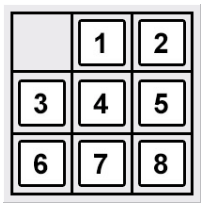
Smart Search: Solving the 8-puzzle

Write a Python program to find solve the 8-puzzle. A solution involves arriving at the final GOAL state given any (random shuffled) set of tiles where the GOAL state is defined as:

```
-------------
| 0 | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
```

```
# possible data structure in Python for GOAL
# the tile zero(0) indicates the OPEN slot
GOAL  = [ [0, 1, 2], [3, 4, 5], [6, 7, 8] ]
```



Input [optional] (`stdin`, keyboard):

A user enters a starting board configuration (or for experiments, a random board is used)

Output (`stdout`, console):

A final path of tile moves from the initial board to the GOAL board. The final path output *must* reflect a realistic set of tile moves (that is, your output can not print boards of your solutions that jump around the search tree; you *can* of course while debugging, etc. print the boards that you *expand*). The final printout must be an accurate set of tile moves. (Note: you will undoubtedly want to dump more output to the console during debugging; just make sure *on submission* that your console output is *only* a final solution path.)

```
>>> [evaluate EightPuzzle_Main.py]

SEARCH ALGORITHM: Best-First     HEURISTIC: Manhatten Distance = (sum of misplaced tiles)
DONE!!

8-Puzzle solved in 4 moves.

STARTING BOARD
-------------
| 3 | 1 | 2 |
-------------
| 4 | 7 | 5 |
-------------
| 6 | 8 | 0 |
-------------

Move:  1
-------------
| 3 | 1 | 2 |
-------------
| 4 | 7 | 5 |
-------------
| 6 | 0 | 8 |
-------------

Move:  2
-------------
| 3 | 1 | 2 |
-------------
| 4 | 0 | 5 |
-------------
| 6 | 7 | 8 |
-------------

Move:  3
-------------
| 3 | 1 | 2 |
-------------
| 0 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------

Move:  4
-------------
| 0 | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------

YES!
```

Output (`.csv` Excel-ready file):

Your program, while solving the problem, should dump statistical information to a comma-separated value (.csv) output file. Your stats should include the current Board# tried (that is, the latest board popped off (*expanded*) from your Queue of boards to try), as well as the sizes of your Queue of Boards and Expanded List of Boards. Timing information would also be valuable. A sample output Table of results is shown on the next page.

Board Number Tried	Size of Queue	Size of Expanded List
0	1	1
1	2	3
2	3	5
3	5	8
4	6	10

Table 1. Runtime information when solving a trivial 8-puzzle using Best-First Search and the Manhatten Distance heuristic for determining distance from the GOAL state.

Algorithm	Average Max Size of Queue	Average Runtime (sec)
Best-First: h(n) = Manhatten		
Depth-First Search		

Table 2. Comparison of two algorithms for 1000 trials of randomly generated (valid) starting 8-puzzle boards.

Algorithm

Implement the Best-First Search strategy discussed in class. You might think of this as the A* algorithm, however, note that all edges (moves) have the same cost (e.g., +1). Your program should use two different heuristics for determining the distance between a given board and the GOAL board: (i) the number of tiles out of place and (ii) the Manhatten Distance = [the sum of tile misplacements] (as discussed in class).

You *must* implement (at least) `class BoardClass(object)`. You should also implement at least one other `.py` file as a test-driver. This class should override the relational operators so that a list of BoardClass objects can be sorted according to the current heuristic used. (Note: you must implement different versions of these relational operators once you change to the alternate heuristic; you may want to make two different versions of your app, e.g., v2.0 uses Manhatten Distance.)

Experiment(s)

So, your program solves the 8-puzzle … (*yawn*) … um, I mean "*I am soooo proud of you*!" Now the important (empirical) work begins. Run a suite of starting game boards (randomly generated) and keep statistics. (Note: plan carefully! Many of these runs may take a long time?) Your program should produce results that will appear in a Final Report (a professional looking **.pdf** file) that includes the sections: (a) Summary of the program/task and a concluding paragraph that mentions what else will be found in the report (a succinct summary is important; it should be a high-level description; it should *not* for instance start talking about user input), (b) Methods: description and number of input set(s); summary of Algorithm(s) used; special cases? (c) Sample Output: a sample solution path (obviously, a small run); (d) Results: the total runtime for finding each solution in your input set(s); Table(s) and Figure(s)/plots of your results. Remember: your Final Report *is* the most significant part of your submission!

Submission: .zip of a folder (named with YourLastName.zip *that produces upon unzip* a folder with YourLastName) containing `_README.txt`, *documented* source code (use Python's triple-quote `"""`class and method documentation), input sets, sample output, and Final Report (.pdf). Please submit a hardcopy of your Final Report.

Grading

Average (C): you get the algorithm working on at least some algorithm; show *correct* results, success on various trials; A final report is submitted.

Above Average (B): you use and document Python objects and run experiment(s); you trap bad boards; you time experimental runs; your report includes Tables and plots of runtime(s).

Superior (A): **[I am open to suggestions, but assuming you have finished the above]** you implement a competing algorithm, e.g., one of the brute force algorithms (e.g., DFS) and/or you implement the 15-puzzle. Your report compares the differences, of course, averaged over many input sets.