Project 2 - A^*, Dynamics, and Sampling

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

I. Using A* with Motion Primitives

A. A* in a small map

As running the A* motion planner without any heuristic, it gets stuck and cannot find a path to the goal. (Fig. 1).

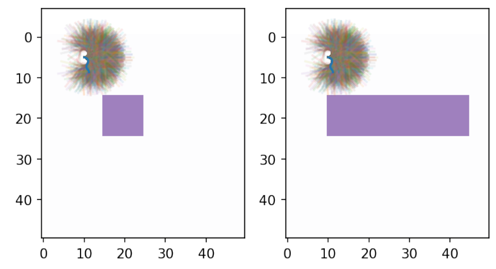


Fig. 1: A^* without heuristic

TASK: As implementing the $heuristic\_fn\_cartesian$ function (provide the Euclidian distance between the start and the goal), the A* motion planner uses the heuristic function, however, it still is not able to find a path to goal with obstacle-free as shown in Fig. 2.

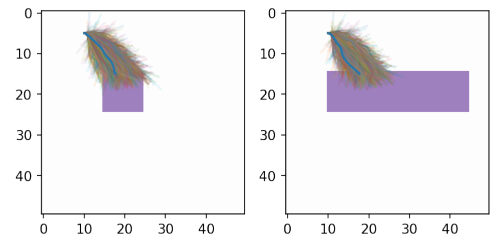
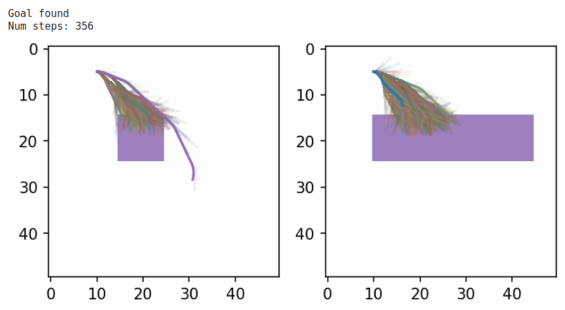


Fig. 2: A^* with heuristic

PLOT: Run the A* planning code with a Cartesean Distance heuristic multiplied by 1.2 (this heuristic is not admissible anymore), it can find a way to overcome the square obstacle, but it still faces challenge to handle the rectangular obstacles as shown in Fig. 3.

Question How can $goal\_distance\_cost\_grid$ be used to implement a heuristic for robot navigation? Explain how this would work and whether or not your heuristic is an admissible one (and why).

Answer: $goal\_distance\_cost\_grid$ can be used as a heuristic for robot navigation. $goal\_distance\_cost\_grid$ consists of cells which store distances from those cells to a goal. The distances are the sum of the distance's segments from the cells to its neighbors toward its goal. The whole process bases on the distances from the cells its available neighbors. If the neighbors are limited, this heuristic will not be an admissible one.



```
Goal found
Num steps: 356
```

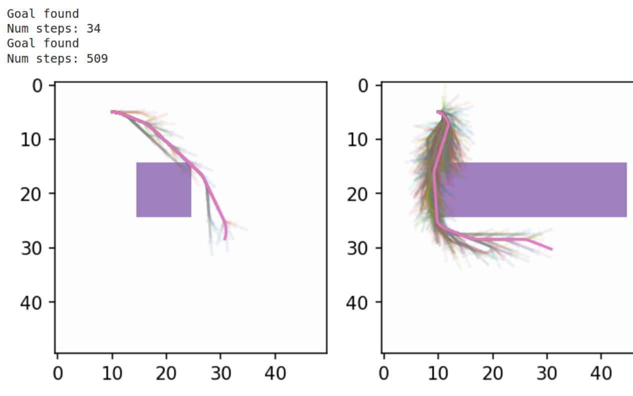
Fig. 3: A^* with 1.2*heuristic

TASK The $heuristic\_fn\_cost\_grid$ function is implemented as shown in Fig. 4. We define the mcp object with the $cost\_grid$ as input. Then we calculate the cost to the goal from all the points by the $find\_costs$ function. Finally, we select the start index from the matrix and return as the heuristic value.

```python
def heuristic_fn_cost_grid(robot_pose, goal_pose):
    mcp = skimage.graph.MCP_Geometric(cost_grid)
    goal_dis_cost_grid = \
            mcp.find_costs(starts=[[goal_pose.x, goal_pose.y]])[0]
    return goal_dis_cost_grid[int(robot_pose.x), int(robot_pose.y)]
```

