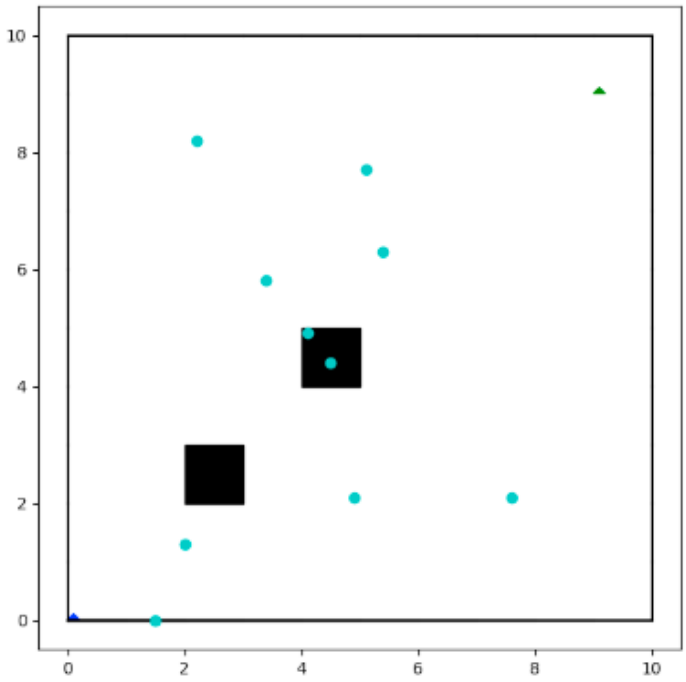
Name: Satyajith Chilappagari
Net ID: sc2100
Course: CS 560

Question 1: 2D Geometric Motion Planning

NOTE: The script file that can be used for running the code is called test.py. I have included a couple of animations from previous runs in the zipped folder in case you have any difficulty running test.py.

Problem 1:

1. The file_parse.py function contains the parse_problem() function. This function reads the world file and the problem file from the given path and returns a tuple with the robot coordinates, obstacles and problem start and goal states.
2. sampler.py contains a sample() function that generates a random point in the world boundary.
   visualizer.py contains a visualize_points() method that shows all the sample points from the sampler. A demo output of that is shown in the below image:

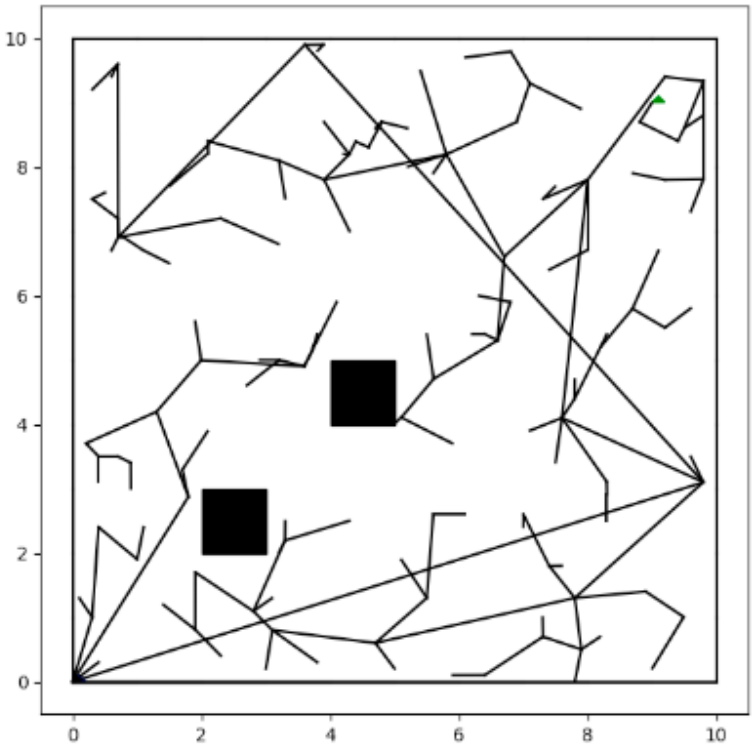


   The points in cyan are the points returned by the sampler function. The blue triangle at the origin is the parsed start point for a triangle robot. The green triangle at 9,9 is the parsed end point for a triangle robot.
3. The collision.py function contains an isCollisionFree(robot, point, obstacles) method. This method uses the following logic:
   a. First, it identifies all possible robot edges when it is at the specified point.

