Assignment 3: Planning

The goal of this assignment is to make you acquainted with graph-based motion planners and the effects of various strategies of graph construction, heuristics, and edge evaluation within the context of *path tracking* with mobile robots. In this assignment you will implement one motion planning algorithm, lazy A*, with two different kinematic constraints. You will also implement postprocessor for the found path. By the end of this assignment you will:

- become familiar with graph-based motion planners and lazy evaluation of the edges
- become familiar with the trade-offs of various heuristics in A* search.
- be able to improve a path by postprocessing.
- have an integrated system which integrates particle filter, motion planner, and controller.

Please reference the "Deliverables" sections early on in the assignment, as some parts of this lab will require you to maintain scrupulous notes and data collection practices. In the provided code, most functions have default parameters to help your impelmentation, but feel free to modify the parameters or write additional functions as necessary.

There is one extra credit question. Four member teams should implement it. We highly encourage everyone to attempt at this since it will be helpful for your final project.

1 Motion Planning

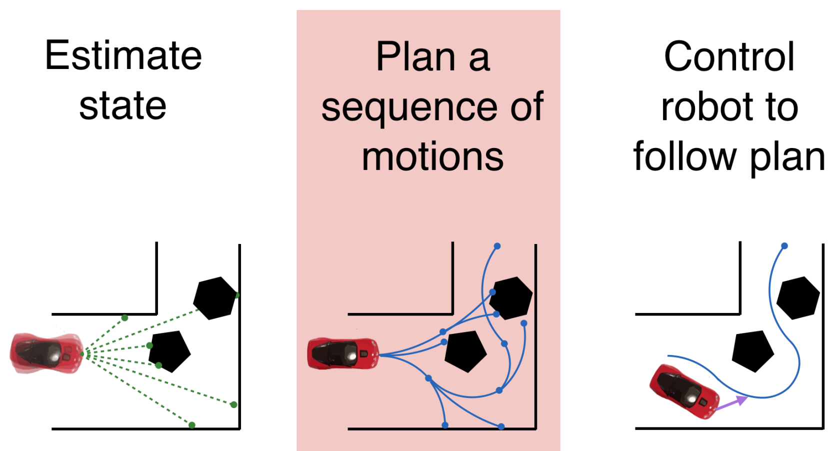

Figure 1: Motion planning within the context of mobile robotics.

1.1 Overview

Motion planning algorithms have a rich history in mobile robotics. There are many variants of motion planners. In this assignment you will focus on one variant, graph-based planners.

As mentioned in the previous assignments, in many land-based navigation scenarios, the robot's navigation task will be divided into two separate tasks: motion planning and control. The purpose of the motion planner is to generate a reference trajectory which the controller can follow.

2 Constructing a graph and providing heuristics for A* algorithm (1.5 hours)

A* is a widely used search algorithm in motion planning. It can be seen as an extension of Dijkstra's algorithm that allows the use of heuristics.

To implement a graph-based planner, you need to first construct a graph. This can be done by first sampling a fixed number of vertices in the configuration space, and connecting vertices with edges. During this process, you should make sure that the vertices and edges are collision free. Also, instead of connecting all vertices, only connect vertices within certain radius r.

For the purpose of this section, assume the configuration space is $(x, y) \in \mathbb{R}^2$.

First, implement methods to construct a graph by implementing the following methods:

- `Sampler.sample` in `Sampler.py`
- `MapEnvironment.state_validity_checker`, `compute_distances` in `MapEnvironment.py`.
- `make_graph` in `graph_maker.py`.

The graph should be collision checked before adding a vertex or an edge.

In order to run the A* algorithm, you need to implement a heuristic function. At node n, A* selects the path that minimize $f(n) = g(n) + \alpha \cdot h(n)$ where $g(n)$ is the cost of the path from the start node to n, $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal, and α is the weight of the heuristic function. For example, one commonly used heuristic function is an L_2-norm from n to the goal.

