Lab 5: Path Planning

CSCI 3302: Introduction to Robotics

Code+Report due 11/25/20 @ 23:59pm

(Early turn-in bonus: +2pts extra credit per day, max of +10)

The goals of this lab are

- Implement a shortest-path finding algorithm (Dijkstra's algorithm) on a grid map
- Use path planning to generate a discrete set of waypoints for a robot to follow
- Showcase our ability to do complex planning by having E-puck plan and execute a collision-free path via the merging of high-level planning and low-level control algorithms

You need

- Lab 5 Webots starter code
- An understanding of Lab 3 (IK Controller), Lab 4 (Mapping), Dijkstra's Algorithm, and State Machines

Dijkstra's algorithm takes a graph with N nodes as an input and returns the minimum cost of reaching every vertex when starting from a user-specified source vertex. Using this information, we can find a path of vertices connecting a start location to a goal location. In order to make use of it, we have to convert the vertices on the path into coordinates we can have the robot travel to. Recall that with our inverse kinematics-based feedback controller from Lab 3, we can use a desired goal pose to figure out the wheel rotations that make the robot travel there! Therefore, combining these two ideas will allow us to use path planning on a graph to enable a robot to traverse a physical, continuous space.

Part 1: Path Planning

1. Modify the **get_travel_cost** function to incorporate the **world_map** into its cost computation, incurring a high cost if a given cell/vertex is occupied (indicated by a value of 1 in the map) or if the two cells are not neighbors. Otherwise, return a cost of 1 if the cells are neighbors, and 0 if the source and destination are the same.
(world_map is an $n \times m$ matrix, where n, m can be found with world_map.shape)

2. Implement Dijkstra's Algorithm in the function **dijkstra**, computing the shortest paths from **source vertex** to all other vertices, making sure to use your **get_travel_cost** function.

3. Implement **reconstruct_path**, which uses a caller-provided goal vertex (in map i, j coordinates – not world x, y coordinates) and the "prev" data structure generated by **dijkstra** to return a list of vertices leading from **source vertex** to **goal vertex**.

4. Implement **visualize_path**, which uses the output of **reconstruct_path** to set each vertex along the robot's path in world_map to 2, the goal vertex to 3, and the starting vertex to 4, to allow us to see the robot's full intended path on the map before execution.

Part 2: Path Following

1. Create a state machine in your while loop with the following properties:
   (Your state machine can have more states if desired, but this should be a good start!)

   a) The "get_path" state will compute a path from the robot's current vertex (determined using its odometry data) to a given goal vertex (determined using the provided target_pose variable).
      - Set the current overall goal vertex (using target_pose) from your map
      - Check whether the generated path reaches back to the source vertex – if it doesn't, you shouldn't transition into another state!
        (If the robot has no viable path to the goal, it won't have a real path to follow.)
      - Call **visualize_path** with the path you get from your planning algorithm

   b) The "get_waypoint" state will assign a goal pose (x, y, θ) for the robot to move toward based on the next waypoint (vertex coordinates) in the path.
      - Make sure that the selected goal pose corresponds to the world coordinates of the next vertex in the path.
        (To compute a goal θ for a given waypoint, try to point the robot in the direction of the waypoint after this one. If you have no future waypoints to follow, use target_pose[2] as your goal θ)
      - If there are no more waypoints left, transition to get_path.
      - Otherwise, transition to state "move_to_waypoint"

   c) The "move_to_waypoint" state will use the Inverse Kinematics-based **get_wheel_speeds** function with the motor.setVelocity functions to move to the currently assigned goal pose
      - Once the robot is at the assigned goal pose, the robot should stop moving and transition back to the "get_waypoint" state
      - Make sure to pick reasonable margins for distance and heading error when checking the odometry values against your goal waypoint to see if you're at your desired position!

2. Test that your implementation works by moving the obstacles and/or target around!

Part 3: Lab Report

1) What are the names of everyone in your lab group?
2) Provide an illustration of the state machine you've implemented. Be sure to include all transition criteria!
3) As E-Puck is traveling along the path that Dijkstra's algorithm provided, it detects that a cell marked as "empty" actually has an object in it. How could you modify your solution to plan around/gracefully handle this unforeseen object in the robot's path?
4) Could we use an algorithm like RRT to generate a viable path instead of Dijkstra's algorithm? If yes, how would the path look different? If no, why not?
5) Roughly how much time did you spend programming this lab?