Assignment 2

Pradeep Roy , Hriday Purohit

March 2024

1 Nearest neighbors with linear search approach (10 pts)

1.1 Implementation Description

The Python script `nearest_neighbors.py` implements the nearest neighbors algorithm. The script takes four inputs: the type of robot (either "arm" or "vehicle"), the target configuration, the number of nearest neighbors to output, and a file containing random configurations. The implementation utilizes the argparse module for parsing command-line arguments and numpy for numerical computations.

The `NearestNeighbour` class contains methods for parsing inputs, computing distances between configurations, finding the nearest neighbors, and visualizing the results using a 3D visualization tool. The `input_parser` method parses command-line arguments, reads configurations from the specified file, and returns the parsed arguments and configurations. The `get_config_distance` method calculates the distance between two configurations based on the type of robot.

The `get_nearest_neighbors` method computes the distances between the target configuration and all configurations in the list, sorts them, and returns the k nearest neighbors. The `visualize` method visualizes the target configuration and its nearest neighbors using a 3D visualization tool.

The main section of the script reads configurations from a file, parses command-line arguments, computes the nearest neighbors, and visualizes the results.

1.2 Topology of Robot's Configuration Space

This distance metric is appropriate because the configuration space of the robotic arm has a Euclidean topology. The joint angles q1, q2, and q3 can vary continuously, and the configuration space is simply a 3D Cartesian space.It represents the length of the straight line connecting the two configurations in the 3D configuration space.

For the "vehicle" robot, which has seven parameters representing translation and rotation, the configuration space is higher-dimensional. The specific topology of this space depends on the kinematic constraints of the robot.

1.3 Distance Metric between Configurations

The distance metric between two configurations depends on the type of robot. For the "arm" robot, the distance metric is the Euclidean distance between the vectors representing the joint angles of the configurations. This metric measures the straight-line distance between two points in the three-dimensional configuration space of the robot.

For the "vehicle" robot, the distance metric is a suitable metric that would involve considering both translational and rotational differences between configurations.

1.4 command to run

```
python ./py160-hp580/nearest_neighbors.py --robot arm --target -0.39 -2.09 -2.09
-k 3 --configs "./py160-hp580/configs.txt"
```

1.5 visualizations

1.5.1 arm 1

The following table lists the three nearest neighbors for the target configuration $[0.0, 0.0, 0.0]$:

Index	Joint 1	Joint 2	Joint 3
1	0.2618	−0.7854	1.0472
2	0.5236	−1.0472	−1.3962
3	−0.7854	−0.7854	−1.5708

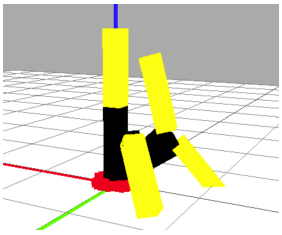

Figure 1: Target : [0.0, 0.0, 0.0]

1.5.2 arm 2

The following table lists the three nearest neighbors for the target configuration $[-2.44, 0.39, 1.22]$:

Index	Joint 1	Joint 2	Joint 3
1	−2.3252	0.7854	1.5717
2	−2.9671	0.1309	0.3927
3	−2.0944	−1.0472	1.5708

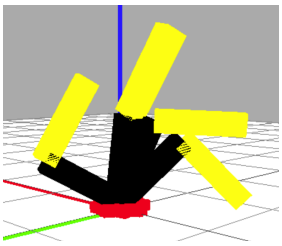

Figure 2: Target : [-2.44, 0.39, 1.22]

1.5.3 arm 3

The following table lists the three nearest neighbors for the target configuration $[1.047, -1.57, 0.78]$:

Index	Joint 1	Joint 2	Joint 3
1	1.2217	−2.0944	0.6545
2	1.0472	−2.0944	1.5708
3	0.2618	−0.7854	1.0472

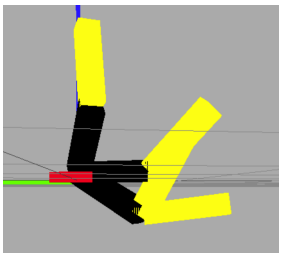

Figure 3: Target : [1.047, -1.57, 0.78]

1.5.4 arm 4

The following table lists the three nearest neighbors for the target configuration $[-1.22, 2.09, 1.57]$:

Index	Joint 1	Joint 2	Joint 3
1	−1.0472	1.5708	1.0472
2	−0.3927	2.4435	0.2618
3	−2.3252	0.7854	1.5717

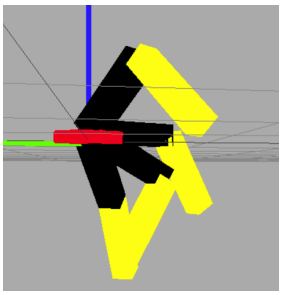

Figure 4: Target : [-1.22, 2.09, 1.57]

1.5.5 arm 5

The following table lists the three nearest neighbors for the target configuration $[-0.39, -2.09, -2.09]$:

Index	Joint 1	Joint 2	Joint 3
1	−0.7854	−0.7854	−1.5708
2	0.5236	−1.0472	−1.3962
3	−1.5708	−1.0472	−2.0944

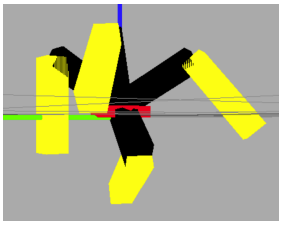

Figure 5: Target : [-0.39, -2.09, -2.09]

1.5.6 free body

check figure 6

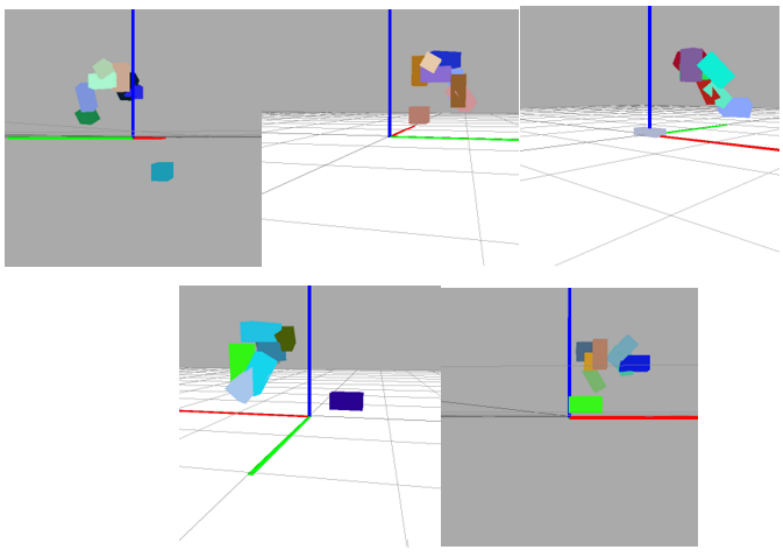

Figure 6: Free body nearest neighbours

2 PRM (25 pts)

```
python .\py160-hp560\prm.py --robot arm --start -2 -2.5 0 --goal
2.0 1.5 -3.0 --map ".\py160-hp560\arm_polygons.txt"
```

2.1 PRM Implementation

The PRM algorithm is implemented in the PRM class, which has the following key components:

2.1.1 Obstacle Handling

1 The obstacles in the environment are read from the specified file `arm_polygons.txt` and stored as a list of spheres, where each sphere is represented by its center coordinates (x, y, z) and radius (r). The obstacles are then visualized using the `threejs_group` module. AABB collision checking is performed for all the rigid boides to check the configurations are not in collision space

2.1.2 Graph Construction

The PRM graph is constructed using the Graph class, which maintains a dictionary of nodes (configurations) and their connections (edges). The graph is built by randomly sampling configurations and adding them as nodes, as long

as they are collision-free. The algorithm then connects each new node to its k nearest neighbors, where k is set to 6 in this implementation.

2.1.3 Path Search

To find the path from the start configuration to the goal configuration, the code uses a discrete search algorithm, in this case, A* search. The A* algorithm is implemented using a priority queue, where the priority is determined by the sum of the actual cost to reach the current node and an estimated heuristic cost to the goal. The heuristic used in this implementation is the Euclidean distance between the current node and the goal. `is_heuristic` command is set as `True` when A* algorithm needs to be implemented or if its set to false Dijskstra's algorithm is implemented

2.2 Design Choices

Nearest Neighbor Search:

- The `NearestNeighbour` class is used to efficiently find the k nearest neighbors of a given configuration. This is a crucial step in the PRM algorithm, as it determines the connectivity of the graph.

