COLLISION AVOIDANCE IN UAVS USING DYNAMIC SPARSE A* TECHNICAL REPORT #CSSE11 - 02

THOMAS CRESCENZI, ANDREW KAIZER, TYLER YOUNG, JAMES HOLT, AND SAAD BIAZ

Abstract. In order for unmanned aerial vehicles (UAVs) to be widely adopted in civilian airspace, they must be capable of safe, autonomous flight. The problem of collision avoidance in UAVs is discussed in its theoretical foundations, and an overview is given of the methods of collision avoidance and path planning most widely represented in the literature. The authors' own approach—an adaptation of A* search for use with dynamic obstacles, such as other UAVs—is discussed in depth. This algorithm, termed Dynamic Sparse A*, is discussed in terms of both the underlying theory and the actual implementation. The benefits, drawbacks, and potential applications of the algorithm are also considered.

Contents

Date: August 22, 2011.

1. Introduction

In recent years, interest in unmanned aerial vehicles (UAVs) has grown steadily in the private sector. Applications as varied as fertilizing crops, surveying land, and patrolling borders could employ UAVs to increase efficiency, lower costs, and keep humans from potentially dangerous situations [**?**].

As civil and commercial interest in UAVs grows, however, a number of obstacles remain which prevent their widespread adoption in nonmilitary situations (i.e., in civilian airspace). Primary among these obstacles, at least in the United States, is the requirement set forth by the Federal Aviation Administration (FAA) that for UAVs to be integrated into the national airspace, they must be at least as competent (capable of operating safely) as an equivalent human pilot *without* cooperative communication (such as commands from a human controller or information from neighboring aircraft) [**?**]. Many UAVs today are designed to operate primarily with the remote guidance of a human pilot. To be widely certified for flight in the national airspace, however, regulatory agencies require the aircraft to be capable of safe autonomous operations in the event of a contingency. In the United States, there are approximately 0.5 midair collisions per million flight hours [**?**]; to be cleared for flight in civil airspace, then, a UAV would have to demonstrate an ability to fly at least this safely autonomously.

Safe air operations are often discussed in terms of an operator's ability to "sense and avoid" potential conflicts. For an unmanned aircraft, this requires that the onboard "pilot" possess some means of detecting nearby aircraft, as well as a means of altering its course in advance to avoid endangering itself or other aircraft.

A UAV needs not only to be able to alter its course, but it needs to do so in a way which is intelligible to human pilots observing it. By following standard flight procedures (such as guidelines for determining right of way), a UAV can take action to resolve conflicts while simultaneously communicating its intent to pilots around it (i.e., by the way in which it alters its trajectory). Additionally, the planned path must be flyable; it cannot call for abrupt 180° turns, or maneuvers which put too much stress on the airframe. The final primary consideration for a UAV's collision avoidance system, of course, is the aircraft's efficiency in negotiating the airspace, whether in terms of time required, fuel used, or danger avoided.

The most desirable path for an autonomous aircraft, then, is a flyable path of lowest possible cost which always maintains a safe distance from other aircraft. Having found this optimal path, the planning computer must also be able to re-evaluate its plans often in order to account for new craft in the airspace, changes in environmental conditions, and so on.

Collision avoidance, then, may be summarily divided into two parts: conflict detection (maintaining awareness of other aircraft and potential obstacles) and conflict resolution (maneuvering to avoid hazards in light of the system's knowledge from the conflict detection system). These two objectives are, respectively, to "sense" and "avoid" in as efficient a manner as possible.

For our purposes, we are concerned only with the path planning and conflict resolution portion of the sense and avoid problem; the algorithm presented here takes for granted that other aircraft are correctly perceived, and that their goals are predicted with some degree of accuracy. We achieve this by means of GPS location transmitters onboard each aircraft, but in other implementations, one might correctly "sense" nearby aircraft visually (using some form of image analysis), by radar, or through a number of other means.

1.1. **Computation of a Best Path.** As discussed, there is a great deal of interest in creating UAVs able to autonomously plan paths which are optimized for both efficiency and safety to take the aircraft from an arbitrary location to its target. Creating such a system is clearly a hard problem, as there are a large number of constraints to satisfy, but it is in fact an even greater challenge than it appears at first glance: finding a best path is NP-complete [**?**, p. 869], meaning that the computation is among the most difficult problems which can be solved algorithmically. More precisely, NP-complete problems are computational problems for which no polynomial-time solution (a solution which can be computed in a reasonable amount of time on a large set of input) is known. These problems are widely believed to be intractable [**?**, p. 9]—that is, they are theoretically solvable, but as far as we know, finding a solution requires too much time to be useful in most cases.

In light of this, any method for computing a safe path for a UAV must compromise either optimality or time; one must settle either for a path which is good without being the best, or one must begin the computation long before its solution is needed. Because conditions in the air may change rapidly (due to changes in weather, the arrival of unexpected aircraft, systems malfunctions, and more), computing an optimal path ahead of time is often impossible.

1.2. **Formulation of the Problem.** Bearing in mind the requirements for autonomy, safety, efficiency of path, and speed of computation, we may formulate the problem of collision avoidance in unmanned aerial vehicles thus:

> *In a short amount of time, find a flyable path of minimum or near-minimum cost which maintains appropriate distance from all other aircraft.*

This formulation is broad enough to be applicable to a very wide range of problems, as differences in implementations amount to changing how one calculates the cost of a path. Using the same formulation of the problem, one might choose to focus either on terrain costs (risks associated with flying over mountains, traveling too far without fuel, etc.), dangers associated with military operations in hostile airspace, the economic cost of fuel, and so on. Indeed, the actual parameters used in calculating cost are of no importance to the search algorithms themselves.

Worthwhile to note is the fact that methods of path planning for collision avoidance may focus either on static or dynamic obstacles. Although it is trivial to add static obstacles to a system designed for dynamic ones, it is not clear that systems described in the literature designed for static obstacles might so easily be adapted to dynamic ones. This is important, as the choice one makes in this area will determine just how short a "short amount of time" really is. When planning for static obstacles only, a 30-second calculation of an optimal path may be sufficiently fast, whereas when dealing with dynamic obstacles, anything over 1 or 2 seconds may be unacceptable.

For our purposes, we are concerned exclusively with dynamic obstacles, in the form of other, independent aircraft. Thus, in order to account for other aircraft changing their goals or altering their paths in ways which may be unexpected, our algorithm aims to check each plane's path for optimality once per second.

2. Literature Review

A large body of work exists addressing how best to confront the apparently inescapable trade-off between computing time and optimality in the path planning problem. The approaches which are best represented in the literature may broadly be divided into the following categories:

- A* search, a method of computing the lowest-cost path from one node to another in a mathematical graph (i.e., a set of nodes connected by edges)
- Artificial potential fields (APF), which simulate potential fields (*à la* magnetic fields) wherein the agent and its goal are of opposite "polarity" (causing the agent to be "attracted" to its goal), while the agent and its obstacles are of the same polarity (causing the agent to be "repelled" from them)
- Geometric approaches, which rely on vectors to calculate a point of closest approach between one point mass (the simplified model of an aircraft) and another, and to suggest a modification to those vectors which avoids conflict or collision
- Mixed-Integer Linear Programming (MILP), a method of solving problems which involve "both discrete decisions and continuous variables" [**?**], used in particular for creating a path optimized for a number of different constraints

2.1. **A* Search Methods.** A* (spoken as "A-star") search is a method of finding best paths in a mathematical graph or tree. For the purpose of path planning in aerial vehicles, the airspace must be divided into a grid, where each square in the grid represents some area of the airspace. These squares are, in actuality, nodes in the graph used by A* to represent the airspace. The algorithm, then, begins at the start node (corresponding to the aircraft's initial position) and considers the possible nodes to which the aircraft could travel. It rates each of these possible nodes, estimating the cost of the best possible final path which incorporates that node. This rating of the nodes which are open to travel from some other node is referred to as "branching." Each node rated during the branching process is added to a heap (a partially ordered data structure whose "top" or "front" element is the element of lowest cost), which stores the list of nodes open to consideration for inclusion in the optimal path.

Having finished the branching process from the start node, and beginning with the top node in the heap (i.e., the node estimated to yield the lowest-cost path to the goal), the algorithm repeats the process: it considers all nodes open to travel from the new node, selects the one it estimates to be best, and continues the search until it reaches the destination. If, at any point, it estimates that the path it is following will be of higher cost than a path leading from some previous node (i.e., a node in the open heap), it will return to that previous node and search from there.

As we have said, the algorithm's consideration of the nodes to which it could travel from its current position is referred to as "branching." The algorithm's estimation of the best possible path from a current position is referred to as "bounding" the future search. A* search, then, is a branch-and-bound method of coping with NP-completeness. It is also referred to as a best-first search, as it follows the "most promising" paths first [**?**, p. 6]—that is, it always branches from the node of (currently) lowest estimated cost.

The A* algorithm's insight lies in the way in which it branches. Since the search problem is NP-complete, a simple in-order search of each possible path would take a prohibitively long amount of time. Instead, A* prioritizes open nodes based on its estimate of the best possible path from the node to the goal. This estimate is known as a heuristic. The estimated lowest cost of a path from the initial node to the goal which passes through a node n is denoted $f(n)$. The calculation of $f(n)$ is

$$f(n) = g(n) + h(n),$$

where $g(n)$ is the known (previously calculated) best cost to go from the start node to node n, and $h(n)$ is the estimated cost of the best path from n to the goal node [**?**, p. 97].

The path calculated by the A* algorithm is guaranteed to be a global optimum whenever the heuristic function fulfills two requirements:

(1) $h(n)$ must never *overestimate* the cost from n to the goal, and
(2) $f(n)$ must be "consistent" in its method of estimation, meaning that a successor n' of a node n should never be estimated to have a higher cost than the step cost from n to n' plus the estimate $h(n')$ [**?**].

