Computational Robotics

Harsh Bhatt (hb371)

8 November 2021

CS 560

1 Problem 1

1.1 Implementing RRT

1.1.1 Implementing parser

Parsing the world and showing the parsed world definition. The start is shown in green dot while the target is shown in red dot.
This is implemented in file_parse.py

```python
def parse_problem(world_file, problem_file):
    current_obstacles, current_problems = [], []

    # 2 coordinates for 2d problem
    n = 2

    # first line is the robot
    is_robot = 0
    current_robot = []
    with open(world_file) as f:
        for current_line in f.readlines():
            current_line = list(map(float, current_line.split()))
            current_line = [current_line[i: i + n] for i in range(0, len(current_line), n)]
            if not is_robot:
                current_robot = tuple(current_line)
                is_robot += 1
            else:
                current_obstacles.append(tuple(current_line))

    with open(problem_file, "r") as f:
        for current_line in f.readlines():
            current_line = list(map(float, current_line.split()))
            current_line = [current_line[i: i + n] for i in range(0, len(current_line), n)]
            current_problems.append(tuple(current_line))

    return [current_robot, current_obstacles, current_problems]


if __name__ == '__main__':
    world = "world_definition_files/robot_env_01.txt"
```

Figure 1: World parsed function

1.1.2 Implementing Samples

This is shown in sampler.py. I have created a function for printing points in visualize which takes a samples list and displays them. The sample list just has the point where the robot's origin would be places. In my diagram, the samples are just points to understand the differences with the other points and the start and goal nodes.

1.1.3 Collision checking

The collision checking checks if a current point is inside a given obstacle or not. We extend our point till outside the boundary and check how many times the segments have crossed other segments. Other segments here are obstacle segments and if the count is even, means the point is outside and if the count is odd, the point is inside the obstacle.
Online reference used was Online reference

1.1.4 Implementing Tree

I have used the discretized version of extending two points. I take a step size which is my parameter and normalize it with the distance between points. Then I extend to the new point which I got through step size and see if this

