Homework 3

In this assignment, you will implement three basic search algorithms discussed in the lecture and apply them to solve three classic puzzles. Two start-up files are provided on Canvas:

- `hw3_utils.py`: containing the class `Node` that represents a search node and the class `Problem` that abstracts the problem-specific operations for the search algorithms. It also contains other utilities.
- `hw3.py`: containing the prototypes of three search algorithms and the empty definitions of the classes for the three classic puzzles.

You must:

- Download the two start-up files and place them in the same directory.
- Rename `hw3.py` by replacing `hw3` with your PSU access ID. For example, if your PSU email address is suk1234@psu.edu, the you must rename it to `suk1234.py`.
- Not change the file name of `hw3_utils.py` nor modify the contents.
- Upload your complete `suk1234.py` to the correct assignment area on Canvas by 11:59pm on the due date. If you upload your file to the wrong assignment area or if you fail to submit it by 11:59pm on the due date, it will not be graded and you will receive an automatic Zero on the assignment.

Note that you are not allowed to change anything in `hw3_utils.py`. Further, you are not to submit `hw3_utils.py` when you submit your complete homework. Even if you submit it, we will replace it with the original version initially distributed. This means that, if your program depends on the changes you make in `hw3_utils.py`, it will very likely fail when it is graded.
In the description below, many examples of use cases are provided for each function or method. These examples are simply the typical use cases to clarify the specification and is not meant to be a comprehensive test cases. You are strongly encouraged to test your code with these examples and to test further with your own test cases before you submit.

1 Uniform-Cost, Best-First, A-Star Search Algorithms.

Implement `best_first_search`, `uniform_cost_search`, and `a_star_search`. In each of the algorithms, we will pick the first node from the search frontier `F` to expand in each iteration. To make the first node in `F` the best node, we must add the extended nodes to `F` in **sorted order** according to the following evaluation function $f(n)$:

$$f(n) = g(n) + h(n)$$

where, $g(n)$ and $h(n)$ are the path cost from the start state to the given state n and the heuristic value at the given state n, respectively. Note that the three algorithms differ in the choice of $f(n)$:

Algorithm	Evaluation function	Assumption
Best-First Search	$f(n) = h(n)$	$g(n) = 0$ for every state
Uniform-Cost Search	$f(n) = g(n)$	$h(n) = 0$ for every state
A-Star Search	$f(n) = g(n) + h(n)$	None

Also note that Best-First Search is any-path algorithm using visted list, while Uniform-Cost Search and A-Star Search are optimal-path algorithm using extended list.

You should implement these algorithms using the interfaces of the classes `Node` and `problem`. For the details of the methods of these two classes, see `hw3_utils.py`. If you have completed implementing the three classic puzzle problems as described in the later sections, you can test your implementation of the algorithms as follows.

```
>>> q = NQueensProblem(8)
>>> best_first_search(q).solution()
[7, 1, 3, 0, 6, 4, 2, 5]
>>> uniform_cost_search(q).solution()
[0, 4, 7, 5, 2, 6, 1, 3]
>>> a_star_search(q).solution()
[7, 1, 3, 0, 6, 4, 2, 5]

>>> romania_map = Graph(romania_roads, False)
>>> romania_map.locations = romania_city_positions
>>> g = GraphProblem('Arad', 'Bucharest', romania_map)
>>> best_first_search(g).solution()
['Sibiu', 'Fagaras', 'Bucharest']
>>> uniform_cost_search(g).solution()
['Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest']
>>> a_star_search(g).solution()
['Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest']

>>> e = EightPuzzle((3, 4, 1, 7, 6, 0, 2, 8, 5))
>>> best_first_search(e).solution()
['LEFT', 'UP', 'RIGHT', 'DOWN', 'DOWN', 'LEFT', 'LEFT',
 'UP', 'RIGHT', 'RIGHT', 'UP', 'LEFT', 'DOWN', 'DOWN',
 'RIGHT', 'UP', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'LEFT',
 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'UP', 'LEFT',
 'LEFT', 'DOWN', 'RIGHT', 'UP', 'LEFT', 'DOWN', 'RIGHT',
 'RIGHT', 'DOWN', 'LEFT', 'LEFT', 'UP', 'UP', 'RIGHT',
 'DOWN', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT']
>>> a_star_search(e).solution()
['DOWN', 'LEFT', 'LEFT', 'UP', 'UP', 'RIGHT', 'RIGHT',
 'DOWN', 'LEFT', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'DOWN',
 'RIGHT', 'UP', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'DOWN']

>>> map = Graph(best_graph_edges, True)
>>> map.heuristics = best_graph_h
>>> g = GraphProblem('S', 'G', map)
>>> best_first_search(g).solution()
['B', 'G']

>>> map = Graph(uniform_graph_edges, True)
>>> g = GraphProblem('S', 'G', map)
>>> uniform_cost_search(g).solution()
['A', 'D', 'G']

>>> map = Graph(a_star_graph_edges, True)
>>> map.heuristics = a_star_graph_admissible_h
>>> g = GraphProblem('S', 'G', map)
>>> a_star_search(g).solution()
['B', 'C', 'G']
>>> map.heuristics = a_star_graph_consistant_h
>>> g = GraphProblem('S', 'G', map)
>>> a_star_search(g).solution()
['A', 'C', 'G']
```

