16782: Planning and Decision Making for Robotics

HW2: Saurav Kambil

1) Running Instructions:

To compile on Linux (Mac or Windows may require a substitute for g++, e.g. clang):

```
>> g++ planner.cpp -o planner.out (optional but may be necessary: -std=c++11)
```

To enable debugging, add a -g tag:

```
>> g++ planner.cpp -o planner.out -g
```

This creates an executable, namely planner.out, which we can then call with different inputs:

```
>> ./planner.out [mapFile] [numofDOFs] [startAnglesCommaSeparated]
[goalAnglesCommaSeparated] [whichPlanner] [outputFile]
```

Example:

```
>> ./planner.out map1.txt 5 1.57,0.78,1.57,0.78,1.57 0.392,2.35,3.14,2.82,4.71
2 myOutput.txt
```

This will call the planner and create a new file called myOutput.txt which contains the resultant path as well as the map it was run on.

2)Helper Functions:

2.1. Checking validity of arm configuration

In my approach, I utilized the provided function IsValidArmConfiguration to ensure the arm configuration's validity at every step. This was employed consistently to verify both the start and goal configurations.

2.2. Checking validity of transition between configurations

I established a procedure to validate the transition between configurations by subdividing the interval between two nodes into a predefined number of steps. I crafted a function that meticulously checks the validity of each configuration within this interval, ensuring a smooth and valid transition.

2.3. Procuring random node

To generate a random node, I implemented a while loop, generating random angles until a valid arm configuration was achieved. For PRM, the start and goal node was inserted into the initial graph.

2.4. Finding closest node

I adopted a method to find the closest node by calculating the root mean square (rms) error between the joint angles of the current node and each node already present in the graph or tree. The node with the smallest rms error was then chosen as the nearest neighbor.

2.5. State of nodes

In my strategy, the state of nodes was represented through a structured format, including attributes such as angles, previous node, cost, and node number. This structured representation provided a consistent and efficient means of storing and accessing node information across different algorithms.

3)Summary of Approach

3.1. RRT

I employed the RRT algorithm with a loop that continues until the goal is reached. Random nodes are generated and the closest node on the existing tree is determined. If the distance to the nearest node exceeds a threshold (epsilon), a new node is added in the direction of the random node at a distance epsilon. This ensures a consistent growth of the tree towards unexplored areas. Validity checks for configurations and transitions are performed at each iteration, ensuring the integrity of the path. My chosen parameters were epsilon = numDOF/3 and step_size of 0.1 for transition validity checks.

3.2. RRTConnect

My implementation of the RRTConnect algorithm involved two trees and utilized the extend function similar to the RRT algorithm. After each extension, the trees are swapped and an attempt is made to connect the most recently added node to its nearest neighbor in the other tree. This process continues until a valid connection is established, with regular checks for configuration and transition validity. Parameters were set to epsilon = numDOF/3, step_size of 0.1.

3.3. RRTStar

In RRTStar, my approach involved extending the tree similar to the RRT algorithm, but with additional steps for rewiring to optimize the path. The goal is not a terminating condition; instead, the algorithm continues to search and rewire to improve the path even after the goal is reached. The number of rewiring attempts post-goal is determined by a predefined number of samples. The radius for neighbor selection decreases as the number of nodes increases, with parameters set to epsilon = numDOF/3 and a constant of 0.5 for radius computation.

3.4. PRM

My PRM approach constructs a graph with 5000 random nodes, including the start and goal states. Nodes are connected if they fall within a specified radius (5 nodes in this case), and are not already connected. The neighbors for each node are selected based on their distances, with a priority queue managing the neighbor selection process. The queue is limited to three contents, ensuring efficient neighbor selection. Once the graph is constructed, A* algorithm is employed to find the optimal path.