In this regard, A* has a clear advantage over other path planning algorithms. No algorithm can *possibly* generate better solutions, given the same search space and resolution; at best, other algorithms can generate equally good solutions. The disadvantage, of course, is that this optimality comes at the cost of computing time: in the worst case (i.e., using a poor or very general heuristic function), the computing time required grows *exponentially* with the size of the input, so large search spaces are completely impractical to work with.

Thankfully, the use of a good heuristic function (one which produces estimates very close to the actual cost without exceeding it) can reduce the time complexity of the problem from exponential to polynomial [**?**, p. 7]. This means that with the right heuristic, one could conceivably perform a search of a reasonably large problem space in seconds using no more than a small, lightweight computer onboard a UAV.

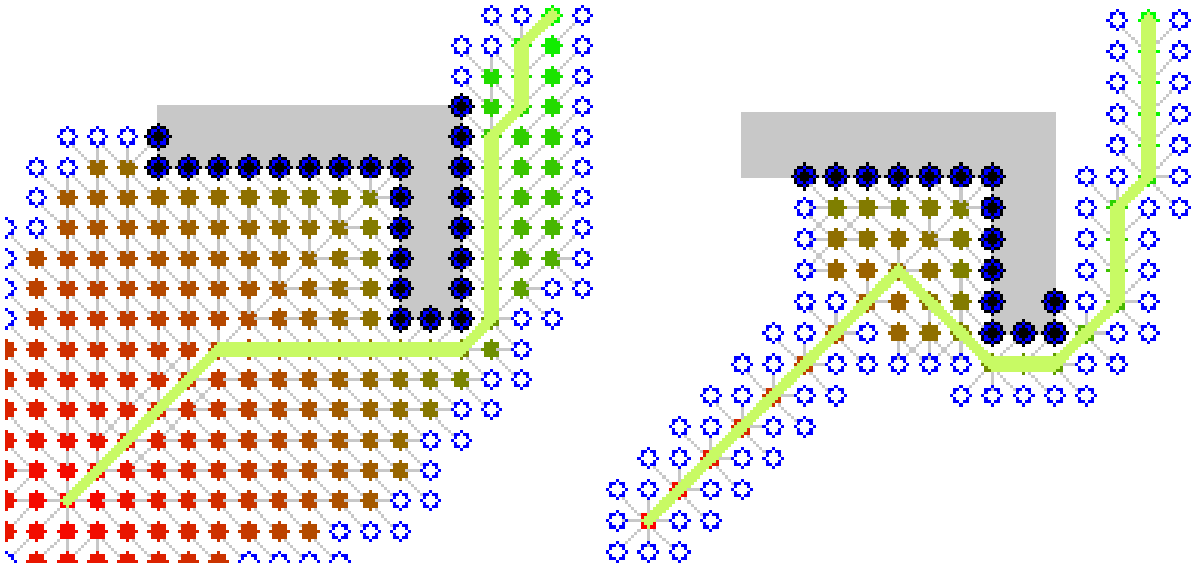

FIGURE 1. *"Classic" A* versus weighted A^*_ϵ search*
Image credit: Subhrajit Bhattacharya, released under the CC BY 3.0 license

2.1.1. *Reduced Complexity Varieties of A*.* In order to further reduce the time complexity of the search, a number of simplifications may be made to the algorithm. For instance, the A^*_ϵ (spoken "A-epsilon-star") algorithm searches several nodes in a row (all nodes within a fixed, positive ϵ value of the lowest-cost node) before each branching. This simplification is made on the (justified) assumption that the selection of the lowest-cost node is among the most time-consuming parts of the computation [**?**, p. 18].

This method trades the guarantee of optimality for reduced search time. As illustrated in Figure 1, A^*_ϵ considers far fewer nodes, which may result in a sub-optimal path. However, if strict optimality is not necessary, considering fewer nodes in this way is a viable way of significantly reducing the time complexity of the search.

The "Sparse A* Search" (SAS) method, designed for military flight planning by Robert Szczerba, Peggy Galkowski, Ira Glickstein, and Noah Ternullo [**?**], stands out among simplified versions of the A* algorithm. SAS significantly reduces the complexity of the

search through a few clever assumptions, nearly all of which preserve the optimality of the path generated.

SAS improves the time complexity of the A* algorithm first by discarding from consideration large portions of the searchable area based on the limitations of the aircraft and pilot. For instance, in keeping pilot fatigue down, a minimum length is imposed on each "leg" of the path; paths with straight-line portions shorter than the minimum leg length are not considered. Likewise, in keeping the route flyable, a maximum turning angle is imposed on the search; paths which require turns beyond the capability of the craft or the comfort of the pilot are not considered. Figure 2 illustrates these constraints on a small, simplified search area.

Further constraints on the path (such as a maximum route distance, maximum distance from a refueling station, or fixed heading when approaching the destination) may be considered to further reduce the area explored by the search. Introducing any of the constraints on SAS listed above will not result in a sub-optimal path; any *ideally* better path which is eliminated from consideration by the above parameters would *in fact* be undesirable anyway, due either to safety concerns or the physical limitations of the aircraft.

In their discussion of SAS, Szczerba et al. provide another method of reducing both time and space complexity (i.e., computing time and memory usage) [**?**]. Their method places a hard limit on the amount of memory used by the algorithm. As previously discussed, during the branching phase, A*'s heuristic function estimates the total cost of a path through a node and places that node on a heap (this heap is later used to select the next node from which to search). In order to limit memory usage, Szczerba et al. "prune" the heap after it reaches a predefined depth. When a new node is to be added to an already full heap (a heap of maximum depth), pruning occurs by:

(1) randomly selecting an element at the bottom of the heap (i.e., an element most likely to lead to a sub-optimal solution),
(2) replacing it with the node to be added, and

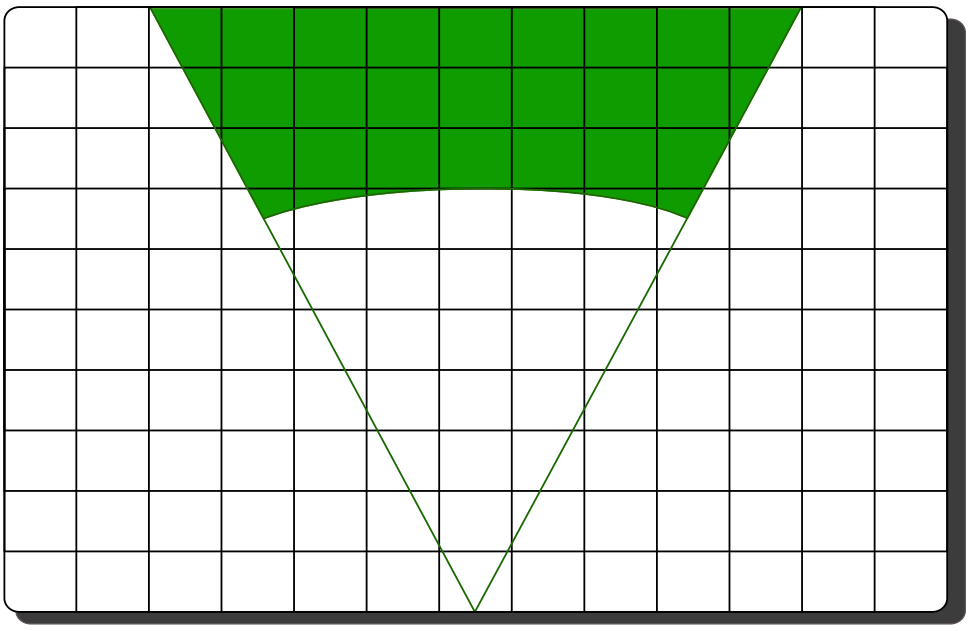

FIGURE 2. *Minimum leg length and maximum turning angle illustrated on a simple grid; only the shaded region is open to consideration.*

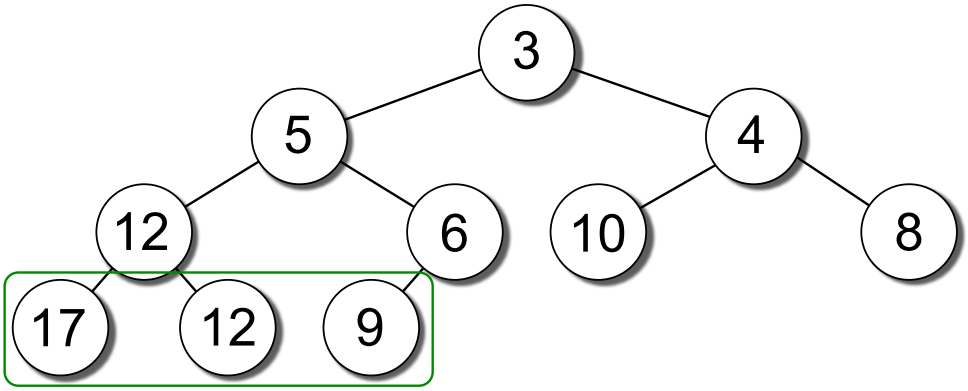

FIGURE 3. *A binary min-heap, whose lowest level will be pruned by SAS once full*
Image adapted from Wojciech Muła under the CC BY-SA 1.0 license

(3) restoring the heap structure (the ordering property), in an operation known as "up-heap" or "heapify-up," among other names.

For instance, in Figure 3, if the maximum queue depth were set to 4, the nodes highlighted in green would be open to pruning once the lowest level became full.

In addition to the obvious memory savings, pruning causes a smaller number of nodes to be searched overall, leading to a decrease in computing time.

Unlike the other optimizations used by Szczerba et al., limiting memory usage in this way may result in a loss of optimality; nodes which are pruned from consideration may in fact lead to excellent solutions. With a relatively large maximum size, however, this will only occur through a failure of the heuristic, in an instance where it judges a possible path to be significantly less promising than it actually is. Nevertheless, in view of this possibility, Szczerba et al. recommend the maximum heap size be set as high as possible in order to prune as few nodes as possible from consideration.

