Sumeet Shanbhag

Motion Planning Assignment 4

Code Explanation:

check_collision() : The method calculates the change in x and y coordinates between the two points, determines the direction of movement (positive or negative) for each axis, and sets the starting point to p1. It then checks each cell along the path between p1 and p2, starting at p1. If a cell along the path is found to be an obstacle or if the path goes out of the grid boundaries, the method returns True indicating a collision. Otherwise, if the entire path is traversed without encountering any obstacles or going out of the grid boundaries, the method returns False, indicating that there is no collision between the two points.

The uniform_sample() method generates a specified number of sample points in a two-dimensional space using uniform distribution. It clears the graph, generates sample points, checks for obstacles at each sample point, and stores only the valid points in the samples list. The method returns a list of valid sample points as tuples of (row, col) values.

The random_sample() method is a Python code that generates n_pts number of random points in a two-dimensional environment and stores valid points that do not collide with obstacles. The method checks each randomly generated point for collision with obstacles and appends the valid points to self.samples. This method can be used for random sampling-based motion planning applications.

The gaussian_sample() method first initializes the graph and generates random points using the 'sample_pts' function. Then, it generates random points at a certain distance from the previously generated points, using a Gaussian distribution with a standard deviation of 10. The newly generated points are checked to see if they are too close to obstacles. If a point is too close to an obstacle, only the original point is added to the list of valid samples. If neither point is too close to an obstacle, neither point is added to the valid samples list.

The bridge_sample() method generates random points and uses bridge sampling to find valid midpoints between pairs of points that form a "bridge" between obstacles. The midpoint is appended to a list of valid samples if it is not an obstacle and lies within the map boundaries. The method is useful for generating samples in narrow passages or other regions of high curvature where traditional sampling techniques may fail to generate enough valid samples.

The sample_pts() takes two arguments: n_pts, the number of points to generate, and random, a boolean flag for generating either random or uniform points. The function generates random points using numpy's randint function if random is True. If random is False, it generates uniform points using numpy's linspace and meshgrid functions. The function returns two arrays of length n_pts each, representing the x and y coordinates of the generated sample points.

The add_weighted_edges() method calculates the weight and adds the weighted edge which will be used in the search method later on.

The sample() method first initializes the sample points, graph, and path. Then, depending on the chosen sampling method, it samples n_pts points in the configuration space. It then uses a k-d tree data structure to find the num_neighbors nearest neighbors for each point and

connects them with weighted edges. The pairs of connected points and their weights are stored in the 'pairs' list. Finally, the method adds nodes and edges to the graph using the 'pairs' list and prints out the number of nodes and edges in the constructed graph.

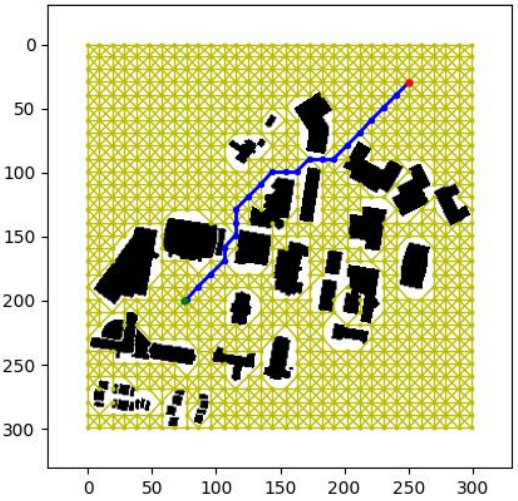
The search() method temporarily adds the start and goal nodes to the self.graph object and finds the nearest neighbors of the nodes using a KDTree. It then creates pairs of points that need to be connected, computes the distance/weight, and adds the weighted edges to the graph using the add_weighted_edges function. It then uses the Dijkstra algorithm to find the shortest path between the start and goal nodes in the graph. If a path is found, it is stored in the self.path variable, and the function calls draw_map to visualize the result. Finally, the temporary nodes and edges are removed from the graph

Algorithm Explanation and results:

Probabilistic Roadmap Algorithm: The PRM algorithm is a sampling-based approach that constructs a roadmap of the configuration space, which is the space of all possible configurations of the robot or object. The roadmap consists of a set of nodes connected by edges, where each node represents a configuration and each edge represents a feasible path between two configurations.

Uniform sampling:

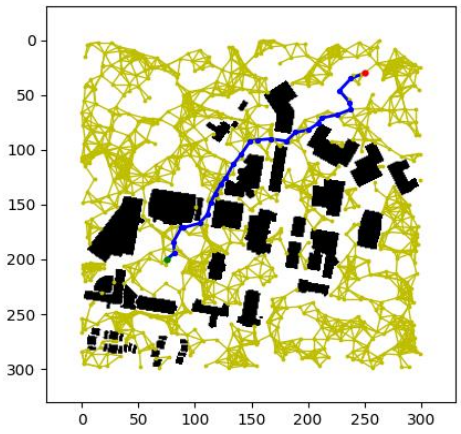
Uniform sampling involves randomly selecting points from a predefined space, such that each point has an equal probability of being selected. The advantage of uniform sampling is that it is easy to implement and can be used for a wide range of problems. However, the disadvantage of uniform sampling is that it may not generate points in areas of high density, which can lead to poor coverage of the space.




```
The constructed graph has 838 nodes and 2740 edges
The path length is 264.06
```

