CMPSC 442: Informed Search

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. For this homework, you will also need to download these files from Canvas:
   `homework2_dominoes_game_gui.py`
   `homework2_grid_navigation_gui.py`
   `homework2_tile_puzzle_gui.py`
2. 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`.

3. Upload your `homework2_abcd1234.py` file to Gradescope by due date.

4. 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 a number of games and puzzles from the perspectives of informed and adversarial 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 case 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`, `Queue`, 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. Tile Puzzle [40 points]

Recall from class that the Eight Puzzle consists of a *3 x 3* board of sliding tiles with a single empty space. For each configuration, the only possible moves are to swap the empty tile with one of its neighboring tiles. The goal state for the puzzle consists of tiles 1-3 in the top row, tiles 4-6 in the middle row, and tiles 7 and 8 in the bottom row, with the empty space in the lower-right corner.

In this section, you will develop two solvers for a generalized version of the Eight Puzzle, in which the board can have any number of rows and columns. We have suggested an approach similar to the one used to create a Lights Out solver in Homework 1, and indeed, you may find that this pattern can be abstracted to cover a wide range of puzzles. If you wish to use the provided GUI for testing, described in more detail at the end of the section, then your implementation must adhere to the recommended interface. However, this is not required, and no penalty will imposed for using a different approach.

A natural representation for this puzzle is a two-dimensional list of integer values between *0* and $r \cdot c - 1$ (inclusive), where *r* and *c* are the number of rows and columns in the board, respectively. In this problem, we will adhere to the convention that the *0*-tile represents the empty space.

1. **[0 points]** In the `TilePuzzle` class, write an initialization method `__init__(self, board)` that stores an input board of this form described above for future use. You additionally may wish to store the dimensions of the board as separate internal variables, as well as the location of the empty tile.

2. **[0 points]** *Suggested infrastructure.*

   In the `TilePuzzle` class, write a method `get_board(self)` that returns the internal representation of the board stored during initialization.

   ```
   >>> p = TilePuzzle([[1, 2], [3, 0]])
   >>> p.get_board()
   [[1, 2], [3, 0]]
   ```

   ```
   >>> p = TilePuzzle([[0, 1], [3, 2]])
   >>> p.get_board()
   [[0, 1], [3, 2]]
   ```

   Write a top-level function `create_tile_puzzle(rows, cols)` that returns a new `TilePuzzle` of the specified dimensions, initialized to the starting configuration. Tiles *1* through $r \cdot c - 1$ should be arranged starting from the top-left corner in row-major order, and tile *0* should be located in the lower-right corner.

```
>>> p = create_tile_puzzle(3, 3)
>>> p.get_board()
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

```
>>> p = create_tile_puzzle(2, 4)
>>> p.get_board()
[[1, 2, 3, 4], [5, 6, 7, 0]]
```

In the `TilePuzzle` class, write a method `perform_move(self, direction)` that attempts to swap the empty tile with its neighbor in the indicated direction, where valid inputs are limited to the strings `"up"`, `"down"`, `"left"`, and `"right"`. If the given direction is invalid, or if the move cannot be performed, then no changes to the puzzle should be made. The method should return a Boolean value indicating whether the move was successful.

```
>>> p = create_tile_puzzle(3, 3)
>>> p.perform_move("up")
True
>>> p.get_board()
[[1, 2, 3], [4, 5, 0], [7, 8, 6]]
```

```
>>> p = create_tile_puzzle(3, 3)
>>> p.perform_move("down")
False
>>> p.get_board()
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

In the `TilePuzzle` class, write a method `scramble(self, num_moves)` which scrambles the puzzle by calling `perform_move(self, direction)` the indicated number of times, each time with a random direction. This method of scrambling guarantees that the resulting configuration will be solvable, which may not be true if the tiles are randomly permuted. *Hint: The `random` module contains a function `random.choice(seq)` which returns a random element from its input sequence.*