4) Experimental Results

Problem Statement:

This report provides a comparison of four different planning algorithms applied to a robotic arm motion planning problem. The environment is described in map2.txt, and 20 random start and goal configurations for the 3 DOF arm configuration are generated and fixed for all planners. The planners in consideration are RRT, RRT Connect, and RRT*. The metrics for comparison are:

1. Average planning times
2. Success rates for generating solutions in under 5 seconds
3. Average number of vertices generated (in a constructed graph/tree)
4. Average path qualities

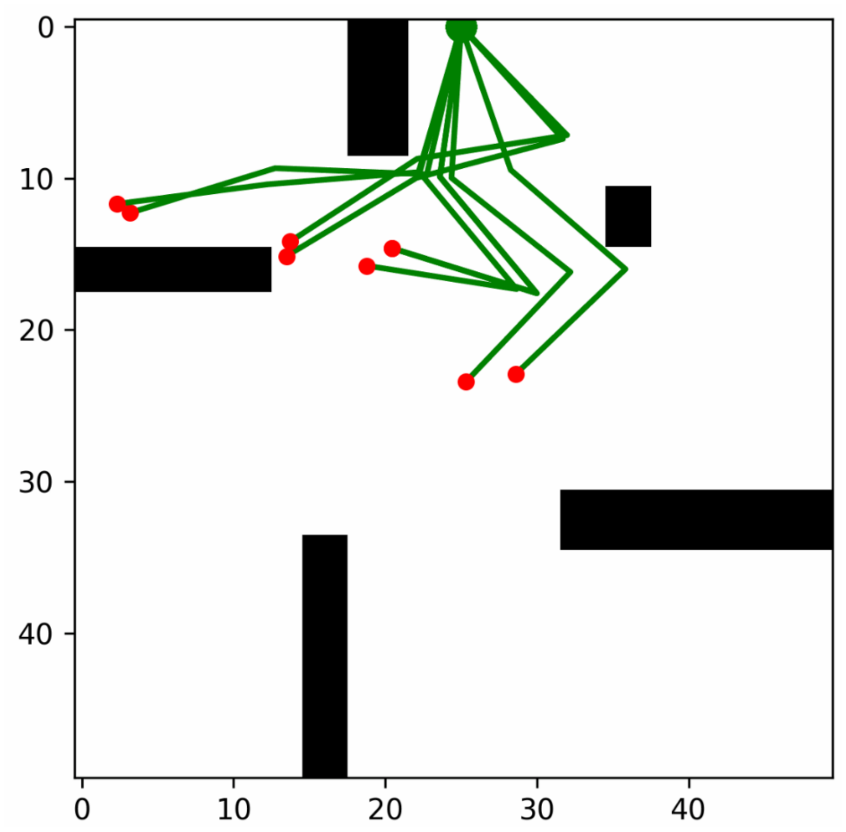

Figure 1: RRT

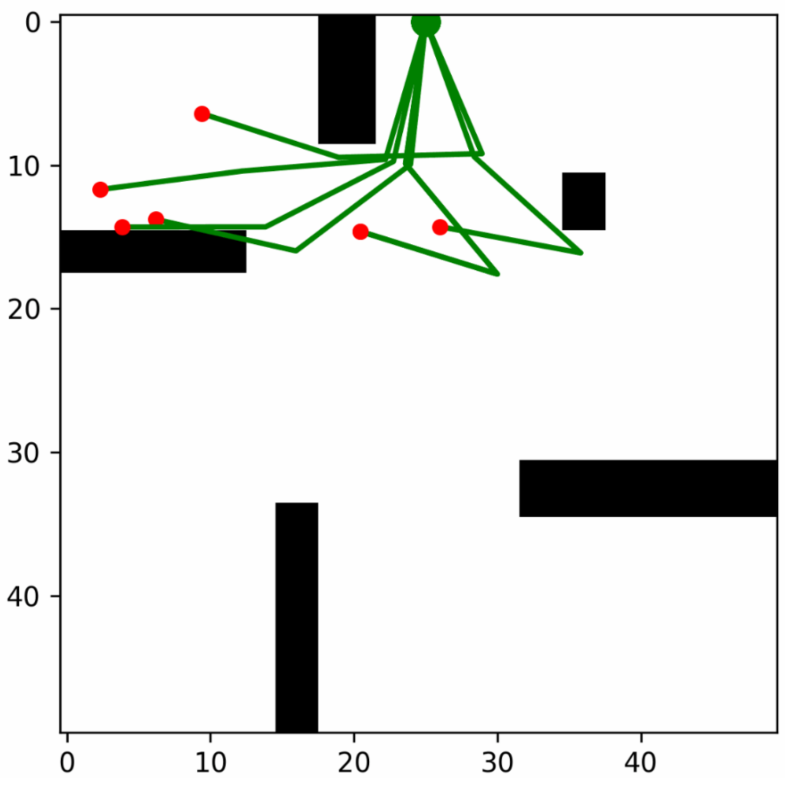

Figure 2: RRT Connect

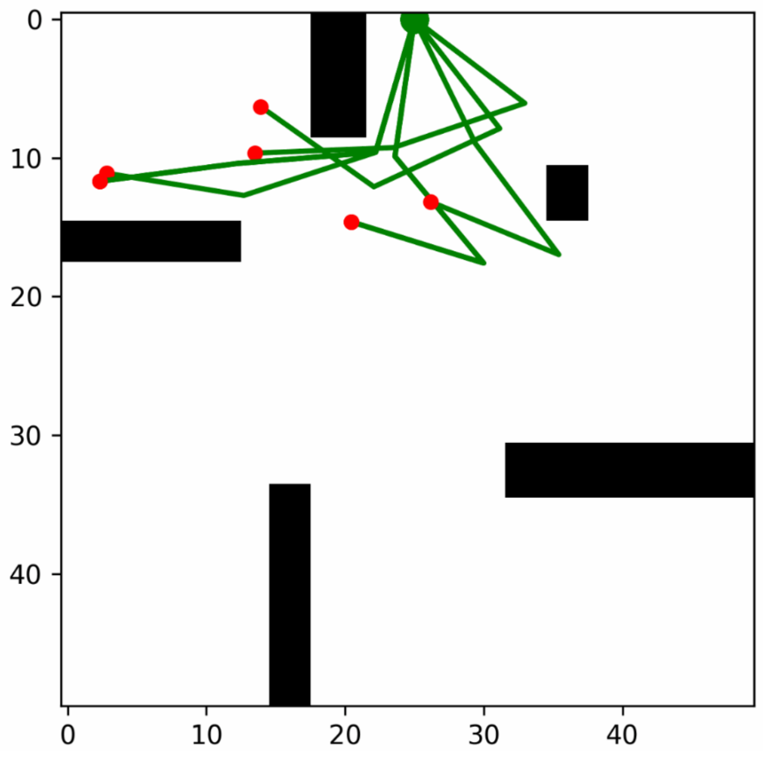

*Figure 3: RRT**

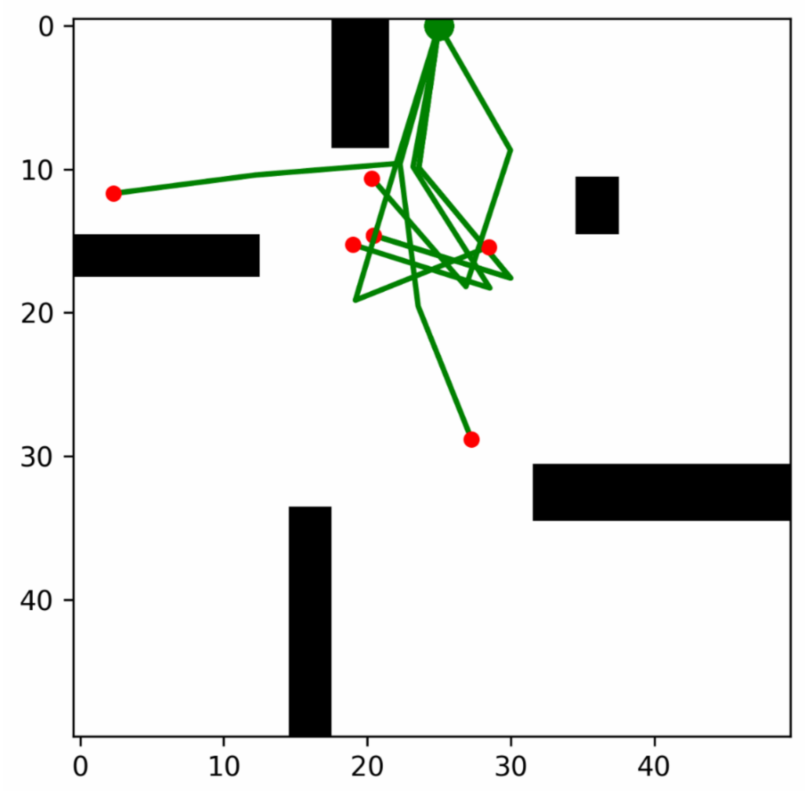

Figure 4: PRM

The results are summarized in the table below:

	Avg. Planning Time(s)	Sub 5s Success Rate	Avg num vertices	Avg. Path Quality
RRT	0.0094	95%	182	7.19
RRT Connect	0.0034	100%	152	8.34
RRT*	0.0054	90%	180	6.12
PRM	1.4	100%	650	7.2

1. Average Planning Time

RRT: The Rapidly-Exploring Random Trees algorithm took an average of 0.0094 seconds for planning, making it relatively efficient.

