William Riggs

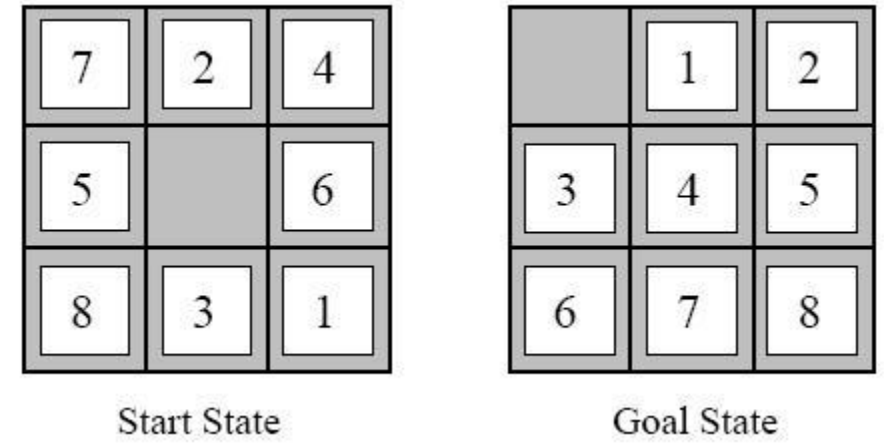
Dr. Joshua Phillips

CSCI 4350 – Into to AI

October 2nd, 2018

Solving the Sliding 8-Puzzle with A-Star Algorithm

The 8-Puzzle problem is a simplistic little puzzle, but it is intricate enough to be a great starter problem for solving with informed search algorithms. The board is a 3x3 matrix with 8 tiles and 1 blank square. The goal of the game is to rearrange the tiles into the original goal state. Moves are made by sliding the adjacent tiles into the blank space.



**1.

The first step in solving the problem is creating one. Solvable puzzles needed to be made that the solution code could run on. For that, a small program was developed that took in a matrix at the goal state, a seed for the random number generator, and the number of moves that the user wants to run. The program makes the number of random moves and outputs the result. In this case, the random board generator was used to create 100 different random puzzles to run the solution code with.

The solution used to solve this problem is the A-star algorithm. It is a shortest path tree search based algorithm not too different from a breath or depth first search. It uses the classic open and closed lists and chooses the next node to expand based on the shortest path costs so far. However, those algorithms are what is called an “uninformed” search, whereas a-star search is an “informed” search. It chooses what nodes to explore next, not just with the path cost so far, but also chooses the next node with a predictive heuristic. The heuristic is a calculation based on the current state for predicting how many more moves it will take to solve the problem. It is important that the heuristic gets as close to the real path cost, but it must not overestimate or it risks not finding the optimal path. The heuristic functions will be gone over more later, but for now know that the open list is ordered by the path cost so far (g) plus the heuristic (h).

$$F(n) = g(n) + h(n)$$

A-star algorithm works as follows. The start state is expanded and put on the closed list. Its children (possible board states) have their F(n) calculated with the path cost so far and the predictive heuristic function, h(n). The children are then added to the open list. The node with the smallest F(n) value gets compared to the goal state to see if the goal has been found. If the goal is not yet reached, the node gets taken off the open list, added to the closed list, and its children get put on the open list if they are not already on the closed list. This cycle is repeated until a node with the goal state is taken off the open list at which point the algorithm is finished.

A node of the tree is a structure made up of a matrix containing the board state, a pointer to its parent node, an integer g to hold the path cost to this point, an integer h to hold the heuristic cost, and a unique integer node id. The node id is used to break tie breakers on the open list in the favor of newer nodes when the F(n) of nodes are the same. Two sets of pointers to nodes make up the two lists, open and closed. When a state gets removed from the open list and put on the closed, its children get expanded. Those children are all the possible moves (board states) that can be made and don't already exist on the closed list themselves. This of course keeps the program from unnecessarily running over previously visited nodes.

