Name: Barathkrishna Satheeshkumar
Andrew ID: bsathees

Approach

PRM

The roadmap was initially generated by randomly sampling points within the configuration space. If a sampled point was valid, it was added to the graph, and its nearest neighbors were evaluated for collision-free connections to establish edges. After constructing the roadmap, a query function introduced the start and goal nodes, ensuring appropriate edge connections. To determine the shortest path within the roadmap, Dijkstra's algorithm was applied to find a route from the start node to the goal node. Finally, a shortcutting process was employed to minimize path cost.

Hyperparameters

- **Connections**: Each node was allowed a maximum of 10 connections, except when incorporating the start and goal nodes to avoid disconnection. Increasing this limit prolonged Dijkstra's search time, while reducing it excessively resulted in suboptimal paths.
- **Max nodes**: The number of nodes was capped at 1000, based on empirical testing. Lower values occasionally led to the failure of finding a path, whereas higher values were unnecessary as fewer nodes sufficiently covered the map.

RRT

The tree was initialized with the start node, followed by random sampling. The closest existing node was identified and extended until either a predefined step size (epsilon) was reached or an obstacle was encountered. Neighboring connections were updated accordingly. A goal bias was introduced to improve the likelihood of connecting the tree to the goal. Once the tree was built, Dijkstra's algorithm identified the least-cost path from the start to the goal.

Hyperparameters

- **Goal bias**: The probability of sampling the goal node directly during random sampling was set to 0.01, ensuring the goal was eventually connected without compromising tree complexity. A higher value would cause excessive sampling of the goal node, potentially affecting tree structure.
- **Epsilon**: This parameter dictated the maximum extension length when linking a sampled node to the tree. A large epsilon reduced the tree's ability to capture environmental complexities, while a small value required significantly more nodes to cover the map.
- **Max nodes**: The number of nodes was set to 1000, based on trial and error.

RRT_Connect

Two trees were initialized, with the start node added to **tree_A** and the goal node added to **tree_B**. A random configuration was sampled, and **tree_A** extended toward it up to a maximum step size (epsilon). **Tree_B** then attempted to connect to this newly created node, with the relaxation of the epsilon constraint allowing it to connect if a collision-free path was found. If the connection was unsuccessful, the trees swapped roles, and the process continued until they merged. Once connected, Dijkstra's algorithm was independently executed on both trees to find paths to the common node, which were then combined to form the final path. Unlike standard RRT, goal biasing was unnecessary since the goal node was already assigned to a separate tree.

Hyperparameters

- **Epsilon**: Functions similarly to RRT but is ignored when **tree_B** attempts to connect.
- **Max nodes**: The total node limit for both trees combined was set to 1000. Since both trees expanded concurrently, controlling the sample size was manageable.

RRT*

Like RRT, this method began by adding the start node to the tree, followed by random sampling and incremental extension. However, unlike RRT, RRT* incorporated a rewiring process that allowed the tree to optimize its paths as the number of nodes increased. After adding a new node, neighboring nodes within a defined radius were evaluated for potential lower-cost connections. If a more efficient path was found, the tree was rewired, updating parent relationships and cost values to facilitate later path reconstruction. Goal bias was included to ensure the goal node was eventually integrated into the tree. When reconstructing the path, the node closest to the goal configuration was identified, and its parent nodes were traced back to the start. This eliminated the need for an additional A* or Dijkstra's search.

Hyperparameters

- **Epsilon**: Governs the maximum extension length, similar to RRT.
- **Goal bias**: The probability of directly sampling the goal node.
- **Gamma**: Influences the search radius for neighboring nodes during rewiring. A smaller gamma may miss lower-cost paths, while a larger gamma increases computation time due to additional node evaluations.
- **Max nodes**: Limited to 1000, based on testing.

Data Structures

State Representation

A custom struct was used to represent nodes:

```
C/C++
struct node {
    int id;
    std::vector<double> angles;
    std::vector<std::pair<int, double>> neighbors;
    double g;
    bool closed;
    int parent;

    node() {
        this->id = -1;
        g = std::numeric_limits<double>::infinity();
        closed = false;
        parent = -1;
    }
};
```

Planners

Planner classes were implemented with member functions handling pathfinding operations and member variables storing necessary data (e.g., `std::mt19937 generator`, `numofDOFs`).

- **RRT Planner**

```C/C++
RRT_Planner rrt(x_size, y_size, numofDOFs, map, eps);
```

- **RRT-Connect Planner**

```C/C++
RRT_Connect_Planner rrt_connect(x_size, y_size, numofDOFs, map,
eps);
```

- **RRT Planner***

```C/C++
RRT_Star_Planner rrt_star(x_size, y_size, numofDOFs, map, eps,
armgoal_anglesV_rad);
```

- **PRM Planner**

```C/C++
PRM_Planner prm(x_size, y_size, numofDOFs, map);
```

Compilation Instructions

To compile and execute the program, run the following commands in the terminal:

```Unset
g++ planner.cpp -o planner.out
g++ verifier.cpp -o verifier.out
g++ config_checker.cpp -o config_checker.out
<delete vertices.txt file>
python results.py
python vertices.py
```

- The config_checker file makes sure that the randomly generated start and goal nodes have valid configurations
- The results.py file compiles the path_cost, time etc. into a results.csv file
- The vertices.py appends the mean and standard deviation of the vertices to and updated_results.csv file

Note: The python files are simply for compiling the results into an easy-to-read form. They have also been included in the submission. The verifier.out and config_checker.out are also needed only to run these python scripts.

