A Decentralized Approach to Space Deconfliction

Paul Scerri, Sean Owens, Bin Yu, and Katia Sycara
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{pscerri, owens, byu, katia}@cs.cmu.edu

***Abstract*—This paper presents a decentralized approach to path planning for large numbers of autonomous vehicles in sparse environments. Unlike existing approaches, which are either computationally expensive or communication intensive, the presented approach allows large numbers of vehicles to plan independently with low communication overhead. The key to the algorithm is to observe that, in sparse environments, collisions are exceptional and that most of the time vehicles will simply not hit each other. Hence, it is reasonable to allow vehicles to plan independently and then resolve the small number of conflicts. We operationalize this by having each vehicle send their planned paths to a small number of their team mates via *tokens*. Each team member is required to check for conflicting paths that they have been informed about via a token and inform those involved when any conflict is detected. Both analytic and empirical results show that the approach has very high probability of detecting all potential collisions for large numbers of vehicles in both 2D and 3D environments.**

I. Introduction

In emerging applications, large numbers of cooperative autonomous vehicles, such as autonomous ground vehicles [1] or unmanned aerial vehicles (UAVs) [2], will be required to share the same physical space. Of particular interest are domains where many sensor assets are simultaneously working in a shared environment. A critical challenge in such applications is to plan paths that ensure collisions do not occur as vehicles move around. While there are many applications with different specific requirements, this paper focuses on ones where there are up to 100s of vehicles performing time-critical missions, e.g., disaster response [3], [4] and battlefield operations [5], [6], in a relatively sparse 2D or 3D environment. Notice that fast moving vehicles typically do not have sensors on board to detect collisions sufficiently far in advance to safely avoid the collision. This is particularly the case for small and medium sized UAVs which have no ability to detect other similarly sized UAVs. Without such sensors, the vehicles must proactively cooperate to avoid collisions.

Recently, a number of researchers have looked at real-time cooperative path planning for multiple fast-moving vehicles such as UAVs, but typically in a centralized and computationally expensive manner [5], [7], [8], [9], [10]. Most previous work casts the problem as a centralized optimization problem and many solve it using linear programming techniques. A notable exception is [7], however, that approach does not consider the communication overhead of the broadcast for sharing the path information among vehicles and is impractical for large teams. Hence, despite much research into path planning, there are not readily available solutions for environments where many vehicles must share an environment.

The main reason why most previous approaches are inefficient is they attempt to ensure that the planner has complete knowledge prior to planning, so that it will never plan a path that will conflict with another vehicle. However, in sparse environments, we observe that collisions between vehicles will be the exception rather than the rule. That is, even if vehicles moved without consideration of one another, extended periods might pass without a collision occurring. Thus, we hypothesize that it will be more efficient to plan paths independently, then resolve conflicts, rather than to gather information about all vehicle paths before planning. Such an approach can allow distributed planning but still ensure collision free paths with low communication overhead.

Motivated by recent work on *token* passing algorithms [11], [12], we have developed a token-based approach to detecting path conflicts in a distributed manner. Specifically, when a vehicle plans a path, it creates a *token* encapsulating its path and sends it to one of its team mates. That team mate checks the path to determine whether it conflicts with any other paths that it knows about and, if not, passes the token to another team mate. This process continues until either a conflict is found or the token has been passed to a fixed, small number of team mates. If a conflict is found, the vehicles with the conflicting paths are notified and generate new, non-conflicting paths. Vehicles plan slightly in advance of when their plans are needed, to allow time for conflicts to be detected and resolved.

In this paper we present an efficient conflict free path planning algorithm for large numbers of autonomous vehicles with various kinematic constraints in both 2D and 3D environments. Additionally, we provide a quantitative study of token movements for collision detection applying random walk theory to the communication graphs. We demonstrate that, for a given team, a lower bound on the number of agent a token must visit can be determined, so that tokens containing conflicting paths will meet at some vehicle with very high probability. We validate the proposed algorithm with two different path planners and both 2D and 3D simulation environments containing both stationary and dynamic obstacles. Both analytic and empirical results show that our approach has very high probability of detecting all potential collisions for large numbers of vehicles.

II. Problem

In the following, we formally describe the cooperative path planning problem. Vehicles $V = \{v_1, \ldots, v_n\}$ are able to move around some 2D or 3D environment. The communication network connecting the team is a connected undirected graph $G = (V, E)$, where E is the set of links between vehicles.[1] For $v_i, v_j \in V$, $\langle v_i, v_j \rangle \in E$ denotes that vehicles v_i and v_j are neighbors and are able to exchange tokens directly. The environment contains stationary obstacles, $O = \{o_1, \ldots, o_n\}$. A predicate $intersects(\langle x, y, z \rangle, o_i) \rightarrow \{true, false\}$ is $true$ if and only if the stationary obstacle o_i does not allow the vehicle to be at location $\langle x, y, z \rangle$.

We define $\mathcal{L}(v_i, t_k) = (\langle x, y, z \rangle, t_k)$ as the location of the vehicle v_i at time t_k. The path planning algorithm must find a continuous path, $\Gamma(v_i) = \langle \mathcal{L}(v_i, t_0), \mathcal{L}(v_i, t_1), \ldots \mathcal{L}(v_i, t_g) \rangle$ satisfying kinematic constraints on the vehicle and ending at the vehicle's destination. $\mathcal{L}(v_i, t_0) = s_i$ is the start location of the vehicle and $\mathcal{L}(v_i, t_g) = g_i$ is the goal location of the vehicle. Paths must not intersect with stationary obstacles, i.e., $\forall t, \forall o \in O, \neg intersects(\mathcal{L}(v_i, t), o)$. $Cost(\Gamma(v_i)) \rightarrow \mathcal{R}$ is a function of the energy exerted traversing the path $\Gamma(v_i)$ and will differ from vehicle to vehicle. For example, tight turns by a UAV consume more energy than broader ones.