In the `TilePuzzle` class, write a method `is_solved(self)` that returns whether the board is in its starting configuration.

```
>>> p = TilePuzzle([[1, 2], [3, 0]])
>>> p.is_solved()
True
```

```
>>> p = TilePuzzle([[0, 1], [3, 2]])
>>> p.is_solved()
False
```

In the `TilePuzzle` class, write a method `copy(self)` that returns a new `TilePuzzle` 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_tile_puzzle(3, 3)
>>> p2 = p.copy()
>>> p.get_board() == p2.get_board()
True
```

```
>>> p = create_tile_puzzle(3, 3)
>>> p2 = p.copy()
>>> p.perform_move("left")
>>> p.get_board() == p2.get_board()
False
```

In the `TilePuzzle` class, write a method `successors(self)` that yields all successors of the

puzzle as (direction, new-puzzle) tuples. The second element of each successor should be a new `TilePuzzle` 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, as long as successors corresponding to unsuccessful moves are not included in the output.

```
>>> p = create_tile_puzzle(3, 3)
>>> for move, new_p in p.successors():
...     print move, new_p.get_board()
...
up    [[1, 2, 3], [4, 5, 0], [7, 8, 6]]
left  [[1, 2, 3], [4, 5, 6], [7, 0, 8]]
```

```
>>> b = [[1,2,3], [4,0,5], [6,7,8]]
>>> p = TilePuzzle(b)
>>> for move, new_p in p.successors():
...     print move, new_p.get_board()
...
up    [[1, 0, 3], [4, 2, 5], [6, 7, 8]]
down  [[1, 2, 3], [4, 7, 5], [6, 0, 8]]
left  [[1, 2, 3], [0, 4, 5], [6, 7, 8]]
right [[1, 2, 3], [4, 5, 0], [6, 7, 8]]
```

3. **[20 points]** In the `TilePuzzle` class, write a method `find_solutions_iddfs(self)` that yields all optimal solutions to the current board, represented as lists of moves. Valid moves include the four strings `"up"`, `"down"`, `"left"`, and `"right"`, where each move indicates a single swap of the empty tile with its neighbor in the indicated direction. Your solver should be implemented using an iterative deepening depth-first search, consisting of a series of depth-first searches limited at first to *0* moves, then *1* move, then *2* moves, and so on. You may assume that the board is solvable. The order in which the solutions are produced is unimportant, as long as all optimal solutions are present in the output.

   Hint: This method is most easily implemented using recursion. First define a recursive helper method `iddfs_helper(self, limit, moves)` that yields all solutions to the current board of length no more than `limit` which are continuations of the provided move list. Your main method will then call this helper function in a loop, increasing the depth limit by one at each iteration, until one or more solutions have been found.

```
>>> b = [[4,1,2], [0,5,3], [7,8,6]]
>>> p = TilePuzzle(b)
>>> solutions = p.find_solutions_iddfs()
>>> next(solutions)
['up', 'right', 'right', 'down', 'down']
```

```
>>> b = [[1,2,3], [4,0,8], [7,6,5]]
>>> p = TilePuzzle(b)
>>> list(p.find_solutions_iddfs())
[['down', 'right', 'up', 'left', 'down',
 'right'], ['right', 'down', 'left',
 'up', 'right', 'down']]
```

4. **[20 points]** In the `TilePuzzle` class, write a method `find_solution_a_star(self)` that returns an optimal solution to the current board, represented as a list of direction strings. If multiple optimal solutions exist, any of them may be returned. Your solver should be implemented as an A* search using the Manhattan distance heuristic, which is reviewed below. You may assume that the board is solvable. During your search, you should take care not to add positions to the queue that have already been visited. It is recommended that you

use the `PriorityQueue` class from the `Queue` module.

Recall that the Manhattan distance between two locations *(r_1, c_1)* and *(r_2, c_2)* on a board is defined to be the sum of the componentwise distances: *|r_1 - r_2| + |c_1 - c_2|*. The Manhattan distance heuristic for an entire puzzle is then the sum of the Manhattan distances between each tile and its solved location.

