Motion Planner: Discrete Planners

This project builds and compares the performance of two kinds of discrete grid-based motion planners - a random planner, and an optimal planner.

Project Description:

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. In this project, an implementation of A* is used for the optimal planner. More details follow.

Installation and Usage

The code base used here only assumes that Python 3.5 with the pre-installed modules (through a conda distribution) is installed. Only one external module -- `numpy_indexed` -- is used in `analyze_planners.py` to generate relevant statistics for analysis; note that planners themselves can be run without this module.

Modules used:

- Numpy
- Sys
- Matplotlib
- Collections
- Heapq
- Random
- Numpy_indexed (`pip install numpy_indexed`, if not already installed)

The code suite contains two files:

1. `two_discrete_planners.py`
2. `analyze_planners.py`

The first one is the main script containing both planners. This document mainly deals with the contents of `two_discrete_planners.py`. Refer the accompanying document to learn more about the analysis part of this project.

Run the following command in terminal after navigating to the relevant folder to execute an example case:

```
python two_discrete_planners.py planner_type
```

where planner_type can be either “random” or “optimal”. It will return a plot of a preselected world, query points, and the generated path using the selected planner. To pass in your own query points, run:

```
python two_discrete_planners.py planner_type a b c d
```

where the query points are `(a, b)` for start, and `(c, d)` for end respectively.

For the random planner, the value of `max_step_number` is preselected to be 200 (empirically chosen as a reasonable value which ensures a successful search for all valid query points). Note that `max_step_number` is used only for the random case.

For the optimal planner, A* was chosen due to its speed and optimality in grid-search. Since only orthogonal moves are permitted, the manhattan distance was selected as an admissible heuristic. Indeed, as analysis in the succeeding sections show, this choice of heuristics come fairly close to the absolute optimal heuristic.

Classes and methods defined in this script:

- `class GridWorld(width, height, obstacle)`
  Creates and initializes a 2D uniform grid with `height` rows and `width` columns. The obstacles or walls in this world are passed in `obstacle` as a list of location tuples. Has four methods associated with this class:
  - `boundary(pos)`
    Takes in `pos` and returns a boolean value indicating if `pos` is within the bounds specified in the constructor of `GridWorld`.
  - `legal_move(pos)`
    Takes in `pos` and returns a boolean value indicating if `pos` is in collision with an obstacle or not, as defined in `GridWorld`.
  - `neighbors(pos)`
    Takes in `pos` and returns a list of neighbors (using the grid’s 4-point connectivity) of `pos`, filtered by `boundary()` and `legal_move()`.
  - `cost(from_node, to_node)`
    Calculates the cost of movement from `from_node` to `to_node`. Since

this is a uniform grid with uniform weights attached to the edges, the cost is returned as 1 irrespective of input. Used only for A*.

- `class PriorityQueue()`
  Creates and initializes a queue with zero elements. Has three methods:
  - `empty()`
    Returns a boolean value indicating if queue is empty or not
  - `put(item, priority)`
    Pushes `item` into queue, location being determined by `priority`. The lowest priority items are placed at the top.
  - `get()`
    Removes item with lowest priority from (the top of) the queue.

- `heuristic(a, b)`
  Calculates heuristic value of estimated distance left from `a` to `b`. Uses Manhattan distance metric.

- `search(g, robot_pose, goal_pose, max_step_number, planner, verbose)`
  `g`: `GridWorld` object
  `robot_pose`: start query point
  `goal_pose`: end query point
  `max_step_number`: maximum steps the algorithm is allowed to take before returning failure. `sqrt(max_step_number)` is used to determine the memory of the planner. Used only when planner is selected as random.
  `planner`: selects type of planner, `"optimal"` for A* and `"random"` for random planner.
  `verbose`: Takes in 1 or 0 to enable or disable all `print` outputs, respectively.

  Initializes a `deque` object `q` with `robot_pose` and fixed length `sqrt(max_step_number)` to serve as the memory bank of the random planner.
  Has an anonymous function `memory_filter` which filters points generated by `GridWorld.neighbors()` by comparing them to `q`.
  Returns a path connecting `robot_pose` and `goal_pose` using planner.
  Reports failure if invalid query points are selected in the case of `"optimal"` and if `max_step_number` isn't enough to reach the `goal_pose` in case of `"random"`.
  Also returns `step_number`, the number of steps taken to reach `goal_pose` when attempt is successful for `"random"`, and returns `counter`, the number of nodes visited for A* to reach `goal_pose`.

- `visualize_planner(g, robot_pose, goal_pose, path)`
  Takes in a `GridWorld` object, the start query `robot_pose`, the end query `goal_pose`, and plots the environment along with the path and query points. Uses `matplotlib.pyplot` to do all the plotting.

Refer the accompanying document to know more about `analyze_planners.py`, results, and analysis of both planners.

Acknowledgments:

The A* implementation was inspired by Amit Patel's excellent resource on discrete motion planners.