b. For each robot edge, and for each obstacle edge, we check if there is an intersection point. If yes, then we return False.
c. Else, we return True
d. To find the intersection point, I used a reference from the following lecture: http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf

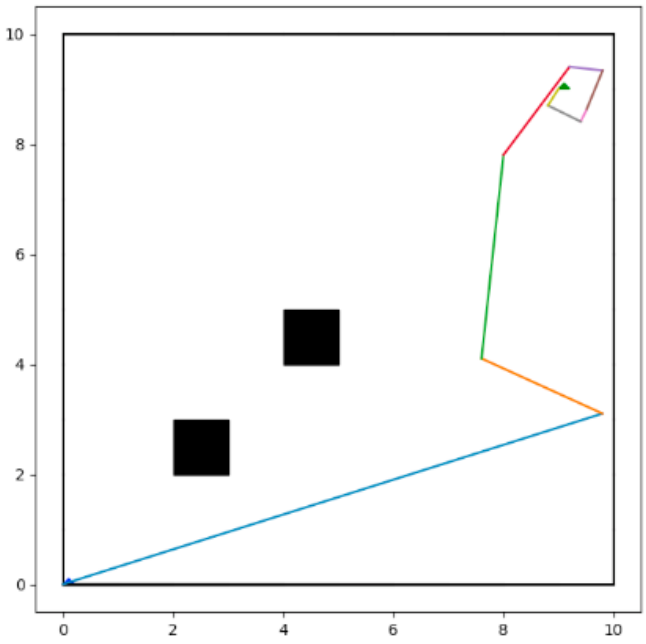
4. Next, the Tree class has the following functions:
    a. __init__(self, robot, obstacles, start, goal): Initializes the tree with the robot and the settings. We use a dictionary to store the tree.
    b. add(self, point1, point2): Adds point2 as a child to point1.
    c. exists(self, point): Returns True if the point is in the dictionary. Else, returns False.
    d. parent(self, point): Goes through the dictionary. Checks all the values. If the point is in the values, then it returns the key which is the parent of the input point.
    e. nearest(self, point): Iterates through the tree dictionary. Finds the closest point in the tree to the input point and returns it.
    f. extend(self, point1, point2): The extend function is implemented as follows:
        i. I used a **discretization** version to implement this. Therefore, there is an additional parameter to this function which is the discretization constant.
        ii. I begin at point1. Along the line segment point1 → point2, I move a length equal to the discretization constant.
        iii. I check if the new place is collision free. If yes, I update the extension to the new place.
        iv. Continue doing this until you collide or reach point2. If you collide, add the last non collision point to the tree. If you do not collide, add point 2 to the tree.
5. Implementing the RRT Algorithm: The RRT algorithm implementation has the following steps involved:
    a. We start at the start point provided by the problem constraints.
    b. We sample a random point within the boundary.
    c. We find the nearest point in the tree to the sampled point. Let's call this np.
    d. We run the extend function from np to the sampled point. Let's assume the extend function adds ext_p to the tree
    e. We continue doing this until ext_p falls in the target radius. As all co-ordinates are floating point numbers, it is difficult for ext_p to exactly match the goal co-ordinates. Hence, we stop when we reach a target radius from the goal state.

    A run of the RRT algorithm looks as follows when plotted using matplotlib:



In this image, origin is the start point and the goal state is 9, 9

The path of from the start to goal that has been computed looks as follows:



The following image shows the relation between the number of iterations and the success rate. For each iter_number, the RRT algorithm is ran 50 times to get the success rate. The numbers look as follows (Left column is number of iterations and right column is the success rate):

```
10 0.04
20 0.1
30 0.16
40 0.2
50 0.26
60 0.26
70 0.26
80 0.24
90 0.3
100 0.46
110 0.38
120 0.38
130 0.38
140 0.4
150 0.64
160 0.58
170 0.46
180 0.6
190 0.7
200 0.52
```