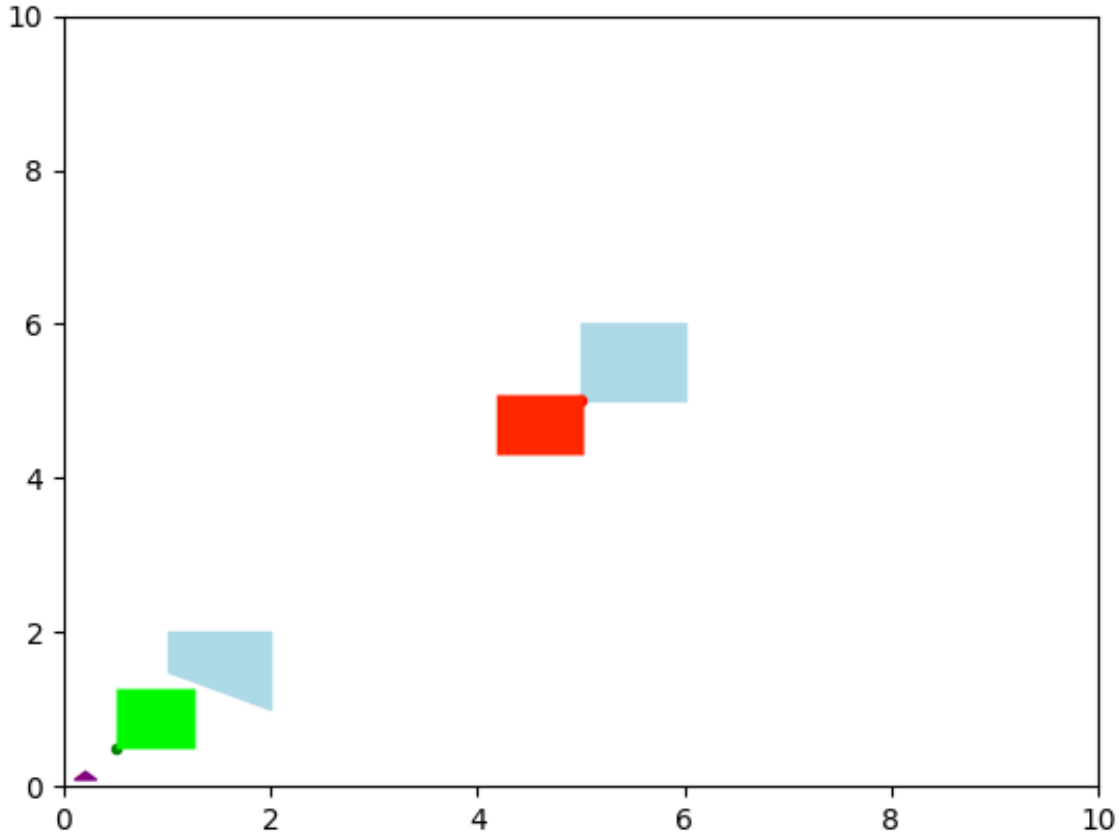

Figure 2: World parsed

is in collision or not. If it is not, I'll move further until I reach the target point. If there is a collision, I add the previous node not in collision with the tree node.

Nearest is just euclidean distance used using L2 norm.

1.1.5 RRT Algorithm

Steps for RRT:

- We have start and goal with the robot and obstacle configuration
- We have to find the random samples within the boundary range
- Finding the nearest point in the tree to the sample point.
- If the connection is valid, we would extend to target otherwise extend to any non-collision path in target.
- If we are in some radius of the target, we add the nodes and our algorithm stops. If not, the algorithm does not get a path.

The success rate for this algorithm heavily depends on the iterations (or samples generated). When iterations were **100** there were 2 conditions:

- If the obstacles were a lot in number, then the algorithm would not get the result.
- If obstacles are not a lot, it was able to get the path infrequently.

