module-01-programming-only

January 27, 2019

1 Module 1 - Programming Assignment

1.1 Directions

You will be putting all your code and comments in a file with the title:

`<jhed_id>.py`

You may start with the provided file and change the name.
My suggestion to you is to make a directory structure something like this:

```
ai-+
   |
   +--module01
   |    |
   |    +--<jhed_id>.py
   |
   +--module02
   |    |
   |    +--<jhed_id>.py
   ...
```

Please submit only the .py file. The `<jhed_id>` is your username like `jsmith245`.
I will be executing your program like so:

```
> python <jhed_id>.py
```

You're program should print out *only* the output I have requested and *exactly* how I have requested it. If it it's wrong, it's wrong.

2 State Space Search with A* Search

You are going to implement the A* Search algorithm for navigation problems.

2.1 Motivation

Search is often used for path-finding in video games. Although the characters in a video game often move in continuous spaces, it is trivial to layout a "waypoint" system as a kind of navigation grid over the continuous space. Then if the character needs to get from Point A to Point B, it does a line of sight (LOS) scan to find the nearest waypoint (let's call it Waypoint A) and finds the nearest, LOS waypoint to Point B (let's call it Waypoint B). The agent then does a A* search for Waypoint B from Waypoint A to find the shortest path. The entire path is thus Point A to Waypoint A to Waypoint B to Point B.

We're going to simplify the problem by working in a grid world. The symbols that form the grid have a special meaning as they specify the type of the terrain and the cost to enter a grid cell with that type of terrain:

token	terrain	cost
.	plains	1
*	forest	3
#	hills	5
~	swamp	7
x	mountains	impassible

We can think of the raw format of the map as being something like:

```
....*..
...***.
.###...
..##...
..#..**
....***
.......
```

2.2 The World

(Don't worry about code being chopped off here; all the starter code is in the accompanying `.py` file)

Given a map like the one above, we can easily represent each row as a `List` and the entire map as `List of Lists`:

```
In [1]: full_world = [
            ['.', '.', '.', '.', '.', '*', '*', '*', '*', '*', '*', '*', '*', '*', '*', '.', '.'
            ['.', '.', '.', '.', '.', '.', '.', '*', '*', '*', '*', '*', '*', '*', '*', '*', '.'
            ['.', '.', '.', '.', 'x', 'x', '*', '*', '*', '*', '*', '*', '*', '*', '*', '*', '*'
            ['.', '.', '.', '.', '#', 'x', 'x', 'x', '*', '*', '*', '*', '~', '~', '*', '*', '*'
            ['.', '.', '.', '#', '#', 'x', 'x', '*', '*', '.', '.', '~', '~', '~', '~', '*', '*'
            ['.', '#', '#', '#', 'x', 'x', '#', '#', '.', '.', '.', '.', '~', '~', '~', '~', '~'
            ['.', '#', '#', 'x', 'x', '#', '#', '.', '.', '.', '.', '#', 'x', 'x', 'x', '~', '~'
            ['.', '.', '#', '#', '#', '#', '#', '.', '.', '.', '.', '.', '.', '#', 'x', 'x', 'x'
            ['.', '.', '.', '#', '#', '#', '.', '.', '.', '.', '.', '.', '#', '#', 'x', 'x', '.'
            ['.', '.', '.', '~', '~', '~', '.', '.', '#', '#', '#', 'x', 'x', 'x', 'x', '.', '.'
            ['.', '.', '~', '~', '~', '~', '~', '.', '#', '#', 'x', 'x', 'x', '#', '.', '.', '.'
```

```
['.', '~', '~', '~', '~', '~', '.', '.', '#', 'x', 'x', '#', '.', '.', '.', '.', '~'
['~', '~', '~', '~', '~', '.', '.', '#', '#', 'x', 'x', '#', '.', '~', '~', '~', '~'
['.', '~', '~', '~', '~', '.', '.', '#', '*', '*', '#', '.', '.', '.', '.', '~', '~'
['.', '.', '.', '.', 'x', '.', '.', '*', '*', '*', '*', '#', '#', '#', '#', '.', '~'
['.', '.', '.', 'x', 'x', 'x', '*', '*', '*', '*', '*', '*', 'x', 'x', 'x', '#', '#'
['.', '.', 'x', 'x', '*', '*', '*', '*', '*', '*', '*', '*', '*', '*', 'x', 'x', 'x'
['.', '.', '.', 'x', 'x', '*', '*', '*', '*', '*', '*', '*', '*', '*', '*', '*', 'x'
['.', '.', '.', 'x', 'x', 'x', '*', '*', '*', '*', '*', '*', '*', '*', '.', '.', '.'
['.', '.', '.', '.', 'x', 'x', 'x', '*', '*', '*', '*', '*', '*', '.', '.', '.', '.'
['.', '.', '#', '#', '#', '#', 'x', 'x', '*', '*', '*', '*', '*', '.', 'x', '.', '.'
['.', '.', '.', '.', '#', '#', '#', 'x', 'x', 'x', '*', '*', 'x', 'x', '.', '.', '.'
['.', '.', '.', '.', '.', '.', '#', '#', '#', 'x', 'x', 'x', 'x', '.', '.', '.', '.'
['.', '#', '#', '.', '.', '#', '#', '#', '#', '#', '.', '.', '.', '.', '.', '#', '#'
['#', 'x', '#', '#', '#', '#', '.', '.', '.', '.', '.', 'x', 'x', 'x', '#', '#', 'x'
['#', 'x', 'x', 'x', '#', '.', '.', '.', '.', '.', '#', '#', 'x', 'x', 'x', 'x', '#'
['#', '#', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '#', '#', '#', '#', '#'
```

Warning

One implication of this representation is that (x, y) is world[y][x] so that (3, 2) is world[2][3] and world[7][9] is (9, 7).

