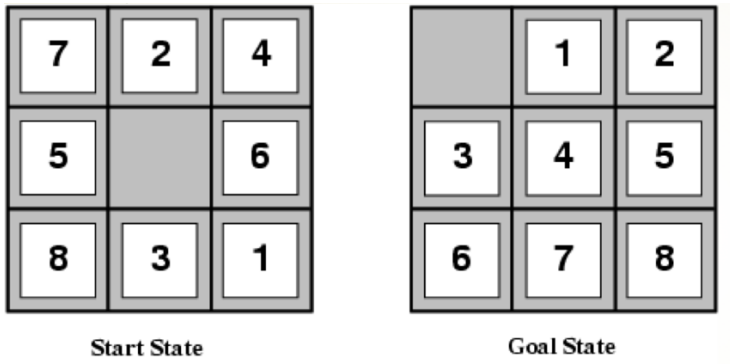
8-Puzzle Problem

Author: Timothy Morren

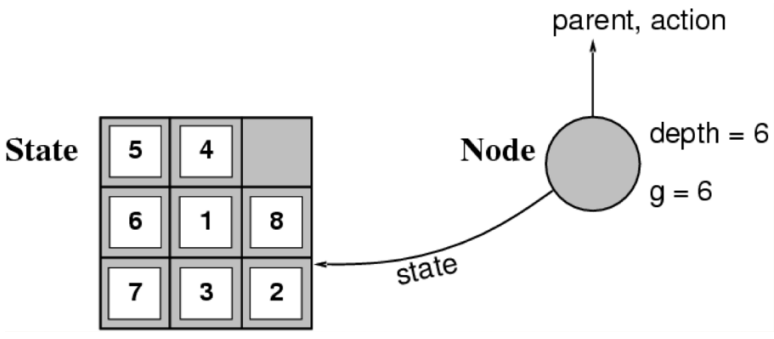
Project Description: Imagine a 3x3 unit board in 2-d space. Insert 8 square tiles -each with side length of 1 unit- onto the board. Notice there is one empty square on the board which can be used to move neighboring adjacent tiles into. The *Start State* of the board is simply the *Goal State* after a set number of random moves. (Figure 1.) Random move: swapping the empty tile with an adjacent nonempty tile at random.



(Figure 1.)

Goal: After a number of random moves are performed on the *Goal State*, we reach a valid *Start State* (i.e. we know there is a solution path since we only took valid moves). Given a valid *Start State* there will be a solution path to *Goal State.* A solution path is all the actions and the resulting states the board took to reach the goal state. There are multiple approaches to finding the solution path of an 8-puzzle problem; the approach described in this paper converts the problem into a graph then applies the a-star informed search algorithm and different heuristics to find a solution path, included in the report is important statistics for each type of search strategy/heuristic and how they compare.

Converting 8-Puzzle Problem into Graph: Convert the current state, starting with the start state, into a node by saving a deep copy of the current state, or a copy of all the states that lead up to the current state, along with its parent node, action, path cost g(x), depth and nodeId (Figure 2.).



(Figure 2.)

Expanding the Graph: To expand the tree, or find new states to compare to the Goal State **until the current node's state is the goal state**, we do the following: First, add the current state of the board to a list of boards we have already visited, preventing reexploration of previously explored states. Next, take the current state and make all valid moves. Then, compare the resulting valid states to the list of already explored states to verify they are a unique board state. If the child state, the state just created by making a valid move from the current/parent state, is unique then create a new node with this board state. Calculate and store the *path cost* to make that move, for now assume the cost to make a move will be 1 for all moves. So, add 1 to the cost of the node, which in this case would represent the number of actions taken to reach the current state since each move adds one to the nodes cost. Lastly choose the child state with the lowest *path cost* and make that the current state/node. Repeat expansion until current node's state equals the goal state.

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
      fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
      s ← a new NODE
      PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
  return successors
```

(Figure 3.)

Path Cost: Step_Cost(node, action, s) in Figure 3. When the path cost is set to always be 1 we will simply search the first child of each state until there are no more "left" children, or children states made by only taking left moves. This results in depth first search and is uninformed since all children node path's cost the same. If we add an intelligent way to update the path cost based on if said move helps us reach the goal faster, we have transformed the algorithm into an informed search strategy. **The intelligent way to update the path cost is called the <u>heuristic</u>**. There are a myriad of different heuristics, some better than other, so next we will look at the heuristics used in this project and compare them.

<u>Heuristics</u>: *Heuristic 1*- displaced_tiles( ) which simply returns the number of tiles that are in the wrong spot. If a move results in having a tile put in its finale goal state spot it will have a lower path cost thus the algorithm will favor the children of that node. *Heuristic II*- manhattan_distance() which returns the total distance each tile is out of place by, thus favoring moves that bring tiles closer to their goal state. *Heuristic III*- manhattan_distance_with_scaled_penalty() which returns the same as Manhattan distance but each distance out of place scales up to cost more, thus adding a

huge cost to making moves that bring tiles farther away from their goal state. This means the algorithm should balance not only bringing titles to their goal state, but also not moving tiles away from their goal state even farther.

Results/Test: We will compare the results from the three different heuristics along with the base case (cost is always 1) using statistics provided by running the different search strategies 100 times on start states that have each been randomly shuffled by making 100 random valid moves as defined above. We will record data points on how many nodes were visited in total (V), the maximum number of nodes stored in memory (closed list + open list size) (N), the depth of the optimal solution (d), and the approximate effective branching factor (b) by using the command:

```
for ((x=0;x<100;x++)); do cat OLA1-input.txt | python3 random_board.py ${x} 100 | python3 a-star.py 0; done | python3 stats.py
```

For base case (cost always 1):

```
V - Minimum: 35, Median: 16918.5, Mean: 53248.42, Maximum: 408076, Standard Deviation: 83436.31866451482
N - Minimum: 28, Median: 9473.0, Mean: 19200.54, Maximum: 73553, Standard Deviation: 22474.45462149721
d - Minimum: 5, Median: 16.5, Mean: 15.78, Maximum: 26, Standard Deviation: 5.304048387121793
b - Minimum: 1.63724, Median: 1.8066, Mean: 1.8225684, Maximum: 2.1361, Standard Deviation: 0.1055092314762413
```

For *Heuristic I (h(1)):*

```
V - Minimum: 11, Median: 2198.0, Mean: 7922, Maximum: 59662, Standard Deviation: 12906.86909746085
N - Minimum: 10, Median: 1369.0, Mean: 4282.34, Maximum: 27186, Standard Deviation: 6370.830767734386
d - Minimum: 5, Median: 16.5, Mean: 16.1, Maximum: 27, Standard Deviation: 5.668448917588756
b - Minimum: 1.46384, Median: 1.590605, Mean: 1.5814307, Maximum: 1.69838, Standard Deviation: 0.053813302817343506
```

For *Heuristic II (h(2)):*

```
V - Minimum: 8, Median: 524.5, Mean: 1597.35, Maximum: 12947, Standard Deviation: 2449.4339167706203
N - Minimum: 8, Median: 340.5, Mean: 985.03, Maximum: 7514, Standard Deviation: 1455.068372547448
d - Minimum: 5, Median: 16.5, Mean: 16.22, Maximum: 28, Standard Deviation: 5.791983313776957
b - Minimum: 1.34851, Median: 1.460005, Mean: 1.460773, Maximum: 1.55479, Standard Deviation: 0.04483858679370442
```

For *Heuristic III h(3):*

```
V - Minimum: 8, Median: 387.0, Mean: 1139.66, Maximum: 7619, Standard Deviation: 1735.1069202347344
N - Minimum: 8, Median: 251.0, Mean: 709.16, Maximum: 4586, Standard Deviation: 1042.3195163391188
d - Minimum: 5, Median: 16.5, Mean: 16.02, Maximum: 30, Standard Deviation: 5.679984350526172
b - Minimum: 1.34642, Median: 1.4407, Mean: 1.443908, Maximum: 1.55769, Standard Deviation: 0.049697540251713745
```

Interpreting results: If we compare the V mean (the average number of visited nodes in 100 iterations) for each heuristic we can see that base case > h(1) > h(2) > h(3) in terms of V. The heuristic with the lower V is the more efficient search algorithm since we needed to traverse less of the state space to find the solution path. This means, h(3) is a better heuristic, on average, for this problem than the rest. Not only is it more time efficient, assuming visiting each node takes the same amount of time, but also space efficient. If we look at the N mean value for h(3) it shows we used less space in memory, on average, for the 100 iterations. H(3) is confirmed to be the best out of the four by looking at the other statistics d and b since both of their mean's are less than the other options discussed.

Implementation details: Efficient implementation of the algorithm comes from using data types that lend to our specific uses. For storing the children nodes not yet explored I used a **priority queue** so that the nodes would be auto sorted by path cost so I could quickly pop the node first off and make that the new current node being expanded. For the closed list I used a set. A set doesn't allow duplicates so it never stored the same explored state twice. The set also allows for very quick .ismember checking since sets function like a hash table which takes O(1) time.

Limitations: The algorithm assumes a valid board state. The current code only works for a 3x3 grid but in theory the algorithm itself could work on any square grids with 1 open spot.

Resources:

https://www.w3schools.com/python/python_sets.asp : goes over python sets and how to add/remove.

https://www.geeksforgeeks.org/take-input-from-stdin-in-python/ : goes over python stdin, used for understanding how to read in from stdin