Fig. 4: Code of $heuristic\_fn\_cost\_grid$

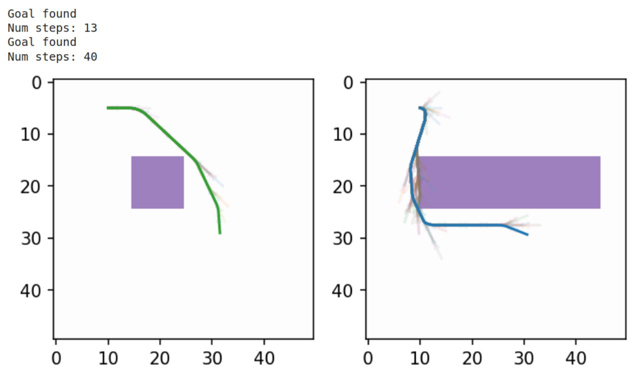
PLOT As running the A* motion planner with the $heuristic\_fn\_cost\_grid$ heuristic, the motion planner can find a way to goals and avoid all the obstacles. (Fig. 5).



```
Goal found
Num steps: 34
Goal found
Num steps: 509
```

Fig. 5: The trajectory of $heuristic\_fn\_cost\_grid$

PLOT As multiplying the $heuristic\_fn\_cost\_grid$ heuristic multiplied by 1.1, the A* motion planner can find the paths with much shorter steps and the paths stay away from the obstacle (Fig. 6). It is obviously that the heuristic function is not admissible.



```
Goal found
Num steps: 13
Goal found
Num steps: 40
```

Fig. 6: Trajectories of $1.1 * heuristic\_fn\_cost\_grid$

Question Which one of these heuristics is better? Why?

Answer: Both the heuristic of cost grids can navigate the agent to the goals, and the one with multiplication of factor 1.1 provided a better result (less steps and make a larger clearance to the obstacle). In my opinion, as we multiple the heuristic function with a factor, we overestimate the heuristic path than the true value. By then, the A^* will try to find path from the other cells and not focus on the shortest path. For the obstacles in this configuration, it is a good way to make the A^* go over the obstacles. However, it is obviously not a optimal (shortest) path.

B. Receding Horizon Planning

PLOT As running the receding horizon code, the motion planner can find a path to goal as shown in Fig. 7. However, there is some points, the path intersect with the obstacle. The motion planner does not work well this case.

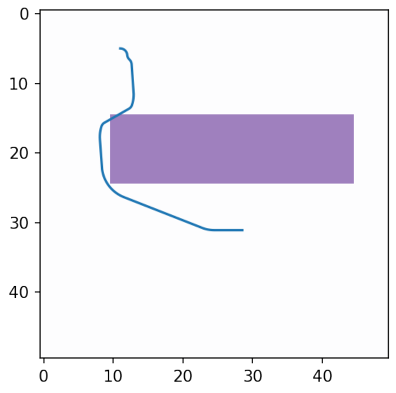


Fig. 7: Trajectory of receding horizon Planning

QUESTION You are asked to use $heuristic\_fn\_cost\_grid$ for this question. Why might the $heuristic\_fn\_cartesian$ not work? Give an example of a scenario when the robot might not reach the goal.

Answer: the $heuristic\_fn\_cartesian$ function is admissible which focus on determining the shortest path to the goal. For simple environment, it works very well. However, it might not work for this environment because the obstacle blocks all the direct path of connecting the goal and the start. Thus, the heuristic function will guide the motion planner running around the local optima and never overcome the obstacle. It is obvious that this heuristic function cannot work for non-convex obstacles.

TASK After running for a simple map, we extend to a much larger map. The result is shown in Fig. 8. The robot crosses the building's walls and reaches the faraway goal, however, it took a significant running time (from google colab, it took several hours to plot the graph.)