Homework 3

2 N-Queens Problem

Implement the class `NQueensProblem`. To make `NQueesnProblem` work with the three search algorithms, two new methods are added to its super class `Problem`. If you have successfully implemented `NQueensProblem` in Homework 2, you may copy your `NQueensProblem` class from Homework 2 and add the implementation of two new methods as described below. If not, see the `NQueensProblem` implementation details described in Homework 2 and complete it before you attempt this part.

1. `g(self, cost, from_state, action, to_state)` returns the cost of the path from `init_state` to `to_state` via `from_state`. The path cost from `init_state` to `from_state` is given as `cost`. Executing `action` in `from_state` will lead you to `to_state`. Assume that each action in `NQueensProblem` costs 1.

```
>>> eight_queens = NQueensProblem(8)
>>> eight_queens.g(0, (-1,-1,-1,-1,-1,-1,-1,-1), 7, (7,-1,-1,-1,-1,-1,-1,-1))
1
>>> eight_queens.g(1, (7,-1,-1,-1,-1,-1,-1,-1), 1, (7,1,-1,-1,-1,-1,-1,-1))
2
>>> eight_queens.g(2, (7,1,-1,-1,-1,-1,-1,-1), 3, (7,1,3,-1,-1,-1,-1,-1))
3
>>> eight_queens.g(3, (7,1,3,-1,-1,-1,-1,-1), 0, (7,1,3,0,-1,-1,-1,-1))
4
>>> eight_queens.g(4, (7,1,3,0,-1,-1,-1,-1), 6, (7,1,3,0,6,-1,-1,-1))
5
>>> eight_queens.g(5, (7,1,3,0,6,-1,-1,-1), 4, (7,1,3,0,6,4,-1,-1))
6
>>> eight_queens.g(6, (7,1,3,0,6,4,-1,-1), 2, (7,1,3,0,6,4,2,-1))
7
>>> eight_queens.g(7, (7,1,3,0,6,4,2,-1), 5, (7,1,3,0,6,4,2,5))
8
```

2. `h(self, state)` returns the heuristic value at `state`. We will use the total number of conflicts present in `state` as the heuristic value at that `state`. For example, consider the state `(7,1,3,0,-1,-1,-1,-1)`. We can interprete this state as the eight queens being placed at `(7,0)`, `(1,1)`, `(3,2)`, `(0,3)`, `(-1,4)`, `(-1,5)`, `(-1,6)`, `(-1,7)` on the board and count the number of conflicts assuming that `-1` is a legitimate row number, i.e., the row above the row `0`. Then, there are 16 conflicts in the state as follows:

Locations	(7,0)	(1,1)	(3,2)	(0,3)	(-1,4)	(-1,5)	(-1,6)	(-1,7)
Conflicts			(-1,6)	(-1,4)	(0,3) (-1,5) (-1,6) (-1,7)	(-1,4) (-1,6) (-1,7)	(3,2) (-1,4) (-1,5) (-1,7)	(-1,4) (-1,5) (-1,6)

