Abstract

Unmanned Aerial Vehicles (UAVs) are becoming increasingly useful for several different applications. One issue faced in the growing popularity of UAVs is collision avoidance. With multiple aircraft in a finite airspace, some sort of algorithm must be used in order to avoid near misses and collisions while staying within reasonable bounds for a given mission. This paper proposes a new algorithm based on three methods of path finding: Dubins path, line of sight (LOS) maximization and a modified version of rapidly exploring random trees (RRT*). The algorithm is used in a realistic ROS based simulator to obtain results presented in this paper.

I. INTRODUCTION

The research that is being dealt with in this composition surrounds collision avoidance in unmanned aerial vehicles. The drones involved run on a very straightforward system. Each plane is sent a destination from the ground station and will then orient itself to travel to said point. The focus here surrounds what happens should there be multiple drones flying in a limited airspace, all at the same altitude and all with different points of destination. What is the best method to keep these airplanes from colliding with one another?

In recent times, this subject has become one of increasing interest to many. The uses for these unmanned aircraft are numerous, military to commercial being two areas of specific example. Oftentimes these tasks can include surveillance or delivery, missions that would be much more efficiently accomplished were multiple aircraft assigned to participate.
Because of this awareness to the benefit of such a situation, there has been a lot of research into the problem of collision avoidance. With multiple unmanned aerial vehicles (UAVs) working in the same space on the same task, collisions or near misses become a real issue. As a result, many solutions have been formulated to both detect and avoid near misses among UAVs.

These solutions are generally termed as ‘algorithms’ because they are a means to accomplishing an end by following a formulated process. Each of these approaches has a different way of approaching the problem and therefore its own benefits and downfalls.

Within this paper, a brief review of a few of these approaches can be found. By using the research behind these previously defined algorithms, a new method was formulated that attempted to achieve both effective and efficient collision avoidance. This new algorithm takes a hybrid approach on the problem, combining three different algorithms in an attempt to gain the positive traits of each and to overcome the areas in which each fell short. First, each algorithm that was utilized will be discussed individually. The three include the Dubins Path route planner, the optimized Rapidly Exploring Random Trees avoidance route planner, and the Inverse Point Navigation collision avoidance algorithm. Then the implementation of them in a hybrid method will be detailed as well as the results gathered that rate this method’s performance in an airspace with multiple agents flying in a simulated environment.

II. PROBLEM DESCRIPTION

A. define problem

As stated earlier, the basis of our problem is to develop an algorithm that will be used by UAVs to avoid collisions while hitting pre-determined waypoints. In our simulations, we define a near miss as a situation where two or more UAVs are within one second of flight time to each other. A near miss is defined as the same thing as a collision, but instead of one second of flight time, we use two seconds. Hitting a waypoint is defined as a plane being within one second of flight time to the target waypoint.

We use a two dimensional airspace for our problem. Constraining the problem to two dimensions makes it much easier to implement our algorithms, and reduces the computational complexity to some degree. At the same time, the algorithm we propose would be able to work in a three dimensional area with a few minor tweaks. This airspace can take any size, but for our simulations we use a 500m x 500m field, as well as a 1000m x 1000m field; this airspace constrains the waypoint’s locations, but not the aircraft.

In our simulations, we will run several cases with different numbers of aircraft. In particular, we will run simulations with 4, 8, 16, and 32 UAVs. This gives us a range of situations, from a very meager 4 UAV on 1000m x 1000m airfield, to a high stress simulation of 32 UAVs on a 500m x 500m airfield.
Near misses are handled in a special way for our simulation. Because we do not want the parameters for a simulation to change over the course of a simulation, planes do not crash after a near miss. Instead, the near miss is logged and the planes are allowed to continue to fly. If we were to delete planes, the number of UAVs flying would decrease over the course of a simulation, which could greatly affect results in the higher end stress tests.

B. Constraints

Because we are simulating a physical situation, some physical constraints must be in place that our algorithm should take into account. The first and most apparent constraint is the UAV’s inability to make perfect turns. Our UAVs are constrained to turns of 22.5 degrees per second, which means they can fly a full circle in 16 seconds. Secondly, we constrain the UAVs to have a constant airspeed of 25 miles per hour.

Our simulated planes also only send telemetry updates every second. This telemetry update contains the UAV’s ID, current latitude, current longitude, current altitude, target latitude, target longitude, and target altitude. The algorithm that we write can only use data from these telemetry updates to decide how to navigate the planes. This translates to real world situations, where we do not have a continuous flow of data, but rather data packets that are sent from the UAV to some controller. Also this form of communications between planes and controller force us to implement a real time algorithm. The algorithm we develop must be low on computational complexity in order to be effective.

