CS30 - Assignment 8

Due at Sunday, April 5 at 11:59pm

For this assignment, you will be using depth first search to find solutions to the N queens problem! As always (and particularly for this assignment), read through the entire handout before you start working.

1 DFS code

You will be designing the state class that will be responsible for representing the state of the board and for asking questions about the board, specifically, if it's a goal state and what valid states you can get to from the current state. Once you have defined this functionality, then you should be able to solve puzzle with any search algorithm.

At

`http://www.cs.pomona.edu/~dkauchak/classes/cs30/assignments/assign8/dfs.txt`

I have provided an implementation of depth first search for you and some additional code that runs the depth first search and prints what is found. You won't need this code to start with, but once you finish writing the `NQueenState` class, you should copy and paste this code at the end of your file to do the *final* testing of your work.

2 A Class for the N Queens problem

To search for the state of possible solutions to the N queens problem (and many similar problems) we need to be able to do three key things:

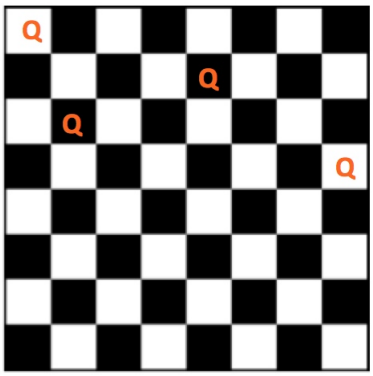
- Construct the starting state
- Determine if a state is a goal state
- From a state, determine what are the "next" states that we can get to by performing an action. In our case it will be putting one queen down in the next available row (working from the top down).

2.1 Specification

To support this functionality, write a class called `NQueenState`. The class should have the following instance variables:

- `self.size`: the size of the a side of the chess board.
- `self.num_queens_placed`: the number of *valid* queens that have been placed on the board. We are going to make sure that we only create states where there are no queens attacking eachother.
- `self.board`: a list of lists (or you can think of it as a matrix) representing the state of the board. Entry `self.board[row][col]` should be 1 if a queen is place there, 0 otherwise.

  For example, the following board:



would be represented as:

```
[[1, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 0, 0],
 [0, 1, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0]]
```

(Note, if you actually were to print this out it would be printed as one long line, but I've formatted it to be more readable.)

Your class should have *at least* the following methods:

- `__init__`

  Should take one parameter, the size of the side of the board and construct a new state with a board with no queens place. For example:

  ```
  start_state = NQueenState(8)
  ```

  would construct a state with a normal sized chess board (8 by 8).

