Programming Assignment #3

Julien Blanchet • COSC 181 • Fall 2019

Project Link: https://github.com/j55blanchet/dartmouth-cs181

Description

Technical Architecture

My design has 3 types of classes and 4 main pieces of functionality. Here, I break them down by class:

- `Path`: Represents a path that a robot could follow to get to a goal position. This takes the raw output of `PathFinder` (in the form of a backpointer dictionary) and constructs a sequence of poses upon creation.
- `PathFinder`: This component performs a graph search using A* to find an optimal path from a starting point to a goal point. It validates these inputs before performing the search.
- `TestCase`: Represents a scenario that can be automatically tested by the script. Includes the start and end points as well as the expected outcome.

Design Decisions

- **A* algorithm**: I chose the A* algorithm because it is the most efficient type of graph-search I know of that guarantees completeness and optimality.
    - Note: Euclidian distance is an admissable heuristic function because it will never overestimate the cost to get to the goal.
    - Note: For this assignment, I consider optimality to be the *shortest* path possible. It's possible that in a scenario with a physical robot, a longer path with more gradual turns would be *faster*.
- **8-way connectivity**: I opted to use 8-way connectivity during the graph search. Doing so allows the robot to take diagonal paths, which lead to more efficent routes that purely orthogonal routes. The cost for diagonal paths is sqrt(2) due to the longer distance of traveling diagonally between cells rather than vertically or horizontally.
- **Automatic Random Testing**: I opted to include both predefined tests and random tests. Predefining some tests allows for targeted testing of edge cases, whereas randomly generated cases are good for testing robustness and against unforseen scenarios.
- **Visualization Augmentations**: I opted to augment the required pose-sequence output with visualization messages specific to rviz (`visualization_msgs/Marker`) that allow us to see the path as a whole. Futhermore, I release pose messages gradually to the `pose_sequence` topic in order to animate the path visualization (rather then releasing them all at one).
    - Note that in a scenario with a physical robot, we'd want to release the poses immediately.

Evaluation

My implmementation meets all requirements of the assignment. The grid is interpreted correctly, and valid paths are pubished in the proper message type (`PoseStamped`) on the specified channel (`pose_sequence`). Furthermore, as required for graduate students, the poses are oriented in a way that reduces rotation of the robot while following the path.

Please see the table at the end of this section for information on the test cases I performed. Note that this list includes input edge cases (start=goal, off the map, edge-of-map, start or goal in an occupied space) as well as various path lengths and connectivity scenarios.

Completeness and Optimality

In all test cases I ran (both manual and randomly generated), the algorithm found an solution when a path was possible. Furthermore, among solutions that go grid-cell by grid-cell with 8-way connectivity (rather then those with unbounded travel options), the solution generated by the algorithm always appeared to be an optimal one - for example, the path never travels orthagonally when diagonals would be faster, and always hugs corners to the maximum extent possible.

Efficency

I believe the A* algorithm was the one of the better options available as far as efficency. Although I did not perform tests using BFS or DFS, there are a few reasons why I believe the algorithm I chose would outperform those. You can see that test case 9 is close to the worst case when it comes to search time, due to the fact that the target is far from the goal and many obstacles are in the way. However, the actual worst case is test case 3, where the algorithm searchs *all* nodes in the large center room in the map before the frontier is exhausted. Note that test case 9 has 25% shorter search time than test case 3 - indicative of the large number of nodes that A* did not need to process. Still, as noted in Future Work, there is room for improvement.

Test Cases

Case #	Expected	Start	Goal	Description	Search Time
1	Failure	`0.1, 0.1`	`5.5, 5.5`	Start cell is occupied	`0.0042` s
2	Failure	`0.7, 0.7`	`9.5, 9.5`	Goal cell is occupied	`0.0019` s
3	Failure	`5, 5`	`9.9, 5`	No possible paths	`0.4159` s
4	Success	`5, 5`	`5, 5`	Start point = end point	`0.0035` s
5	Success	`5, 5`	`5.0001, 5.0001`	Start in same cell as end point	`0.0010` s

Case #	Expected	Start	Goal	Description	Search Time
6	Success	`0.1, 5`	`9.9, 5`	Going around edge of map	`0.0389` s
7	Success	`9, 9.3`	`8.8, 9.1`	Short path	`0.0015` s
8	Success	`5.8, 5.65`	`9.9, 5`	Medium path	`0.3362` s
9	Success	`1, 8`	`4.5, 2.5`	Long path	`0.3051` s
10	Success	`0.02, 0.9`	`0.9, 0.01`	Points on border of map	`0.0027` s
11	Failure	`-1, 5`	`5, 5`	Start cell at negative column	`0.0016` s
12	Failure	`5, 5`	`5, 15`	Goal cell beyond grid row	`0.0011` s

To see image captures of these test cases, look at the accompaning `.png` files.

Future Work

- Interactivity: subscribe to `/clicked_point` and use those points to allow tester to specify start and goal locations.
- Condense path: if multiple sequential poses lie along a line, the intermediate poses are unnecessary; just the start and end poses would suffice.
- Enforce a minimum distance between path and occupied cells to take into account physical dimensions / clearance requirements of path-following robot.
- Path smoothing: after "rough" path is constructed using current methods, make it easier to follow by smoothing out curves in path and identifying free areas were direct straight lines may be followed.
- Enhance rviz visualization in edge cases and failures. Can send visualization messages with text indicating failure reasons or data. Could also public path metadata (distance, a metric for directness / complexity, etc.) to a different topic.
- Generalize pathfinder to support multiple search algorithms (instead of just A*). Candidates:
    - BFS, DFS
    - Greedy BFS / DFS
    - Dijikstra, etc.
- Enhance search algorithm to model occupancy with geometrical primitives and use raycasting or other methods to identify a smaller set of important potential paths.
- Enhance launchfile to also run a `turtlesim_node` that makes use of the map and executes the path. Can position turtlebot using ROS services and control bot using another node that subscribes to path planner.

Allocation of Effort

All coding, mathmatical calculations, and documentation on this project was done independently. I consulted various online programming resources while working on this assignment - citations are in the relevant comments. I also discussed high-level strategy and approaches for this assignment with peers from class and the robotics lab.

Sources Consulted

See `src/pa3/README.md`