Explain the Starter Code

The starter code is more or less the same as the code for the first project, however the finite-state machine (FSM) now includes a “Planning” state, such that the FSM now resembles figure 1.

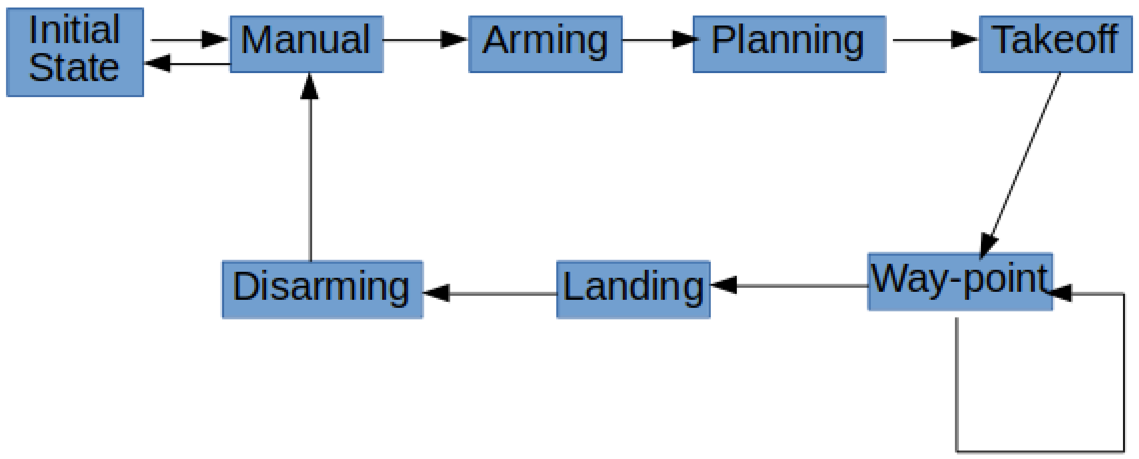

Figure 1: Finite-State Machine

The Planning state corresponds to running the plan_path function. This method by default:

- Loads the map (motion_planning.py line 133)
- Calculates the grid representation from the map data using the create_grid function from planning_utils.py (motion_planning.py line 136)
- Uses the A* Algorithm to calculate a path to the goal (motion_planning.py line 150)
- Generates way-points and sends them to the simulator (motion_planning.py line 155-159)

Please note that the above line numbers refer to the starting code not the finished code, but all other line numbers will refer to the finished code.

Implement Your Planning Algorithm

The first thing required of the planning algorithm is to initialize the drone’s home position from the ***colliders.csv*** file instead of assuming that the home position of the drone is the initial location. This is accomplished at lines 126 – 135 of ***motion_planning.py***.

The next thing required by the rubric was to retrieve the geodetic coordinates and convert these to local NED coordinates. This was accomplished by feeding the *self.global_position* attribute into the *global_to_local* function, as opposed to gathering and feeding the individual longitude, latitude, and altitude attributes. This occurs at line 138.

The *grid_start* variable needed to be set using these local NED values. The *grid_start* value is set to the middle of the grid by default. By setting this value ourselves we should be able to takeoff from anywhere. The calculating and setting of the new start occurs at lines 150 – 152.

We then needed to set the goal location (*grid_goal*) ourselves. Here I added the *goal_lon* and *goal_lat* parameters to the function and an if statement allowing the default goal (north + 10 and east + 10) to be used if *goal_lon* or *goal_lat* are None. If geodetic coordinates are specified in the function call, they will instead be used. I found it unintuitive how precise the GPS coordinates had to be in order for the planning algorithm to work (I blame my lack of precision on Kerbal Space Program). This piece of code makes up lines 154 – 167.

The fifth thing the project required was to modify the A* algorithm to allow for diagonal motion. This change was made in the ***planning_utils.py*** file. The first thing that needed to be done was to add *NORTH_WEST*, *NORTH_ EAST*, *SOUTH_WEST*, and *SOUTH_EAST* actions to the Action class at lines 59 – 63. The *valid_actions* function then had to be modified to filter through which of these actions would be deemed valid given the current position. This filtering occurs at lines 93 – 101. The rest of the default *a_star* method was then left unmodified.

The final thing needed for the planning algorithm was to write methods for pruning the way-points generated by A*. For this, I made a new file called ***pruning_utils.py***. This file contains the functions for collinear pruning. Three of the functions (*point*, *collinear*, and *collinearity_prune*) come more or less from class. These occupy lines 3 – 29. The other two methods were made after discovering that the drone would sometimes fly long zigzags with the in class *collinearity_prune* function. To fix this, I defined a modified collinearity pruning method that allows for the consideration of more than three points called *var_prune* (lines 31 – 69). This function looks to see if the beginning, end, and median of the range of points are collinear. If they are, the function removes the points between beginning and end (just like *collinearity_prune*). This serves to “anti-alias” the paths. I have only tested this function with point_range’s of 3 and 5 (this is a parameter to *var_prune* defining how many points to consider). These point ranges correspond to the default behaviour of collinearity_prune and the new “anti-aliasing” behaviour. To utilize the anti-aliasing behaviour, I defined the funciton *run_prune*. *run_prune* first runs *var_prune* with a *point_range* of 3 followed by a *point_range* of 5 to anti-alias the output. It then checks that the new path appears to be valid (longer than 0 but shorter than what we fed in). This function will continue to prune until it has pruned as much as it can and then return the fully pruned path. My path still ends up a little wonky in places, but I think the result is better than it was.

Executing the Flight

After the planning algorithm had been properly modified the only thing left was to test the algorithm. I completed this by manually flying the drone to a secluded part of the map (making sure the drone needed to turn a few times to get there) to get the GPS coordinates. I then loaded the GPS coordinates into the plan_path function, reset the simulator, and ran the code.