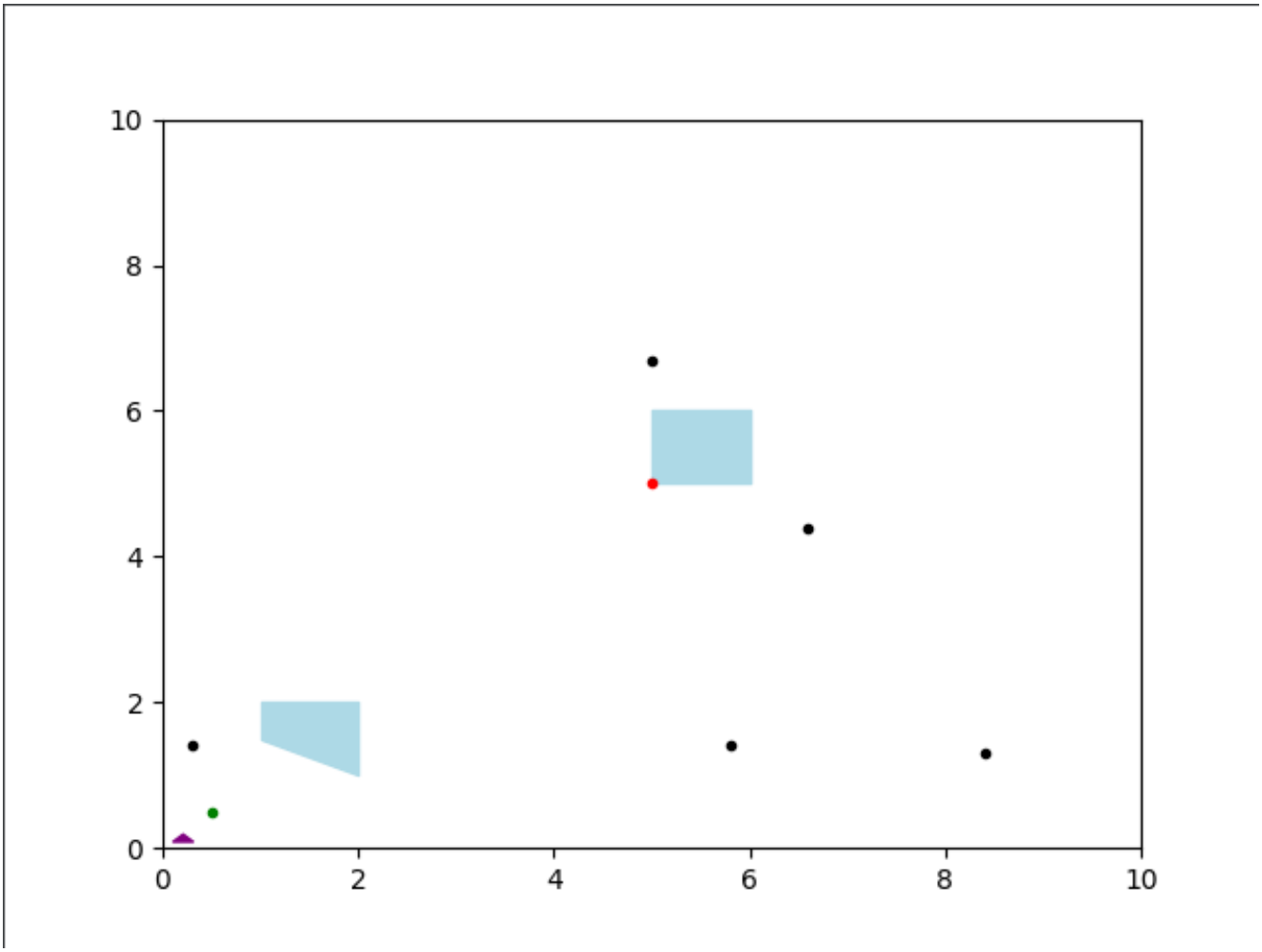

Figure 3: Point world parsed with 5 samples

So, if we have more iterations for the algorithm, the success rate increases.

1.2 Visualizations

The visualize path for rrt can be seen as:

1.3 Implementing RRT*

1.3.1 Cost and rewiring

Cost is just going till the parent and finding the cost on the way.
Rewiring is if we find any point already in the tree which can get a better path with the newly added node, we remove the old node there and add the new node with the new cost between the edges.

1.3.2 RRT*

The algorithm remains the same as RRT, but instead of completing an iteration after **extend**, we check if the cost for any other node in the tree can be minimized, which is rewiring.

For the same samples, the path for rrt was
Path is [(0.5, 0.5, 0.0), (1.29, 0.6, 4.32), (1.77, 0.72, 3.97), (1.86, 0.7, 3.94), (2.88, 0.37, 5.18), (4.55, 5.08, 2.27), (0.42, 8.48, 4.63), (2.74, 7.53, 5.75), (4.39, 8.6, 0.14), (4.5, 7.93, 1.49), (6.66, 8.69, 1.11), (6.7, 8.79, 0.84), (6.01, 8.9, 5.77), (6.0, 8.24, 4.81), (6.0, 8.0, 0.0)]

