CMPSC 442: Homework 2 [100 points]

Release Date	Monday, August 30, 2021, 12:00 am
Due Date	Thursday, September 16, 2021, 11:59 pm

TO PREPARE AND SUBMIT HOMEWORK

Follow these steps exactly, so the Gradescope autgrader can grade your homework. Failure to do so will result in a zero grade:

1. You *must* download the homework template file `homework2_cmpsc442.py` from Canvas. Each template file is a python file that gives you a headstart in creating your homework python script with the correct function names for autograding.

2. You *must* download the code for the GUI `homework2_lights_out_gui.py` from Canvas. You will use this to interactively play the Lights Out puzzle.

3. You *must* rename the file by replacing `cmpsc442` with your PSU id from your official PSU. For example, if your PSU email id is abcd1234, you would rename your file as `homework2_abcd1234.py` to submit to Gradescope.

4. Upload your *py file to Gradescope by its due date. It is your responsibility to upload on time. If the submission to Gradescope closes before you upload, your homework will be counted late, and you might get a zero grade.

5. Make sure your file can import before you submit; the autograder imports your file. If it won't import, you will get a zero.

Instructions

In this assignment, you will explore three classic puzzles from the perspective of uninformed search.

A skeleton file `homework2_cmpsc442.py` containing empty definitions for each question has been provided. Since portions of this assignment will be graded automatically, none of the names or function signatures in this file should be modified. However, you are free to introduce additional variables or functions if needed.

You may import definitions from any standard Python library, and are encouraged to do so in cases where you find yourself reinventing the wheel. If you are unsure where to start, consider taking a look at the data structures and functions defined in the `collections`, `itertools`, `math`, and `random` modules.

You will find that in addition to a problem specification, most programming questions also include a pair of examples from the Python interpreter. These are meant to illustrate typical use cases to clarify the assignment, and are not comprehensive test suites. In addition to performing your own testing, you are strongly encouraged to verify that your code gives the expected output for these examples before submitting.

You are strongly encouraged to follow the Python style guidelines set forth in PEP 8, which was written in part by the creator of Python. However, your code will not be graded for style.

1. N-Queens [25 points]

In this section, you will develop a solver for the *n*-queens problem, wherein *n* queens are to be placed on an *n x n* chessboard so that no pair of queens can attack each other. Recall that in chess, a queen can attack any piece that lies in the same row, column, or diagonal as itself.

A brief treatment of this problem for the case where $n = 8$ is given in Section 3.2 of the 3rd edition of the textbook, which you may wish to consult for additional information. See the Canvas Homework 2 page for a link to the AIMA 3rd edition *discussion of uninformed search problems*

1. **[5 points]** Rather than performing a search over all possible placements of queens on the board, it is sufficient to consider only those configurations for which each row contains exactly one queen. Without taking any of the chess-specific constraints between queens into account, implement the pair of functions `num_placements_all(n)` and `num_placements_one_per_row(n)` that return the number of possible placements of *n* queens on an *n x n* board without or with this additional restriction. Think carefully about why this restriction is valid, and note the extent to which it reduces the size of the search space. You should assume that all queens are indistinguishable for the purposes of your calculations.

2. **[5 points]** With the answer to the previous question in mind, a sensible representation for a board configuration is a list of numbers between *0* and *n - 1*, where the *i*th number designates the column of the queen in row *i* for $0 \leq i < n$. A complete configuration is then specified by a list containing *n* numbers, and a partial configuration is specified by a list containing fewer than *n* numbers. Write a function `n_queens_valid(board)` that accepts such a list and returns `True` if no queen can attack another, or `False` otherwise. Note that the board size need not be included as an additional argument to decide whether a particular list is valid.

```
>>> n_queens_valid([0, 0])
False
>>> n_queens_valid([0, 2])
True
```

```
>>> n_queens_valid([0, 1])
False
>>> n_queens_valid([0, 3, 1])
True
```

3. **[15 points]** Write a function `n_queens_solutions(n)` that yields all valid placements of *n* queens on an *n x n* board, using the representation discussed above. The output may be generated in any order you see fit. Your solution should be implemented as a depth-first search, where queens are successively placed in empty rows until all rows have been filled. *Hint: You may find it helpful to define a helper function `n_queens_helper(n, board)` that yields all valid placements which extend the partial solution denoted by `board`.*

   Though our discussion of search in class has primarily covered algorithms that return just a single solution, the extension to a generator which yields all solutions is relatively simple. Rather than using a `return` statement when a solution is encountered, `yield` that solution instead, and then continue the search.

```
>>> solutions = n_queens_solutions(4)
>>> next(solutions)
```

```
>>> list(n_queens_solutions(6))
[[1, 3, 5, 0, 2, 4], [2, 5, 1, 4, 0, 3],
```

```
[1, 3, 0, 2]
>>> next(solutions)
[2, 0, 3, 1]
```

```
 [3, 0, 4, 1, 5, 2], [4, 2, 0, 5, 3, 1]]
>>> len(list(n_queens_solutions(8)))
92
```

2. Lights Out [40 points]

The Lights Out puzzle consists of an *m x n* grid of lights, each of which has two states: on and off. The goal of the puzzle is to turn all the lights off, with the caveat that whenever a light is toggled, its neighbors above, below, to the left, and to the right will be toggled as well. If a light along the edge of the board is toggled, then fewer than four other lights will be affected, as the missing neighbors will be ignored.

