CSE 571 - Robotics
Homework 2 - Motion Planning

Due Tuesday May 16th @ 11:59pm

The key goal of this homework is to get an understanding of motion planning methods including A*, RRT, and RRT*. For the programming assignment, you will be implementing A*, RRT, and RRT* for robot arm control, and RRT for a non-holonomic car. The code for this homework can be found here Link

Collaboration: Students can discuss questions, but each student MUST write up their own solution, and code their own solution. We will be checking code/PDFs for plagiarism.

Late Policy: This assignment may be handed in up to 5 days late (Tuesday May 16th @ 11:59pm), at a penalty of 10% of the maximum grade per day.

1 Programming Assignments

In this section you will be required to implement different motion planning algorithms and study the parameters that govern their behaviors.

1.1 Code Overview

The starter code is written in Python and depends on NumPy and Matplotlib. This section gives a brief overview.

- `run.py` - Contains the main function. Note the command-line arguments that you can provide.
- `envs/ArmEnvironment.py` - Environment-specific functions for the robot arm.
- `envs/CarEnvironment.py` - Environment-specific functions for the non-holonomic car.
- `car_map.txt` - Maps that you will work with the non-holonomic car.
- `AStarPlanner.py` - A* Planner. Logic to be filled in by you.
- `RRTPlanner.py` - RRT planner. Logic to be filled in by you.
- `RRTStarPlanner.py` - RRT* planner. Logic to be filled in by you.
- `RRTPlannerNonholonomic.py` - RRT planner for non-holonomic car system. Logic to be filled in by you.
- `RRTTree.py` - Contains datastructure that can be useful for your implementation of RRT and RRT*.
- `urdf` - This folder contains the urdf files of the robot arms. To learn more about building your own urdf refer to Link

1.2 Robot Arm

1.2.1 Environment Modeling

You are provided with two robot arms - First with 2 links [2dof_planar_robot.urdf] and is planar (Able to move in 2D only). The second is a robot with 3 links [3dof_planar_robot.urdf] (Able to move in 3D).

Note that in case of search algorithms like A*, the environment is considered to be a discrete grid while in sampling-based techniques the environment is assumed to be continuous. However, in this case since the underlying world is given to be grid, you can snap any continuous sample points onto the grid.

The cost of the final path is simply the length.

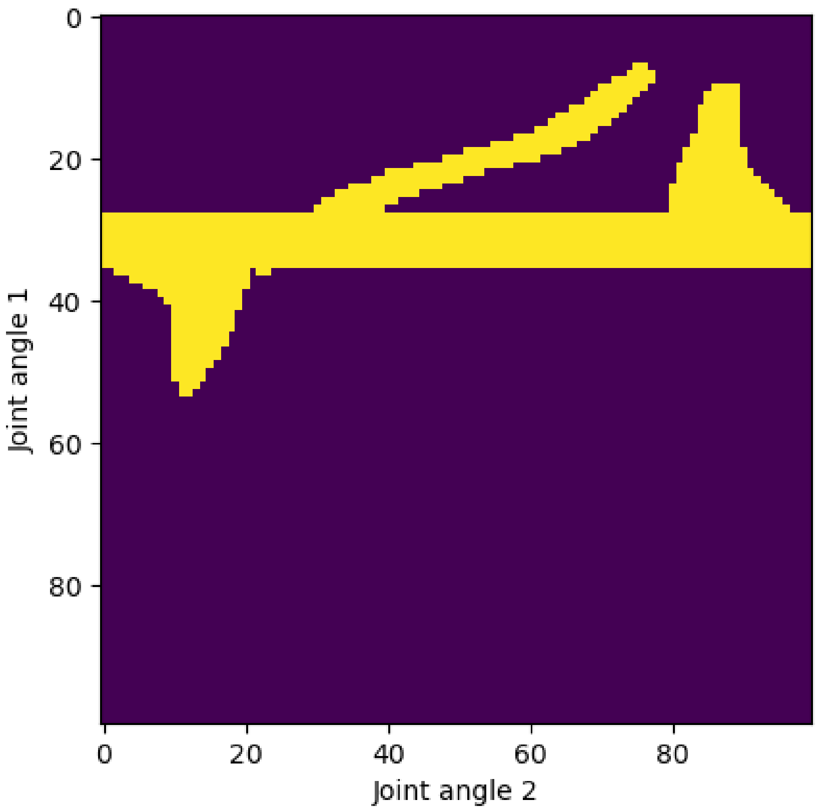
To run A* on 2-dof robot arm, you would run

```
$ python run.py -s 2dof_robot_arm -p astar
```

