Airplanes Group 2 Report

Harjot Gill, Tiernan Garsys, Sam Raper

December 22, 2013

Contents

1 Introduction

The goal of this project was to create a strategy to efficiently schedule and control aircraft. While avoiding collisions was obviously the most important goal of the project, delivering all planes to their destination in as few steps as possible proved to be a suitable metric in measuring effectiveness. Our player was given the departure times and destinations of all planes but little other information. Using only these two pieces of information, we strove to quickly deliver all planes. While in the air, planes were not permitted to be within five units of each other.

Throughout the project two important additional challenges were provided. The first was a specific kind of board, which we will refer to as a “flow board”. On these boards singular planes were not simply flying to various destinations. Instead, there were large numbers of planes departing and going to the same place. It quickly became apparent that a conventional strategy was not sufficient in satisfying these boards.

The second challenge presented was dependencies. In this case, some flights were dependent on the succesful arrival of one or more other planes. This too proved to require additional changes to our initial implementation.

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 maneuver 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 maneuvered 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.

It became quickly apparent that in the special case of flows a separate approach need be taken. We define a flow as 5 planes departing from the same airport at the same time and going to the same place. While identifying these flows was fairly simple, defining their behavior proved to be more interesting. As flows were in play on the board for much longer than a single airplane, picking a path that was both quick and unobtrusive was extremely important. The problem here lay in the fact that a single flow could obstruct a large amount of other planes so it could not simply be given a higher priority. However, flows were often the last planes to land so giving them ample time to land was equally important.

Our strategy for dealing with these flows once we identified them was essentially to treat them like walls as their steady stream of airplanes would effectively block all other activity in their area. To do so we simply added to the set of walls used by our A* object. These walls were then used for calculating paths for lone planes as well as additional flows. Unlike the lone plane paths, the flow paths were not time sensitive and would thus be considered at every step of simulation.

We found that one of the most important considerations when scheduling our flows was the number of planes in the flow. As our primary metric for evaluating our strategy was total number of steps, we found that prioritizing the flows with a large number of planes was important. In the cases of ties we simply gave precedence to the flow with shorter distance. This may seem counterintuitive longer flows will obviously take a longer time to land. However, in testing, we found that the shorter flows were much less problematic than the longer ones. As long flows often have to go across a large part of the board, they would effectively put a wall across a very important part of the board that disrupted many of the other planes, including some of the shorter flows. These flows would have to grow their paths unnecessarily large to avoid the larger flow. In practice, this was not worth it.

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.

```
Function A*(start,goal) is
    closedset ← EmptySet() // The set of nodes already evaluated.
    openset ← {start} // The set of tentative nodes to be evaluated, initially containing the
    start node
    came_from ← EmptyMap() // The map of navigated nodes.
    g_score[start] ← 0 // Cost from start along best known path.
    // Estimated total cost from start to goal through y.
    f_score[start] ← g_score[start] + heuristic_cost_estimate(start, goal)
    while openset.empty() != true do
        current ← openset.leastFScoreNode()
        if current == goal then
            return reconstruct_path(came_from, goal)
        end
        openset.delete(current)
        closedset.add(current)
        foreach element neighbor in neighbor_nodes(current) do
            tentative_g_score ← g_score[current] + dist_between(current, neighbor)
            tentative_f_score ← tentative_g_score + heuristic_cost_estimate(neighbor, goal)
            if closedset.contains(neighbor) == true && tentative_f_score >= f_score[neighbor] then
                continue
            end
            if openset.contains(neighbor) == false || tentative_f_score < f_score[neighbor] then
                came_from[neighbor] ← current
                g_score[neighbor] ← tentative_g_score
                f_score[neighbor] ← tentative_f_score
                if openset.contains(neighbor) != true then
                    openset.add(neighbor)
                end
            end
        end
    end
    return failure
end
```