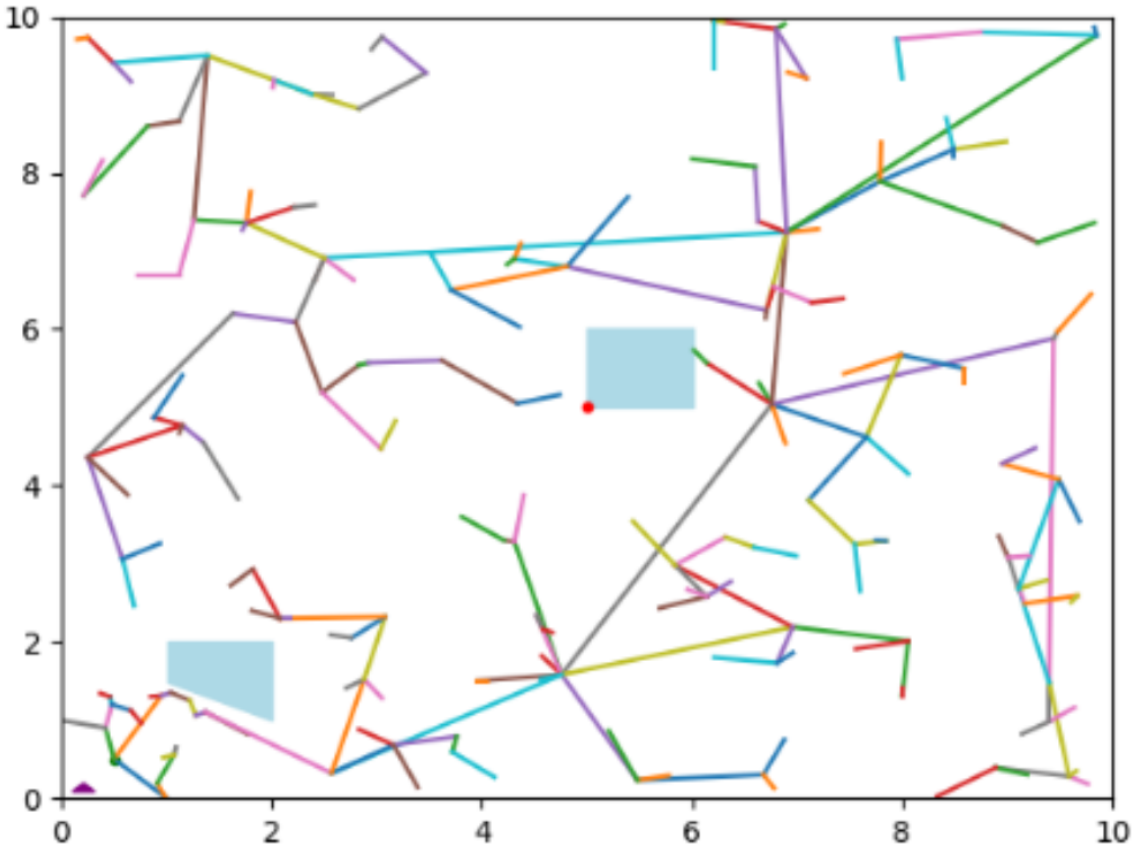

Figure 4: Tree with world shown

whereas path for rrt* was

Path is [(0.5, 0.5, 0.0), (1.06, 0.92, 0.14), (1.29, 0.6, 4.32), (1.77, 0.72, 3.97), (1.86, 0.7, 3.94), (2.88, 0.37, 5.18), (4.55, 5.08, 2.27), (0.42, 8.48, 4.63), (2.74, 7.53, 5.75), (4.39, 8.6, 0.14), (5.69, 9.2, 3.6), (6.01, 8.9, 5.77), (6.0, 8.24, 4.81), (6.0, 8.0, 0.0)]

We can see that rrt* was a little bit better. There were small paths in rrt* but the actual cost was less as seen from the figure.

1.3.3 Visualize

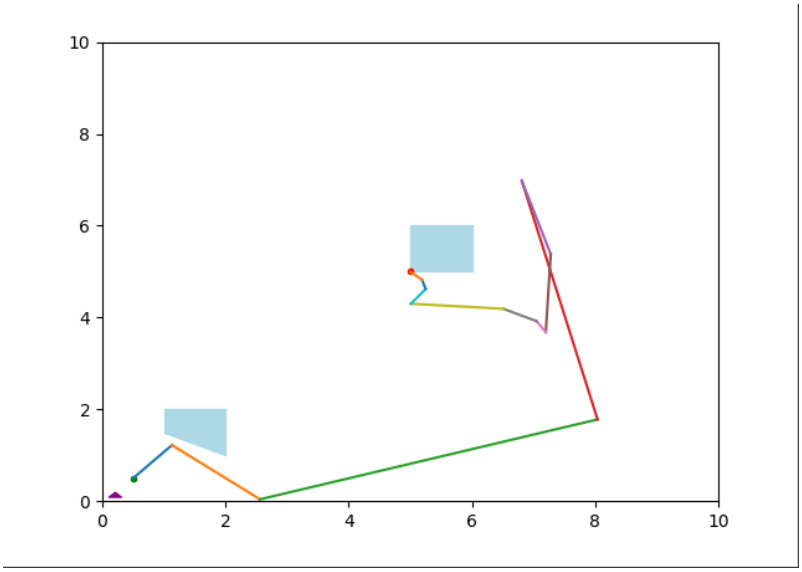

Figure 5: RRT

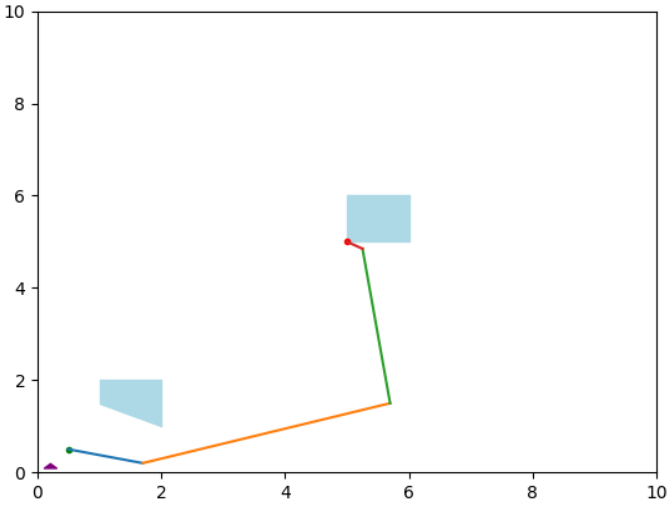

Figure 6: RRT*

2 Problem 2 - Geometric Motion Planning

2.1 Implementing RRT

2.1.1 Model the Robot

The *robot.py* file has the implementation of the below methods:

- ***__init__(self, width, height)*** : This function will initialize the robot with it's width and height. It also sets it's start coordinates.
- ***set_pose(self, pose)*** : This function sets the robot pose by setting it's transformation variables.
- ***transform(self)*** : This function is responsible for the actual transformation to the pose set using set_pose.

