Brian Boxell

RBE550

Valet

For this assignment, I implemented a Hybrid A* planning algorithm to accomplish the path planning for a car. The algorithm behaves like a normal A* algorithm, but instead of using grid cell, the search domain is all poses of a car that are kinematically possible to reach by the next loop iteration. I chose this algorithm because it is what I am using for my final project. However, there are a number of complexities regarding this algorithm that unfortunately prevented me from finishing this assignment on time. In this document, I will both go over how the algorithm is supposed to work theoretically and explain the issues that I am running into.

Just like a normal search algorithm, it works by exploring nodes from a frontier. The first node in the frontier is obviously the starting point of the car. At each node, there is a call to the function `produce_neighbors`, which returns a list of discrete positions that are kinematically possible to reach. Of course, there are infinitely many positions, so this algorithm works by discretizing the search space. This algorithm is where the kinematics of the car is implemented. Once the "neighbors" have been calculated, each neighbor must be vetted for validity. A neighbor is valid if it is not already in the frontier, has not been previously explored, and does not collide with other obstacles. If it is valid, the neighbor gets added to the frontier. New neighbors are inserted into the frontier in order of their cost, which is calculated by a separate cost function. The algorithm has succeeded in planning if the x,y, and theta are within some threshold of the target.

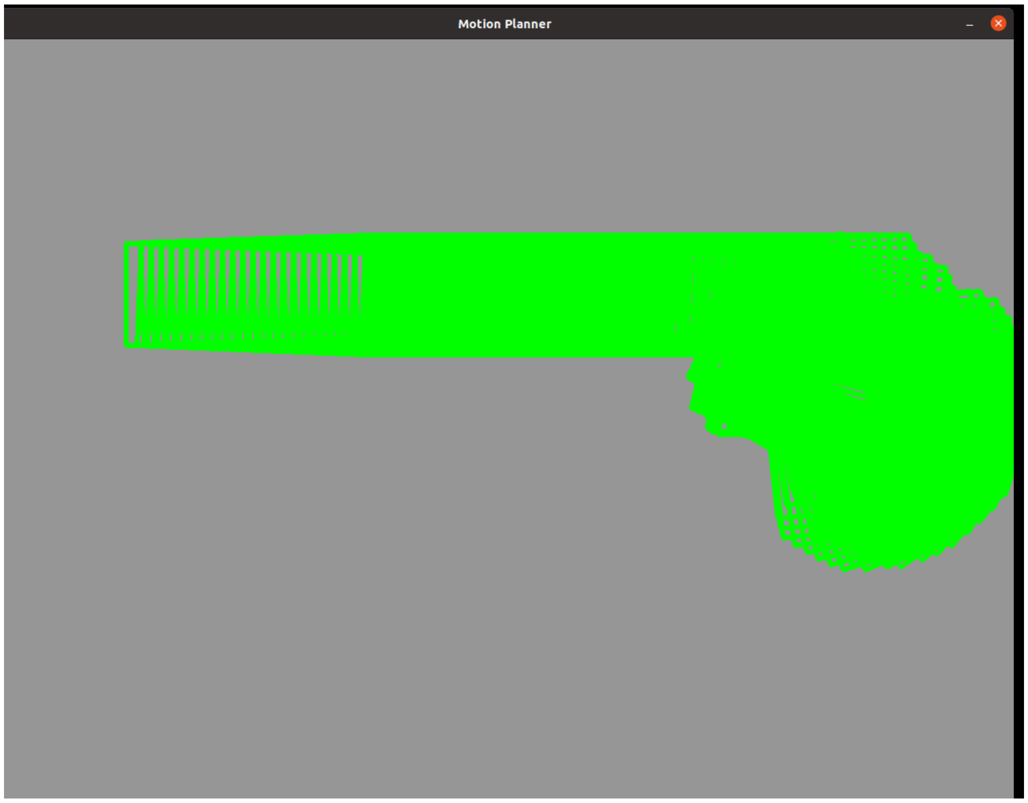
The car kinematics works by assuming that the car can drive a given distance over one loop iteration, based on the car's speed. This makes it easy to calculate a new position that is directly in front of or behind the current position. For turning for an Ackerman steering car, I begin by calculating the instantaneous center of curvature (ICC) for various wheel angles. Then, I figure out the arc that the car drives on, and the new coordinates expected if the car were to drive the previously decided distance along the arc. The selection set of wheel angles is -70 to 70 degrees in increments of 10.

Collision detection works by modeling the cars and obstacles as rectangles. Collision detection works on 2 objects at any given time: the car and the obstacle. For each rectangle (car and obstacle), I calculated orientation unit vectors that align with the axis of the rectangle at any angle. I also calculate the coordinates of the corners of each rectangle. I then project the points onto the orientation vectors, and measure the resulting magnitudes. If the resulting magnitudes are less than the width/2 or length/2 of the obstacle, then there is a collision.

All of the previously mentioned code works. The issue is the cost function- currently the algorithm is really bad at prioritizing nodes that better align with the target. Currently, the cost function is just the euclidean distance from the target with a large bias against poses that are close to the target but off heading. **This is really bad and does not work.** It successfully plans paths for some targets, but if the target pose is way off to the side of the robot, this cost function falls apart. Unfortunately, given that the assignment was about parallel parking, all of the trajectories require pursuing a target that is to the side of the robot.

Additionally, my code for collision detection is way too slow for my computer. It never finishes planning because there is currently so much collision detection.

Thus, the code that I submit is a demonstration of a car planning a route from one arbitrary point to another. You can see the tree that the car explores, and then an animation of the car following that path. Hopefully this is worth some points, but I am for sure going to continue working on this.



Example Path Planned