HW3: Hill Climbing

Due Oct 1 by 11am **Points** 100 **Submitting** a file upload **File Types** py
Available Sep 24 at 12am - Oct 2 at 11:59pm 9 days

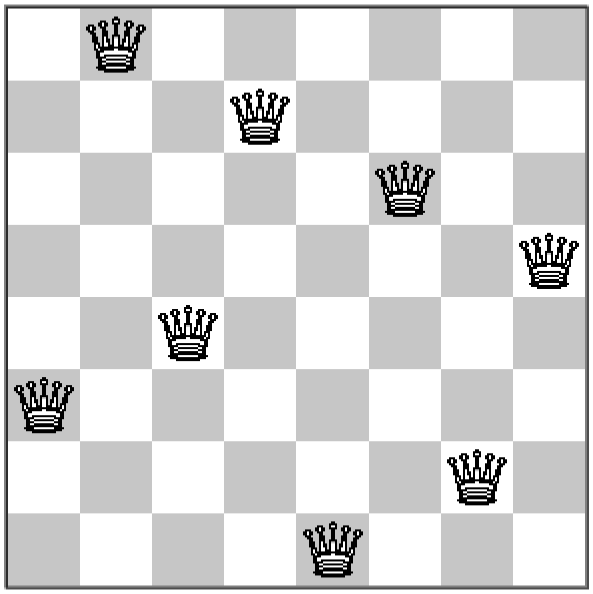
This assignment was locked Oct 2 at 11:59pm.

Program Goals

- Deepen understanding of state space generation
- Practice implementation of search algorithms

Summary

We've introduced the 8-queens problem in class: place 8 queens on a chessboard such that no queen is attacking any other queen.



A solution for an 8x8 board with state [2, 7, 3, 6, 0, 5, 1, 4]

Recall that in the game of chess, queens attack any piece they can "see" (i.e., there is nothing between the queen and the attacking piece) in the same row, column, or diagonal.

In this assignment, you will generalize this problem to a board of arbitrary (square) size and the equivalent number of queens and add a "static point" to the board. It means that at any time, there must be a queen on the static point. You only have to place the other N-1 queens on the board.

Given the size of the board (N > 0) and location of the static point (0 <= static_x < N and 0 <= static_y < N), you will implement a hill-climbing algorithm to solve the problem.

Program Specification

The code for this program must be written in Python in a file called **nqueens.py**. **You should only submit one file with the name nqueens.py, and make sure it includes all functions needed.** Do not submit a Jupyter notebook .ipynb file.

Your state representation must be a list of N integers, 0-indexed, representing the row the queen is occupying in each column (see figure above). For simplicity of representation, we will retain our assumption that there is a single queen per column. When you generate the successor states, there should not be any states with two queens in the same column.

Goal State

There are multiple possible configurations for a goal state, defined as a state in which no queen is attacking any other queen per the rules above. One of your tasks will be to define an evaluation function for the states such that the goal state has the lowest value (0) when the function is applied to it.

Functions

For this program you should write 5 Python functions:

1. **succ(state, static_x, static_y)** -- given a state of the board, **return** a list of all valid successor states
2. **f(state)** -- given a state of the board, **return** an integer score such that the goal state scores 0
3. **choose_next(curr, static_x, static_y)** -- given the current state, use succ() to generate the successors and **return** the selected next state
4. **n_queens(initial_state, static_x, static_y)** -- run the hill-climbing algorithm from a given initial state, **return** the convergence state
5. **n_queens_restart(n, k, static_x, static_y)** -- run the hill-climbing algorithm on an n*n board with random restarts

You may add other functions as you see fit, but these functions must be present and work as described.

Generate Successors

This function should return a list of lists, containing all valid successor states of the given current state. For the purposes of this program, a successor of a current state is any valid state that differs from the current state by ONE queen's location, which means you could only move one queen by one tile.

- We WILL allow multiple queens to occupy the same row or the same diagonal line, but not the same column, as it could not be described by the representation of the state.
- We WILL NOT allow moving the queen on the static point.
- If there is not a queen on the static point in the input state, return an empty list
- The returned states should be sorted by the ascending order

For example, if the static point is at (0,0) and n = 3, the valid successors of the state `[0, 1, 2]` are:

```
>>> succ([0, 1, 2], 0, 0)
=> [[0, 0, 2], [0, 1, 1], [0, 2, 2]]
```

There is no valid successors of the state `[1, 1, 2]`, as there is no queen on the static point. The function should return an empty list in this case.

```
>>> succ([1, 1, 2], 0, 0)
=> []
```

For sorting the states, using **sorted()** method will be sufficient to get the expected ascending order.

Evaluate a State

As in class, we'll define our f() to be the number of queens that are **being attacked**. Even if a queen is attacked by multiple other queens, it will only be counted once.