Algorithm 1: A* Search Algorithm

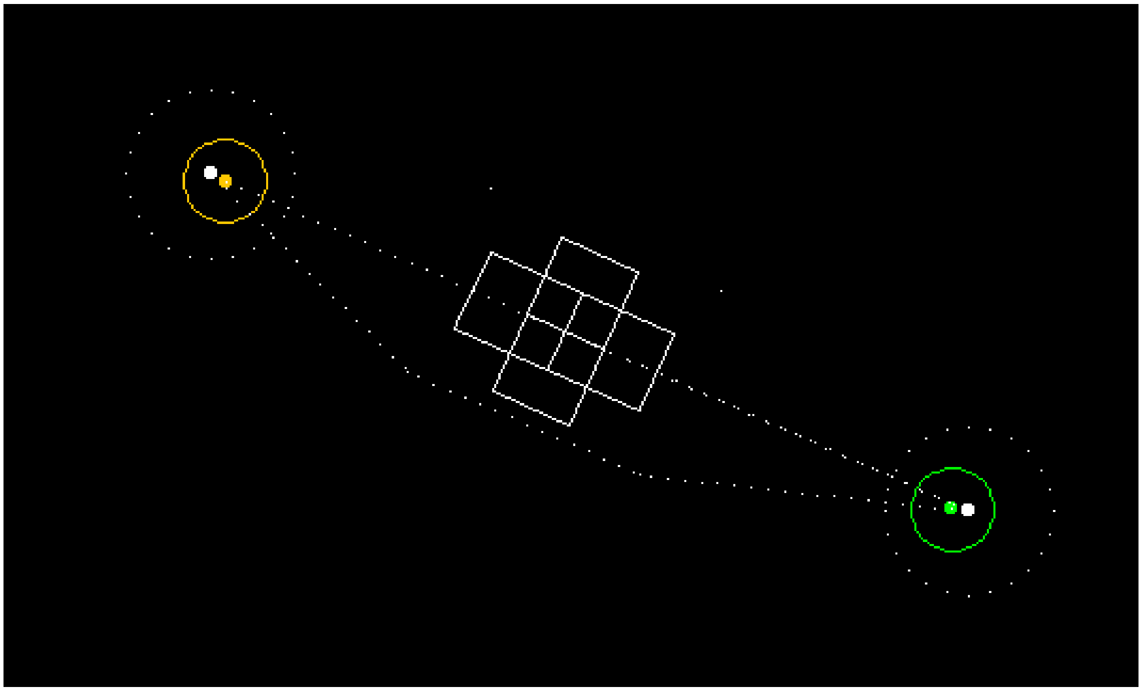

Figure 1: Ad-hoc Obstacles and Waypoint Placement

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 explained in Algorithm 1. Our A-Star algorithm implementation takes as input a set of waypoint locations on the board to traverse to get from a source to destination. The waypoints have to be placed so that A* is able to find path between any two given points on the board despite the obstacles and the path distance has to be reasonably close to the shortest possible distance between these points. Both these criterion are satisfied when the waypoints are near the edges of the obstacles.

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.

Figure 1 shows a typical obstacle and waypoint placement configuration from one of the simulation runs. As mentioned earlier, an obstacle (two perpendicular rectangles resembling a cross) is placed at the collision point and one of the colliding planes is forced to go around this virtual obstacle using the four waypoints around this obstacle, so as to avoid "real" collision. Waypoints were also added in a radial pattern around the airports, so as to allow planes to take-off in opposite direction than the destination, if they can. Figure 2 shows obstacle and waypoint placement for group of planes forming "flows". Unlike ad-hoc, real-time approach for pathfinding before taking-off new planes, the flow planes fly along pre-computed paths. This closely resembles circuit establishment approach in communication networks, in which traffic flows along a pre-determined circuit/path. This approach helps increase traffic throughput especially in scenarios in which multiple flows intersect each other. As mentioned earlier, the flows are detected before running the actual simulation and paths/circuits are pre-computed too. The non-flow

