

Lightweight simulation of air traffic control using simple temporal networks

Russell Knight

keywords: simple temporal networks, 3d path planning, lightweight simulation

Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive
MS 126-347
Pasadena, CA 91109
russell.knight@jpl.nasa.gov

Abstract

Flight simulators are becoming more sophisticated and realistic, and the requirements of those using them are becoming more demanding. Air traffic control simulation for such simulators is now approaching the real problem of air traffic control, where several aircraft need to be scheduled for safe approach and landing or for safe passage through the air space that an air traffic controller is responsible for. Also, the computational resources available to dedicate to air traffic control simulation are sparse, given that most simulations focus on a realistic graphical user interface and aircraft physics modeling. We provide a formulation of the air traffic control problem and a solver for this problem that makes use of temporal constraint networks and simple geometric reasoning. We provide results showing that this approach is practical for realistic simulated problems.

Introduction

Current flight simulators that run on home computers are sophisticated and realistic. Part of this realism is the interaction with airport traffic control towers. This interaction includes ground traffic control and air traffic control. Automated air traffic control is addressed in detail in [3], but the amount of computation required can be prohibitive for a lightweight application such as a game. We focus on the problem of simulating a realistic air traffic controller without “breaking the bank” with respect to computational resources.

Air traffic control

The job of the air traffic controller (ATC) is to ensure that all aircraft in a controlled air space are safe from collision and afforded fair access to the air space. This requires knowledge about each aircraft in the air space being controlled. This controlled air space is called the *bubble*.

Upon entering the bubble, aircraft pilots announce their intentions to the ATC. Given this information, the ATC provides instructions to all aircraft in the bubble to ensure that 1) no two aircraft ever come within a prescribed distance to each other, 2) all aircraft reach their destination, 3) no aircraft is overly delayed. Also, the ATC tries to keep communications to a minimum.

This requires the ATC to reason about the paths through the air that the aircraft will travel, the capability of each aircraft with respect to minimum and maximum speeds and minimum turning radius, and the timing of aircraft passing through shared air space.

The air traffic control problem

Here we formalize the air traffic control problem (ATCP). The ATCP is, given a safe distance s that all aircraft must maintain from other aircraft, a collection of airports, aircraft and aircraft destination, and a bubble, generate a set of commands to the aircraft that allow each aircraft to land or pass through the bubble without violating any operating constraints.

An example

We now describe a scenario to illustrate the capabilities of our technique.

1. Consider an airport with no aircraft in its bubble. Obviously, no commands need to be sent to any aircraft.
2. Then, an aircraft A enters the bubble and requests/expects landing instructions. Given that this is the first aircraft to land, the ATC opts for a full procedure approach and routes the aircraft to an initial approach fix (IAF, i.e., let the pilot navigate the approach).
3. Then, an aircraft B enters the bubble. B is much faster than A, and the ATC discovers that B can

start its approach before A if a direct path is taken. The ATC chooses this route, and the plan is updated.

4. Then, an aircraft C enters the bubble. If B's approach were modified, then C could land directly, then B, then A, but the ATC does not modify existing routes (while possible, changing existing routes is avoided in practice as it is a technically and theoretically harder problem). Instead, an alternate vectoring is chosen that accommodates the landing of B, then C, then A.
5. B lands.
6. A fast aircraft D enters the bubble, and no set of vectors can accommodate it without violating constraints. D is then put into a holding pattern and is scheduled to land after A.
7. A slows unexpectedly, resulting in instructions to D not to exceed a certain speed.

An airport consists of:

- a 3d coordinate **position** that represents the start of the runway
- a vector **landingDirection** representing the direction in which aircraft land
- a distance **length** representing the length of the runway

Notationally, for an airport p , we would refer to the associated vector as p . **landingDirection**.

An aircraft consists of:

- a coordinate in 3d space **position** that represents the location of the aircraft
- a vector **direction** defining direction of travel
- a minimum rate of travel **minSpeed**
- a maximum rate of travel **maxSpeed**
- a nominal rate of travel **nomSpeed**
- a current rate of travel **speed**
- a maximum climb rate **maxClimb**
- a nominal climb rate **nomClimb**
- a maximum dive rate **maxDive** (represented as a negative climb rate)
- a nominal dive rate **nomDive**
- a current climb/dive rate **zRate**
- a minimum height **minAltitude**
- a maximum height **ceiling**
- a turn-rate function (that returns values the form of radians/second) **maxTurnRate(speed)** that indicates the maximum rate of a turn for a given *speed*
- a turn-rate **nomTurnRate(speed)** that indicates the nominal rate of a turn for a given *speed*
- a set of airports **landingSites** containing the airports that this aircraft could land at
- a coordinate in 3d space **destination** that indicates either the point at which the aircraft wishes to leave the bubble or p .**position** for some $p \in$ **landingSites**.