```
>>> b = [[4,1,2], [0,5,3], [7,8,6]]
>>> p = TilePuzzle(b)
>>> p.find_solution_a_star()
['up', 'right', 'right', 'down', 'down']
```

```
>>> b = [[1,2,3], [4,0,5], [6,7,8]]
>>> p = TilePuzzle(b)
>>> p.find_solution_a_star()
['right', 'down', 'left', 'left', 'up',
 'right', 'down', 'right', 'up', 'left',
 'left', 'down', 'right', 'right']
```

If you implemented the suggested infrastructure described in this section, you can play with an interactive version of the Tile Puzzle using the provided GUI by running the following command:

```
python homework2_tile_puzzle_gui.py rows cols
```

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

In the GUI, you can use the arrow keys to perform moves on the puzzle, and can 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.

2. Grid Navigation [15 points]

In this section, you will investigate the problem of navigation on a two-dimensional grid with obstacles. The goal is to produce the shortest path between a provided pair of points, taking care to maneuver around the obstacles as needed. Path length is measured in Euclidean distance. Valid directions of movement include up, down, left, right, up-left, up-right, down-left, and down-right.

Your task is to write a function `find_path(start, goal, scene)` which returns the shortest path from the start point to the goal point that avoids traveling through the obstacles in the grid. For this problem, points will be represented as two-element tuples of the form (row, column), and scenes will be represented as two-dimensional lists of Boolean values, with `False` values corresponding empty spaces and `True` values corresponding to obstacles. Your output should be the list of points in the path, and should explicitly include both the start point and the goal point. Your implementation should consist of an A* search using the straight-line Euclidean distance heuristic. If multiple optimal solutions exist, any of them may be returned. If no optimal solutions exist, or if the start point or goal point lies on an obstacle, you should return the sentinal value `None`.

```
>>> scene = [[False, False, False],
...          [False, True , False],
```

```
>>> scene = [[False, True, False],
...          [False, True, False],
```

```
...             [False, False, False]]
>>> find_path((0, 0), (2, 1), scene)
[(0, 0), (1, 0), (2, 1)]
```

```
...             [False, True, False]]
>>> print find_path((0, 0), (0, 2), scene)
None
```

Once you have implemented your solution, you can visualize the paths it produces using the provided GUI by running the following command:

```
python homework2_grid_navigation_gui.py scene_path
```

The argument `scene_path` is a path to a scene file storing the layout of the target grid and obstacles. We use the following format for textual scene representation: `"."` characters correspond to empty spaces, and `"X"` characters correspond to obstacles.

3. Linear Disk Movement, Revisited [15 points]

Recall the Linear Disk Movement section from Homework 1. 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 square.

In a variant of the problem, the disks were distinct rather than identical, and the goal state was amended to stipulate that the final order of the disks should be the reverse of their initial order.

Implement an improved version of the `solve_distinct_disks(length, n)` function from Homework 1 that uses an A* search rather than an uninformed breadth-first search to find an optimal solution. As before, the exact solution produced is not important so long as it is of minimal length. You should devise a heuristic which is admissible but informative enough to yield significant improvements in performance.

4. Dominoes Game [30 points]

In this section, you will develop an AI for a game in which two players take turns placing *1 x 2* dominoes on a rectangular grid. One player must always place his dominoes vertically, and the other must always place his dominoes horizontally. The last player who successfully places a domino on the board wins.

As with the Tile Puzzle, an infrastructure that is compatible with the provided GUI has been suggested. However, only the search method will be tested, so you are free to choose a different approach if you find it more convenient to do so.

The representation used for this puzzle is a two-dimensional list of Boolean values, where `True` corresponds to a filled square and `False` corresponds to an empty square.

1. **[0 points]** In the `DominoesGame` class, write an initialization method `__init__(self, board)` that stores an input board of the form described above for future

use. You additionally may wish to store the dimensions of the board as separate internal variables, though this is not required.

2. **[0 points]** *Suggested infrastructure.*

   In the `DominoesGame` class, write a method `get_board(self)` that returns the internal representation of the board stored during initialization.

