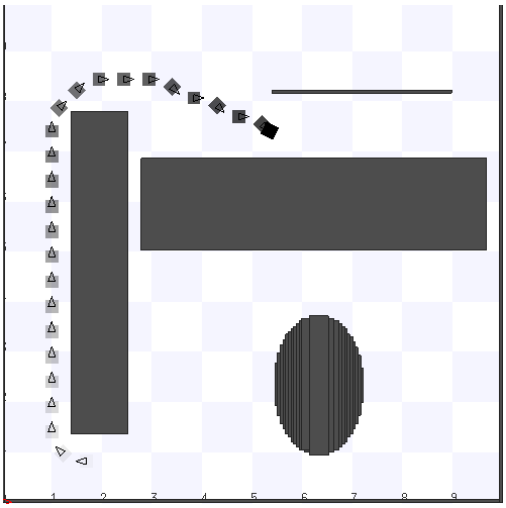
Assignment 2. Planning

Complete by **TBA**

In this assignment, we will develop and implement simple planning algorithms for mobile robot navigation and test them in simulation.



Mobility is essential to intelligent robots. As a burgeoning roboticist, you plan to develop a food delivery robot to operate in the new SoC building under construction now. The robot picks up a delivery order at the cafe, navigates the building, and sends the order to the final destination. Over time, the robot may learn to perform other useful navigation tasks, *e.g.*, guiding a first-time visitor to the building. The system will be implemented on TurtleBot, a low-cost personal robot. Hence its name *FoodTurtle*.

For this assignment, assume that FoodTurtle has a map of the building and knows its exact location all the time. The objective is to reach the specified goal by following a collision-free path. You will develop and implement a motion planning system with

- *Input*: a map, start robot pose, and goal robot pose;
- *Output*: a sequence of robot actions that (i) avoid collision with obstructions in the map and (ii) minimize travel distance to the goal.

You will test your implementation in the ROS Stage simulator. The simulator simulates TurtleBot motion control as well as well motion execution uncertainty. So you cannot always assume that the computed actions are executed perfectly.

You will develop the motion planning system, using three increasingly more sophisticated and accurate models:

- discrete state space and deterministic actions;
- continuous state space and deterministic actions;

- discrete state space and probabilistic actions.

More accurate models potentially improve the system performance, but they also require more sophisticated algorithms and greater computational resources. It is natural to ask whether there are models with continuous state space and probabilistic actions. The answer is yes, but they require complex mathematical and algorithmic tools beyond the scope of this class.

To measure system performance, we use the *Success weighted by Path Length* (SPL) metric. Mathematically, SPL is defined as

$$SPL = S\frac{l}{\max(p, l)} \tag{1}$$

where S is a binary indicator of the success, l is the length of the shortest path and p is the length of the path your agent traverses. Intuitively, SPL measures both the success rate and the length of the planned trajectory.

1 Discrete State Space and Deterministic Actions

To begin with, we use a much simplified robot motion model.

States

The state space is discretized as a grid of size $M \times N$. A discrete state is represented as a tuple (x, y, θ), where $x, y \in \mathbb{Z}$ specify the horizontal and vertical positions of the robot in the grid and $\theta \in \{0, 90^\circ, 180^\circ, 270^\circ\}$ specifies the orientation.

Actions

There are four discrete actions: FORWARD, LEFT, RIGHT, STAY.

- FORWARD: move the robot forward by 1 grid cell in its current orientation;
- LEFT: turn left by 90°;
- RIGHT: turn right by 90°;
- STAY: stay in place with no movement.

TurtleBot motion controller executes these discrete actions. However, it does not avoid collision automatically.

Motion Planning

Any forward search algorithm can solve the motion planning task here. For a complex environment, A* search likely provides the best performance. For A* search, it is important to choose the search heuristic carefully to find an optimal path efficiently.

2 Continuous State Space and Deterministic Actions

In real-world scenarios, state space and action space are continuous. The discrete assumption in the previous section no longer holds.

2.1 State Space

In natural settings, the state space is a continuous but constrained space. A robot pose (x, y, θ) is given by

$$x, y, \theta \in \mathbb{R} \quad s.t. \quad x \in [x_{min}, x_{max}], y \in [y_{min}, y_{max}], \theta \in [0, 2\pi) \tag{2}$$

2.2 Action Space and Motion Model

In continuous setting, the action space of the Food Turtle is a tuple (v, ω), where $v \in [0, 1]$ is the forward velocity and $\omega \in [-\pi, \pi)$ is the angular velocity.