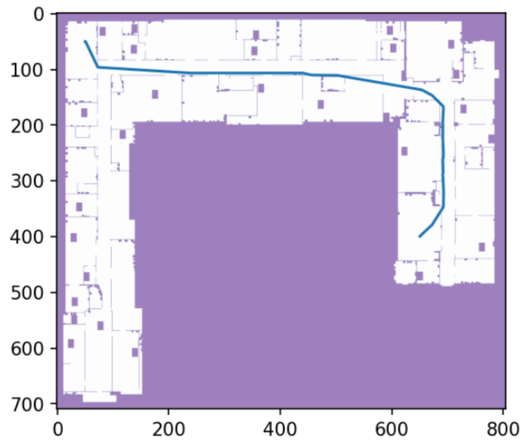
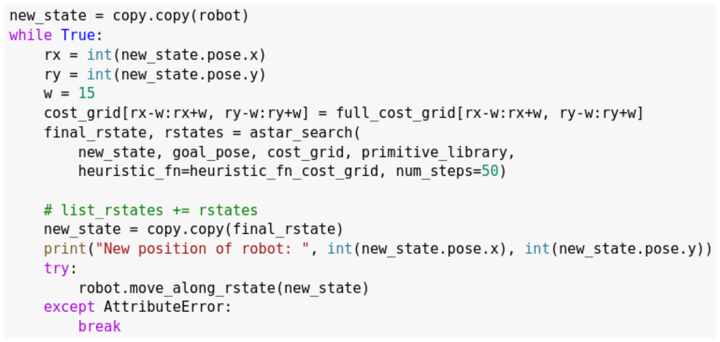


Fig. 8: Trajectory of Receding Horizon Planning for larger map

C. Planing in a Partial Map



```
new_state = copy.copy(robot)
while True:
    rx = int(new_state.pose.x)
    ry = int(new_state.pose.y)
    w = 15
    cost_grid[rx-w:rx+w, ry-w:ry+w] = full_cost_grid[rx-w:rx+w, ry-w:ry+w]
    final_rstate, rstates = astar_search(
        new_state, goal_pose, cost_grid, primitive_library,
        heuristic_fn=heuristic_fn_cost_grid, num_steps=50)

    # list_rstates += rstates
    new_state = copy.copy(final_rstate)
    print("New position of robot: ", int(new_state.pose.x), int(new_state.pose.y))
    try:
        robot.move_along_rstate(new_state)
    except AttributeError:
        break
```

Fig. 9: The code for partial Map

In this section, we use Receding Horizon Planning for Partial Map. The robot updates the map by current information, planning a path with T steps, and execute the path. The procedure is repeated until the robot can find its goal. The code is shown in Fig. 9. We set a new_state variable which is the current position of robot after each time of calling $astar\_search$, and is used as the start of the next call. In the $while$ loop, the cost grid is updated by current location and its local information; then the $astar\_search$ is invoked, new_state is updated. After each call of $astar\_search$, the new pose and trajectory of the robot is update by the command: $robot.move\_along\_rstate(nw\_newstate)$.

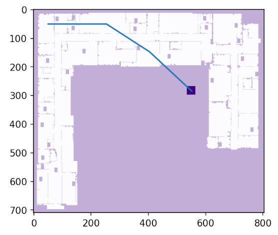
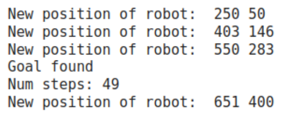


Fig. 10: The path for partial Map

PLOT After running the code, the result is shown in Fig. 10. It is obviously that the path is long because the robot is not provided full information at the beginning. I printed out the robot's position after each receding horizon planning, and it shows that the robot reached the goal as shown in Fig. 11. However, it is not clear why the path is shown as like in Fig. 10.



```
New position of robot:  250 50
New position of robot:  403 146
New position of robot:  550 283
Goal found
Num steps: 49
New position of robot:  651 400
```

Fig. 11: The intermediate points for Partial Map

II. Monte Carlo Sampling

A. Computing Area with MC Sampling

PLOTS Include a two plots (one for each environment), showing the points that were inside and outside the different obstacles (after 2500 samples).

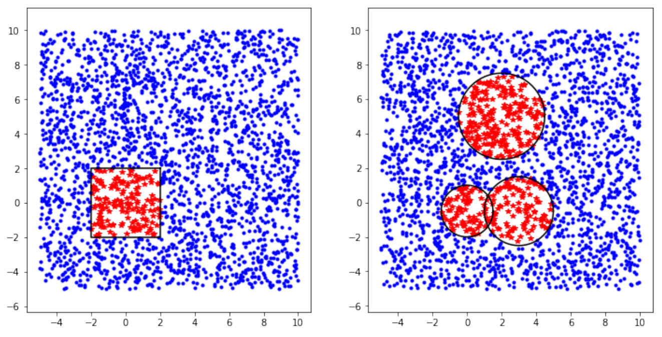


Fig. 12: Monte Carlo method

RESULTS/QUESTION The areas for both environments with 100, 1000, 10000, and 100000 samples are shown in Table I.