To visualize the configuration space, you would add the `-v` argument like

```
$ python run.py -s 2dof_robot_arm -p astar -v
```

Figure 1: Visualization of the configuration space of `2dof_planar_robot.urdf`, with each angle discritized into 100 parts



1.2.2 A* Implementation [25 points]

You will be implementing the weighted version of A-star where the heuristic is weighted by a factor of ϵ. Setting $\epsilon = 1$ gives vanilla A*. The algorithm is to be implemented in `Astar.py` file. The current version provides a get_neighbour function that provides a list of all the neighbors. Each action has a cost equal to the length of the action i.e. cost of action $(dx, dy) = \sqrt{dx^2 + dy^2}$. Use the Euclidean distance from the goal as the heuristic function.

Inorder to implement the Astar algorithms you will be using priority queues. To know more about it check out the following link.

Your results from a successful A* implementation should be comparable to the following results.

```
$ python run.py -s 2dof_robot_arm -p astar -o 0 --seed 0
...
cost: 78.0
```

```
$ python run.py -s 2dof_robot_arm -p astar -o 2 --seed 0
...
cost: 123.0
```

```
$ python run.py -s 3dof_robot_arm -p astar -o 2 --seed 0
...
cost: 168.0
```

A typical result for `python run.py -s 2dof_robot_arm -p astar -o 2 --seed 0` would be similar as Fig. 2

1. Try out different values of epsilon to see how the behavior changes. Report the final cost of the path and the number of states expanded for $\epsilon = 1,\ 10,\ 20$. Also report the time takes to run the solution. Discuss the effect of ϵ on the solution quality.
2. In the setting of `2dof_robot_arm`, visualize the final path in each setting and the states visited.
3. What do you observe when you increase or decrease the discretization of the c_space
4. Discuss any challenges you faced and describe your debugging process.

Hint:

- You will be using the following functions to complete your assignment - `heapq.heappop, heapq.heappush and get_neighbors`
- Be aware of the shape of the array when using environment functions

1.2.3 RRT Implementation [25 points]

You will be implementing a Rapidly-Exploring Random Tree (RRT) for the same 2D world. The algorithm is to be implemented in `RRTPlanner.py` file. Note that since this method is non-deterministic, you will need to provide statistical results (mean and standard deviation over 5 runs, at least). Your results from a successful RRT implementation should be comparable to the following results.

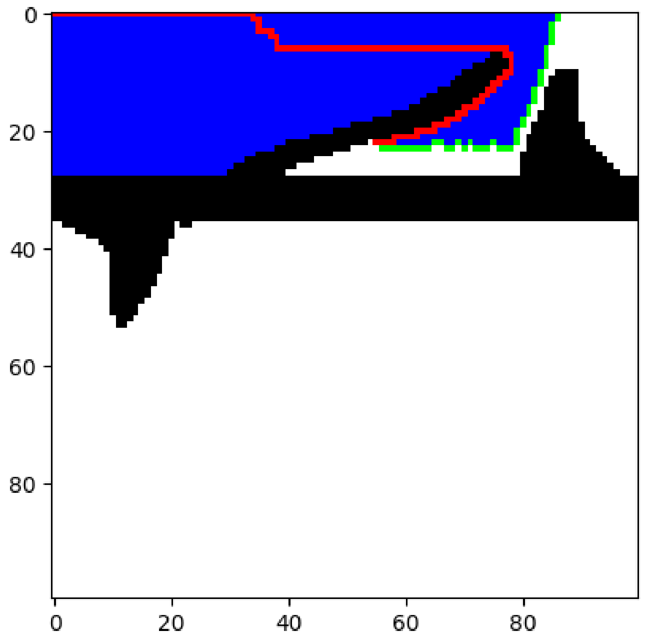


Figure 2: Example A* solution

```
$ python run.py -s 2dof_robot_arm -p rrt -o 0 --seed 0
...
cost: 198.84383834015168


$ python run.py -s 2dof_robot_arm -p rrt -o 2 --seed 0
...
cost: 170.48992200228264


$ python run.py -s 3dof_robot_arm -p rrt -o 2 --seed 0
...
cost: 166.89315959188136
```