Given the robot pose (x, y, θ), action (v, ω) and time interval Δt, the next robot pose (x', y', θ') is determined by a high-level motion model

$$\begin{aligned} x' &= x - \frac{v}{\omega}\sin\theta + \frac{v}{w}\sin(\theta + \omega\Delta t) \\ y' &= y + \frac{v}{\omega}\cos\theta - \frac{v}{w}\cos(\theta + \omega\Delta t) \\ \theta' &= \theta + \omega\Delta t \end{aligned} \tag{3}$$

The above equations assume at the high-level, Food Turtle can be abstracted as a point robot. In ROS, the default simulation frequency is 10Hz, which gives $\Delta t = 0.1$. A more detailed instruction can be found in our GitHub repository.

2.3 Expected Algorithm and Challenges

For motion planning in continuous spaces, you are recommended to use the Hybrid A* algorithm introduced in the lecture. There are two main challenges

1. Determine a proper cell size for discretization.
2. Design an efficient heuristic for the A* algorithm

3 Uncertainty Modeling

Modeling the motion uncertainty is critical planning algorithms. For small amount of noise, we could directly "inflate" the obstacle to leave some safety margin; for large amount of noise, we would have to formulate the problem as an Markov Decision Process (MDP).

In the previous two task, ROS stage simulator has a small amount of system noise for the TurtleBot dynamics and it is not necessary to use an MDP. You should expand the obstacle margins by a small value ϵ to compensate for the noise.

4 Task 3: Planning with Uncertainty

Unfortunately, the motion of the real robot is always coupled with uncertainty and we have to model it as a Markov Decision Process (MDP). For simplicity, we consider the discrete action and state space again (see Sect. 1).

4.1 Transition Function

As defined in Sect. 1, the action space includes {FORWARD, LEFT, RIGHT}. Suppose the TurtleBot has slippery wheels, which results in an inaccurate forward model.

Specifically, when the robot is at $s = (x, y, \theta)$ and want to take an action $a =$ FORWARD, it has to execute the action $(v, \omega) = (1, 0)$ for one second. 90% of the times, the robot can perform an accurate forward action i.e., $P(s_{forward} \mid s, a) = 0.9$. However, 5% of the times, the robot will falsely have an angular speed and a forward velocity of $\pi/2$, i.e., $p(s_{forward\_left} \mid s, a) = 0.05$; 5% of the times, robot will have an angular speed of $-\pi/2$, i.e., $p(s_{forward\_right} \mid s, a) = 0.05$. The states $s_{forward}$, $s_{forward\_left}$ and $s_{forward\_right}$ can be computed using the dynamic model Eq. 3 by setting the corresponding actions and let $\Delta t = 1$.

For simplicity, we assume the turning actions and stay action are deterministic, i.e., $p(s' \mid s, a) = 1$, where $s' = (x, y, \theta + \pi/2)$ and $a =$ LEFT. The same transition function applies to the RIGHT and STAY action.

5 Setup the Environment

To set up your environment, you should install additional packages by

```
$ sudo apt install ros-kinetic-turtlebot-stage
$ sudo apt install ros-kinetic-joint-state-publisher-gui
```

After installation, clone the assignment GitHub repository and set up the PATH

```
$ git clone git@github.com:AdaCompNUS/CS4278-CS5478_assignments.git
$ cd CS4278-CS5478_assignments
$ catkin_make
$ source devel/setup.bash
```

Now, you should be able to launch your ROS stage simulator

```
$ roslaunch planner turtlebot_in_stage.launch
```

6 Coding Template

You are provided with a template of the script for both the discrete planner and continuous planner that already handles collision checking. Your task is to fill in the missing code snippets which are marked with "FILL ME":

- Subscribing and storing of origin and goal poses.
- Computing the shortest collision-free path from the origin to goal position.

- Publishing the control commands found to the controller.

More details on the implementation are available in the GitHub repository.

7 Grading Criteria

You will be given 5 different maps, with 2 (origin, goal) pairs for each map. You should save your waypoints into txt files (please refer to example.txt), named as x_y.txt, where x is the id of the map and y is the id of the (origin, goal) pair.

We will evaluate your planned path by *Success weighted by Path Length (SPL)* metric. Mathematically, SPL is defined as

$$SPL = S\frac{l}{\max(p, l)} \tag{4}$$

where S is a binary indicator of the success, l is the length of the shortest path and p is the length of the path your agent traverses. Intuitively, SPL measures both the success rate and the length of the planned trajectory.

Your grade will be decided by averaging all the SPL scores across all the test cases.