Path Search Algorithm:

- The A* search algorithm and Dijkstra's algorithm are used to find the shortest path between the start and goal configurations.A* is a well-known and efficient algorithm for this type of problem, as it can find the optimal path while considering the heuristic cost to the goal. The `is_heuristic` command is set as `True` when the A* algorithm needs to be implemented, or if it's set to `False` Dijkstra's algorithm is implemented.

2.3 Conclusion

The implemented PRM algorithm successfully finds a collision-free path for the robotic arm from the given start configuration to the goal configuration, while avoiding the obstacles in the environment. The design choices made in the implementation, such as the obstacle representation, nearest neighbor search, and path search algorithm, are well-suited for the problem at hand and result in an efficient and effective solution.The .mp4 videos can be found in the /out/ass_2/prm_1,2.mp4 folder

2.4 visulization

2.4.1 Run 1 :

Final Path between :
Start configuration: $[0.0, 0.0, 0.0]$ and Goal configuration: $[1.0, 1.5, 2.0]$
$[0.0, 0.0, 0.0]$

[0.26197185, 0.06783544, 0.98832054]
[0.05601653, 1.09336798, 1.10827669]
[0.15921605, 1.55427258, 1.82853545]
[1.0, 1.5, 2.0]

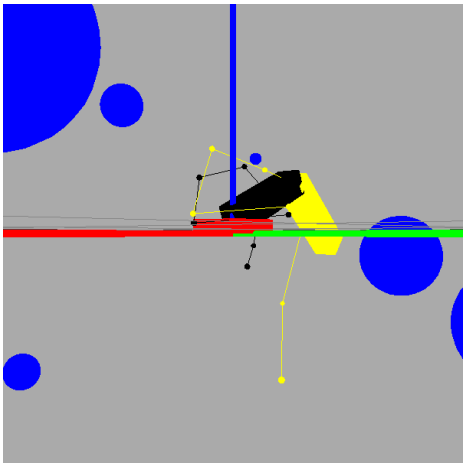

Figure 7: PRM Run 1

2.4.2 Run 2

Final Path between :
Start configuration: $[-2.0, -2.5, 0.0]$ and Goal configuration: $[2.0, 1.5, -3.0]$
$[-2.0, -2.5, 0.0]$
$[0.26836096, -2.07514034, -0.40144743]$
$[-0.02702836, -0.90962346, -1.31933721]$
$[-0.0336966, -0.30175138, -2.57197673]$
$[-0.16962538, 0.80278603, -2.5589348]$
$[0.04254466, 1.65117746, -2.98636353]$
$[2.0, 1.5, -3.0]$

3 PRM* and Comparison of Planners (25 pts)

3.1 PRM* Algorithm

The PRM* algorithm is an extension of the PRM algorithm that aims to improve the quality of the roadmap and the resulting path. The key differences between PRM and PRM* are:

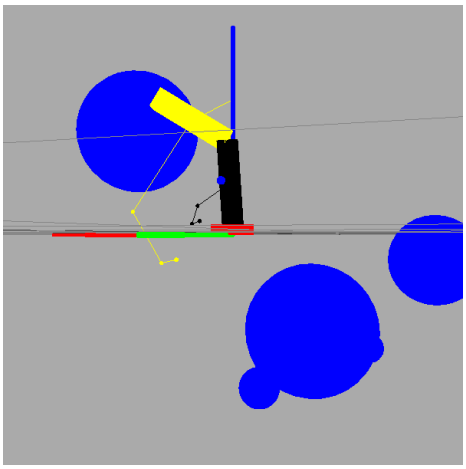

Figure 8: PRM run 2

3.1.1 Neighbor Selection

In the PRM* algorithm, the number of neighbors for each new node is not a constant k, but rather a function of the number of nodes in the graph. Specifically, the number of neighbors is proportional to $\log(N)$, where N is the number of nodes in the graph.

