CISS450: Artificial Intelligence
Assignment 5

Objectives

1. The goal is to solve the several AI search problems using brute force or blind or uninformed search using BFS and DFS.

Note that you must implement the late graph search algorithm for this assignment (i.e., not the early graph search algorithm).

Q1. Maze problem (again)

Modify the maze program (from a04):

1. Add `ucs` and `iddfs` (for uniform cost search and iterative deepending DFS).
2. For `ucs`, and only for `ucs`, allow users to add "dangerous rooms" where the cost is high. the program prompts for a list of row, column pairs and the cost (until a blank line is entered). For instance, in the example below, the inputs

   ```
   1 1 10
   2 2 10
   3 3 100
   ```

   mean that any action landing in $(1, 1)$ will have a cost of 10, any action that lands in $(2, 2)$ will have a cost of 10, and any action that lands in $(3, 3)$ will have a cost of 100. All other acitons will have a cost of 1. Draw the "dangerous rooms" in your visualization yellow with RGB value of (255,255,0).

Here's an example to fix input/output:

```
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: 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
```

If you test the ucs option where there is a cluster of consecutive dangerous rooms you will see that the fringe expands in such a way that it avoids this cluster.

See the sections below for help/hints/suggestions.

`SearchNode.py`

Make sure there is a `path_cost` member in a `SearchNode` object since this is needed by the ucs search:

```
    def __init__(self,
                 state,
                 parent=None,
                 parent_action=None,
                 path_cost=0):
        self.state = state
        self.parent = parent
        self.parent_action = parent_action
        self.path_cost = path_cost
```

`ClosedList.py`

A new `ClosedList.py` is included. Make sure you study it carefully. You need not use the closed list class that allows compression of states.

`Fringe.py`

You must include the class `UCSFringe` which is the fringe class to be used for ucs search. This class is in the file `Fringe.py` and is a subclass of the `Fringe` class. Here's the beginning of my `UCSFringe`:

```
class UCSFringe(Fringe):
    """
    This implements a unique fringe as a priority queue. If a search node n is
    inserted into this fringe (using the put method), node.priority() is
    executed for the priority of the node (which should return the path_cost
    of the node. If node.state does not appear in the fringe, the node
    joins the fringe. If another node n1 has the same state as n, then
    1. If n.priority() >= n1.priority(), n is rejected.
    2. If n.priority() < n1.priority(), then n1 is overwritten by n and
       is then heapify-up.
    """
    def __init__(self):
        Fringe.__init__(self)
        self.array = [] # binary minheap
        self.d = {}      # dictionary of (state, index)
```

(The above `UCSFringe` implementation requires a `priority()` method in the `SearchNode` class which returns the path cost of the search node.) This class should be added to the file `Fringe.py` (from a04). (We already talked about this: a ucs fringe object is a fringe that contains a minheap implemented using an array and a dictionary with key-value pairs of (state, index) which allows this fringe to find the index of the search node for a given state. The following is some test code that you should add to the bottom `Fringe.py`. after the test code from a04. The new test code are some test cases for `UCSFringe`. You are strongly advised to add more test cases.

```
if __name__ == '__main__':

    # ... SAME TEST CASE FROM a04 ...

    class TestSearchNode:
        def __init__(self, state, pri):
            self.state = state
            self.pri = pri
        def priority(self):
            return self.pri
        def __str__(self):
            return '<%s %s>' % (self.state, self.pri)
    n0 = TestSearchNode('a', 8)
    n1 = TestSearchNode('b', 7)
    n2 = TestSearchNode('c', 4)
    n3 = TestSearchNode('d', 2)
    n4 = TestSearchNode('e', 6)
    n5 = TestSearchNode('f', 1)

    ns = [n0, n1, n2, n3, n4, n5]
    f = UCSFringe()
    print(f)
    for n in ns:
        f.put(n)
        print "insert", n, "...", f

    print "Checking fringe ..."
    f.check()

    print "Testing inserting duplicate state with larger priority"
    f.put(TestSearchNode('e',10))
    print(f)

    print "Testing inserting duplicate state with smaller priority"
    f.put(TestSearchNode('e', 0))
    print(f)

    while len(f) != 0:
        n = f.get()
        print "get", n, "...", f
```