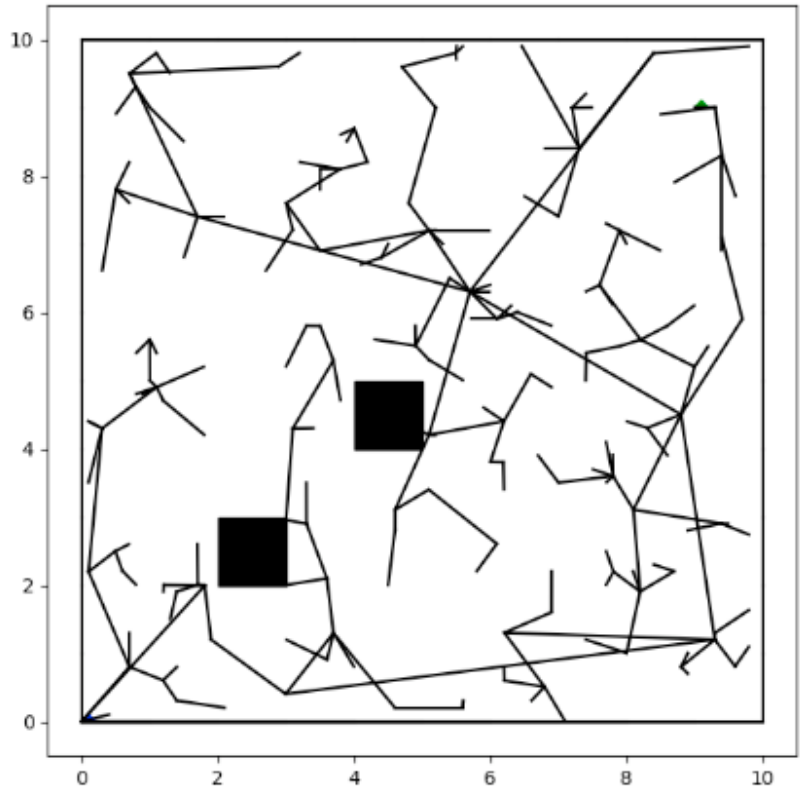
As a general pattern, it can be observed that as the number of iterations increases, the success rate also increases because the number of sampled points increases.

6. I have already included above the path visualization for the computed result by the RRT algorithm. However, there is one point that needs to be addressed here which is about the configuration space.
    a. I handled the configuration space using discretization. As I used discretization in the extend function, I'm performing an additional step for checking collisions after every discrete jump.
    b. This ensures that I never step into the collision space and that the robot can be treated as a point robot in this system for path visualization.
7. [10 extra pts] Animated visualization of the found path and the search tree:
    a. I implemented the animated visualization of the found path in the EnvVisualizer class. The function name is animate_path(). The function takes the path as input and displays an animation using matplotlib. In the animation the robot translates and moves along the computed path to the target. The animation is also additionally saved as a .gif file in the same folder as that of execution
    b. I also implemented the animated visualization of the search tree construction in the EnvVisualizer class. The name of the function is animate_tree_construction(). This function takes a tree object as input and displays all connections in order using matplotlib. A .png file containing an image of the final tree is stored at the end in the folder of execution for reference.

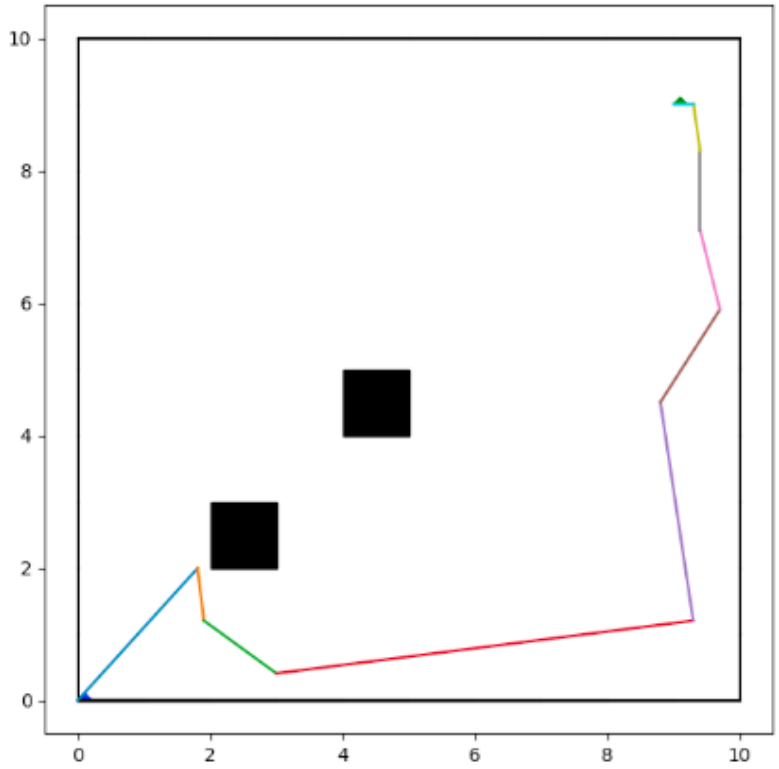
NOTE: I have provided functionality both for animation and no animation. If you do not want to see animation, you just need to use the plot functions I provided instead of the animate functions.