C. Metrics

When testing our algorithm there are certain metrics that can be used to determine how well the algorithm performs. In a general sense, we want an algorithm that is both effective and efficient. Effective meaning our algorithm avoids near misses and keeps conflicts to a minimum. On the other hand an efficient algorithm will keep the UAVs on a near optimal path and hit as many waypoints as possible in a given time frame. Our work emphasized effectiveness over efficiency, but both classes of metrics were considered.

In specific, our effectiveness metrics are number of near misses, number of conflicts, and life expectancy. Number of collisions and conflicts are fairly straightforward, we just add up the total number of collisions and conflicts for a given simulation. Life expectancy is the number of near misses divided by the number of UAVs and divided by time:

\[
\text{Life expectancy} = \frac{\text{near misses}}{N \times t}
\]

Efficiency is mainly measured by number of waypoints reached. Because we are running our simulations for a set amount of time, we can see how efficient our algorithm is by measuring the total amount of waypoints reached for a given simulation. Alternatively, if we were allowing the simulation to run until all waypoints were reached, it could be possible to use a ratio between
the actual distance flown by our UAVs to the optimal distance (UAVs flying in straight lines to each waypoint).

All of these metrics will be averaged over a number of simulations for different course sizes and UAV numbers. These metrics will then be compared to a base case with no collision avoidance implemented in our results section.

III. LITERATURE REVIEW

A. Decentralized collision avoidance

A method of collision avoidance that is more popular in larger, more powerful unmanned aircraft systems is decentralized collision avoidance. This family of algorithms takes a more ‘divide and conquer’ approach when faced with the issue of drone navigation. A centralized system has a single control unit that handles all of the navigational commands. The units (planes, in this instance), simply receive these commands and go to the specified location.

In a decentralized system, the vehicles are involved in the calculations. Each either uses radars or local broadcast communication to detect any other nearby vehicles. They then compensate for whatever situation arises. Impending collisions are dealt with locally rather than on a global scale [1].

1) Generalized Roundabout Policy: This local control makes it much more feasible to scale the problem to a larger system. [2] discusses this in detail. It gives a specific instance of an algorithm that was developed using decentralized control, referred to as Generalized Roundabout Policy. It is a reactive algorithm that gives each individual UAV a disc of reserved space surrounding it. This disc is drawn with respect to the plane’s maximum turning angle. It is rendered so that when the UAV enters its maximum turn, the center of the disc does not change position. This results in the disc not only being able to change direction of motion as the plane does, but it is also able to stop moving altogether should the situation arise.

The range of motion afforded by these reserved discs aids the unmanned aircraft greatly in avoiding near misses. When a UAV detects that it is entering the range of another UAV’s reserved disc, it can change direction to circle around in order to avoid entering the space. In the case of a multiple UAV near miss, sometimes one of the aircraft will be caught in between two or more other aircraft.

When this situation occurs, it is advantageous for the central aircraft to halt the motion of its disc and remain in place, circling. It can then wait until one or more of the aircraft have freed themselves from the situation, allowing it to move forward unimpeded.

2) Multiple Party Collision Avoidance: Another documented algorithm that utilizes the framework of dividing the problem is Multiple Party Collision Avoidance. According to [3], this algorithm does not divide the airspace. Instead, it divides the aircraft into groups, or ‘parties’. 
Fig. 1. Different conflict possibilities between UAVs using the Generalized Roundabout Policy algorithm

Each party contains the aircraft that are nearby and have a chance of being involved in a near miss in the near future.

In each party, a master plane is appointed. This can be done by any means, as which plane is chosen does not affect the outcome whatsoever. The master plane then begins computations to detect future near misses in the group. When it detects one, it begins the process of attempting to resolve the problem.

In order to circumvent the discovered near miss, the master plane sends a request to the planes involved in the near miss. It asks them to each generate a modified plan that would result in the near miss not occurring [3]. Each one generates avoidances and sends them back. The master plane then expands on these plans, making sure that they do not interfere with any other planes. If necessary, it also invites planes to join the group should the current members get close to a plane not yet in the group. Planes may also drop from the group or be invited to join other groups when the situation calls for it. A plane may be a member of two parties at the same time, however, it may only be an active member in one. Whichever one it is actively participating in depends on which group’s planes pose the most danger to it at the time [3]. This allows planes to more smoothly traverse the field and be aware of any UAVs in the area, even when they may traverse along the border of two groups. Although, if a plane is actively participating in one party and changes its flight path as a result, it must inform the other groups that it is passively a member of that the old flight plan it was following no longer applies.

Both the Generalized Roundabout Policy and Multi-Party Collision Avoidance are extremely effective in cases of high UAV density. Decentralized algorithms tend to serve this purpose in a much better regard than their centralized counterparts. This is a logical result of dividing the problem rather than taking care of the entire system linearly. However, the computational power necessary to decentralize a system is much greater than that necessary to have a single central
unit. Not all aircraft have the necessary hardware to perform the required tasks. Because of this, sometimes it is necessary to use a centralized system. This is why, for this research assignment, it was not felt that either of the above algorithms were a good choice for implementation on the software currently available to this project.

