Background

The 8 puzzle is a puzzle where 8 tiles are moved around to arrange a given number order or pattern. In practice, it may be implemented as a sliding puzzle where there is an empty slot that the others move into. For example, figure one left shows what this puzzle may look like in memory, while figure one right may serve as a familiar example of how this puzzle looks in graphical form. In practice, there are 181,440 possible states that the puzzle can be in, and the optimal solution of the puzzle is in the NP-hard class of problems (1 Phillips, 2022).

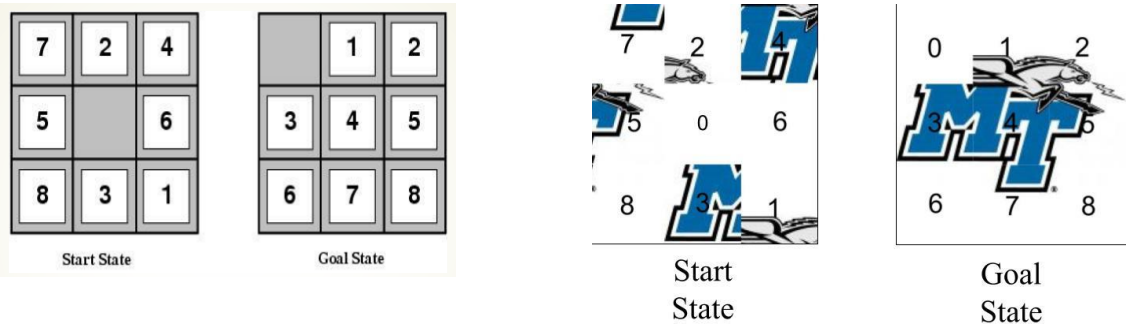


Figure 1 (left) **8 puzzle.** Phillips, Joshua Dr. *Complexity Analysis and Search.* 26. *(right) Picture form of left.* George Gannon.

In open lab one, the task was to create a Python agent that could effectively solve this puzzle given a generated randomized starting state, such as that in figure one. As brute force searching through the number of states that are possible would be an arduous task, the agent uses the A* informed search strategy. It's similar to Dijkstra's algorithm—while searching it explores the lowest cost nodes by means of a min-priority queue and then places explored nodes into a closed list set, but one major distinction is that A* uses a heuristic in its searching. A heuristic is defined as "a rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood" (2 Phillips, 2022). For example, if an algorithm was tasked with finding the shortest path between two cities, it might use the straight-line distance between the current city and the target city as its heuristic. If the heuristic is increasing, it's a likely guess that the current path is veering away from the target city. A* adds a heuristic to the cost of the nodes in the search tree so that when the lowest cost node is picked, said node also is more likely to be closer to the goal. (2 Phillips, 2022) To better understand the performance and impact of different heuristics, the agent has multiple heuristics implemented that can be accessed using command line arguments: no heuristic, number of tiles displaced from the goal, the sum of the Manhattan distances from the goal, and a heuristic of my own design.

Design

Development began with the random_board.py script. It takes two command line args: a random seed and an integer number of moves to make to shuffle the board. First, the script takes a beginning state from standard input. For all the runs, this was the goal state. After seeding the random library using the seed specified, the script then runs through a loop the number of moves

specified long, each time generating numbers within a range of zero to three inclusive. Each number in this range maps to a different move on the board: up, down, left and right, respectively. The set_test.py support file is the basis for the movement operations and state Python object. Once the script has sufficiently shuffled the board, it is printed in the terminal to be sent to the A-star script.

Before beginning development of the a-star.py script, it was deemed wise to place the support objects in one file: datastructures.py. This made it easier to get objects as the code could import datastructures as ds and then fetch whatever it needed e.g. ds.node(…) or ds.HEURISTIC. Four essential datastructures can be found in this file: ds.PriorityQueue() which is used for the frontier, ds.Set() which is used for the closed list, ds.state() which is responsible for acting as the holder of the current tile configuration which is itself a numpy array so that it can use special methods, and finally ds.node() which contains the state object and necessary pointers to parent nodes. The PriorityQueue is a slightly modified version of the one in heapq_test.py [the only difference is the thisNode.totalCost instead of thisNode.val]. The Set object remains identical to the one found in set_test.py. The Node object contains a few extra methods and attributes: there is a parent pointer, an h cost, and a few heuristic methods that determine the totalCost attribute.

As for the a-star.py code, it was straightforward: take in the heuristic from the command line as well as the shuffled board from stdin, create a root node with the initial state and then begin the loop: while the frontier is not empty, pop the head node, and then check if it's the goal state. If it is, print out the relevant information and if it isn't, generate all the children and then add the node to the closed list, sorting by least cost and resolving ties by selecting the newer node of the two.

Analysis and Results

After each run, no matter the heuristic, a few lines of output occur: the first is the number of visited/expanded nodes (V), second is the total number of nodes (N) that are stored in memory, third is the depth (d) of the current solution, and fourth is the branching factor (b) which is a function of the N and d values. After these values, the stack of nodes is printed, which corresponds to the solution of the search and shows the moves necessary to reach the goal state from the initial state. To experiment with the performance of each of the four heuristics, 100 different randomly seeded boards were solved for each one, and all their V, N, d, and b values were collected into text files. These text files were parsed using a bash script that sent data to a python stats program which printed out the min, median, mean, max and standard deviation of each data point.