Problem 2: Implementation of RRT* for a Polygonal Robot with Polygonal Obstacles:

1. get_cost(self, point) method is implemented in the tree.py file. This method goes through the tree dictionary to iteratively identify parents until the start point is reached. It is to be noted that we maintain an additional dictionary that contains the distances between all edges of a tree. The path cost is computed using this dictionary and the tree dictionary together.
   rewire(self, point, r): This method performs the following tasks:
   a. First, it obtains all the tree nodes in the radius r of the point.
   b. Then for each tree node, we check if a rewire is valid. A rewire is valid if and only if both of the following conditions are true:
      i. The distance between the point and the tree node is lesser than the min distance already in that radius.
      ii. The rewire does not cause any collision. We once again use discretization here. For every discrete step of length discretization_constant, we check if the step causes a collision. If there is no collision from node to the point, then the rewire is labeled collision free
   c. The best rewire is computed from all the tree nodes in the radius r and the point is added as a child to that node in the tree dictionary. Also, the tree cost is updated.
2. The RRT* algorithm is nearly identical to the RRT algorithm described above except for one significant change. The RRT* algorithm performs a rewire immediately after every extend call to identify the most optimal path from the start within the tree constraints. Hence, this algorithm is asymptotically optimal.
3. As I mentioned above in the RRT algorithm and point 1.b.ii, for each iteration, in the extend function, we are making sure that each discrete step is valid by checking for collisions. And then while rewiring, we are repeating this check to ensure that there are no collisions while joining the rewiring edge. This ensures that we can consider our robot to be a point robot for path visualization as we used discretization to eliminate irrelevant spaces.
   A sample run of RRT* algorithm looks as follows (Note: The places where it seems like the path is moving into the obstacle is actually valid because the robot in question only extends to the right of the point in consideration. This can be verified using the animation in which the robot never enters the obstacle boundaries.)



The path computed by the RRT* algorithm in the above graph looks as follows:



The iterations vs Success rate of the RRT* algorithm can be found below. Once again, success rate is proportional to the iterations.

Iterations	Success Rate
10	0.02
20	0.04
30	0.06
40	0.08
50	0.1
60	0.16
70	0.14
80	0.18
90	0.2
100	0.22
110	0.16
120	0.38
130	0.26
140	0.3
150	0.24
160	0.38
170	0.26
180	0.28
190	0.32
200	0.28