2.2. **Artificial Potential Field (APF) Methods.** The use of an artificial potential field (APF) in collision avoidance is adapted from particle physics and, in particular, the attractive and repulsive qualities of polarized charges. APF methods model the obstacles that an aircraft needs to avoid (such as other airplanes) as repelling point charges while modeling an aircraft's goal as an attractive point charge. When a plane comes into contact with the repelling APF (RAPF) of another plane, a force vector is calculated and the planes are given new waypoints which push them away from one another. Figure 4 illustrates the vectors calculated in the implementation of such fields.

The force function responsible for creating these fields must meet the following requirements:

(1) It must be continuous and differentiable.
(2) Its strength must increase inversely with the distance to obstacles.
(3) Its strength must decrease directly with the distance to goals [**?**].

Creating a force function suitable for a given situation is rather difficult. However, once the general mathematical function is developed, all calculation can be done by directly consulting the function's output. Thus, the time complexity of APF calculations is minimal; indeed, this is the significant advantage to using APFs over A*-based planning methods. While the functions themselves are complex, once they are programmed, one need only calculate the total force acting on a single plane in order to generate a force vector [**?**]. Once this vector is calculated, its value is input into a function that generates a new waypoint for the plane.

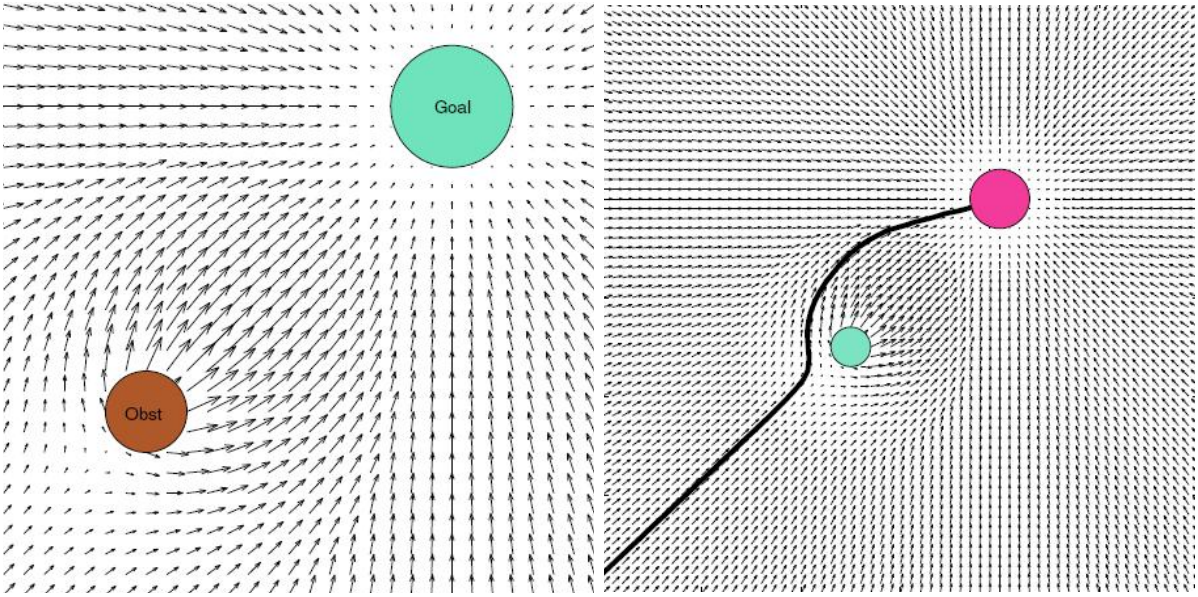

FIGURE 4. *Force vectors created by summing the potential fields, with the avoidance path taken by an aircraft around the obstacle*
Image credit: [?]

Artificial potential fields cannot, in a strict sense, be used for path *planning*; aircraft blindly follow the pushes and pulls that act on them, without regard for future positions or even optimality. It is thus more accurate to think of APFs as being used purely for collision avoidance, under the assumption that true path planning is not necessary. Indeed, for uncrowded airspaces, this is a reasonable assumption; if an aircraft only encounters a potential conflict, say, once per hour, long-term path planning would provide at best a small advantage in the efficiency of the path flown. However, in more crowded airspaces, artificial potential fields have significant drawbacks.

2.2.1. *Local Minima.* The first, and largest, of the problems associated with APFs is the existence of local minima. A local minimum is an area with a net force of zero [?]. This can happen when the aircraft is surrounded by obstacles, or when its goal is crossed by another aircraft (situations which are also problematic for A* search). However, local minima may be created in other ways as well. The most devastating of these conditions occurs when two aircraft are situated with respect to one another in a way that causes the force repelling the first plane from the second to be exactly the opposite of the force pulling the first plane to its goal [?]. In such a case, the first aircraft might "think" that it had already reached its goal, or worse, that it was safe to continue in a straight line, leading to a collision.

Numerous solutions to this problem have been proposed. The first is to cause the aircraft to default to an A*-determined path if it reaches a point in space with a net force of zero [?]. While using an A* path as a failsafe does overcome local minima, it also increases computation time significantly. Another solution to local minima is to use genetic algorithms to perfect the field calculations [?]. In the long run, this method reduces computation time. It does, however, incur the typical problems associated with genetic programming, including the many generations required to find an excellent function.

2.2.2. *Oscillatory Movements.* A second issue which must be addressed when using artificial potential fields is the problem of oscillatory movements. Such movements occur when the fields around an aircraft sum to forces moving in a wave or in small circles [?]. These

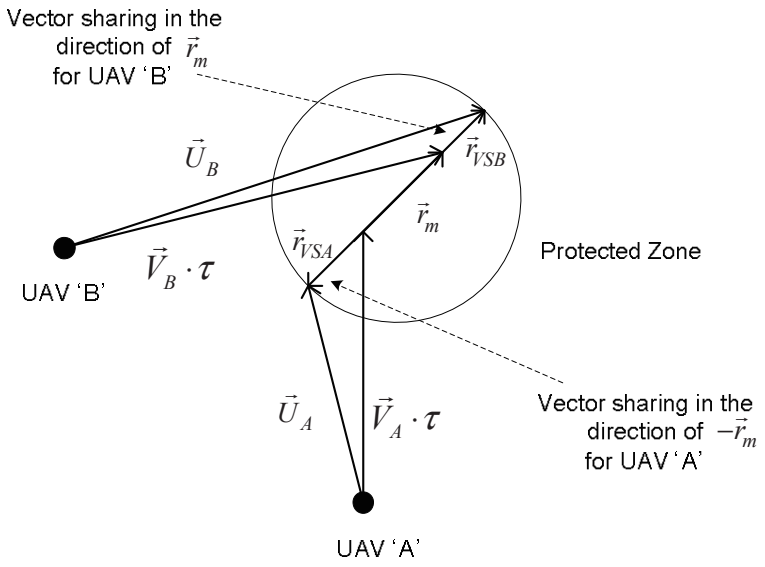

FIGURE 5. *Creation of an avoidance vector via the geometric approach*
Image credit: [**?**]

circular movements cause the aircraft to take paths that are clearly suboptimal, or even to backtrack. While such movements may not cause collisions, they do greatly increase the time required for a plane to traverse a given path. This problem can be addressed by requiring a minimum distance between consecutive waypoints [**?**]. When the waypoints are spaced out by a set distance, most oscillatory movements can be eliminated. In all cases, however, the paths are not guaranteed to be optimal.

2.2.3. *Flyable Paths.* The fixes to the problems associated with potential fields discussed above work very well when working mainly with ground-based robots. When working with aircraft, however, it is quite difficult to ensure that the paths generated are flyable. Unflyable paths are those with too much oscillation or with waypoints outside the craft's maximum turning radius.

Under traditional methods of APF creation, it is difficult to take into consideration the aircraft's turning radius. Using a genetic programming approach to the creation of the force function, however, the potential fields may be tuned in such a way as to work within the aircraft's limitations after many generations [**?**].

Thus, while APFs are promising tools for collision avoidance in aircraft, they have many limitations that typically require a secondary programming method to be included on top of their simple force vector calculations.

2.3. **Geometric Approaches.** Geometric approaches to collision avoidance are built on 3-dimensional vectors which represent an aircraft as a point mass with a velocity (i.e., speed and heading) and location. An airplane's vector is calculated out to a specified time, and if it is predicted to come too close to another craft's vector, the aircraft are considered to be in conflict [**?**]. When a conflict is discovered, each airplane works to avoid the predicted conflict zone by calculating a new path known as their avoidance vector. The aircraft then adjust their speed, heading, and altitude as necessary to follow this avoidance vector.

Calculation of an avoidance vector is very fast; it requires even less computing time than the force vector calculation used in an artificial potential field. However, while this method works well with two aircraft given one waypoint each, it becomes more difficult

to calculate a good avoidance vector as the airspace grows more crowded [**?**]. In addition to the problem of scaling, this method also hinges on the effective use of altitude control, which may not be desirable in all situations.

2.4. **Mixed-Integer Linear Programming (MILP) Methods.** Mixed-integer linear programming is a method of path planning which relies on a description of the problem in terms of a desired optimization of discrete decisions and continuous or integer variables. The mixed-integer linear program which results from this description of the problem may be solved using any of a number of extant software packages, both open-source and proprietary.

MILP has primarily been used in linear path planning around static obstacles. To adapt this strategy to collision avoidance with dynamic obstacles (e.g., other UAVs), one must first model the aircraft's flight paths in a linear, static way [**?**]. Performance characteristics of the aircraft such as their turning radii and acceleration capabilities must be rendered as linear, though they are not, of course, in reality. These limits on the aircraft's performance may effectively be represented as force magnitude limits [**?**]. For instance, the instantaneous turning rate ω may be limited in the program by adding the constraint:

$$\omega \leq \frac{f}{mV}.$$

Adding constraints in this way into the formulation of the problem ensures that the (optimal) path generated is a flyable one.

