Flying Car and Autonomous Flight Engineer Nanodegree

Project 2 – 3D Motion Planning

OBJECTIVE : Build a 3D motion planner that allows the quadcopter to successfully go from point A to point B in a simulated model of the San Francisco city.

APPROACH : The solution makes use of a simple A* search and Bresenham's ray tracing algorithm to plan the path from points A to B.

STARTER CODE : The starter code motion_planning.py is a modified version of backyard_flyer_solution.py. The key differences are listed below

- motion_planning.py makes use of a new state 'PLANNING' while searching for a path.
- Within the state_callback() function, when the state is ARMING, the plan_path() function is called. plan_path() in turn calls the a_star() search function from planning_utils.py.
- The plan_path() function sets a start location and a goal location. It replaces the square vertices from backyard_flyer_solution.py with waypoints obtained using a_star() search.
- The create_grid() function utilizes data from the colliders.csv file to generate an occupancy grid representation of the map for a given target altitude.
- The search happens in a 2D plane at the target altitude.
- a_star() search uses the Euclidean distance between a given state and the goal location as a heuristic to reduce the search space.

IMPLEMENTATION :

Set home position

```
        # Read lat0, lon0 from colliders into floating point values
        with open('colliders.csv', newline='') as f:
            reader = csv.reader(f)
            row1 = next(reader)  # gets the first line
        lat0 = float(row1[0].split()[1])
        lon0 = float(row1[1].split()[1])

        # Set home position to (lon0, lat0, 0)
        self.set_home_position(lon0, lat0,0)
```

Get current position

```
# Retrieve current global position
current_global_position = [self._longitude, self._latitude, self._altitude]
# Convert to current local position using global_to_local()
current_local_position = global_to_local(current_global_position, self.global_hom
e)
```

Set start location

```
# Convert start position to current position rather than map center
grid_start = (int(current_local_position[0]-
north_offset),int(current_local_position[1]-east_offset))
```

Set goal location

```
# Set goal as latitude / longitude position and convert
global_goal_position = [-122.397143, 37.793768, 0.0]
local_goal_position = global_to_local(global_goal_position, self.global_home)
grid_goal = (int(local_goal_position[0]-north_offset),int(local_goal_position[1]-
east_offset))
```

Search

The solution uses A* star search using diagonal motions with a cost of sqrt(2). A* search uses the following data structures

1. A priority queue – queue – to store nodes and corresponding queue cost.
2. A set – visited – to store the list of visited nodes.
3. A dictionary – branch – to store the mapping from node to a tuple containing branch cost, preceding node and action taken from preceding node.
4. A list – path – to store the path from start to goal.

First, the start node is added to the priority queue with a queue cost of 0. Then, in a loop that continues until either the queue is empty, or the goal is found, the node with the lowest cost is extracted from the priority queue. The associated branch cost is fetched from the dictionary (branch). Looping through each of the valid actions from that node, the next node is found.

If the next node has not been visited yet i.e. the node is not in the visited set, its branch cost is calculated by adding the cost of the action taken to the branch cost of the current node. Its queue cost is calculated by adding the branch cost and the heuristic value – an optimistic estimate of the expected cost of getting to the goal. The node is then added to the dictionary (branch) and priority queue (queue).

If the loop is terminated because a path is found, the dictionary (branch) is used to retrace the steps from the goal node to the start node.

The following code was added to the starter code to incorporate diagonal motions.

Class variables:

```
NORTHEAST = (-1, 1,np.sqrt(2))
NORTHWEST = (-1,-1,np.sqrt(2))
SOUTHEAST = ( 1, 1,np.sqrt(2))
SOUTHWEST = ( 1,-1,np.sqrt(2))
```

Code within the valid_actions() function:

```
    if x - 1 < 0 or y + 1 > m or grid[x - 1, y + 1] == 1:
        valid_actions.remove(Action.NORTHEAST)
    if x - 1 < 0 or y - 1 < 0 or grid[x - 1, y - 1] == 1:
        valid_actions.remove(Action.NORTHWEST)
    if x + 1 > n or y + 1 > m or grid[x + 1, y + 1] == 1:
        valid_actions.remove(Action.SOUTHEAST)
    if x + 1 > n or y - 1 < 0 or grid[x + 1, y - 1] == 1:
        valid_actions.remove(Action.SOUTHWEST)
```

Cull waypoints

Bresenham's ray tracing algorithm was used to remove adjacent collinear points. The function bresenham() returns a list of cells that lie along the line from cell 1 to cell 2 in a 2D grid. Looping through sets of three adjacent cells – p1, p2, p3 – along the path from start to goal, bresenham() function is used to get the list of cells that lie along the line from cell p1 to p3. If all the cells between p1 and p3 lie in free space, p2 is unnecessary and is removed.

```
def prune_path(grid, path):
    if path is not None:
        pruned_path = [p for p in path]
        i = 0
        while i < len(pruned_path) - 2:
            p1 = pruned_path[i]
            p2 = pruned_path[i+1]
            p3 = pruned_path[i+2]

            cells = list(bresenham(int(p1[0]), int(p1[1]), int(p3[0]), int(p3[1])
))
            free = 1
            for j in cells:
                if grid[j[0]][j[1]]==1:
                    free=0
                    break
            if free == 1:
                pruned_path.remove(pruned_path[i+1])
            else:
                i += 1
    else:
        pruned_path = path

    return pruned_path
```

Link to simulation video : https://youtu.be/l_0MA_1FAuI

CHALLENGES : There are errors or discrepancies in the simulator code which causes some unexpected results. The starting location provided is within an obstacle in the simulator map. This causes the algorithm to fail to find a path unless the quadcopter is flown in manual mode to a different location before beginning to test in guided mode. The simulator also gives a misleading error (int has no attribute time) if the start and goal locations are too far away. I was unable to use my local machine to complete the project because of a “host machine closed connection” error. This error did not crop up when I tried the same code in the VM.