Assignment 2: Planning for High-DOF Planar Arm

Name: Aman Manish Chulawala
Andrew ID: `achulawa`

1 Algorithm Discussion

1.1 RRT Implementation

The plannerRRT function utilizes the Rapidly-exploring Random Tree (RRT) algorithm to discover a path from a given start configuration (armstart_anglesV_rad) to a desired goal configuration (armgoal_anglesV_rad) within a specified environment map (map). It operates in a 2D space defined by the dimensions x_size and y_size. The robot arm is characterized by numofDOFs degrees of freedom.

The function employs a probabilistic approach to iteratively extend the tree of sampled configurations towards the goal. It sets a step size (stepSize) for each extension, and limits the number of iterations to a maximum value (maxIterations) to avoid indefinite computation. If a valid path is found, the resulting plan is returned through the plan and planlength pointers. If no path is found, the function recursively restarts the planning process. This ensures that a solution is always found. Max Iteration limit is set to 30000 and step size is set to 0.15.

1.2 RRT Connect Implementation

The plannerRRTConnect function employs the RRT-Connect algorithm to discover a path from a specified start configuration (armstart_anglesV_rad) to a desired goal configuration (armgoal_anglesV_rad) within a given environment map (map). Operating in a 2D space defined by dimensions x_size and y_size, the robot arm has numofDOFs degrees of freedom.

This function initializes two trees with the start and goal configurations and iteratively extends them towards each other with a defined step size (stepSize). It limits the number of iterations to a maximum value (maxIterations) to avoid indefinite computation. If a valid path is found, the resulting plan is returned through the plan and planlength pointers. If no path is found, the function returns NULL plan and a plan length of 0. Max Iteration is set to 100000 and interpolation step size during extendTree is set to 0.05.

1.3 RRT Star Implementation

The plannerRRTStar function employs the RRT* (Rapidly-exploring Random Tree Star) algorithm to find an optimal path from a specified start configuration (armstart_anglesV_rad) to a desired goal configuration (armgoal_anglesV_rad) within a given environment map (map). It operates in a 2D space defined by dimensions x_size and y_size, with the robot arm possessing numofDOFs degrees of freedom.

This function initializes a tree with the start configuration and iteratively extends it towards the goal. It sets a step size (stepSize) for each extension and limits the number of iterations to a maximum value (maxIterations) to avoid indefinite computation. If a valid path is found, the resulting plan is returned through the plan and planlength pointers. If no path is found, the function recursively restarts the planning process. Max Iterations are set to 5000 and stepSize is 0.25.

1.4 PRM Implementation

The plannerPRM function utilizes the Probabilistic Roadmap (PRM) algorithm to discover a path from a specified start configuration (armstart_anglesV_rad) to a desired goal configuration (armgoal_anglesV_rad) within a given environment map (map). It operates in a 2D space defined by dimensions x_size and y_size, with the robot arm having numofDOFs degrees of freedom.

This function generates a PRM roadmap with a specified number of nodes (numNodes) and considers a given number of nearest neighbors (k). It then identifies the start and goal nodes within the roadmap and finds a path connecting them. The resulting plan is returned through the plan and planlength pointers. Number of nodes for the planner is set to 5000 and each node connects to 3 neighbours.

2 Results

The table below summarises the performance of the four planners for 20 different problem sets. The problem sets were generated using **generateValidSamples()** and paired randomly. The tests used are available in grader.py.

Problem	Average Steps	Average Cost	Average Time	Success Rate (<5s)
RRT	60.60	11.428	0.0838	1
RRT Connect	34.05	11.368	0.0364	1
RRT Star	34.35	10.493	0.0175	1
PRM	10.90	13.385	0.1151	1

Table 1: Comparison of Planning Algorithms

The RRT Star implementation is the most effective of the the four planners, giving the least time and cost solution. RRT and RRT Connect were the next best in terms of implementation. PRM was not as good in terms of cost and time, but performed consistently better in terms of average steps to reach the solution. The cost for PRM, was almost on par with the RRT variants. The time of execution for PRM is a direct function of PRM roadmap generation. Reducing the amount of nodes significantly decreases the computation time.

3 Discussion

3.1 What planner you think is the most suitable for the environment and why?

RRT Star seems to be performing the best in the environment in terms of the computation time. The nature of the problem is such that RRT Star was always expected to give the best result.

3.2 What issues that planner still has?

The planners still have sub optimal parameters and require more tuning, especially PRM. It is my belief that the number of nodes can be further reduced through more testing, and this will bring the computation time close to the RRT variants.

Apart from this, the planner ends once it finds a solution. There are no attempts to refine the path because of its time greedy nature. This delivers a suboptimal solution in the minimum time.

3.3 How you think you can improve that planner?

A time cap of 5 seconds can be implemented to improve the quality of the path. While path shortcutting was implemented for PRM, the results were not optimal enough to warrant the extra computation. This can be improved and implemented for the other planners as well. A strong trait of the planners is that they **Guarantee** a solution, but this takes more computation as the planning structures are deemed too noisy after a point and discarded before starting the planner again. This can be improved.

4 Extra Credit

A total of 4 runs were performed with the planners and the statistics of each grader script is listed below.

Problem	Avg Steps	StD Steps	Avg Cost	StD Cost	Avg Time	StD Time	Success
RRT	56.525	6.160	11.388	0.404	0.106	0.051	1
RRTConnect	36.512	2.070	11.670	0.348	0.046	0.008	1
RRTStar	34.525	0.344	10.393	0.237	0.020	0.005	1
PRM	12.238	2.882	13.674	2.412	0.234	0.119	1

Table 2: Comparison Across Multiple Runs

The RRT Planner occasionally generates too many bad configurations and needs to dump the result and restart. This makes it have a slightly higher average time in terms of the planners. RRT Connect and RRT Star on the other hand show great behaviour under multiple runs. Each planner was executed a total of 80 times (20 problems ran 4 times). A solution was always provided within 5 seconds.

