation of an exhaustive search otherwise known as a \textit{brute force} approach. Essentially, this approach consists of iterating through all possible routes from source $S$ to target $T$ until the optimal path is found. Although this algorithm is more exhaustive and will determine the exact solution to the shortest path problem in all cases, compared to the MILP solution, it is significantly slower in execution time. In addition, the algorithm is heavily limited by the number of obstacles and vertices before the computation of the exact solution becomes intractable. For reasonable run-times, the obstacle fields must be significantly limited relative to that of a MILP based implementation with the same run-time constraint. Similarly to the A-Star approach, the exhaustive search requires the same visibility graph calculations before running its algorithm. An adjacency matrix or list can be used to store the connectivity of the visibility graph, which will be discussed in Section 2.3.1. Once all visibility calculations have been completed the exhaustive search algorithm may begin.

The algorithm consists of a multi-branch tree structure where the source node $S$ is the head or \textit{parent} of the tree. Figure 3 depicts a general flow diagram of the multi-branch tree structure discussed in this section for better understanding.
Figure 3. Exhaustive Search multi-branch tree structure

The first step in the algorithm begins at Level 0, in which branches or children are added/connected to the parent until all connections from source $S$ to visible obstacle vertices are confirmed within the adjacency matrix. Recursion can easily be used to continue the process until all children are added to their respective parents. Notice, the source $S$ will always be a parent, the target $T$ will always be a child, and the obstacle vertices will either be a child or parent as the Levels progress. This rule is very similar to the flow constraints (2) listed in the MILP problem formulation, which will be further discussed in Section 2.3.3. If the target $T$ is added as a child to a specific branch, the total path cost is stored in the solution set $C_{ES}$ as a reference to future path costs. Furthermore, if a current branch’s cost is greater than a cost stored within the solution set $C_{ES}$, the branch is terminated since a more optimal path exists. Thus, one can describe
the algorithm as Breadth-first due to the elimination of negligible branches. However, for the purposes of this thesis, the algorithm is described as an Exhaustive Search. The algorithm is finished once there are no more children to be added to the tree structure. The general algorithm describing the Exhaustive Search is shown below in Algorithm 2.

**Algorithm 2** Multi-branch tree structure for finding optimal path given an adjacency matrix $A_m$ populated via the visibility graph algorithms

do
  function addChildren
    if at lowest layer then
      for all children of current parent do
        if child is visible then
          check if child is taken
          if target reached then
            check if path is optimal
          else if child is not taken then
            if current path is less than current optimal path then
              create new child node
          end if
        end if
      end loop
    else
      for all children of current parent do
        recursive call on addChildren, update new parent as child
      end loop
    end if
  end if
end function
while there are children left to be added

The Breadth-first search algorithm described in Algorithm 2 is implemented and used as a test benchmark for performance when examining the MILP implementation.
2.3 Related Work

As previously mentioned, there are numerous path-planning or navigation algorithms for unmanned vehicles in regards to land, air, and sea operations. Regardless of the environment or the algorithm itself, there are three general requirements that need to be met. These three requirements include: 1) the resulting path should have the lowest possible cost to prevent any unnecessary indirection, 2) the algorithm should be fast, robust, and correct, eliminating the possibility of collision, and 3) the algorithm should be adaptable with respect to different maps or environments [13]. Although, there are several solutions to the Euclidean shortest path problem, this thesis provides a solution using a MILP algorithm that minimizes the distance from source to target in complex obstacle fields. MILP has been chosen due to its robustness and ability to solve complex linear problems in real time. For instance, traditional algorithms such as A-Star or multi-branch search trees can take up to several minutes if not hours when solving obstacle fields consisting of thousands of obstacles. On the other hand, an MILP based solution can solve the same complex fields in seconds or minutes, making it useful for real-time applications. Before examining an MILP based approach, it is important to understand the visibility or connectivity of the obstacle field. Thus, the next section will explain how the connectivity is modeled and how this information is useful for the problem formulation.

2.3.1 Visibility Graphs

Visibility graphs, otherwise characterized as road maps, are graphs containing both visible and non-visible paths, typically for a set of nodes and obstacles in the
Euclidean plane. Visible paths are deemed flyable whereas non-visible paths are deemed not flyable. For the purposes of this section, the visibility graph is denoted as $G_{\text{vis}}(V)$, where $V$ describes the set of vertices contained in the Euclidean plane. There is an arc between vertices $v_i$ and $v_j$ if they see each other, that is; if the segment $v_i v_j$ does not intersect the interior of any obstacle in the Euclidean plane [14]. Note if a given path is deemed visible from $v_i \rightarrow v_j$ then the path from $v_j \rightarrow v_i$ is also mutually visible. Once the visibility graph is formulated the entire connectivity of the system between obstacle vertices is known.

The connectivity of the system is represented as an undirected graph or network, where all visible paths are bidirectional. An adjacency matrix or list is used to store this information where the rows and columns represent every possible connection from vertex to vertex. The information stored in the adjacency matrix $A_m$ consists of the distance from vertex to vertex if visible and a distance value greater than the fixed horizon if the path is not visible. The fixed horizon is defined as the outer most edge of the two-dimensional geospatial map. Thus, if the path from vertices $v_i$ and $v_j$ is not visible, the distance stored in $A_m$ is equal to the maximum limit or horizon of the map. An example of an obstacle field along with its adjacency matrix is shown in Figure 4 and Table 1 respectively. Figure 4 demonstrates the visibility from vertex four to all other vertices within the obstacle vertex set, where the light green lines represent the obstacles that cannot be penetrated. The red dotted lines from vertex four represent non-flyable paths and the solid blue lines represent a visible or flyable path.
Figure 4. Visibility of an obstacle field from vertex 4

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>T</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>INF</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>INF</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>INF</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>INF</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>INF</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>INF</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>INF</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>INF</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>INF</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Adjacency matrix for obstacle field including source and target
Note the adjacency matrix in Table 1 can be stored as a square matrix due to the property of mutual visibility previously discussed for undirected graphs. As a square matrix, the matrix can be represented as a lower or upper triangle, which significantly reduces the memory required to store the connectivity of the Euclidian plane. However, if the problem contains a directed graph, the matrix can no longer be considered as a square matrix.

2.3.2 Mixed Integer Linear Programming

Mixed Integer Linear Programming (MILP) is defined as a linear optimization problem in which some, but not all, variables are restricted to be integers [15]. In more formal terms, a MILP problem consists of the following: 1) a linear objective function \( f^T x \), where \( f \) is a column vector of constants and \( x \) is the column vector of unknowns, 2) bounds and linear constraints, and 3) restrictions on some components of \( x \) to have integer values. In mathematical terms, given vectors \( f \), \( l_b \), and \( u_b \), matrices \( A \) and \( A_{eq} \), and corresponding vectors \( b \) and \( b_{eq} \), the MILP problem is represented as the following:

\[
\min_x f^T x \text{ subject to } \begin{cases} 
    A \cdot x \leq b \\
    A_{eq} \cdot x = b_{eq} \\
    l_b \leq x \leq u_b 
\end{cases}
\]

where vector \( x \) is the solution set to the minimization problem and the lower and upper bounds are represented as \( l_b \) and \( u_b \) respectively [16]. Additionally, the matrices \( A \) and \( A_{eq} \) and corresponding vectors \( b \) and \( b_{eq} \) restrict the solution \( x \), and are defined as constraints given linear inequalities and linear equalities. Although the example provided
is a minimization problem, the objective function could consist of a maximization problem as well. Furthermore, if the given problem defines vector \( x \) to contain all integers, the problem is called a pure integer program, otherwise known as an Integer Linear Program (ILP) [15]. Both ILP and MILP problem formulations allow for fast, realistic solutions to linear optimization problems, including the path-planning problem presented in this thesis.

MILP based path-planning algorithms for UAVs typically take into account the aircraft flight dynamics and linear constraints associated with a given problem. Additionally, waypoints and obstacles can be represented as constraints within the problem as described in Refs. [3][17][18]. Although most research has been done in regards to modeling the aircraft flight dynamics relative to simple obstacle fields, this thesis focuses on solving the linear flight paths from waypoint to waypoint for complex obstacle fields. Possible waypoints are defined at the corners of any simple, convex or concave polygon-shaped obstacle within the two-dimensional field. This is a key difference between the approach presented in this thesis and the algorithms presented in Refs. [3], [17], and [18]; in that, they limited obstacles to convex rectangles.

