Project 2 - A^* and Sample-Based Planning

Hoang-Dung Bui,
George Mason University
Fairfax, USA
hbui20@gmu.edu

I. A^* for Sliding Puzzles

A. Implementing A^* Search

Question: The code block to check your A* starch uses a heuristic function I have provided you with, sliding_puzzle_zero_heuristic, that returns 0 no matter the inputs. Is this an admissible heuristic? Explain why or why not?
Answer: The heuristic function *sliding_puzzle_zero_heuristic* is admissible, because it meet the requirement of admissibility: it is underestimate the actual distance from the current position to the goal.

B. Heuristics for the Sliding Puzzle

As running the four heuristic functions, the results are shown in Table I.

	$Zero$	$Incorrect$	$Man.$	$Incorrect^2$	Man^2
Time	0.258	0.082	0.197	0.0098	0.051
Iterations	5079	1189	1354	122	271
Path Len	21	21	21	23	45

TABLE I: Results of five heuristic functions

Question: Which of your two heuristics (not the squared one I provided you with) reaches the goal in fewer iterations. Will this be true (or at least close enough) for any puzzle? Explain why.
Answer: Between *Incorrect tiles* and *Manhattan distance*, the former one provided a better results with 1189 iterations (comparing with 1354) for size of 4x2. But Manhattan distance takes advanced as we change puzzle with size 3x3, 2x4. In my opinion, as the sizes increase, *Manhattan distance heuristic function* goes closer to the true path length than the *incorrect tiles* one.
Question: Which of the four heuristics are admissible? What happens when you use a non-admissible heuristic to plan? How do the path lengths compare between the admissible and the non-admissible heuristics? How do the number of iterations compare between your heuristics and the ”squared” versions I provided you with? Why does this happen?
Answer: In four heuristic functions, the *incorrect tiles* and *Manhattan distance* functions are admissible. Both square functions are non-admissible, and they make the A^* not able to provide the optimal paths. From the table, we can see that *incorrect tiles square* and *Manhattan distance square* functions provide with 23 and 45 path lengths. However, their iteration's numbers are much smaller than the two regular heuristic functions. In my opinion, the non-admissible functions force the A^* algorithm to find a solution as fast as possible, therefore the algorithm will miss the optimal solution.

II. Monte Carlo Sampling

A. Computing Area with MC Sampling

The two environments with the different obstacles with points are shown in Fig. 1

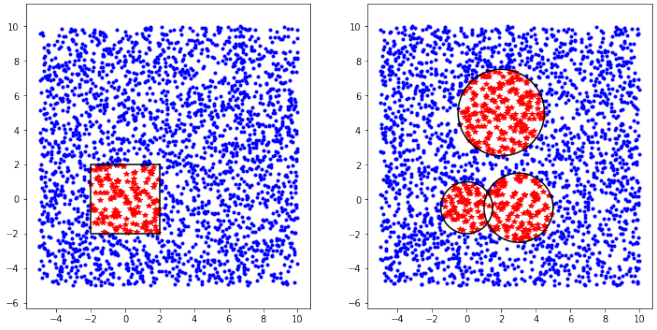

Fig. 1: Two environments with obstacles

Results/Question: Compute the area for both environments after 100, 1000, 10000, and 100000 samples and include them in your writeup.
Answer: The number of points inside and outside the obstacles in two environments are shown in Table II.

	Sample points			
	100	1000	10000	100000
Points inside - square	8	66	715	6955
Points outside - square	92	934	9285	93045
Obstacle area	18.0	14.85	16.09	15.65
Points inside - multi	26	174	1696	17175
Points outside - multi	74	826	8304	82825
Obstacle area	58.5	39.15	38.16	38.64

TABLE II: Number of points inside and outside the obstacles

For the square environment, as the number of sample increases, the square area's approximation goes closer to the true value of the area (15.65). There is still a difference in the true value (16.00), because in this computation, all the point on the boundary is considered outside the obstacles.

