Annie Hernandez
Bill Hunt
Trevor Rocks

RBE 3002 - D01
April 5, 2016

Lab 3: Mobile Robot Path Planning

Introduction

This lab comprises a local implementation of A* ("A star") search, a widely-used path planning algorithm. This lab also demonstrates a method for producing visualizations of information in the RViz ROS diagnostic program.
This report is broken into subsections that explain the answers to specific questions provided by course staff.

Methodology

How do you load the map? *If the map you receive is not in 2D, what do you do about it?*

We used map_server to publish the map to the ROS topic /map. This publication comprises a ROS OccupancyGrid message, to which our planning node subscribed. While we used a pre-made 2D occupancy grid, it would be necessary in a real world application to "flatten" a complex 3D map into a meaningful 2D occupancy grid. One method for this (in an environment with a flat floor) would be to divide the 3D environment into columns extending upwards from the floor to the top of the robot. Each column could then correspond to an occupancy grid location, and the state of that location could be a function of the obstacles within the column (e.g. the maximum obstacle probability).

Another challenge arises when the occupancy grid (pre-generated or not) is not stored in real dimensions, but instead using compound parametric information. In ROS, occupancy grids are stored as arrays of dimensionless points, with metadata encoding resolution and map origin information. In order to take a point selected in the virtual environment and superimpose it within the map, or to publish graphical information for display in the virtual environment, it is necessary to transform dimensioned point data into the local, dimensionless coordinate system of the occupancy grid. In this laboratory, we created helper functions that performed each of the requisite transformations into and out of the dimensionless map frame.

Colour the map. What topics do you publish to and how do you setup RViz/?

We used GridCells and Path visualizations within RViz to produce displays of our algorthm's progress. To introduce these visualizations to RViz, we simply created new views for GridCells and Path message types, assigning them contrasting colors to facilitate easy debugging. We then configured these views to subscribe to several topics, on which our planning node published debugging information.

The colors, and topics we used are as follows:

Initial Position (GridCells) - green - lab3_visualization/initialViz
Goal (GridCells) - red - lab3_visualization/goalViz
Explored (GridCells) - pink - lab3_visualization/exploredViz
Frontier (GridCells) - blue - lab3_visualization/frontierViz
Path (Path) - green - lab3_visualization/pathViz

If the user published a new map, initial position or goal position during a search, we decided our program shouldn't start running simultaneous searches or ignore user input. Instead we placed the path planning algorithm in a terminable thread. This allowed the main thread of our planning node to monitor for user input, and restart the algorithm if new input was received.

What is your A algorithm (what are your g(n) and h(n)?) How does it work? (If not well, explain how you could do to improve on it.) How do you create waypoints?*

Our A* algorithm maintains a list of points in the frontier, and a list of expanded points. Each of these points is stored along with a "path cost", the distance travelled to get to the point. This distance is our *g(n)*. Searching starts by placing

the initial location in the frontier. The algorithm then repeatedly addresses the frontier as follows, until there is nothing left in the frontier:

1. Compute the heuristic "as the crow flies" distance to the goal for each element in the frontier (this is our *h(n)*).
2. Select the node from the frontier with the smallest combined path cost and heuristic distance (the "current" node). Remove this node from the frontier
3. Check to see if the current node is the goal; if it is, stop looping and terminate the algorithm.
4. Check to see if the current node is an obstacle; if it is, skip the rest of this loop iteration.
5. Determine each of the neighbors of the current node (both cartesian and diagonal). Form a list of neighbors, which also includes the path cost for each neighbor computed using the path cost so far, and known unit movement dimensions.
6. For each neighbor, search the frontier and expanded lists. If the neighbor is found in the frontier or expanded list, check to see whether the path cost from step 5 is less than the path cost for the neighbor in either list.
7. Add neighbors with the lowest path cost encountered so far (or which don't appear in the frontier or expanded list) to the frontier. When adding a neighbor to the frontier, store it with its path cost from this iteration, to facilitate the check performed in step 6.
8. For each neighbor added to the frontier, store in a reverse lookup list a pair relating the neighbor to the current node. Erase from the reverse lookup list any other pairs relating the neighbor to any other node.

After the frontier is empty and the algorithm terminates, it is necessary to determine the optimal path. This is facilitated by the reverse lookup list generated by step 8 of the algorithm. The path generator simply looks for the goal node in the reverse lookup list. It should be paired with the adjacent point along the best path to the goal. This point then becomes the index into the reverse lookup list, and the process continues until the initial node is reached. A similar technique is used in the pseudocode example at (https://en.wikipedia.org/wiki/A*_search_algorithm). Once the list of points along the path from the initial pose to the goal exists, it is easily translated into a Path message and transmitted on lab3_visualization/pathViz.

Conclusion

Using the A* algorithm, we were able to create a program which takes in a starting point, ending point and a map and can find and return an optimal path between those two locations while properly navigating around obstacles on the map. This is very helpful as upon receiving a map of a location our robot can now navigate around obstacles to achieve a certain pose. This has many applications in the field of robotics such a search will be required in virtually any environment to allow a robot to navigate around obstacles.

References

http://stackoverflow.com/questions/323972/is-there-any-way-to-kill-a-thread-in-python
https://en.wikipedia.org/wiki/A*_search_algorithm

Image of A* search in RViz environment on simple_map (cropped)