- `__str__`

  Doesn't take any parameters and returns a string representation of the board. Something like:

  ```
  Board size: 8
  Number of queens placed: 8
  [1, 0, 0, 0, 0, 0, 0, 0]
  [0, 0, 0, 0, 1, 0, 0, 0]
  ```

```
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

- `is_valid_move`

  Takes two parameters, the row and column (indexed starting at 0, so 0, 0 is the upper left corner), and return `True` if putting a queen at row, col on this board would result in a valid state, `False` otherwise. A placement of a queen is valid if:

  1. We haven't already placed the size of the board queens on the board.
  2. The space doesn't have a queen in it already.
  3. A queen there wouldn't attack any other queens on the board.

  The last of these constraints is going to be the most work to check since you'll need to check that there isn't any queen anywhere in that row already, that there isn't a queen anywhere in that column already and that there isn't a queen anywhere diagonally on the board from this queen (see the hints for more on this last one).

  Note that this function does NOT modify the board. It only asks IF putting a queen there would be valid.

- `add_queen`

  Takes two parameters, the row and column, and *returns a new state* that is a copy of the current state, but has the queen put down at row, col. You may assume that the position is a valid position (though it won't hurt you to put in a call to your `is_valid_move` method and print out an error, just in case.

  Recall that the `copy` module has a function called `deepcopy` that will allow you to create a deep copy of the current state:

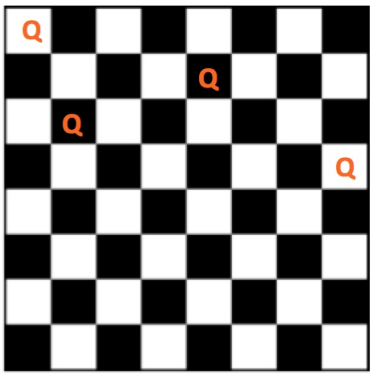
  ```
  new_state = copy.deepcopy(self)
  ```

  After copying, you will then need to update this new state appropriately then return it. Make sure that you're updating *all* of the instance variables of this new state appropriately, not just `self.board`.

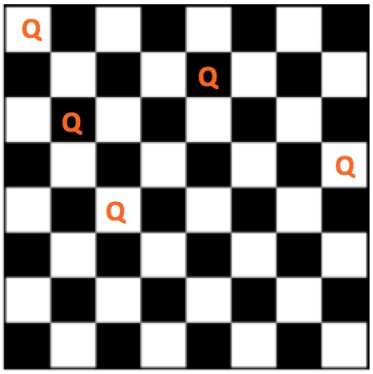
- `next_states`

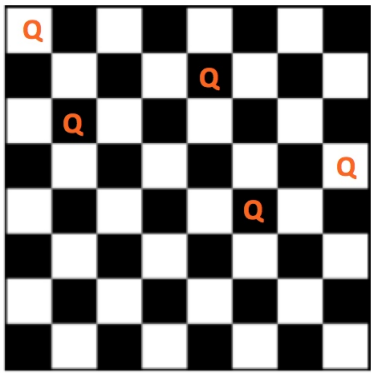
  Doesn't take any parameters, but returns a list of *valid* states that can be reached by adding one more queen to the board. There are many ways that this could be implemented. We're going to impose the additional restriction that we're going to add queens from the top row and work our way down. So, from the start state (with an empty board) the `next_states` method will return a list of all the states with the queen placed at any of the entries in the top row only.

  For example, if we were to call the `next_states` function on:



we would get back a list of two new states:





To generate these we only considered the 8 possible locations in the next row, of which only these two don't result in conflicts.

Note, this method should utilize the previous two methods.

- `is_goal`

  Doesn't take any parameters and returns `True` if this state represents a goal state and `False` otherwise. Note that since we're making sure that only valid states are generated, then this function is very easy. (Hint: if your board is of size 8 and you have 8 queens on the board AND it's a valid board, then it's a solution.)

In all methods above the parameter should also include `self` as a parameter. You may (and I would encourage you to) implement additional methods that may help make that above methods easier to do. Like writing functions, it is a good idea to break a method down into sub-methods (in reality, methods are just special versions of functions so all the "good" style things we've talked about with respect to functions also applies to methods).