```
>>> b = [[False, False], [False, False]]
>>> g = DominoesGame(b)
>>> g.get_board()
[[False, False], [False, False]]
```

```
>>> b = [[True, False], [True, False]]
>>> g = DominoesGame(b)
>>> g.get_board()
[[True, False], [True, False]]
```

   Write a top-level function `create_dominoes_game(rows, cols)` that returns a new `DominoesGame` of the specified dimensions with all squares initialized to the empty state.

```
>>> g = create_dominoes_game(2, 2)
>>> g.get_board()
[[False, False], [False, False]]
```

```
>>> g = create_dominoes_game(2, 3)
>>> g.get_board()
[[False, False, False],
 [False, False, False]]
```

   In the `DominoesGame` class, write a method `reset(self)` which resets all of the internal board's squares to the empty state.

```
>>> b = [[False, False], [False, False]]
>>> g = DominoesGame(b)
>>> g.get_board()
[[False, False], [False, False]]
>>> g.reset()
>>> g.get_board()
[[False, False], [False, False]]
```

```
>>> b = [[True, False], [True, False]]
>>> g = DominoesGame(b)
>>> g.get_board()
[[True, False], [True, False]]
>>> g.reset()
>>> g.get_board()
[[False, False], [False, False]]
```

   In the `DominoesGame` class, write a method `is_legal_move(self, row, col, vertical)` that returns a Boolean value indicating whether the given move can be played on the current board. A legal move must place a domino fully within bounds, and may not cover squares which have already been filled.

   If the `vertical` parameter is `True`, then the current player intends to place a domino on squares `(row, col)` and `(row + 1, col)`. If the `vertical` parameter is `False`, then the current player intends to place a domino on squares `(row, col)` and `(row, col + 1)`. This convention will be followed throughout the rest of the section.

```
>>> b = [[False, False], [False, False]]
>>> g = DominoesGame(b)
>>> g.is_legal_move(0, 0, True)
True
>>> g.is_legal_move(0, 0, False)
True
```

```
>>> b = [[True, False], [True, False]]
>>> g = DominoesGame(b)
>>> g.is_legal_move(0, 0, False)
False
>>> g.is_legal_move(0, 1, True)
True
>>> g.is_legal_move(1, 1, True)
False
```

In the `DominoesGame` class, write a method `legal_moves(self, vertical)` which yields the legal moves available to the current player as (row, column) tuples. The moves should be generated in row-major order (i.e. iterating through the rows from top to bottom, and within rows from left to right), starting from the top-left corner of the board.

```
>>> g = create_dominoes_game(3, 3)
>>> list(g.legal_moves(True))
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1),
 (1, 2)]
>>> list(g.legal_moves(False))
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0),
 (2, 1)]
```

```
>>> b = [[True, False], [True, False]]
>>> g = DominoesGame(b)
>>> list(g.legal_moves(True))
[(0, 1)]
>>> list(g.legal_moves(False))
[]
```

In the `DominoesGame` class, write a method `perform_move(self, row, col, vertical)` which fills the squares covered by a domino placed at the given location in the specified orientation.

```
>>> g = create_dominoes_game(3, 3)
>>> g.perform_move(0, 1, True)
>>> g.get_board()
[[False, True,  False],
 [False, True,  False],
 [False, False, False]]
```

```
>>> g = create_dominoes_game(3, 3)
>>> g.perform_move(1, 0, False)
>>> g.get_board()
[[False, False, False],
 [True,  True,  False],
 [False, False, False]]
```

In the `DominoesGame` class, write a method `game_over(self, vertical)` that returns whether the current player is unable to place any dominoes.

