THE COLLEGE OF SAINT ROSE
CSC 535 ARTIFICIAL INTELLIGENCE
PROJECT ONE

REQUIREMENTS:
You will be implementing an A* search with the Manhattan distance heuristic to solve an 8-block ("sliding tile") puzzle.

The Manhattan distance heuristic function, *h*, can be defined as follows:
Let $moves(n)$= minimum number of moves required to put tile *n* it its goal position

Therefore, the Manhattan distance function, *h*, for *T* tiles is:

$$\sum_{i=1}^{T} moves(i)$$

For example, a calculation of *h* for the starting configuration (almost at goal) would be:
1 + 1 + 0 + 0 + 1 + 1 + 0 + 0 = 4 [See text page 103 and lecture slides for another examples]

Your program must use standard input to allow the user to input the *initial* state of the puzzle (read left to right, top to bottom). For example, the tiles for the puzzle below would be input as follows:

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

… where 0 indicates the blank tile. Your program must also use standard output to display the series of moves required to solve the puzzle:

For example (your program should simply print these vertically... I just did this to save space):

```
  1 3       1   3       1 2 3       1 2 3       1 2 3
4 2 5       4 2 5       4   5       4 5         4 5 6
7 8 6       7 8 6       7 8 6       7 8 6       7 8
```

We will define a *state* of the game to be **(B_i, M, B_{i-1})**, where:

B_i: the board position at step *i*

M: the number of moves made to reach the board position, and

B_{i-1}: the previous state (NULL if *i=0*)

Hints to get you started:
Since we are using A*, we will utilize a priority queue. Place the initial state of the board in this queue. That is: (B_0, 0, NULL). Now in the priority queue:

repeat until goal state is dequeued
delete the state with the minimum priority from the priority queue
insert all neighboring states (those that can be reached in one move)

➔ To prevent redundant expansion of states, do not place a neighbor of a state into the queue if its board position is identical to the *previous* state.

➔ Not all initial board configurations are solvable! Your program will only be tested on solvable configurations, however, your program should not crash and burn if one is encountered!

ADDITIONAL REQUIREMENTS:
Your program must be robust. For example, your program should not crash and burn if the input is invalid. That is, you should validate that only numbers 0 – 8 are used for positions within the puzzle and that 9 positions are specified (recall that 0 indicates the initial position of the blank space). Reject any invalid input with an appropriate error message and termination.

There should be some elegance to the code you write and must contain some documentation. Your may lose a point if your code looks sloppy.

PROGRAMMING LANGUAGES ALLOWED:
You may use any standard language to write this program (Java, C/C++, Python). If you wish to use something non-standard, please contact me prior to starting.

SUBMISSION:
Please submit all of the source code for your assignment along with your code-walkthrough video.