A typical result for `python run.py -s 2dof_robot_arm -p rrt -o 2 --seed 0` would be similar as Fig. 3

Hint:

- You may be using the following functions to complete your assignment - `self.env.compute_distance`, `self.env.goal_criterion` and `self.env.edge_validity_checker`
- Be aware of the shape of the array when using environment functions

1. Bias the sampling to pick the goal with 5%, 20% probability. Report the performance (cost, time). For `2dof_robot_arm`, include at least one figures for each setting showing the final state of the tree for both values.

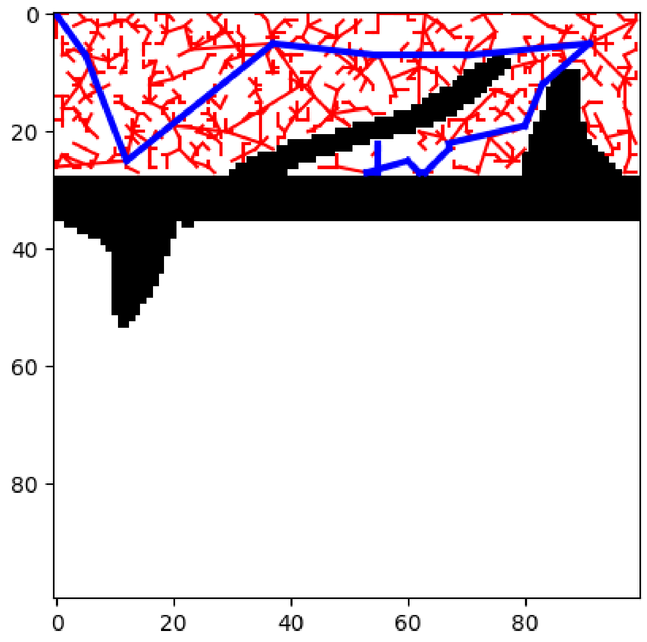


Figure 3: Example RRT solution

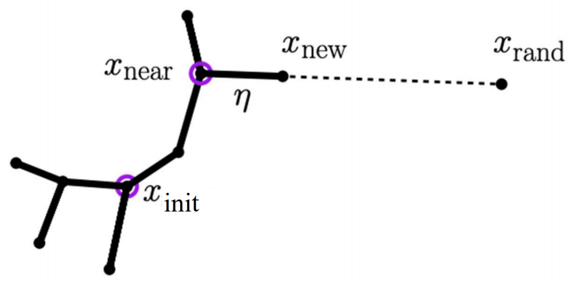
2. For this assignment, you can assume the point robot to be able to move in arbitrarily any direction i.e. you can extend states via a straight line (see Fig. 4 for an illustration). You will implement two versions of the `extend()` function:

   - the nearest neighbor tries to extend to the sampled point only by a step-size η. Set $\eta = 0.5$ and report results in your write-up.
   - the nearest neighbor tries to extend all the way till the sampled point (i.e. $\eta = 1$).

   As before, report the performance (cost, time) and include at least one figure showing the final state of the tree for each `2dof_robot_arm` setting. Which strategy would you employ in practice?

3. Discuss any challenges you faced and describe your debugging process.

Figure 4: Visualization of `extend()` with step size η, which controls the ratio of the distance from x_{near} (the state to be extended) to x_{near} and the distance from x_{rand} (a sampled state) to x_{near}. If it is set to one, $x_{\text{new}} = x_{\text{rand}}$.