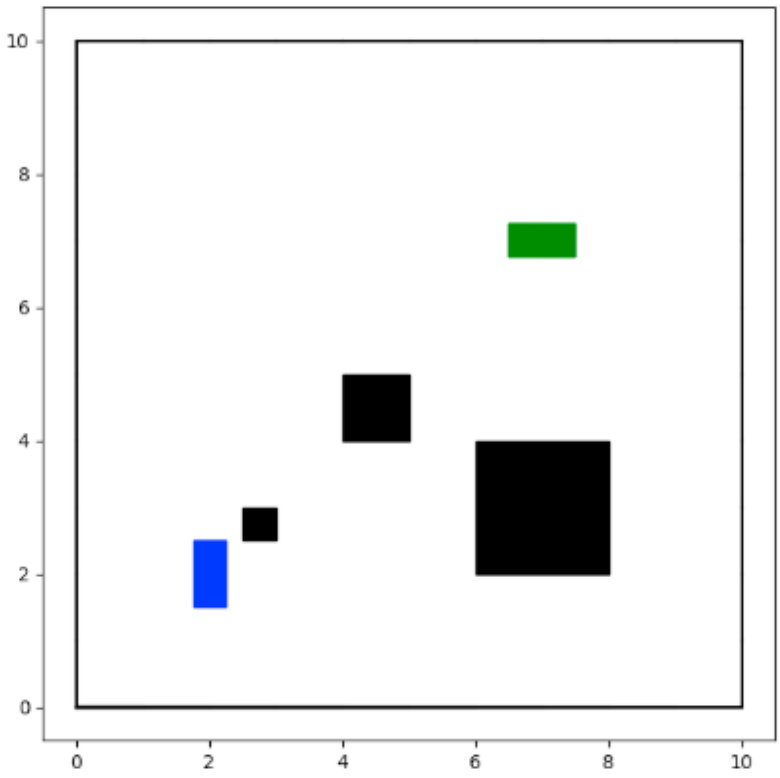
The animations designed above for the rrt algorithm can also be implemented in the exact same way for the rrt_star algorithm.

Question 2: Geometric Motion Planning for a car

NOTE: The script file that can be used for running the code is called test.py. I have included a couple of animations from previous runs in the zipped folder in case you have any difficulty running test.py.

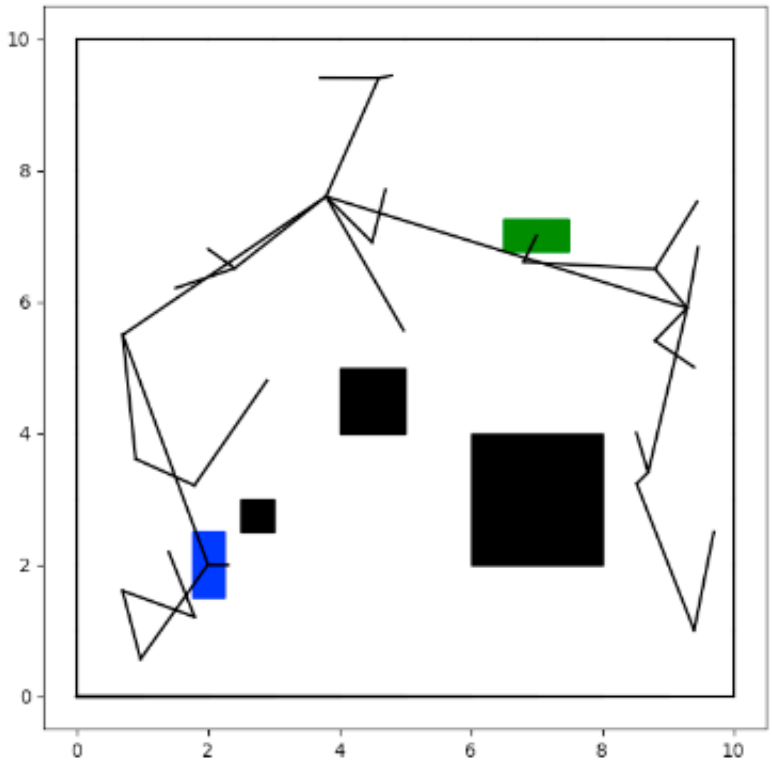
Problem 1: Implementation of RRT and RRT* for a Car with Polygonal obstacles

1. Now in the robot.py file, a Robot class is implemented that has the following methods:
    a. __init__(self, width, height): Initializes a robot with width and height. Sets the beginning co-ordinates and the dimensions.
    b. set_pose(self, pose): Sets a robot pose. Doesn't transform yet but sets the transformation variables to be used.
    c. transform(self): Uses the set_pose() variables to transform itself to a state mentioned by the pose parameters.
2. In file_parse.py, a parse_problem() function is implemented that processes the world file and the problem file. And the visualizer class contains the visualize problem method that can show the parsed data. One example of the parsed data can be found in the below image:

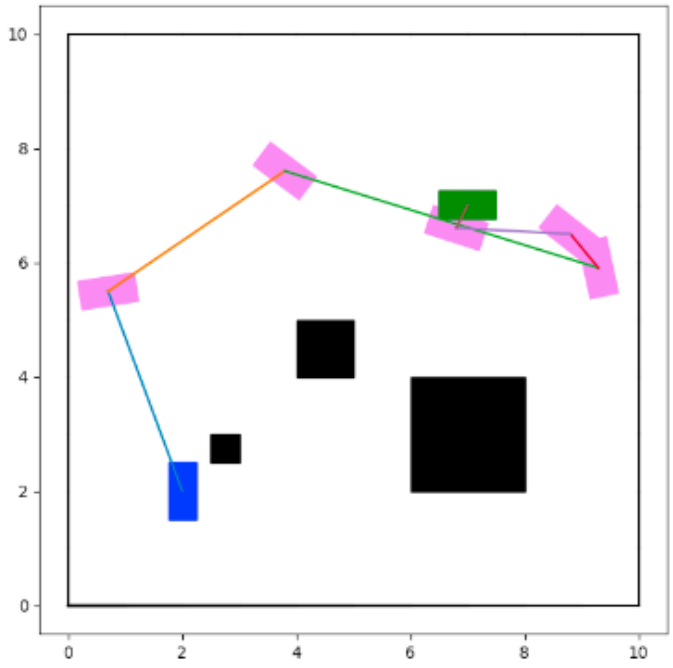


Blue is the start state of the car, Green is the goal state, and obstacles are in black

3. Extensions to previously implemented functions are done as follows:
    a. sample() method now returns an angle along with the coordinates to specify a point in the space.
    b. IsCollisionFree() method has to rotate and translate a copy of the robot to a target point, check for a collision and return the result.
    c. The tree class' nearest() function has a distance method that now includes the angular distance as well and not just the Euclidean distance.
    d. The tree class also comprises of a tree_distances dictionary that stores the distance between each pair of extended/rewired vertices.
4. The RRT Algorithm has been implemented just as before but with the aforementioned updates to the collision checking and nearest functions.
   A sample RRT algorithm implementation in the example we have chosen looks as follows:



The path computed using the above RRT tree can be found in the below image (Blue: start, purple: intermediate, green: goal)

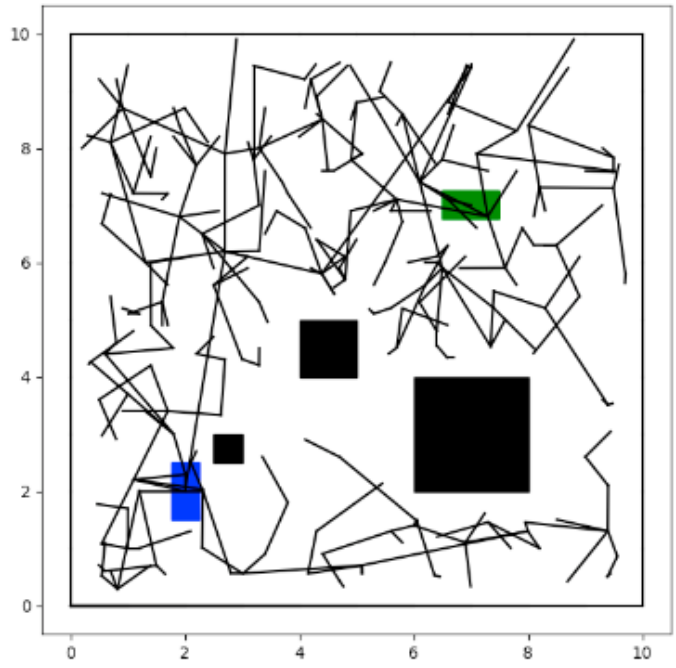


The iterations number vs success rate for RRT can be seen in the below image:

