CS 695: Programming assignment (P2)

Md. Ridwan Hossain Talukder
Department of Computer Science
George Mason University
Fairfax, USA
mtalukd@gmu.edu

I. A* for Sliding Puzzles

A. Implementing A* Search

1) Answer: Yes, *sliding_puzzle_zero_heuristic* is an admissible heuristic as it always underestimates the assumption and using this heuristics makes the A* is equal to UCS and UCS is optimal.

B. Heuristics for the Sliding Puzzle

TABLE I
HEURISTICS FOR THE SLIDING PUZZLE

Planner	Time	Path Len	NumItr
No Heuristic	0.21677422523498535	21	5079
Num Incorrect Heuristic	0.05820870399475098	21	1189
Manhattan Distance Heuristic	0.06675267219543457	21	289
Num Incorrect Squared Heuristic	0.005792856216430664	23	122
Manhattan Distance Squared Heuristic	0.03058004379272461	37	128

1) Answer 1: Manhattan Distance Heuristic reaches the goal in fewer iterations. It will be true for any puzzle because the Manhattan Distance heuristic (h1) approximates the actual distance better than the Number of Incorrect tiles heuristic(h2). Where h2 doesn't take into account how far away that tile is from being correct, h1 does take this information into account. So, the lowest the Manhattan distance heuristic can possibly be is equal to Number of Incorrect tiles heuristic.

2) Answer 2: The Manhattan Distance Heuristic and the Number of Incorrect tiles heuristic are admissible because they produces optimal solution and always underestimates the approximation. A non-admissible heuristic is not guaranteed to produce an optimal solution, so using this heuristic to plan will not result in a shortest (optimal) path solution. The path lengths for admissible heuristics are smaller (NI:21, MD: 21) than the non-admissible (NIS: 23, MDS: 37) heuristics. Number of iterations for admissible heuristics (NI:1189 MD:289) are greater than non-admissible (NIS:122, MDS:128) heuristics. The "squared" versions heuristics reach the goal faster but do not result in optimal path where the original heuristics find the optimal path but number of iterations are higher. This is because non-admissible heuristics can reach the goal in shortest time (low number of iterations) but not having an optimal path length as it has overestimated the approximation and reached the goal earlier but missed the shortest path to the goal.

II. Monte Carlo Sampling

A. Computing Area with MC Sampling

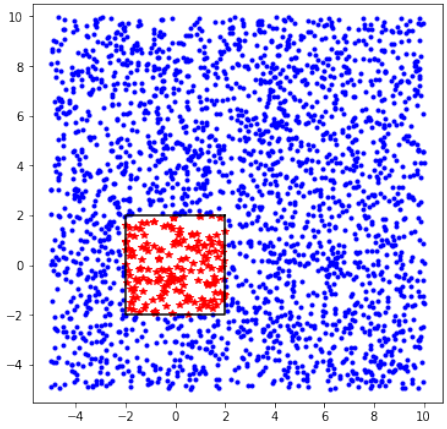


Fig. 1. Monte Carlo Sampling for Square Environment (2500 Samples)

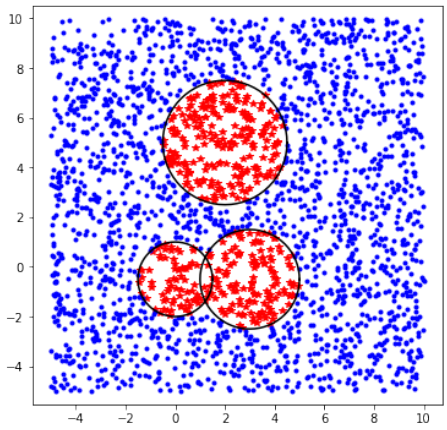


Fig. 2. Monte Carlo Sampling for Multi Environment (2500 Samples)

TABLE II
COMPUTING AREA WITH MC SAMPLING

	Area	
Samples	Square	Multi
100	13.5	45.0
1000	13.05	40.05
10000	15.5475	37.4625
100000	16.15725	38.448

TABLE III
ACCURACY OF THE MEASUREMENT (COMPARED TO THE TRUE VALUE) FOR THE AREA OF SQUARE

Samples	Difference
100	2.5
1000	2.95
10000	0.4525
100000	0.15725

1) Answer: The measurement gets closer to the true value as the number of samples increases by a large margin. And for 100000 samples the measurement is almost same to the true value.

B. More MC Sampling

1) Answer: The likelihood is 0.2338 or 23.38% that a random unit-length line (of random orientation) contained within x=[-1, 1] and y=[-1, 1] does not intersect a box centered at the origin with side length 1. (For 10000 iterations)

III. PROBABILISTIC ROAD MAPS

1) Answer: If we look at the figure 4, 5 and 6 we will see that as the number of sample point increases the PRM finds a shorter path length to reach the goal. Because in figure 5 there were no sample point close to the obstacle's (bottom right) edge and thus the planner went around the obstacle where in figure 4 there was a point near the edge (on the edge of the corner) which contributed towards reducing the path length and making a shorter straight path. In figure 6 more points were added and a more straight and shorter path was found. As PRM is a sampling based planner the more the sample points will be there the probability of getting a shorter path will increase, and adding a significant amount of sample points will significantly reduce the path length that's the reason I think why there is a distinct "drop" at one or two places in the plot of figure 7.