Given n=3, here are some example f() values:

```
[1, 2, 2] - f=3
[2, 2, 2] - f=3
[0, 0, 2] - f=3
[0, 2, 0] - f=2
[0, 2, 1] - f=2
```

Select Next State

Given a current state and the location of the static point, use the previous two functions to select the next state to evaluate per the following rules:

- If one of the possible states (including the current state) has a uniquely low score, select that state
- Otherwise, sort the states in ascending order (as though they were integers) and select the "lowest" state

If the state is invalid, which means there is no queen on the static point, return `None`.

```
>>> choose_next([1, 1, 2], 1, 1)
=> [0, 1, 2]
>>> choose_next([0, 2, 0], 0, 0)
=> [0, 2, 0]
>>> choose_next([0, 1, 0], 0, 0)
=> [0, 2, 0]
>>> choose_next([0, 1, 0], 0, 1)
=> None
```

N-Queens

Run the hill-climbing algorithm as specified in class on a given initial state until convergence, that is, until your algorithm finds a local minimum and gets stuck (or solves the problem). Here, we define that a local minimum means that you encountered two states with the same f value in the consecutive two steps. You should use choose_next() to choose the next state from the current state, which means the next state could be the same with your current state and your function should return in this situation.

Your **printed** output for this function should look like the black text below, followed by the **returned** minimum state in green:

```
>>> n_queens([0, 1, 2, 3, 5, 6, 6, 7], 1, 1)
[0, 1, 2, 3, 5, 6, 6, 7] - f=8
[0, 1, 2, 3, 5, 7, 6, 7] - f=7
[0, 1, 1, 3, 5, 7, 6, 7] - f=7
=> [0, 1, 1, 3, 5, 7, 6, 7]
# Here function encounters two consecutive states [0, 1, 2, 3, 5, 7, 6, 7] and [0, 1, 1, 3, 5, 7, 6, 7] with
 the same f value 7, so it gets stuck and returns

>>> n_queens([0, 7, 3, 4, 7, 1, 2, 2], 0, 0)
[0, 7, 3, 4, 7, 1, 2, 2] - f=7
[0, 6, 3, 4, 7, 1, 2, 2] - f=6
[0, 6, 3, 5, 7, 1, 2, 2] - f=4
[0, 6, 3, 5, 7, 1, 3, 2] - f=3
[0, 6, 3, 5, 7, 1, 4, 2] - f=0
=> [0, 6, 3, 5, 7, 1, 4, 2]
```

Each step of the hill-climbing function should print its **current state** along with that state's **f() value**, and ultimately return the best state you find.

N-Queens with Random Restarts

Generate a random (valid!) initial state for your n*n board, and use your n_queens() function on that state. If the converged state does not solve the problem with an f() value of 0, generate a new random (valid!) initial state and try again. Try again **up to k times**.

If you find an optimal solution before you reach k restarts, print the solution and terminate.

If you reach k before finding an optimal solution with a score of 0, print the best solution(s) in sorted order.

An example of the output is illustrated below. The function does not need to return anything. Note that your output **may not be the same with the examples below as the init state is generated randomly**:

```
>>> n_queens_restart(7, 10, 0, 0)
[0, 2, 0, 6, 4, 1, 5] - f=3
[0, 3, 5, 2, 4, 6, 4] - f=3
[0, 5, 4, 2, 6, 3, 1] - f=3

>>> n_queens_restart(8, 1000, 0, 0)
[0, 6, 4, 7, 1, 3, 5, 2] - f=0
```

Python's random module will be helpful in generating random initial states. Each column's coordinate should be uniformly distributed over the possible coordinates, except for the column with the static point. To generate a single uniform integer between 0 (inclusive) and n (**inclusive**), the Python code is:

```
>>> import random
>>> n = random.randint(0, 8)
```

Be sure to account for the static point! Set the random_seed to 1 before using randint to make sure that the result of your program is reproducible, like the example below:

```
import random

def n_queens_restart(n, k, static_x, static_y):
  random.seed(1)

  random_int_1 = random.randint(0, 8)
  random_int_2 = random.randint(0, 8)
  ...
  // Other part of your program
```

Remember that you just need to set the seed **once** at the beginning of your function, or you will always get the same number.

Test & Grading

We provide a unit test module which includes all examples mentioned above to help you test your program and make sure that your program is compatible with the grading script. Remember that **we will use more test cases when grading**, and you are encouraged to add your own test cases to test your program.

To use the test script, first upload your nqueens.py to a CSL machine. In your working directory, run the commands below:

```
wget http://pages.cs.wisc.edu/~ywang2395/cs540/nqueens_test  (http://pages.cs.wisc.edu/~ywang2395/cs540/nqueens_test) 
mv nqueens_test nqueens_test.py
python3 nqueens_test.py
```

You could also directly click the link to download it. The test is also written in Python, so feel free to open it by a text editor and modify it. If you passed all tests you will see an OK message. The unit test module could only be executed under python3 environment, and we **assume that your program should be compatible with python3.**

We do not have a strict limit on the running time, but please make sure that your program could finish in a reasonable period of time. In normal cases, your program could finish running all test cases above in less than 0.1s, and finish running `n_queens_restart(8, 1000, 0, 0)` in less than 1s.

Some Rubric (1)

Criteria	Ratings		Pts
Test Cases The program will be graded based on test cases for all five functions. For n_queens_restart() the output of the function may not be the same as the standard output but for other functions, their output should be exactly the same with the test cases.	**80.0 pts Full Marks** Passed all test cases	**0.0 pts No Marks** Passed no test cases	80.0 pts
Valid Submission Your submission should be a valid submission that is compatible with our test script, i.e. a file named nqueens.py with five functions mentioned above	**20.0 pts Full Marks**	**0.0 pts No Marks**	20.0 pts
			Total Points: 100.0