RRT Connect: This approach was even faster with an average planning time of 0.0034 seconds, marking it as the most time-efficient method among the ones listed.

RRT*: The optimized version of RRT, RRT*, took a little longer at 0.0054 seconds. While it's slower than RRT Connect, it's still faster than the basic RRT.

PRM: The Probabilistic RoadMap method was considerably slower in planning with a time of 1.4 seconds, making it the slowest among the four.

2. Sub 5s Success Rate

RRT and RRT*: Both of these methods have a high success rate, with RRT at 95% and RRT* at 90%. This suggests that while they are efficient in terms of time, they might not always find the best path in every scenario.

RRT Connect and PRM: Both these algorithms achieved a perfect 100% success rate within the 5-second benchmark, indicating their robustness in varied scenarios.

3. Average Number of Vertices

RRT: Utilized an average of 182 vertices.

RRT Connect: Used slightly fewer vertices than RRT, averaging at 152 vertices, which could be a factor contributing to its faster planning time.

RRT*: Was close to RRT with 180 vertices, suggesting a similar approach in terms of exploration.

PRM: Had the highest average number of vertices at 650, which is expected given its method of constructing a dense graph before searching for the path.

4. Average Path Quality

RRT: Achieved a path quality of 7.19, which is decent but not the best among the algorithms.

RRT Connect: Scored an impressive 8.34, making it the algorithm with the best path quality in this dataset.

RRT*: The path quality was the lowest at 6.12, suggesting that while it is optimized for speed and efficiency, it might not always yield the most optimal paths.

PRM: Achieved a good path quality score of 7.2, close to RRT's score but achieved with a significantly longer planning time. This is mostly because of the initial node map generation.

In conclusion, while RRT Connect shines in terms of speed and path quality, PRM offers robustness with its perfect success rate but at the cost of longer planning times. RRT and RRT* offer a balance between planning time and path quality, with RRT* leaning more towards efficiency over path optimality. Depending on the specific requirements of the application, one might prioritize speed (RRT Connect), robustness (PRM), or a balance (RRT, RRT*).