Project: Autonomous Drone Racing Competition

Syed Shoaib Hasan, Wanshun Xu

Overview of the algorithms and equations used and considered for the final project, as well as the justifications behind your decisions (10 %).

The project aims to develop a motion plan for a quadcopter in a 3D simulation environment. Given that all gates are centered at a height of 1 meter, we've simplified the z-axis adjustments by considering only the processes of take-off and landing. Consequently, the main trajectory planning through the gates to the target point can be treated as a 2D problem. For this, two primary motion planning methods were evaluated: search-based (e.g., A*) and sampling-based (e.g., RRT and RRT*) algorithms.

A* Algorithm: The search-based algorithm A* finds the shortest path between a start and a goal in a grid-like environment. Each grid has nodes, which is an allowable position the robot or drone can move to, and it's connected to other nodes via links. These links have traversal costs associated with them, which gives the distance costs from one node to the other. With a given start and end node, we have 2 functions, cost g(n), and heuristic h(n). The former keeps a tally of the cost from any one node in the environment to the start node, and the latter tells the cheapest cost from any one node to the goal point, which could be a diagonal line if Euclidean distance is allowed, or a straight line only if using the Manhattan distance. Obstacles in this environment are considered inaccessible nodes, and the total cost of each node is the sum of the cost and heuristic function, f(n) = g(n) + h(n). The algorithm goes through a loop of searching the neighbors of each node (starting with the start node) and picking the one with the least total cost. It then searches that node's neighbors and updates the total cost (it can revisit the same node if a cheaper path is found). This process repeats until the goal node is reached, or terminates if all nodes have been searched and the goal hasn't been found (indicating an unreachable goal). You can see a pseudo code of this process below:

```
make an openlist containing only the starting node
make an empty closed list
while (the destination node has not been reached):
    consider the node with the lowest f score in the open list
    if (this node is our destination node) :
        we are finished
    if not:
        put the current node in the closed list and look at all of its neighbors
        for (each neighbor of the current node):
            if (neighbor has lower g value than current and is in the closed list) :
                replace the neighbor with the new, lower, g value
                current node is now the neighbor's parent
            else if (current g value is lower and this neighbor is in the open list ) :
                replace the neighbor with the new, lower, g value
                change the neighbor's parent to our current node

            else if this neighbor is not in both lists:
                add it to the open list and set its g
```

Figure 1: Pseudo-code for A* search algorithm, A* Search. Brilliant.org. Retrieved, April 26, 2024, from https://brilliant.org/wiki/a-star-search/

In our implementation, we took notes from the lecture code given and defined the environment as a grid map. This algorithm is sensitive to the grid map's size, influencing the number of waypoints and the trajectory's accuracy, so we had three grid sizes:

[-3.5,-3.5,3.5,3.5]m, [-350,-350,350,350]cm, and [-3500,-3500,3500,3500]mm. The middle size proved to be the most practical, allowing the quadcopter to navigate through all gates effectively. However, the A* algorithm's tendency to trace close to obstacle boundaries raised concerns about potential collisions with gate frames. To mitigate this, we expanded the boundaries around obstacles, and set waypoints before and after the gate to ensure safer, central navigation through them.

RRT* Algorithm: The sampling-based RRT/RRT* algorithms also try to find the shortest path between a start and goal configuration. In RRT, the environment is a configuration space, where a random node is generated, and connected to the nearest node by links or edges (given a max distance), where the initial node can be at the start. Once again, obstacles are considered inaccessible areas. The advantage here is that a node pointing in another direction doesn't affect the path of nodes in the right direction (the ones rapidly going toward the goal). This process repeats until the path is within some threshold of the goal, and the branch connecting it to the start will highlight the path. This process uses fewer nodes than A*, as the max distance between said nodes can be set higher. The only difference between RRT and RRT* is where the connection of new nodes (links or edges) is placed. Instead of connecting it to the nearest node, RRT* considers a bunch of nodes in a specified search radius, and whichever node has a shorter complete path tracing back to the start is connected instead, minimizing path length. Adding more nodes continuously shortens the path, and with enough nodes, one can find a path for any environment.

