# Airplanes Group 2 Report

Harjot Gill, Tiernan Garsys, Sam Raper

December 21, 2013

# Contents

# 1 Introduction

# 2 Initial Insights and Observations

When we first started working on this project, our initial intuition was to solve the problem at a single-agent level without doing any sort of pre-calculation or simulation; one would simply need to develop some simple rules for directing the airplanes toward their goal while avoiding other airplanes, and the rest should fall into place. Our initial implementation, to this end, was a simple agent wherein each plane would be influenced by a "force vector" at each timestep, similar to a "boid" simulation seen in computer graphics. The basis for this force vector would be a vector pointing from the plane's current location to its destination. At every time step, one would modify this vector by adding a "repulsive" vector pointed away from any nearby planes or walls. Once the final vector was calculated, the plane would manuever toward this vector unwaveringly.

Additionally, we made the insight (along with several other groups) that the total runtime for any particular map was bounded by the maximum across all planes $P$ of $distance(P_{source}, P_{destination}) + P_{departuretime}$. In light of this, our initial vector implementation contained a prioritization mechanism wherein planes which would land later in an ideal solution were granted precedence when deciding which plane manuevered out of the way during collision scenarios.

Upon testing our player, it became apparent that our initial approach was insufficient for this problem. While scenarios with very few planes were solved easily, the scaling of the problem to maps with tens of airplanes would result in frequent collisions as planes attempts to avoid one another would inevitably result in collisions with other planes within the simulation. This limitation primarily arose from the fact that each agent only considered the immediate state of the board at each timestep, disregarding both the implications of its potential move and the potential actions of all other planes in the simulation; by disregarding such information, it was easy to fall into cases where the "optimal" actions taken by any two individual planes in the simulation would lead them to a course which was uncorrectable, and thus would result in a collision. From this, we realized that a more unified, intelligent strategy was necessary to succeed.

# 3 Strategies & Concepts

In a radical shift from our initial strategy, we ultimately decided on a strategy that involved the pre-calculation of flight paths prior to launch. The various aspects of this strategy are outlined below.

## 3.1 Launch-Time Simulation and Pathfinding

Instead of attempting to calculate the flight path of a plane dynamically in the air, our final solution instead set the flight path of a plane once and let it run its course. Once a plane $P_i$'s (the i'th plane to depart in the current session) departure time has been reached in the session, the player will begin simulating this plane's path to the destination as follows...

- Simulate a path between $P_i$'s source and destination, considering the presence of $P_0 ... P_{i-1}$'s pre-calculated paths with a set of obstacles. On the first iteration, this will be a straight-line path between $P_i$'s source and destination, as no obstacles will have been recorded.
- If $P_i$ reaches its destination successfully, then set that as the path for $P_i$ in this session and launch it. $P_i$ will follow this path until the end of the session.
- If $P_i$ experiences a collision during the simulation, then restart the simulation. On this new simulation, add an obstacle for the pathfinding where the collision took place.

## 3.2 Flow Detection

In response to the trend of "flow" boards that arose during this project, our team added a special "flow detection" routine during the training phase of the player. If, during training, the player noted the existence of a "flow" on the board (a sequence of five or more planes which approximately shared their source, destination, and departure time), then the simulator would, instead of generating a path as outline above, generate a serialized path where planes would be dispatched in a single-file line from the flow source to the flow destination.

# 4 Implementation

## 4.1 Flight Plan Determination

At each call of the `updatePlanes()` method, our solution will iterate through the collection of planes and perform the appropriate update action based on its current status within the session.

- For each plane that has not taken off, we calculate a flight plan using the pathfinding implementation described below, which ultimately returns a `List<Waypoint>` representing the waypoints that must be traversed by the plane on the way to its destination. The plane is then dispatched heading toward its first waypoint.
- For each plane that has taken off, check its current bearing and location relative to its current waypoint. If it is within a certain radius of its current sought `Waypoint` , pop that `Waypoint` off the front of the list and take the next `Waypoint` in the list to be the current `Waypoint` . If the plane's bearing is directed toward its current `Waypoint` , then maintain course; else, turn the plane either `TURN_RADIUS` or $Angle(CurrentBearing, GoalBearing)$ degrees (whichever is smaller) toward the current waypoint.

## 4.2 Pathfinding Implementation

Pathfinding in our implementation is based off of the A* algorithm, using straight-line distance as a heuristic. Because this heuristic is admissible (i.e. it never over-estimates the actual distance needed to reach the goal), one knows that for any particular board configuration it will generate the optimal path between the source and destination. The procedure for determining a path is as follows.

- TODO: Make this detailed.
- Set some waypoints
- Find the shortest path among visible waypoints.
- Simulate that path. If it finishes, great! If it doesn't, add an unpathable obstacle around the collision point, and add waypoints around this obstacle. Repeat.

