Robot Motion Planning

CMS 430, Spring 2017

Due Thursday, March 9, at 11:59 PM

Description

You may work with a partner on this project.

In this project, you'll use search techniques to solve a *motion planning problem*. This was the original application that motivated the development of A* search.

Submit a Python script named `motion.py` through your GitHub repo. To make things easier to grade, you and your partner (if you work with one) should each submit identical copies of your code to your own repos.

Motion Planning

Consider a robot planning a path through a grid-based world. Its goal is to move from a start square in the upper-left to a goal square in the lower-right, moving around any obstacles in its way.

Let's make some assumptions:

- The world is a R by C grid, indexed from 0 in both dimensions
- The start square is always at position `(1,1)`.
- The goal square is always at position `(R - 2, C - 2)`.
- Obstacles in the grid have the value 1 and free squares have the value 0.
- The boundaries of the world are always enclosed by walls of 1's, so we don't need to consider special cases for stepping off the edge of the grid.
- The robot can move up, down, left, and right, but not diagonally.

Here's an example 10x30 grid world. The shortest path is marked in stars. For clarity, the zeros have been replaced by spaces.

```
1111111111111111111111111111
1**********           1      1
1    11111*           1      1
11111   ***           11111111
1       *111111111111111     1
1       *****                 1
1     111111*                 1
1           *                 1
1           ******************1
1111111111111111111111111111
```

Write a program that uses A* search to find the shortest path from the start to the goal in a randomly generated grid world, or discover that no path exists. Use the included Python script as a starting point, which includes code to randomly generate ten trial worlds.

Your program should print out the solution grid, showing the shortest path in stars, as in the example above, or a message that no path exists if the search fails.

Use the *Manhattan distance* from the current position to the goal square as your heuristic.

Tips:

- The project repo includes an example of A* solving the 8-puzzle; study it carefully for tips on how to implement the algorithm and use the priority queue
- A* is based on a breadth-first search, so it should use the same basic structure as our other breadth-first search problems
- Import the `Queue` package for a priority queue.
- To insert items into the priority queue, package them as tuples:
  `queue.put((priority, state))`
- When you get the smallest item from the queue, it will be returned as a tuple, the same way it was inserted:
  ```
  queue_entry = queue.get()
  state = queue_entry[1]
  ```
- A* also requires you to keep track of each solution's distance from the start space. *This is not the same as the Manhattan distance to the start space*! Each state must have a variable that tracks how many moves have been taken to reach it.