Once the flight characteristics of the aircraft are modeled in the input equations, the system of equations is solved in a way that creates paths which prevent collision for all planes and produce optimal times to waypoints. These paths are calculated at once, from the aircraft's starting positions, and do not need to be calculated again except in the case of contingency.

The MILP approach benefits heavily from having only cooperative aircraft in the airspace. By giving each aircraft a path which is optimized for the "greater good," a

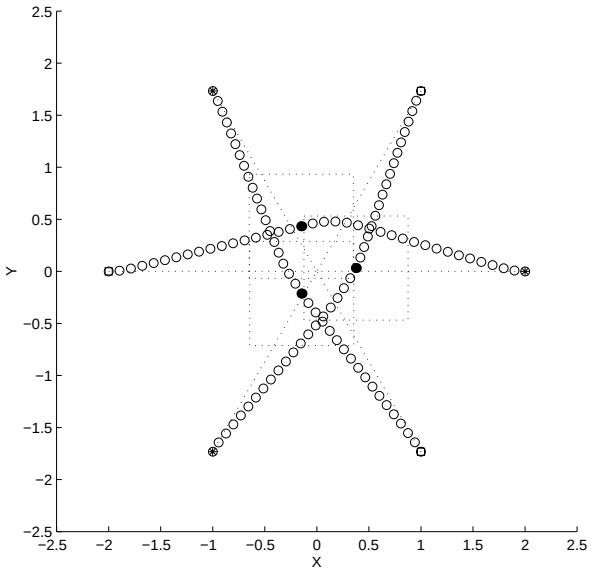

FIGURE 6. *Paths created for 3 planes. The black circles represent the three aircraft, the starred circles the goal waypoints, and the white circles the paths planned by the MILP algorithm*
Image credit: [**?**]

globally optimal path may be used which is suboptimal (and thus not chosen otherwise) for an individual aircraft. For instance, Figure 6 shows three cooperative aircraft each being given a path modified slightly from a straight line in order to collectively reach their goals in minimal time. Sadly, when using aircraft in cooperation, the time required to solve a system of MILP equations increases sharply with the number of aircraft.

MILP-planned paths are guaranteed to produce optimal paths. In practice, however, the time complexity renders problems of even moderate size intractable. Unlike A* search, whose time complexity decreases with the number of constraints on the path—that is, with the specificity of the problem—the time complexity of MILP *increases* as more constraints are added; more equations must be optimized, resulting in a significant increase in time complexity.

2.5. **Discussion.** The path planning and collision avoidance algorithms reviewed above represent different means of dealing with the fact that the problem is NP-complete. Each algorithm sacrifices either speed of computation or optimality of path, and it is clear that each is suited to specific applications. Instances where computing power is very limited would benefit from APF or geometric approaches, whereas situations with greater computing power or less strict time requirements would wisely implement an A*- or MILP-based approach.

3. Dynamic Sparse A* Search

For our own research, we implement a version of A* search drawing largely from the Sparse A* Search method of Szczerba et al. [**?**]. Our reasons for this are as follows:

- A* provides globally optimal paths, given the algorithm's current state of knowledge.
- The time complexity of the algorithm increases linearly with the number of aircraft used.
- The algorithm is massively parallelizable; each aircraft, and even each estimate produced by the heuristic function, can potentially use its own processor with near-perfect scaling.
- Using a strong heuristic, A* can plan paths in our airspace in a very short amount of time.

Our algorithm, Dynamic Sparse A* Search (DSAS), differs from traditional implementations of A* in a number of ways. First, the node expansion by A* is sparse in that it considers a minimal area outside the straight-line path from an aircraft's current location to its goal. Furthermore, in situations with no predicted danger between an aircraft and its goal, no search via A* is necessary; by checking this case first, the algorithm refrains from any unnecessary computation.

Next, DSAS is dynamic in that it is designed to plan around moving obstacles; it searches the airspace as it is expected to be through time. This means the paths planned are bounded in their optimality only by our accuracy in predicting the future locations of other aircraft, and in estimating a plane's own speed in traversing the airspace. Finally, the search is dynamic in that it is able to quickly respond to changing conditions in the air—in our simulations, we find that more than 20 aircraft can plan their paths via A* once per second on a single laptop computer.

Conceptually, the DSAS algorithm may be divided into two parts: heuristic generation and true search. The heuristic function is calculated by rating the danger of each location in the airspace through time (obtained by predicting the future locations of all other aircraft in the airspace), then using that information to estimate the cost of a best path from each node in the search space to the aircraft's goal. This heuristic is then used in

the A* search itself, which consists of three steps: setup for a sparse search, Dynamic Sparse A* search, and A* collision analysis.

3.1. **Creation of the Heuristic Function.** As discussed previously, the key to reducing the time complexity of A* lies in the heuristic function [**?**]. The heuristic function is calculated as

$$f(n) = g(n) + h(n).$$

Since $g(n)$ is the *known* best cost from the start node to a node n, the only thing differentiating a good heuristic from a bad one is $h(n)$, the estimated cost of the best path from n to the goal node. In order to guarantee optimality in the generated path, $h(n)$ must never overestimate the cost from n to the goal. However, the closer its estimate is to the actual cost, the faster A* will be in finding the nodes constituting the optimal path.

3.1.1. *Background Concepts (Dynamic Planes, Static Maps).* Searching a continuous, dynamic, real-world airspace via A* requires that the airspace be represented as discrete "squares" in a grid, and that dynamic changes over time be modeled statically. While the idea of using A* in dynamically changing search spaces has been discussed elsewhere in the literature, it has not, to our knowledge, been fully explored by other authors.

To represent the cost of traversing our airspace, we use a model with length, width, and time dimensions (corresponding to the length and width of our airspace, divided into grid squares based on a pre-defined resolution, and the maximum distance in the future for which we predict the locations of other aircraft). This is our "danger grid," which indicates the likelihood of encountering an aircraft at each (x, y, t) triple. The set of all such (x, y, t) triples, corresponding to all spatial locations at all times, constitutes our "prediction space."

The $h(n)$ function—the estimated cost of the best path from n to the goal—is calculated using the danger grid. It is stored as a "best cost" grid, inspired by, but significantly modified from, the BC array used by Szczerba et al. due to its inclusion of a time dimension [**?**, **?**].

For our purposes, we are concerned only with collision avoidance in two spatial dimensions. Though adapting our algorithm to three spatial dimensions would be relatively straightforward (one would simply add a fourth dimension to the prediction space), doing so would bring a significant increase in time complexity. (This increase could potentially be alleviated by implementing a *preference* for two-dimensional avoidance, and searching in three dimensions only when a two-dimensional path is not viable or of sufficiently low cost.)

In our algorithm the search space is a grid (a set of nodes to be searched by A*) of fixed width and length, divided according to some pre-defined grid resolution. It has a "depth" determined by the number of time steps in the future to which we predict. In our case, we use time steps of 1 second, and we predicted plane locations up to 20 seconds in the future. Thus, using a 1000 m by 1000 m airspace, with a resolution of 10 m per grid square, our prediction space is 100 by 100 by 21 grid squares (where the time component is 1 greater than the number of future time steps because we store $t = 0$, the present).

Each square in the danger grid stores a "danger rating," a value representing how likely it is to encounter another aircraft in that square. This value corresponds to the map cost referred to elsewhere in the literature [**?**, **?**]; as our cost metric is concerned only with the danger posed by other aircraft, it is simply more evocative to think of this as a danger rating than a map cost.

To calculate the danger rating for a square, we predict the future locations of all aircraft based on their latitude, longitude, and bearing as reported by the GPS onboard each aircraft, and we add danger to squares in the set of grids based on these predicted

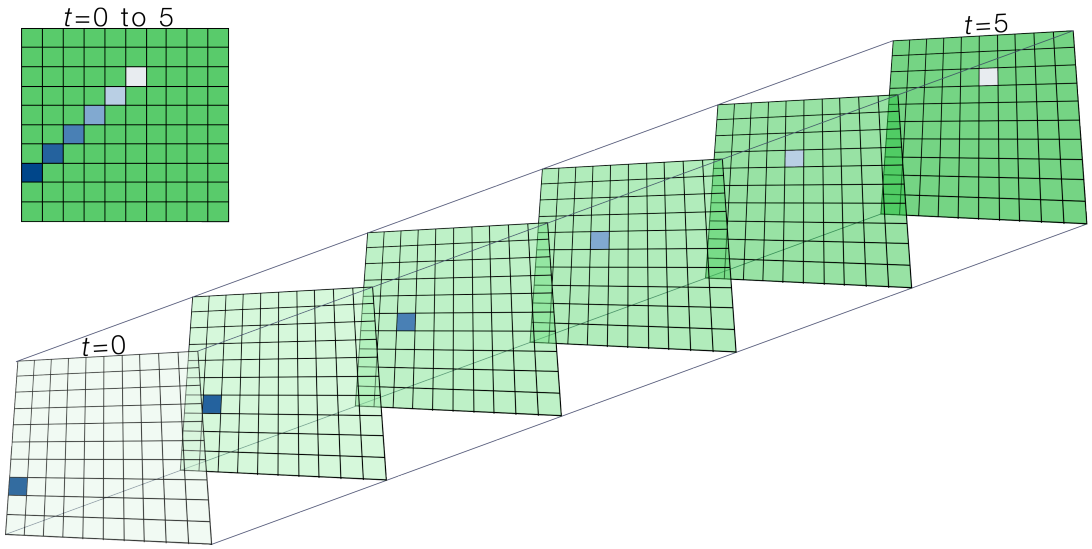

FIGURE 7. *Danger grids through time*

locations. Since each aircraft creates its own danger grid to refer to when planning a path, each aircraft is said to "own" a danger grid. In the grid owned by each aircraft, the future locations of the owner are not predicted (if they were, a plane might attempt to avoid *itself* when planning its path via A*).

