Rachel Chiang

Project 1: 8-Puzzle

CS 420.01

Solving the 8-Puzzle Using the A* Search with Two Different Heuristic Functions

The 8-puzzle consists of a three by three grid populated with eight tiles of numbers (the digits one to eight) and one empty tile (a blank tile). These tiles start out scrambled, and to solve it, the tiles must be rearranged such that the numbers on the tiles are displayed in increasing order with the blank first (in the upper left corner), but only the blank tile can move by swapping places with an adjacent tile in the same row or column. Formally, the problem can be defined like so:

Number of States:	9!/2
Initial State:	Any one of the states
Actions:	The blank tile's movements up, down, left, or right
Transition Model:	Given a state and an action, return the next state
Goal Test:	If the state matches the goal state (the sorted puzzle with the blank first), then the problem has been solved
Path Cost:	Each move costs 1

The problem definition.

To solve the puzzle, the A* search can be applied, which combines the actual accumulated cost so far with a heuristic to estimate the total cost of a path or sequence of actions to the goal. For the 8-puzzle, the two heuristic functions that will be used are as follows: h_1 will be the number of misplaced tiles; h_2 will be the sum of all the numbers' distances from their goal position (the Manhattan distance). These heuristic functions are indeed both admissible, which is significant because A* will be used with the graph search

algorithm, in which a separate set will maintain the nodes that have been explored. The frontier uses a priority queue and the explored set will be an unordered map for fast searches.

As one might expect, the heuristic function that reveals more information about the puzzle's true configuration ended up performing much better.

	Search Cost [nodes generated]		Runtime [milliseconds]		
d	A*(h_1)	A*(h_2)	A*(h_1)	A*(h_2)	count
1	3.333333333	3.333333333	0.0809533	0.08165778	45
2	7	7	0.1373571	0.13781786	28
3	10.94736842	10.44736842	0.1873921	0.18057632	76
4	13.34848485	13.36363636	0.2242424	0.22660606	66
5	18.5034965	17.79020979	0.3068182	0.29156783	143
6	26.25	23.52884615	0.4330865	0.38535192	104
7	37.97979798	31.08080808	0.5927848	0.5054399	198
8	48.7007874	39.37007874	0.7535	0.63187402	127
9	79.7125	50.675	1.1553333	0.79278208	240
10	109.7179487	64.63247863	1.5794197	0.98293846	117
11	172.7475248	88.16336634	2.554103	1.29001782	202
12	250.7204301	125.8924731	3.7962871	1.86499785	93
13	411.9210526	173.5	6.0774184	2.60203355	152
14	616.525	239.4125	9.0186325	3.66747625	80
15	979.015873	320.9047619	14.647591	4.83452064	126
16	1523.561404	520.5087719	25.609042	7.75818246	57
17	2371.012346	634.0617284	43.123781	9.60802099	81
18	3708.565217	1071.043478	76.399343	17.0766826	23
19	6060.694444	1493.611111	149.96979	25.0122778	36
20	8572.125	2223.125	248.75325	42.21325	8
21	14825.15385	3091.692308	572.89623	66.8439462	13
22	19767.625	3260.125	948.77313	67.9606875	8
23	30013.875	6377.625	1891.679	169.869313	8
24	39759	5763.5	3037.0675	142.146775	4
25	59600.25	9323	6338.5775	299.81525	4
26	111536.2	13586.2	18760.926	506.8706	5
27	146220.4	23754.2	30277.52	1292.549	5
30	357126	64377	132569	6929.78	1

Comparison of the average search costs, in number of nodes generated, and average runtimes, in milliseconds, for the A algorithms implemented with h_1 and h_2. The number of random cases tested for each solution length d is provided in the "count" column.*

The Manhattan distance heuristic function had lower search cost and time elapsed starting from somewhat early on, and the two started to deviate heavily from when puzzles had solution lengths of 5 and greater, though prior to that they were about the same, which should make sense since an easy puzzle will have a low number of possible states to explore and therefore will have very few choices to make, so making the right choice is more likely. Clearly, the Manhattan distance heuristic function is better in terms of time and space, but the one aspect that the misplaced tiles heuristic function is (subjectively) superior in is that it is easier to implement, since it simply counts how many numbers do not match the tiles of the goal state, which needs only one loop with one if-statement to count mismatching tiles, whereas the Manhattan distance heuristic function needs a few if-statements to count the tiles. As such, the Manhattan distance heuristic function *alone* actually has an extremely slight, arguably negligible, time disadvantage to the misplaced tiles heuristic function, which is visible at $d \leq 2$, when the two heuristic functions shared the same search costs. However, the number of nodes generated is what truly dictates the overall performance of the search, and the more sophisticated heuristic naturally elects the more intelligent decisions and gets to expand fewer nodes.

Mentioned previously were the assertions that both heuristic functions were admissible. This is necessary because graph searches will only be optimal if the heuristic functions are admissible. To be admissible, the estimated cost must always be equal to or less than the true cost. The misplaced tiles heuristic function h_1 is clearly admissible because if a tile is out of place, then it must be moved at least one time. Notably, the blank space is not

counted; for example, given a puzzle 102345678, to count both the 1 and the 0 as misplaced would give an estimated cost of 2, even though the true cost is 1 move. The Manhattan distance heuristic function h_2 is also admissible because all of the estimated costs count the distances of tiles to their goal without considering the existences of the other tiles or even of the blank's movements, so they are always best-case scenarios. In reality, any move can only move one of the tiles closer to the goal, so h_2 is always equal or underestimating the true cost. Similar to h_1, the blank's distance is also not counted.