Shrudhi Ramesh Shanthi
srameshs

Planner Comparison:

	Average planning times (milliseconds)	**Success rates for generating solutions in under 5 seconds**	**Average number of vertices generated**	**Average path qualities**
RRT	18.8210703	100%	45.6	3.442971
RRT-Connect	3.4297321	100%	11.15	3.3182275
RRT*	20.948374	100%	45.6	3.157945
PRM	3884.863769	100%	1002	3.7618305

* Data supporting the above results can be found in this sheet.

Summary of the results:

The results above summarize the comparison between the four planners implemented. They were generated from 20 randomly selected configurations for a 3DOF arm. I generated each configuration by making use of the grader.py script. In specific, I had one fixed start configuration and obtained the goal configuration by checking the validity of the random generation. The metrics above were arrived at by taking the average of all trials run for each planner, with each trial being specific to a certain start & goal configuration. Further information on the data collected with respect to each configuration is provided in the excel sheet linked above.

The comparison above gives conclusive results that RRT-Connect performed the best in terms of average planning time. This corresponds to the growth of the start & goal trees, hence decreasing the planning time significantly. Secondly, RRT* had the largest planning time out of the planners due to the computation required to rewire the nodes. Lastly, PRM performs poorly

in comparison to the other planners with the largest planning time out of them all. This makes sense as PRM generated the entire graph before actually constructing the path. It's important to note that all planning implementations were able to compute a solution within 5 seconds. In conclusion, RRTConnect performed the best and PRM performed the worst in terms of planning time & computation intensity.

The average number of vertices amongst the RRT algorithms are varied. Although, the average number of vertices generated are reduced in comparison to a vanilla implementation as I have included a 10% goal bias in all RRT algorithms. RRT-Connect has the highest average number of vertices. RRT and RRT* have the same average number of vertices lesser than RRT-Connect. This indicates that RRT and RRT* are more exhaustive in their exploration. RRT-connect's strategy of quickly connecting trees allows it to find solutions with significantly fewer vertices, demonstrating greater efficiency in this specific context with the same vertex count for RRT and RRT* reflects their similar exploration methods. Although this might be an outlier case as I had run many tests before, this is the one case where I have found the number of nodes generated to be less than that of RRT-Connect. In contrast, PRM always generates 1002 vertices on an average. This is because I have set the number of nodes to compute in my graph as 1000, and the additional 2 nodes are my start & goal nodes.

The average path quality is calculated as the total sum of joint angle movements. With the lowest sum, RRT* had the best path quality out of all the planner implementations.This correlates with RRT*'s behavior to rewire nodes to give the shortest path. RRT-Connect comes in better having a slightly lesser path quality in comparison to that of RRT*.

Conclusions:

1. What planner do you think is the most suitable for the environment and why?

The most suitable planner for this environment is RRT-Connect. It performed the best in terms of path planning time and was not too far from RRT* in terms of path quality. It also handles larger degrees of freedom better in comparison to that of RRT.

PRM had the worst performance out of all the planner implementations. It's computation time is too high as we have to create the graph in advance to find a solution and did not have the best path quality either. RRT is a single-shot planner, it doesn't need to create a graph prior to finding a solution. Although RRT* produces the best path quality out of all the planners, it is not my first choice as it doesn't balance out for the high computation time required to obtain that output. With RRT-Connect's planning time and path quality in consideration, I would pick that as the planner most suitable for this environment.

2. What issues does the planner still have?

Although RRT-Connect is the most suitable for this environment, it might not produce the most optimal solution in comparison to that of RRT*. In this case of testing with a 3DOF arm, there was not much of a difference. But, with higher dimensionality, the quality of the path can significantly reduce with RRT-Connect in comparison to that of RRT*.

Shrudhi Ramesh Shanthi
srameshs

3. How do you think you can improve that planner?

One way to improve the planner would be to introduce local path-refinement. This could be in the form of post-processing, such as spline interpolation, or shortcutting to identify the segments of the path that can be replaced with straighter paths to minimize the overall length and improve path quality. In addition, some measurements, such as euclidean distance, may not work as well for this case as absolute distance may work for example.

