Instructions

In this assignment, you will explore two classic search problems from the perspective of informed search. A skeleton file homework3.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, and should not be taken as comprehensive test suites.

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.

Once you have completed the assignment, you should submit your file on Eniac using the following turnin command, where the flags -c and -p stand for “course” and “project”, respectively.

```bash
turnin -c cis391 -p hw3 homework3.py
```

You may submit as many times as you would like before the deadline, but only the last submission will be saved. To view a detailed listing of the contents of your most recent submission, you can use the following command, where the flag -v stands for “verbose”.

```bash
turnin -c cis391 -p hw3 -v
```

1 Tile Puzzle [60 Points]

Recall from class that the Eight Puzzle consists of a 3 × 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.

You will note that the development process closely mirrors the steps used to create a Lights Out solver in Homework 2. Indeed, this pattern can be abstracted to cover a wide range of puzzles, and will be applied in the next section as well.

1. **2 Points** 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.
In the TilePuzzle 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, as well as the location of the empty tile.

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

2. [3 Points] 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]]
```

3. [3 Points] 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]]
```

4. [2 Points] 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.

5. [3 Points] 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
```

6. [2 Points] 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
```

7. [5 Points] 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.

```python
>>> 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]]
```

8. **[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 direction strings. Your solver should be implemented using an iterative deepening depth-first search, which consists 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.

```python
>>> 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']
```

9. **[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.

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

Once you have implemented the functions and methods 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 homework3_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 Polygon Navigation [35 Points]

In this section, you will investigate the problem of navigation on a two-dimensional surface with polygonal obstacles. Your goal will be to produce the shortest path between a provided pair of points, taking care to maneuver around the obstacles as needed.

To help get you started, you have been provided with a number of useful classes and functions, including:

- A Point class representing a single point. Because the appropriate methods have been implemented, Point objects are hashable, and can therefore be used as dictionary keys or set members.
- A Polygon class representing a polygonal obstacle. This class has a get_vertices(self) method that returns the polygon’s vertices as a list of points, and a get_edges(self) method that returns the polygon’s edges as pairs of points.
- A function distance(p, q) which returns the Euclidean distance between points p and q.
- A function line_segments_intersect(p1, p2, q1, q2) which returns whether the line segment with endpoints p1 and p2 intersects the line segment with endpoints q1 and q2.
- A function load_scene(scene_path) which loads the obstacles specified in the provided file and returns them as a list of polygons. In a scene file, the coordinates of each obstacle’s vertices are listed on separate lines in a simple "x y" format, and obstacles themselves are separated by one or more empty lines. See the provided files for examples.

In order to complete the exercises below, you will not need to modify or augment these definitions, nor will you need to understand their internal workings. However, the implementations have been kept as simple as possible, and you are free to extend this miniature geometry library if you find it helpful to do so.

1. [5 Points] The naïve approach to the navigation problem assumes that all paths between the start point and the destination must be considered. However, this formulation of the state space is much too large. Explain why it is sufficient to consider only paths that travel directly between the vertices of the obstacles, except perhaps at the very beginning and end. You may assume that the obstacles are convex when making your argument.

2. [30 Points] Write a function find_path(start, goal, obstacles) which returns the shortest path from the start point to the goal point that avoids traveling through the polygonal obstacles. Your output should be a list of points, including both the start point and the goal point. Your implementation should consist of an A* search using the straight-line Euclidean distance heuristic. To avoid certain edge cases, you may assume that all of the obstacles are non-intersecting triangles, and that no pair of obstacles shares a vertex or an edge.

   Hint: You will likely find it helpful to first define a successor function that generates the possible successors of an intermediate point during the search, keeping in mind the location of the goal and the arrangement of the obstacles.
```python
>>> triangle = Polygon(
...     [Point(-2, 0), Point(0, 1),
...     Point(2, 0)])

>>> find_path(Point(1, 1),
...     Point(1.5, -1), [triangle])
[Point(1, 1), Point(2, 0),
  Point(1.5, -1)]

>>> triangle = Polygon(
...     [Point(0, 0), Point(1, 0),
...     Point(1, 1)])

>>> find_path(Point(-0.1, 0.1),
...     Point(1.1, 0.5), [triangle])
[Point(-0.1, 0.1), Point(0, 0),
  Point(1, 0), Point(1.1, 0.5)]
```

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

```bash
python homework3_polygon_navigation_gui.py scene_path min_x min_y max_x max_y
```

The first argument `scene_path` is a path to a scene file storing the layout of a set of obstacles. The remaining group of arguments is optional, and denotes the boundary of the viewing region. If omitted, these arguments’ default values are `min_x = 0, min_y = 0, max_x = 1, and max_y = 1.`

## 3 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?