B. Artificial Potential Fields

Artificial Potential Fields, or APF, is a reactive algorithm that is inspired from the behavior exhibited from charged particles. The idea is to artificially give each UAV a negative charge, and the current waypoint a positive charge. With this setup, any UAV will be repelled from all other UAVs while being attracted to its goal. In order to accomplish this, a force vector calculated from the potential field is generated at each time step. The UAV can then apply this force to itself. If the plane cannot physically achieve such a force vector, then it will apply the
closest possible force to the calculated force.

The actual field generated by this algorithm differs from the classical electromagnetic model. In electromagnetism, the potential field is proportional to \( q/r \) where \( q \) is the charge of a particle, and \( r \) is the distance from said particle. In this application, the field is tweaked for greater performance. An angular dependance is introduced to the field generated by UAVs, this field is generally stronger in front of the UAV, while weaker behind the UAV. This change guides the UAVs to avoid head on situations which is where most near misses occur due to less time for UAVs to react. Another adjustment is made to the potential field generated by waypoints. The field from a waypoint is linear and stronger than the UAV’s individual field; this assures that UAVs will always feel an attraction to its waypoint, and the numerous other UAVs will not completely determine the path of the UAV. [4]

![Fig. 3. A visualization of the potential field generated by a UAV [4]](image)

APF is a popular algorithm for several reasons. Its reactive nature makes it perfect for real-time collision avoidance; calculating a force vector for each UAV is an extremely fast process. This simplicity allows the algorithm to ramp up the number of planes effectively. In high density airspace simulations, APF does extremely well [5].

There are a few issues with the APF algorithm. APF can suffer from local minimum in the potential field. That is, the UAV gets attracted to a point that is not actually the waypoint (global minimum) but some point that happens to have a local minimum. However, this problem is much less pronounced here than in static environments, as the dynamic nature of our simulation usually means the local minima will disappear after a few time steps. APF has some special cases that must be handled outside of the basic algorithm previously described. In particular, the algorithm developed by Ruchti, Senkbeil, Carroll, and Dickinson had two special issues, head on collisions and the right hand turn rule. In a direct head on collision, the repulsive force of
another UAV would only cancel part of the waypoints attraction without changing the direction of a UAV. To fix this, a 15-degree right turn was applied to break the deadlock [4]. In some cases where planes fly parallel to one another with waypoints on either side of the opposite plane, a deadlock occurs where the planes will fly parallel for much longer desired. In this case Ruchti, Senkbeil, Carroll, and Dickinson applied the "Right hand turn rule" where one UAV will be forced to fly behind the other parallel plane [4]. APF is also a greedy algorithm, taking only the current time step into consideration rather than planning ahead. This leads to a lack of optimization in some cases.

C. Fuzzy Logic

The Fuzzy Logic algorithm is another possible approach to the problem proposed. [6] states that fuzzy logic was founded based on the idea that humans think in terms of concepts rather than exact numbers. This idea forms the structure of the logic behind the algorithm. While [6] looks at the algorithm from a robot soccer point of view, the process can be applied simply to collision avoidance as well by looking at how the Fuzzy Logic allows the robots to navigate without running into obstacles.

Two inputs in this instance are given to the Fuzzy Logic system. They consist of the distance of the robot from the goal and the angle at which the goal is oriented with respect to the robot. These values are then fuzzified, or made more general than specific numbers. This fuzzification process can consist of assigning values such as Far or Near for the distance value and Negatively Small or Positively Large for the angle [6].

These values are then combined into a single value representing a velocity value for the vehicle’s motion. The method for combining them is the use of a matrix format [6]. Rows correspond to angles and columns to distances, or vice versa. The combination of the two in each instance can be found at the cross where the two values meet. Figure 4 demonstrates these matrices. Because the robot has two wheels with separate functionality that can be used to control its position/orientation, it requires a separate matrix for each wheel. A vehicle with less variance of motion (for example, a car with a fixed axle) would only require one matrix. After this value is created, however, it must be defuzzified in order for the robot to know what action to take. There are many different methods with which to accomplish this. The basic idea behind it, however, is to perform the reverse of our fuzzification step. Instead of transforming real numbers into a generalized set, we want to transform our generalized set back into real numbers. This is not, however, an inverse function, for the original values are not returned exactly as they were. For further clarification, [7] performed an analysis of many of the different techniques for defuzzification. It is this step that tells the robot where it needs to go in order to meet its goal.