The output for the part that tests the `UCSFringe` should be

```
<UCSFringe []>
insert <a 8> ... <UCSFringe [<a 8>]>
insert <b 7> ... <UCSFringe [<b 7>, <a 8>]>
insert <c 4> ... <UCSFringe [<c 4>, <a 8>, <b 7>]>
insert <d 2> ... <UCSFringe [<d 2>, <c 4>, <b 7>, <a 8>]>
insert <e 6> ... <UCSFringe [<d 2>, <c 4>, <b 7>, <a 8>, <e 6>]>
insert <f 1> ... <UCSFringe [<f 1>, <c 4>, <d 2>, <a 8>, <e 6>, <b 7>]>
Checking fringe ...
Testing inserting duplicate state with larger priority
<UCSFringe [<f 1>, <c 4>, <d 2>, <a 8>, <e 6>, <b 7>]>
Testing inserting duplicate state with smaller priority
<UCSFringe [<e 0>, <f 1>, <d 2>, <a 8>, <c 4>, <b 7>]>
get <e 0> ... <UCSFringe [<f 1>, <c 4>, <d 2>, <a 8>, <b 7>]>
get <f 1> ... <UCSFringe [<d 2>, <c 4>, <b 7>, <a 8>]>
get <d 2> ... <UCSFringe [<c 4>, <a 8>, <b 7>]>
get <c 4> ... <UCSFringe [<b 7>, <a 8>]>
get <b 7> ... <UCSFringe [<a 8>]>
get <a 8> ... <UCSFringe []>
```

You are very strongly advised to fully test your UCSFringe class.

`State.py`

`State.py` is included. Study it carefully. You can also choose not to use a class for states. For instance you can just use a 2-tuple to represent states for the maze problem.

Q2. $n^2 - 1$ Problem (again)

Now add two most search strategies to your $n^2 - 1$:

1. `ucs` (uniform-cost search)
2. `iddfs` (iterative deepening DFS)

For uniform-cost search, the program prompts the user for step cost of `'N'`, `'S'`, `'E'`, `'W'` action. (Note that in general a step cost can depend on starting state and not just action. But for this question, teh step cost only depends on the action.)

As before in a04, the actions processed in your search algorithm must be in the order of `'N','S','E''W'`.

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

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

with ucs where the step cost of `'N'` is 1, step cost of `'S'` is 3, step cost of `'E'` is 2, step cost of `'W'` is 4. Here an execution:

```
size: 3
initial: 0,1,2,3,4,5,6, ,7
bfs or dfs or ucs or iddfs: 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
```

(The output above is incorrect. This is just to fix the output format.) The first input is the number of rows and number of columns of the puzzle, the second is the initial state of the puzzle, and the third input is `bfs` or `dfs` or `ucs` or `iddfs` for BFS or DFS or Uniform cost search or iterative deepening DFS respectively. For the case of Uniform Cost search above, the input `1 3 2 4` are the step costs of N, S, E, W respectively (they are separated by a space). There are 4 outputs and are the same as in the previous assignment (a04).

Obviously for ucs, if N has a higher cost than other actions, then the graph search will prefer to test the other directions first.

Q3. n–Queens (backtrack)

Prompt the user for `n` and find one solution to the n-queens problem using backtracking search in the following manner:

1. For the first recursive backtrack search function call, the function run through all options in the first column. For each option, it makes a second recursive backtrack search function call.
2. The second recursive backtrack search will run through all remaining valid queen positions for the second column (for a fixed chosen choice for a queen position in the first column). This function call makes a third recursive backtrack search function call.
3. The third recursive backtrack search will run through all remaining valid queen positions for the third column (for a fixed chosen choice for a queen position in the first column and a fixed chosen choice for a queen position in the second column). Etc.

Once a valid n-queens solution is found, the solution is printed and the program ends. An execution is given below (to fix format):

```
4
+-+-+-+-+
|Q| | | |
+-+-+-+-+
| | |Q| |
+-+-+-+-+
| |Q| | |
+-+-+-+-+
| | | |Q|
+-+-+-+-+
```

(Clearly the above is an incorrect solution.)

Note that the backtracking algorithm above does not use the graph search algorithm, i.e., it does not use a fringe.