3D Motion Planning - Flying Car NanoDegree
Mark R. Helms
June 20th, 2018

Introduction

This project reads in 2.5D obstacle information for the local area and plans a route from a beginning latitude and longitude to a destination latitude and longitude. A progressively more complex a* graph search is used until a maximum size is reached, after which an a* grid search is necessary. Other path refinements and speedups are incorporated. The resulting path is converted into waypoints and used to control the drone.

Starter Code

The starter code provided by Udacity (motion_planning.py) is a modified version of the provided solution for an earlier project (backyard_flyer_solution.py). Both code files use an event-driven architecture to control a drone to navigate within the simulator.

The backyard flyer code uses a state and transition model with the following states: Manual, Arming, Takeoff, Waypoint, Landing, Disarming – and transitions for each state – and the following callbacks: Local_Position, Local_Velocity, State. The gist of the code is that it arms the drone, commands it to take off to a navigable altitude, commands it to advance to a series of waypoints by updating a target position as each waypoint is reached, then commands it to land and disarms it. Position tolerance for waypoints and the final landing point are controlled separately.

The starter motion planning code is similar, but adds a Planning state and graphically shows the waypoints in the simulator as green balls connected by white bars. The planning phase comes immediately after the arming phase and includes the following steps: 1) read in the 2.5D representation of the city. 2) create a grid representation of the city. 3) picks a goal position 4) uses A* search to find a path to it and assigns the set of destination waypoints to the path chosen.

The motion_planning.py code is supported by functions provided in planning_utils.py. The functions included and their descriptions are:

create_grid	uses the collision.csv data to create a 2D map of the city at a specific altitude
a_star	uses the A* algorithm to find a path using a Euclidean distance heuristic
support functions for a*: Action, valid_actions, heuristic	

Implementing the Path Planning Algorithm

Because the starter code from Udacity includes a basically functioning solution, all that's required is refining the process and product for a drone flight application.

There is an option to convert to using a graph search, which is taken. Despite the considerations that the search area is 2D (fixed altitude), the obstacles are simply-shaped and the hardware is a laptop/desktop, the area is still big enough to benefit from a graph search in terms of speed. However, using the grid also ensures a superiorly-asymptotically optimal path search, so a hybrid solution was devised. The graph search was performed, then the resulting path was converting into a grid path, after

which point path pruning was used. This results in a more efficient path than using graph search alone, but benefits from the graph search speed.

In order to further improve the speed of a graph search, the probability of placing graph nodes is biased to find 50% of a graph's size within the area of the city most likely to be traversed.

A task of adding in the diagonal movement was completed, even though graph search was used. This was accomplished by adding additional actions and adding checks to remove them if they are invalid.

The a* graph search builds a progressively more complicated grid if it's unable to find a path between the start and goal nodes. If the graph search fails, the slow, but optimal grid search will be used.

In order to further improve the speed of the algorithms, a "straight_line_possible" function was used inside the a* searches in order to cut down unnecessary searches. This is of most help in the grid version of a*. The same function is used to build the graph nodes in graph_from_grid and to prune unnecessary steps in prune_path_bresenham. The a* graph algorithm adds the start and goal locations to the graph so that it's possible to find them!

Improvements

The graph search creates nodes that may be suboptimal. While the path pruning algorithm helps keep some of the wasted travel down, it's possible that if a node is around a corner, the path may still take a longer-than-necessary route. Additional checks could be made to prune this slightly extra travel.