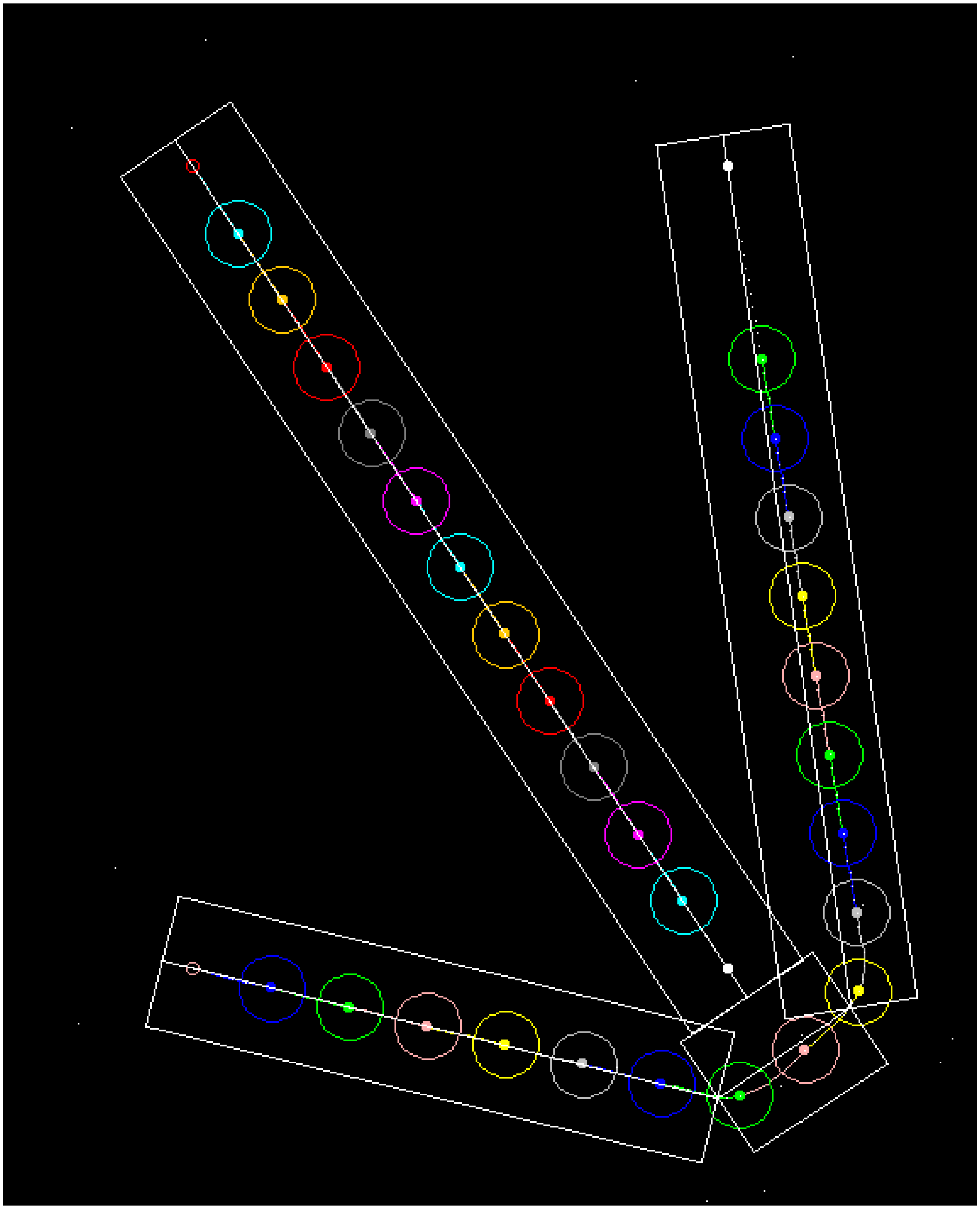

Figure 2: Flow Obstacles and Waypoint Placement

planes attempt to fly around the flow-planes using the normal ad-hoc approach at runtime.

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 where the simulator would crash due to a bearing change of more than 10 degrees, so a value of 9.5 was ultimately used.

5 Results

6 Contributions

6.1 Alternative Success Metrics

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

Overall, this was an extremely interesting and fun project.

One potential improvement in the future would be to introduce a better metric (or metrics) for success in this project. As indicated in class, our group felt that the metric of total runtime for the simulation was flawed, as it made the task too dependent on the performance of the last-arriving plane in the simulation; as long as they didn't directly interfere with the last arriving plane or delay themselves so much as to become the last arriving plane themselves, it became less important to optimize the behavior of other planes. Our group feels a better success metric would be to minimize the sum of total delay and total power used over the course of a simulation.