In a situation where obstacles exist that need to be avoided, the robot will detect the position of the obstacle. It will then compute the distance between the obstacle and the robot’s own current line of travel towards the goal. If this distance is not long enough, the robot is in danger of
hitting the obstacle. Based on whether the obstacle is on the left or the right, the robot will modify its angle of approach to avoid contact with the obstacle [6].

Fuzzy logic is a very different system from many of the others because of its inexact nature. This is not, however, a bad thing. [6] received very good results when implementing the algorithm in a real-time environment to avoid static obstacles. [8] also performed experiments using mobile robots, however, their research dealt with a problem more like this paper describes. They created situations with a large number of robots navigating in a confined space. The robots were able to find their way to the goal point without colliding with one another along the way. This shows that Fuzzy Logic is indeed able to hold up under stress.

D. Mixed Integer Linear Programming

Mixed Integer Linear Programming, or MILP, is a preemptive algorithm that has gained popularity for collision avoidance. MILP is a constraint-based optimization technique that can be used in hundreds of applications outside of this problem. In the case of collision avoidance, constraints such as airspeed requirements, minimum turning radius, and collision rules are programmed via
linear equations. The system of linear equations is then solved via techniques that are included in several third party MILP optimizers. These solutions yield the optimal trajectory for each UAV in the system.

Solving systems of more and more UAVs becomes an increasingly complex task. As the number of UAVs and obstacles grow linearly, the solve time grows exponentially [9]. In our problem, a real-time solution is required. In order to allow for MILP to work in real-time, a technique known as receding horizon control (RHC) is deployed. RHC splits the larger problem into several smaller problems, which can be solved real-time. Trajectories are only partially considered, then any groups of UAVs that are far enough away from other UAVs can be split into separate problems; these “groups” are then solved in their own system. The cost of using RHC is trajectories that are not completely optimized. While not as greedy as other algorithms such as APF, MILP with RHC does not consider the entire problem, and therefore cannot give the optimal path for each UAV. Figure 5 shows the difference of using MILP with and without RHC.

Fig. 5. RHC vs Full Path Planning [9]

Even with receding horizon control, MILP fails to solve high stress scenarios in real time. With airfields of 500x500m and 32 planes, MILP showed little benefit over no collision avoidance [5].
However, Holt’s thesis shows the MILP algorithm outperforming others in low stress scenarios, as it is able to plan highly optimal paths while avoiding near misses.

E. A*

The A* (or A-star) algorithm is a preemptive, path generating collision avoidance method. [10] discussed this algorithm extensively in their research. According to their summary, A* beings by dividing the airspace into a grid of equally-sized squares, or ‘nodes’. After doing so, it uses a ‘branching’ method to find the most optimal of the possible paths for each UAV. It does this by rating each node that may be included in the flight path of the UAV, estimating the costs of the best path that is possible using that specific node. An idea called ‘bounding’ is also included in this process. This makes it so that the algorithm only branches out from the node with the lowest estimated cost, helping to ensure that the most cost efficient path will be found as a result.

To find this estimated cost, something known as a ‘heuristic’ is used. The heuristic consists of two parts for each node: the known cost that it took to reach that node and the estimated cost of reaching the goal from that node. The final estimated cost is the resulting sum of these two elements. The heuristic method used for estimating this cost is really the defining factor in the computational complexity and, therefore, the runtime of the program. [10] even say that the choice of the correct heuristic brings the problem down from an exponential computation time to a polynomial one. This is a huge difference and a very important factor in real-time application with moving UAVs. In order to reduce computation time, constraints such as maximum turning angle can be used as well to determine which nodes it is possible for the UAV to travel through. This can limit the nodes that need rated and allow the field of branching to be lessened by quite a bit.

Collision avoidance with this method is achieved by once again rating the nodes. However, this rating applies to how likely it is that another plane will be within that node at the same time that this plane would be traversing it. This danger rating is then factored into the heuristic function [10]. This ensures that, with the bounding above, only those nodes that present the lowest danger level are traveled through by the plane.

Even when heavily optimized, however, this algorithm had issues keeping up with the computations for a large number of planes. In the instances where less planes were involved, the A* successfully computed a near miss free, optimal path for them. In the stress cases, however, it failed to avoid losing some of the planes. [10] found that the problem may lie in computing where planes may be while turning in the airspace due to the square nature of the grid space. It also may be that the heuristic needs to be more efficient to be able to compute the solutions more quickly in order to avoid near misses.
IV. Our Approach

A. Introduction

The main task behind this research was to create a new algorithm that improved the field of collision avoidance. In researching, we found a few algorithms with very strong positive traits. Because of this, creating a hybrid combination of them seemed the most logical way to go. Initially the plan was to combine the Dubins Path, RRT* (Optimized Rapidly Exploring Random Trees), and LOS (Line of Sight) algorithms in a single collision avoidance method. By doing so we hoped to benefit from the positives of each and let them cancel out one another’s downfalls. Though this hybrid method did not quite work out the way we planned, it still could be implemented were more research done to improve upon it.