1.2.4 RRT* Implementation [25 points]

You will be implementing RRT* for the same 2D world. You can implement this on top of your RRT planner with consideration for rewiring the tree whenever necessary.

```
$ python run.py -s 2dof_robot_arm -p rrtstar -o 0 --seed 0
...
cost: 187.23981592334917
```

```
$ python run.py -s 2dof_robot_arm -p rrtstar -o 2 --seed 0
...
cost: 109.91791144353034
```

```
$ python run.py -s 3dof_robot_arm -p rrtstar -o 2 --seed 0
...
cost: 166.89315959188136
```

A typical result for `python run.py -s 2dof_robot_arm -p rrtstar -o 2 --seed 0` would be similar as Fig. 3

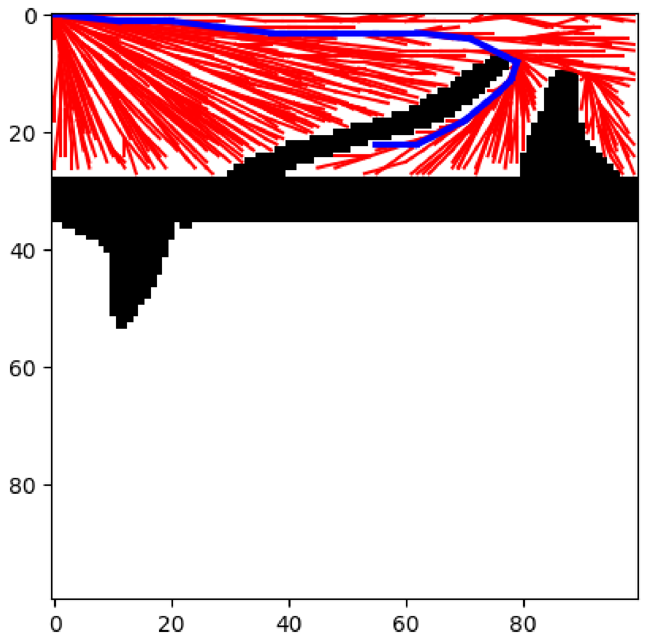


Figure 5: Example RRT* solution

Hint:

- You may be using the following funtions to complete your assignment - `self.env.compute_distance`, `self.env.goal_criterion` and `self.env.edge_validity_checker`

- Be aware of the shape of the array when using environment functions

Compare the performance of RRT* with RRT on points 1 and 2 in Section 1.2.3 (RRT implementation). In summary, make sure to:

1. Bias the goal sampling to 5% and 20%, and compare it with RRT.
2. Set $\eta = 0.5$ and $\eta = 1$ and compare it with RRT.

Again, report the performance (cost, time) in terms of statistical results (mean and standard deviation over 5 runs, at least), and provide at least one figure showing the final state of the tree for each `2dof_robot_arm` setting. After that, discuss any challenges you faced and describe your debugging process.

1.3 Non-holonomic Car

For this problem, we have a car-like system with non-holonomic constraints (i.e the system cannot instantaneously move to any state from any other state). The state of the car is $[x, y, \theta]$ where (x, y) are the coordinates of the center of the car and θ is the orientation/heading. The controls that the system can apply are (v, ω) where v is the linear velocity of the car and ω is the steering angle. Note that the car can move both forward and backward as well as turn right and left.

Unlike the previous problem where we extended the graph by moving along the straight line to the random state, we will be sampling controls to generate possible motion "rollouts" from the graph.

Given a randomly sampled state, we will find the closest graph node. Next, in the `extend()` function, we randomly sample control sequences and simulate forward from this graph node and find the one that gets us close to the randomly sampled state. The end-point of this "best" rollout is the next state we add to the graph.

1.3.1 Environment Modeling

You have been provided with a 2D map `car_map.txt` (400×400) that you will test your implementation in. This map includes a few rectangular obstacles for the car to avoid. Report all results with the start=[40, 100, $\frac{3\pi}{2}$] and goal=[350, 150, $\frac{\pi}{2}$]. Note that environment-specific functions have been provided in `CarEnvironment.py` such as dynamics simulation, action sampling, and distance computation. Unlike the 2D point robot problem, all implementation for the environment has been provided, and you may investigate the code for your own understanding.

The cost of the final path is the (simulated) time taken to execute the trajectory. See the `simulate_car()` method for more details.