The first heuristic was just a uniform cost search which has an h cost of zero. This caused it to be rather poor in its searching. Looking at the data in the table below, one can see that the average branching factor was 1.9162 and it stored on average 37,572 nodes which is not exactly ideal. It visited more nodes (V) in comparison to the other heuristics. In the maximum case: it expanded 164,794 nodes.

h(n) =0

V, N, d, b	Minimum	Median	Mean	Max	Standard Deviation
V	10.0	7048.5	23,151.23	164794.0	34,284.6554
N	19.0	11,997.0	37572.84	222376.0	53017.6534
d	3.0	15.0	14.66	26.0	5.3614
b	1.6057	1.8487	1.9162	2.8284	0.2139

The next heuristic test was based on the count of displaced tiles from the goal state. This was calculated using a call of len(np.argwhere(current!=GOAL)). One can see that the number of nodes visited was around 78 percent smaller, and it only used around a fourth of the maximum amount of memory that the first case used. In terms of depth, it was basically identical to the first round of results. Even with a relatively simple heuristic, significant improvements were made to the overall algorithm.

h(n) = Number of tiles displaced from the goal

V, N, d,b	Minimum	Median	Mean	Max	Standard Deviation
V	3.0	387.0	1903.61	36584.0	4442.9805
N	8.0	637.5	3039.66	55811.0	6895.0524
d	3.0	15.0	14.66	26.0	5.3614
b	1.4029	1.5227	1.5326	2.0	0.0644

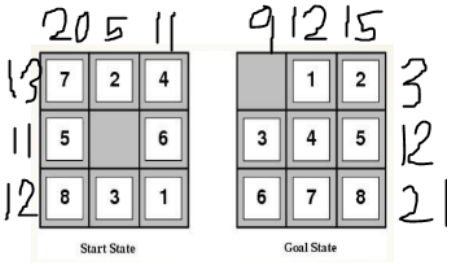
The third heuristic test was based on the Manhattan distance of the tiles from their goal positions. Out of all the heuristics, it performed the best in terms of storage and visiting nodes. It visited the fewest nodes on average with a mean of 290.15 and stored the least number of nodes in memory on average with a value of 479.91. The depth did perform a little worse on average, but not enough to warrant not using the heuristic.

h(n) = Sum of Manhattan (city-block) distances of all tiles from the goal

V, N, d,b	Minimum	Median	Mean	Max	Standard Deviation
V	3.0	105.5	290.15	1763.0	422.1099
N	8.0	179.0	479.91	2903.0	687.4882
d	3.0	15.0	14.68	26.0	5.3663
b	1.3064	1.4211	1.4424	2.0	0.0984

The fourth and final heuristic test was a custom heuristic. It operates off the idea that each column and each row of the goal state will have their own unique sums. One can add up the absolute difference between the current state and the goal states' sums and get a number that tends towards zero for puzzles that are "closer" to the goal. For example, the state below would

have a value of (11+7+4) [columns] + (10+1+9) [rows]=42. In terms of performance, it is in a tier between the Manhattan distance and misplaced tiles heuristics in terms of storage, as its V and N values are between these two. However, as indicated by the high standard deviation across all categories, it doesn't operate all that consistently. It appears it performs well in certain situations but when it's not in those situations, it "drops the ball", so to speak. It had a deeper traversal in its maximum depth case and the branching factor also had a significantly higher standard deviation than the others which means the user can't really depend on it performing consistently.



h(n) = Difference of sum of columns and rows from goal state

V, N, d,b	Minimum	Median	Mean	Max	Standard Deviation
V	3.0	301.5	854.11	9633.0	1323.4203
N	8.0	509.5	1409.61	15588.0	2159.2931
d	3.0	18.0	17.16	31.0	7.3426
b	1.2640	1.4256	1.5355	2.6878	0.2873

Conclusions and Future Work

In comparing their performance, the Manhattan distance fared the best in its number of nodes visited and stored, and the uniform cost search fared the worst as expected [it had no heuristic]. As the heuristic got "better", the number of nodes visited and stored decreased, and the branching factor gradually approached one. All except the custom heuristic had around the same depth average.

The custom heuristic performed better than initially expected, but it didn't do enough to warrant using it over the Sum of Manhattan heuristic. It is doubtful that it is an admissible heuristic, as it can possibly underestimate the cost of a solution because it has no real notion of whether the sum is high because of misplaced rows or misplaced columns. It can make a move that has a lower cost, but in reality, works against getting in the proper order with respect to the least number of moves. It would be interesting to see how it performed in an 8+ puzzle because the sums of the rows and columns would be larger.

Overall, the results were consistent with what one would expect from these heuristics.

References

Phillips, Joshua Dr. (2022, August 29). *Complexity Analysis and Search*. [26]. Department of Computer Science. Middle Tennessee State University. https://jupyterhub.cs.mtsu.edu/azuread/services/csci4350-materials/private/2022-08-29_Chapter_3a.pdf

Phillips, Joshua Dr. (2022, September 7). *Informed Search*. [7-9, 10]. Department of Computer Science. Middle Tennessee State University. https://jupyterhub.cs.mtsu.edu/azuread/services/csci4350-materials/private/2022-09-07_Chapter_3c.pdf

Russell, Stuart J. (Stuart Jonathan). (2020). *Artificial intelligence : a modern approach.* Upper Saddle River, N.J. :Prentice Hall,