Another key difference between this approach and common MILP solutions to the path-planning problem is the representation of obstacles as constraints. Stationary obstacles, which are considered as pre-determined no-fly areas at the GCS level, are defined by the lower left and upper right corner points: \((x_{\text{min}}, y_{\text{min}})\) and \((x_{\text{max}}, y_{\text{max}})\) [3]. At every time step \( i \) the position \((\text{lat}_i, \text{lon}_i)\) of the UAV must be positioned in the area outside of the obstacle [3]. As previously mentioned, this can be modeled as constraints within the MILP problem. However, for this thesis this approach is not used.
Instead of considering a digitized two-dimensional plane, only the vertices of the obstacles and the current position of the aircraft are considered, allowing for a significantly reduced MILP problem space. Figure 5 and Figure 6 demonstrate an example where the problem space is drastically reduced from a 40x40 grid of nodes to an obstacle field only containing 15 nodes, respectively.

Figure 5. Grid based obstacle field
A visibility graph is used to model the visibility or possible flyable waypoints from all other visible waypoints within the obstacle field. The significant reduction in problem space witnessed in Figure 6 also aids with reducing the complexities associated with the visibility graph algorithms and adjacency matrix storage, which will be further discussed in Section 3.2. It is important to note that the solution presented herein is developed in only two-dimensions. Although aircraft can obviously operate in three-dimensions, avoiding obstacles by increasing or decreasing altitude, i.e., utilizing the third-dimension, can result in increased energy consumption and decreased speed on the part of the vehicle as well as adding additional uncertainty in the overall conflict-free solution. In addition, many smaller-scale vehicles do not have the sensor capability to determine the overall height of an obstacle in addition to its two-dimensional position and
extent. This two-dimensional MILP based path-planning problem is solved using the well-known, open source solver GLPK. The next section will explain the advantages and disadvantages of GLPK versus other commercial and open source solvers.

2.3.4 GNU Linear Programming Kit

The GNU Linear Programming kit otherwise known as GLPK, is an open source package intended for solving large-scale linear programming (LP), mixed integer programming (MIP), and other related problems [19]. The GLPK package is written in ANSI C and organized in the form of a callable library [19]. Although there are numerous commercial and free, open source solvers for linear optimization problems, GLPK was chosen due to its compatibility with ARM processors and its open source convenience. Furthermore, GLPK was chosen for its superior performance in comparison to other open source solvers; which will be discussed later. Other popular open source and commercial solvers are shown in Table 2 and Table 3 respectively. Analysis of commercial and open source for linear optimization problems can be found in [20].
| **LP_SOLVE** | [21] Open source solver that can be used to solve linear, mixed integer programs and is written in ANSI C. |
| **CLP** | [22] Created within the Coin-OR project and is written in C++ to handle linear optimization problems. The Coin-Or project intentions are to create open software for the operations research community/ |
| **SCIP** | [23] Contains a framework for solving integer and constraint programs. It is available as an ANSI C callable library or a standalone solver with LP solver support. |
| **SoPlex** | [24] A linear programming solver that is based on the revised simplex algorithm. It is implemented in C++ and can be used as a standalone solver or can be embedded into other programs using a C++ class library. |

Table 2. Popular and well-known open source MIP/LP solvers

| **Cplex** | [25] Otherwise known as the IBM ILOG CPLEX Optimization Studio. It is designed to handle large scale, mixed integer linear problems. It also features several interfaces and is able to connect the solver to different programming languages or modeling systems. |
| **Xpress** | [26] Commercial, proprietary software that is designed to solve mixed integer linear problems. It is available on most common computer platforms and provides several interfaces including callable library APIs for several programming languages. |
| **Gurobi** | [27] This software is a modern solver for mixed integer linear and well as non-linear mathematical optimization problems. It is written in C and is available in all computing platforms and accessible from several programming languages. |

Table 3. Popular and well-known commercial MIP/LP solvers
Table 2 and Table 3 provide minimal descriptions of the features and specifications these optimization suites provide. For more information regarding the specifics of these aforementioned solvers, the corresponding solver pages provide a great deal of information. Although there are key differences between the list of commercial and open source solvers, there are numerous case studies providing performance analysis comparison between the solvers. One such case study in [20] investigated the performance comparison between the following solvers: CPLEX (12.4.0.0), Gurobi (4.6.1), SCIP-C (2.1.1 using CPLEX as LP-solver), SCIP-L (2.1.1 using CLP as LP-solver), SCIP-S (2.1.1 using CLP as LP-solver), CBC (2.7.4), XPRESS (7.2.1), GLPK (4.47), and LP_SOLVE (5.5.2). In conclusion, the study found GLPK had the best performance of the open source solvers after scaling the running times of all test cases. Although GLPK was clearly slower than CPLEX and Gurobi, GLPK is proven to be the highest performing free, open source solver currently available.
Chapter 3 – Implementation and Problem Formulation

3.1 Chapter Overview

Chapter 3 investigates the implementation and problem formulation of the Mixed Integer Linear Programming (MILP) based path-planning algorithm described in this thesis. Section 3.2 explains the different visibility graph algorithms used in determining the visibility or connectivity of the obstacle field. Section 3.3 describes the MILP problem formulation mathematically, listing the different constraints and objectives of the given problem.

3.2 Visibility Graph Implementation

The visibility graph implementation consists of four separate problems containing their own unique algorithms. The first case comprises of any triangular obstacle containing three edges and three obstacle vertices. Considering this case, the visibility can be defined as the following: given any triangular obstacle \( O_T \), defined by the set of vertices \( V \) and the set of edges \( E \) connecting the vertices, the visibility from each vertex \( v_i \) to all other vertices \( v_j \) within the set are considered visible. Furthermore, it can also be concluded that the visibility for any given adjacent vertices sharing an obstacle edge is visible. This fact greatly simplifies the algorithm for the second case, which consists of any convex simple polygon \( O_{cx} \) with a set of edges \( E > 4 \). The general algorithm for determining visibility for this case is described in Algorithm 3.
Algorithm 3 Determining visibility for convex obstacles
given a convex simple polygon $O_{cx}$ with edges $E(\epsilon_0, \epsilon_1, \ldots, \epsilon_{n-1})$, $n > 4$ and
vertices $V(v_0, v_1, \ldots, v_{n-1})$
for each vertex $v_i \in V$ do
  for each vertex $v_j \in V$ != to vertex $v_i$ do
    if vertex $v_j$ is adjacent to vertex $v_i$ then
      $v_j$ is visible from $v_i$
    else
      $v_j$ is not visible from $v_i$
    end if
  end for
end for

The third case, as it pertains to the visibility graph, consists of examining concave
simple polygon obstacles $O_{cv}$. This is a much more complex problem compared to that
of case one and two, since adjacent vertices does not satisfy all visible paths form vertex
to vertex. A simple polygon triangulation algorithm known as Ear Clipping is used to
solve this problem. The Ear Clipping algorithm discussed in this paper is a slightly
modified version of Meister’s algorithm conducted by Rourke, and runs at $O(n^2)$ time
[28]. Additionally, Rourke’s algorithm is considered for its ease of implementation and
straightforwardness. However, it is important to note that other algorithms with better
asymptotic order exist. An ear of a polygon is defined as a triangle formed by three
consecutive vertices $v_{i_0}, v_{i_1}, v_{i_3}$ for which $v_{i_1}$ is a convex vertex with the interior angle at
the vertex is smaller than $\pi$ radians. Furthermore, the line segment from $v_{i_0}$ to $v_{i_2}$ lies
entirely inside the polygon and no vertices from the set of vertices defined by polygon
$O_{cv}$ are contained within the triangle [29]. Note; the notation $O_{cv}$ used for describing
simple convex polygons is used in describing this algorithm, even though this algorithm
can work for any simple polygon. This is because the triangulation algorithm is only
necessary for determining the visibility of simple convex polygons in regards to the path-
planning algorithm. An example of an ear is depicted in Figure 7, where the dashed line represents the interior line segment from vertices \( v_{i0} \) to \( v_{i2} \).