The bubble consists of:

- a distance **radius** that represents the radius of the hemisphere of space in which the air traffic controller will be responsible for posting commands to the aircraft
- a 3d coordinate **position** that represents the location of the bubble

A solution consists of a set C of commands to the aircraft that allow each aircraft to land or pass through without violating any constraints. A command consists of:

- a time **executeTime** that the command is to be given and executed by the aircraft
- an aircraft **aircraft** that is the recipient of the command
- a turn-rate **turnRate** that represents the rate to turn
- a turn destination vector **goalDirection**
- a climb/dive **zRate**
- a goal altitude **zDestination** that indicates the altitude to climb or descend to
- a goal rate of speed **goalSpeed**

Note that in practice a command is usually agnostic with respect to most parameters, e.g., in the voice interface we would normally only expect to hear commands such as "turn to 215 degrees" or "descend to 1000 meters".

So, an ATCP consists of finding C given $\langle P, A, b, s \rangle$ such that, for all possible times, no aircraft is within the limit s of any other aircraft, no command is issued that violates the operational bounds of any aircraft, and all aircraft reach their destinations (where C is a set of commands, P is a set of airports, A is a set of aircraft, b is the bubble, and s is the minimum safe distance).

But, in reality, we actually find ourselves executing a solution to the ATCP, only to learn that a new, heretofore unknown aircraft has entered the bubble and requires air traffic control. So, the actual solution to the ATCP needs to be incremental in nature—that is, it needs to start with a previous solution and then update that solution to accommodate unforeseen events. We call this is the online ATCP.

We also desire a measurement of fairness. A fair ATCP solution is one that minimizes the delay for the aircraft. The simple metric we use for delay is the mean squared delay, where delay is measured as the difference between the time it takes to arrive at the destination and the time it would take for direct travel to the destination at nominal speed, turn rate, and dive rate.

Finally, we desire ATCP solutions that require minimal "chat" with aircraft. The metric we use is the average number of commands sent to an aircraft.

Preliminaries

Here we provide some background on the basic algorithms and techniques to be used in our solution to the ATCP.

Simple temporal networks

We represent the plan of each aircraft as a series of timepoints in a temporal constraint network [2]. A temporal constraint network is an edge-labeled, directed graph where each node represents a moment in time and each edge represents a temporal relationship between the source node and the sink node of the edge. In a simple temporal network, each edge is labeled with the minimum and maximum difference allowed between the time assigned to the source node and the time assigned to the sink node. It is possible to know that a temporal constraint network is infeasible or that some of the minimum and maximum labels of the edges are more constrained due to other edges [xxx]. From this propagated network we can compute a schedule (i.e., an assignment of a specific time to each moment represented in the temporal constraint network).

More formally, a STN is a 3 tuple $\langle G, l, u \rangle$, where $G = (N, E)$ is a directed graph with edge-label functions $l(e \in E) \rightarrow \mathcal{R}$ (i.e., minimum duration) and $u(e \in E) \rightarrow \mathcal{R}$ (i.e., maximum duration) A *schedule* is a vertex-label function $T(n \in N) \rightarrow \mathcal{R}$ (i.e., the time of each timepoint) where,

$$\begin{aligned} \forall (x, y) \in E, \\ T(y) - T(x) \geq l((x, y)) \wedge \\ T(y) - T(x) \leq u((x, y)) \end{aligned}$$

Note that a relabeling of l and u that takes into consideration all intervals can be computed in either $|N|^3$ or $|N|^2|E|$, which ever is less. It should be noted that one candidate T after this relabeling is the solution indicated by assuming all intervals to be $l(e \in E)$, that is, one possible execution is an execution where we use the shortest durations.

