Homework 2

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

- `hw2_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.
- `hw2.py`: containing the prototypes of two 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 `hw2.py` by replacing `hw2` 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 `hw2_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 `hw2_utils.py`. Further, you are not to submit `hw2_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 `hw2_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 Uninformed Any-Path Search Algorithms.

Implement `depth_first_search` and `breadth_first_search`. 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 `hw2_utils.py`. If you have completed implementing one of the puzzle problems, e.g., `NQueensProblem`, you can test your implementation of these algorithms as follows.

```
>>> q = NQueensProblem(1)
>>> d = depth_first_search(q); d.solution()
[0]
>>> b = breadth_first_search(q); b.solution()
[0]
>>> d = depth_first_search(NQueensProblem(2)); print(d.solution())
None
>>> b = breadth_first_search(NQueensProblem(2)); print(b.solution())
None
>>> depth_first_search(NQueensProblem(4)).solution()
[2, 0, 3, 1]
>>> breadth_first_search(NQueensProblem(4)).solution()
[1, 3, 0, 2]
>>> depth_first_search(NQueensProblem(8)).solution()
[7, 3, 0, 2, 5, 1, 6, 4]
>>> breadth_first_search(NQueensProblem(8)).solution()
[0, 4, 7, 5, 2, 6, 1, 3]
```

2 N-Queens Problem

The N-queens problem is the problem of placing N queens on an $N \times N$ chess board so that no queen attacks another. A queen attacks any piece in the same row, column, or diagonal. Rather than searching for N squares that are not conflicted from $N \times N$ squares on the board to place N queens, it is sufficient to consider only those configurations for which each row contains exactly one queen. A state of the problem, therefore, is represented by an N-tuple, $(Q_0, Q_1, \cdots, Q_{n-1})$, where Q_i represents the position (i.e., the row number) of a queen in the column i. For example, for the 4-Queens problem, the state $(2, 0, 3, 1)$ represents the follwing board configuration

	Q_1		
			Q_3
Q_0			
		Q_2	

Note that the value of -1 in a state means that a queen is not yet place in the corresponding column. Hence, the initial state of the 4-Queens problem is $(-1, -1, -1, -1)$. From the initial state, we will fill in the columns from left to right. Complete the implementation of the class `NQueensProblem,` a solver for the n-queens problem.

1. `__init__(self, n)` should first initialize the parent portion of the instance by calling the parent's `__init__` method with the proper initial state (i.e., n-tuple of -1's). Then, it should initialize its own instance variables as needed.

```
>>> eight_queens = NQueensProblem(8)
>>> eight_queens.init_state
(-1, -1, -1, -1, -1, -1, -1, -1)
>>> eight_queens.n
8
```

2. `actions(self, state)` should first locate the leftmost column to be filled, find all the valid rows on that column that do not attack any queens already on the board, and return the list of the valid row numbers on that column.

```
>>> eight_queens.actions((-1, -1, -1, -1, -1, -1, -1, -1))
[0, 1, 2, 3, 4, 5, 6, 7]
>>> eight_queens.actions((7, 2, -1, -1, -1, -1, -1, -1))
[0, 4, 6]
>>> eight_queens.actions((7, 2, 6, 3, 1, 5, -1, -1))
[]
```

3. `result(self, state, action)` returns a new state that results from executing the given action on the leftmost empty column in the given state.

```
>>> eight_queens.result((-1, -1, -1, -1, -1, -1, -1, -1), 7)
(7, -1, -1, -1, -1, -1, -1, -1)
>>> eight_queens.result((7, 2, -1, -1, -1, -1, -1, -1), 6)
(7, 2, 6, -1, -1, -1, -1, -1)
```

4. `goal_test(self, state)` returns `True` if all N queens are placed on all columns (i.e., one queen on each column) such that no queen attacks another queen. Returns `False` otherwise.

```
>>> eight_queens.goal_test((7, 3, 0, 2, 5, 1, 6, 4))
True
>>> eight_queens.goal_test((0, 4, 7, 5, 2, 6, 1, 3))
True
>>> eight_queens.goal_test((7, 2, 6, 3, 1, 4, 0, 5))
False
>>> eight_queens.goal_test((1, 4, 6, 3, 0, 2, 5, 7))
False
```

3 Farmer's Problem

Consider the following problem:

> A farmer must move a bag of grain, a chicken, and a fox from left bank of the river to the right by boat. The boat can hold only the farmer and one other object at a time. Apparently, only the farmer can row the boat. If the chicken is left unattended on a bank with the grain, the chicken will eat the grain. The fox will also eat the chicken if they are left unattended. Help the farmer find his way to move everything from left bank of the river to the right without losing the bag of grain or the chicken.