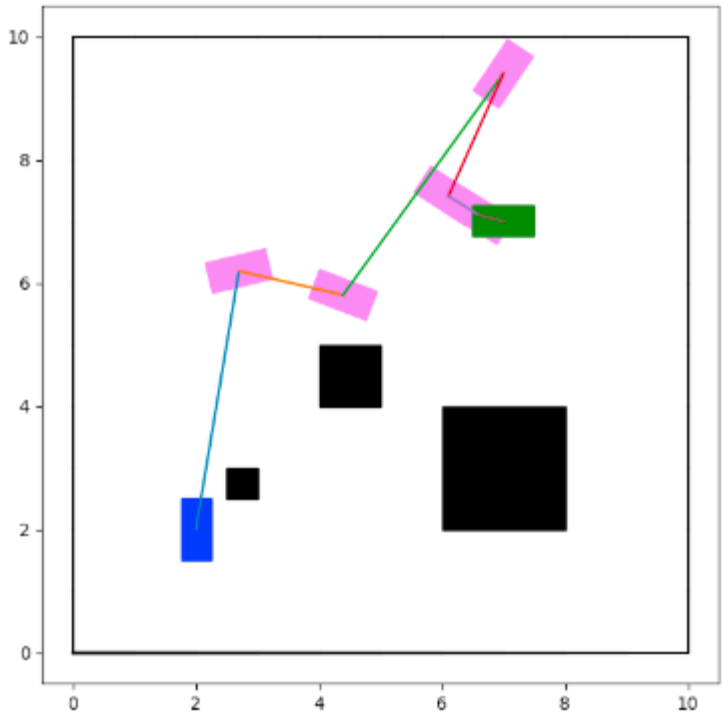
Iterations	Success Rate
10	0.0
20	0.0
30	0.0
40	0.0
50	0.05
60	0.05
70	0.05
80	0.05
90	0.05
100	0.1
110	0.2
120	0.15
130	0.35
140	0.1
150	0.05
160	0.35
170	0.2
180	0.1
190	0.15
200	0.35

5. The RRT* Algorithm has been implemented just as before but with the aforementioned updates to the collision checking and nearest functions.
   A sample RRT* algorithm implementation in the example we have chosen looks as

follows:



The path computed using the above RRT tree can be found in the below image (Blue: start, purple: intermediate, green: goal):



The iterations number vs success rate for RRT* can be found in the image below:

Iterations	Success Rate
10	0.0
20	0.0
30	0.0
40	0.05
50	0.0
60	0.05
70	0.05
80	0.1
90	0.05
100	0.0
110	0.05
120	0.15
130	0.2
140	0.2
150	0.0
160	0.15
170	0.05
180	0.0
190	0.05
200	0.1

The success rate for both these algorithms is lesser than before because the orientation adds to distance component which means that the goal is now harder to reach. Also, RRT_star isn't guaranteed to be complete as it is still a randomized algorithm. However, the distance from the start might be more optimal than that of RRT because of rewiring.

6. The animated visualization described below and the non-animated visualization described above both allow you to visualize how the car geometry changes. The only difference between this and problem 1 is that the angle theta is also taken into consideration here.
7. [5 extra pts] Animated visualization of the found path and the search tree:
    a. I implemented the animated visualization of the found path in the EnvVisualizer class. The function name is animate_path(). The function takes the path as input and displays an animation using matplotlib. In the animation the robot translates and moves along the computed path to the target. The animation is also additionally saved as a .gif file in the same folder as that of execution
    b. I also implemented the animated visualization of the search tree construction in the EnvVisualizer class. The name of the function is animate_tree_construction(). This function takes a tree object as input and displays all connections in order using matplotlib. A .png file containing an image of the final tree is stored at the end in the folder of execution for reference.

NOTE: I have provided functionality both for animation and no animation. If you do not want to see animation, you just need to use the plot functions I provided instead of the animate functions.

Question 3: Kinematic Motion Planning for a Holonomic Car

Problem 1: Implementation of RRT for a Non-Holonomic Car with Polygonal Obstacles.

1. Extend the planning problem to involve kinematics of the car:
    a. I extended the problem to include kinematics by making the state of the robot changeable only using the state, controls and durations. This ensures that we do not have direct control over the robot and that we respect the state change constraints of the robot.
    b. In the robot class, I added the following functions:
        i. kinematics(self, state, control) that implements the kinematics equation of the robot. It takes the current state and the control as input and returns a computed configuration velocity vector that can be used to change the state by using dt
        ii. propagate(self, state, controls, durations, dt): This function computes the sequence that represents the state trajectory. It calls the kinematics function using the state and a specific control. And then uses the output of the kinematics function to then compute the specific trajectory. The final state of the robot is also stored at the end in the self.state_vector.
    c. To sample random control values and durations, I implemented a function in the tree class called sample_u_durations().