Cylinder intersection

We represent the path of an aircraft through the bubble as a cylinder in 3D space where the radius of the cylinder is the prescribed safety distance. If two of these cylinders intersect for different aircraft, then a proximity violation is possible. We use a simple form of cylinder intersection to compute this intersection. Specifically, we are interested in subdividing each cylinder into smaller sub-cylinders; where either no intersection occurs over the sub-cylinder, or an intersection occurs over the entirety of the sub-cylinder, e.g., see Figure 1.

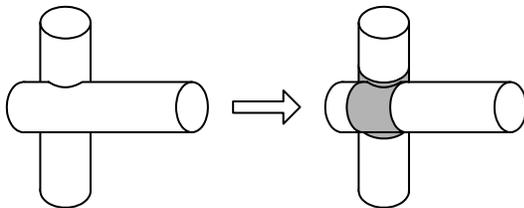


Figure 1 Computing contending cylinders (shaded area)

This is a simple process of finding where the distance between the infinite lines representing the center of each cylinder is exactly the sum of the diameters of the cylinders. If there is only one point where this occurs, then the cylinders touch, but do not intersect. If there are 2 points (for each line), then the cylinders could intersect, and we need to check the actual starting and ending points of the cylinders. Of course, the lines could be parallel, and all points are contained, in which case the linear intersection of the cylinders is computed.

Solving the ATCP

The ATCP demands reasoning in both temporal and spatial realms. First, we make some simplifications that actually add to the fidelity of the simulation. Then, we decompose the problem into aircraft landing problems and 3d path finding problems. We abstract away the spatial geometry by encoding the problem as a simple temporal network (STN). We use the STN to provide us with temporal and spatial violation information. Finally, during execution and monitoring, we use the STN to dictate the command sequence, as well as change the command sequence in the event of off-nominal behavior on the part of aircraft.

Simplifications

To simplify the problem, we will divide the 3d airspace into stacks of 2d layers. This reflects how the space is usually divided up in real air traffic control [4], so this actually adds to the simulation.

A further simplification is the pre-processing of the set of airports that are in the bubble, resulting in prescribed holding spirals and vectors of approach. Again, this reflects real ATC operations. This helps guide our search for a solution, and also affects some of our criteria, because we can give a vector-of-approach command that would be a series of turns and descents that result in a landing.

Algorithms

There are four components to the ATC algorithm. These are the *controller*, the *plan*, the *vector generator*, and the *constraint verifier*. See Figure 2.

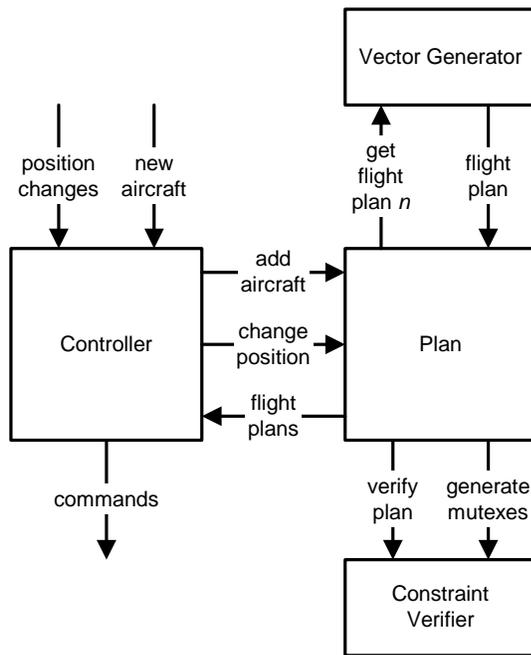


Figure 2 ATC block diagram

The controller

The *controller* is event driven and quite simple. Three events kick off the controller:

1. An aircraft enters the bubble. Try all vector options in a shortest path first order, checking each using the constraint verifier. When a solution is found, announce instructions to the appropriate pilot.
2. An aircraft exits the bubble. Remove the aircraft from the plan.
3. An aircraft changes location as prescribed by the plan. Update the plan with the new location/time of the aircraft. If needed, give instructions to the appropriate pilot.

Thus, the controller is the input/output interface for the ATC algorithm.