Physical vehicles will typically not be able to exactly follow a planned path, hence the paths need to ensure some miniumum safe distance, $MinDist$, apart to ensure no collisions. If two planned paths never take vehicles within $MinDist$ of one another, we say they are *conflict free*. Formally, paths are conflict free if and only if

$$\forall t, \forall v_i, v_j \in V, i \neq j, Dist(\mathcal{L}(v_i, t), \mathcal{L}(v_j, t)) > MinDist$$

where $Dist(\mathcal{L}(v_i, t), \mathcal{L}(v_j, t))$ is the Euclidean distance between two locations.

In a sparse environment, if each vehicle plans an optimal path from its start location to its goal location not taking into account any other vehicles, there is a good probability that the paths will be conflict free. Baseline experiments in Section V show this to be the case.

The optimization problem is to have the vehicles arrive as quickly as possible to their goal locations, avoiding collisions and minimizing energy use (we assume each vehicle has enough fuel). Thus, we can write the optimization problem as

$$\text{minimize } \alpha \max_{v_i \in V} t_g + \beta \sum_{v_i \in V} Cost(\Gamma(v_i))$$

$$\text{s.t. } \forall t, \forall v_i, v_j \in V, i \neq j, Dist(\mathcal{L}(v_i, t), \mathcal{L}(v_j, t)) > MinDist$$

and

$$\forall t, \forall o \in O, \neg intersects(\mathcal{L}(v_i, t), o)$$

where α and β are constants. Notice that this optimization function minimizes the time that the last vehicle arrives at its goal, but other functions, e.g., the average time the vehicles reach their goals, would also be possible.

[1] An alternative would be to assume a broadcast network, which is used by some types of autonomous vehicles. We believe that using a point-to-point model is more general, since a logical point-to-point network can be created even if physical communication is done by broadcast. Moreover, in the next section, we briefly describe how the algorithm would be adapted for broadcast communications.

III. Cooperative Path Planning

The approach presented in this paper, relies on two key components. First, a path planner is used to determine a path, given known dynamic and stationary obstacles. Second, a token-based algorithm is used to detect conflicts between planned paths. Both of these algorithms run at each vehicle and run asynchronously to one another and their counterparts at other vehicles. Two different planners are described, an A* planner for optimal planning in smaller environments and an RRT planner for heuristic planning in larger environments. The intention of illustrating the algorithm with two different path planners for different types of domain is to emphasize that the basic algorithm is independent of of the planner.

A. Cooperative Deconfliction Algorithm

Algorithm 1 has two basic tasks: (1) planning paths for the vehicle (lines 4-13), and (2) ensuring the whole team avoids conflicts (lines 14-35). The vehicle starts with accurate knowledge of the stationary obstacles in the environment[2], $SObs$, and has an initial path planned, $path$ (which is assumed to be conflict free), but no knowledge of the paths of other vehicles (line 1). If the vehicle has almost completed traversing its current planned path (line 5), it invokes the planner to plan a new path (line 7). Planning slightly in advance gives the team some time to detect and resolve conflicts before the vehicle starts traversing the path. The new path is passes to a team mate, encapsulated in a token (lines 8-10). If the current path is complete, the vehicle switches to its next planned path, which was planned earlier and should have been vetted of conflicts.

Next the vehicle checks to see whether it has new messages. If it has received a token, it updates its knowledge of dynamic obstacles (i.e., other paths) (line 16), updates its token model (line 17) then checks for conflicts with any other plans it knows about (line 19) and reports any conflicts that it finds (lines 20-21).

If the incoming message informed the vehicle that it has planned a conflicting path (line 26), a negotiation with the other involved vehicle determines which vehicle should change paths. We currently use a simple heuristic to determine who changes path, specifically, the vehicle with the lexiographically earliest name changes path, but more sophisticated policies are envisioned. If the vehicle changes its own path, it sends out a *Revoke* message revoking its originally planned path (line 30) and informing the team of its newly planned path (lines 32-34). On receipt of a Revoke message, the vehicle adjusts its knowledge base to reflect the fact that no vehicle is using that path (line 36) and, if required, forwards the message on (lines 37-41).

[2] This is not strictly required, but makes the description of the algorithm clearer

Algorithm 1: The overall path planning algorithm

AGENT($SObs, plan$)
(1) $DObs \leftarrow []$
(2) $tokenM.init()$
(3) $nextPlan \leftarrow NULL$
(4) **while** true
(5) **if** $plan.endTime > t + PLAN\_AHEAD$
(6) $g \leftarrow$ NEXTGOAL()
(7) $nextPlan \leftarrow$ PLANPATH(SObs, Dobs, g)
(8) $token \leftarrow$ MAKETOKEN(token)
(9) $dest \leftarrow tokenM.dest(token)$
(10) SEND(token, dest)
(11) **else if** $plan.endTime \geq t$
(12) $plan \leftarrow nextPlan$
(13) $nextPlan \leftarrow NULL$
(14)
(15) $msg \leftarrow$ RECV_NO_BLOCK()
(16) **if** msg is $Token$
(17) $DObs.update(msg)$
(18) $tokenM.recv(msg)$
(19) $conflict \leftarrow tokenM.getConflicts()$
(20) **foreach** $c \in conflict$
(21) SEND(c)
(22) $msg.TTL \leftarrow msg.TTL - 1$
(23) **if** $msg.TTL \geq 0$
(24) $dest \leftarrow tokenM.dest(token)$
(25) SEND(token, dest)
(26) **else if** msg is $Conflict$
(27) $replan? \leftarrow$ NEGOTIATE()
(28) $tokenM.adjust(msg)$
(29) $DObs.adjust(msg)$
(30) **if** $replan?$
(31) SEND(MAKEREVOKE(nextPlan))
(32) $nextPlan \leftarrow$ PLANPATH(SObs, Dobs, g)
(33) $token \leftarrow$ MAKETOKEN(token)
(34) $dest \leftarrow tokenM.dest(token)$
(35) SEND(token, dest)
(36) **else if** msg is $Revoke$
(37) $DObs.remove(msg)$
(38) $msg.TTL \leftarrow msg.TTL - 1$
(39) **if** $msg.TTL \geq 0$
(40) $dest \leftarrow tokenM.dest(token)$
(41) SEND(token, dest)