2.1.2 Implementing parser

file_parse.py : We implement the ***parse_problem()*** function that will be responsible to process the world file and the problem file. We have the ***visualize_problem()*** function in the ***visualizer.py*** that will visualize the parsed data.

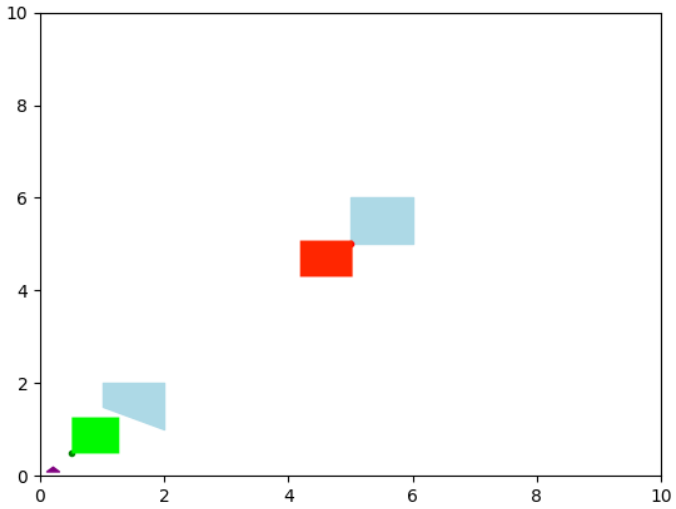

Figure 7: Parsed World

2.1.3 Extending the previous functions

I add one more parameter for sample with theta. I convert from degrees and the range is from $[0, 360)$ Visualize is shown in the previous section.

The distance is the L2 norm but I have a angle distance which I have to calculate.I just take the difference between the angles in degrees and if they are out of bounds, I convert them in the range again and convert them to radians.

2.1.4 RRT algorithm

This algorithm is same as the previous question. the only thing changes is the third parameter theta which we have to consider. And the distance formula is changed so any changes in the third parameter changes how we would get the nearest neighbors in the tree.

2.1.5 RRT*

This is same as the previous rrt star for a normal robot. Here, we check if the rewiring is possible and if that is, we rewire the new node with the existing node in the tree.

Path is [(0.5, 0.5, 0.0), (0.6, 3.77, 6.08), (0.45, 5.12, 5.33), (2.67, 7.67, 0.93), (5.15, 9.07, 1.15), (5.89, 8.61, 4.34), (5.83, 9.49, 3.68), (5.99, 9.1, 1.99), (6.65, 8.83, 1.45), (6.7, 7.94, 0.51), (6.21, 7.71, 6.1), (6.0, 8.0, 0.0)] for RRT and

Path is [(0.5, 0.5, 0.0), (0.6, 3.77, 6.08), (0.45, 5.12, 5.33), (2.67, 7.67, 0.93), (5.15, 9.07, 1.15), (5.99, 9.1, 1.99), (6.65, 8.83, 1.45), (6.7, 7.94, 0.51), (6.21, 7.71, 6.1), (6.0, 8.0, 0.0)] for RRT*. We can see that for a car, the RRT* algorithm is a little bit better.