![Figure 7. Demonstrates an ear of a concave polygon](image)

Rourke’s *Ear Clipping* algorithm is described by four steps, which are listed via pseudo-code in Algorithm 4 and is described as follows. Step one consists of computing the interior angles on each vertex of \( O_{cv} \). If the interior angle of a vertex is less than 180°, the vertex is convex [28]. The next step is to find all ear tips of \( O_{cv} \) and initiate the ear tip status for each vertex according to the following condition: three consecutive vertices \( v_{i-1}, v_i, v_{i+1} \) of \( O_{cv} \) form an ear if 1) \( v_i \) is a convex vertex and 2) the triangle formed by \( v_{i-1}, v_i, v_{i+1} \) does not contain any interior vertices [28]. Step three is to select and delete the ear tip \( v_i \), and update the connection relationship, angle, and ear tip status for \( v_{i-1} \) and \( v_{i+1} \) [28]. The fourth and final step is to repeat step three until \((n - 2)\) triangles are constructed [28].
**Algorithm 4** Triangulation of a simple concave polygon (modified for path-planning algorithm)

Given a concave simple polygon $O_{cv}$ with edges $E(e_0, e_1, \ldots, e_{n-1}) \geq 4$ and the set of vertices $V(v_0, v_1, \ldots, v_{n-1})$, and a triangulation $T_\Delta(t_0, t_1, \ldots, t_{n-1})$ with $n - 2$ triangles

- Task compute interior angles of each vertex in $O_{cv}$ **end task**
- Task identify whether each vertex is an ear tip or not **end task**

**while** number of triangles in $T_\Delta < n - 2$ **do**

- find ear tip $v_i$ which has the smallest interior angle
- construct a triangle $\Delta(v_{i-1}, v_i, v_{i+1})$ and add it onto $T_\Delta$
- let $v_i$ no longer be an ear tip
- update connection of $v_{i-1}$ and $v_i$, and $v_i$ and $v_{i+1}$, and $v_{i+1}$, and $v_i$ and $v_{i-1}$
- refresh the ear tip status of $v_{i-1}$ and $v_{i+1}$

**end while**

Once the polygon or obstacle has been cut into triangles the visibility can be determined. This process consists of four iterative loops for iterating through the obstacle vertices, the obstacle edges, and the triangle line segments generated in the triangulation algorithm. Essentially, iterating through each vertex of the obstacle $O_{cv}$ and creating a linear path to all other obstacle vertices can determine the visibility. If the linear path intersects a triangular line segment located within the interior of the obstacle or the linear path intersects an obstacle edge, the visibility is deemed false. Otherwise, the visibility can be concluded true. However, it is important to note, if the linear path or line segment is equal to that of an obstacle edge or triangular line segment, then the visibility is not considered false. Another way of describing this exception is by referring to the basic principal of adjacent vertices; in that, any two adjacent vertices that share the same edge are visible. **Algorithm 5** describes the iterative processes and tasks associated with this visibility problem.
Algorithm 5 Determining visibility based on triangulation
given a concave simple polygon $O_{cv}$ with edges $E(e_0, e_1, \ldots, e_{n-1}) > 4$ and vertices $V(v_0, v_1, \ldots, v_{n-1})$, and a triangulation $T_{\Delta}(t_0, t_1, \ldots, t_{n-1})$ with $n - 2$ triangles
for all obstacle vertices $v_i \in V$ do
  for all obstacle vertices $v_j \in V \neq v_i$ do
    calculate linear path $LP$ from $v_i$ to $v_j$
    for all obstacle edges $e_i \in E$ and triangular line segments $t_i \in T_{\Delta}$ do
      if $LP$ intersects $e_i || LP$ intersects $t_i$ && $LP$ is not an adjacent vertex path then
        $v_j$ is not visible from $v_i$
      else
        $v_j$ is visible from $v_i$
      end if
    end for
  end for
end for

Thus far, the visibility has been found for vertices shared by the same obstacle. The fourth and final case in completing the visibility graph of the obstacle field comprises of looking from the perspective of obstacle vertices to different obstacle vertices. Similarly to Algorithm 5, this case can be solved through an iterative process. A linear path $LP$ needs to be formulated between Each vertex $v_i$ and all other vertices $v_j$ within the set $V$, in respect to the set of all obstacles $O(O_T, O_{cx}, O_{cv})$. If the linear path $LP$ intersects any obstacle edge $e_i$ within the set of obstacle edges $E$, for all obstacles $O(O_T, O_{cx}, O_{cv})$, then the path can be declared false or not visible. This is further explained below in Algorithm 6.
Algorithm 6 Determining visibility from obstacle to obstacle
given obstacles $O(o_0, o_1, ..., o_n)$ with edges $E(e_0, e_1, ..., e_{n-1})$ and vertices
$V(v_0, v_1, ..., v_{n-1})$

for all obstacle vertices $v_i \in V$ in respect to all obstacles in $o_i \in O$ do
    for all obstacle vertices $v_j \in V$ in respect to all obstacles in $o_j \in O \neq o_i$ do
        calculate linear path $LP$ from $v_i$ to $v_j$
        for all obstacle edges $e_l \in E$ in respect to all obstacles $o_k \in O$ do
            if $LP$ intersects $e_l$ then
                $v_j$ is not visible from $v_i$
            else
                $v_j$ is visible from $v_i$
            end if
        end for
    end for
end for
end for

The visibility from source to all obstacle vertices and target, along with the visibility
from target to all obstacle vertices and source can be solved in a very similar manner. A
slight modification can be made to Algorithm 6 where every edge and vertex in respect to
the obstacles is iterated through. After this final process the visibility graph for the entire
obstacle field is complete.

3.3 Mixed Integer Linear Programming Problem Formulation

The basic path-planning problem of determining the shortest route in real-time for
complex obstacle fields consists of the minimization of a linear cost function. Thus, the
shortest path problem comprises of finding a minimum-cost path between two nodes $s$
(source) and $t$ (target), given an undirected graph $G = (V, A)$ and the arc costs $c_{i,j}$
associated with the flight-path from vertex $i$ to vertex $j$. Let $V$ denote the set of vertices
described in the visibility formulation, $A$ denote the set of arcs or paths, and vertices
$i,j \in A$. It is important to note, arcs are considered in this paper due to the curvature of
the earth even though the obstacle field is represented as a planar 2-dimensional grid.
The Haversine formula is used for calculating these geospatial arc costs $c_{i,j}$ [30]. A path is defined as a sequence of vertices $v_1, \ldots, v_n$, and is defined as elementary if no vertex appears in the path more than once [31]. The set of outgoing and incoming arcs of vertex $i$ are denoted by $\delta^+(i)$ and $\delta^-(i)$ [31]. Furthermore, $\delta^+(S)$ and $\delta^-(S)$ represent the arcs leaving and entering the set $S \subseteq V$, and $A(S)$ represent the set of arcs with both ends in $S \subseteq V$ [31]. In all case formulations, it is assumed w.l.o.g. that $|\delta^-(i)| = |\delta^+(i)| = 0$ [31]. In mathematical terms, the MILP shortest path problem is further expressed as the following:

$$\min \sum_{(i,j) \in A} c_{i,j} x_{i,j} \quad (1)$$

subject to

$$\sum_{(i,j) \in \delta^+(i)} x_{i,j} - \sum_{(j,i) \in \delta^-(i)} x_{j,i} = \begin{cases} 1 & \text{if } i = s \\ -1 & \text{if } i = t \\ 0 & \text{else} \end{cases} \quad \forall i \in V \quad (2)$$

$$\sum_{(i,j) \in \delta^+(i)} x_{i,j} \leq 1 \quad \forall i \in V \quad (3)$$

$$x_{i,j} \in \{0,1\} \quad \forall (i,j) \in A \quad (4)$$

The objective function is represented by (1), where the objective is to minimize the total flight path cost from source to target. The arc costs are denoted by $c_{i,j} \in \mathbb{R}$ and the binary decision variables are represented by $x_{i,j}$. The binary decision variables shown in (4), declare whether the path from $v_i$ to $v_j$ is taken. If $x_{i,j} = 1$ the arc $(i,j)$ belongs to the path and conversely, if $x_{i,j} = 0$ the arc $(i,j)$ does not belong to the path. Constraints (2) are flow conservation constraints, while constraints (3) warrant that the
The flow constraints in (2) account for three different cases as it pertains to the outgoing and incoming arc flight paths as shown in Figure 8. The first case represents an outgoing flight path from the source node $S$ with the value equal to 1. Conversely, the second case represents an incoming flight-path to the target node $T$ with the value -1. The final case describes all other flight-paths associated with the obstacle vertices; in that, all obstacle vertices comprise of both incoming and outgoing flight-paths. Thus, if the vertex has both incoming and outgoing flight-paths the value is equal to 0 (-1 (incoming) + 1 (outgoing) = 0).

Figure 8. Flow constraints representation
Chapter 4 – Hardware and Software Platforms

4.1 Chapter Overview

This chapter provides information in regards to the hardware and software of the overall UAS. Section 4.2 provides background information regarding the Unmanned Aerial Vehicle (UAV) used for real-world flight-testing and a high level overview of the architecture of essential components. Section 4.3 explains the basic functionality of the Ground Control Station (GCS) and role it plays in the providing the operators a Graphical User Interface (GUI) of the current and future state of the plane. Section 4.4 describes the RAMS simulator and its influence in flight simulation and testing. Section 4.5 briefly discusses the custom, in-house Flight Control System designed and developed in the Virginia Commonwealth University UAV laboratory. Lastly, Section 4.6 covers the Mission Control System (MCS) and how the path-planning algorithm is integrated within the overall UAS, internally and externally. This section will also describe the communication protocol used for transferring information along with the MCS microcontroller chosen for real-time MILP path-planning implementation.