Notice that this algorithm translates easily to a situation where communication is performed by broadcast instead of point-to-point. Specifically, rather than creating a token with the planned path (lines 8-10), it simply broadcasts that plan to whoever is within hearing range. Those hearing the broadcast message should check for conflicts with other paths they know about and broadcast the detection of a conflict if one is found.

1) Optimal Path Planning for Small Environments: To plan optimal paths in an environment containing both stationary and dynamic obstacles, A* search is used [13].[3] Typically, optimal algorithms like A* will only be appropriate for vehicles with relatively simple kinematic constraints operating in 2D environments. Following [14], an efficient representation of the space for path planning, called *a quad-tree*, is adopted. The space is divided into a quad-tree (or its 3D-equivalent), with the division of area stopping either when a *quad* contains either no obstacles or is of some minimum size. In the case where a quad contains another vehicle, it is marked with the expected entry and exit times of the other vehicle allowing paths to be planned to pass through that area before or after the vehicle. Figure 1 shows a 2D quad tree overlayed on several stationary obstacles (solid squares) and vehicle paths (lines with starting points marked with a filled circle.)

Because vehicles typically have kinematic constraints it will not always be possible for a vehicle to move from one quad to an adjacent quad, even if both are free of obstacles. For example, a ground vehicle may require a certain radius to make a turn. To account for this, open states in the A* search are annotated with the current state of the vehicle, and new states are only opened if a feasible kinematic path can be computed from the current quad to the new adjacent quad.

Additional desirable properties on paths, e.g., long slow turns are preferred over fast tight ones, are built into the cost function for state transitions. Our simulation results show that, partly due to the relative sparsity of the environment, low cost paths are quickly generated with this approach.

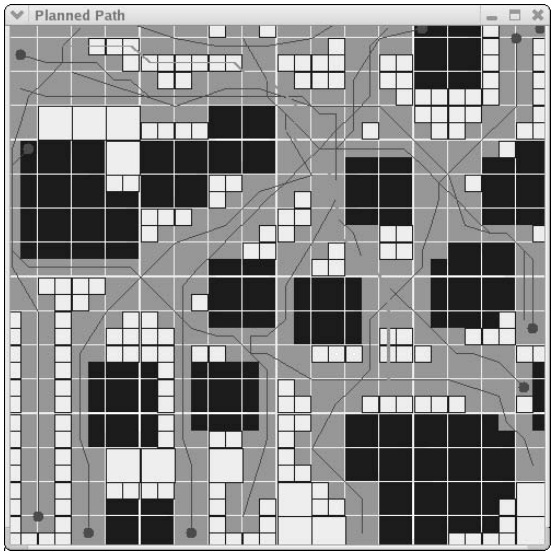

Figure 1. Example quad-tree and paths in a 2D environment, where solid squares are stationary obstacles, lightly shaded squares have dynamic obstacle, and lines are vehicle paths.

A* search proceeds by opening up new states along the most promising paths to the goal. Since some of the quad-tree cells are annotated with times at which they may not be entered (i.e., when there will be another vehicle there), it is important for the vehicle performing the planning to know at which time it will enter that cell. Moreover, because the vehicles have kinematic constraints, it may not be possible for any spatially adjacent cells to be reached by the vehicle. Hence, the current kinematic state of the vehicle and the current time must be stored for open states in the search.

A function $K : X \times Y \times State \times c \rightarrow [0, \infty)$ exists to evaluate the cost of moving from one state to another. If $K(\circ) = \infty$ then it is not kinematically feasible for the

[3]Successors to A* such as D* or AD*[10], could just as easily be used.

vehicle to move from its current state to the cell c withou passing through some other cell. For example, it may n be possible for a vehicle to make a tight turn when movir quickly. Notice that for some vehicles, computing this functic may be computationally expensive, but for the purposes c this algorithm, a reasonable, pessimistic approximation wi suffice.

Algorithm 2 shows the details of how next states ar generated for the A* search. $state$ is whatever vehicle specif parameters about the current state of the vehicle need to b known to calculate $K(\circ)$. The function GETNEXT in Algo rithm 3 is used to determine which cell to consider dependir on current state (line 4-15). The function SHARESFACE retur true if and only if two cells share a side (in 2D) or a face (i 3D) (line 3).