Sample numbers	100	1000	10000	100000
Square Obstacle	13.5	15.75	14.94	16.21
Multi Obstacle	31.5	37.35	39.4	38.73

TABLE I: Areas by Monte Carlo Sampling

For the square environment, the area's error with the sample number is shown in Table II. The error reduces as the number of sample increases.

Sample numbers	100	1000	10000	100000
Square Obstacle	13.5	15.75	14.94	16.21
Error	0.156	0.016	0.066	0.013

TABLE II: Area's Error for Square shape by Monte Carlo Sampling

B. More MC sampling

PLOT Our function generated 100 random lines within the specified region that avoid the side-length-1 square at the center, and those lines is shown in Fig. 13.

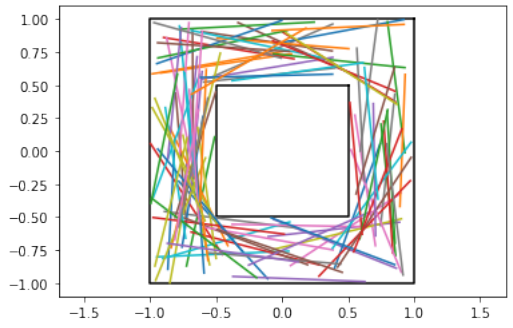


Fig. 13: Monte Carlo method for line

RESULT What is the likelihood 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?

Answer: After running with 50000 random lines, the percentage of not collision with the unit-square is: 0.2322.

III. Probabilistic Road Maps

PLOT As running the PRM graph for $num\_iterations = 200$, the results is shown in Fig. 14. The is a graph connecting all the points outside the obstacle, and we can find a path to connect the start and goal.

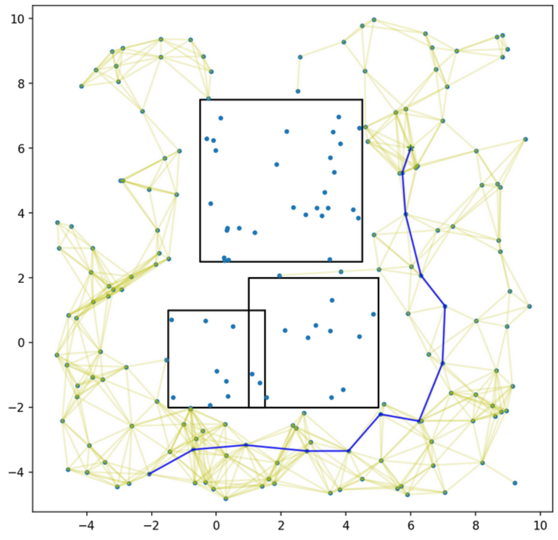


Fig. 14: PRM

RESULTS+PLOT We write a function to compute the length of the paths between the start and the goal, with varied of

$num\_iterations$ from 100 to 2500 in increments of 100 points. The paths and its lengths are shown in Fig. 15 and 16.

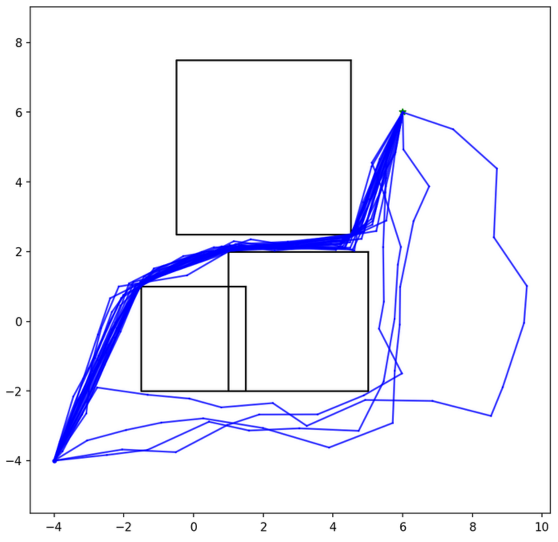


Fig. 15: Paths with varied sample number

As the number of sample increases, the algorithm can determine a optimal path which goes through the narrow gap. Off course, the path's length is shorter and shorter.