3.2 visulization

The .mp4 videos can be found in the /out/ass_2/prm_star_1,2.mp4 folder

3.2.1 Run 1

Start configuration: $[0.0, 0.0, 0.0]$ and Goal configuration: $[1.0, 1.5, 2.0]$
$[0.0, 0.0, 0.0]$ $[-0.00819641, 0.34615266, 0.75448349]$
$[0.07315106, 0.66016948, 0.80624265]$
$[0.00406596, 1.34662715, 0.93510644]$
$[0.13999086, 1.45884551, 1.24346878]$
$[0.42802142, 1.41566512, 1.46044258]$
$[1.0, 1.5, 2.0]$

3.2.2 Run 2

Start configuration: $[-2.0, -2.5, 0.0]$ and Goal configuration: $[2.0, 1.5, -3.0]$
$[-2.0, -2.5, 0.0]$
$[0.76675062, -2.53866634, -0.41364908]$
$[0.75413178, -1.66179708, -1.26808659]$
$[1.18950771, -0.57327433, -1.92540031]$

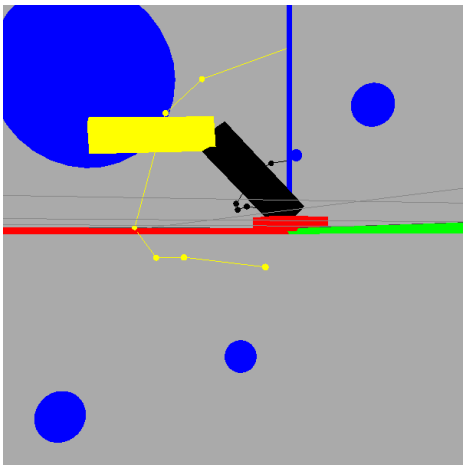

Figure 9: PRM* Run 1

$[2.05716214, -0.28058342, -3.08531971]$
$[2.65643957, -0.54354383, -2.98771493]$
$[2.97121233, -0.0437303, -2.57234223]$
$[3.10041994, 0.80729075, -2.9336598]$
$[3.13146821, 2.13934325, -3.04464212]$
$[2.0, 1.5, -3.0]$

We conducted experiments in 5 different environments, each with 5 pairs of start and goal configurations. For each environment and start/goal pair, we executed both the PRM and PRM* planners 10 times, with a maximum of 500 iterations.

The key metrics we measured are:

- Success rate: All the runs have been successful in reaching the goal states
- Path quality: The quality of the path can be argued based on the randomization and the obstacle locations. Multiple runs from the same start/goal configuration produced different results
- Computation time: The table below gives a detailed information about each and every run for the arm

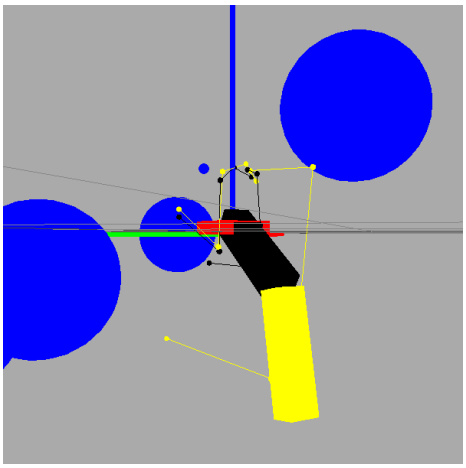

Figure 10: PRM* Run 2

Iteration	Alg	start	goal	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Mean	Best
	PRM	[0, 0, 0]	[1. , 1.5, 2.]	516.11	690.7	468.91	492.35	453.19	466.75	474.21	488.77	494.95	515.18	506.11	
1	PRM*	[0, 0, 0]	[1. , 1.5, 2.]	386.68	588.73	437.37	390.91	383.22	366.05	367.18	395.78	397.8	384.75	409.85	PRM*
	PRM	[-2. -2.5 0.]	[2. 1.5 -3.]	783.3	581.19	639.06	533.8	806.58	562.34	532.93	485.75	448.08	448.49	582.15	
2	PRM*	[-2. -2.5 0.]	[2. 1.5 -3.]	602.16	493.59	499.03	536.74	415.52	627.27	552.79	408.14	422.45	395.65	495.33	PRM*
	PRM	[0. 0. 0.]	[2. 1.5 -3.]	460.14	508.56	488.73	459.15	443.46	455.86	534.15	615.65	456.1	437.98	485.98	
3	PRM*	[0. 0. 0.]	[2. 1.5 -3.]	378.81	410.19	369.07	427.05	361.8	362.8	368	365.26	479.26	496.48	401.87	PRM*
	PRM	[1. 1.5 2.]	[2. 1.5 -3.]	735.23	787.39	738.68	718.89	541.07	543.04	501.81	597.72	449.24	450.14	606.32	
4	PRM*	[1. 1.5 2.]	[2. 1.5 -3.]	568.72	657.34	648.01	598.89	549.82	451.89	437.75	401.07	466.26	443.84	522.36	PRM*
	PRM	[1. 1.5 2.]	[0. 0. 0.]	546.9	513.25	44.22	42.9	40.4	38.58	37.67	39.95	37.6	37.55	137.9	
5	PRM*	[1. 1.5 2.]	[0. 0. 0.]	434.02	429.94	402.44	31.11	31.73	30.28	30.33	30.27	30.07	29.97	148.02	PRM