In the initial implementation, a collision obstacle resulting from a collision in a prior simulation would be present in each time step of subsequent simulations, thus causing all pathing decisions to attempt to move around it. In subsequent implementation, we implemented that the obstacle would only appear in timesteps around that in which the collision generating the obstacle occurred, to simulate the presence of the collided plane at that particular point in time.

## 4.3 Referenced Constants

- `TURN_RADIUS` : The number of degrees the plane may turn during a timestep. The theoretical limit of 10.0 by the simulator was found to cause issues due to floating point imperfections, so a value of 9.5 was ultimately used.

# 5 Results

# 6 Contributions

# 7 Future Directions & Limitations

## 7.1 Flow Optimization

One minor shortcoming of our solution as presented was its performance on so-called "flow" boards, characterized by large numbers of planes that shared their source, destination, and departure times and so-named from the serialized flow of planes that would form between the source and destination. While we were able to improve our performance on these boards by adding flow detection to the pre-simulation training in the code (implemented by detecting the presence of five or more planes sharing a source an destination), our solution was limited by the fact that only one flow of planes was allow between any source-destination pair. On boards such as `DiagonalFlows` with large amounts of free airspace, Group 5's player was able to detect the possibility of multiple flows between the source and destination and subsequently schedule planes to proceed to the destination in two slightly-staggered flows. While the staggering (necessary for any particular plane to avoid collision with a plane in another flow immediately at takeoff, before that other plane had cleared the airspace) severely reduced the runtime improvements of this strategy, it was nonetheless better than a one-flow solution; Group 5's player demonstrated a runtime of 666 steps on `DiagonalFlows` , while our player demonstrated a runtime of 711 steps.

One could improve on this limitation by adding detection for multiple flow paths between a source-destination pair during the training phase of our player. Our current implementation of flow-detection works by finding a shortest path between the source-destination pair, treating other flows as obstacles obstructing this path. One possibility would be to generate some number of paths between the source-destination pair, determine which paths are close enough to the optimal path as to not increase the overall runtime of the simulation after necessary staggering was taken into account, and dispatch planes to each of these flows in turn. Potential implementation difficulties would be being able to determine prior to simulation that such a splitting would not simply increase the runtime of the entire simulation.

## 7.2 Pathfinding Prioritization / Sorting

Another problem with our solution was the possibility of giving planes whose paths were determined last in the sequence of planes overly long paths. As outlined above, our method of determining paths was greedy in that we would determine the path for any particular plane $P_i$ by simply simulating the shortest A* path between the source and destination of $P_i$ in an environment with planes $P_0 ... P_{i-1}$, resetting the simulation and trying again with a obstacle placed at the collision point in the event of a collision. This methodology resulted in a greater number of collisions for the last planes to have their path decided, which would lead them to be given longer paths to avoid collisions. Problems arose in that the ordering for resolving plane paths was more-or-less arbitrary; it was very possible that a plane with a short path in an optimal solution would be given a longer path, potentially to the detriment of simulation runtime, due to the fact that it had to consider more obstacles than other planes in determining its final path.

We attempted to address this problem by prioritizing the order with which planes' paths were determined in our pre-flight simulations. Methods tried include...

- `Shortest Path First:` Order the planes in ascending order by path length, and resolve flight paths in that order. The intuition behind this was that shorter paths would have fewer intersections with other paths, and thus their resolution would generate fewer obstacles for later-resolved flights.
- `Longest Path First:` Order the planes in descending order by path length, and resolve flight paths in that order. The intuition behind this was that longer flights are more likely to be the limiting factor in the overall runtime of the simulation, so resolving them first would ensure their runtime would not be increased by collisions with shorter flights.
- `Least Intersections First:` Determine the straight line paths between all source-destination pairs in the simulation. Order the planes in descending order by number of intersections with other straight line paths, and resolve flight paths in that order. This method attempted to resolve flights that would be interfered with by many other flights first, thus prevent their runtimes from skyrocketing.

In experiments, we found that each of the above methods yielded superior results in different simulations, with no clear trend of certain strategies working better on certain maps. Due to the fact that our implementation of sorting was incompatible with our flow detection, our final solution ultimately scrapped prioritization of plane flights. One issue that would have to be solved if this were implemented in the future would be gathering useful information for prioritizing flights from the information that is available at the beginning of the simulation. One only knows when the simulation starts what the source, destination, and departure time of each plane is. As of time of writing, we were unable to find any way of extrapolating from this data a prioritization that would reliably yield better results on most boards.

# 8 Acknowledgments

`Chris Murphy` , for the awesome class.

`Tanveer Gill` , for having helped developed the A* package used by our group with Harjot during the Mosquitos project.

# 9 Conclusion