B. Dubins Path

The first component to our algorithm was Dubins Path. Dubins Path is a purely path finding algorithm, meaning it is not used as a means to collision avoidance, but instead for finding a UAVs path to a given waypoint. Dubins Path achieves this by finding the shortest path for a Dubins vehicle between two states (position and velocity) [11].

A Dubins vehicle is a non-holonomic vehicle, which is constrained to a minimum turning radius[11] This is exactly the case for our UAVs which have constant speed of 25 miles per hour and a minimum turning radius of 22.5 degrees per second. The Dubins Path for such a vehicle is a series of circular arcs of minimum turning radius, and straight lines. Example Dubins Paths are given below.

\[ R_\alpha S_d L_\gamma \quad R_\alpha L_\beta R_\gamma \]

Fig. 6. two example Dubins Paths
In order to achieve Dubins Path for our algorithm, we used Andrew Walkers C++ Dubins Path Library, which supplied us all the necessary data structures and functions to properly implement Dubins Path into our ROS simulation. Andrew Walkers Library can be found at github.com/AndrewWalker/Dubins-Curves [12]. In our implementation, we used the current UAV coordinate and bearing for an initial state, and the UAVs next waypoint, and the bearing towards the waypoint after the next as the desired final state.

This implementation ran into one major issue when actually performing. In high-density UAV fields, the requirement for a bearing on the desired final state would cause a large amount of unnecessary turning. The cause for this is any collision avoidance will displace the UAV from its current Dubins Path. A new Dubins Path must then be generated for the UAV, which again has some final bearing requirement. Due to this bearing requirement, the UAV will take a longer path than necessary to achieve a waypoint, causing for more of these displacements.

Our short-term solution to this problem was to turn off Dubins Path and have the UAV go straight for the waypoint without regard for a desired bearing. This is not optimal, as not using the Dubins Path can lead to waypoint looping in some cases. A long-term fix that was not implemented would be to modify the Dubins Path code to allow for a wide range of bearings, rather than focusing on one. By allowing the Plane to achieve several different bearings when passing through a waypoint, it should easily be able to pass through without the unnecessary turning that is currently happening.

C. LOS maximization

LOS (Line Of Sight) maximization is a geometrical, reactive algorithm that is purely used for collision avoidance. The simplicity of this algorithm makes it easy to implement and computationally efficient.

This algorithm considers two UAVs on a near miss trajectory with one another. The LOS is then drawn as a vector from one plane to another. Now to avoid the near miss, a UAV needs to turn in a way that maximizes the magnitude of the LOS vector. This is done by considering the next position of our UAV on the next time-step, then considering the UAV in maximum turn left or right, we look at what the LOS would be in either scenario and choose the scenario where the LOS is maximized.

A question that needs to be raised is for a given UAV, which other UAV should it avoid? This question is central to this algorithm, as LOS maximization only considers one plane at a time. Currently our algorithm uses a combination of the LOSs current magnitude along with its derivative. If a LOS is less than 100m and has a negative derivative, the UAV will consider that plane. If two or more other UAVs meet those requirements, the LOS with the smallest Abs(Distance/Derivative) is chosen.

Future work for this segment of our algorithm is detailed in Manathara and Ghoses inverse PN algorithm [13]. In specific, we would like to explore zero effort miss to help determine
which UAV to avoid, and explore using LOS rate rather than the LOS magnitude to avoid near misses.

The major issue with LOS maximization is its greedy nature. LOS maximization only considers a single plane during one timestep. By choosing the best option for that plane on a time-step without considering the future or other planes, we run the risk of choosing a non-optimal solution. We hope with our approach the RRTstar can alleviate this shortcoming.

**D. RRTstar**

1) *RRT and RRT*\(^*\): The RRT algorithm runs in a very simple manner. The tree starts with an initial position, typically the starting location of the vehicle. The algorithm then generates a random position somewhere in the field. This position will be used to extend the tree. A search is run through the other positions already within the tree (in this first instance, the starting position would be the only one) to find the one nearest to this new point. Once found, a line segment is 'drawn' between the two. Only a portion of this segment will be taken into account, for it is desirable that motions be contained in order to get better paths [14]. The point along the segment that is the new vertex on the tree is then added as a child of the nearest point in the tree (found previously) and the nearest point is considered the 'parent' of the new point. This parent/child relationship allows the algorithm to trace back through the tree in the future [14].
The algorithm continues to generate new states and add them to the tree until it detects that a solution has been found. It then traces back through the tree to find the solution path. Figure 8 illustrates this expansion process. RRT* (pronounced RRT-star) is an optimized variation on the Rapidly Exploring Random Trees method. RRT runs once, finds the solution, and returns it. RRT*, however, continues to run until it either finds the optimal solution or the time limit provided is reached. This "anytime" approach allows for a much better path than the original. The beginnings of the algorithm are much the same as RRT. A random state is generated and added to the tree. It stores within this state not only the position and the parent, however, but also the cost of the state so far (generally the distance taken to reach it). Then the algorithm runs through other nearby states and determines whether there is another state nearby that can replace the current parent of the node that would cause a lower cost. If there is, it will replace the parent and store the new cost.[15]. The algorithm then goes through a 'rewire' process that does the inverse of the previous step. Instead of checking the nearby states for a lower cost parent, it checks whether the new state would be a lower cost parent to one of the nearby states. If so, it replaces the nearby state's parent and updates the cost [15]. These shifting and rewiring functions are repeated each time the algorithm runs and allow only the optimal, lowest cost path to surface.