Figure 11: Computation times

4 Motion Planning via Potential Functions (25 pts)

4.1 Implementation Description

We have implemented a motion planning algorithm using potential functions and gradient descent in the Python script `potential.py`. The implementation utilizes argparse for parsing command-line arguments, numpy for numerical computations, and the `ModifiedRoboticArm` and `threejsgroup` classes for visualization.

The `PotentialPlanner` class contains methods for computing attractive and repulsive potentials, evaluating the overall potential function, performing gradient descent, and visualizing the resulting path. The attractive potential is defined with respect to the goal, and the repulsive potential is defined with respect to the obstacles. The gradient descent algorithm iteratively updates the configuration based on the gradient of the potential function until convergence.

The main section of the script parses command-line arguments, creates a `PotentialPlanner` object with the specified start and goal configurations, computes the path using gradient descent, and visualizes the resulting path.

4.2 Gradient Descent Algorithm

The `gradientdescent` method of the `PotentialPlanner` class implements the gradient descent algorithm for finding the path from the start configuration to the goal configuration. The algorithm starts at the initial configuration and iteratively updates the configuration based on the gradient of the potential function until the gradient magnitude falls below a predefined threshold (set to (1e-6) in this implementation).

The `gradientpotential` method computes the gradient of the potential function with respect to the robot's configuration. This is done using finite differences with a small delta value (set to (1e-6) in this implementation).

4.3 Command to run the script

The script can be run with the following command:

```
python potential.py --start -3.14 1.0 0.8 --goal 3.14 0.5 2.4
```

4.4 Visualization

The `visualize` method of the `PotentialPlanner` class generates an animation of the robot moving from the start configuration to the goal configuration without collisions. The resulting path is represented by lines connecting the configurations.