Note that we double count every conflict for simplicity.

```
>>> eight_queens = NQueensProblem(8)
>>> eight_queens.h((-1,-1,-1,-1,-1,-1,-1,-1))
56
>>> eight_queens.h((7,-1,-1,-1,-1,-1,-1,-1))
42
>>> eight_queens.h((7,1,-1,-1,-1,-1,-1,-1))
```

```
32
>>> eight_queens.h((7,1,3,-1,-1,-1,-1,-1))
24
>>> eight_queens.h((7,1,3,0,-1,-1,-1,-1))
16
>>> eight_queens.h((7,1,3,0,6,-1,-1,-1))
8
>>> eight_queens.h((7,1,3,0,6,4,-1,-1))
4
>>> eight_queens.h((7,1,3,0,6,4,2,-1))
0
>>> eight_queens.h((7,1,3,0,6,4,2,5))
0
```

3 Graph Problem

Implement the class `GraphProblem` so that it will work with the three search algorithms. If you have successfully implemented `GraphProblem` in Homework 2, you may copy your `GraphProblem` class from Homework 2 and add the implementation of two new methods as described below. If not, see the `GraphProblem` implementation details described in Homework 2 and complete it before you attempt this part.

1. `g(self, cost, from_state, action, to_state)` returns the cost of the path from `init_state` to `to_state` via `from_state`. The path cost from `init_state` to `from_state` is given as `cost`. Executing `action` at `from_state` will lead you to `to_state`. Note that the action you can execute in a given state is simply moving to an adjacent state, leading you to that adjacent state. Hence, `action` argument will be the same as `to_state` argument. The cost of `action` is given as the weight (or cost) on the corresponding edge of the graph.

```
>>> romania_map = Graph(romania_roads, False)
>>> romania = GraphProblem('Arad', 'Bucharest', romania_map)
>>> romania.g(0, 'Arad', 'Zerind', 'Zerind')
75
>>> romania.g(0, 'Arad', 'Sibiu', 'Sibiu')
140
>>> romania.g(140, 'Sibiu', 'Rimnicu', 'Rimnicu')
220
>>> romania.g(220, 'Rimnicu', 'Pitesti', 'Pitesti')
317
>>> romania.g(317, 'Pitesti', 'Bucharest', 'Bucharest')
418
```

2. `h(self, state)` returns the heuristic value at `state`. The heuristic value of a state is computed as follows:

```
if an attribute called heuristics exists in the embedded graph then
    /* heuristics must be a dictionary of state : heuristic-value pairs      */
    return the heuristic value associated with the given state;
else if an attribute called locations exists in the embedded graph then
    /* locations must be a dictionary of state : GPS-coordinate pairs        */
    /* GPS coordinate is a tuple (x,y)                                        */
    /* where, x and y are latitude and longitude, respectively               */
    find the GPS coordinate of the given state;
    find the GPS coordinate of the goal state;
    calculate the straight-line distance (or Euclidean norm) between them;
    return the distance;
else
    /* neither heuristics nor locations exists                                */
    /* meaning that no heuristic information is available                    */
    return a large value (i.e., infinity);
end if
```

Note that `__init__` method of `GraphProblem` takes an instance of `Graph` as an argument, i.e.,

```
class GraphProblem(Problem):
   def __init__(self, init_state, goal_state, graph):
   ...
```

Depending on the attributes of `graph`, the heuristic values are computed differently. If `graph` has no relavant attributes, the heuristic value of any state is simply a large value (i.e., python's `math.inf`):

```
>>> romania_map = Graph(romania_roads, False)
>>> romania = GraphProblem('Arad', 'Bucharest', romania_map)

>>> romania.h('Arad')
inf
>>> romania.h('Sibiu')
inf
>>> romania.h('Fagaras')
inf
>>> romania.h('Pitesti')
inf
>>> romania.h('Rimnicu')
inf
>>> romania.h('Bucharest')
inf
```

If `graph` has an attribute called `locations`, the straight line distance from the given state to the goal state is used as the heuristic value of the given state:

```
>>> romania_map = Graph(romania_roads, False)
>>> romania_map.locations = romania_city_positions

>>> romania = GraphProblem('Arad', 'Bucharest', romania_map)
>>> romania.h('Arad')
350.2941620980858
```

```
>>> romania.h('Sibiu')
232.69937687926884
>>> romania.h('Fagaras')
154.62535367784935
>>> romania.h('Pitesti')
89.89438247187641
>>> romania.h('Rimnicu')
186.48860555004424
>>> romania.h('Bucharest')
0.0
```

If `graph` has an attribute called `heuristics`, which is a dictionary of state : heuristic value pairs, we will use the heuristic value associated to the given state in the dictionary:

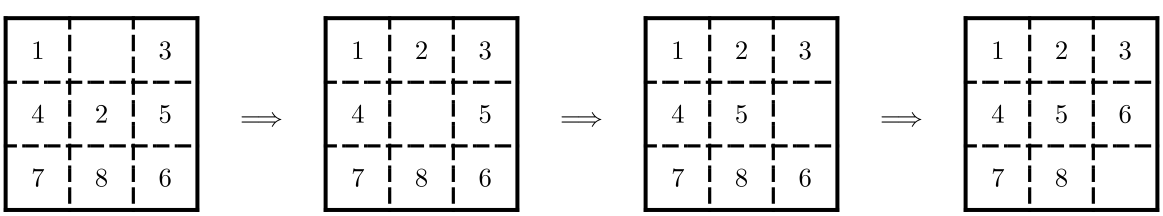
```
>>> roads = dict(S = dict(A=1, B=2), A = dict(C=1),
                 B = dict(C=2), C = dict(G=100))
>>> roads_h = dict(S=90, A=100, B=88, C=100, G=0)
>>> roads_map = Graph(roads, True)
>>> roads_map.heuristics = roads_h

>>> problem = GraphProblem('S', 'G', roads_map)
>>> problem.h('S')
90
>>> problem.h('A')
100
>>> problem.h('B')
88
>>> problem.h('C')
100
>>> problem.h('G')
0
```

Note that the algorithm above prefers `heuristcs` to `locations`, when both attributes exist in `graph` argument.

4 Eight Puzzle

The 8-puzzle consists of a 3×3 grid with eight square tiles labeled 1 through 8 and one blank space. The object of the puzzle is to reach a goal state by rearranging the tiles so that the numbers on the tiles are in order from left to right and top to bottom. You are only allowed to slide tiles horizontally or vertically into the blank space. For example, the following shows a sequence of legal actions from the initial state to the goal state.



Initial Goal

A state of 8-puzzle is represented as a tuple of 8 numbers on the tiles from left to right and top to bottom. For example, the initial state in the figure above is represented as `(1,0,3,4,2,5,7,8,6)` and the final state as `(1,2,3,4,5,6,7,8,0)`, using `0` for the blank space. There are at most 4 possible actions that can be taken in any state of the puzzle. We will use the following keys to represent these actions:

- `'UP'`: Slide the tile above the blank space to the blank space
- `'DOWN'`: Slide the tile below the blank space to the blank space
- `'LEFT'`: Slide the tile on the left of the blank space to the blank space
- `'RIGHT'`: Slide the tile on the right of the blank space to the blank space

The figure above shows the result of applying a sequence of actions `['DOWN', 'RIGHT', 'DOWN']` to solve the puzzle with the given initial state. Note that we will use your `EightPuzzle` to test heuristic search algorithms only, i.e., `best_first_search` and `a_star_search`.
Complete the implementation of the class `EightPuzzle`, a solver for the 8-puzzle problem.

1. `__init__(self, init_state, goal_state)` should simply initialize the parent portion of the instance by calling the parent's `__init__` method with `init_state` and `goal_state` as arguments.

```
>>> puzzle = EightPuzzle((1,0,6,8,7,5,4,2,3),(0,1,2,3,4,5,6,7,8))
>>> puzzle.init_state
(1, 0, 6, 8, 7, 5, 4, 2, 3)
>>> puzzle.goal_state
(0, 1, 2, 3, 4, 5, 6, 7, 8)

>>> puzzle = EightPuzzle((1,0,3,4,2,5,7,8,6))
>>> puzzle.init_state
(1, 0, 3, 4, 2, 5, 7, 8, 6)
>>> puzzle.goal_state
(1, 2, 3, 4, 5, 6, 7, 8, 0)
```

2. `actions(self, state)` returns a list of the valid actions that can be executed in `state`. Note that, if the blank space is on an edge of the 3×3 grid, some actions become invalid. For example, if the blank space is at the bottom right corner of the grid, the actions `Right` and `DOWN` are invalid and should be excluded.

```
>>> puzzle = EightPuzzle((1,0,3,4,2,5,7,8,6))
>>> puzzle.actions((0,1,2,3,4,5,6,7,8))
['DOWN', 'RIGHT']
>>> puzzle.actions((6,3,5,1,8,4,2,0,7))
['UP', 'LEFT', 'RIGHT']
>>> puzzle.actions((4,8,1,6,0,2,3,5,7))
['UP', 'DOWN', 'LEFT', 'RIGHT']
>>> puzzle.actions((1,0,6,8,7,5,4,2,3))
['DOWN', 'LEFT', 'RIGHT']
>>> puzzle.actions((1,2,3,4,5,6,7,8,0))
['UP', 'LEFT']
```