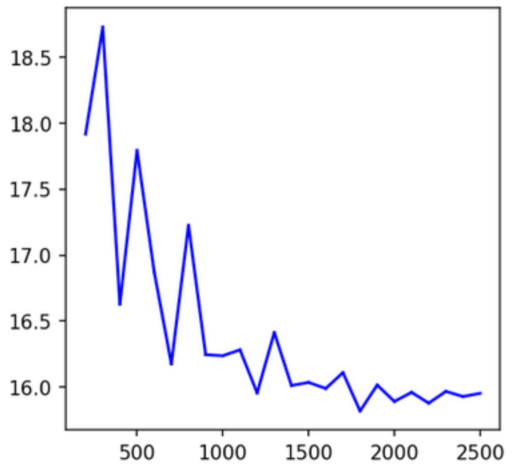


Fig. 16: Path's Length with varied sample number

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: There is several distinct "drop" in the curve of of path length versus $num\_iterations$ (Fig. 16). In my opinion, it is due to the asymptotic convergence feature of PRM. As we increase the number of sample points, we increase the probability of finding the better (optimal) solution for the problem. In the environment, the optimal path goes through the small gap between the two obstacles. With more sample points, it has more chances the sample points be in the gap, then we can build edges going through it.

IV. RRT and RRT*

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)?

Answer: The function $steer\_towards\_point$ determines how far the new point will move comparing to the current link. The limit distance of the new point to the current link is less than or equal to a threshold $step\_size$. This step will make the collision checking easier and stabilize the growing of the tree. If adding the vehicle dynamics, we need to check the dynamic constraints of the vehicle to determine the next point.

PLOTS After running the "random chain" code, we have two paths as shown in Fig. 17.

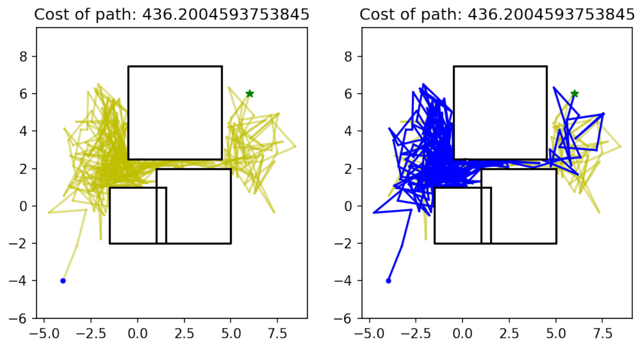


Fig. 17: Path with random move

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: In all the running time, the algorithm is able to determine a path from the goal to the object. Although the paths is messy, but as we add more sample point, finally we can find a path connecting from start to goal. It is obvious because as the number of sample points increases, then there will be a point close enough to the goal to set up a path. However, it is obvious the path will be never optimal due to the new point is always connecting to the latest point which can be close to the goal than the new point.

B. RRT

PLOTS After running the RRT plotting code, we have 4 plots in Fig. 18. After determining the path of connecting the start and goal, there is no change of the path even though we increase significant the number of sample points.

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

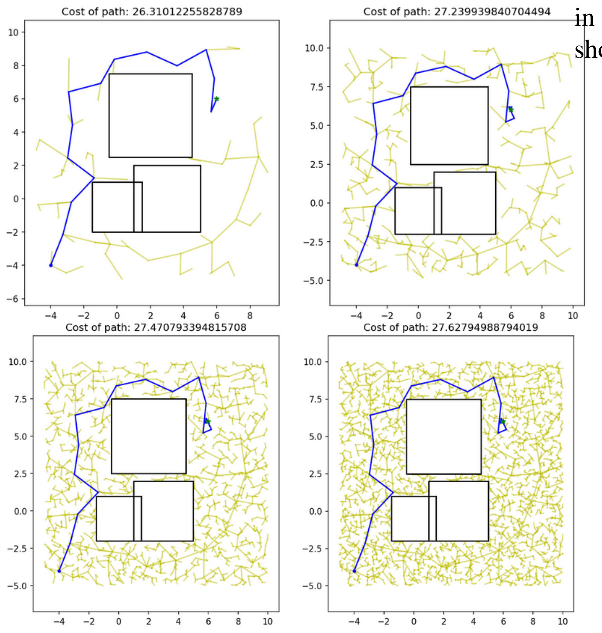
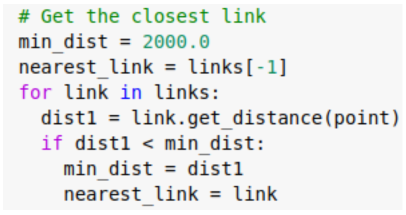


Fig. 18: Trajectories with varied sample numbers

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)?

Answer: This RRT planner is complete. As we increasing more sample point in the environment, then it is definitely there is a point which is close enough to the goal, so we can choose the link associated with the point, and connect back the start point via the tree. The found path will be shorter than the path in RRT latest points, however, RRT will never be optimal. The early sample points are crucial to the algorithm, and the path length depends on them. Those early points construct a frame, and the latter points will connect them step by step. Thus, if those early sample points are selected poorly, there is no way to fix it, and the path's length will be very long.