2) Collision Avoidance Implementation: The RRT and RRT* algorithms can be very easily implemented as collision avoidance methods. As stated above, when a new vertex is introduced to the tree, a segment is extended between it and its parent. Before this new vertex is added, however, this segment can be checked. If an obstacle is in the way of the extension, then it can be dropped and a different path can be tried [16]. This allows only valid motions to be considered and near misses to be avoided.

For the sake of this experiment, we decided to use the Open Motion Planning Library (OMPL) [17]. It is a large open-source motion planning repository that is written in Object-Oriented C++. It contains many different path planners for vehicles, all pre-written and ready for implementa-
tion. This saved a lot of the time that would have been taken trying to write the RRT* algorithm from scratch.

The OMPL library does not have collision avoidance included in its planners, but it does have a function called the Motion Validity Checker. This function can be defined by the user and performs the action of checking whether or not a motion between two points is valid. In the case of RRT*, this determines whether or not a parent/child relationship can be established between two points. If the connection is valid, then the segment is added to the tree as a possible path section. If not, the connection is not established. This ensures that no plane travels where it should not.

This method works very well for both static and dynamic obstacles. However, with dynamic obstacles, a few more variables come into play. Not only does positioning of the obstacles matter, but also the time at which they will be in those positions. In our problem specifically, the constant speed of the planes allows this to be much more easily done. We know that the planes will never slow down or speed up, so if we can calculate the distance they must travel to reach a point, we can gain a fair approximation of the time at which they will reach it.

In the case of multiple UAVs, it made the most sense to approach avoidance in an ordered list. Each UAV would only worry about those that came before it in the order. This means that, initially, the first UAV would simply draw a path. Then the second UAV would draw a path that avoided the first UAV’s trajectory. In this way, each UAV, in theory, ends up flying safely to its waypoints.

To start with, a vector is created that contains the addresses of the other UAVs that the current UAV must avoid. Each other UAV is a class, and within that class is stored the UAV’s currently assigned RRT* path. This allows the current UAV to see where the other UAV will be in the foreseeable future. This list of UAVs is then passed into the Motion Validation function mentioned above.

The Validation function is fairly straightforward. By iterating back through the parents of the child currently being added, an approximate time that the UAV will be starting and finishing the path segment being validated can be found. This time is then compared with the paths of the other UAVs. Once segments of the other paths are found that will be traversed at about the same time, these paths can be compared to one another. If they cross, then the motion is not valid. If they do not, then it is.

When calculating these path segments in order to check for intersections, turns must be taken into account. Because of the way that the UAVs move, we assumed that the planes would hit a point and just after adjust their trajectory in order to orient themselves towards the next point. It is known that each UAV is able to turn 22.5 degrees each second, so the path that the UAV will follow through the turn can be estimated. Figure 9 illustrates the estimation of a segment drawn between two path points. A temporary point is drawn approximately every 11.125 meters along the path and checked against temporary points drawn along the paths of other UAVs. If
the points are too close (judged by whether or not a point is in another’s ‘buffer zone’) then a near miss could occur and the path is not valid. See Figure 10 for a demonstration of how these paths may be compared.

![Figure 9](image-url)

**Fig. 9.** An estimation of the path a UAV will take to travel between two path points.

3) **Issues, Corrections, and Future Work:** There were a few issues that arose in the implementation of the RRT* algorithm to generate paths. Some of these were pretty easily fixed, but others were not so successfully corrected in the time given. This leads to a good opening for future research in order to optimize and more successfully implement this collision avoidance approach. Currently the RRT* algorithm performs pretty badly in terms of preventing near misses. It is still our belief, however, that if it were properly debugged and honed, RRT* would make a very good primary choice for collision avoidance and be efficient at avoiding near misses. Future research could determine whether or not this hypothesis is correct.