State n:	Goal State:	
5 2 6	0 1 2	h1(n)=6
3 0 1	3 4 5	h2(n)=15
8 7 4	6 7 8	h3(n)=17

For this problem, three different heuristic functions were developed and tested against. Again, the goal of a heuristic function is to predict the number of moves required to get to the solution from node n without over estimating. The first heuristic function looks at the board state and counts the number of tiles that are not in their goal state. In the example above, the result from the heuristic function would be six, as 3 and 7 are the only tiles in their correct location. The second heuristic calculates the Manhattan Distance of the tiles. The Manhattan Distance is the number of moves that it would take for the tile to move from where it is to its goal state. In the example, 5 needs to move right twice and down once. So its Manhattan Distance is three. Doing this for all of the tiles and summing the values gives fifteen, which is the value returned by the second heuristic function for this case. For the third heuristic function, it calculates the Manhattan Distance and also adds two moves per linear conflict that it finds. A linear conflict occurs when two tiles are next to each other in the same row or column, their goal states are in that same row or column, and at least one is blocking the other from reaching its goal state **2. In this example, there is one linear conflict between 7 and 8, as they are next to each other, share the same row, share the same goal row, and the 8 is being blocked from its goal by the 7. Therefore the third heuristic returns fifteen plus two from the linear conflict.

Once the algorithm is done running and the solution is found, the program prints the number of nodes that it has expanded as the value "V," the number of nodes stored in memory as "N," the depth in the tree that the solution was found as "d," and the approximate branching factor of the algorithm as "b" where $b = N^{1/d}$. The program then prints out the game boards for the solution by traveling up the tree using the parent node pointers. The output looks as follows.

V=273
N=446
d=18
b=1.3457522

6 7 0
4 1 3
8 5 2

6 7 3
4 1 0
8 5 2
…

The A-star solution program was run against the hundred random boards four times, each time with different heuristic. For the first run the heuristic was set to zero, essentially running the A-star without a heuristic function. Unfortunately, it was so inefficient that it was not able to run all one hundred of the boards. Instead, the program ran the first five boards and the board with the median results was selected to compare to the other heuristics' run of that board. Although this is far from ideal, one can still clearly see how vast the gap is from the heuristic-less run and the other runs. Without a heuristic, A-star preforms the same as a breadth first search. Even a very simplistic and limited heuristic, like the number of misplaced tiles, makes a huge difference for how the algorithm preforms. Comparing the h0 V and the h1 V you can see that h1 expands 2% of the nodes that h0 does.

	h0	h1	h2	h3
V	13022	265	52	39
N	20879	610	93	71
d	16	16	16	16
b	1.85868	1.49308	1.32749	1.30528

For the other heuristics, they were run against the one hundred boards and after testing, the data was compiled and the mean, median, minimum, maximum and standard deviation were calculated for each statistic of the three runs.

Headers	**Mean**	**Median**	**Min**	**Max**	**SD**
V (h1)	4947.63	1588.5	8	40083	7568.561
V (h2)	411.2	183	6	2999	570.9062
V (h3)	272.09	118.5	6	1971	367.7267
N (h1)	7753.52	2547	16	59868	11537.83
N (h2)	660.64	311.5	13	4716	898.8971
N (h3)	446.7	199	13	3125	590.1155
d (h1)	18.1	18	6	26	4.844032
d (h2)	18.1	18	6	26	4.844032
d (h3)	18.1	18	6	26	4.844032
b (h1)	1.518566	1.51575	1.43097	1.70998	0.032329
b (h2)	1.367008	1.364635	1.24679	1.61887	0.054951
b (h3)	1.345752	1.338465	1.23397	1.61887	0.057415

If all the heuristics are admissible (don’t over estimate), then they will always find the optimal solution. Since the statistics for the depth of the solution is the same for all the heuristics, it helps to confirm that the heuristics are all admissible. As the heuristic gets closer to the actual cost of the goal, the statics for the mean, median, min, and max all drop drastically.

In conclusion, A-star search demonstrates the ability of informed search strategies to find complete and optimal solutions with low space and time complexity. In the case of this 8-puzzle problem, using heuristics made problems where one would run out of memory or time, solvable in milliseconds and with very little memory cost.

Works Cited

1. Pandare, Rohan. “Solving 8 Puzzle Using RBFS.” *Google Sites*, sites.google.com/site/rohanpandare/projects/solving-8-puzzle-using-rbfs.
2. AlgorithmsInsight. “Implementing A-Star(A*) to Solve N-Puzzle.” Insight into Programming Algorithms, 3 May 2016, algorithmsinsight.wordpress.com/graph-theory-2/a-star-in-general/implementing-a-star-to-solve-n-puzzle/.