Algorithm 2: Next state generation

GETNEXT($x, y, QuadTree, time, state$)
(1) $returnList \leftarrow []$
(2) $startC \leftarrow QuadTree.cellFor(x, y)$
(3) $possible \leftarrow \{c \in QuadTree, c.children == [] \wedge$ SHARES-FACE(c, startC) $\}$
(4) **foreach** $c \in possible$
(5) **if** K(x, y, state, c) is less than a threshold
(6) **if** $\neg c.StationaryObs$
(7) **if** $cell.dynamicObs == []$
(8) $returnList.append(cell)$
(9) **else**
(10) $ok \leftarrow true$
(11) **foreach** $< t_s, t_e > \in cell.dynamicObs$
(12) **if** $t_s < time - startC.diagSize \wedge t_e > time + startC.diagSize$
(13) $ok \leftarrow false$
(14) **if** ok
(15) $returnList.append(cell)$

COMPTIMES determines when known paths will be entering and leaving a cell, leaving a small amount of slack to handle minor uncertainty (Handling a large degree of uncertainty in execution of a path is an area for future work.). The function DIVIDE breaks a cell into four, if 2D, or eight if 3D pieces. In this paper, our function simply limits the maximum turn and climb rates of the vehicle, but more sophisticated algorithms could be used if vehicles have more complex constraints.

Example: Figure 2 shows the expansion of one state in the A* search in a 2D environment. The current open state is the large cell in the middle of the figure, with the path to that point shown by the double line. A stationary obstacle prevents the vehicle from going NorthWest. The path of v_2 prevents the vehicle from going EastNorthEast, but going NorthNorthEast is OK because v_2 will have already passed that point. Kinematic constraints prevent the vehicle for going back to the west. Hence, in this example, the A* search will open up three new states from exploration. These paths are shown with dotted lines and marked OK. In an environment with up to 200 by 200 cells, 50 stationary obstacles and 100 dynamic obstacles, the A* algorithm finds paths, if they exist, in on the order of 1s.

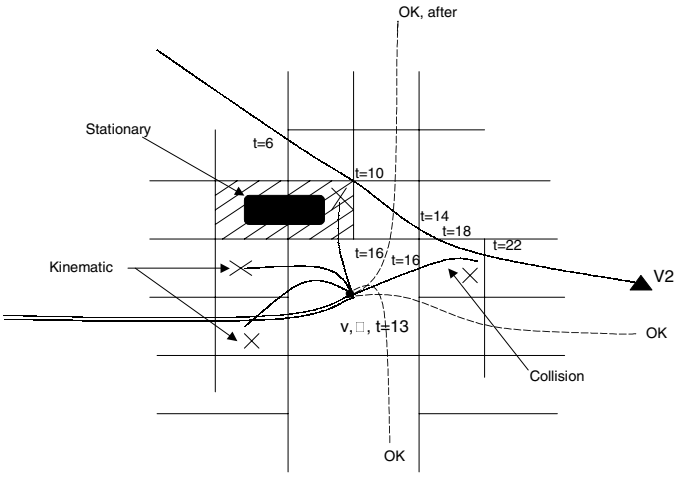

Figure 2. An example of evaluating new states for a vehicle entering from the left.

2) RRT Planners for High Dimensionality Environments: When path-planning in high dimensional environments, e.g., for UAVs, optimal plan planning algorithms such as A* can be infeasibly computationally expensive. Hence, for large environments a heuristic planner is required. Specifically, we have experimented with a Rapidly-Expanding Random Tree (RRT) planner[15], which has been shown to be a fast effective way of planning in such dynamic environments with both stationary and dynamic obstacles. RRT planners use a map of the costs in the environment to determine the cost of small segments of path. In this case, the paths of other UAVs are stored precisely as they are received in this cost map. Unlike A* search, which relies on states, RRT planners use an underlying continuous distribution and hence costs are represented in a continuous way. Notice, that the algorithm below works slightly differently to a normal RRT planner, specifically differing in the way that new nodes are added to the search tree.

Algorithm 3 shows the modified RRT planning process. Input to the algorithm includes a cost map encoding the goals of the vehicle and another cost map with the known paths of other vehicles. Lines 1-5 initialize the algorithm, creating a priority queue ($plist$) and initial node (n). The ordering of the priority queue is very important for the functioning of the algorithm, since the highest priority node will be expanded. The function COMPUTEPRIORITY uses both the cost of the node and the number of times it has been expanded to determine a priority. Intuitively, the algorithm works best if good nodes that have not been expanded too many times previously are expanded. The main search loop is lines 6-17 and is repeated 20,000 times (about 500ms on a standard desktop computer.) The highest priority node is taken off the queue (then added again with new priority). This node, representing the most promising path, is expanded 10 times in the inner loop, lines 10-17. The expansion creates a new node, representing the next point on a path, extending the previous best path by a small amount. The EXPAND function

is designed so that all new nodes lead to kinematically feasible paths. The function COMPUTECOST then determines the cost for the new search node, taking into account the cost of the node it succeeds and the *cost maps*. The cost map representing other paths will return $+\infty$ if the new node leads to a path segment that would lead to a collision. The expanded nodes are added to the priority list for possible future expansion and the process continues. Finally, the node with the lowest cost is returned. The best path is found by iterating back over the $prev$ pointers from the best node.