3. `result(self, state, action)` returns a new state that results from executing `action` in `state`.

```
>>> puzzle = EightPuzzle((1,0,3,4,2,5,7,8,6))
>>> puzzle.result((0,1,2,3,4,5,6,7,8), 'DOWN')
(3, 1, 2, 0, 4, 5, 6, 7, 8)
>>> puzzle.result((6,3,5,1,8,4,2,0,7), 'LEFT')
(6, 3, 5, 1, 8, 4, 0, 2, 7)
>>> puzzle.result((3,4,1,7,6,0,2,8,5), 'UP')
(3, 4, 0, 7, 6, 1, 2, 8, 5)
>>> puzzle.result((1,8,4,7,2,6,3,0,5), 'RIGHT')
(1, 8, 4, 7, 2, 6, 3, 5, 0)
```

4. `goal_test(self, state)` returns `True` if `state` is the goal state. Returns `False` otherwise.

```
>>> puzzle = EightPuzzle((1,0,6,8,7,5,4,2,3),(0,1,2,3,4,5,6,7,8))
>>> puzzle.goal_test((6,3,5,1,8,4,2,0,7))
False
>>> puzzle.goal_test((1,2,3,4,5,6,7,8,0))
False
>>> puzzle.goal_test((0,1,2,3,4,5,6,7,8))
True

>>> puzzle = EightPuzzle((1,0,3,4,2,5,7,8,6))
>>> puzzle.goal_test((6,3,5,1,8,4,2,0,7))
False
>>> puzzle.goal_test((0,1,2,3,4,5,6,7,8))
False
>>> puzzle.goal_test((1,2,3,4,5,6,7,8,0))
True
```

5. `g(self, cost, from_state, action, to_state)` returns the cost of the path from `init_state` to `to_state` via `from_state`. The path cost from `init_state` to `from_state` is given as `cost`. Executing `action` at `from_state` will lead you to `to_state`. Assume that each action in `EightPuzzle` costs 1.

```
>>> puzzle = EightPuzzle((1,0,3,4,2,5,7,8,6))
>>> puzzle.g(0, (4,8,1,6,0,2,3,5,7), 'UP', (4,0,1,6,8,2,3,5,7))
1
>>> puzzle.g(3, (8,0,1,4,6,2,3,5,7), 'DOWN', (8,6,1,4,0,2,3,5,7))
4
>>> puzzle.g(8, (8,1,2,4,5,6,3,7,0), 'UP', (8,1,2,4,5,0,3,7,6))
9
>>> puzzle.g(11, (1,2,8,4,5,6,3,0,7), 'RIGHT', (1,2,8,4,5,6,3,7,0))
12
```

6. `h(self, state)` returns the heuristic value at `state`. The heuristic value of `state` is the sum of the Manhattan distance of misplaced tiles to their final positions. For example, given a state $(4, 1, 2, 3, 0, 6, 5, 7, 8)$ and the goal state of $(1, 2, 3, 4, 5, 6, 7, 8, 0)$ as shown in the figure below:

4	1	2
3		6
5	7	8

Given state

1	2	3
4	5	6
7	8	

Goal state

the Manhattan distance of each tile to its final position is:

Tile number	4	1	2	3	6	5	7	8
Manhattan Distance	1	1	1	3	0	2	1	1

Hence, the heuristic value of the state $(4,1,2,3,0,6,5,7,8)$ to the goal state is 10.

```
>>> puzzle = EightPuzzle((1,0,3,4,2,5,7,8,6))
>>> puzzle.goal_state
(1, 2, 3, 4, 5, 6, 7, 8, 0)
>>> puzzle.h((1,2,3,4,5,0,7,8,6))
1
>>> puzzle.h((1,2,0,4,5,3,7,8,6))
2
>>> puzzle.h((1,0,2,4,5,3,7,8,6))
3
>>> puzzle.h((4,1,2,0,5,3,7,8,6))
5
>>> puzzle.h((4,1,2,6,8,0,3,5,7))
13
```