Another comparision for the path in RRT and RRT*

2.1.6 Visualization of the found path

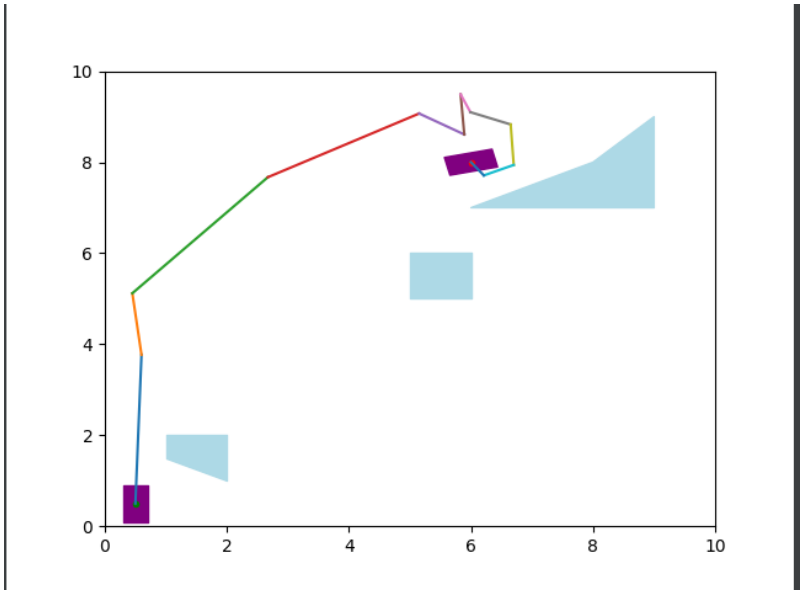

Figure 8: RRT for car

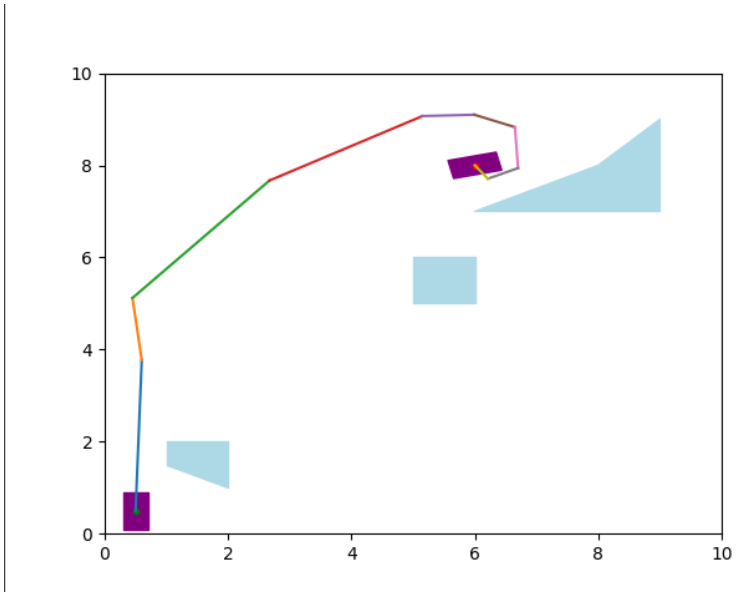

Figure 9: RRT* for a car