For instance, Figure 7 illustrates a grid with a single aircraft which is predicted to move diagonally across the airspace, beginning in the lower left. With the six time steps overlaid, we see a smooth diagonal path predicted, whereas in the grid for each individual time step, the plane is predicted to be in one and only one square.

Note that Algorithm 3 details the process of creating a danger grid.

3.1.2. *Predicting Plane Locations.* Predicting the straight-line path a plane will take in a continuous space is a rather simple task. However, once the area is discretized, things become more difficult. The difficulty comes from the fact that for a line to be "straight" in a grid-based system, it has to bisect every square it passes through. In most cases however, a plane's path will take it through different portions of many different squares. Because of this limitation, we can only predict the probability that a plane will move through a square over a certain period of time. This probability is calculated by finding the percentage offset from the plane's bearing to the goal, θ_{goal}, to the closest angle that perfectly bisects a neighboring square, θ_{bisector}. The percentage offset is representative of how far from a perfect bisector θ_{goal} is and how much time, over the next second, the plane will spend in the square bisected by θ_{bisector}. The remainder from 100% of the offset is the amount of time that the plane will spend in the next closest square to the one bisected by θ_{bisector}. Once the two probabilities are calculated, they are inserted into their respective squares and the plane is predicted to have moved into the square with the higher probability. This process is then repeated until the plane is predicted to be in the goal square. Algorithm 1 depicts this process.

In this way, the straight-line movement of an aircraft from place to place can be predicted fairly accurately. Accurately predicting turns, though, proves to be more difficult. Our airplanes can change their heading approximately 22.5°/second, so the simplified method we use to predict their turns is to refrain from predicting straight lines until the plane's bearing is within 22.5 degrees of θ_{goal}. This occurs on line 4 in Algorithm 1. While this method does predict *plannable* turns (such as those required to go from one goal to

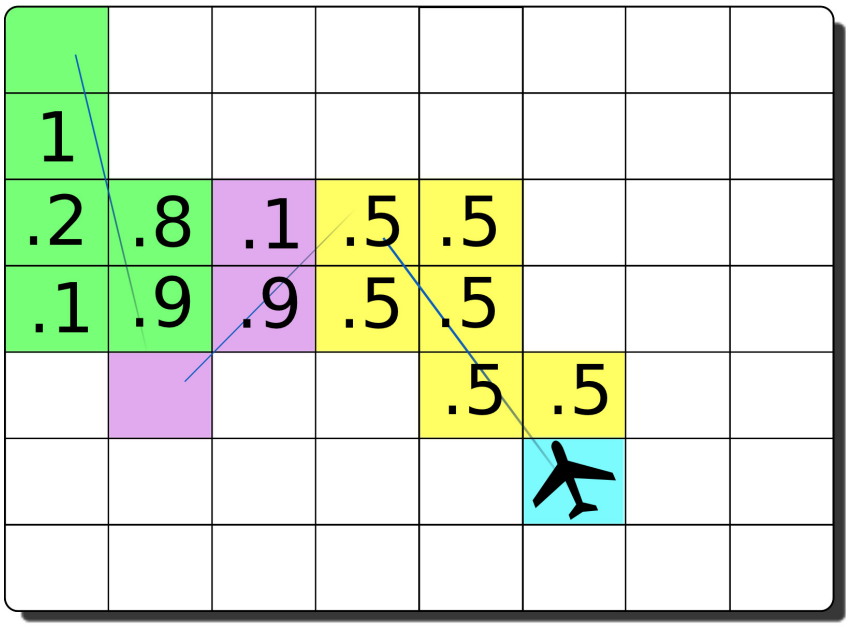

FIGURE 8. *A plane's predicted path. The yellow path is a predicted turn. The purple represents the path to an intermediate waypoint and the green is the final path taken to the planes goal.*

another), it is unable to predict turns made for collision avoidance. Predicting collision-avoidance turns, of course, would require a collision avoidance plan ahead of time. In most cases, though, the prediction we have implemented is adequate. An example of a predicted path is given in Figure 8

After the generation of a plane's predicted path, a high-danger buffer is added in the grid squares surrounding the plane. This buffer is needed to prevent conflicts and to increase the leeway given to the predictions. In the danger grid algorithm, Algorithm 3, this occurs on line 9.

To illustrate, the plane in Figure 9 represents the predicted location of an aircraft at some time step, with its calculated bearing indicated in blue. The danger rating of this square (colored green) in the danger grid would be the maximum danger allowable to a square, indicating that under no circumstances should A* plan for another aircraft to be in that location in space and time. The danger of the three squares directly in the path of the aircraft (colored dark purple) would be scaled down from the maximum allowable danger, perhaps by one half. The squares directly to the left and right of the aircraft (colored light purple) might have a further scaled danger applied to them, perhaps one quarter the maximum danger. The number of squares included in this buffer must depend on the speed of the aircraft relative to the resolution. However, in our implementation, it happens to be the case that a buffer of one grid square around each aircraft is sufficient to maintain a comfortable distance between all airplanes.

3.1.3. *Estimating the Best Cost to the Goal.* Having predicted each aircraft's location through time and stored each related "danger rating," we are able to easily produce an $h(n)$ estimate, which we store in a "best cost" grid. Like the danger grid, this has dimensions corresponding to our prediction space. Unlike the danger grid, however, which stores the likelihood of encountering another aircraft at each location, the best cost grid stores the estimated cost of the best path from each location to an aircraft's goal (that is, of course, $h(n)$).

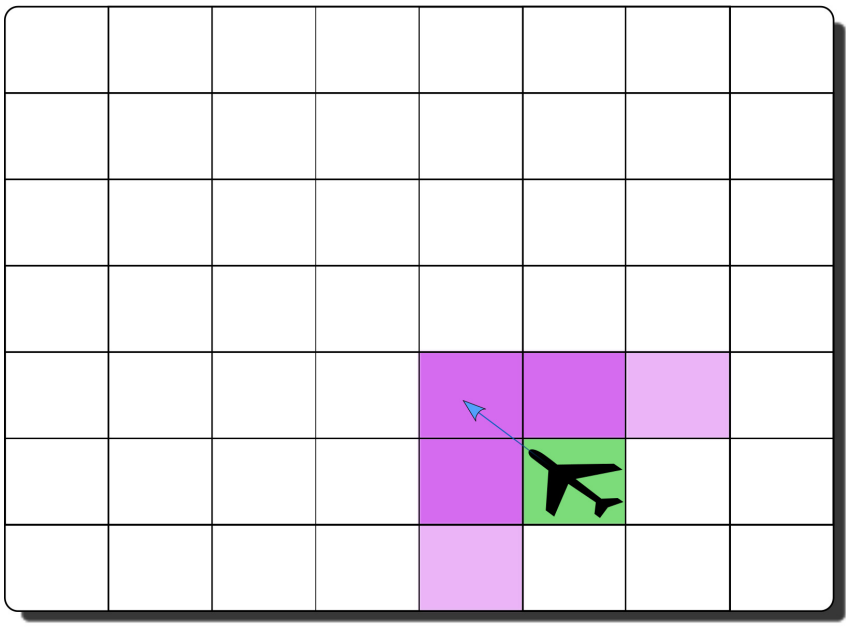

FIGURE 9. *Plane buffer zones*

Because our airspace is relatively sparse (as any reasonably safe airspace must be) and without static obstacles, our $h(n)$ computation is far simpler than others in the literature. As indicated by Algorithm 2, the estimated best cost from a node n to the goal is simply:

$$\begin{gathered}\text{distance from } n \text{ to the goal} + \\ \text{danger rating of } n \cdot \text{scaling factor.}\end{gathered}$$

That is, we consider only the cost of traveling the distance to the goal and danger cost associated with the node itself. The rationale for this is as follows.

In a relatively sparse airspace, the additional distance required to deviate around individual aircraft which may lie between node n and the goal will be low—low enough that it will never be preferable to pass through a dangerous area and cause a conflict between the paths of the aircraft involved. Since we are concerned only with danger associated with flying too close to other aircraft and not large, static, dangerous regions such as mountainous terrain, we can safely assume than any danger between n and the goal can be avoided with a small deviation in the flown path.

Furthermore, traditional means of estimating best cost (such as in [**?**]) fail to model the distances associated with traversing a real-world airspace; they assume that there is only one possible path from an arbitrary node n to a successor n', a path of constant distance. The DSAS algorithm, however, will command an aircraft to simply fly a straight line from the node to the goal if it finds that there are no obstacles between the two. In this case, unless n happens to have a bearing which is an exact multiple of 45° away from the goal (causing the aircraft to pass through each grid square at the same bearing, at least across small distances), the path traveled will often be *shorter* than that predicted by the traditional best cost calculation; the aircraft will pass through only small portions of many grid squares. Since A* is only guaranteed to find optimal paths when $h(n)$ never *overestimates* the cost from a node to the goal, this failure of the heuristic will result in a loss of optimality. In contrast, because our estimate relies on the straight-line distance to the goal along with the cost associated with the only grid square which must *always* be traversed in a given time step (i.e., the node itself), it cannot overestimate the path cost.