It is often easier to begin your programming by operating on test input that has an obvious solution. If we had a small 7x7 world with the following characteristics, what do you expect the policy would be?

```
In [2]: test_world = [
            ['.', '*', '*', '*', '*', '*', '*'],
            ['.', '*', '*', '*', '*', '*', '*'],
            ['.', '*', '*', '*', '*', '*', '*'],
            ['.', '.', '.', '.', '.', '.', '.'],
            ['*', '*', '*', '*', '*', '*', '.'],
            ['*', '*', '*', '*', '*', '*', '.'],
            ['*', '*', '*', '*', '*', '*', '.'],
        ]
```

2.3 States and State Representation

The canonical pieces of a State Space Search problem are the States, Actions, Transitions and Costs.

We'll start with the state representation. For the navigation problem, a state is the current position of the agent, `(x,y)`. The entire set of possible states is implicitly represented by the world map.

2.4 Actions and Transitions

Next we need to specify the actions. In general, there are a number of different possible action sets in such a world. The agent might be constrained to move north/south/east/west or diagonal moves might be permitted as well (or really anything). When combined with the set of States, the *permissible* actions forms the Transition set.

Rather than enumerate the Transition set directly, for this problem it's easier to calculate the available actions and transitions on the fly. This can be done by specifying a *movement model* as

offsets to the current state and then checking to see which of the potential successor states are actually permitted. This can be done in the successor function mentioned in the pseudocode.

One such example of a movement model is shown below.

```
In [3]: cardinal_moves = [(0,-1), (1,0), (0,1), (-1,0)]
```

2.5 Costs

We can encode the costs described above in a `Dict`:

```
In [4]: costs = { '.': 1, '*': 3, '#': 5, '~': 7}
```

2.6 A* Search Implementation

As Python is an interpreted language, you're going to need to insert all of your helper functions *before* the actual `a_star_search` function implementation.

Please **read the Syllabus** for information about the expected code structure. I expect a "literate" style of "functional" programming.

---

a_star_search

The `a_star_search` function uses the A* Search algorithm to solve a navigational problem for an agent in a grid world. It calculates a path from the start state to the goal state and returns the actions required to get from the start to the goal.

- **world** is the starting state representation for a navigation problem.
- **start** is the starting location, `(x, y)`.
- **goal** is the desired end position, `(x, y)`.
- **costs** is a `Dict` of costs for each type of terrain.
- **moves** is the legal movement model expressed in offsets.
- **heuristic** is a heuristic function that returns an estimate of the total cost $f(x)$ from the start to the goal through the current node, x. The heuristic function might change with the movement model.

The function returns the offsets needed to get from start state to the goal as a `List`. For example, for the test world:

```
['.', '*', '*', '*', '*', '*', '*'],
['.', '*', '*', '*', '*', '*', '*'],
['.', '*', '*', '*', '*', '*', '*'],
['.', '.', '.', '.', '.', '.', '.'],
['*', '*', '*', '*', '*', '*', '.'],
['*', '*', '*', '*', '*', '*', '.'],
['*', '*', '*', '*', '*', '*', '.'],
```

it would return:

```
[(0,1), (0,1), (0,1), (1,0), (1,0), (1,0), (1,0), (1,0), (1,0), (0,1), (0,1),
(0,1)]
```

Do not make unwarranted assumptions. For example, do not assume the starting point is always `(0, 0)` or that the goal is always in the lower right hand corner. Do not make any assumptions about the movement model beyond the requirement that they be offsets (it could be offets of 2!).

```
In [5]: def a_star_search( world, start, goal, costs, moves, heuristic):
            ### YOUR SOLUTION HERE ###
            ### YOUR SOLUTION HERE ###
            return []
```

pretty_print_solution

The `pretty_print_solution` function prints an ASCII representation of the solution generated by the `a_star_search`. For example, for the test world, it would take the `world` and `path` and print:

```
v******
v******
v******
>>>>>>v
******v
******v
******G
```

using `v`, `^`, `>`, `<` to represent actions and `G` to represent the goal. (Note the format of the output...there are no spaces, commas, or extraneous characters). You are printing the path over the terrain.

```
In [6]: def pretty_print_solution( world, path, start):
            ### YOUR SOLUTION HERE ###
            ### YOUR SOLUTION HERE ###
            return None
```

Execute `a_star_search` and `print_path` for the `test_world` and the `real_world`.

Describe and define your heuristic function here. You can change the arguments to whatever you use in your `a_star_search` implementation.

```
In [7]: # heuristic function
        def heuristic():
            pass
```

```
In [8]: test_path = a_star_search( test_world, (0, 0), (6, 6), costs, cardinal_moves, heuristi
        print( test_path)
```

```
[]
```

```
In [9]: pretty_print_solution( test_world, test_path, (0, 0))
```

```
In [10]: full_path = a_star_search( full_world, (0, 0), (26, 26), costs, cardinal_moves, heuris
         print( full_path)
```

```
[]
```

```
In [11]: pretty_print_solution( full_world, full_path, (0, 0))
```

3 Advanced/Future Work

This section is not required but it is well worth your time to think about the task

Write a *general* `state_space_search` function that could solve any state space search problem using Depth First Search. One possible implementation would be to write `state_space_search` as a general higher order function that took problem specific functions for `is_goal`, `successors` and `path`. You would need a general way of dealing with states, perhaps as a `Tuple` representing the raw state and metadata: `(<state>, <metadata>)`.