The plan

The *plan* is a data structure that is used to verify current and hypothetical routings for each aircraft. It consists of the following:

- a set of aircraft A
- a simple temporal network $STN = \langle G=(N, E), l, u \rangle$
- an indexed set M that maps nodes to sets of nodes
- a labeling function $\text{mutex}(n \in N) \rightarrow N' \subseteq N$, i.e., returns the set of timepoints that must be totally ordered (none can occur simultaneously) as explained in more detail below
- a function $\text{addMutex}(n_1 \in N, n_2 \in N)$ that adds the information that n_1 and n_2 cannot occur simultaneously
- a mapping function $\text{aircraft}(n \in N)$ that returns the aircraft associated with timepoint n

- **addFlightPlan**(G', a) adds the flight plan represented by the directed graph G' for the airplane a to the plan P . It is assumed that the flight plan is a valid path with respect to edge labels and topology. It then introduces ordering constraints at mutex points to avoid collisions. If no ordering is found, it returns false
- **removeAircraft**(a) removes the associated aircraft and any flight plan associated with it
- **validate** returns true if the plan is possible and modifies the edge labels if necessary

An approach that an aircraft follows consists of a directed path in the STN. Since each aircraft gets its own unique path, we need to enforce that two aircraft are not in the same place at the same time. We use **mutex** to determine this. If two timepoints overlap in space, then we must ensure that they do not overlap in time. We do this by introducing temporal constraints. Figure 3 illustrates an example where the nodes n_1, n_2, n_3 , and n_4 are mutex. Note that n_1 and n_2 are ordered already, as are n_3 and n_4 ; the decision to be made is this: does n_1 follow n_4 or does n_3 follow n_2 ? The nodes were determined to be mutex-related by computing the contending cylinders of the flight paths.

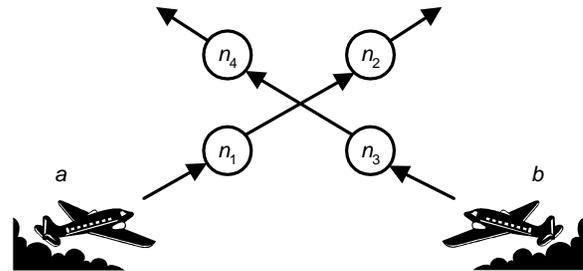


Figure 3 Mutex nodes of aircraft a and b

Operations on the plan include insertion of an aircraft, deletion of an aircraft, and updating an aircraft's position and time. An aircraft is inserted into the plan by inserting its associated approach using the **addFlightPlan** method.

To add a flight plan, we need to decide on an ordering for timepoints that are mutex (mutually exclusive with respect to time).

For aircraft that wish to land, we employ a heuristic approach that if an aircraft a follows an aircraft b for any timepoint, then aircraft a follows aircraft b for all timepoints. This allows us to attempt a "quick and dirty" scheduling where a follows b . We use the constraint verifier to validate that a proposed plan is feasible. We iterate through all the possible aircraft orderings, which is linear in the number of aircraft currently landing at the airport. If this fails, then **addFlightPlan** fails, and we need to generate a new plan using the vector generator.

For aircraft that wish to pass through the bubble, we use a similar technique, but choose an ordering from all aircraft in the bubble.

The vector generator

The *vector generator* generates a series of timepoints and temporal constraints that represents a hypothetical flight plan for an aircraft to an airport or to its exit destination. It also generates the information for the **mutex** sets for each timepoint of the plan. This vector generator is rather simple, but can be made arbitrarily rich, as the problem in general is NP complete. The vector generator iterates over options from most to least preferred.

For landing aircraft, the first vector generated is a full-procedure approach, followed by a most direct approach, followed by incrementally more distant vectors along the direct approach, followed by a single 360° delay in the holding spiral and a most direct approach, and so on, with as many 360° delays as needed. Once a vector is generated, it needs to be verified. If verified, it is incorporated in the plan.

For aircraft that are passing through, we use a similar technique to [1], where we start with a straight-line path, and randomly add waypoints, and then tighten the waypoints in 2d space to reduce flight-time. The 2d space is determined by the direction of travel, where certain altitude slices are used for northbound, southbound, eastbound, and westbound travel. If we have more than one altitude slice, we choose the slice that is closest to the current altitude of the aircraft. We break ties by choosing the least-recently used slice, and if we still have a tie, we break it randomly.