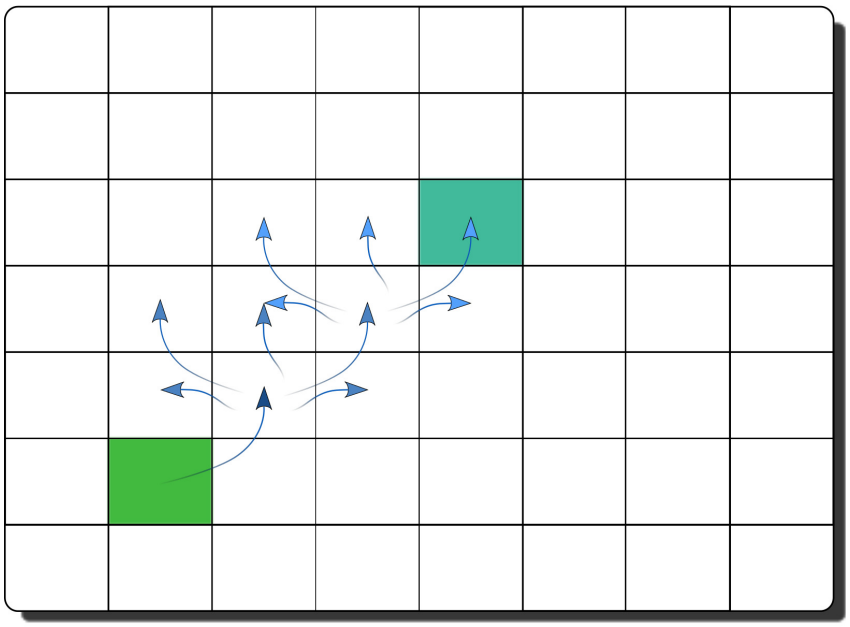

FIGURE 10. *Updating select squares in a traditional best cost grid*

Our method for creating a best cost grid also has a significant advantage in terms of time complexity. The method of Stiles and Glickstein (detailed in Algorithms 4 and 5) is potentially of exponential time, depending on the distance of the aircraft from the goal and the "threshold cost"; each node may update its neighbors, which may in turn update their own neighbors, and so on (as illustrated in Figure 10) until the relationship between all nodes and their neighbors has stabilized.

In contrast, our "Sparse Best Cost Grid" is $O(lwtp)$ (where l and w are the length and width of the airspace in grid squares, respectively, t is the number of time steps predicted in the future, and p is the number of planes in the airspace). Thus, given a real-world airspace, wherein the number of aircraft is always limited by the size of the airspace, the time required to create a Sparse Best Cost Grid grows linearly with the number of grid squares in the prediction space.

Using a modern laptop CPU, we find that it takes an average of 28.36 ms to generate a best cost grid of 210,000 total grid squares (the number of squares resulting from using a resolution of 10 m per square, with 20 seconds of future prediction, on a 1000 m square field); as predicted, we find that the time required grows linearly with the size of the prediction space, as indicated by Figure 11. To obtain these measurements, we found the number of grids generated in 2 minutes using each of 30 random (seeded) sets of airplanes, with one set of 20 planes per grid size tested. This measurement represents the time required for all components of our algorithm *except* the actual search by A*—it includes the prediction of plane locations, creation of the danger grid, and the creation of the best cost grid therefrom.

3.2. **Search via Dynamic Sparse A*.** As indicated above, DSAS is differentiated from other implementations of A* in its use of sparsity and its ability to search through time.

When a traditional implementation of A* branches from a given node, it assumes that all neighbors of that node are open to travel. This is not the case in the real world; if A* believes that it is more optimal for an aircraft to reverse direction immediately, there is no way the plane can comply. In light of this, we limit the nodes open to branching based on the plane's bearing at a given time and its maximum turning angle. For instance,

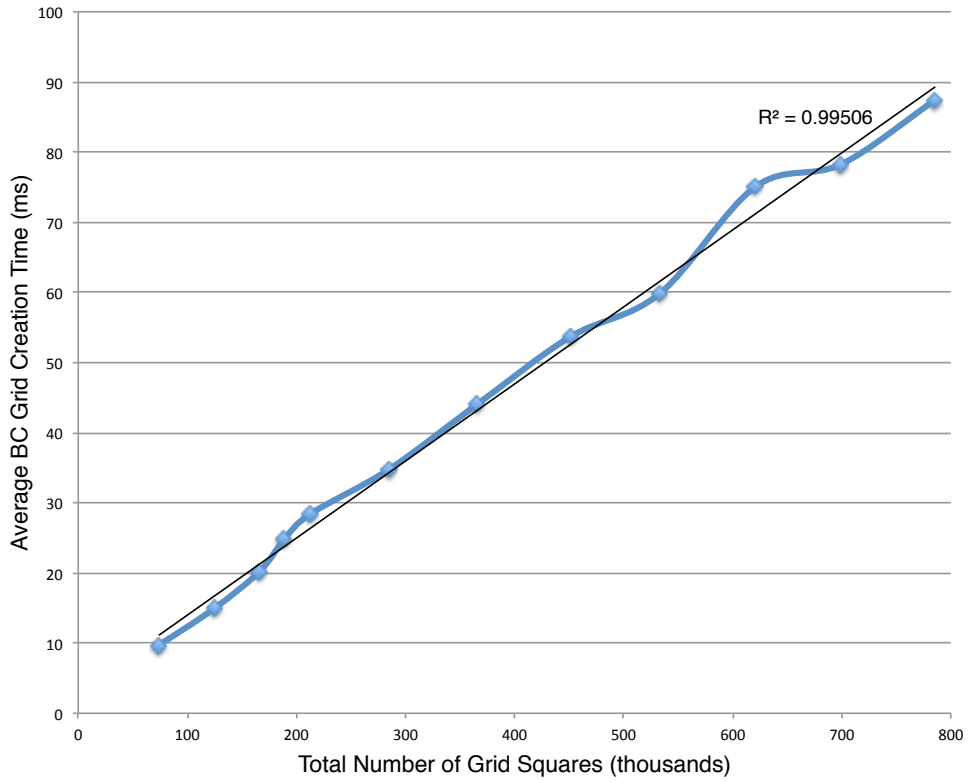

FIGURE 11. *Best cost grid creation time by total number of grid squares*

in our simulations, we limit the aircraft to 22.5° deviation per second. Correspondingly, in the first time step, A* never branches to nodes outside of a 45° "swath" in front of the plane. Then, in future time steps, this 45° swath is calculated based on the bearing between the current node and its parent. By reducing the area that A* searches in this way, we not only ensure that the paths generated are flyable, but we also reduce the time required for search. This time reduction adds great value to a sparse implementation of A*.

Further reductions in computation time are obtained by checking whether the straight line path from a plane's location to the goal is clear. If the path is clear (i.e., without danger in future time steps), the aircraft is commanded to head directly to its goal, since no search could generate a better path.

In our implementation, the DSAS algorithm is used as a function call from the component of our UAV control system responsible for commanding the planes' autopilots. Each time the controller receives a telemetry update from an aircraft, it creates the best cost grid for that aircraft and runs the A* search on that grid.

Because our implementation searches via A* approximately once per second, we do not return the full set of waypoints planned in the optimal path as a command to the aircraft's autopilot. Instead, we have the autopilot head to the first point of deviation between the "ideal" path (a straight line from the aircraft to the goal) and the path planned by A*. Thanks to the turning radius constraints included in A*, this point of first deviation is at least 2-3 seconds away from the aircraft's current location and reachable with regard to the plane's performance characteristics. In cases where the airspace is predicted perfectly from an aircraft's start location to its goal, the set all such points of first deviation will be identical to the full set of optimal waypoints planned in the first search by A*.

In addition to the deviation between the "ideal" path, A* also has to check for several other types of problematic scenarios. These problematic scenarios fall into three categories. In the first, the plane's bearing is roughly opposite of the goal. In the second, the plane has encountered a dangerous point in A*'s recommended path. In the third, the plane is told to change its bearing according to A*'s recommended path.

The first scenario occurs when our plane is currently facing away from its goal and will need to turn around. Without considering this situation, we would always find that the first point of deviation between A*'s path and the optimal one is the first square in the path. If we always return a collision avoidance waypoint directly in front of the plane, the plane would continue to fly forward in a straight line. Our aircraft, though, are only allowed to deviate 22.5° per second; a diagonal move in a grid requires up to a 45° turn in the grid space, meaning we can only command significant turns that are 2 time steps or more in the future.

In order to resolve this situation, we only use A*'s queue of points to determine our next move. If we encounter danger during our turn, we calculate a safer move that is close to the turning point if possible otherwise we continue on the current path. If we manage to reorient our plane toward the goal without encountering danger, then we can execute a safe turn; we return that position as the next for the plane.

The second scenario is similar to what happens if we encounter danger in the first scenario, but with one significant difference. If we encounter danger between three and eight seconds of time, we determine where the danger is coming from using an immediate- to medium-term avoidance planner. The medium-term planner determines which threat is closest to our aircraft and then gives an avoidance point with respect to that specific threat. For example, if a threat is coming from the east and we are coming from the south, this planner would command our UAV to fly behind the eastbound aircraft by giving the threat's position at $t = 0$ (the present). This would give our aircraft a few seconds to divert itself toward a position we know will not be occupied by our current threat (as we will then be behind it), instead of trying to race ahead of it.

The third scenario is the opposite of the first. In this case, we need to turn our aircraft toward a goal, but we already have a bearing similar to the goal. In this event, we command the aircraft to go to the point of divergence between the straight line from our current position to the goal and the position at which we need to turn. As we get closer to this point, our orientation will start to change as we turn to face our new path.

3.3. **Results.** Our results are divided into two parts. In the first, we use a simulation that limits a virtual aircraft to moving to any of the five grid squares which are in front of and to the side of the aircraft. This corresponds to a maximum change of heading between 45° and 90°/second, depending on the aircraft's position and bearing within its grid square. In the second simulation, all aircraft are limited to 22.5° changes of heading per second, a good estimate of the real-world performance of our small (1370 mm-wingspan), low-speed UAVs.

Because our testing aims to push the algorithm to its limits, the simulations use airspace sizes ranging from 430 m by 470 m (in the case of the high turning capability simulations), to 500 m by 500 m and 1000 m by 1000m (in the case of the limited turning capability simulations). The tests vary the number of aircraft, from 1 to 15 in the case of the small-airfield, high-turning capability simulation, and from 8 to 32 aircraft in the case of the large-airfield, limited-turning simulation. All tests used sets of random waypoints (which stayed consistent across trials) for all planes.