Random sampling:

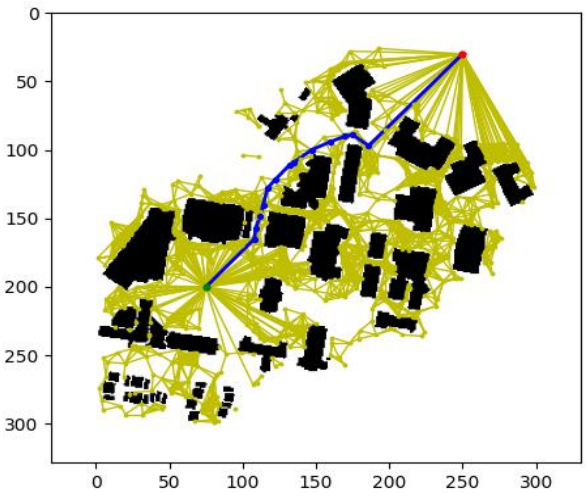
Random sampling involves selecting points from a predefined space with a random distribution. The advantage of random sampling is that it can generate points in areas of high density, which can improve coverage of the space. However, the disadvantage of random sampling is that it may not be suitable for problems that require a specific distribution.




```
The constructed graph has 839 nodes and 2713 edges
The path length is 285.75
```

Gaussian sampling:

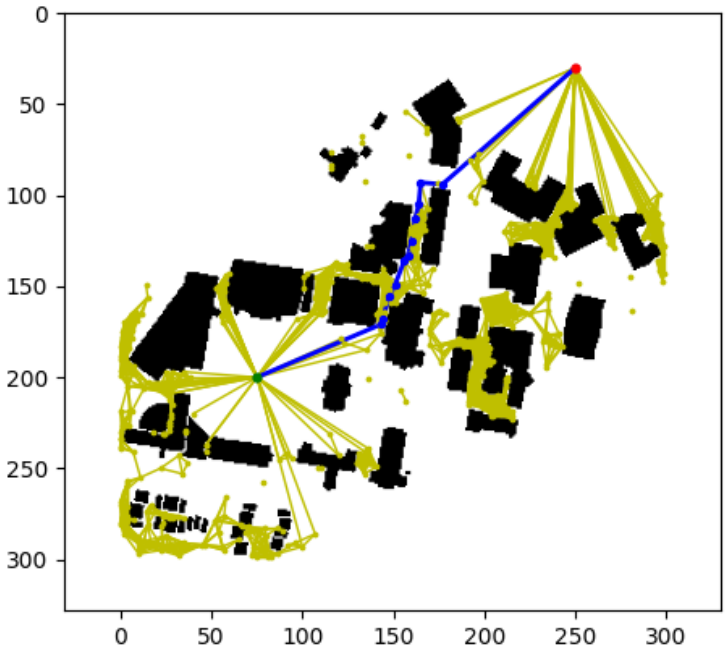
Gaussian sampling involves generating points with a Gaussian distribution centered around a mean point. The advantage of Gaussian sampling is that it can be used to generate points in areas of high density and can be used to mimic natural distributions. However, the disadvantage of Gaussian sampling is that it can be computationally expensive, particularly when generating points with high dimensionality.



```
The constructed graph has 638 nodes and 2615 edges
The path length is 257.70
```

Bridge sampling:

Bridge sampling involves generating points in a way that bridges the gap between two or more subspaces. The advantage of bridge sampling is that it can be used to connect subspaces that may not be directly connected. However, the disadvantage of bridge sampling is that it can be difficult to implement, particularly when dealing with high-dimensional spaces.




```
The constructed graph has 553 nodes and 3892 edges
The path length is 264.76
```

In summary, the choice of sampling method depends on the problem at hand, and each method has its own advantages and disadvantages. Uniform sampling is easy to implement but may lead to poor coverage of the space. Random sampling is suitable for problems that require a specific distribution and can generate points in areas of high density. Gaussian sampling can generate points with a natural distribution but can be computationally expensive. Bridge sampling can be used to connect subspaces but can be difficult to implement, particularly in high-dimensional spaces.

The above results will be generated for the values given below. As we increase n_pts and also increase the value of num_neighbors, the probability of finding a path increases. If the value is too small, the algorithms might not be able to find a path as the sample points are spaced too wide apart and might not be able to detect a path.

```
# Search with PRM
PRM_planner.sample(n_pts=1000, sampling_method="uniform")
PRM_planner.search(start, goal)
PRM_planner.sample(n_pts=1000, sampling_method="random")
PRM_planner.search(start, goal)
PRM_planner.sample(n_pts=4000, sampling_method="gaussian")
PRM_planner.search(start, goal, num_neighbors=200)
PRM_planner.sample(n_pts=40000, sampling_method="bridge")
PRM_planner.search(start, goal, num_neighbors=200)
```