

Motion Planning: Discrete planner

You have been assigned to extend a library of discrete motion planners for an holonomic robot. The robot moves in a static and flat environment and must find a path to a goal. All the planners share the same common interface and should implement the **search** method that has the following signature:

```
search(world_state, robot_pose, goal_pose) return path
```

Inputs:

world_state is a 2D-grid representation of the environment where the value 0 indicates a navigable space and the value 1 indicates an occupied/obstacle space.

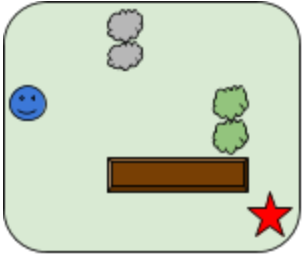
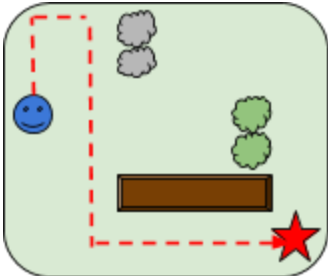
robot_pose is a tuple of two indices (x, y) which represent the current pose of the robot in world_state.

goal_pose is a tuple of two indices (x, y) which represent the goal in world_state coordinate system.

Output:

path is a list of tuple (x, y) representing a path from the robot_pose to the goal_pose in world_state or None if no path has been found.

Here is a simple example that shows the correspondence between an environment and a representation.

<p style="text-align: center;">Environment</p>  <p>Legend:</p> <ul style="list-style-type: none">Rock (obstacle)Tree (obstacle)Rock (obstacle)RobotGoal	<p style="text-align: center;">Corresponding world_state, robot_pose and goal_pose</p> <pre>world_state = [[0, 0, 1, 0, 0, 0], [0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 1, 0], [0, 0, 1, 1, 1, 0], [0, 0, 0, 0, 0, 0]]</pre> <pre>robot_pose = (2, 0) goal_pose = (5, 5)</pre>
	<p>Example of a valid path shown in red on the left figure</p> <pre>path = [(2, 0), (1, 0), (0, 0), (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5)]</pre>

We want you to write two discrete planners as described below.

- **Random planner**

The random planner tries to find a path to the goal by randomly moving in the environment (only orthogonal moves are legal). If the planner can not find an acceptable solution in less than `max_step_number`, the search should fail. The random planner, while being erratic, has a short memory, and it will never attempt to visit a cell that was visited in the last `sqrt(max_step_number)` steps except if this is the only available option.

- **Optimal planner:** A planner that goes to the goal with the shortest (non-colliding) path. Again, only orthogonal moves are legal.

Compare the performance of the two planners.

You are expected to **write documentation** and **test correctness** for your code. You should also indicate the complexity of each solution in the documentation. You can either use Python 2.7 and/or C++. For Python, Numpy library may be used. For C++, boost and stl libraries may be used.