In this section, you will investigate the behavior of Lights Out puzzles of various sizes by implementing a `LightsOutPuzzle` class. Once you have completed the problems in this section, you can test your code in an interactive setting using the provided GUI. See the end of the section for more details.

1. **[2 points]** A natural representation for this puzzle is a two-dimensional list of Boolean values, where `True` corresponds to the on state and `False` corresponds to the off state. In the `LightsOutPuzzle` class, write an initialization method `__init__(self, board)` that stores an input board of this form for future use. Also write a method `get_board(self)` that returns this internal representation. You additionally may wish to store the dimensions of the board as separate internal variables, though this is not required.

```
>>> b = [[True, False], [False, True]]
>>> p = LightsOutPuzzle(b)
>>> p.get_board()
[[True, False], [False, True]]
```

```
>>> b = [[True, True], [True, True]]
>>> p = LightsOutPuzzle(b)
>>> p.get_board()
[[True, True], [True, True]]
```

2. **[3 points]** Write a top-level function `create_puzzle(rows, cols)` that returns a new `LightsOutPuzzle` of the specified dimensions with all lights initialized to the off state.

```
>>> p = create_puzzle(2, 2)
>>> p.get_board()
[[False, False], [False, False]]
```

```
>>> p = create_puzzle(2, 3)
>>> p.get_board()
[[False, False, False],
 [False, False, False]]
```

3. **[5 points]** In the `LightsOutPuzzle` class, write a method `perform_move(self, row, col)` that toggles the light located at the given row and column, as well as the appropriate neighbors.

```
>>> p = create_puzzle(3, 3)
>>> p.perform_move(1, 1)
>>> p.get_board()
[[False, True, False],
 [True,  True, True ],
 [False, True, False]]
```

```
>>> p = create_puzzle(3, 3)
>>> p.perform_move(0, 0)
>>> p.get_board()
[[True,  True,  False],
 [True,  False, False],
 [False, False, False]]
```

4. **[5 points]** In the `LightsOutPuzzle` class, write a method `scramble(self)` which scrambles the puzzle by calling `perform_move(self, row, col)` with probability *1/2* on each location on the board. This guarantees that the resulting configuration will be solvable, which may not be true if lights are flipped individually. *Hint: After importing the `random` module with the statement `import random`, the expression `random.random() < 0.5` generates the values `True` and `False` with equal probability.*

5. **[2 points]** In the `LightsOutPuzzle` class, write a method `is_solved(self)` that returns whether all lights on the board are off.

```
>>> b = [[True, False], [False, True]]
>>> p = LightsOutPuzzle(b)
>>> p.is_solved()
False
```

```
>>> b = [[False, False], [False, False]]
>>> p = LightsOutPuzzle(b)
>>> p.is_solved()
True
```

6. **[3 points]** In the `LightsOutPuzzle` class, write a method `copy(self)` that returns a new `LightsOutPuzzle` object initialized with a deep copy of the current board. Changes made to the original puzzle should not be reflected in the copy, and vice versa.

```
>>> p = create_puzzle(3, 3)
>>> p2 = p.copy()
>>> p.get_board() == p2.get_board()
True
```

```
>>> p = create_puzzle(3, 3)
>>> p2 = p.copy()
>>> p.perform_move(1, 1)
>>> p.get_board() == p2.get_board()
False
```

7. **[5 points]** In the `LightsOutPuzzle` class, write a method `successors(self)` that yields all successors of the puzzle as (move, new-puzzle) tuples, where moves themselves are (row, column) tuples. The second element of each successor should be a new `LightsOutPuzzle` object whose board is the result of applying the corresponding move to the current board. The successors may be generated in whichever order is most convenient.

```
>>> p = create_puzzle(2, 2)
>>> for move, new_p in p.successors():
...     print(move, new_p.get_board())
...
(0, 0) [[True, True], [True, False]]
(0, 1) [[True, True], [False, True]]
(1, 0) [[True, False], [True, True]]
(1, 1) [[False, True], [True, True]]
```

```
>>> for i in range(2, 6):
...     p = create_puzzle(i, i + 1)
...     print(len(list(p.successors())))
...
6
12
20
30
```

8. **[15 points]** In the `LightsOutPuzzle` class, write a method `find_solution(self)` that returns an optimal solution to the current board as a list of moves, represented as (row, column) tuples. If more than one optimal solution exists, any of them may be returned. Your solver should be implemented using a breadth-first graph search, which means that puzzle states should not be

added to the frontier if they have already been visited, or are currently in the frontier. If the current board is not solvable, the value `None` should be returned instead. You are highly encouraged to reuse the methods defined in the previous exercises while developing your solution.

