Sheet 11 solutions

September 9, 2021

Global (Path-) Planning

Graph-search algorithms like Dijkstra or A^* can be used to plan paths in graphs from a start to a goal. If the cells of a grid map are represented as vertices of a graph with edges between the neighboring cells, graph-search algorithms can be used for robot path planning. For this exercise sheet we consider the 8-neighborhood of a cell $\langle x, y \rangle$, which is defined as the set of cells that are adjacent to $\langle x, y \rangle$ either horizontally, vertically or diagonally.

You can find an implementation of graph-based 2D path planning in the planning_framework tarball provided on the website. Complete the missing pieces following the instructions below.

Exercise 1: Dijkstra Algorithm

The Dijkstra algorithm can be used to calculate minimum cost paths in a graph. During search, it always chooses the vertex from the graph with the lowest cost from the start and adds its neighboring vertices to the search graph.

(a) *Let $M(x, y)$ denote an occupancy grid map. During search, the grid cells are connected to their neighboring cells to construct the search graph. Complete the function `get_neighborhood` in the provided planning framework. The function takes the coordinates of a cell and returns a $n \times 2$ vector with the cell coordinates of its neighbors, considering the boundaries of the map.*

```
def get_neighborhood(cell, occ_map_shape):
    """
    Arguments:
    cell -- cell coordinates as [x, y]
    occ_map_shape -- shape of the occupancy map (nx, ny)

    Output:
    neighbors -- list of up to eight neighbor coordinate tuples [(x1, y1), (x2, y2), ...]
    """
```

```
    neighbors = []
    for i in [-1, 0, 1]:
        for j in [-1, 0, 1]:
            x = cell[0] + i
            y = cell[1] + j

            if x < 0 or x >= occ_map_shape[0]:
                continue
            if y < 0 or y >= occ_map_shape[1]:
                continue
            if i == 0 and j == 0:
                continue

            neighbors.append((x, y))
    return neighbors
```

(b) *Formulate a function for the edge costs between two cells that allows for planning of the shortest collision free path on the grid. Include occupancy information in your edge cost function to prefer cells with low occupancy probability over cells with higher probability. Regard a cell as an obstacle if its occupancy probability exceeds a certain threshold. Which threshold would you choose? Implement this function in* `get_edge_cost`.

An obvious threshold for defining obstacles would be an occupancy probability of $p \geq 0.5$, which is the prior for unobserved cells. However, a more conservative value might be a better choice in practice.

```
def get_edge_cost(parent, child, occ_map):
    """
    Calculate cost for moving from parent to child.

    Arguments:
    parent, child -- cell coordinates as [x, y]
    occ_map -- occupancy probability map

    Output:
    edge_cost -- calculated cost
    """

    occ = occ_map[child[0], child[1]]
    if occ >= .5:
        return np.inf
    edge_cost = np.linalg.norm(parent - child)
    return edge_cost + 10 * occ
```

(c) *Implement the update step of the Dijkstra algorithm in* `run_path_planning`. *For the current parent node, consider all of its neighbors and calculate their tentative distances from the start location (cost) and their predecessor in the grid. Under*

which condition should the update be done? You are now ready to run the Dijkstra algorithm with `python planning_framework.py`.

```
        # update costs and predecessor for neighbors
        neighbors = get_neighborhood(parent, occ_map.shape)
        for child in neighbors:
            child_cost = costs[x, y] + get_edge_cost(parent, child, occ_map)
            if child_cost < costs[child]:
                costs[child] = child_cost
                predecessors[child] = parent
```

Exercise 2: A^* Algorithm

The A^* algorithm employs a heuristic to perform an informed search with higher efficiency than the Dijkstra algorithm.

(a) *What properties of the heuristic are required to ensure that A^* is optimal?*

To find the optimal path, the heuristic function must be *admissible*, meaning that it never overestimates the actual cost to get to the goal. Also, it must be *consistent*, meaning that for every node, the estimated cost of reaching the goal from that node must be no greater than the estimated cost of reaching its successor, plus the edge cost between node and successor.

(b) *Define a heuristic for optimal 2D mobile robot path planning. Complete the function* `get_heuristic` *in the planning framework. The function takes the coordinates of a cell and the goal and returns the estimated costs to the goal. You are now ready to run the A^* algorithm with* `python planning_framework.py`.

The Euclidean distance gives a lower bound for the cost and therefore is a good heuristic. It is equivalent to the actual cost in case of a straight path along cells with zero cost.

```
def get_heuristic(cell, goal):
    """
    Estimate cost for moving from cell to goal based on heuristic.

    Arguments:
    cell, goal -- cell coordinates as [x, y]

    Output:
    cost -- estimated cost
    """

    heuristic = 1 * np.linalg.norm(cell - goal)
    return heuristic
```

(c) *What happens if you inflate your heuristic by using h_2, which is a multiple of your defined heuristic h? Try different multiples: $h_2 = \{1, 2, 5, 10\} \cdot h$*

The larger the heuristic, the more the algorithm is forced to choose a path as close as possible to the goal. Less exploration is done, which leads to a faster termination of the algorithm. However, the algorithm is not guaranteed to find the shortest path anymore. This is the case with a multiple of 5 or 10 in the example.