CODE: In the code (Fig. 19), after getting a new point, we run all the $link$ list to find the closest link to the new point. We then build a new link for the new point, and its parent is the closest link.



```
# 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. 19: The code of RRT

C. RRT*

PLOTS Running the RRTstar plotting code, we get 4 plots to show the performance as a function of number of iterations in Fig. 20. As the number of sample increases, the path is shorter and smoother.

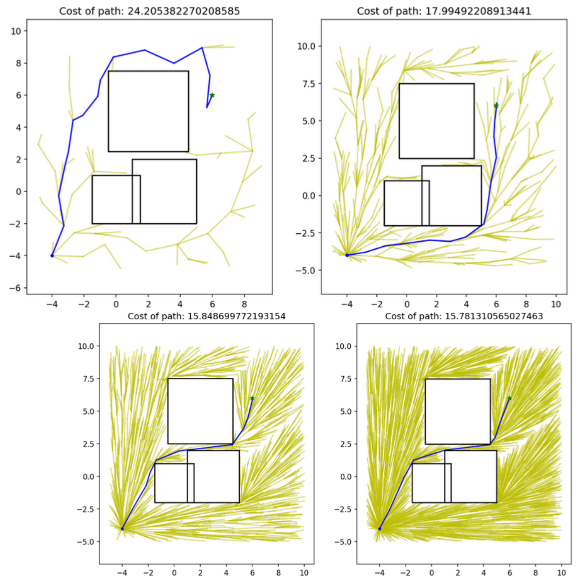
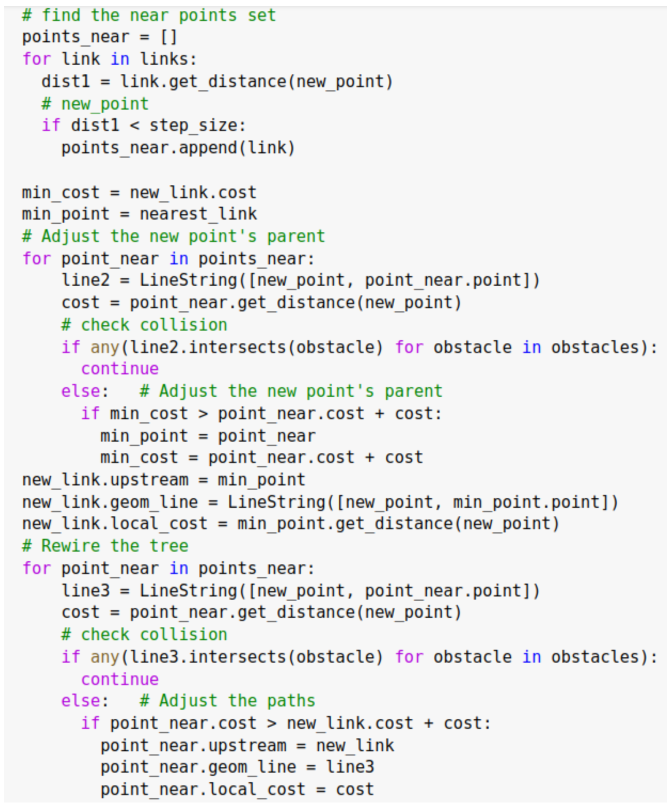


Fig. 20: Trajectories with rrt*



```
# find the near points set
points_near = []
for link in links:
  dist1 = link.get_distance(new_point)
  # new_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)
# Rewire the tree
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. 21: The code of 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)? How do your theoretical claims match up with the results?

Answer: The $RRT*$ motion planner is complete with similar argument in RRT motion planner. The significant advantage of this motion planner is it will provide a shortest path if the number of sample goes to infinity. That is because of two adjustment steps in the algorithm. After setting up the new link as similar in RRT, $RRT*$ checks whether the path can be shorter with the new link. It runs through all the new link's neighbor, and if there is any neighbor can give a path with lower cost, the new link is then connected with the neighbor to set up a new branch for the tree. The second adjustment checks whether there is any neighbor which can lower its cost by connecting to the new tree branch. If yes, the neighbor will break up with his upstream and be attached to the branch. The algorithm does the adjusted steps iteratively, and finally can provide a optimal path in Fig. 20. The adjustment steps detail is shown in Fig. 21 which are the two for loop with *for point_near in points_near*.