Test Results

Five distinct test cases were generated for testing on Map 2. In each case, the number of DOFs, start configurations, and goal configurations were randomly generated and validated using the `config_checker` file. The test cases and their results are presented below:

Test Cases

Test Index	Num of DOFs	Start Config	Goal Config
0	5	0.48,3.58,1.32,5.43,0.91	1.82,3.23,0.55,1.37,0.39
1	5	0.48,2.27,1.87,0.46,5.34	1.69,2.06,0.66,4.01,0.16
2	3	1.23,2.39,0.42	1.76,1.85,0.8
3	3	1.2,2.6,3.12	1.79,2.26,3.78

Results

Planner ID	Test Case	Mean Time (s)	Std Time (s)	Mean Cost (rad)	Std Cost (rad)	Success Rate	Mean Vertices	Std Vertices
RRT	0	0.32	0	13.3	2.37	1	1001	0
RRT Connect		0.01	0	20.06	5.76	1	14.5	3.840573
RRT*		0.55	0.01	21.1	7.58	1	1001	0
PRM		1.31	0.01	9.07	0.61	1	1002	0
RRT	1	0.33	0.01	14.24	4.58	1	1001	0
RRT Connect		0.01	0	12.2	3.91	1	10	10.3923
RRT*		0.57	0.01	12.11	3.56	1	1001	0

PRM		1.3	0.01	11.31	0.65	1	1002	0
RRT	2	4.37	1.64	1.45	0	1	1001	0
RRT Connect		0.01	0	4.08	1.78	1	4	0
RRT*		3.3	0.68	1.45	0	1	1001	0
PRM		0.86	0.01	1.45	0	1	1002	0
RRT	3	3.76	0.57	1.59	0	1	1001	0
RRT Connect		0.01	0	2.84	0.79	1	4	0
RRT*		3.18	0.6	2.26	1.15	1	1001	0
PRM		0.83	0	1.59	0	1	1002	0
RRT	4	3.73	0.56	1.35	0	1	1001	0
RRT Connect		0.01	0	2.56	0.97	1	4.5	0.866025
RRT*		3.43	0.61	1.35	0	1	1001	0
PRM		0.86	0.01	1.35	0	1	1002	0

Conclusions

Key Observations

a. Execution Time

- Planner 1 consistently exhibited the lowest mean execution time, averaging around 0.01 seconds across all test cases.
- Planner 3 had the second-best performance, with execution times ranging from 0.85 to 1.32 seconds.
- Planner 0 and Planner 2 had significantly higher execution times, with times ranging between 0.3 and 4.64 seconds.

b. Cost Efficiency

- Planner 1 achieved the lowest mean cost in most cases, indicating efficient path generation.

- Planner 3 also demonstrated good cost efficiency, but it occasionally generated higher costs in some cases (e.g., problem index 3, where it had a cost of 10.55).
- Planner 2 produced paths with the highest mean cost, which suggests that it was less efficient in finding optimal routes.

c. Success Rate

- All planners had a 100% success rate, meaning they successfully solved all test cases. However, this does not account for efficiency.

d. Number of Vertices Used

- Planner 0 and Planner 2 used a significantly larger number of vertices (1001) in almost all cases, suggesting that they rely on denser sampling, which likely contributes to their higher execution times.
- Planner 1 used the fewest vertices (averaging 5.5 to 25 vertices), making it the most computationally efficient.
- Planner 3 used around 1002 vertices, similar to Planner 0 and Planner 2, but achieved better execution times than Planner 2.

Selecting the Most Suitable Planner

Based on the observations:

- Planner 1 appears to be the best choice for this environment because it achieves the lowest execution time, the lowest cost in most cases, and requires significantly fewer vertices to find a solution.
- Planner 3 is a strong alternative, as it performs significantly better than Planner 0 and Planner 2 in execution time while maintaining relatively low costs.

Issues and Potential Improvements

a. Issues Identified

- Planner 1, while the fastest, may not always find the absolute optimal path. Its low number of vertices suggests that it relies on minimal sampling, which may sometimes overlook better paths.
- Planner 3, while efficient, still uses a high number of vertices, which increases memory usage.

b. Possible Improvements

- For Planner 1: Consider fine-tuning the sampling density to ensure that speed does not come at the expense of path quality.
- For Planner 3: Implement adaptive sampling techniques to reduce the number of vertices without compromising path quality.
- For Planner 2 and Planner 0: Optimize their graph expansion strategy to reduce unnecessary computations and improve execution time.