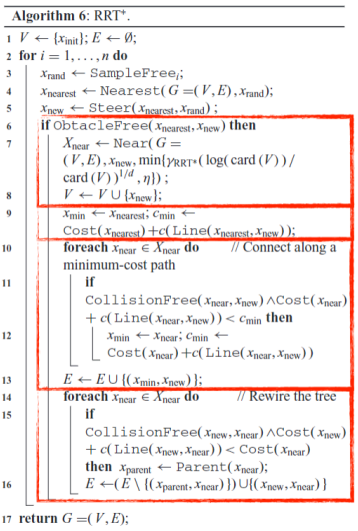

Figure 2: pseudo-code for RRT* algorithm (From aer1516_winter_2024_lecture_08, Professor J. Kelly, April 27, 2024)

The RRT* algorithm initiates with a loop that continues until either the maximum number of iterations is reached or the goal area is attained. Initially, it selects a random point on the map and identifies the nearest existing node to this point based on Euclidean distance. Using a local planner, the algorithm attempts to connect this nearest node (q_{near}) to the random point

(q_{rand}). Before establishing this connection, it is crucial to verify two conditions: whether q_{rand} is situated inside an obstacle, and whether the path between q_{rand} and q_{near} intersects with any obstacle. If no collisions are detected, the algorithm proceeds to add an edge from q_{near} to q_{rand}, its distance will depend on the step size. a rewrite function for reconsidering the parent node of the new node based on the shortest Euclidean distance.

In our project, we have used the RRT* code provided in lectures. The convergence is improved by modifying the algorithm so that it selects the target point as the next step in the 10% move. This adaptive method helps the algorithm to obtain the asymptotic optimal path faster. Given that RRT* can produce varied trajectories on each run, with significant differences between them, we've found it necessary to fine-tune hyperparameters such as step size, maximum iterations, and path rewriting range to effectively prevent collisions. Additionally, utilizing the entire map often leads to suboptimal route selections, for example, if the start and goal points are on the bottom side, they q_{rand} will highly chance select the top part of the map. To address this, we adjusted the map dimensions based on the specific starting and goal points. This modification enhanced the efficiency of the RRT*. We add goal points before and after the gate and consider the gate frame as an obstacle to ensure the algorithm will pass through its center.

Decision: Note that we only used the A* algorithm during the test day, as even when we had both A* and RRT* running in simulation, the RRT* took much longer to find a path and would often result in not so straight trajectory. The benefit of RRT* however is its wider coverage, so theoretically if there were slight obstacle changes during each trial on the test day, or the end target was to be moved on each trial, RRT* would be more beneficial as it explores the environment more evenly. We show both A* and RRT* in simulation results for completeness sake.

Discussion of the simulation results (5 %)

Gate Sequence: [Gate 1, Gate 3, Gate 4, Gate 2, Gate 1, Gate 4]

The team used A* and RRT* for path planning, and a combination of high-level and low-level commands to finish the tasks. The sequence for finishing the task is 4 seconds for 'Takeoff', then we switch to low-level 'Cmdfullstate' for the trajectory following, the operation time depends on the duration of the polynomial fit. After that, a 'Notifysetpointstop' is for transferring low-level commands to high-level commands, and then a 2-second 'GOTO' command is for moving the drone to the target point. Lastly, there is 3 seconds for the 'LAND' command. The 'STOP' command is to be sent once the trajectory is completed.

A* Algorithm (Deg = 45, Duration = 30)

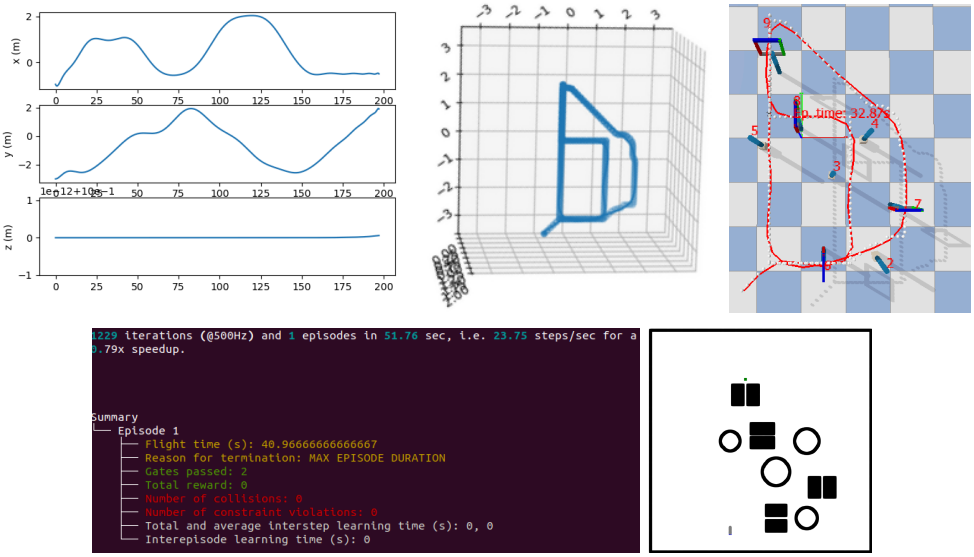

Figure 3: The results for the A* algorithm (the last figure is the environment of the simulation)

This figure illustrates the simulation results with the A* algorithm where our quadcopter successfully navigated through all gates without collision. The trajectory is smooth and free from abrupt paths that might cause the drone to flip. The red paths accurately align through the center of each gate, maintaining a safe distance from all obstacles. The drone completed its entire trajectory, including take-off and landing, in 41 seconds. This duration can be adjusted by changing the duration number. The team used a more conservative estimate of 30 seconds to minimize the impact of turns. The shortest duration tested was 9 seconds. Overall, the results are both satisfactory and desirable.