A state of the problem is represented by a 4-tuple of booleans, i.e., (b_{farmer}, b_{grain}, b_{chicken}, b_{fox}), where the value of `True` means the corresponding entity is on the left bank and `False` on the right bank of the river. For instance, the state (`False`, `True`, `True`, `False`) means that the bag of grain and the chicken are on the left bank while the farmer and the fox are on the right. Unfortunately, this is an invalid state since the bag of grain and the chicken are left unattended together and so the chicken will consume the grain. With this representation, the initial state and the final state of the problem are (`True`, `True`, `True`, `True`) and (`False`, `False`, `False`, `False`), respectively.
Note that there are at most 4 possible actions that can be taken in any state of the problem. We will use the following keys to represent these actions:

- `'F'` : Farmer crosses the river alone
- `'FG'`: Farmer crosses the river carrying the bag of grain
- `'FC'`: Farmer crosses the river carrying the chicken
- `'FX'`: Farmer crosses the river carrying the fox

Complete the implementation of the class `FarmersProblem`, a solver for the farmer's problem.

1. `__init__(self, init_state, goal_state)` should first initinalize the parent portion of the instance by calling the parent's `__init__` method with proper arguments. Then, it should initialize its own instance variables as needed.

```
>>> farmer = FarmerProblem((True, True, True, True), (False,False,False, False))
>>> farmer.init_state
(True, True, True, True)
>>> farmer.goal_state
(False, False, False, False)
```

2. `actions(self, state)` returns the list of valid actions that can be executed from the given state.

```
>>> farmer.actions((True, True, True, True))
['FC']
>>> farmer.actions((False, False, True, False))
['F', 'FG', 'FX']
>>> farmer.actions((False, True, False, True))
['F', 'FC']
```

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

```
>>> farmer.result((True, True, True, True), 'FC')
(False, True, False, True)
>>> farmer.result((False, False, True, False), 'FX')
(True, False, True, True)
>>> farmer.result((False, True, False, True), 'F')
(True, True, False, True)
```

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

```
>>> farmer.goal_test((True, True, True, True))
False
>>> farmer.goal_test((False, False, True, False))
False
>>> farmer.goal_test((False, False, False, False))
True
```

4 Graph Problem

Given a graph representation of a road map, we would like to find a path from a city to another. Unfortunately, the road map given (`romania_roads`, Figure 3.1, p.64) is so crude and unreliable that we will rely only on the roads connecting cities and will ignore any and all other information on the map. Thus, we will be happy to find any path from the city we are in to the desitnation.

Complete the implmentation of the class `GraphProblem`, a solver for the route planner.

1. `__init__(self, init_state, goal_state, graph)` should first initinalize the parent portion of the instance by calling the parent's `__init__` method with proper arguments. Then, it should initialize its own instance variables as needed.

```
>>> romania_map = Graph(romania_roads, False)
>>> planner = GraphProblem('Arad', 'Bucharest', romania_map)
>>> planner.init_state
'Arad'
>>> planner.goal_state
'Bucharest'
>>> planner.graph
<Graph {'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118}, ..... ,
        'Vaslui': 'Iasi': 92, 'Urziceni': 142, 'Neamt': {'Iasi': 87}}>
```

2. `actions(self, state)` returns the list of adjacent cities from the given city (i.e., `state`)

```
>>> planner.actions('Arad')
['Zerind', 'Sibiu', 'Timisoara']
>>> planner.actions('Sibiu')
['Arad', 'Fagaras', 'Oradea', 'Rimnicu']
>>> planner.actions('Eforie')
['Hirsova']
```

3. `result(self, state, action)` returns a new state that results from executing the given action in the given state. Note that `state` is a city we are currently in and `action` is one of the adjacent city to move to from the currently city.

```
>>> planner.result('Arad', 'Zerind')
'Zerind'
>>> planner.result('Sibiu', 'Fagaras')
'Fagaras'
>>> planner.result('Eforie', 'Hirsova')
'Hirsova'
```

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

```
>>> planner.goal_test('Arad')
False
>>> planner.goal_test('Timisoara')
False
>>> planner.goal_test('Bucharest')
True
```