4.2 Unmanned Aerial Vehicle, Skyhunter EP

The unmanned aerial vehicle (UAV) used for real-world flight testing presented in Chapter 6, is performed on an electric powered (EP) Skyhunter with an approximate flight time of 20 minutes depending upon wind, weather, and the payload [32]. The Skyhunter EP was assembled and modified for UAS operations in the Virginia Commonwealth University UAV laboratory. The payload consists of an onboard custom
FCS and separate MCS, which will be further discussed in this chapter. Figure 9 shows the Skyhunter EP used for real-world flight-testing.

![Image of Skyhunter EP]

**Figure 9. Skyhunter EP used for real-world flight-testing**

The Skyhunter EP can be controlled either autonomously by the GCS, or manually where a safety-pilot is in radio control of the UAV. The ability to switch between autonomous and manual flight is vital to ensuring safety in the airspace and on the ground. The three core components used in the overall UAS consist of the MCS and FCS embedded on the plane, and the GCS, which is communicating via a wireless modem. A high-level architectural overview of main components and their interconnections for the overall UAS is given in Figure 10. It is important to note, the system described in Figure 10 represents the real-world flight-testing configuration, in which the MCS and FCS are embedded within the plane. The GCS communicates with the MCS through the User
Datagram Protocol (UDP), where the MCS is responsible for transferring packets to and from the GCS and FCS. The GCS achieves wireless communication through a *BULLETM* wireless modem.

![System Architecture Diagram]

**Figure 10.** High-level system architecture of the UAS used in flight-testing

### 4.3 Ground Control Station

The Ground Control Station (GCS) has been developed over the past fifteen years within the Virginia Commonwealth University UAV laboratory and is an essential component in autonomous flight. However, just recently a newer version has been created within the past year using the Qt platform allowing for modularity and use among multiple operating systems and application interface. The GCS incorporates a visual map that displays the current position of the UAV(s) and their waypoints, along with flight characteristics such as heading, attitude, airspeed, altitude, etc., which can be seen in Figure 11. For the purposes of this thesis, manual obstacle generation along with
updating the solved waypoint flight-path from source to target was added to the GCS platform. This is also shown in Figure 11, where the red transparent polygons represent manually entered obstacles and the waypoint triangle represents the waypoint flight-path from source to target. Also, notice the line segment that penetrates the obstacle is considered the direct path from source to target. However, this flight path is not considered due to the obstacle penetration.

![GCS Control Screen](image)

**Figure 11. GCS obstacle avoidance being executed**

### 4.4 RAMS Simulator

The RAMS simulator is a multiple agent, low-fidelity discrete-event simulator, designed and implemented by the Virginia Commonwealth University UAV research
laboratory to preform simulations in the field of collaborating agents [33]. Although the RAMS simulator has the ability to handle multiple agents or planes, only one agent is considered in the given path-planning problem. Figure 12 and Figure 13 demonstrate the modular architecture of the RAMS simulator given an internal or external Mission Control System (MCS). The difference between the internal and external MCS configurations is that the internal MCS is run on a desktop workstation whereas the external MCS is run on a microcontroller. Section 4.6 will go more in depth as it pertains to the MCS and its role in the overall UAS.

Figure 12. RAMS simulator featuring internal MCS

Figure 13. RAMS simulator featuring external MCS
The RAMS simulator is comprised of three different modules; which are the network, the MCS, and the agent module. The network module is capable of both low-fidelity mesh networking and high-fidelity wifi or mesh networking when connected to ns-3 [34]. Ns-3 was not used in this thesis but more information regarding the integration of Ns-3 with the RAMS simulator can be found in [34]. The agent module is used to model the high-fidelity aerodynamics of the UAV when connected to the open-source simulator called FlightGear. The RAMS simulator launches a FlightGear instance for each agent or UAV in the simulation. FlightGear sends both native-ctrl and native-fdm packets to the RAMS simulator while the RAMS simulator only sends native-fdm packets to FlightGear. Communication between the RAMS simulator and FlightGear is at a rate of 50 Hz and the packet information is in big endian form.

4.5 Flight Control System

The Flight Control System (FCS) is a custom, in-house designed and built hardware platform conducted over the past decade by the Virginia Commonwealth University UAV laboratory. The latest version of the FCS, known as the Aries v1.1, was designed and built by a graduate student as part of a master’s thesis [35]. The Aries v1.1 uses a STM32F4 ARM Cortex-M4F processor, which features a single-precision floating-point unit. The datasheet for this processor can be found here [36]. Some essential system specifications regarding the FCS include: an on-board IMU [37] and a fully integrated sensor suite. The Aries v1.1 includes a high level of connectivity containing multiple timer capture compare channels, UARTs, I2C, and SPI capability.
Figure 14 depicts the top and bottom sides of the Aries v1.1 FCS. Its dimensions are 1.5 inches by 3.2 inches, allowing for a small and compact FCS.

![Aries v1.1 FCS](image)

**Figure 14. Aries v1.1 (top side on left and bottom side on right)**

### 4.6 Mission Control System

The Mission Control System (MCS) is software created by a Virginia Commonwealth University PhD graduate student, as part of his dissertation in 2013 [38]. The MCS software is fully written in C++ and uses Object Oriented Programming (OOP) wherever possible [38]. The original intention of the MCS software framework was for the execution of task allocation algorithms, but has evolved into a system for running real-time embedded, mission-based UAS algorithms. A modification or “skeleton” of the MCS was used to help run the path-planning algorithms discussed in this thesis, along with handling the pass-through communications between the FCS and GCS. As
previously shown in Figure 12 and Figure 13, the MCS can be either internally or externally run depending upon the use of a desktop PC or microcontroller, respectively. Section 4.6.2 covers the microcontroller used to implement the external MCS for a real-time solution to the path-planning problem. Examples of GCS to FCS communication data include waypoint command changes for latitude, longitude, and altitude. Similarly, examples of FCS to GCS communication data would include the report information regarding the current position, altitude, or orientation of the UAV. Obstacle data is handled in this manner; in that, obstacles can be manually entered by the GCS operator and sent through the MCS to the FCS. The MCS is responsible for handling this communication and modifying packets if needed. For instance, if obstacle avoidance is engaged at the GCS level, waypoint information packets will be halted and modified with an updated flight path before continuing to the FCS. The updated flight path is the solved flight path found from the MILP path planning algorithms presented in this thesis. A block diagram is provided in Figure 15 to help further clarify the MCS software architecture with the integration of the MILP based path-planning algorithm.
Figure 15. MCS software architecture modified for path-planning algorithms

After the initialization routine, three threads are spawned, the main thread along with two listener threads for Ethernet and UDP sockets. The main thread is responsible for processing the incoming VACS packets (refer to Section 4.6.1) received and passing any waypoint and obstacle information to the MILP path-planning solver. The path-planning algorithm is launched whenever the UAV reaches within a certain distance of the current target command waypoint determined by the GCS operator. It is important to note the path-planning algorithm does not launch when traversing the solved flight-path waypoints. Hence, the difference between command waypoints inputted at the GCS level and the flight-path waypoints solved in the Main thread. Once the MILP path-planning
algorithm returns an optimal flight-path to the main thread, it updates the waypoint flight-path and sends it to both the FCS and GCS. The Logging stage of the main thread is pre-existing functionality in the MCS that allows for statistical data collection. The distance required to launch the path-planning algorithm is based on the time necessary to find a solution. Thus, the distance can be varied to account for the real-time flight-path calculations.