- Implement `MapEnvironment.compute_heuristic` using L_2-norm.

Once you implement the heuristic function, you can run a A* planner from start to goal by using `run.py` which calls `astar.py` we provide. Try the following command:

```
python run.py -m ../maps/map1.txt -s 1 1 -g 8 7 --num-vertices 15
```

The script provides some visualizations of the generated graphs and plans. Use these for debugging and deliverables.

With `map1.txt`, you should be able to generate a plan as `examples/map1_example.png`. Note that, since this is a sampling based method, you may not always get a graph with a feasible path between the start and the goal.

2.1 Deliverables

Use `map2.txt` to run your experiments. Run the experiments with the following setup:

```
python run.py -m ../maps/map2.txt -s 321 148 -g 106 202 --num-vertices 300
--connection-radius 100
```

You will now vary parameters of your search and report two metrics - the path length and the total planning time. Note that you will have to write code to compute and save these metrics. Since the graph is random, you will need to run the planner 10 times and report the success rate and the average of the successful ones. Use `map2.txt` and include generated figures. For each parameter that you are changing, explain why the metrics (path length and planning time) vary as they do.

1. Change the weight on the heuristic function by choosing three different α (suggested values $1.0, 50.0, 20.0$. Report metrics.
2. Keeping the number of vertices constant, decrease the connection radius till the planner fails to find a solution. Report metrics for this number. Now, double the connection radius from default value. Report metrics for this number.
3. Keeping the connection radius constant, reduce the number of vertices till the planner fails to find a solution. Report metrics for this number. Report metrics for this number. Now, double the number of vertices from default value. Report metrics for this number.

Modify the scripts as necessary to run the experiments efficiently.

3 Lazy A* (1 hour)

A graph-based motion planner has to evaluate edges between vertices. One of the main burden in motion planning algorithm is checking whether an edge is collision free, which requires checking collision between the robot and the world across all waypoints along an edge. Instead of checking all edges during graph construction, we can choose to be **lazy**: add all edges and check only when A* algorithm chooses this edge during the search.

- Implement the lazy version of `make_graph` in `graph_maker.py`.
- The provided `lazy_astar.py` is merely a copy of `astar.py`. Convert it to evaluate an edge lazily as the edge between the current node and its neighbor is getting evaluated. The evaluation function must be passed to the algorithm as `weight` function.

3.1 Deliverables

Once you implement these, you can run a lazy A* planner by using `run.py` with `--lazy` option. Run it with `map1.txt` and discuss the qualitative difference between the original A* and the lazy A* graph. Also discuss how the final paths are different, if at all. Include one figure for each A* and lazy A* that show the graph and the path. You can use `MapEnvironment.visualize_plan` for generating figures.

Run the two algorithms with `map2.txt`. Use your choice of `num_vertices`, `connection_radius`, and α you found in problem 2. You can save a graph by `make_graph(...saveto="graph.pkl")` and load a saved graph with `load_graph`.

In the writeup, include the following:

- Figures of generated plans with the two algorithms.
- Report the graph construction time, number of edges evaluated, and planning time, and solution length of the two algorithms.
- Report the setup you used for planning parameters (`num_vertices`, `connection_radius`, and α).

4 Dubins path (1 hour)

Since our car has a heading, its configuration space is in fact (x, y, θ) where θ corresponds to the heading. The path connecting start and goal should then be Dubins path where the curvature is a parameter of our choice.

We provide a Dubins path generator `dubins_path_planning` in `Dubins.py`. Your goal is to implement `DubinsMapEnvironment` and `DubinsSampler`. Implement all missing functions in `DubinsMapEnvironment`. Note that you can use the same `graph_maker` as before by passing in `DubinsMapEnvironment` and `DubinsSampler`. For `compute_heuristic`, implement it with the Dubins path length between the two configurations, which is returned by the methods in `Dubins.py`.

