Problem Set 2: Due February 13

Consider the classic child's game called the *8-puzzle* which is a 3x3 grid with 8 tiles (numbered 1 through 8) and an empty space. By sliding adjacent tiles into the empty space, new configurations of the puzzle can be formed. The puzzle is considered solved if the tiles are in numerical order from left to right and top to bottom.

The goal of this assignment is to firm up your understanding of how search can be used for general problem solving.

Begin by downloading the skeletal code `EightPuzzle.zip`. The code is incomplete, but provides a basic framework for implementing search in the Russell & Norvig style. You will find two files: `search.py` and `nn_puzzle.py`.

`search.py` contains base/abstract classes and is built on the pseudo-code modeled in Russell and Norvig chapter 3. While `nn_puzzle.py` contains the problem-specific logic to encode the 8-puzzle and all $N^2 - 1$-puzzle variants as search problems. You should consider these read-only files, as we will use our own copy when testing your code.

`my_search_stub.py` contains some stub code to implement the abstract 'Search' class. You'll need to implement the given methods and may need to add additional methods to support your search. You should do all of your implementation in the file `my_search_stub.py`

You may find that you want to mess around with the `__main__` clause for testing.

As always, your code will be graded based on correctness and elegance; concise programs that are readable and documented for clarity will receive the highest scores.

If you have a strong hankering to implement this assignment in a language other than python, please talk to me before you get going.

1. (40 points) Implement Breadth First Search (BFS) using a class `BFS(Search)`. This will require you to manage an open list, generate nodes for the search tree and test for goal conditions. For BFS, it is recommended that you also maintain a closed list, although this is not required. Make sure, however, that your open list is ordered correctly for breadth first search. Your `__init__()` method should perform object instance initialization as follows:

```
def __init__(self, problem):
    Search.__init__(self, problem)
```

   This has the effect of calling the super-class's constructor. Note that while this is implicity done for you in Java, in Python you must be explicit.

   Your class will have a `search` method as is shown in the code provided. Add a brief description to the method's doc-string that describe how your search-tree nodes are encoded so this is clear during grading.

   An appropriate and well documented search-tree node is worth 15 points.

2. (45 points) Implement A* Search using a class `AStar(Search)`.

   This will require you to define an admissible heuristic for the NNPuzzle and to order the open list based on the sum of path cost and heuristic value for each node in the search tree.

   As with BFS, document your encoding of search-tree nodes in the doc-comment of the `search()` method. Note that you will probably want to encode g, h, f in the search state to easily compute these values for successor nodes and to easily order the queue. For this search method, you should be sure to maintain a closed list in addition to the open list.

   Finally, note that somewhere, you will need to calculate the heurstic value $h(n)$ during the search. You can encode the heurstic function in your AStar class eventhough this is somewhat unsatisfiying in that it adds domain specific logic to what is otherwise a general search process.

3. (10 points) Print the search path A* generates to solve the problem with the configuration: `(1,0,8,5,2,4,7,3,6)`. You should put this path (as a series of states like the one above) in the doc comment at the start of your `my_search_stub.py` file.

4. (5 points) Russell and Norvig's UniformCostSearch algorithm on page 84 suggests that when a seach node is a being examined, if another node with the same state is found on the frontier (open-list) but with a higher cost, the new search node should replace it. Assume that I use a heap as a priority queue to order nodes. What is the runtime of performing this check? Put the answer to this question in a doc-comment block at the top of the code you submit.

Turn in:

- Change the name of `my_search_stub.py` to include your last name (e.g., `eightpuzzle_wallace.py`).
- Send me your `.py` file by email before 11:59pm on the due date.