IV. RRT AND RRT*

A. Building a (random) chain

1) Answer: The function steer_towards_point checks if the distance between the given point and given link is within the step size (if it's not it resizes the distance coordinates by increasing the dx and dy) and creates a new link with that point where the given link becomes the upstream of that new link (which is similar to the concept of steering towards that point). The function naturally supports the idea of having vehicle dynamics. We simply compute the control input that best moves the vehicle in the direction of our randomly sampled

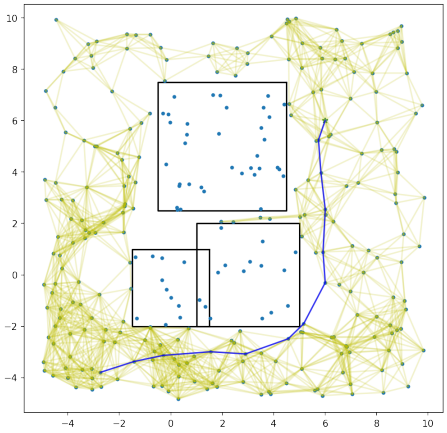


Fig. 3. PRM graph for 300 sample

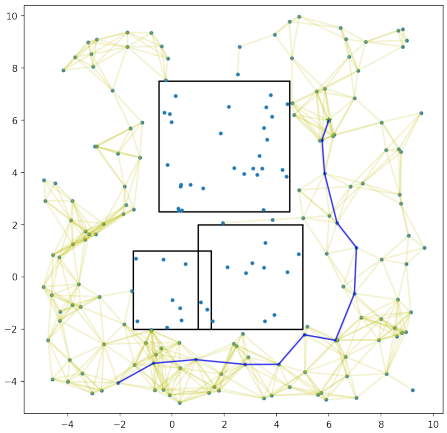


Fig. 4. PRM graph for 200 sample

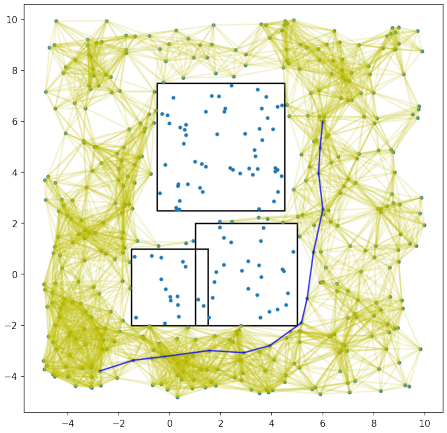
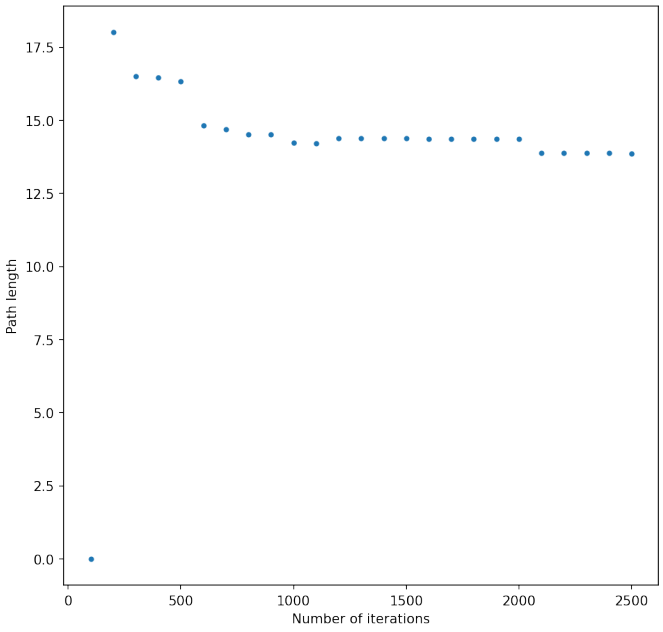


Fig. 5. PRM graph for 500 sample



point. That means that every branch of the tree is automatically dynamically feasible.

2) Answer: The algorithm randomly chooses a point and adds link to that point with the latest link (if it does not collide with any obstacle) which makes it act like a tree, where we start from the start point and randomly add new actions to that tree. Which is not a smart way to expand a tree. Though this planner eventually reaches the goal (if we keep increasing the number of iterations it will eventually reach the goal avoiding obstacles), it will never find an optimal solution (as it does not correct the edges). So this planner is complete but not optimal. Is it asymptotically optimal? No, it's not.