RRT* Algorithm (Deg = 25, Duration = 30)

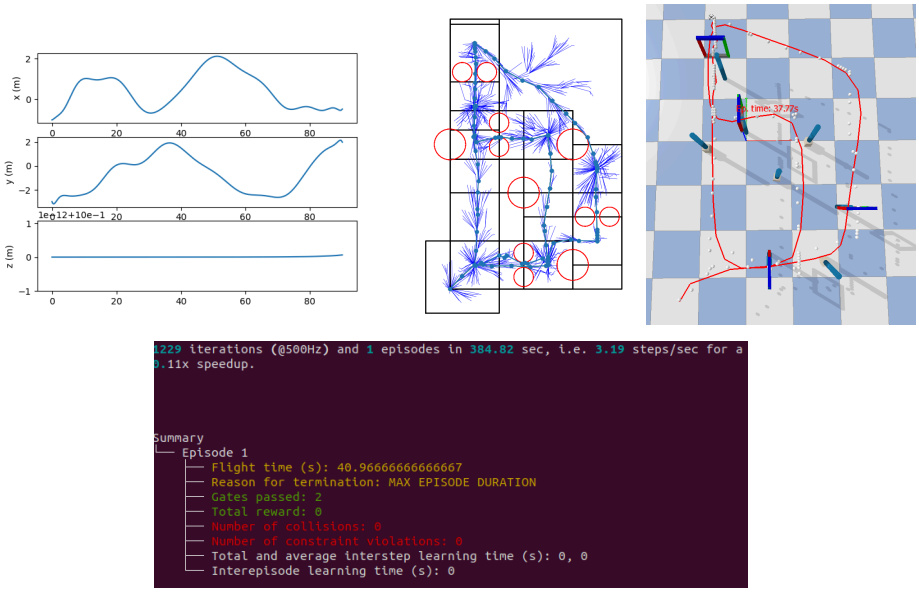

Figure 4: The results for the RRT* algorithm

This figure illustrates the simulation results of the RRT* algorithm. In comparison to the A* algorithm, RRT* exhibits less predictability; each run generates a different trajectory influenced by factors such as step size, rewrite size, and sampling range. This variability sometimes leads to collisions, occurring in approximately two out of five simulations. Additionally, the variability in the number of waypoints affects the degree of the polynomial used, which in turn impacts the polynomial fit calculation. RRT* also requires a longer runtime, but those problems can be solved by allowing the Algorithm to keep running until a specific iteration, it can get the shortest path, and save the path to avoid the computation time. Unlike A*, which follows a more deterministic path, RRT* explores a broader range of possibilities, allowing the drone to attempt various approaches to navigate through gates.

–Discussion of the difficulties faced during the final project, including what went well or as expected and what did not go well (5 %).

What went well: For both A* and RRT*, we were able to complete virtual simulations in Pybullet with relative ease, with a smooth takeoff, passing through all gates, and achieving a controlled landing. During the (non-graded) practice session, we were able to go through all the 4 experimental gate sequences (1234) and achieve a controlled landing. We also passed through the 6 gate sequences (134214) in the Pybullet simulation on the test day, and while we couldn't go through all 6 of them in the experiment, we did pass through the first 4, which incidentally are all the distinct gates (1-4).

What did not go well: The major thing was that we could not go through the complete 6 gate sequences on test day. In our first trial, the drone flipped after the third gate, in the second it went too fast and collided with the second gate, and in our last try, it hit an obstacle after the 4th gate (in between gate 1 and 2).

What we learned afterwards: The major problem we were having was controlling the drone's speed. In the simulation we controlled it by increasing the control input iteration in the cmdFirmware, not knowing how it would later affect the low-level cmdFullState commands. The correct way to change the speed was to change the duration variable. In addition, at the time we didn't understand the difference between going from high-level to low-level (cmdFullState) commands and chose to have a controlled landing by appending waypoints to lower the drone's height (the correct way was to switch back to high level and use the land command). This messed up the polyfit of the trajectory, causing the drone to sway around a little bit in simulation. Add in the latency of control inputs from the VICON system to the drone, and the experimental results during test day were very unfavorable for us. Note that these fixes are implemented in the code given.