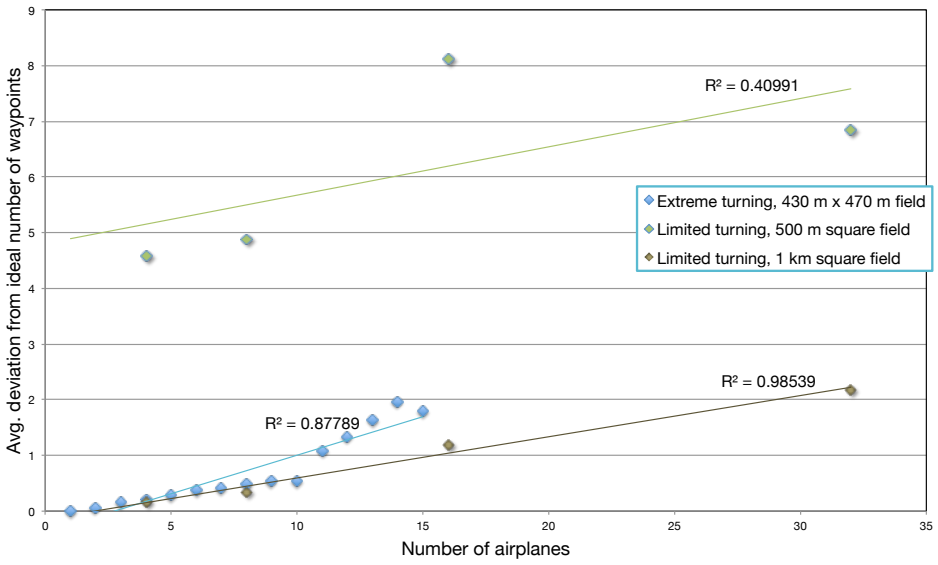

FIGURE 12. *Average deviation per plane from ideal number of waypoints (430 m by 470 m field, 45°/second turns)*

The two types of simulation are included as a means of demonstrating the algorithm's performance in two very different applications. The simulation with extreme turning capability evaluates the algorithm without regard for its accuracy in predicting other aircrafts' turns, or for planning its own. In applications using goal waypoints which are laid out in a roughly linear fashion, requiring only slow changes of bearing, this simulation is a good representation of the algorithm's performance.

In contrast, the other simulation relies heavily on the algorithm's accuracy in predicting the turns made by other aircraft and its efficiency in planning its own; in the small airspace, a great deal of each aircraft's time is spent turning from one waypoint to the next. In applications using goal waypoints which may not be arranged linearly, with some waypoints requiring complete changes of direction, this is a much better evaluation of the algorithm.

As indicated by Figures 13 through 12, our results in the limited-turning simulations are less than desirable. With 16 airplanes in the 500 m square airspace, we have as many as 4 potential collisions (aircraft within 12 meters of one another). Even moving to the 1 km square field, two simulations in three produce a potential collision using 16 UAVs. In contrast, using the extreme turning capability simulation, we have *zero* total collisions over 27 simulations using 15 aircraft. Ironically, our deviation from the ideal number of waypoints *decreases* when moving from 16 to 32 aircraft in the 500 m square field, due to the speed with which the 8 to 10 collisions occur—over half of the time spent in the 32-plane tests had fewer than 20 aircraft remaining.

In the extreme turning capability scenario, we find that 10 airplanes average only 0.55 fewer waypoints than ideal (where the ideal number of waypoints is the number achievable by flying entirely straight lines, with no penalty for collisions). This deviation from the ideal number of waypoints rises as more aircraft are added to the small airspace, up to a maximum of 1.95 with 14 aircraft.

3.4. **Analysis and Discussion.** The results above summarily indicate that the algorithm's primary weakness lies in dealing with turns. This may be due to a number of factors, but apart from further testing, we are unable to say exactly which portion of the

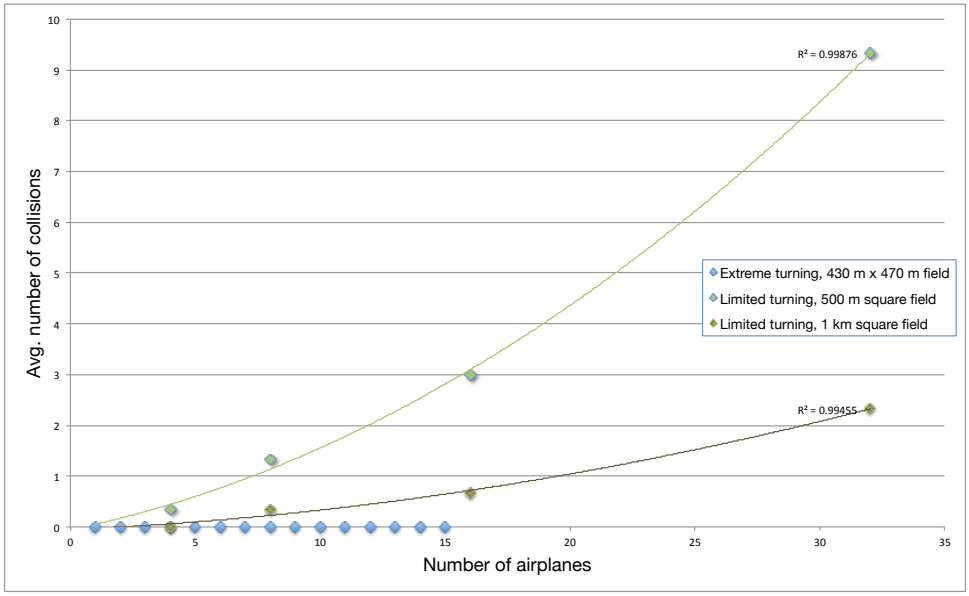

FIGURE 13. *Number of collisions by number of planes in the airspace (500 m by 500 m field, 22.5°/second turns)*

algorithm is to blame. Failures in the plane prediction function are likely; simply put, it is extraordinarily difficult, without sending all path-planning commands from one source, to discern when and how other aircraft will turn toward their goals. In examining the flown courses, we find that a very high percentage of collisions occur during or immediately after a plane changes its heading in a major way. Because the goal waypoints were random and in a small area, we find that much of each aircraft's time is spent in such turns.

The algorithm does quite well when given extreme turning capabilities. In its present state, then, it is only truly viable in situations where few major turns are expected. We expect that further testing, using non-random waypoints arranged in a roughly linear fashion, would produce results more in line with those of the extreme capability situation, where the difficulties in predicting turns pose less of a liability. We believe, however, that the results from the extreme turning scenario indicate the potential performance which may be realized after further work with our algorithm.

4. Future Research

Regarding the best cost grid, changes through time represent a significant difficulty. The Sparse Best Cost Grid described here takes no account of a node's neighbors, either in space or time. While this is often a justifiable assumption in sparse, dynamic airspaces, a much more accurate $h(n)$ estimate might be obtained by incorporating into this estimate the danger of other nodes which lie along the path from n to the goal. Two difficulties exist which prevent us from using such a method here.

First, calculating the distance costs from a node to the goal cannot be done accurately using methods described elsewhere in the literature, as these implicitly assume that the distance associated with moving from one node to a given neighbor remains constant; the discretized airspace belies the fact that a real-world aircraft which is represented in the grid as moving from, for instance, node $(10, 20)$ to $(11, 21)$ might actually move from what would be $(10.89, 20.7)$ to $(11.02, 21.1)$ in a continuous airspace.

Second, and more importantly, using a method of $h(n)$ estimation which relies on nodes updating their neighbors, nodes must update their neighbors through both space

and time. For instance, changing the best cost estimate at $(15, 15, 2)$ (in (x, y, t) order) may change the estimate for $(15, 16, 3)$ and $(15, 16, 1)$. Since the distance an aircraft travels through a square is not constant, however, one cannot assume that an aircraft will traverse a constant number of grid squares in a time step; thus, the squares $(15, 16, 2)$, $(15, 17, 4)$, $(15, 17, 1)$, and so on. Updating even a very small number of nodes in this way will take far too long to be feasible in real-time. Difficulties relating to the grid-distance traveled in a given amount of time could be solved by using a resolution much greater than could be traversed in a given second, leading to occasional inaccuracies in prediction and significantly worse paths planned (corresponding to the larger deviations made by aircraft to avoid one another).

Thus, a great deal more research may be done regarding efficient, accurate implementations of the $h(n)$ heuristic generation, whether these focus on overcoming the drawbacks to the "traditional" approach to the BC creation or on entirely new methods.

In addition to improvements that could be made in heuristic generation, future research could aid in creating much more accurate plane predictions. In our current implementation, turns are particularity hard to predict, especially due to the square nature of the grid space we work within. Research using hexagonal grids, for their 22.5° angles between sides, or more accurate ways of predicting turns in a grid space would bring about major improvements to the algorithm as a whole. Using methods such as heat maps or mean weighted averages would allow past (actual) plane locations to greatly improve the accuracy of the planes' predicted paths. The more accurate our plane prediction system is the more accurate A* can be at guiding planes around collisions; as such, any improvements in plane prediction would be of great value to the algorithm.

With respect to A*, future research revolves around two distinct concepts: parallel A* processing and medium-term planning. If A* could expand nodes in parallel, challenges not withstanding, then it would drastically cut down the time demands to expand most nodes. Consider the case of a 100 x 100 grid with 20 time steps and 8 possible bearings per square per time step: $100 \cdot 100 \cdot 20 \cdot 8 = 1,600,000$ potential positions. With 1.6 million possible nodes to expand, A* could stand to benefit from balancing the workload across CPUs.

For medium-term planning, A* returns a path, but it is not entirely obvious what would be the best point of deviation to pass to an aircraft. If A* always follows the deviation at $t = 2$ or $t = 3$, it is missing the deviations at $t = 7$, $t = 8$, and so on. The medium-term planning becomes even more important with turn restrictions below 45°, since it will take a medium amount of time (3-8 seconds) to execute a proper deviation.