The software architecture for the MILP Path-planning Algorithm block shown in Figure 15 is broken into sub processes that handle the formulations of the obstacle field visibility graph and the MILP path-planning. Figure 16 demonstrates the progression from processing the incoming VACS packets in the Main thread, to storing the necessary obstacle and command waypoint information required for the path-planning algorithms. If obstacle and command waypoint packet information is received from the GCS, the MCS will proceed by formulating the visibility graph and MILP problem, calculating the MILP solution, and returning the solved flight-path to the Main thread. The MCS is then responsible for sending the newly updated waypoint sequence to the FCS and GCS. The solved flight-path will appear on the GCS map and the FCS will navigate the new sequence of waypoints.
The processing of obstacle data described in Figure 16 is stored as a two-dimensional vector, containing the individual obstacles and the vertex information for each of the respective obstacle vertices. Furthermore, a vertex ID is assigned to each of the obstacle vertices starting at the ID equal to 2. The source and target vertices are stored as one-dimensional vectors including the same information in terms of latitude, longitude, and vertex ID. However, the source vertex ID is always defined as 1 and the target vertex ID is always assigned the greatest node ID. Thus, if there are 99 vertices including the source, the target ID is equal to 100. Once the obstacle and waypoint information is known, the obstacle edge information is required before proceeding into the visibility graph formulation.
The obstacle edges are defined by the obstacle vertices and the line segments that connect them. In the *Obstacle Initialization* block shown in Figure 17, there are functions responsible for storing the obstacle and waypoint information, along with computing the edge equation for every edge of every obstacle. This information is stored in a two dimensional vector containing the slope, intercept, and two vertex IDs associated with forming each line segment. Once the edge information is attained, the algorithm proceeds to the visibility graph formulation shown in Figure 17. As previously described algorithmically in Chapter 3, the visibility graph algorithms are separated into four different cases: triangular obstacles, convex obstacles, concave obstacles, and obstacle-to-obstacle/ source or target to obstacle. After completing the visibility graph formulation, the next step is converting the adjacency matrix to an adjacency list as shown in Table 4.

![Figure 17. Visibility Graph algorithms software architecture](image-url)
The adjacency list comprises of the arc paths from vertex \( i \) to vertex \( j \) and is sorted according to the C++ standard library sort function. The reason for the conversion from an adjacency matrix to an adjacency list is due preference of an adjacency list in formulating the GLPK problem. However, before building the GLPK problem, the arc path distances must be computed and stored in the adjacency list, overriding the previous visible integer values of 1. This is shown on the right side of Table 4. It is also important to note, the non-visible arc path costs are considered null with a very large value equal to the fixed horizon as previously mentioned in Chapter 3. Once the distances or costs are known and stored in the adjacency list, the GLPK problem can be formulated.

Table 4. Sorted adjacency list without cost (left), with cost (right)
4.6.1 GLPK Software Implementation

The GLPK software primarily consists of using the GLPK Application Programming Interface (API) to formulate the problem, along with formulating the solution tour for the optimal path from source to target. Code-block 1 demonstrates the pseudo-code for formulating or building the problem according the number of vertices in the problem space.

**Code-block 1** Pseudo-code for building the GLPK problem - columns given the total number of vertices $v$ and adjacency vector pointer $adj$

```c
P = glp_create_prob( ) ;

for(i = 1; i <= v; i + 1){
    for(j = 1; j <= v; j + 1){
        k = glp_add_cols(P,1) ;
        glp_set_col_kind(P,k,GLP_BV) ;
        glp_set_obj_coef(P,k,adj->at(k-1).cost) ;
    }
}
```

The first step in formulating the problem is calling the `glp_create_prob` function, which creates a new problem object. The new problem object is initially empty, containing no rows or columns. Once the problem object is declared, the columns are added to the problem according to the two loops shown in Code-block 1. The total number of columns is the number of vertices in the problem space squared. Furthermore, the column kind, otherwise known as the variable type for the columns, is set to a binary variable. This is due to the use of binary decision variables in the given problem. It is also important to note the column coefficients are represented by the costs stored in the adjacency list. This is shown by the function `glp_set_obj_coef`.

45
Once the columns have been established in the problem, the rows are added to the problem, accounting for all degree constraints, which is shown in Code-block 2

**Code-block 2** Pseudo-code for building the GLPK problem - rows given the total number of vertices $v$

```
for (i = 1; jj = 1; i ≤ v; i + 1, jj + 1){
    kk = glp_add_rows(P, 1);
    if (i == 1){
        // case 1
        glp_set_row_bnds(P, kk, GLP_FX, 1, 1);
    } else if (i == v){
        // case 2
        glp_set_row_bnds(P, kk, GLP_FX, -1, -1);
    } else{
        // case 3
        glp_set_row_bnds(P, kk, GLP_FX, 0, 0);
    }
}
```

The function `glp_set_row_bnds` fixes the bounds according to the flow conservation constraint cases described in Chapter 3. For the first case, the row bounds are set to the value of one, accounting for an outgoing flight-path from the source node. Similarly, for case two, the row bounds are fixed to negative one to account for the target having an incoming flight-path. The last case, accounts for all other vertices having incoming and outgoing flight-paths; in that, the row bounds equal zero.
4.6.2 VACS Communication Protocol

The next aspect of the MCS and the overall UAS is the communication protocol used for transferring information. The Virginia Commonwealth University UAV laboratory follows the internal VCU Aerial Communications Standard (VACS). The VACS packet format is shown in Table 5.

<table>
<thead>
<tr>
<th>Byte</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sync 1</td>
<td>First synchronization byte</td>
</tr>
<tr>
<td>2</td>
<td>Sync 2</td>
<td>Second Synchronization byte</td>
</tr>
<tr>
<td>3</td>
<td>Destination</td>
<td>Destination of packet (tail number)</td>
</tr>
<tr>
<td>4</td>
<td>Source</td>
<td>Source address of packet (tail number)</td>
</tr>
<tr>
<td>5</td>
<td>Msg. ID H</td>
<td>Unique message ID (high byte)</td>
</tr>
<tr>
<td>6</td>
<td>Msg. ID L</td>
<td>Unique message ID (low byte)</td>
</tr>
<tr>
<td>7</td>
<td>Data length H</td>
<td>Length (N) of data field (high byte)</td>
</tr>
<tr>
<td>8</td>
<td>Data length L</td>
<td>Length (N) of data field (low byte)</td>
</tr>
<tr>
<td>9 + 9 + (N-1)</td>
<td>Data field</td>
<td>Payload of message</td>
</tr>
<tr>
<td>9 + N</td>
<td>Checksum 1</td>
<td>First checksum (Fletcher’s)</td>
</tr>
<tr>
<td>10 + N</td>
<td>Checksum 2</td>
<td>Second checksum (Fletcher’s)</td>
</tr>
</tbody>
</table>

Table 5. VACS packet format

For this thesis, a specific message ID equal to 311 is given for obstacle packet information. Obstacle packet information includes a reset byte, the total number of obstacles, and obstacle specific information, such as the latitude, longitude, and altitude of each corner vertex of a given obstacle. The obstacle packet information is described as an Extensible Markup Language (XML) file and is shown in Figure 18.
The external MCS microcontroller chosen for this thesis is a Raspberry Pi 2 Model B. The Raspberry Pi 2 Model B was mainly chosen due to its sophisticated processing ability and large memory capacity of 1GB. A further reason for choosing the
Raspberry pi 2 Model B is due to its ease of use and abundant documentation online. Some essential specifications for the microcontroller are shown in Table 6 along with the connector information in Table 7.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Broadcom BCM2836 SoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Architecture</td>
<td>Quad-core ARM Cortex-A7</td>
</tr>
<tr>
<td>CPU</td>
<td>900 MHz</td>
</tr>
<tr>
<td>GPU</td>
<td>Dual Core VideoCore IV Multimedia Co-Processor</td>
</tr>
<tr>
<td>Memory</td>
<td>1GB LPDDR2</td>
</tr>
<tr>
<td>Operating System</td>
<td>Boots from Micro SD card</td>
</tr>
<tr>
<td>Dimensions</td>
<td>85x56x17mm</td>
</tr>
<tr>
<td>Power</td>
<td>Micro USB socket 5V, 2A</td>
</tr>
</tbody>
</table>

**Table 6. Raspberry pi 2 Model B specifications**

<table>
<thead>
<tr>
<th>Ethernet</th>
<th>10/100 BaseT Ethernet Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Video Output</td>
<td>HDMI (rev 1.3 &amp; 1.4)</td>
</tr>
<tr>
<td>Audio Output</td>
<td>3.5mm jack, HDMI</td>
</tr>
<tr>
<td>USB</td>
<td>4 x USB 2.0 Connector</td>
</tr>
<tr>
<td>GPIO Connector</td>
<td>40-pin 2.54mm expansion header providing (2x20 strip)</td>
</tr>
<tr>
<td>Camera Connector</td>
<td>15-pin MIPI Camera Serial Interface (CSI-2)</td>
</tr>
<tr>
<td>JTAG</td>
<td>Not populated</td>
</tr>
<tr>
<td>Display Connector</td>
<td>Display Serial Interface (DSI) 15 way flat flex cable connector</td>
</tr>
<tr>
<td>Memory Card Slot</td>
<td>Micro SDIO</td>
</tr>
</tbody>
</table>

**Table 7. Raspberry Pi 2 Model B Connectors**
The preferred operating system for Raspberry Pi microcontrollers is Raspbian. However, there are numerous third party operating system images including Ubuntu Mate, Snappy Ubuntu Core, Windows 10 IOT Core, and OSMC to name a few [39]. Ubuntu Mate 16.04 was chosen as the operating system due to familiarity with Ubuntu and personal preference with the overall operating system.
Chapter 5 – Testing and Results

5.1 Chapter Overview

This chapter investigates the performance of the MILP based path-planning algorithm presented in the paper with respect to execution time and correctness. Correctness is defined by the achievement of obstacle avoidance along with an optimal flight-path solution from source to target. Section 5.2 describes the simulation and testing process used throughout this Chapter. Section 5.3 provides simulation results conducted in laboratory while Section 5.4 provides real-world results from flight-testing. The aircraft used in these experiment is a fixed-wing aircraft and will be later discussed in Section 5.5.