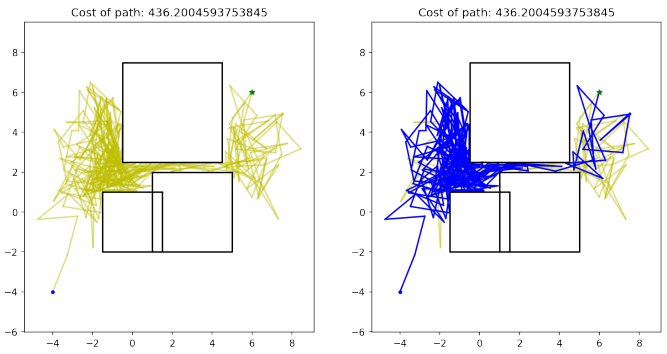


Fig. 7. Plot Random Link Chain

B. RRT

1) Answer: From the results we can see that the path length to goal increases as the number of iteration increases. If we look at the algorithm, it defers from the Random Link Chain Algorithm in one case, where instead of choosing the latest link we choose the closest link to that new point. This property helps the tree to grow towards finding the goal in less number of iterations than RLC. Also the same property makes it not optimal. Because whenever we add new points, we extend old branches but we never look for new ways to reach the same point that might have been shorter. So this planner is complete but not optimal. Is it asymptotically optimal? No, it's not. The theoretical claims match up with the results because from the results we see the cost increases (instead of decreasing) as the number of iteration increases.

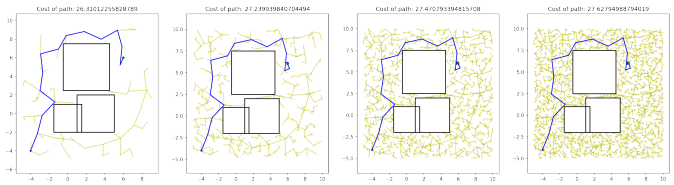


Fig. 8. Plot RRT

```python
def RRT(start, goal, obstacles, region_x, region_y,
                                num_iterations,
    step_size, seed=695):
    random.seed(seed)
    links = [Link(start)]
    for _ in range(num_iterations):
        # Generate a random point
        px = random.uniform(region_x[0], region_x
    [1])
        py = random.uniform(region_y[0], region_y
    [1])
        point = [px, py]

        # Get the closest link
        latest_link = links[-1]
        min_dist = float('inf')
        for link in links:
            if link.get_distance(point) < min_dist:
                latest_link = link
                min_dist = link.get_distance(point)

        # Steer towards it
        new_link, new_point = steer_towards_point(
    latest_link, point, step_size)

        # If it collides, return
        if new_link.does_collide(obstacles):
            continue

        # Add to chain if it does not collide
        links.append(new_link)

    return links
```

C. RRT*

1) Answer: In RRT*, unlike RRT whenever we add new points, we check if that new point allows us to reach other points sooner means that if the route through the new point to the nearby point is shorter than the old path, we change the "root" of the nearby point to the new point and update the cost. So the new links are optimized and now if any path to goal exists, RRT* will reach it and it will be asymptotically optimal as the cost will always be lower. The theoretical claims match up with the results because the cost decreases as the number of

iteration increases that means with higher number of iterations (tends to infinity) it gets closer to finding the shortest path.

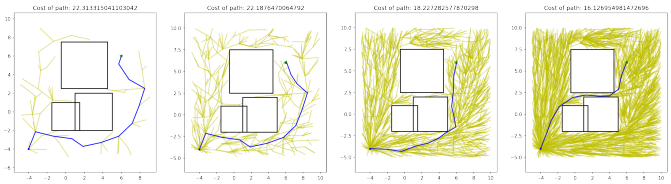


Fig. 9. Plot RRT*

```
def RRTstar(start, goal, obstacles, region_x, region_y,
                                num_iterations, step_size, seed=695):
    random.seed(seed)
    links = [Link(start)]
    radius = step_size
    for _ in range(num_iterations):
        # Generate a random point
        px = random.uniform(region_x[0], region_x[1])
        py = random.uniform(region_y[0], region_y[1])
        point = [px, py]

        # Get the closest link
        latest_link = links[-1]
        min_dist = float('inf')
        for link in links:
            if link.get_distance(point) < min_dist:
                latest_link = link
                min_dist = link.get_distance(point)

        # Steer towards it
        new_link, new_point = steer_towards_point(latest_link, point, step_size)

        if new_link.does_collide(obstacles):
            continue

        neighborhood = []
        nearest_neighbor = None

        for link in links:
            if link.get_distance(new_point) <= radius:
                neighborhood.append(steer_towards_point(link, new_point, step_size)[0])

        min_dist_2 = float('inf')

        for neighbor in neighborhood:
            if neighbor.cost + neighbor.get_distance(new_point) < min_dist_2:
                min_dist_2 = neighbor.cost
                nearest_neighbor = neighbor

        # If it collides, return
        if nearest_neighbor == None or nearest_neighbor.does_collide(obstacles):
            continue


        new_neighborhood = []
        for neighbor in neighborhood:
            if neighbor.upstream is not nearest_neighbor.upstream:
                new_neighborhood.append(neighbor.upstream)


        for neighbor in new_neighborhood:
            alt_link, alt_point = steer_towards_point(neighbor, nearest_neighbor.point, step_size)
            if not alt_link.does_collide(obstacles) and alt_link.cost < neighbor.cost:
                neighbor.upstream = neighbor

        # Add to chain if it does not collide
        links.append(nearest_neighbor)

    return links
```