5 Helper Functions and Structures

- The **Node struct** represents a node in a graph. It contains information about its angles, parent node, cost, and a list of neighboring nodes. The angles are stored as a pointer to an array of doubles. Each node can have a parent node and an associated cost. Additionally, it can have a list of neighboring nodes represented as a vector of pointers to Node objects. Various definitions are possible for this struct. RRT and RRT Connect just need the parent node and angle array. RRT Star needs a list of nearby neighbour's in addition to the basic data, and PRM needs an associated cost for backtracking. Various constructors are defined to allow this.
- **getRandomConfiguration()** function generates a random valid configuration for a given number of degrees of freedom (numofDOFs). It uses a Mersenne Twister engine for random number generation within the range of 0 to 2π. The resulting configuration is returned as a dynamically allocated array of doubles.
- **getRandomConfiguration_Biased()** generates a random valid configuration for a given number of degrees of freedom (numofDOFs). It includes a bias towards the goalConfig based on the tuning parameter. The resulting configuration is returned as a dynamically allocated array of doubles The bias value is set to 0.35 (35 percent chance of choosing the goal node).
- The **nearestNeighbor()** function identifies the closest node in a provided tree to a given sample configuration. It operates by computing the Euclidean distance between the angles of the sample configuration and each node in the tree. The function then returns a pointer to the closest node.
- The **extendTree()** function extends a given tree towards a randomly sampled configuration. It does so by calculating a new configuration based on a specified step size. The function then creates a new node using the calculated angles, with the nearest node as its parent.
- **extendTree_RRTConnect** takes in two vectors of pointers to Node objects, treeFrom and treeTo, as well as several other parameters. The function generates a random configuration and extends treeFrom towards the random configuration using the extendTree function. If the new node has a valid configuration, the function finds the nearest node in treeTo to the new node and interpolates towards the new node in treeFrom in step sizes until either the new node is reached or an obstacle is hit. Each interpolated configuration is checked for validity using the IsValidArmConfiguration function. If a valid configuration is found, a new node is created with the interpolated angles and added to treeTo.
- **isValidEdge()** takes in two pointers to Node objects, node1 and node2, as well as a pointer to a 2D array of doubles called map, and the dimensions of the map, x_size and y_size. The function checks if the edge between node1 and node2 is valid, meaning that it does not intersect with any obstacles in the map. It calculates the (x, y) coordinates of each joint in the robot arm using the angles of the two nodes and the $LINKLENGTH\_CELLS$ constant. It checks if the line segment between each pair of adjacent joints is valid using the IsValidLineSegment function. If any of the line segments are invalid, the function returns false. If all of the line segments are valid, the function returns true.
- **findNearNodes()** is a function that takes in a vector of pointers to Node objects, tree, as well as a pointer to a Node object, q, and a constant double value, stepSize. The function returns a vector of pointers to Node objects that are within a certain distance of q. The function calculates the Euclidean distance between q and each node in tree, and adds the node to the output vector if the distance is less than stepSize.
- **distance()** is a function that takes in two pointers to double arrays, point1 and point2, as well as an integer numofDOFs. The function calculates the Euclidean distance between the two points in numofDOFs-dimensional space. It iterates over each dimension and calculates the difference between the two points in that dimension. It then squares the difference and adds it to a running total. Finally, the function returns the square root of the total, which is the Euclidean distance between the two points.
- **findNearestNeighbors()** is a function that takes in a pointer to a Node object, node, a vector of pointers to Node objects, roadmap, an integer k, and an integer numofDOFs. The function finds the k nearest neighbors of node in roadmap based on Euclidean distance. It calculates the distance between

node and each node in roadmap using the distance function, and stores the distances in a vector. It then sorts the vector of distances and adds the k nearest nodes to the nearestNeighbors vector. The function returns the nearestNeighbors vector.

- **generatePRMRoadmap()** is a function that generates a Probabilistic Roadmap (PRM) for a robot arm. The function takes in several parameters, including the number of nodes to generate, the number of nearest neighbors to consider, a pointer to a 2D array of doubles called map, and the dimensions of the map, x_size and y_size. The function generates numNodes random configurations and checks if each configuration is valid using the IsValidArmConfiguration function. If a configuration is valid, a new Node object is created with the angles of the configuration and added to the roadmap vector. The function then generates edges between the nodes in the roadmap vector by finding the k nearest neighbors of each node and checking if the edge between the two nodes is valid using the isValidEdge function.
- **dijkstra()** and **astar()** are two function which traverse through the generated roadmap, looking for a path connecting the start and goal position
- **shortCut()** is an optional function that takes in a vector of pointers to double arrays, path, a pointer to a 2D array of doubles called map, and the dimensions of the map, x_size and y_size, as well as an integer numofDOFs. The function generates a shortcut path by removing any unnecessary nodes from the input path. It starts by adding the first node in path to the shortcutPath vector. It then iterates over each node in path and checks if the edge between the previous node and the next node is valid using the isValidEdge function. If the edge is valid, the function skips the current node and moves on to the next node. If the edge is not valid, the function adds the current node to the shortcutPath vector. Finally, the function adds the last node in path to the shortcutPath vector and returns it.
- **generateValidSamples()** was a one time use function created to generate 40 valid configurations that could be used to test the planners.