Once you've implemented the class, add the DFS code to the end of your program. You will submit your class with the working program.

3 Hints/guidance

- Make sure you understand exactly what the class is storing and representing and what each of the methods are supposed to do. If you have questions, come talk to me.
- Do NOT try and write all of the methods and then test your class by running the dfs code and hoping that it works. It very likely won't and then you won't know where your problem is. Write one method at a time and then test that one method to make sure it works.

  Here are a few examples of how you can do this:

  - `init`

    ```
    state = NQueenState(4)
    print state.size
    print state.num_queens_placed
    print state.board
    ```

    would print

    ```
    4
    0
    [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
    ```

  - `str`

    ```
    state = NQueenState(8)
    print state
    ```

  - `is_valid_move`

    ```
    state = NQueenState(4)
    print state.is_valid_move(0, 0) # should be True
    state.board[0][0] = 1
    state.num_queens_placed = 1
    print state.is_valid_move(0, 0) # should be False
    print state.is_valid_move(0, 3) # should be False
    print state.is_valid_move(1, 2) # should be True
    ```

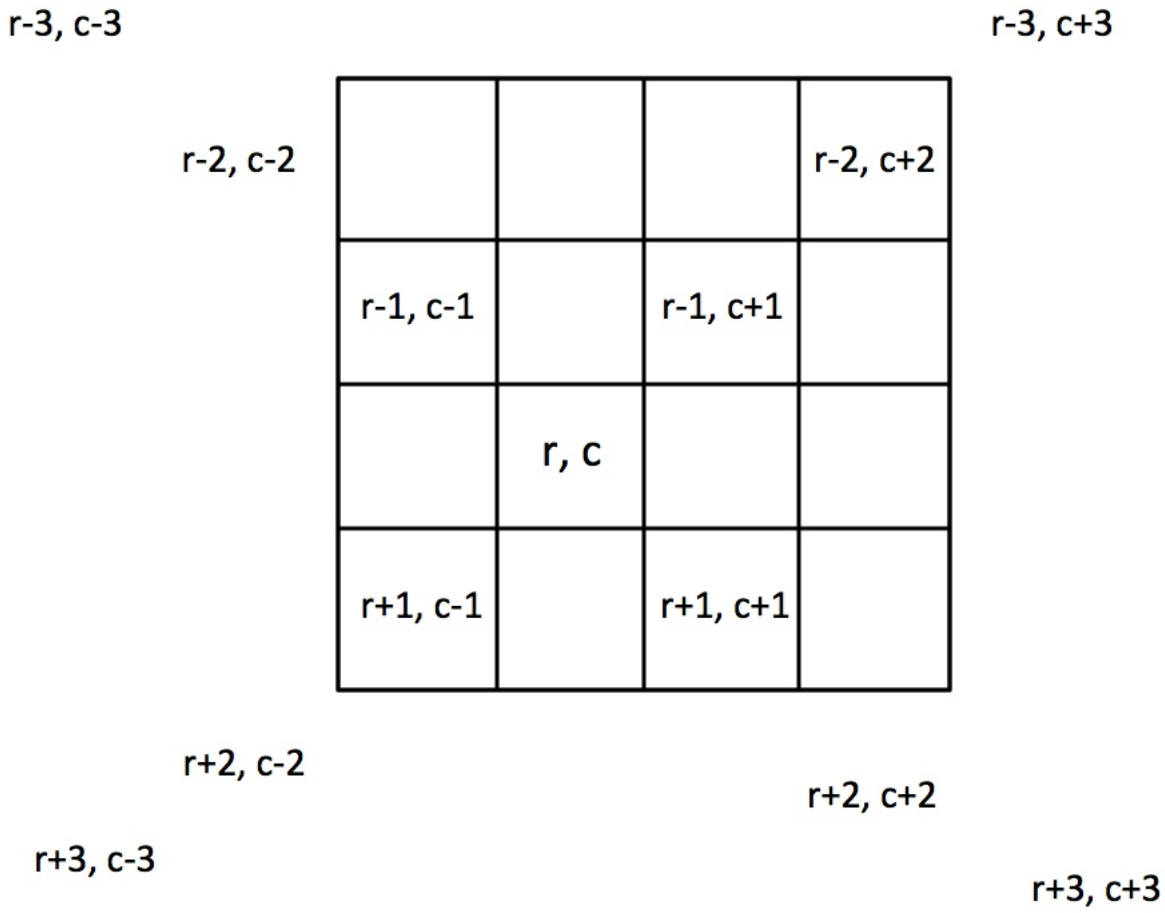
  - `add_queen`

    ```
    state = NQueenState(4)
    new_state = state.add_queen(0,0)
    print state # should print out the original, empty state
    print new_state # should print out the new state with a 1 in the upper left corner
    ```

  This list of examples is not meant to be extensive, but is meant to give you an idea of how you can test your methods one at a time.

- As always, look at the examples we looked at in class. They often have portions of code that do similar things to what you will be doing here.

- The hardest part about checking if there is a conflict, is checking the if a new queen attacks any of the existing ones diagonally. Consider the following diagram where we're trying to check if an queen at row, colum `(r, c)` would conflict diagonally on a board of size 4:



  We need to check all of the entries that fall inside the board (besides `r,c`). Think about how you can do this with one or more loops. You can either do this by enumerating only those in the board and checking those *or* enumerating all of the ones in the figure and then also checking if they're actually in the bounds of the board. Either way is fine.

- Finally, make sure you think about how all of the methods fit together. Some of the later methods I've asked you do implement will use (or will be much easier to write if you use) the previous methods. I've written them in the description above in roughly the way I'd suggest implementing them.

- There are 2 solutions to the 4-queens problem and 92 to the 8-queens problem.

When you're done

You should have a single .py file with both the `NQueenState` class as well as the dfs program. Put this all in a file named with your first name and last name followed by `assign8.py`.

- You should have comments at the very beginning of the file stating your name, course, assignment number and the date.
- Each function should have an appropriate docstring.
- Your class should have an appropriate docstring.
- Include other miscellaneous comments to make things clear.

Submit your .py file online using the courses submission mechanism.

Grading

	points
`init`	3
`str`	3
`is_valid_move`	5
`add_queen`	3
`next_states`	3
`is_goal`	2
`meets specifications`	3
comments/style	3
total	25