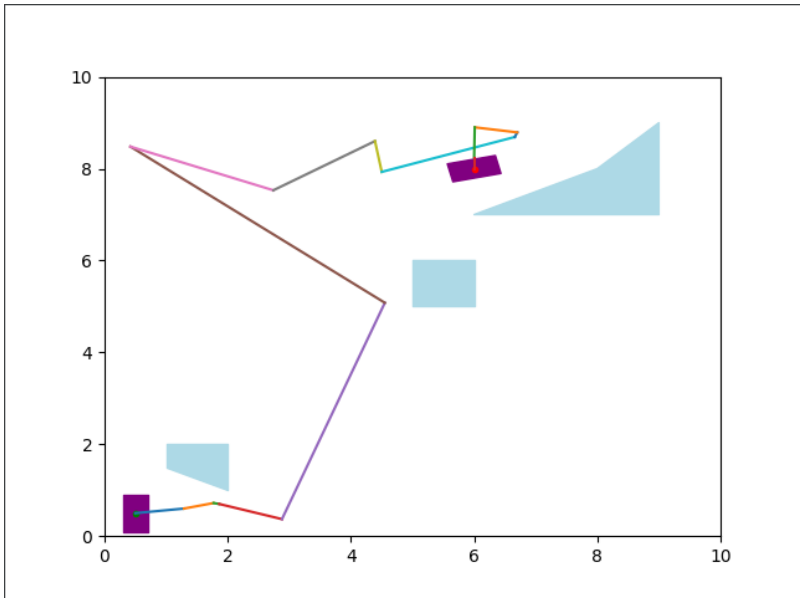

Figure 10: RRT

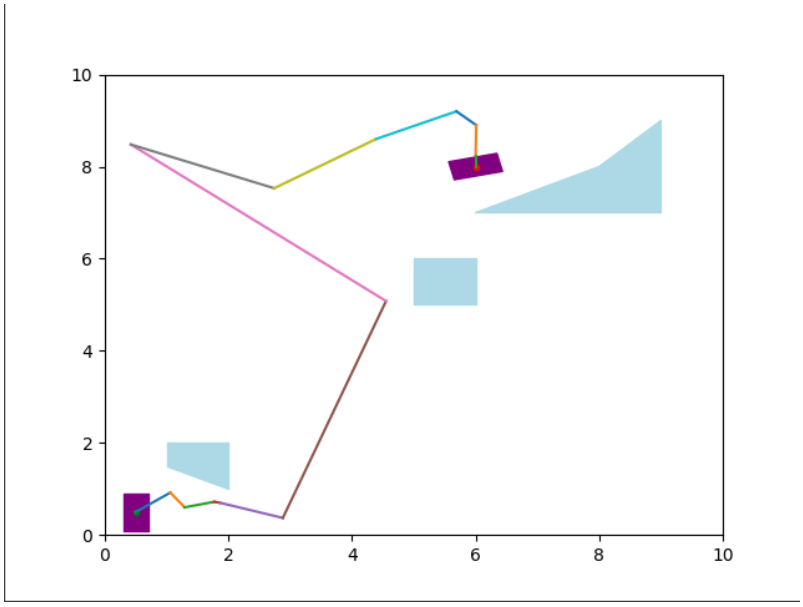

Figure 11: RRT*

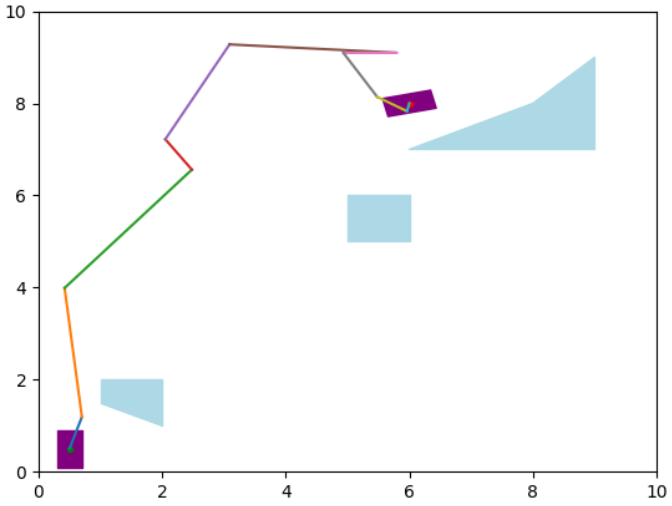

Figure 12: RRT Car

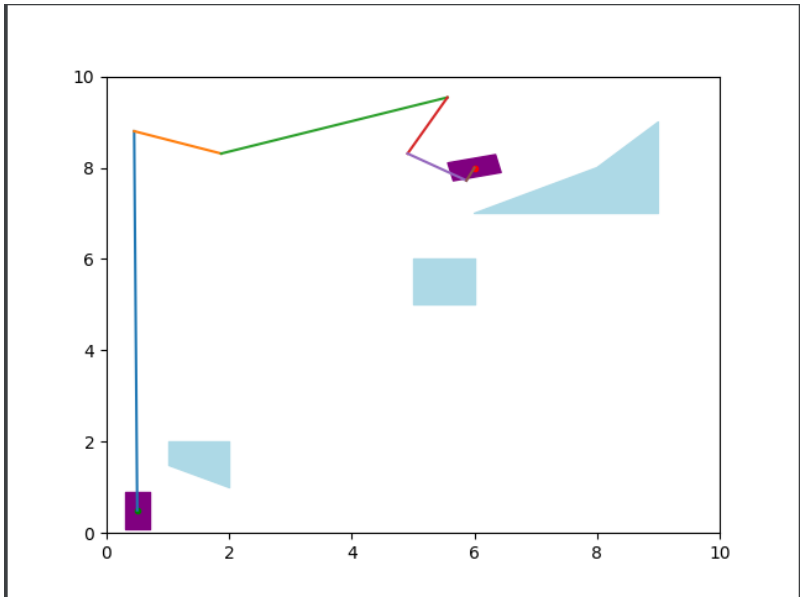

Figure 13: RRT* Car