Algorithm 3: RRT Planning Process
RRTPLANNER($x, y, CostMaps, time, state$)
(1) $plist \leftarrow []$
(2) $n \leftarrow \langle x, y, t, cost = 0, prev = \emptyset, priority = 0\rangle$
(3) $n \leftarrow$ COMPUTEPRIORITY(n)
(4) $plist.insert(n)$
(5) $best = n$
(6) **foreach** 20000
(7) $n \leftarrow plist.removeFirst()$
(8) $n.priority \leftarrow$ COMPUTEPRIORITY(n)
(9) $plist.insert(n)$
(10) **foreach** 10
(11) $n' \leftarrow$ EXPAND(n)
(12) $n'.prev = n$
(13) $n'.cost =$ COST($n, CostMaps$)
(14) $n'.priority \leftarrow$ COMPUTEPRIORITY(n')
(15) $plist.insert(n')$
(16) **if** $n'.cost < best.cost$
(17) $best \leftarrow n$
(18) Return $best$

IV. BOUNDING TOKEN MOVEMENT

The communication efficiency of the algorithm is derived from the fact that the token does not need to visit all vehicles in order to detect conflicts. However, visiting too few can lead to potential conflicts escaping detection. In this section, a qualitative analysis of how far a token should optimally move, its TTL (time to live), is presented. Specifically, the theory of random walks is applied to determine a lower bound on the TTL.

Given a graph G with n nodes and m edges, a *random walk* starts at some node v_i of G, and at each step moves to one of the neighbors of the current node. For example, if the random walk is at a node v_j, it moves to a neighbor of v_j with probability $1/d(v_j)$, where $d(v_j)$ is the number of neighbors of v_j in G.

The probability that at step h the token is at v_i can be denoted as $P_h(v_i)$. The theory of random walks [16], says that if a walk starts from any node in an undirected connected graph G, $P_h(v_i)$ converges to $\pi(v_i) = d(v_i)/2m$, where $\pi(v_i)$ represents the probability a token will be at node v_i at any particular time.

Two measures of a random walk are helpful for the analysis: *hitting time* and *commute time*. The hitting time $H(v_i, v_j)$ is the expected number of steps before node v_j is visited by a token that starts from node v_i. The sum $\kappa(v_i, v_j) = H(v_i, v_j) + H(v_j, v_i)$ is called the commute time, which is the expected number of steps in a random walk starting at v_i, before node v_j is visited and then node v_i is reached again.

Proposition 1 The commute time of a random walk starting at v_i visits v_j before returning to v_i can be estimated as follows

$$\kappa(v_i, v_j) \geq 2m/d(v_i)$$

Proof: Let τ be the first time when a random walk starting at v_i returns to v_i, and σ the first time when it returns to v_i after visiting v_j. We can determine $E(\tau) = 2m/d(v_i)$ from the probability $\pi(v_i) = d(v_i)/2m$. By definition $E(\sigma) = \kappa(v_i, v_j)$. Clearly we have $\tau \leq \sigma$. Hence the proposition $\kappa(v_i, v_j) \geq 2m/d(v_i)$ holds.

The idea is that if we start a random walk from v_j, the random walk will choose a neighbor of v_j uniformly and then visit other nodes before it returns to v_j. From Proposition 1, we can find that, for a given undirected connected graph G, if we choose v_i uniformly from the set of neighbors of v_j, then the expectation of $H(v_i, v_j)$ is exactly one step less than the commute time from v_j to v_i. We know that the hitting time $H(v_j, v_i) = 1$ and the commute time $\kappa(v_j, v_i) \geq 2m/d(v_i)$. Therefore,

$$H(v_i, v_j) = \kappa(v_j, v_i) - 1 \geq 2m/d(v_j) - 1$$

The following proposition holds when we choose v_i uniformly from the stationary distribution over V and $v_j \neq v_i$. Note that for a node v_j and a uniformly chosen node v_i, the hitting time $H(v_i, v_j)$ is minimal when v_j is a neighbor of v_i. According to Proposition 1, we have $H(v_i, v_j) \geq \kappa(v_j, v_i) - 1$ and $\kappa(v_i, v_j) \geq 2m/d(v_i)$. Hence, the following proposition holds.

Proposition 2 If we choose v_i uniformly from the stationary distribution over V and $v_j \neq v_i$, then we have $H(v_i, v_j) \geq (2m/d(v_j)) - 1$.

In order to detect a collision, the second token needs to visit any node visited by the first token. Assume that the first token moves TTL steps in graph G. The first token generates a random spanning tree in a subgraph of graph G. Formally, we can take the subgraph G' covered by the first token and represent it as a single node v_j in the new graph G''. Specifically, we assume (1) G' has c_1 edges; (2) the single node v_j has c_2 neighbors in G''. Then the problem of determining an optimal TTL reduces to determining the hitting time of the second token to this new node v_j in graph G'', which has $m - c_1$ edges. The degree of v_j in G'' is c_2.

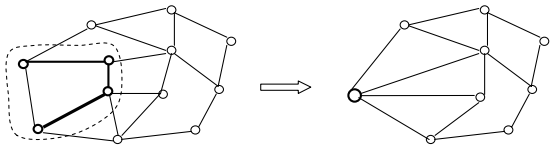

Figure 3. A subgraph of G, G', which covers the random spanning tree traversed by the first token

According to Proposition 2, we have

$$TTL \geq H(v_i, v_j) \geq \frac{2(m - c1)}{c_2} - 1$$

Thus, a lower bound on the TTL is $TTL = \frac{2(m-c1)}{c_2} - 1$. In order to compute this value, we have to know the average number of distinct nodes N visited by the first token (N is also the size of the subgraph G').

This average number of distinct nodes N was first studied by Dvoretzky and Erdos[17]. They found that number $N \sim \frac{TTL}{lnTTL}$ for a connected undirected graph when TTL is relatively small compared to N. The relationship between the average number of distinct nodes N and its relationship to $\frac{TTL}{lnTTL}$ for a connected undirected random graph with 100 nodes can be represented as $N \approx \alpha \frac{TTL}{lnTTL}$ for small TTLs, where $1.91 \leq \alpha \leq 2.22$, and $10 \leq TTL \leq 20$. For simplicity, we choose $\alpha = 2$ in this paper.