4.1 Deliverables

Once you have implemented these, run `runDubins.py` with `map1.txt`. Try three different parameters of `curvature` – suggested values are $1.5, 3.0, 5.0$ times the default. E.g.,

```
python runDubins.py -m ../maps/map1.txt -s 1 1 0 -g 7 1 0 --num-vertices 60
--connection-radius 15 -c 3
```

Since Dubins path is directed, you will need more samples (number of vertices) to get a connected graph. Tune the parameters (number of vertices, connection radius) as necessary.

In the writeup, include the following:

- Figures of generated plans for each curvature on `map1.txt`.
- Discuss how the generated paths are different qualitatively as well as in terms of path length.
- Report the tuned parameters. You may find it necessary to change the parameter setup per curvature.

5 Postprocess a planned path (1 hour)

The returned path from A* is the shortest path in the graph, but is not necessarily shortest path in the map, since not all vertices are directly connected by edges. Thus shortening the returned path is one common technique used with graph-based planners. Shortening a path can be done by randomly selecting a pair of vertices along the path, checking whether the edge connecting the two vertices is collision free, and if so, shortcutting the path by directly connecting them.

For `map1.txt` and `map2.txt`, you can run the shortcut by adding `--shortcut` to the run script, e.g.

```
python run.py -m ../maps/map1.txt -s 1 1 -g 8 7 --num-vertices 15 --shortcut
```

See an example pair of original and shortcut paths in `examples/map1_original_plan.png` and `examples/map1_shortcut_plan.png`.

5.1 Deliverable

Implement a shortcutting algorithm in `MapEnvironment.shortcut`.

In the writeup, include the following:

- Compare the path length, total planning time, and post processing time for `map2.txt` with and without shortcut.
- You can choose one parameter setup from the previous section regarding the planner construction and Dubins path curvature. Report the planner parameter setup you chose.
- Figures for shortcut and no shortcut paths. Discuss how they are qualitatively different.

6 Integration (1.5 hour)

You are now ready to integrate the Dubins path planner with the rest of the system you have built throughout the course. In `ROSPlanner.py`, fill in `plan_to_goal` to plan a path to goal whenever a new goal is received from RViz. Use `lazy_astar` and shortcut.

6.1 Deliverables

In simulation, start the car, map server, and a controller from lab2. Initialize the car in the cse022 map (`cse022.yaml`) and set a goal via RViz. See `README.md` for a more detailed instruction.

You may need to tune the number of vertices and the connection radius to get a good coverage of the space. Note that `ROSPlanner.py` saves the constructed graph in `ros_graph.pkl`. If there is an existing file with the same name, it loads the file without reconstructing it. That means, whenever you need to reconstruct the graph, you need to delete the existing file.

Save the RViz simulation in rosbag.

In the writeup, include the following:

- Screenshot of robot following the shortcut path in Rviz.
- Generated plan in `plan.png` and `plan_shortcut.png`.
- Graph construction time, number of edges evaluated, planning time.
- Parameters used to construct the graph.

7 Extra Credit 1: Multiple goals (1 hour)

Now you will extend the ROSPlanner to take multiple goals and route through them. The API has been implemented for you. You need to run ROSPlanner with `num_goals` > 1. In RViz, use `2D Nav goal` to set multiple goals in sequence. Implement `ROSPlanner.plan_multi_goals`.

8 Deliverables

Put the associated work in the directories and `tar` your repository.

```
$ tar czf lab3.tar.gz lab3
```

1. `lab3/bags/`: Bags mentioned in above deliverables. Include a README file for all filenames and their corresponding section number.
2. `lab3/videos/`: (Optional) Any videos of driving the car in cse 022.
3. `lab3/src/`: All codes must be submitted.
4. `lab3/launch/`: Launch files if any.
5. `lab3/writeup.pdf`: PDF Document documenting your writeup. It should answer all the deliverables mentioned above.

9 Demo

On demo day, you will be asked to demonstrate to the TAs:

1. Run the planner with a controller of your choice and particle filter in **real**. Your car should be able to find a path from the localization corner to outside of the corner.