The constraint verifier

The *constraint verifier* checks a plan and introduces temporal constraints where necessary to maintain the veracity of a plan. It works by identifying timepoints that lack temporal ordering constraints between themselves and members of their **mutex** set. It then incrementally inserts ordering constraints and verifies that the constraint is possible using temporal propagation. It can prefer to either schedule new points before or after existing points, according to the preferences of the designers, e.g., we call this preferring that once an aircraft is scheduled before another at any point, it is always scheduled before that other aircraft. Once a verified plan is found, the verifier returns true. If none can be found, it returns false and removes any extraneous temporal constraints it inserted.

See Appendix A for pseudo-code implementations of the associated functions.

Results

Our implementation is in Visual C++ on a 2.53 GHz Pentium 4 processor. For problems with 32 or fewer aircraft, the complete solution requires less than a millisecond. Figure 4 shows the time to schedule a single aircraft given a number of aircraft already in the schedule. As we can see, even for large problems, we require less than a second of CPU time to add a new aircraft to the plan. This curve appears to scale in $O(n^3)$, where n is the number of aircraft already scheduled. This makes sense, in that temporal propagation is bounded by n^3 , and we are using heuristic techniques to perform the scheduling. Of course, we are most likely failing when we could succeed. In fact, for problems with 300 aircraft, half of the time a schedule was not found.

Future work

Future work in this area would be to improve the success of scheduling for larger problems and to include in our scheduling algorithms the built-in algorithms that exist for various on-board waypoint generators. This would render a more realistic experience for the simulation, but might require considerable work as the reasoning algorithms of each of these devices are proprietary.

Acknowledgements

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by Microsoft Corporation.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, the (name of sponsor if it is not a federal government organization), or the Jet Propulsion Laboratory, California Institute of Technology.

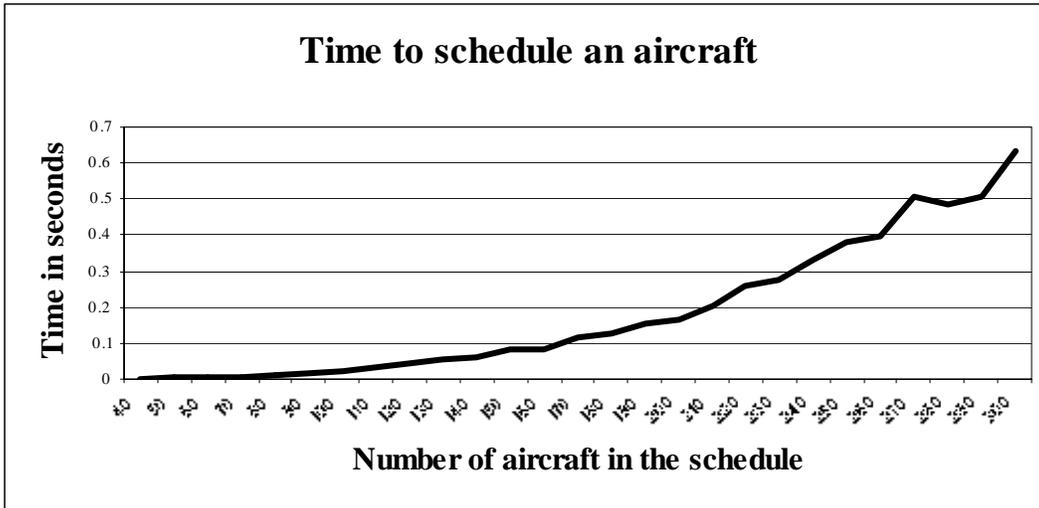


Figure 4 ATC scheduling performance

Appendix A Pseudo implementation

Assumed functions

Our algorithm descriptions assume many common functions that are not part of the original contribution of this work. These are summarized here.

We assume $\min(x_1, x_2, \dots, x_n)$ and $\max(x_1, x_2, \dots, x_n)$ which are functions that return the minimum and maximum values of the items given it as arguments.

For a set S , $\text{insert}(S, x)$ places x into S . If $x \in S$, S doesn't change. $\text{remove}(S, x)$ removes x from S . For an indexed set S , $\text{insert}(S, x, y)$ places y into S indexed by x . Only unique key values are kept. Subsequent insertion results in clobbering the data. $\text{remove}(S, x)$ removes the key x from S , as well as the datum associated with it. $\text{contains}(S, x)$ returns true if x is a key in S . $S[x]$ returns the datum associated with x . $\text{keys}(S)$ returns the set of key values.