Now we can estimate c_1 and c_2 for the subgraph G', where we have $|G'| = N = \frac{2TTL}{lnTTL}$. Erdos and Renyi showed that in general the number of leaves in a random spanning tree on N nodes approaches the normal distribution $\mathcal{N}(N/e; N(e-2)/e^2)$ [18]. In particular, the expected number of leaves, the mean of this distribution, approaches N/e.

Hence, we can estimate c_1 and c_2 as follows.

$$c_1 \approx N/e(1 + (\bar{d}-1)\frac{N}{n}) + \frac{(N - N/e)}{e}(1 + (\bar{d}-1)\frac{N}{n}) + \dots$$

where $\bar{d}$ is the average degree of graph G. The item $N/e(1+(\bar{d}-1)\frac{N}{n})$ includes two parts: (1) the edges between leaf nodes and their parents nodes; (2) the edges between leaf nodes and other nodes in the subgraph G'. $(N-N/e)$ denotes the number of nodes in G' after removing N/e leaf nodes.

Similarly,

$$c_2 \approx N/e(\bar{d}-1)\frac{n-N}{n}) + \frac{N - N/e}{e}(\bar{d}-2)\frac{n-N}{n}) + \dots$$

As an example, let's estimate the lower bound of TTLs when $n = 50$, $m = 100$, $\bar{d} = 4$. When $TTL = 12$, we have $N \approx 9.6$, $c_1 \approx 10$, and $c_2 \approx 12$

$$TTL < 2 * 90/12 - 1$$

When $TTL = 13$, we have $N \approx 10$, $c_1 \approx 12$, and $c_2 \approx 14$

$$TTL > 2 * 88/14 - 1$$

Therefore we choose $TTL = 13$ for the given graph.

V. Experiments

To evaluate this approach, we used two simulators. The first was a 2D simulator appropriate for the A* planner and capable of running much, much faster than real-time, hence allowing many experiments to be performed. The second was a fully distributed UAV simulation environment, appropriate for the RRT planner, but much slower than the 2D planner, making large numbers of experiments infeasible. The UAV simulation environment has the additional complicating factor that in 3D space collisions very rarely occur, so many hours of simulated flight time was required. We partially compensated for this, by having a definition of "collision" where UAVs came within some relatively large distance of each other. Notice that the smaller set of results for the 3D environment largely matches the 2D results and we expect that all 2D results can be extrapolated to the 3D environment.

A. A* Planner

The 2D environment size was fixed to be 2000 by 2000 units, with cells limited to be no smaller than 15 units wide. All vehicles move at 1 unit per step. 50 stationary obstacles, 50 by 50 units in size were randomly placed in the environment potentially overlapping. Vehicles were considered to have crashed *at the time* they committed to a path that would bring them within 5 units of another vehicle's path (Algorithm 1, line 26).

This metric for collisions, specifically counting it from the time at which the vehicle started moving along the path is very conservative but gives a good measure of how well the cooperation worked at preventing collisions. New destinations for vehicles were generated when they were ten units from their current goal. New goals were randomly selected, but they were constrained to be between 100 and 400 units from the current goal. Unless otherwise stated, each token had a TTL of 20 and moved to 10 vehicles in the time a vehicle moved one unit. Each point in the experiments was averaged over 20 runs.

In the first experiment, we tested the basic functionality of the algorithm by testing four different configurations. First, we used a centralized path planner (labeled as CENTRAL) to provide a baseline. Second, we used the algorithm described above (labeled as TOKEN). Third, we used the token algorithm to send tokens with planned paths but did not report conflicts (labeled as PASSIVE). This algorithm allowed us to distinguish between the effect of sending out paths and the effect of having team mates report conflicts. Finally, as a lower baseline we used no coordination and allowed the vehicles to plan their paths independently (labeled as NONE). Vehicles were allowed to travel until one committed to a path that would cause a collision with another planned path or until 5,000 time steps had passed (enough to go back and forth across the environment 2.5 times).

We measured three values: (1) the time before the first vehicle committed to a path that would lead to a collision; (2) the number of new goals that the vehicles accepted which is an indirect measure of efficiency and; (3) the number of times a vehicle had to replan due to a potential collision to gauge the amount of work being done by TOKEN. Each data point represents the average over 20 runs.

Figure 4 shows that both the TOKEN and CENTRAL approaches reliably prevent all collisions when there are up to 60 vehicles. When there are more vehicles the TOKEN approach will begin to have some collisions, however notice that the TTL is kept fixed at 20 and that raising the TTL leads to the TOKEN approach avoiding collisions for higher numbers of vehicles. Surprisingly, except for very small numbers of vehicles, where acts like the CENTRAL approach, the PASSIVE approach is barely better than NONE. This shows that the key to TOKEN is that team mates inform one another of potential collisions, rather than having to know of all potentially conflicting paths when they plan. Figure 5 shows that CENTRAL is slightly more efficient, because the

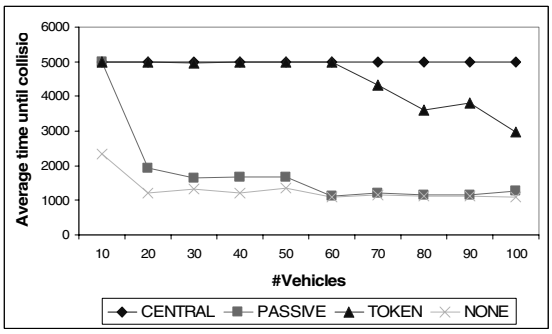

Figure 4. The average time until a vehicle commits to a conflicting path, with a maximum of 5000.