```
>>> b = [[False, False], [False, False]]
>>> g = DominoesGame(b)
>>> g.game_over(True)
```

```
>>> b = [[True, False], [True, False]]
>>> g = DominoesGame(b)
>>> g.game_over(True)
```

```
False
>>> g.game_over(False)
False
```

```
False
>>> g.game_over(False)
True
```

In the `DominoesGame` class, write a method `copy(self)` that returns a new `DominoesGame` 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.

```
>>> g = create_dominoes_game(4, 4)
>>> g2 = g.copy()
>>> g.get_board() == g2.get_board()
True
```

```
>>> g = create_dominoes_game(4, 4)
>>> g2 = g.copy()
>>> g.perform_move(0, 0, True)
>>> g.get_board() == g2.get_board()
False
```

In the `DominoesGame` class, write a method `successors(self, vertical)` that yields all successors of the puzzle for the current player as (move, new-game) tuples, where moves themselves are (row, column) tuples. The second element of each successor should be a new `DominoesGame` object whose board is the result of applying the corresponding move to the current board. The successors should be generated in the same order in which moves are produced by the `legal_moves(self, vertical)` method.

```
>>> b = [[False, False], [False, False]]
>>> g = DominoesGame(b)
>>> for m, new_g in g.successors(True):
...     print m, new_g.get_board()
...
(0, 0) [[True, False], [True, False]]
(0, 1) [[False, True], [False, True]]
```

```
>>> b = [[True, False], [True, False]]
>>> g = DominoesGame(b)
>>> for m, new_g in g.successors(True):
...     print m, new_g get_board()
...
(0, 1) [[True, True], [True, True]]
```

Optional.

In the `DominoesGame` class, write a method `get_random_move(self, vertical)` which returns a random legal move for the current player as a (row, column) tuple. *Hint: The `random` module contains a function `random.choice(seq)` which returns a random element from its input sequence.*

3. **[30 points]** In the `DominoesGame` class, write a method `get_best_move(self, vertical, limit)` which returns a 3-element tuple containing the best move for the current player as a (row, column) tuple, its associated value, and the number of leaf nodes visited during the search. Recall that if the `vertical` parameter is `True`, then the current player intends to place a domino on squares `(row, col)` and

`(row + 1, col)`, and if the `vertical` parameter is `False`, then the current player intends to place a domino on squares `(row, col)` and `(row, col + 1)`. Moves should be explored row-major order, described in further detail above, to ensure consistency.

Your search should be a faithful implementation of the alpha-beta search given on page 170 of the course textbook, with the restriction that you should look no further than `limit` moves into the future. To evaluate a board, you should compute the number of moves available to the current player, then subtract the number of moves available to the opponent.

```
>>> b = [[False] * 3 for i in range(3)]
>>> g = DominoesGame(b)
>>> g.get_best_move(True, 1)
((0, 1), 2, 6)
>>> g.get_best_move(True, 2)
((0, 1), 3, 10)
```

```
>>> b = [[False] * 3 for i in range(3)]
>>> g = DominoesGame(b)
>>> g.perform_move(0, 1, True)
>>> g.get_best_move(False, 1)
((2, 0), -3, 2)
>>> g.get_best_move(False, 2)
((2, 0), -2, 5)
```

If you implemented the suggested infrastructure described in this section, you can play with an interactive version of the dominoes board game using the provided GUI by running the following command:

```
python homework2_dominoes_game_gui.py rows cols
```

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

In the GUI, you can click on a square to make a move, press 'r' to perform a random move, or press a number between *1* and *9* to perform the best move found according to an alpha-beta search with that limit. The GUI is merely a wrapper around your implementations of the relevant functions, and may therefore serve as a useful visual tool for debugging.