For a list L , $\text{push}(L, x)$ places x onto the head of L . $\text{pop}(L)$ returns the item at the head of the list, deleting it. $\text{peek}(L)$ returns the item at the head of the list without removing it from the list. $\text{pushTail}(L, x)$, $\text{popTail}(L, x)$, and $\text{peekTail}(L, x)$ is as previous except the tail of the list is affected/accessed. We assume an implementation for lists.

A position p is a representation of a point in 3-dimensional space. $\text{distance}(p_1, p_2) \rightarrow \mathbb{R}$. $\text{crossPoints}(p_1, p_2, p_3, p_4, r)$, where p_1 and p_2 represent a line segment, p_3

and p_4 represent a line segment, and r represent a radius, returns either:

- (\emptyset, \emptyset) if no point on $p_1 \rightarrow p_2$ comes within r of $p_3 \rightarrow p_4$
- (x, \emptyset) if there is only one point on $p_1 \rightarrow p_2$ that comes within r of $p_3 \rightarrow p_4$, where x is that point
- (x, y) where x and y are both points on $p_1 \rightarrow p_2$ that comes within r of $p_3 \rightarrow p_4$ and x is the closest of x and y to p_1 .

$\text{minDuration}(a, p_1, p_2)$ returns the minimum amount of time required for aircraft a to traverse from position p_1 to position p_2 . $\text{maxDuration}(a, p_1, p_2)$ returns the maximum amount of time. $\text{currentDuration}(a, p_1, p_2)$ returns the amount of time at the current speed. $\text{nominalDuration}(a, p_1, p_2)$ returns the amount of time at the nominal speed.

$\text{propagate}(\text{STN} \langle G, l, u \rangle)$ which re-evaluates l and u such that all implied minimum and maximum intervals are reflected (i.e., temporal propagation.)

Pseudo-code

We now describe in pseudocode the implementation for **mutex**, **addMutex**, **removeAircraft**, and **addFlightPlan**,

```
mutex( n ) ≡ return M[n]
```

```
addMutex( n1, n2 ) ≡
  if contains( M, n1 )
    insert( M[ n1 ], n2 )
  else
    insert( M, n1, { n2 } )
  endif
  if contains( M ∈ P, n2 )
```

```

insert( $M[n_2], n_1$ )
else
insert( $M, n_2, \{n_1\}$ )
endif

```

```

removeAircraft( $a$ )  $\equiv$ 
 $X \leftarrow \{n \mid \text{aircraft}(n) = a\}$ 
for  $\forall x \in X$ 
  for  $\forall (n \mid n \notin X, x) \in E$ 
    remove( $E, (n, x)$ )
  endfor
  for  $\forall (x, n \mid n \notin X) \in E$ 
    remove( $E, (x, n)$ )
  endfor
endfor
remove( $A, a$ )

```

```

addFlightPlan( $G', a$ )  $\equiv$ 
for  $\forall (n_1, n_2) \in G'.E$ 
  for  $\forall (n_3, n_4) \in E$ 
    insert( $V, n_1$ )
    insert( $V, n_2$ )
    setAircraft( $n_1, a$ )
    setAircraft( $n_2, a$ )
     $l(n_1, n_2) \equiv \text{minDuration}(a, n_1, n_2)$ 
     $u(n_1, n_2) \equiv \text{maxDuration}(a, n_1, n_2)$ 
     $p \leftarrow \text{crossPoint}(n_1, n_2, n_3, n_4)$ 
    if  $p \neq \emptyset$ 
      if  $\text{distance}(n_1, n_3) \leq \varepsilon \vee$ 
         $\text{distance}(n_1, n_4) \leq \varepsilon \vee$ 
         $\text{distance}(n_2, n_3) \leq \varepsilon \vee$ 
         $\text{distance}(n_2, n_4) \leq \varepsilon$ 
        if  $\text{distance}(n_1, n_3) \leq \varepsilon$  addMutex( $n_1, n_3$ )
        if  $\text{distance}(n_1, n_4) \leq \varepsilon$  addMutex( $n_1, n_4$ )
        if  $\text{distance}(n_2, n_3) \leq \varepsilon$  addMutex( $n_2, n_3$ )
        if  $\text{distance}(n_2, n_4) \leq \varepsilon$  addMutex( $n_2, n_4$ )
        else
          handleIntersection( $p, n_1, n_2, n_3, n_4$ )
        endif
      endif
    endif
  endfor
endfor
return deconflict( $G'$ )