4) **Waypoint Looping:** One problem that was a fairly simple fix was the issue of waypoint looping. Because the UAVs have a limited turning radius, sometimes they would approach a waypoint in such a way that it was impossible for them to reach it. Instead they would get stuck endlessly going in circles around the waypoints, turning at their maximum angle to reach the goal. This was a big problem when it came to efficiency. Some planes would not hit a single waypoint throughout the entire test because they would get caught up in this looping right away. The RRT* made this problem even more apparent by introducing a path of waypoints and thus increasing the chances that such a hangup would occur.
The first step in fixing this looping was to detect when it was occurring. This was done by calculating the UAV’s bearing, the bearing from the UAV to the target, and the UAV’s maximum turning radius. The UAV’s bearing is found by taking its previous position, \( P_i = (x_i, y_i) \), and its current position, \( P_c = (x_c, y_c) \), and computing the angle of travel with a simple trigonometric formula: \( B_{uav} = \arctan\left(\frac{y_c - y_i}{x_c - x_i}\right) \). The bearing from the UAV to the target can be found using the same method, but instead using the UAV’s current position as the first position and the target’s position as the second. The maximum turning angle is a bit more complicated a trigonometric formula that uses a few more parameters. It relies on the speed of the UAV as well as the UAV’s maximum turning angle.

\[
R_{\text{maxturn}} = \text{speed} \times \frac{\sin\left(\frac{\pi - \text{Angle}_{\text{maxturn}}}{2}\right)}{\sin(\text{Angle}_{\text{maxturn}})}
\]

If the UAV is looping, then its distance from the target will be about the same as the maximum turning radius. Also, because the UAV would be circling the waypoint, it’s bearing would be approximately perpendicular to the target bearing. Figure 11 helps to illustrate this situation. The next question is, of course, how do we fix this? We know that it is happening, but what next? We decided to look at two situations differently in order to optimize our time for this algorithm. The first is that the UAV is looping around a point along the RRT* path. It is not required, however, to hit this point. If this is the case, instead of forcing the UAV to correct it’s path and hit the waypoint, we instead redraw the RRT* path to the goal. This allows the UAV to continue flying towards the goal instead of stalling and trying to hit the unnecessary
path point. Sometimes, however, the UAV will get stuck on an actual goal waypoint. If this occurs, drawing the RRT* will do us no good because of the proximity to the goal. To rectify this special case, we decided to make the UAV follow a straight line for a bit in order to adjust the turning trajectory. This was done by extending the line the UAV was currently following and introducing a new path point outside of the loop. The UAV would then break out of the loop and be more suited to circle around and hit the waypoint.

5) Timing: Calculating the time that each UAV was going to hit each pathpoint proved to be a much larger challenge than was at first expected. It should have been a pretty straightforward calculation, but uncertainties led to discretions in time. Logically we deduced that the biggest problem was the above looping issue. Even though these loops were corrected fairly quickly, they still made the paths unexpectedly longer. This would throw the hypothesized time off each time it happened and would put the UAVs that had been stuck behind schedule. The problem increased when we introduced near miss checking into the correction of the second, goal-waypoint looping instance. We tried to make it so that if following a straightline path would result in a near miss, the UAV would wait and see if it could correct in a non-conflicting manner shortly in the future.

Fig. 11. UAV looping infinitely around a waypoint.
However, this caused it to get further and further behind its scheduled timeline.

While time did not allow for a fix to be found in the current algorithm that was created in this research, future work could improve this. One way that this problem could be rectified would be to force the RRT* algorithm to take into account limitations such as the turning angle when drawing the tree. This would help prevent such looping from occurring. Also, a better guess of the UAV’s pathlength would perhaps allow the timing to be more accurate. Because the OMPL library is an extensive collection of classes, some methods were difficult to specialize to our algorithm’s implementation. Calculating the distance traveled so far along the tree at each new vertex was one of those difficult tasks. It may be improved if written into the core of the program rather than as an afterthought.

6) Parameter Optimization: Another area that more work could definitely be done in the future is in the honing of the different parameters RRT* deals with. These variables are extremely important to the working of the algorithm and could adjust how well it preforms in different areas. They include the size of the buffer zone, the time allowance, the length of the path segments, and the goal bias of the RRT* algorithm. The buffer zone size determines how close two paths must be in order to consider one of them as conflicting with the other. Currently this is set at 25 meters because this is approximately the distance required for a conflict according to the problem’s metrics. A larger buffer zone would create a more cautious system but also could cause the RRT* algorithm to fail in finding a solution more often, especially in high density situations. The time allowance is the amount of leeway you give the other paths in reference to time. For example, say the current path segment that is being checked occurs five seconds in the future. Were the time allowance set to three seconds, then we would check against all other UAV positions from two seconds to eight seconds for near misses. This helps to allow for discretions in the calculation of time. A greater time allowance, like the larger buffer zone, would cause a more cautious system. However, it too could cause efficiency issues and force RRT* to fail more often in finding a valid path. The length of the path segment is the distance between each path point drawn in the RRT*. Longer path segments give better results in efficiency but can cause more near misses with their greater uncertainty. Currently the path segment length we are using is 100 meters. Finally, the goal bias is the frequency that the RRT* algorithm samples the goal as a new state. This can help to direct the path more efficiently towards the goal. However, if this number is too high, it might be difficult for the planner to come to a solution in the prescribed amount of time. Most of these parameters were optimized by hand in a sort of ‘plug and test’ manner. Perhaps with more calculations and regularity they could be fine-tuned to their optimal values instead.