To run the non-holonomic version of RRT, you would run

```
$ python run.py -s car -p nonholrrt --seed 0
...
cost: 475.0
```

1.3.2 RRT Implementation [25 points]

You will be implementing RRT for this question. Note that since this method is non-deterministic, you must provide statistical results (mean and standard deviation over 10 runs, at least). Make sure to read the provided comments in the `extend()` method in detail before implementing the code.

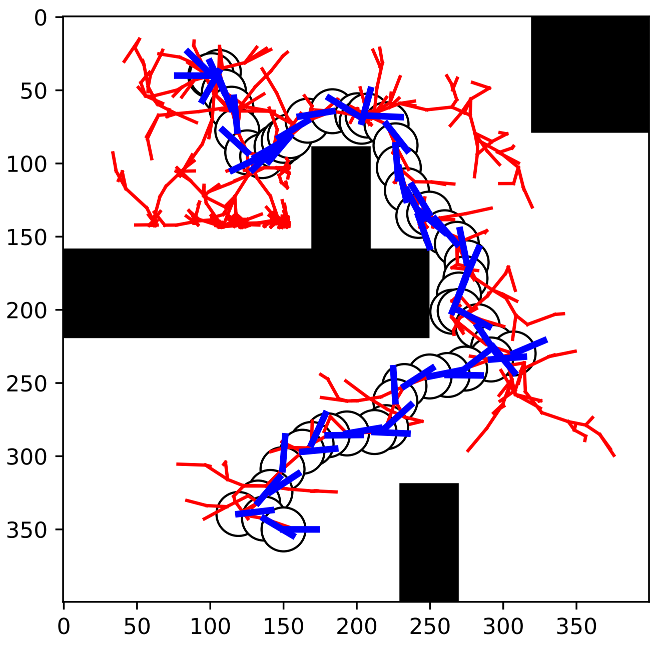


Figure 6: Example RRT solution of the non-holonomic car system.

- Bias the sampling to pick the goal with 5%, 20% probability. Report the performance (cost, time) and include at least one figure showing the final state of the tree for each value. Figure 6 shows an example solution tree found by RRT.
- Try implementing the distance function

$$d(s_1, s_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + \alpha(\theta_1, \theta_2)^2}$$

  where $\alpha(\theta_1, \theta_2) = |\theta_1 - \theta_2|\frac{180}{\pi}$ is the angular difference in degrees. Run the planner a few times with this distance function and different seeds. What are the downfalls of this distance function? How does it compare (cost, time) to the provided distance function? Investigate the results and compare them to the provided distance function.
- Discuss any challenges you faced and describe your debugging process.

1.4 Notes

- Please do not import any extra libraries. This will break the autograder.
- For the non-deterministic algorithms (RRT/RRT*), do NOT write any code for sampling (e.g. calls to `np.random`). We have provided all sampling methods (location sampling, action sampling, etc) required in the environment code and planning code for this. Please use the provided sampling methods. Otherwise, the autograder will detect incorrect results for your submission.
- Please read all function/method comments before starting to code. These will provide you with the required input/output data types.

2 Extra Credit

2.1 Dynamic path planning [25 Points]

Dynamic path planning is crucial in situations where the environment is constantly changing or the objectives of a robot may evolve over time. For this extra credit opportunity, you will implement a dynamic path planning algorithm to address a scenario in which the environment randomly changes after the robot moves a few steps.

1. Implement either LPA* or D* lite to solve this. You will need to provide adequate comments and explain your code.
2. Compare the time required for each re-planning between the A* algorithm and the one you implemented. Discuss the differences you observe.
3. Reflect on the challenges you encountered during the implementation process and describe your debugging strategies.

Note:

1. Support will not be provided for this extra credit assignment.
2. You might need to modify the source code to successfully implement the chosen algorithm. Be sure to include comments in your submission indicating any changes made.
3. You can increase or decrease the `env.change_env_step_threshold` variable in ArmEnvironment.py to change how frequently you want to replan.

3 Submission

We will be using the Canvas for submission of the assignments. Please submit the written assignment answers as a PDF. For the code, submit a zip file of the entire working directory.