```

The new plan is integrated with the previous plan, but we have not made sure that aircraft do not collide at intersections. We do this by introducing ordering constraints. We know what order the aircraft should cross by determining the order that they should land (this is an approximation to avoid huge computational complexity). We try to place the new aircraft at each possible landing ordering until we find one that works. To find out what is the current ordering, we tally how many planes are landing before each other. Thus, *beforeCount* is a mapping of aircraft to counts of aircraft landing before it. Each aircraft has a unique number landing before it, starting with 0, thus this gives us a total ordering on the

aircraft for landing without performing a topological analysis on the network.

```

deconflict( $G'$ )  $\equiv$ 
propagate( $STN$ )
let beforeCount be a set of integers indexed by airplanes,
as above
allMutexAircraft  $\leftarrow$  keys(beforeCount)
let allMutexEdges be an empty edge-set
 $n \leftarrow \text{entryVertex}$ 
while  $n \neq \text{landingNode}$ 
  for  $\forall n_1 \in \text{mutex}(P, n)$ 
    insert(allMutexEdges, ( $n, n_1$ ))
  endfor
  ( $n_2, n$ )  $\leftarrow e \in \text{outdegree}(N \in P, n) \mid \text{aircraft}(P, n) = a$ 
endwhile
for  $i \leftarrow | \text{beforeCount} |$  down to 0
  if schedule(beforeCount, allMutexEdges,  $a, i$ ) return true
endfor
return false

```

```

schedule(beforeCount, allMutexEdges,  $a, i$ )  $\equiv$ 
for  $\forall (n_1, n_2) \in \text{allMutexEdges}$ 
  if getValue(beforeCount, airplane( $P, n_2$ )) <  $i$ 
    setMin( $N \in P, n_2, n_1, 0$ )
    setMax( $N \in P, n_2, n_1, \infty$ )
  else
    setMin( $N \in P, n_1, n_2, 0$ )
    setMax( $N \in P, n_1, n_2, \infty$ )
  endif
endfor
result  $\leftarrow$  canPropagate( $N \in P$ )
if  $\sim$ result
  for  $\forall (n_1, n_2) \in \text{allMutexEdges}$ 
    remove(edges( $N \in P$ ), ( $n_1, n_2$ ))
    remove(edges( $N \in P$ ), ( $n_2, n_1$ ))
  endfor
endif
return result

```

```

handleIntersection( $p_1, n_1, n_2, n_3, n_4$ )  $\equiv$ 
 $p_2 \leftarrow p_1$ 
 $a_1 \leftarrow \text{aircraft}(n_1)$ 
 $a_2 \leftarrow \text{aircraft}(n_3)$ 
 $n_{1a} \leftarrow \text{wayPoint}(p_1, n_1, \varepsilon)$ 
 $n_{2a} \leftarrow \text{wayPoint}(p_1, n_2, \varepsilon)$ 
 $n_{3a} \leftarrow \text{wayPoint}(p_2, n_3, \varepsilon)$ 
 $n_{4a} \leftarrow \text{wayPoint}(p_2, n_4, \varepsilon)$ 
insert( $V, p_1$ )
insert( $V, p_2$ )
insert( $V, n_{1a}$ )
insert( $V, n_{2a}$ )
insert( $V, n_{3a}$ )
insert( $V, n_{4a}$ )
aircraft( $p_1$ )  $\equiv a_1$ 
aircraft( $n_{1a}$ )  $\equiv a_1$ 
aircraft( $n_{2a}$ )  $\equiv a_1$ 