5.2 Simulation and Testing Process

The process used for testing the algorithms presented in this thesis begins with the entry of obstacles in the GCS by its operator. The obstacles can be either entered one at a time by clicking on the obstacle icon (traffic cone) or by loading saved obstacle fields by clicking on the File tab. Both methods for entering obstacles are shown in Figure 19, where the left side displays the obstacle icon and the right side shows the tab for loading obstacles. For entering obstacles one at a time via the obstacle icon, the GCS operator must right click and apply the obstacle before the obstacle is recognized by the GCS and sent to the MCS for further processing. The right-click display for applying an obstacle is shown in Figure 20.
Also note in Figure 20, the GCS operator has the ability to cancel the current obstacle or clear all the obstacles from the GCS map.

Once the obstacle field is determined and all obstacles are applied, the next step is to enter a sequence of command waypoints indicating the source and target vertices for
the flight-path. Similarly to applying obstacles, the command waypoints are applied by clicking on *Apply waypoints* as shown in Figure 20. Applying the command waypoints will trigger the MCS to solve the optimal flight-path with obstacle avoidance, and display the flight-path on the GCS map. An example of a solved flight-path while avoiding a triangular obstacle is shown in Figure 21. Note, the source is indicated by $wp[1,0]$ and the target is indicated by $wp[1,2]$. The straight line shown connecting the source and target, indicates the optimal route without obstacle avoidance. Once the UAV has reached within 100 meters of target, the target becomes the source and the next command waypoint in the sequence is the target. The MCS then proceeds to recalculate the flight-path and update the GCS. This process loops through until the GCS operator indicates otherwise.

![Figure 21. Example of solved flight-path](image)

Figure 21. Example of solved flight-path
5.3 Simulation Results

The simulations conducted in laboratory consist of algorithm testing, internal MCS configuration testing, and external MCS configuration testing. Before discussing the performance of the MILP based path-planning algorithm presented in this thesis, the visibility graph correctness is examined. For this section, an internal MCS configuration was used along with the GCS for obstacle generation. The next section investigates the performance of the Exhaustive, Breadth-first Search relative to the MILP based path-planning algorithm presented in this thesis. Pre-determined, grid based obstacle fields that vary in size, are used in this comparison. The grid based obstacle field is created by generating squares and triangles of random sizes, angles, and quantity. The platform used for this section consists of running the algorithms independently from the MCS on a desktop PC. The next two sections examine the internal and external MCS configurations, which uses the processing power of a desktop PC or Raspberry Pi 2 Model B, respectively.

5.3.1 Visibility Graph Correctness

The visibility graph correctness is defined by performance in obstacle avoidance for numerous complex cases. The test cases were chosen based the algorithms used in Section 3.2 for determining the visibility for convex and concave obstacles, along with obstacle-to-obstacle visibility. The first test case shown in Figure 22 demonstrates successful obstacle avoidance and visibility for a triangular obstacle. Notice the plane (named plane 1) is flying the solved flight-path around the obstacle from command waypoint wp[1,0] (source) to command waypoint wp[1,2] (target). As a reference, a line
segment from source to target represents the shortest path through the obstacle, which is not flown. Similarly, Figure 23 demonstrates successful obstacle avoidance for a convex obstacle. The solved flight-path consists of the following waypoint sequence: \( wp[1,0] \) (source) -> \( wp[1,1] \) -> \( wp[1,2] \) -> \( wp[1,3] \) (target).

**Figure 22. Obstacle avoidance for a triangle**
Figure 23. Obstacle avoidance for a simple convex polygon

The next figures display successful obstacle avoidance and visibility graph correctness for simple concave polygons. Both Figure 24 and Figure 25 demonstrate a scenario in which the source node is located within a cutout of a polygon (command waypoint \(wp[1,0]\)). As shown in both figures, the solver successfully determines a flight-path out of the cutout and around the obstacle. The solved waypoint flight-path is as follows: \(wp[1,0]\)(source) \(\rightarrow\) \(wp[1,1]\) \(\rightarrow\) \(wp[1,2]\) \(\rightarrow\) \(wp[1,3]\) \(\rightarrow\) \(wp[1,4]\)(target).
Figure 24. Obstacle avoidance for a simple concave polygon (example 1)

Figure 25. Obstacle avoidance for a simple concave polygon (example 2)
The third aspect of testing the visibility graph correctness consists of obstacle-to-obstacle visibility or obstacle avoidance. A tunnel between two simple concave polygons was chosen to test extreme cases in which the solver would have to negotiate sharp turns and corners. Figure 26 depicts a simple S-shaped tunnel for the plane to fly while Figure 27 demonstrates a much more difficult tunnel-path for the plane to negotiate. In both test cases, obstacle avoidance and visibility was proven to be correct. The solved flight-path from source to target in Figure 26 is as follows: wp[1,0](source) -> wp[1,1] -> wp[1,2] -> wp[1,3] -> wp[1,4] -> wp[1,5](target). Correspondingly, the solution flight-path for Figure 27 is: wp[1,0](source) -> wp[1,1] -> wp[1,2] -> wp[1,3] -> wp[1,4] -> wp[1,5] -> wp[1,6] -> wp[1,7] -> wp[1,8] -> wp[1,9](target).

Figure 26. Obstacle avoidance for tunnel (example 1)
The final two test cases consist of a random number of simple convex and concave obstacles placed by the GCS operator. Although only two of the random test cases are shown in Figure 28 and Figure 30 respectively, hundreds of randomized cases have been tried and tested throughout the testing and simulation phase of this thesis. Thus far, not a single randomized case tested has broken the visibility or path-planning based algorithms presented in this thesis. The solution waypoint flight-path for Figure 28 is described as follows: \( wp[1,0](\text{source}) \rightarrow wp[1,1] \rightarrow wp[1,2] \rightarrow wp[1,3] \rightarrow wp[1,4] \rightarrow wp[1,5] \rightarrow wp[1,6](\text{target}) \). Likewise, The waypoint flight-path solution for Figure 29 is the same.
Figure 28. Obstacle avoidance for random obstacle field (example 1)

Figure 29. Obstacle avoidance for random obstacle field (example 2)
5.3.2 Path-planning Algorithm Performance Independent of MCS

This section provides simulation performance in terms of execution time and path-planning optimality in regards to the Exhaustive, Breadth-first Search algorithm described in Section 2.2.2 relative to the MILP based path-planning algorithm described in this thesis. As previously mentioned, both algorithms were implemented in C and C++ and use the same visibility graph formulations described in Section 3.2. The first test case consists of four obstacles (two squares and two triangles) and is shown graphically in Figure 30. Subsequent figures demonstrate the flight-path solutions for increasing obstacle fields up to one hundred obstacles.

Figure 30. Test case 1, 4 obstacles, total distance = 386.35 meters
Figure 31. Test case 2, 9 obstacles, total distance = 485.28 meters

Figure 32. Test case 3, 16 obstacles, total distance = 1087.88 meters
Figure 33. Test case 4, 25 obstacles, total distance = 1353.50 meters

Figure 34. Test case 5, 64 obstacles, total distance = 2173.14 meters
An additional test case was conducted with four hundred obstacles but is not pictorially included due to size and constraints of this document. Once the visibility graph is formulated, the execution time is determined from the start of the MILP algorithm to its completion of the optimal path from source to target. Correspondingly, the execution time is recorded from the start of the Exhaustive Search algorithm to the end in finding the optimal path. The algorithm execution times are compared, in addition to the optimality of the flight path. It is important to note, the problem space is not only defined by the number of obstacles, but the total number of vertices attributed to the obstacles. Table 8 below shows the execution time results of the seven scenarios running on a 2.5 GHz Intel Core i7 workstation. Table 9 lists the total cost, or flight path distance in meters from
source to target. It is important to note for the Exhaustive Search implementation, test cases 4 through 7 required curve fitting estimation for execution times due to the extreme runtime lengths. In addition to the path-planning algorithm computation times, visibility graph computation time was taken as a gauge for real-time, in-flight execution ability. Table 10 lists the execution time in seconds relative to the number of obstacles in each test case.