5. List of Algorithms

Input: A plane object that is aware of both its current location, the location of its goal and its current bearing θ_{plane}.
Result: A list containing the predicted locations of a plane as it moves towards its goal.

set current location to plane's current location
while not at goal square
  calculate θ_{goal}
  while $\theta_{\text{goal}} - \theta_{\text{plane}} > 22.5$
    if goal is on right **then**
      insert danger into 45 degree neighbor on right
    if goal is on left **then**
      insert danger into 45 degree neighbor on left
    change current location to square with danger decrease θ_{plane} by 22.5
  find θ_{bisector}, the closest angle to θ_{goal}that bisects a neighboring square
  calculate % offset of θ_{goal} from $\theta_{\text{bisector}_1}$
  insert % offset into square bisected by $\theta_{\text{bisector}_1}$
  insert $1 -$ % offset into square bisected by $\theta_{\text{bisector}_2}$
  if $1 -$ % offset $> .3$ **then**
    change current location to square bisected by $\theta_{\text{bisector}_2}$
    branch to step 3
  change current location to square bisected by$\theta_{\text{bisector}_1}$
  branch to step 3

Algorithm 1: Predicting planes for use in the heuristic.

Input: A set of aircraft in the airspace, dimensions of the airfield, and the "owner" of this grid
Result: A set of representations of the airspace, each (x, y, t) square of which stores $h(x, y, t)$

Create *Danger Grid* (Algorithm 3) using set of aircraft, dimensions of airfield, and "owner" of the grid
foreach $(x, y, t) \in$ prediction space
  $BC_{xyt} :=$ distance from (x, y) to $(x_{\text{goal}}, y_{\text{goal}}) + DG_{xyt} \cdot$ danger multiplier

Algorithm 2: Creation and Initialization of the Sparse Best Cost Grid

Input: A set of aircraft in the airspace, dimensions of the airfield, and the "owner" of this grid

Result: A set of representations of the airspace, each (x, y, t) square of which stores the likelihood of encountering an aircraft there

Create the "prediction space" array, with danger of all grid squares zero
foreach plane $\in$ set of aircraft in the airspace
  if plane $\neq$ owner of this danger grid **then**
    / Set danger at plane's starting location to max danger */*
    $DG_{x_{\text{start}} y_{\text{start}} 0}$:= plane danger
    Get a list of likely future positions for this plane from *Plane Prediction*
    foreach danger estimate $i \in$ list of future positions
      DG_i += danger_i */* Add to danger at this square*/*
      foreach neighbor j of i in the direction of plane's travel
        $DG_j := \text{danger}_i \cdot$ scaling factor

Algorithm 3: Creation of the Danger Grid

Input: A set of aircraft in the airspace, dimensions of the airfield, and the "owner" of this grid

Result: A set of representations of the airspace, each (x, y, t) square of which stores $h(x, y, t)$

begin Initialization
  Create *Danger Grid* (Algorithm 3) using set of aircraft, dimensions of airfield, and "owner" of the grid
  foreach $(x, y, t) \in$ prediction space
    $BC_{xyt} := \infty$ */*Max double */*
  foreach $(x, y, t) \in$ prediction space
    MF_{xyt} := FALSE
  To-do list is empty
begin Set best cost along starting path
  foreach $t \in$ time predicted
    Set of updating squares := 3 squares adjacent to $(x_{\text{goal}}, y_{\text{goal}})$ which are closest to $(x_{\text{start}}, y_{\text{start}})$
    while $\neg((x_{\text{start}}, y_{\text{start}}) \in$ set of updating squares)
      "Branching square" := square closest to $(x_{\text{start}}, y_{\text{start}}) \in$ set of updating squares
      Set of squares to change := 3 squares adjacent to the branching square which are closest to $(x_{\text{start}}, y_{\text{start}})$
      foreach $(x, y) \in$ set of squares to change
        Optimal square := square of minimum cost which is adjacent to (x, y)
        BC_{xyt} := cost of optimal square + distance from (x, y) to optimal square + $DG_{xyt} \cdot$ danger multiplier
        Add (x, y, t) to the end of the to-do list
        MF_{xyt} := TRUE
      Set of updating squares := set of squares to change
Call *Minimize Cost* (Algorithm 5)

Algorithm 4: Creation and Initialization of the Traditional Best Cost Grid (Adapted from [**?**])

Input: A best cost grid with a starting path and to-do list initialized
Result: A set of representations of the airspace, each (x, y, t) square of which stores $h(x, y, t)$

```
1  begin Minimize cost
2     Cost threshold := BC_{x_start y_start t}
3     while ¬(to-do list is empty)
4        Square i := last element of to-do list        /* where i is an (x, y, t) triple */
5        Remove last element from to-do list
6        MF_i := FALSE
7        foreach neighbor j of square i
8           cost to check := BC_i + DG_j · danger multiplier + distance from
            (x_i, y_i) to (x_j, y_j)
9           if cost to check ≤ BC_j ∧ cost to check ≤ cost threshold then
10             BC_j := cost to check
11             if MF_j == FALSE then
12                Add j to the to-do list
13                MF_j := TRUE
```

Algorithm 5: Cost Minimization in the Traditional Best Cost Grid (Adapted from [**?**])

Input: A set of representations of the airspace, each (x, y, t) square of which stores $h(x, y, t)$, a structure with all planes bearing and current location, current plane ID, start position, goal position
Result: A collision avoidance point (x, y, t)

```
1  begin Dynamic Sparse A*
2     initial bearing := bearing of the plane in the airspace
3     goal bearing := bearing of the goal from the planes current position
4     if initial bearing == goal bearing then
5        if direct path to goal is clear then
6           return goal position
7  Call Construct Optimal Path in Grid Space (Algorithm 7)
8  Call Dynamic-Sparse A* Search (Algorithm 8)
9  return Call Dynamic-Sparse A* Collision Avoidance and Analysis (Algorithm 9)
```

Algorithm 6: Dynamic-Sparse A* Algorithm

Input: Start Position, End Position
Result: A queue of optimal points from start to end

```
1  begin Construct an Optimal Path in Grid Space
2     Optimal Path := ∅
3     Current := Start Position
4     while (Current != End Position)
5        Push Current onto Optimal Path
6        Cheapest Next Move := ∞
7        foreach x ∈ squares around Current
8           distance := Euclidean Distance of x to End Position
9           if distance < Cheapest Next Move then
10             Cheapest Next Move := distance
11             Current := x
```

Algorithm 7: Construct an optimal path in grid space using Euclidean distances.

Input: Start Node, End Position, Heuristic values: $h(x, y, t)$
Result: A queue of the path A* creates

```
1  begin Actual Path Construction using Dynamic-Sparse A*
2     Current Node := Start Node
3     Current Node Possible Children = Next Grid Space in Bearing of Plane while
      Current Node != End Position
4        Time := Parent Node Time + 1
5        if Time > maximum t of h(x, y, t) then
6           Time := maximum t
7        while Current Nodes Possible Children is not Empty
8           Child := Front of Current Nodes Possible Children queue
9           Child Nodes Possible Children := (Add child iff in bounds)
10                First Node := If we alter current bearing - 1
11                Second Node := If current bearing is continued
12                Third Node := If we alter current bearing + 1
13          Push Child onto A* Node list with time Time and h value of h(x, y, t)
14       Current Node = Next Least Expensive Node in A* list
15    A* Solution Queue := ∅
16    while A* Solution List is not Empty
17       Current Node = Top of Solution List
18       Push Current Node into A* Solution Queue
19       Erase Current Node from Solution List
20    return A* Solution Queue
```

Algorithm 8: Dynamic-Sparse A* Search

Input: Queue of A* Path, Queue of Optimal Path, Start Position, End Position, Plane Bearing, Goal Bearing, $h(x, y, t)$
Result: Collision Avoidance Point

```
1  begin Dynamic-Sparse A* Collision Avoidance and Analysis
2    medium term planning range := 3 seconds to 8 seconds
3    if Plane Bearing is not similar to Goal Bearing then
4      while A* Path is not Empty
5        Current Position := Front of A* Path Queue
6        Current Position Danger := h(Current Position)
7        Euclidean Position Distance := Current Position distance from goal
8        if Current Position Danger > Euclidean Position Distance then
9          if Current Position time is within medium term planning range then
10           return Call Immediate Avoidance Point (Algorithm 10)
11         else
12           return Current Position
13       if Current Position bearing is similar to Goal Bearing then
14         return Call Immediate Avoidance Point (Algorithm 10)
15   else
16     while A* Path is not Empty and Optimal Path is not Empty
17       Current Position := Front of A* Path Queue
18       Current Position Danger := h(Current Position)
19       Euclidean Position Distance := Current Position distance from goal
         Optimal Position := Front of Optimal Path Queue
20       if Current Position Danger > Euclidean Position Distance then
21         if Current Position time is within medium term planning range then
22           return Call Immediate Avoidance Point (Algorithm 10)
23         else
24           return Current Position
25       if Current Position != Optimal Position then
26         return Current Position
27       if Current Position is different from Plane Bearing then
28         We are starting to turn, so we must return the first turning point
29         return Current Position
30   Since we have no return yet, one of the two queues has run out of points
31   return Current Position
```

Algorithm 9: Dynamic-Sparse A* Collision Avoidance and Analysis

Input: List of Planes, Start Position of Plane, Dangerous Point, $h(x, y, 0)$
Result: Collision Avoidance Point

begin Immeidate Avoidance Point
  Search a square area from *Dangerous Point* +/- a size determined by the length between *Start Position* to *Dangerous Point*
  if A hostile plane is found in a specific square at time = 0 and is closer to the *Start Position* compared to previous Hostile Planes **then**
    Hostile Plane = Plane at $(x, y, 0)$
  After Searching the Space:
  if No hostile plane found **then**
    return *Dangerous Point*
  else
    Avoidance Position := determine based on *Hostile Plane* bearing and position compared to current Plane bearing and position
    return *Avoidance Position*

Algorithm 10: Immediate Avoidance Point