CISS450: Artificial Intelligence
Assignment 7

OBJECTIVES

- Implement Greedy Best First Search
- Implement A* Search

Q1. Maze problem

Modify the maze program to do the following: Add options `gbfs` (greedy best first search) and `astar` (A^* search). For heuristic function use Euclidean distance.

Like `ucs`, for `astar`, allow users to add `dangerous rooms` as in a04. Here the format of the text input/output from a04:

```
enter random seed: 5
0-random maze or 1-stored maze: 0
initial row: 1
initial column: 2
goal row: 3
goal column: 4
bfs or dfs or ucs or iddfs or gbfs or astar: ucs
1 1 10
2 2 10
3 3 100

solution: ['S', 'N', 'E', 'N']
len(solution): 4
len(closed_list): 0
len(fringe): 0
```

(As in a04, note that the above is not the correct output.)

SPOILER SUGGESTIONS: HEURISTIC AND PRIORITY

Here are some suggestions. You don't have to follow them. Also, you have to think very carefully about the ideas below.

Recall that for each node, there are three costs involved: the depth, the path cost, and now the heuristic cost (to go to a goal state). Since heuristic search (`gbfs` and `astar`) also use a priority queue, you want to study your `Fringe` class and your `UCSFringe` class very carefully so as not to rewrite code unnecessarily.

Note that for the earlier `ucs` you are using the path cost as priority. Now, depending on whether it's `ucs`, `gbfs`, or `astar`, there are three different priorities

- path cost
- heuristic
- path cost + heuristic

Recall that at this point the `SearchNode` class contains `depth` and `path_cost`. Recall that in the notes I used f to denote the priority for heuristic search (`gbfs` and `astar`). I'm going to use f to denote any priority value (i.e., for the priorities used by `ucs`, `gbfs` and `astar`).

Add `f_cost` to the `SearchNode` class:

```
class SearchNode:
    def __init__(self, ..., f_cost=0)
        ...
        self.f_cost = f_cost
        ...
```

Look at your graph search: You have to create search nodes. For each search node, we now have to compute the `f_cost` to be used in the search node constructor.

Note that the heuristic function is based on the state. So let's put the heuristic function into the a problem class (for this question it's the `MazeProblem` class). In the case when you are experimenting with one single heuristic function, you can add a heuristic function directly into your problem class:

```
class MazeProblem(...):
    ...
    def h(self, state):
        ... return heuristic value of state ...
```

```
    ...
```

This is the case for this question and is probably the simplest solution. (This is not the cleanest/slickest way to do it. But it's the simplest.)

Now to compute the `f_cost`. Hold the `f_cost` computation in subclasses of the problem class:

```
class MazeProblem(...):
    ...
    def h(self, state):
        ... return heuristic value of state ...
    def f_cost(self, state, path_cost):
        ... (you can return path_cost as the default) ...
    ...

class GBFSProblem(MazeProblem):
    ...
    def f_cost(self, state, path_cost):
        ... return a value ...

class AStarProblem(MazeProblem):
    ...
    def f_cost(self, state, path_cost):
        ... return a value ...
```

With all the above in place, you can now go to your graph search and look at the places where you create search nodes and include the `f_cost`. After this is done, besides state, parent, parent action, each search node will now hold depth, path cost, and f-cost.

Recall that `UCSFringe` already has a priority queue. You want to reuse the code as much as possible. `UCSFringe` uses the `priority()` method in the `SearchNode` class to decide on the priority of the search node. Fortunately we did not hardcode or directly use the path cost in the search node. This means that our `UCSFringe` is actually a general priority queue with fast look up. So all you need to do is to return the `f_cost` in the `priority()` method of `SearchNode`. Just make sure the `f_cost` is the path cost in the case of `ucs` search.

Q2. $n^2 - 1$ problem.

Now add two heuristic search strategy to your $n^2 - 1$: `gbfs` (Greedy best first search) and `astar` (A* search)

For your heuristic function, use "sum of Manhattan distance for all tiles to their right place".

To fix the input/output, here's an execution. Suppose you want to solve the following:

```
0 1 2
3 4 5
6   7
```

Here's a screenshot of your console window:

```
size: 3
initial: 0,1,2,3,4,5,6, ,7
bfs or dfs or ucs or iddfs or gbfs or astar: ucs
step costs: 1 3 2 4
solution: ['N', 'N', 'S', 'S', 'E', 'E', 'W', 'W']
len(solution): 26
len(closed_list): 42
len(fringe): 9
```

where the inputs the step cost (only for `ucs` and `astar`) are for the costs of `'N'`, `'S'`, `'E'` and `'W'` respectively. Except for experiments, you probably want to enter 1 1 1 1 for step costs. (The output above is incorrect. This is just to fix the output format.)