<table>
<thead>
<tr>
<th>Case</th>
<th>Number of Obstacles</th>
<th>GLPK Execution Time [sec]</th>
<th>Exhaustive Search Execution Time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>&lt;0.001</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>&lt;0.001</td>
<td>1024.95</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>&lt;0.001</td>
<td>5410.20</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>0.01</td>
<td>16106.1*</td>
</tr>
<tr>
<td>5</td>
<td>64</td>
<td>0.19</td>
<td>128205*</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>0.84</td>
<td>326513*</td>
</tr>
<tr>
<td>7</td>
<td>400</td>
<td>55.09</td>
<td>5519480*</td>
</tr>
</tbody>
</table>

* Notes $O(\log(N^2))$ curve fitting of $y = 35.123x^2 - 251.61x + 444.46$

Table 8. Execution time in seconds relative to the number of obstacles for each algorithm
<table>
<thead>
<tr>
<th>Case</th>
<th>Number of Obstacles</th>
<th>GLPK Total Cost [m]</th>
<th>Exhaustive Search Total Cost [m]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>386.35</td>
<td>386.35</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>485.28</td>
<td>485.28</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>1087.88</td>
<td>1087.88</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>1353.50</td>
<td>1353.50</td>
</tr>
<tr>
<td>5</td>
<td>64</td>
<td>2173.14</td>
<td>N/A</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>2506.18</td>
<td>N/A</td>
</tr>
<tr>
<td>7</td>
<td>400</td>
<td>5800.35</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 9. Total cost in meters relative to the number of obstacles for each algorithm

<table>
<thead>
<tr>
<th>Case</th>
<th>Number of Obstacles</th>
<th>Visibility Graph Execution Time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>0.03</td>
</tr>
<tr>
<td>5</td>
<td>64</td>
<td>0.46</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>1.74</td>
</tr>
<tr>
<td>7</td>
<td>400</td>
<td>110.83</td>
</tr>
</tbody>
</table>

Table 10. Execution time relative to the visibility graph algorithm

5.3.3 Internal MCS Simulations

The simulations pertaining to an internal MCS configuration consist of running the MCS, GCS, and RAMS simulator all on the same PC. Four test cases were investigated ranging from four obstacles to 25 obstacles. The obstacles were entered into the GCS at random and saved for comparison with the External MCS simulations shown...
in the next section. Both Save obstacles and Load obstacles functionality was added to the GCS to achieve this comparison between the internal and external MCS simulations.

The first test case consists of four obstacles and is shown pictorially in Figure 36. The MILP solution flight-path is as follows: \( wp[1,0] \) (source) \( \rightarrow wp[1,1] \rightarrow wp[1,2] \rightarrow wp[1,3] \) (target). The MILP algorithm execution time is less than 1 millisecond and the total cost or solution flight-path distance is 11839.48 meters. For the second test case, shown in Figure 37, nine obstacles are examined. As expected, the MILP algorithm execution time took slightly longer than test case one. This is primarily due to the increase in the number of obstacle nodes, which dramatically increases the adjacency list size and binary decision variables within the MILP problem formulation. The execution time took 0.02 seconds to complete and the total flight-path solution is 7,059.86 meters. Furthermore, the MILP solution flight-path is described as follows: \( wp[1,0](source) \rightarrow wp[1,1] \rightarrow wp[1,2] \rightarrow wp[1,3] \rightarrow wp[1,4] \rightarrow wp[1,5](target) \).
Figure 36. Internal MCS test case 1, 4 obstacles, total cost = 11839.48 meters

Figure 37. Internal MCS test case 2, 9 obstacles, total cost = 7059.86 meters
The third test case is shown in Figure 38 and comprises of sixteen obstacles. The solved waypoint flight-path is described as follows: \texttt{wp[1,0]}(source) -> \texttt{wp[1,1]} -> \texttt{wp[1,2]} -> \texttt{wp[1,3]} -> \texttt{wp[1,4]} -> \texttt{wp[1,5]} -> \texttt{wp[1,6]}(target). Contrary to expectation, the algorithm execution time is less than test case 2, taking 0.01 seconds to complete. This is primarily due to the similarity of the number of nodes in the problem space. Test case two contains 65 nodes versus 74 nodes in the test case three problem space. As it pertains to test case one, the problem space consisted of 30 nodes, which is significantly smaller. This justifies why test case one took much less than test cases two, three, and four, which will be discussed next.

Test case four contains 25 obstacles comprised of 109 nodes, which is shown in Figure 39. The MILP algorithm execution time is 0.03 seconds and the total solution flight-path distance is 2,274.12 meters. As expected, this test case took the longest and contains the largest adjacency matrix of the four test cases. The solution flight-path is described as follows: \texttt{wp[1,0]}(source) -> \texttt{wp[1,1]} -> \texttt{wp[1,2]} -> \texttt{wp[1,3]} -> \texttt{wp[1,4]} -> \texttt{wp[1,5]} -> \texttt{wp[1,6]} -> \texttt{wp[1,7]} -> \texttt{wp[1,8]}(target). The performance data from all four test cases is shown in Table 11, including the number of obstacles, GLPK execution time, and total cost.
Figure 38. Internal MCS test case 3, 16 obstacles, total cost = 16180.42 meters

Figure 39. Internal MCS test case 4, 25 obstacles, total cost = 2274.12 meters
<table>
<thead>
<tr>
<th>Case</th>
<th>Number of Obstacles</th>
<th>GLPK Execution Time [sec]</th>
<th>Total Cost [m]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>&lt;0.001</td>
<td>11839.48</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>0.02</td>
<td>7059.86</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>0.01</td>
<td>16180.42</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>0.03</td>
<td>2274.12</td>
</tr>
</tbody>
</table>

Table 11. Internal MCS Simulation performance data

5.3.4 External MCS Simulations

Similarly to the internal MCS simulations conducted in the previous section, the external MCS simulation test cases incorporate the same obstacle fields. All four test cases in this section contain the same solved flight-paths as previously mentioned in Section 5.2.3. The only differences between the two sections consist of the GLPK execution time and the total cost of the solved flight-path. The difference in the solution flight-path is due to the slight difference in manually placing the source and target command waypoints. Nonetheless, the command waypoints are placed similarly to Section 5.2.3 and the solution flight-path costs are very close.

Test case one, consisting of four obstacles and 30 nodes, is shown in Figure 40 while test case two consisting of nine obstacles and 65 nodes, is shown in Figure 41. The execution times for these test cases are 0.03 and 0.23 seconds respectively. Notice, the execution times are much longer in comparison to the previous section.
Figure 40. External MCS test case 1, 4 obstacles, total cost = 12113.95 meters

Figure 41. External MCS test case 2, 9 obstacles, total cost = 7392.66 meters
This is expected since the MCS is now running externally on the Raspberry Pi 2 Model B, containing worse processing and memory capability compared to the internal MCS simulations on a desktop workstation. The total costs for these test cases are 12,113.95 meters and 7,392.66 meters respectively. As previously mentioned the solution flight-paths are the same as the flight-paths indicated in Section 5.2.3 respective to the given test case.

Test case three, containing sixteen obstacles and 74 nodes, is shown in Figure 42. As expected the GLPK algorithm execution time is slightly slower than test case two, taking 0.27 seconds to complete. The solved flight-path is indicated in the previous section and the total cost flight-path distance is 15,969.34 meters. The final test case, consisting of 25 obstacles and 109 nodes, was unable to successfully work due to the memory limitations of the Raspberry Pi 2 Model B. For this given case problem space, the MCS requires greater than 925 MB to perform the necessary visibility graph, MILP calculations, and MCS operations. However, further optimization could greater enhance the overall algorithm, which will be discussed in Chapter 6. The results from the external MCS simulations are shown in Table 12 according to the four test cases.
Figure 42. External MCS test case 3, 16 obstacles, total cost = 15969.34 meters

<table>
<thead>
<tr>
<th>Case</th>
<th>Number of Obstacles</th>
<th>GLPK Execution Time [sec]</th>
<th>Total Cost [m]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>0.03</td>
<td>12113.95</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>0.23</td>
<td>7392.66</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>0.27</td>
<td>15969.34</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 12. External MCS simulation performance data

5.4 Real-world Results

The final section of Chapter 5 entails the real-world flight-testing and verification of the algorithms presented in this thesis. Flight-testing was conducted at a local RARC model aircraft flying field with regulated boundaries regarding the airspace. One such
restriction is the safety pilot must have line of sight (LOS) of the plane at all times. Thus, it was difficult to test large instances of obstacles varying in complexity. Nonetheless, test cases are provided for individual obstacles, two obstacles, and three obstacles varying in shape and size.