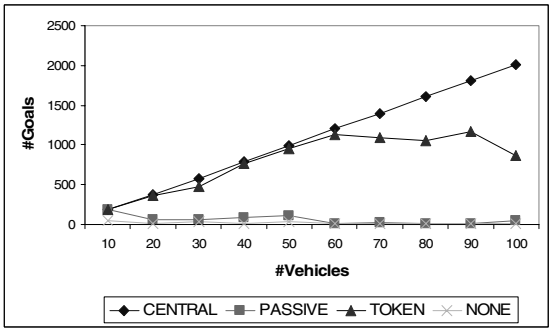

Figure 5. The number of goals visited by all vehicles within 5000 time steps

TOKEN algorithm occasionally plans around paths that have been revoked, but that information has not been propagated to the vehicle doing the planning.

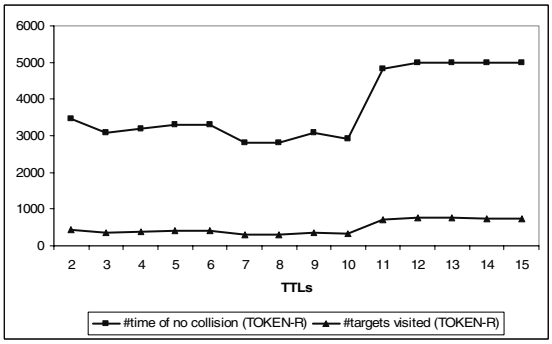

Figure 6. The time of no collision and the number of targets being visited for different TTLs

In the second experiment for A* planner the number of vehicles was held fixed at 40 and the tokens were routed randomly. Figure 6 shows the average time until a collision happens and the number of goals completed for different TTLs. Note that there is a distinct shift to reliable collision avoidance for a TTL of 11. This corresponds exactly to the predicted lower bound computed via the analysis described in the previous section.

B. RRT Planner

The deconfliction algorithm using the RRT planner was implemented within the Machinetta proxy[19] framework and integrated with the Sanjaya UAV simulation environemnt[20]. Figure 7 shows a 2D screenshot of the simulator, with 50 UAVs, illustrating the complexity of the deconfliction problem. The code is used is exactly the same code as being used in an ongoing flight test, with the exception of the code between the proxy and the autopilot. The simulated environment is 50km by 50km and the results below represent approximately 150 hours of simulated flying time. The UAVs move at approximately 25km/h and can change altitude at 1m/s. The UAVs are planning paths in 15km segments. A collision is recorded if vehicles come within 1km horizontally and 150m vertically. For each number of UAVs, TTL is set to three times the log of the number of UAVs. Notice that for larger numbers of UAVs this makes the space very highly constrained, as shown in Figure 7.

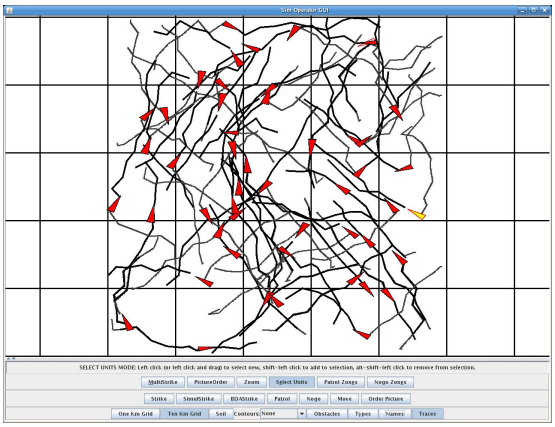

Figure 7. A screenshot of the UAV simulation environment showing planned paths (black lines) of UAVs (triangles).

Notice that there were several engineering issues that make the UAV case difficult. Most importantly, the UAVs, under the control of an autopilot, do not always follow the planned path precisely, deviating either spatially or temporally. This led to a need to require UAVs try to avoid each other by a long distance, making the planning more difficult. Since the simulation was fully distributed over 13 machines, communication latencies and path planning times required careful attention. Figure 8 summarizes the results, with each bar in the graph representing an average of 10 runs, with deconfliction "on" (ACTIVE) or "off" (NONE). Notice that while having deconfliction on dramatically improves the performance, the deconfliction algorithms do not remove all conflicts. We believe that at least some of these conflicts are due to unresolved engineering issues, not algorithmic issues.

VI. Related Work

Cooperative path planning for multiple UAVs was recently studied by [21], [5], [7], [9]. Most of the existing approaches consider the problem as a centralized optimization problem

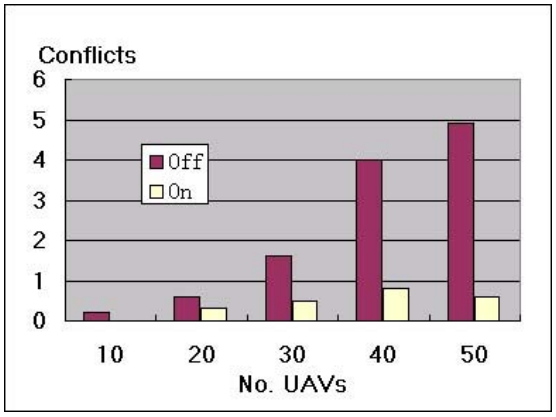

Figure 8. Results with RRT planner in 3D simulation environment.

and solve it using linear programming techniques. These centralized approaches only work for a small number of UAVs and tasks and cannot meet the real-time requirements for large-scale UAVs teams due to either communication or computation bottlenecks. The only exceptions are [7] and [22], where Lechliter presents a distributed task allocation framework for UAVs, and Alami el al. give a framework for coordinating multiple robots. However, both approaches do not consider the communication overheads for sharing the information about dynamic obstacles. They simply assume all vehicles know what every other vehicles is doing. This is impractical for large-scale teams, where communication bandwidth is rare in a real-time environment. We develop a distributed path planning algorithm for a UAVs team under communication constraints. Our approach reduces the amount of information shared for coordination using tokens and enables each UAV to produce acceptable and collision-free paths in nearly real-time.