B. More MC Sampling

In this task, we use the idea of MC sampling to determine the probability of a line intersecting the obstacles. There is an environment - a box with dimension $x \in [-1.0 : 1.0]$ and $y \in [-1.0 : 1.0]$, and there is an obstacle at the center with

dimensions $x \in [-0.5 : 0.5]$ and $y \in [-0.5 : 0.5]$. If we just draw a random line with length 1 inside the environment, the probability for it not intersecting the obstacle is 0.2338.

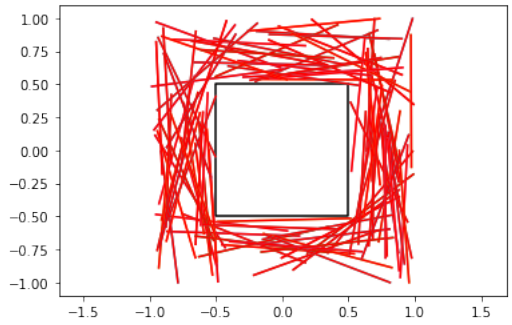

Fig. 2: The lines do not intersect the box at the center

III. Probabilistic Road Maps

In this task, we will construct a simple probabilistic Roadmap (PRM). The algorithm provides a path connecting the starting point and the goal in Fig. 3

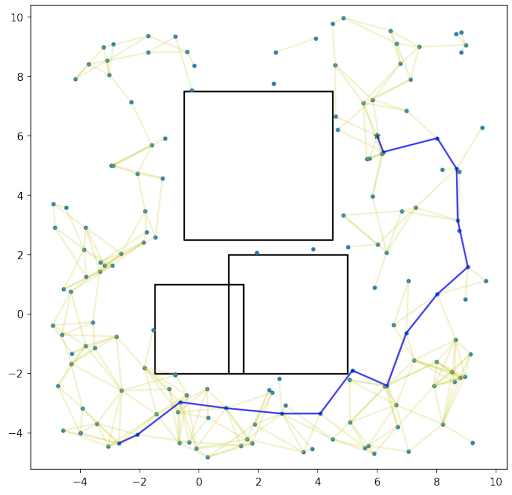

Fig. 3: PRM and the solution

The table III lists the path's lengths (use the variable *distance*) as we increasing the number of sample from 300 to 2400.

Sample	300	400	500	600	700	800	900
Path Len.	17.69	17.69	17.34	17.29	16.53	16.53	17.02
Sample 1000	1100	1200	1300	1400	1500	1600	1700
Path Len.	16.92	14.85	14.62	14.01	13.99	13.97	13.95
Sample	1800	1900	2000	2100	2200	2300	2400
Path Len.	13.71	13.71	13.71	13.71	13.71	13.71	13.56

TABLE III: Path Lengths as number of samples are increasing

Question: In your results, you should notice a distinct "drop" at one or two places in the curve of path length versus num_iterations. Why do these drops occur? Feel free to include pictures if you feel they can help you answer the question.
Answer: As increasing the sample's number from 800 to 1000, there is a little increasing the path lengths, and then go down again. In my opinion, that is due to the random of milestones. Sometimes, the random picking the milestones is not optimal, and too far to the optimal path, and it could increase a little bit the path length. As we have enough the sample points, the path length change very little (number of sample points is greater than 1800).

IV. RRT and RRT*

In this part of this project, we will familiarize with RRT and RRT^* algorithms.

A. Building a (random) chain

Question: What does the function *steer_towards_point* do? How would this function need to change if the vehicle had dynamics (and could not simply move in the direction of the new point)?

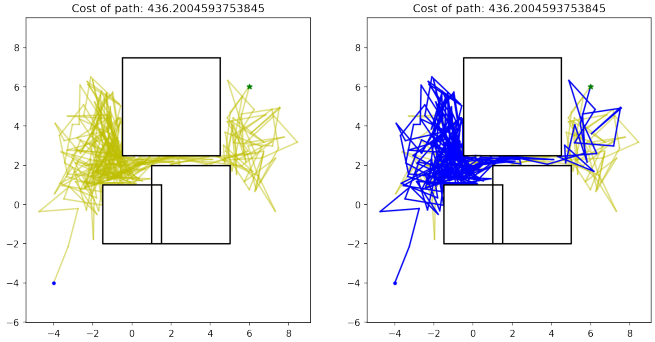

Fig. 4: RRT with random chain

Answer: The function *steer_towards_point* will determine how much the tree will be expanded toward the random point. If there are multiple obstacles in the environment, this step will ensure that we will never collide or go inside an obstacle.