```

```

aircraft( $p_2$ )  $\equiv a_2$ 
aircraft( $n_{3a}$ )  $\equiv a_2$ 
aircraft( $n_{4a}$ )  $\equiv a_2$ 
 $l(n_1, n_{1a}) \equiv \text{minDuration}(a_1, n_1, n_{1a})$ 
 $l(n_{1a}, p_1) \equiv \text{minDuration}(a_1, n_{1a}, p_1)$ 
 $l(p_1, n_{2a}) \equiv \text{minDuration}(a_1, p_1, n_{2a})$ 
 $l(n_{2a}, n_2) \equiv \text{minDuration}(a_1, n_{2a}, n_2)$ 
 $l(n_3, n_{3a}) \equiv \text{minDuration}(a_2, n_3, n_{3a})$ 
 $l(n_{3a}, p_2) \equiv \text{minDuration}(a_2, n_{3a}, p_2)$ 
 $l(p_2, n_{4a}) \equiv \text{minDuration}(a_2, p_2, n_{4a})$ 
 $l(n_{4a}, n_4) \equiv \text{minDuration}(a_2, n_{4a}, n_4)$ 
 $u(n_1, n_{1a}) \equiv \text{maxDuration}(a_1, n_1, n_{1a})$ 
 $u(n_{1a}, p_1) \equiv \text{maxDuration}(a_1, n_{1a}, p_1)$ 
 $u(p_1, n_{2a}) \equiv \text{maxDuration}(a_1, p_1, n_{2a})$ 
 $u(n_{2a}, n_2) \equiv \text{maxDuration}(a_1, n_{2a}, n_2)$ 
 $u(n_3, n_{3a}) \equiv \text{maxDuration}(a_2, n_3, n_{3a})$ 
 $u(n_{3a}, p_2) \equiv \text{maxDuration}(a_2, n_{3a}, p_2)$ 
 $u(p_2, n_{4a}) \equiv \text{maxDuration}(a_2, p_2, n_{4a})$ 
 $u(n_{4a}, n_4) \equiv \text{maxDuration}(a_2, n_{4a}, n_4)$ 
addMutex( $p_1, p_2$ )
addMutex( $p_2, n_{1a}$ )
addMutex( $p_2, n_{2a}$ )
addMutex( $p_1, n_{3a}$ )
addMutex( $p_1, n_{4a}$ )

```

A flight-path builder computes all feasible paths to land a given aircraft at a given airport. **initialPath**(a, p) returns a graph representing a flight path from aircraft a to airport p . The flight-path builder is initialized for subsequent calls. **nextPath**() returns a graph as in initialPath for the aircraft and airport given to initialPath. If the returned graph = \emptyset , no subsequent paths are possible.

The flight-path builder assumes an aircraft a , an airport p , a boolean *turnRight* indicating whether or not the runway is reached using a right hand turn for a single turn approach, an integer r indicating the approach row, an integer c indicating the approach column, and an integer h indicating the number of holding loops. We assume the values *maxRow*, *maxColumn*, and *maxHolds*.

```

initialPath( $a, p, ap$ )  $\equiv$ 
 $a \leftarrow a$ 
 $p \leftarrow p$ 
 $ap \leftarrow ap$ 
 $c \leftarrow 0$ 
 $r \leftarrow 0$ 
 $h \leftarrow 0$ 
 $\text{turnRight} \leftarrow \text{needsToTurnRight}(a, p)$ 
return nextPath()

```

```

nextPath()  $\equiv$ 
if  $c = \text{maxColumn} \wedge$ 
 $r = \text{maxRow} \wedge h = \text{maxHolds}$ 
return  $\emptyset$ 
 $G \leftarrow \text{path}()$ 
if  $r = \text{maxRow}$ 
 $r \leftarrow 0$ 

```

```

if  $c = \text{maxColumn}$ 
 $c \leftarrow 0$ 
 $h \leftarrow h + 1$ 
else
 $c \leftarrow c + 1$ 
endif
else
 $r \leftarrow r + 1$ 
endif
return  $G$ 

```

path returns the path as a series of points in space that respect the limitations of the aircraft and use the waypoints as described earlier.

References

- [1] J. Barraquand, et al, "A Random Sampling Scheme for Path Planning," *International Journal of Robotics Research*, 16(6):759-774, 1997.
- [2] R. Decheter, I. Meiri, and J. Pearl, "Temporal Constraint Networks," *Artificial Intelligence Journal*, 49:61-95, 1991.
- [3] H. Erzberger, "Design Principles and Algorithms for Automated Air Traffic Management," *AGARD Lecture Series 200 Presentation*, Madrid, Spain, Paris, France, and Moffett Field, California, USA, November 1995.
- [4] M. Nolan, *Fundamentals of Air Traffic Control*, 3rd Ed., Brooks/Cole Publishing Company, Pacific Grove, California, 1998.