Sridevi Kaza
Planning & Decision Making HW2

Results

Number of Degrees of Freedom: 4

	RRT	RRTConnect	RRT*	PRM
Average Planning Time (milliseconds)	143.7	0.35	1411.75	2884.9
Success Rate for Generating Solutions in Under 5 Seconds	100%	100%	100%	100%
Average Number of Vertices Generated in the Graph	217.5	16.95	486.4	1002
Average Path Quality	3.6501	3.6128	1.4513	4.4699

For extended results for each of the twenty configurations, please see this excel sheet.

Discussion

The table above shows a comparison of the results for my planner implementations. To generate the results, I used 4 degrees of freedom. I created 20 random start and end configurations and ran the four planners using them. Then, I took the average metrics over the 20 samples which are displayed above. Further details on the values generated for each of the 20 start and end configurations can be found on the excel spreadsheet linked above.

From the data above, we can see that for planning times, RRTConnect performed the best and PRM performed the worst. RRT* was the next slowest and RRT was the second fastest. Out of the three RRT algorithms, RRT* is the most computationally expensive, so it makes sense that it had the longest planning time out of the RRT algorithms. RRTConnect was able to plan the fastest since it plans from both the start and goal positions, growing two graphs until they connect.

The success rate for generating solutions in under five seconds was the same for all the algorithms. All four planners were able to generate solutions in under five seconds when using four degrees of freedom.

The average number of vertices varied between the algorithms. PRM always generates 1002 vertices since it builds the graph beforehand. I wrote the algorithm so that it would always generate 1000 feasible vertices to begin with. Then, it adds the start and goal configurations as vertices and connects them to the closest vertices in the graph. Therefore, my PRM implementation always has 1002 vertices. RRT varies in the number of vertices generated since it stops planning once the goal has been reached within a certain threshold. Additionally, in my RRT implementations I included goal biasing while sampling, which reduced the number of vertices needed. RRT* had the highest number of average vertices while RRTConnect had the least, and RRT was in between the two.

My average path quality metric calculated the total sum of joint angle movements. RRT* had the best average path quality since it had the lowest sum, so it had the lowest cost. This result makes sense because RRT* generates the most optimal paths due to the path shortening and rewiring that it does. PRM had the worst path quality. RRT and RRTConnect had very similar path qualities, but RRTConnect's was slightly better.

Conclusions

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

I think the most suitable planner for the environment is RRT*. PRM did not perform well in terms of planning time and path quality. PRM is suitable for scenarios when we have multiple start and end configurations for the same map. Because we generate the map ahead of time, PRM makes sense to run multiple queries on the map, so it can be reused since it's explicitly built. In our case, every time I run the planner script a new graph is generated. Because we aren't necessarily running multiple queries in this case, PRM does not need to be used.

RRT works well for single-shot planning. RRT does not need to create the graph before searching it. Of the three RRT implementations, RRT* produces the most optimal path. Although it takes longer and is more computationally expensive, it produces the best results. Using RRTConnect also is not a bad option if we wanted to favor speed and memory since it had the lowest planning time and least number of vertices generated. Because this case only used four degrees of freedom, RRT* produced more optimal paths and still had fast planning times and low number of vertices. Therefore, RRT* is the best choice in this case. However, if we had larger maps or a higher number of degrees of freedom, RRTConnect may be a better option.

2. What issues does that planner still have?

Although RRT* is the most suitable option for this environment, it still has some issues. It is the most computationally expensive out of the RRT options because of the rewiring process. As the dimensionality of the search space increases, the time needed to find a feasible or optimal path will also greatly increase. From the table above, we can see that RRT* did not perform well for planning time and number of vertices generated.

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

There are certain ways to improve RRT*. For example, goal-biased sampling can be used to reduce the number of nodes which need to be generated and time needed to find the goal position. Additionally, other parameters can be tuned to improve the solution, such as the epsilon distance or neighborhood size. Also, the way the cost is defined in order to rewire the paths can help produce an improved solution. Some measurements, such as euclidean distance may not work as well for this case as absolute distance may work for example. To increase speed, parallelization can be implemented using multiple threads.

