Final Project Report: Dynamic Path Planning

Team: Flying Ninja-Geckos

Ben Zaeske - beza2015@colorado.edu
Chandler de Spirlet - chde5331@colorado.edu
Kyle Mock - kymo9890@colorado.edu

Github Repo: https://github.com/ChandlerdeSpirlet/flying_ninja_geckos_final_project

Abstract

Our final project focuses on researching and developing our own dynamic path planning algorithm based on RRT/RRT*. Our algorithm is able to find paths through complex areas and efficiently recalculate routes when moving obstacles obstruct the path of the robot.

Introduction and Background

Path planning is a widely explored field in robotics and computer science. Considering this, we thought it would be more interesting to do a path planning project by adding a layer of complexity with moving/ changing obstacles in our environment.

We decided to implement our own version of an Rapidly-exploring Random Tree Star (RRT*) algorithm. The concept behind RRT* is relatively straight-forward. Firstly, a tree is incrementally built by sampling a random point from the world and connecting it to the nearest neighboring point. When a sufficient number of points have been sampled a path between the start node and the node nearest to the goal is formed.

It should be noted that the star in RRT* denotes that the algorithm converges to an optimal solution. This is done through the use of a heuristic during the point generation phase. Making more educated guesses about the final path helps with the convergence of the problem on the optimal solution. We are tentatively calling our algorithm an RRT* variant, since we did use a heuristic to bias our tree's growth.

We used concepts from this research paper, but did not implement our algorithm following their specific methods. We found that their RRT* algorithm was a bit out of scope, as it used several statistical techniques that fell outside the time constraints of this project. That said, we did incorporate their concept of saving parts of our previous tree when recalculating our path. This is discussed in more detail in the methods section of this report.

Methods

Initial Work - Kyle Mock

We started with Python3 and Jupyter Lab for our initial development environment since it allows us to quickly build and test our different ideas for the algorithm. Later, this code would be ported over to raw Python so that it could be more easily run in the simulation environment. Both the notebook and raw python are attached with this submission.

The first milestone of the project was to get a rough version of RRT working. This started with creating an n-dimensional world representation consisting of the following:

Robot - Represents the robot in the world
Objective - Represents the objective in the world
RectangleObject - Represents a rectangle-collidable obstacle in the world
It was initially planned to have several types of objects in the world but for the simplicity of our solution this was cut back to only rectangular objects in 2D spaces.
Map - Represents the “world” of the problem. Worlds have a robot, obstacles, and an objective.

Once the world representation was completed a basic RRT algorithm was created to serve as a starting point to build on. This worked by first sampling nearby areas to the already explored tree. Once a point was found that could have an edge connected to the tree without intersecting any obstacles, it was added to the tree. This process was repeated for a defined number of iterations. At completion, the point nearest to the goal was backtraced, connecting each parent until the starting node was reached. This collection of vertices was then returned as the path.

At this stage, both static and time-based visualizations were created using Matplotlib. These visualizations were created to aid in debugging and to help present our algorithm to others.

RRT* Heuristic, Dynamic Path Planning - Ben Zaeske

Our heuristic chooses to expand the tree randomly, but there is a bias towards creating new edges that are nearer to the goal in terms of euclidean straight line distance. Specifically, our algorithm maintains a list of vertices that have been added since the edge with the minimum distance to the goal was added. New edges branch from these vertices, and when a new vertex is randomly created which is closer to the goal, the list restarts with only this new vertex. The creation of edges is still fairly random to allow for exploring. This is needed when obstacles mandate a more winding path to reach the goal. We tested several other heuristics and this was our best result in terms of both time to run the algorithm and efficiency of the final path to the goal.

To make our path planning run well with the addition of dynamic obstacles, we added a feature to our algorithm that allows you to pass it a path which was previously used to reach the same goal. If the algorithm detects that you've passed it a path, it searches the object space along the path backwards to see if there are any new obstacles which now block it. If it detects that there are, it deletes the path up until that point, saving the valid parts of the path which are connected to the goal position. The algorithm then runs its RRT* portion as discussed above, but it now tries to minimize distance to pieces of the old path, and prioritizes connecting with the old path if that is more efficient than heading to the goal.

Moving the project into Gazebo - Chandler de Spirlet

To get our algorithm to generate paths that would move a robot in gazebo, we implemented things very similarly to lab 7. We first converted our gazebo world (obstacles and world edges) into the corresponding classes that we created (mentioned above). To reduce our scope we assumed that we had a top camera which would tell us the location of obstacles. We used the /gazebo/get_model_state topic to grab information on the obstacles within the gazebo world. Once we had generated a model of our world, we passed it to our RRT* algorithm, which returns a list of waypoint vertices along a path to the goal. The idea is to have our turtlesim robot travel to each of these waypoints one at a time. Once it reaches a waypoint, it will check for any new obstacles that may have obstructed its path. If an obstruction is detected, the RRT* algorithm is rerun with the robot's new position, and we pass in the previous path we had been following. This runtime loop continues until the robot is within a threshold of the goal.

Results Discussion

Overall, we think our project saw impressive results given the timeframe of the project. We are also impressed with the runtime and computational efficiency of our RRT* algorithm.

Finding a suitable heuristic for our problem was one of the most difficult parts of the project. Given the infinite possibilities of world configurations, finding a heuristic that performed well in most of them was a challenge. This all paid off when a good heuristic was found. Before adding our heuristic we would run this algorithm with around 5,000-10,000 iterations and 10-20 seconds before it would find the goal. After implementing our current heuristic, our algorithm took only 50-1000 iterations and the execution time was reduced to the range of milliseconds.

The dynamic path portion of our algorithm works fairly well but has some room for improvement. For example, when a path needs to be reconstructed the computational efficiency is widely variable. Sometimes only 10-50 edges would be added before finding a suitable path to the goal while in some cases it could take up to 1000 iterations. The smoothness of the final path also has room for improvement. Often the path becomes windy and jagged due to the mostly random generation of the points. Path smoothing after the path generation, or an improved heuristic which favors more straight paths could serve as good continuation points for this project in the future.

Gazebo ended up giving us a lot of trouble. However, we were able to implement our RRT* algorithm into Gazebo. Our turtlesim robot follows a list of waypoints provided by the algorithm, and we were able to correctly model obstacles to pass to our algorithm. The implementation of our RRT* algorithm was successful using the /move_base_simple/goal publisher. The only aspect of the Gazebo implementation that was unsuccessful was the moving obstacle. The moving obstacle was unsuccessful in the fact that the obstacle did not have a collision attribute. The moving obstacle was not detected by the turtlesim's LiDAR sensors, however, it was represented in the raw camera feed. The moving obstacle would simply move through the robot and would not collide as the object should have behaved. Aside from the moving obstacle not having full collision properties, the Gazebo integration was successful and our RRT* algorithm works well with the turtlebot3 simulation.

Gazebo was very difficult to work with given how little time we had to get used to it, but we are happy with the algorithmic portion of our results. Team leads for the subcomponents of our project are annotated in the methods section of this report.