Hint: For efficient testing of duplicate states, consider using tuples representing the boards of the `LightsOutPuzzle` objects being explored rather than their internal list-based representations. You will then be able to use the built-in `set` data type to check for the presence or absence of a particular state in near-constant time.

```
>>> p = create_puzzle(2, 3)
>>> for row in range(2):
...     for col in range(3):
...         p.perform_move(row, col)
...
>>> p.find_solution()
[(0, 0), (0, 2)]
```

```
>>> b = [[False, False, False],
...      [False, False, False]]
>>> b[0][0] = True
>>> p = LightsOutPuzzle(b)
>>> p.find_solution() is None
True
```

Once you have implemented the functions and methods described in this section, you can play with an interactive version of the Lights Out puzzle using the provided GUI by running the following command:

```
python homework2_lights_out_gui.py rows cols
```

The arguments `rows` and `cols` are positive integers designating the size of the puzzle.

In the GUI, you can click on a light to perform a move at that location, and use the side menu to scramble or solve the puzzle. The GUI is merely a wrapper around your implementations of the relevant functions, and may therefore serve as a useful visual tool for debugging.

3. Linear Disk Movement [30 points]

In this section, you will investigate the movement of disks on a linear grid.

The starting configuration of this puzzle is a row of L cells, with disks located on cells 0 through $n - 1$. The goal is to move the disks to the end of the row using a constrained set of actions. At each step, a disk can only be moved to an adjacent empty cell, or to an empty cell two spaces away, provided another disk is located on the intervening cell. Given these restrictions, it can be seen that in many cases, no movements will be possible for the majority of the disks. For example, from the starting position, the only two options are to move the last disk from cell $n - 1$ to cell n, or to move the second-to-last disk from cell $n - 2$ to cell n.

1. **[15 points]** Write a function `solve_identical_disks(length, n)` that returns an optimal solution to the above problem as a list of moves, where `length` is the number of cells in the row and `n` is the number of disks. Each move in the solution should be a two-element tuple of the form (from, to) indicating a disk movement from the first cell to the second. As suggested by its name, this function should treat all disks as being identical.

   Your solver for this problem should be implemented using a breadth-first graph search. The exact solution produced is not important, as long as it is of minimal length.

Unlike in the previous two sections, no requirement is made with regards to the manner in which puzzle configurations are represented. Before you begin, think carefully about which data structures might be best suited for the problem, as this choice may affect the efficiency of your search.

```
>>> solve_identical_disks(4, 2)
[(0, 2), (1, 3)]
>>> solve_identical_disks(5, 2)
[(0, 2), (1, 3), (2, 4)]
```

```
>>> solve_identical_disks(4, 3)
[(1, 3), (0, 1)]
>>> solve_identical_disks(5, 3)
[(1, 3), (0, 1), (2, 4), (1, 2)]
```

2. **[15 points]** Write a function `solve_distinct_disks(length, n)` that returns an optimal solution to the same problem with a small modification: in addition to moving the disks to the end of the row, their final order must be the reverse of their initial order. More concretely, if we abbreviate `length` as *L*, then a desired solution moves the first disk from cell *0* to cell *L - 1*, the second disk from cell *1* to cell *L - 2*, . . . , and the last disk from cell *n - 1* to cell *L - n*.

   Your solver for this problem should again be implemented using a breadth-first graph search. As before, the exact solution produced is not important, as long as it is of minimal length.

```
>>> solve_distinct_disks(4, 2)
[(0, 2), (2, 3), (1, 2)]
>>> solve_distinct_disks(5, 2)
[(0, 2), (1, 3), (2, 4)]
```

```
>>> solve_distinct_disks(4, 3)
[(1, 3), (0, 1), (2, 0), (3, 2), (1, 3),
 (0, 1)]
>>> solve_distinct_disks(5, 3)
[(1, 3), (2, 1), (0, 2), (2, 4), (1, 2)]
```

4. Feedback [5 points]

1. **[1 point]** Approximately how long did you spend on this assignment?
2. **[2 points]** Which aspects of this assignment did you find most challenging? Were there any significant stumbling blocks?
3. **[2 points]** Which aspects of this assignment did you like? Is there anything you would have changed?