VII. Conclusions

This paper presents a decentralized approach to conflict-free path planning with low communication overhead. The algorithm relies on vehicles passing their planned paths to a small number of team mates who proactively check for conflicts in paths they know about. Both analytic and empirical results show that each token needs to visit only a relatively small proportion of the team to have very high probability of detecting all potential collisions. We find that for sparse environments the token-based approach is as reliable and efficient at avoiding collisions as a centralized approach. Even when the environment is crowded with vehicles and obstacles the token-based approach is almost as effective as a centralized approach. Two separate planners, one optimal and one heuristic, were used to illustrate the independence of the basic algorithm from the actual planner.

However, while this paper presents a significant advance, there are a range of issues that require further work. One key area for future work is whether there are ways of intelligently routing the tokens to increase the probability of detecting conflicts, without increasing the TTL. For example, to determine where to send a particular token, each vehicle might guess which of its neighbors is most likely to know about conflicts with a particular path.

References

[1] C. L. Ortiz, R. Vincent, and B. Morisset, "Task inference and distributed task management in centibots robotic systems," in *AAMAS*, 2005.

[2] P. Scerri, Y. Xu, E. Liao, J. Lai, and K. Sycara, "Scaling teamwork to very large teams," in *Proceedings of AAMAS'04*, 2004.

[3] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjoh, and S. Shimada, "Robocup rescue: Searh and rescue in large-scale disasters as a domain for autonomous agents research," in *Proc. 1999 IEEE Intl. Conf. on Systems, Man and Cybernetics*, vol. VI, 1999.

[4] R. Nair, T. Ito, M. Tambe, and S. Marsella, "Task allocation in robocup rescue simulation domain," in *Proc. of the International Symposium on RoboCup*, 2002.

[5] J. Bellingham, M. Tillerson, M. Alighanbari, and J. P. How, "Cooperative path planning for multiple uavs in dynamic and uncertain environments," in *Proc. of the 41st IEEE Conference on Decision and Control*, 2002.

[6] A. Vick, R. M. Moore, B. R. Pirnie, and J. Stillion, *Aerospace Operations Against Elusive Ground Targets*. RAND Documents, 2001.

[7] M. C. Lechliter, "Decentralized control for uav path planning and task allocation," Ph.D. dissertation, Department of Mechanical and Aerospace Engineering, West Virginia University, 2004.

[8] D. Silver, "Cooperative pathfinding," in *Proceedings of the First Conference on AI and Interactive Digital Entertainment*, 2005.

[9] R. Vincent, O. Yagdar, and A. Agno, "UAV airspace management system UAMS," in *AAMAS*, 2006.

[10] D. Ferguson, M. Likhachev, and A. Stentz, "A guide to heuristic-based path planning," in *Proceedings of the International Workshop on Planning under Uncertainty for Autonomous Systems*, 2005.

[11] P. Scerri, Y. Xu, E. Liao, J. Lai, and K. Sycara, "Scaling teamwork to very large teams," in *Proceedings of Third International Joint Conference on Autonomous Agents and Multiagent Systems*, 2004, pp. 888–895.

[12] Y. Xu, P. Scerri, B. Yu, S. Okamoto, K. Sycara, and M. Lewis, "An integrated token-based algorithm for scalable coordination," in *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, 2005.

[13] N. Nilsson, *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.

[14] A. Yahja, A. Stentz, S. Singh, and B. L. Brumitt, "Framed-quadtree path planning for mobile robots operating in sparse environments," in *ICRA*, 1998.

[15] S. LaValle and J. Kuffner, "Rapidly-exploring random trees: Progress and prospects," in *Proceedings of the Workshop on Algorithmic Foundations of Robotics*, 2000.

[16] L. Lovasz, "Random walks on graphs: A survey," in *Combinatorics, Paul Erdōs is Eighty*, T. S. D. Miklós, V. T. Sós, Ed. János Bolyai Mathematical Society, 1993.

[17] A. Dvoretzky and P. Erdos, "Some problems on random walk in space," in *Proc. 2nd Berkeley Symposium on Math. Statist. and Prob*, 1950.

[18] P. Erdos and A. Renyi, "On random graphs," *Publ. Math. Debrecen*, 1959.

[19] P. Scerri, D. V. Pynadath, L. Johnson, P. Rosenbloom, N. Schurr, M. Si, and M. Tambe, "A prototype infrastructure for distributed robot-agent-person teams," in *The Second International Joint Conference on Autonomous Agents and Multiagent Systems*, 2003.

[20] S. Owens, P. Scerri, R. Glinton, B. Yu, and K. Sycara, "Synergistic integration of agent technologies for military simulation," in *Demonstration Track at AAMAS'06*, 2006.

[21] J. Bellingham, M. Tillerson, A. Richards, and J. P. How, "Multi-task allocation and path planning for cooperative uavs," in *Proceedings of the 40 IEEE Conference on Decision and Control*, 2001.

[22] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and F. Robert, "Multi robot cooperation in the martha project," *IEEE Robotics and Automation Magazine*, vol. 5, 1998.