Sridevi Kaza
Planning & Decision Making HW2

Extra Credit

Below are additional statistics representing the consistency of solution for a specified 4 degree of freedom start and end configuration. I ran 10 trials using the same start and goal configurations for each of the trials. I calculated the mean and standard deviation for planning time, path quality, and vertices generated. I included the total path length for each of the planners as well. The table below shows the results of the 10 trials for each planner.

From the results, it's clear that the planning times were consistently low for all the RRT variations. PRM took on average 3848 milliseconds with a standard deviation of 133 milliseconds. This is a very low standard deviation, so PRM consistently took about 3.5 seconds to plan for this configuration. The average path qualities showed that RRT* consistently was able to generate the same optimal path. The standard deviation in path quality for RRT* was 0 and the average path quality was the lowest (indicating the best cost value). The other three planners had variation in path quality, indicating less consistency in the solutions. PRM had the highest deviation for this metric. RRT* also was the most consistent when it came to path length. It had the shortest path length and 0 standard deviation. PRM also had very low standard deviation in path length, showing somewhat consistent path length in the solutions. For the number of vertices generated, PRM had zero standard deviation because it always generated the same number of vertices. However, it generates many more vertices that the RRT algorithms generate. RRT had a high level of variation in the number of vertices generated since it relies heavily on random sampling.

Metric	RRT	RRTConnect	RRT*	PRM
Average Planning Time (milliseconds)	0.000	0.000	0.000	3484.200
Planning Time Standard Deviation	0.000	0.000	0.000	133.629
Average Path Quality (Cost)	3.906	3.866	2.510	4.164
Path Quality (Cost) Standard Deviation	0.604	0.528	0.000	1.107
Average Path Length	6.700	8.100	2.000	7.700
Path Length Standard Deviation	2.214	2.132	0.000	0.823
Average Vertices Generated	15.900	9.100	12.700	1002.000
Vertices Generated Standard Deviation	10.093	2.846	4.900	0.000

Hyperparameter Tuning

For RRT, the hyperparameters I tuned are the epsilon distance, threshold distance to goal, and the probability of biasing towards the goal, and interpolation step size. RRT* also included neighborhood size and determining a cost function.

Increasing the epsilon distance helped reduce the planning time but led to paths that are less smooth. I used 0.5 for the epsilon value for all the RRT algorithms. I could have reduced this value to generate smoother paths, but I wanted to ensure fast planning times. For the distance threshold I used 1e-3 since it was close enough to 0, and I did not face any issues with that value. I used a 10% bias towards the goal state when random sampling. This means that 10% of the time, the goal state was sampled rather than a random configuration. Reducing this value led to slower planning times, but also a more uniform graph which could lead to smoother plans. I used an equation for the interpolation step size (stepsize = min(epsilon,dist) / dist). This basically takes the minimum value the arm will move and divides it by the maximum distance to create a scaling factor. For RRT*, I used 10 as the neighborhood size and euclidean distance as the cost.

For PRM, the hyperparameters involved are interpolation steps, neighborhood size, number of vertices in the graph, and the cost function for the A* search. I started with 100 as the number of steps taken during interpolation (while collision checking), but because my PRM implementation was much slower that the RRT implementations I steadily reduced this value to reduce planning time and settled on 20 steps. I played around with the neighborhood size to choose a value that was ideal. Having too small of a neighborhood led to slower planning times, but having too large of a neighborhood reduced the path quality. I found 20 to be a good size for that as well. I chose the number of vertices in the graph to be 1000. I wanted to ensure the graph was large enough that the search would not fail for higher dimension configurations. However, I did not want too many nodes where the planning would be too slow. Also, I used euclidean distance as a cost metric. Additionally, I did add in an optimization where each node could have a maximum of 10 connected edges. Reducing this number helped make the search time faster.

Compiling the Code

To compile the code the following command can be run:

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