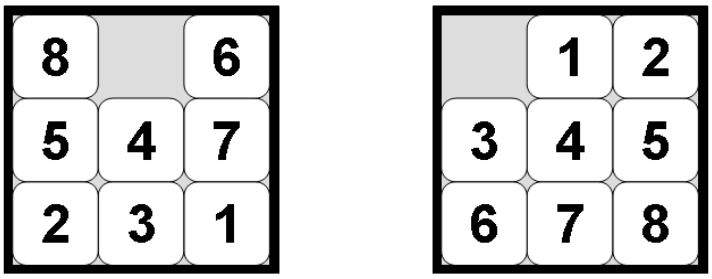
Matthew Schroder

9-25-2020

4350

The 8-puzzle problem starts with a randomized 3 by 3 board with the numbers 1 to 8 and a single blank space (see below). To solve this puzzle, one must rearrange it to match the goal state. This is done by repeatedly swapping the single blank space with an adjacent tile.



link

Once the goal state on the right has been reached through a series of valid moves, we can say that the puzzle has been solved.

A search algorithm can be used to find a series of moves that will solve the puzzle. In this experiment we use a searching algorithm called A* to find the path from the initial state and the goal state. The A* algorithm can be generally expressed as

$$\mathbf{F(n) = g(n) + h(n)}$$

where g(n) is the total path cost from the root node to the current node and h(n) is the heuristic function used. In this implementation the path cost from one node to the next is simply +1. If the heuristic function, h(n), is admissible then the path we get from the initial state to the goal state will be optimal. Optimal meaning that the series of moves returned from A* will be the shortest series of moves required to get back to the goal state. Three heuristic functions were written for this program; the first, *h*1, returns the number of tiles displaced. The second, *h*2, returns the sum of the Manhattan distance, which can be found with the equation below.

$$|x_1 - x_2| + |y_1 - y_2|$$

link

The previous heuristics are commonly used to solve the 8-puzzle problem while the third is one of my own design. The third heuristic looks at the state of the board and returns the number of valid moves that can be made. If the blank space were in the middle of the board then there would be 4 available moves and if it were in a corner, there would only be 2 available.

My implementation of the A* search uses three different data structures: a node, a heap, and a set. The node class stores information about the current state such as the depth, the current

board configuration, the F(n) and g(n) values, the node id, and the parent node. The heap is implemented as a priority queue that sorts nodes first by the smallest F(n) value and then by the largest node id. Using the largest node id means that newer nodes are preferred over older nodes. Finally, the set is used as a closed list. The closed list keeps track of all the nodes that have been visited and their corresponding board configurations. The closed list allows the program to evaluate an incoming board configuration and decide, based on whether a similar state has already been expanded, if should be pushed onto the queue. Though the solution could be found without using the closed list it does save memory resources by pruning branches of the search tree that have already been explored.

Now that you are more familiar with the project and the various components that are apart of it. Let's discuss how the analysis of this program was done. The program keeps track of a few key numbers: the total number of nodes visited/expanded (V), the maximum number of nodes stored in memory (N), the depth of the optimal solution (d), the -approximate- effective branching factor (b) where N = b^d. Below are the statistics for these performance measures.

V				
Statistic	**h0**	**h1**	**h2**	**h3**
min	134	12	17	49
median	172619	4414.5	846.5	51140.5
mean	385383.7	13533.27	1821.46	108288.6
max	2113346	120360	20401	523998
std dev	486037.1	24038.2	2766.665	125109

N				
Statistic	**h0**	**h1**	**h2**	**h3**
min	67	10	13	28
median	62541.5	2326	486.5	27746.5
mean	95593.44	7001.44	1056.14	50032.89
max	288446	58056	11480	189449
std dev	89206.46	11854.82	1580.735	51104.66

d				
Statistic	**h0**	**h1**	**h2**	**h3**
min	4	4	4	4
median	18	18	18	18
mean	17.92	17.92	17.96	17.92
max	26	26	26	26
std dev	4.538678	4.538678	4.556979	4.538678

b				
Statistic	**h0**	**h1**	**h2**	**h3**
min	1.61902	1.474211	1.304092	1.589929
median	1.828009	1.525411	1.401681	1.747107
mean	1.850251	1.527894	1.413599	1.767557
max	2.861006	1.778279	2	2.300327
std dev	0.164006	0.036672	0.073421	0.109854

Based on the tables above, we can see that h2 outperforms the other heuristics by a wide margin. Since this is the case, we can say that h2 is the dominant heuristic. The second-best performing heuristic was h1, which calculated the number of tiles displaced on the board. It makes sense that h1 would be dominant over h2 since h2 expands the idea displacement. It quantifies how displaced a tile is from where it should, which is a more intelligent observation than h1 was making.

I believe the overall strategy used in this program to be good. The data structures used here provided a framework A* to work in, but it may be that other data structures would prove to be more efficient. If I were to revisit this project, I would experiment with other data structures and heuristics to see where improvements could be made in both efficiency and solution.