If the vehicle has dynamics, this function can be used to find how the segment can be shaped. It need more data from the tree's node and its upstream, to compare with the node and the random point's orientation. If there is conflict between two orientation, the change could set equal to zero (means that the vehicle is impossible to move.)
Question: Describe the results and discuss the algorithmic properties of the algorithm. Is this planner complete (will it always eventually find a path to the goal)? Is it asymptotically optimal (will that path approach the shortest path in the limit of infinite samples)?
Answer: With random chain, the algorithm is a mess. We can connect the random points with any vertex in the tree, and it makes the path is very long. But I think it is complete. As increasing the number of sample point, we will finally find a point close to the goal and connect it to the tree. Off course, the path is never asymptotically optimal.

B. Implementing RRT

Question: Describe the results and discuss the algorithmic properties of the algorithm. Is this planner complete (will it always eventually find a path to the goal)? Is it asymptotically

optimal (will that path approach the shortest path in the limit of infinite samples)?

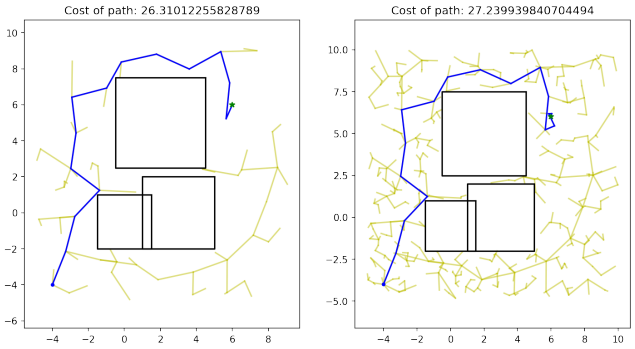

Fig. 5: RRT with 100 and 500 sample's points

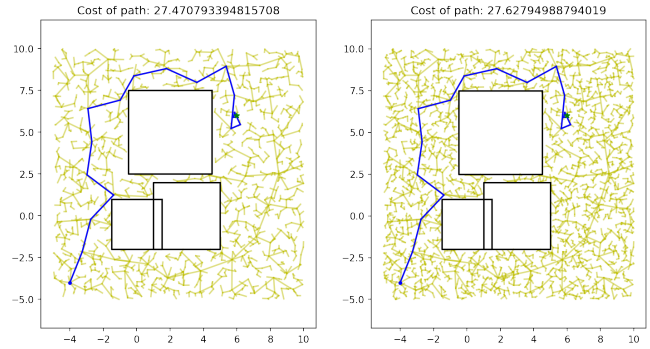

Fig. 6: RRT with 2000 and 4000 sample's points

Answer: The RRT algorithm can find out a path if it is exist, however, it fails to find the optimal one even the number of sample is increased significantly. That is because as adding more sample's points, the path is still the same. As the number of sample's points increases, we can have better path, however, the algorithms is not allowed to make adjustment, so it will stay with the found one. This makes the algorithm not asymptotically optimal. We can see it clearly in the four scenarios in Fig. 5 and 6.

Code: Comparing to RRT with random Chain, the RRT find the best tree node to connect with. In the added code, we run through all the link set, and find which link is closest to the random point. In this case, we use *Euclidean distance* to compare. After finding out the closest node, the algorithm just add a line with length of stepsize from the closest node toward the random point.

```
# Get the closest link
min_dist = 2000.0
nearest_link = links[-1]
for link in links:
  dist1 = link.get_distance(point)
  if dist1 < min_dist:
    min_dist = dist1
    nearest_link = link
```

Fig. 7: Added code in RRT

C. *RRT**

The results of RRT^* algorithm is shown in Fig. 8 and 9 with the number of samples increasing to 100 to 4000 points.