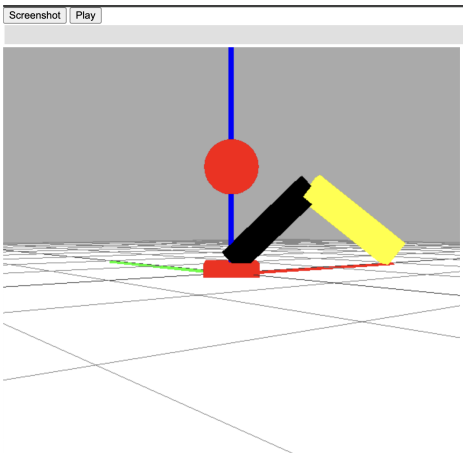

Figure 12: Potential Function Visualization 1

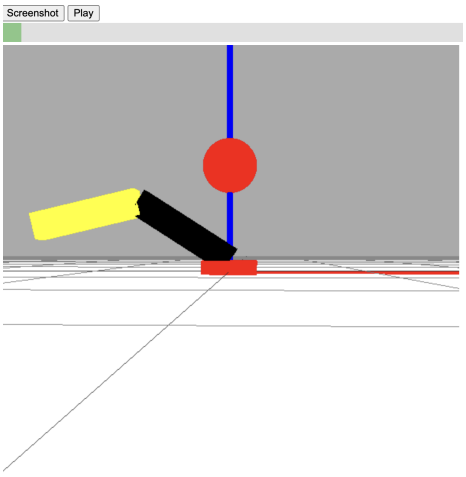

Figure 13: Potential Function Visualization 2

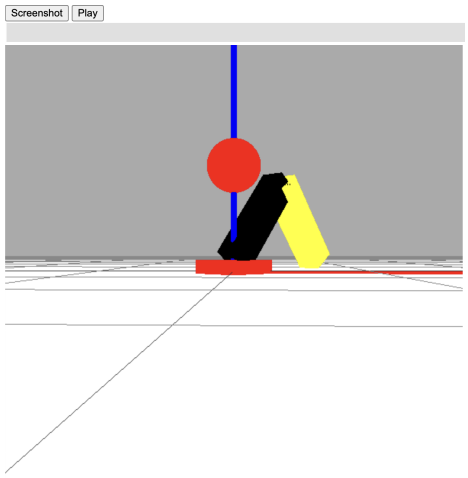

Figure 14: Potential Function Visualization 3

5 Motion Planning for a car-like robot(25 pts)

5.1 Forward propagation of the dynamical model of the car

5.1.1 Example Visualizations

The script can be run with the following three input control vectors:

1. $\mathbf{u}_0 = (1, 0)$

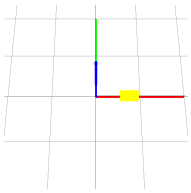

Figure 15: u (1 , 0)

2. $\mathbf{u}_1 = (1, 1)$
3. $\mathbf{u}_2 = (-1, -1)$

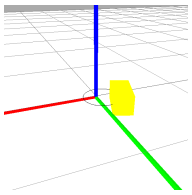

Figure 16: u (1 , 1)

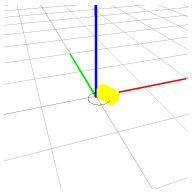

Figure 17: u (-1 , -1)

5.2 Forward propagation of the dynamical model of the car

We have implemented an kinodynamic planning RRT-based solution to find a path from the start configuration $(x, y, \theta) = (5, 25, 0.0)$ to the goal configuration $(x, y, \theta) = (17, 15, 0.0)$ for the car-like robot.

The RRT algorithm samples the control space (v, ϕ) and the duration dt to explore the state space and build a tree-like structure. The resulting path is shown in the figure below.

The figure shows the car-like robot's trajectory from the start configuration colored in green to the goal configuration colored in red00. The yellow box represents the car, and the black line depicts the path taken by the car's center of mass.

The RRT-based solution was able to find a feasible path that navigates the car-like robot from the start to the goal configuration, taking into account the car's dynamics and constraints. The .mp4 videos can be found in the /out/ass_2/rrt.mp4 folder

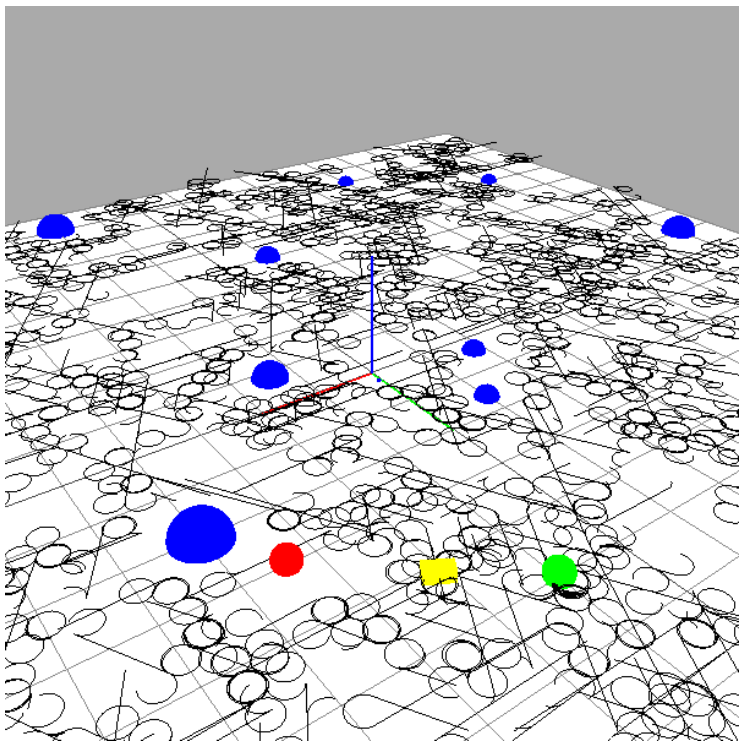

Figure 18: RRT-based solution for the car-like robot