Test case one and two provided in Figure 43 consists of individual obstacles and the successful navigation with obstacle avoidance, from source to target within a command waypoint triangle. Both test case one and two finished their GLPK algorithm solutions in less than 0.001 seconds. Test case one had a total flight-path cost from source to target of 212.59 meters while the flight-path for test case two was 389.24 meters. The solved waypoint flight path for test one is described as the following: \( wp[5,0](source) \rightarrow wp[5,1] \rightarrow wp[5,2](target) \), while the solution for test two is as follows: \( wp[5,0](source) \rightarrow wp[5,1] \rightarrow wp[5,2] \rightarrow wp[5,3](target) \). Note, the first number within the \( wp \) brackets is the plane ID.

Figure 43. Real-world flight testing, test case 1 (left), test case 2 (right)
The next two cases shown in Figure 44 represent test case three and four. Test case three comprises of a simple concave polygon and a triangle, while test case four comprises of a triangle and a simple convex polygon. The execution time for the MILP algorithm solution was 0.01 seconds for both cases, taking slightly longer than test cases one and two as expected. The total cost for case three was 382.73 meters and 372.97 meters for case four. The solved waypoint flight-path for case three is as follows: \( wp[5,0](\text{source}) \rightarrow wp[5,1] \rightarrow wp[5,2] \rightarrow wp[5,3](\text{target}) \), while the flight-path for case four is: \( wp[5,0](\text{source}) \rightarrow wp[5,1] \rightarrow wp[5,2](\text{target}) \). Although the algorithms presented in this thesis were successful in solving an optimal waypoint flight-path, obstacle avoidance in case three did not work as well as in simulation. This was primarily due to the close proximity of the two obstacles; in that, the plane had a difficult time maneuvering the sharp banks required to follow the flight-path. Nonetheless, the plane achieved satisfactory obstacle avoidance given the limitations of the airspace and complexity of the obstacle field.

![Figure 44. Real-world flight testing, test case 3 (left), test case 4 (right)](image-url)
Test cases five and six consisted of further complexity with the addition of a third obstacle. Once again, both simple concave and convex polygons were examined to verify the correctness and success of the algorithms presented in this thesis. The execution time for finding the flight-path solution in both test cases was 0.01 seconds. The flight-path distance in case five is 463.26 meters and 227.35 meters in case six. The solution waypoint flight-path for case five and six is as follows: \( wp[5,0] \) (source) -> \( wp[5,1] \) -> \( wp[5,2] \) (target). Both cases were successful in solving a flight-path from source to target while achieving obstacle avoidance. This is primarily due to spacing the obstacles farther apart and maximizing the airspace as much as possible, considering the limited airspace.

![Real-world flight testing, test case 5 (left), test case 6 (right)](image)

**Figure 45. Real-world flight testing, test case 5 (left), test case 6 (right)**

The final case depicted in Figure 46 demonstrates the successful navigation from source to target through an obstacle tunnel formed by two obstacles. The GLPK algorithm execution time for this case is 0.03 seconds, which is the slowest time of all
seven test cases. The total flight-path cost is 44.36 meters and the waypoint flight path is as follows: \textit{wp}[5,0](source) \rightarrow \textit{wp}[5,1] \rightarrow \textit{wp}[5,2] \rightarrow \textit{wp}[5,3](target). In addition, the plane achieved obstacle avoidance throughout the flight, only clipping the outer edges of the obstacles. This is acceptable as previously mentioned in the Problem Statement under the assumption of a safety buffer between the physical obstacle and the GCS entered obstacle.

Lastly, the performance data for all seven test cases in terms of the number of obstacles, GLPK execution time, and total flight-path cost is given in Table 13.
<table>
<thead>
<tr>
<th>Case</th>
<th>Number of Obstacles</th>
<th>GLPK Execution Time [sec]</th>
<th>Total Cost [m]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>&lt;0.001</td>
<td>212.59</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>&lt;0.001</td>
<td>389.24</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0.01</td>
<td>382.73</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0.01</td>
<td>372.97</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0.01</td>
<td>463.26</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0.01</td>
<td>227.35</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>0.03</td>
<td>444.36</td>
</tr>
</tbody>
</table>

Table 13. Real-world testing performance data
Chapter 6 – Conclusions and Future Work

6.1 Conclusions

In this thesis, a MILP based navigation algorithm with obstacle avoidance for UAS has been introduced. The algorithms presented, in terms of modeling the visibility graph and the MILP problem formulation, have been proven to work in simulation and in real-world flight-testing. The MILP model achieves an identical solution to that of the Exhaustive, Breadth-first Search algorithm 100% of the time. With that said, only instances of 25 obstacles or less were able to be compared due to the limitations in computation time for the Exhaustive Search algorithm. In addition, there were not any differences in terms of computing the visibility of the obstacle field due to both algorithms incorporating the same visibility graph algorithms. The visibility graph algorithms were proved to be correct for all three cases of command waypoint (source and target) to obstacle(s) visibility, individual obstacle vertex visibility, and obstacle-to-obstacle vertex visibility. This allowed for successful obstacle avoidance given an obstacle safety buffer.

As it pertains to internal and external MCS simulation testing, there were not any differences in the respective flight-path solutions found. However, as expected, there were differences in GLPK execution time. On average, the external MCS simulation execution times increased by 2183.33% percent. This was due to the MCS algorithms running on a Raspberry Pi 2 Model B as opposed to a desktop workstation with much better processing ability. It is also important to note, the external MCS configuration could only handle up to 25 obstacles due to the memory limitations of the Raspberry Pi 2
Model B. Ways in which to improve larger instances of obstacles will be discussed in the next section. Nonetheless, the external MCS simulations verified the ability to model the connectivity of the obstacle field and find a solution flight-path all within an embedded system in real-time.

The verification of the external MCS working with the GCS and RAMS simulator lead to real-world flight-testing. Seven test cases were examined to prove the algorithms presented in this thesis perform in real-life versus only in theory. All seven cases successfully calculated a solution path from source to target while achieving obstacle avoidance per the assumption of a safety buffer. However with that said, test case three demonstrated the need for the MILP problem formulation to take into account UAV flight dynamics. This was primarily due to the close proximity of the obstacles and command waypoints, creating a difficult flight path for aircraft maneuverability. The next section will further discuss improvements that could be made to help with this problem and others mentioned in this thesis.

6.2 Future Work

Although the MILP algorithm presented in this thesis has shown true potential for autonomous navigation applications, there are numerous extensions and optimizations for improvement. First of which entails dynamic obstacle avoidance; in that, the position, heading, and velocity of the moving obstacle are taken into account to ensure a collision free flight-path. The addition of dynamic obstacle detection allows for a vast number of collaborative UAS applications, assuring collision avoidance for the UAVs within the network. Essentially, the UAVs would be considered as dynamic obstacles respective to
one another, and sampled at every time step. In addition to incorporating dynamic obstacles and a collaborative network of UAVs, there could be much improvement in the overall MILP problem formulation.

As previously mentioned from examining the real-world flight-testing cases, UAS flight dynamics could be incorporated as MILP constraints allowing for more precise aircraft maneuverability and obstacle and/or collision avoidance. Aircraft velocity, fuel or energy consumption, and the increase in number of nodes along the linear flight path are all possible additions to the problem formulation for optimizing the solution flight path. Another area of future work to be done consists of general code optimizations in regards to the visibility graph algorithms and possibly to the GLPK algorithm.

Currently, the visibility graph algorithms are running serially and could be parallelized, taking advantage of the four cores on the Raspberry Pi 2 Model B. Furthermore, memory optimizations in the MCS need to be made to help solve larger instances of obstacles. In addition to improving the visibility graph algorithm performance, there is the potential to parallelize the GLPK algorithm by separating the obstacle fields into quadrants and stitching the paths of each quadrant. Previous work had been done in solving the Traveling Salesman problem (TSP) using this method via stitching the Hamiltonian paths of several quadrants. Although the TSP is much simpler with respect to not having an obstacle field, it confirms the ability to parallelize the GLPK solver, enabling much opportunity for future research.
Bibliography