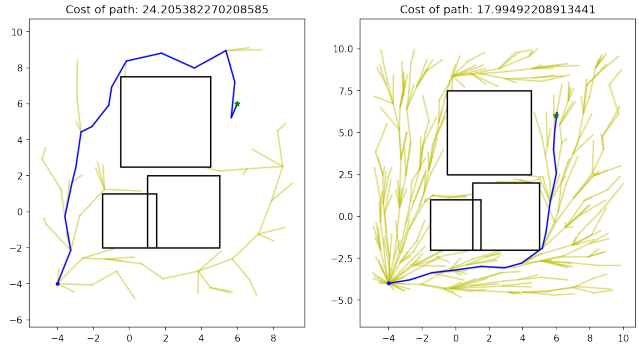

Fig. 8: Paths generated by RRT* with 100 and 500 samples

Question: Describe the results and discuss the algorithmic properties of the algorithm. Is this planner complete (will it always eventually find a path to the goal)? Is it asymptotically optimal (will that path approach the shortest path in the limit of infinite samples)? How do your theoretical claims match up with the results (from your plots above)?

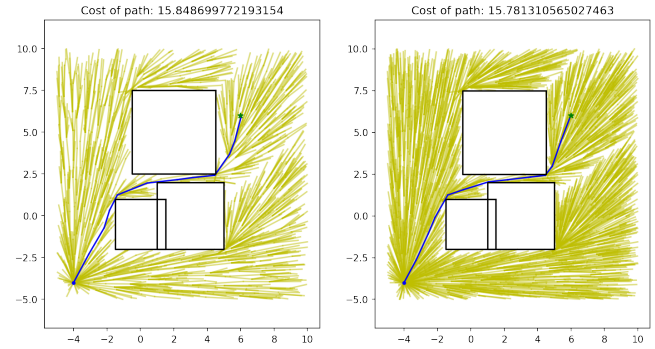

Fig. 9: Paths generated by RRT* with 2000 and 4000 samples

Answer: As we can see that, as the number of sample increases, the found path is smoother and shorter. It is reduced from 24.2 to 17.9, 15.85 and 15.78 as we add more samples into the map. In my opinion, the RRT^* algorithm is complete and asymptotically optimal. Because, as the number of sample goes up, we will then find a path to connect the start and the goal and the path will be shorter and smoother. The reason is, with a new node, we will adjust the current path and make it shorter to the goal. It is completely fit with the experimental results.

Code: The difference between RRT and RRT^* is in the following code (Fig. 10). After adding the new link into the links set, we will check what is the optimal way of connecting it to the tree. In the first *for* loop, we find its neighbors (whose distances to it is less than a step size). In the second *for* loop, we check in the set, which vertex is connected the the new vertex and provide the shortest path. After finding the optimal one, we connect it with the new vertex, and update the cost between two vertices. In the third step, we check with the new

node, can we reduce to the cost of its neighbor. In the last *for* loop, again, we run through the neighbor set, and compare the current cost of the neighbor nodes to the cost if it is connected to the new node. If the current cost is more, we remove the connection between the neighbor nodes and theirs parents, then link them with the new node, and update the new cost to the neighbor node. With this step, we will assure that the path will be asymptotically optimal.

```
# Add to chain if it does not collide
links.append(new_link)
# find the near points set
points_near = []
for link in links:
  dist1 = link.get_distance(point)
  if dist1 < step_size:
    points_near.append(link)

min_cost = new_link.cost
min_point = nearest_link
# Adjust the new point's parent
for point_near in points_near:
    line2 = LineString([new_point, point_near.point])
    cost = point_near.get_distance(new_point)
    # check collision
    if any(line2.intersects(obstacle) for obstacle in obstacles):
      continue
    else:   # Adjust the new point's parent
      if min_cost > point_near.cost + cost:
        min_point = point_near
        min_cost = point_near.cost + cost
new_link.upstream = min_point
new_link.geom_line = LineString([new_point, min_point.point])
new_link.local_cost = min_point.get_distance(new_point)

for point_near in points_near:
    line3 = LineString([new_point, point_near.point])
    cost = point_near.get_distance(new_point)
    # check collision
    if any(line3.intersects(obstacle) for obstacle in obstacles):
      continue
    else:   # Adjust the paths
      if point_near.cost > new_link.cost + cost:
        point_near.upstream = new_link
        point_near.geom_line = line3
        point_near.local_cost = cost
```

Fig. 10: Added code in RRT*