V. Results

To obtain these results, we generated 3 random courses for each case, and ran a ten minute simulation on each case. The final results below are the averages from each simulation.
A. With Collision Avoidance

<table>
<thead>
<tr>
<th>500x500 meters</th>
<th>4 UAV</th>
<th>8 UAV</th>
<th>16 UAV</th>
<th>32 UAV</th>
</tr>
</thead>
<tbody>
<tr>
<td>near misses</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.333</td>
</tr>
<tr>
<td>conflicts</td>
<td>1</td>
<td>0</td>
<td>10.33</td>
<td>47</td>
</tr>
<tr>
<td>waypoints</td>
<td>57.66</td>
<td>63</td>
<td>83</td>
<td>69.33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1000x1000 meters</th>
<th>4 UAV</th>
<th>8 UAV</th>
<th>16 UAV</th>
<th>32 UAV</th>
</tr>
</thead>
<tbody>
<tr>
<td>near misses</td>
<td>3.33</td>
<td>29</td>
<td>136.66</td>
<td>596.66</td>
</tr>
<tr>
<td>conflicts</td>
<td>42</td>
<td>123.33</td>
<td>528.66</td>
<td>2448.66</td>
</tr>
<tr>
<td>waypoints</td>
<td>62.33</td>
<td>180</td>
<td>311.66</td>
<td>666.66</td>
</tr>
</tbody>
</table>

B. Without Collision Avoidance

<table>
<thead>
<tr>
<th>500x500 meters</th>
<th>4 UAV</th>
<th>8 UAV</th>
<th>16 UAV</th>
<th>32 UAV</th>
</tr>
</thead>
<tbody>
<tr>
<td>near misses</td>
<td>3.33</td>
<td>29</td>
<td>136.66</td>
<td>596.66</td>
</tr>
<tr>
<td>conflicts</td>
<td>42</td>
<td>123.33</td>
<td>528.66</td>
<td>2448.66</td>
</tr>
<tr>
<td>waypoints</td>
<td>62.33</td>
<td>180</td>
<td>311.66</td>
<td>666.66</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1000x1000 meters</th>
<th>4 UAV</th>
<th>8 UAV</th>
<th>16 UAV</th>
<th>32 UAV</th>
</tr>
</thead>
<tbody>
<tr>
<td>near misses</td>
<td>1</td>
<td>12.33</td>
<td>44.66</td>
<td>159.33</td>
</tr>
<tr>
<td>conflicts</td>
<td>18.66</td>
<td>57.22</td>
<td>179.66</td>
<td>661.33</td>
</tr>
<tr>
<td>waypoints</td>
<td>31</td>
<td>78</td>
<td>185.66</td>
<td>378.66</td>
</tr>
</tbody>
</table>

Overall, we are happy with these results. As seen in the figures above, the algorithm drastically reduces the number of near misses, especially in dense airspaces. It should be noted that we are seeing strange results for the 1000x1000m, 16 and 32 UAV case. The airspace is less dense, yet we are seeing more near misses in these cases than the 500m counterpart. Part of this is due to random error. Due to time constraints, we were only able to run three courses for each case, this number may be too few to smooth out the random difficulty in a course. Moreover, we observed for cases that did have near misses, oftentimes two UAVs would move into one second of flight time between each other, and continue to stay in near-miss range for several time steps. This phenomenon inflates our near miss numbers.

Another Number to focus on is the waypoints reached. For low density

VI. CONCLUSION AND FUTURE WORK

In this paper we have discussed methods of collision avoidance in multiple UAV systems as well as introduced an algorithm of our own design. While we are happy with our results, there is a
lot of work that could be done to improve on our idea. The major component of this future work would be improving RRT* so it can take a more active role in the collision avoidance. Because the main interest of this research project was collision avoidance, LOS performed well enough to satisfy this requirement. As stated before, however, LOS maximization is a greedy algorithm which can result in suboptimal solutions. By improving RRT*, we reduce these suboptimal solutions by picking non-greedy solutions. This could help to raise the waypoints reached and the efficiency of our UAVs. This is the area that LOS has problems with because of its reactive nature. In situations that require efficiency over the effectiveness of collision avoidance, having RRT* in the mix would (theoretically) provide a much better solution.

REFERENCES