Extra Credit:

Below are additional statistics for a start and goal configuration to portray how consistent the planner implementations work. I ran 20 trials for each planner implementation, and I calculated the average & standard deviation for planning time, number of vertices, path quality and path length. The tables below represent the results from the 20 trials.

From the results, RRT-Connect & RRT* had consistently low planning time. But, RRT had a slight deviation from a consistently lesser planning time. PRM on the other hand, took an average of 2570 milliseconds with a standard deviation of 66. So, we can say PRM took 2.57 seconds on average to plan for this configuration.

From the data, PRM had the longest planning time, averaging 2570.5 milliseconds, likely due to the large and consistent number of vertices generated (1002). RRT had a high variation in both planning time and number of vertices, indicating its reliance on random sampling, which introduces variability. RRT-Connect showed the least variability across planning metrics, making it the most consistent among the RRT-based methods.

Regarding path quality, RRT-Connect had the lowest standard deviation, indicating it consistently produced paths with similar quality. PRM had the longest average path length and the highest variation in path quality, showing it was less effective in finding shorter, optimal paths. RRT and RRT* both had moderate path lengths and path quality, with RRT* slightly more consistent in path quality than RRT.

Overall, RRT-Connect emerged as the most reliable option in terms of both planning time and path consistency, while PRM was computationally intensive and had less efficient path solutions.

Shrudhi Ramesh Shanthi
srameshs

	Planning Time (milliseconds)		Number of Vertices	
	Average	**Std. Deviation**	**Average**	**Std. Deviation**
RRT	0.2	0.6155870113	11.95	8.331771784
RRT-Connect	0	0	4.9	0.3077935056
RRT*	0	0	9.5	3.187145233
PRM	2570.5	66.37176001	1002	0

	Path Quality		Path Length	
	Average	**Std. Deviation**	**Average**	**Std. Deviation**
RRT	3.7094	0.6609536514	5.15	1.386969434
RRT-Connect	3.4264395	0.3028273986	4.9	0.3077935056
RRT*	3.551421	0.422196813	5.1	1.518309309
PRM	3.90401	0.6681966908	9.25	3.385339911

Hyperparameter Tuning:

For RRT, I tuned the epsilon distance, goal threshold distance, goal-bias probability, and interpolation step size. Increasing epsilon sped up planning but made paths less smooth; a smaller value could smooth paths further but would slow planning. I went ahead with an epsilon of 0.5 for all planner implementations. A goal distance threshold of 1e-3 worked reliably. I used a 10% goal-bias, meaning the goal was sampled 10% of the time during random sampling; lowering this increased graph uniformity, which could improve path smoothness but slowed planning. I set the interpolation step size as `min(epsilon, dist) / dist` to adapt movement scaling based on distance. For RRT* I set a neighborhood size of 10 and used Euclidean distance as the cost function.

For PRM, I focused on interpolation steps, neighborhood size, vertex count, and cost function for A*. Starting with 100 interpolation steps, I reduced it to 10 to improve planning speed. A neighborhood size of 5 allowed me to reduce the planning time significantly and improve the path quality, while setting 1,000 vertices supported higher-dimensional configurations without overly slowing planning. Euclidean distance served as the cost metric, and I limited each node to a maximum of 10 edges to optimize search speed.

Shrudhi Ramesh Shanthi
srameshs

Compiling the code:

To compile the code, within the build directory run the following:

```
cmake ..
cmake –build . –config Debug
```

Important Notes about Submission:

- In code/scripts, *test_grader.py* is the grader file that I had modified and made use of during the progress of my assignment
    - This file is how I was able to generate the data for the 20 random configurations
- In code/output, *report_path_quality.csv* is the output file from *planner.cpp* that consolidates the data required for the planner comparison not produced through the *grader.py*
    - In addition to this, *extraCredit_path_quality.csv* is the output file used to consolidate results for the extra credit question
- In code/output/grader_out, *report_grader_results.csv* is the output file from the *test_grader.py* used to consolidate the results for the planner comparison
- In addition to the above, I have a *test_verifier.py* that I had modified to locate the *map#.txt* on my system

In conclusion, I have included the **original** provided ***grader.py*** and ***verifier.py*** in my submission in addition to their modified variants.