A* Algorithm Analysis

TREE SEARCH & GRAPH SEARCH
ON AN 8-PUZZLE PROBLEM

Pedro Beltran | CS 4200 | 9-22-19

Test Approach

To obtain the best results, I tested each algorithm 1,000 times. Test cases where generated randomly and each algorithm tested the same cases. To ensure that algorithms did not get stuck on any unsolvable case, I checked the number of inversions in each case and only tested it if it had an even number of inversions. Otherwise, the case was discarded. In order for the test to run faster, I capped the search cost at 800,000 nodes. This was really only for A* Tree Search; all A* Graph Search solutions were found in under 800,000 nodes. I timed the execution of each algorithm by storing system time immediately before algorithm execution and immediately after, subtracting the before time from the after time to result in algorithm runtime. For efficiency, 8-puzzle configurations were stored in a one dimensional integer array, where every 3 elements are a row in the 8-puzzle. Solution data was written to a file of each respective algorithm with each heuristic in a formatted fashion for easier processing.

Tree Search vs Graph Search

It is no surprise that my data showed A* Graph Search (A*GS) to be significantly faster than A* Tree Search (A*TS) with each respective heuristic. This is expected because unlike A*GS, A*TS has no explored set and it does not check the frontier set for repeating puzzle configurations, resulting in the generations of significantly more nodes. Although both algorithms of are greedy best first searches and they maintain a time and space complexity of O(b^d) where b is branching factor and d is depth, A*GS has more restrictive branching constraints. So the branching factor for A*GS is smaller than the branching factor for A*TS, resulting in a more efficient use of time and space. Consider this, the largest search cost recorded for A*TS with heuristic function 1(h1) was 799,489 nodes at a depth of 20, while the largest search cost recorded from A*GS with h 1 was 89,953 nodes at a depth of 29. On average solutions found by each of these two algorithm cost 237,594 nodes at an average depth of 17 and 13,880 nodes at an average depth of 22 respectively. A*TS and A*GS both with heuristic function 2(h2) further emphasized the efficiency of A*GS. On average solutions found by each of these two algorithm cost 97,373 nodes at an average depth of 21 and 1,208 nodes at an average depth of 22 respectively. Similarly, runtimes times showed that A*GS is significantly faster than A*TS.

Misplaced Tiles (h1) vs Manhattan Distance (h2)

As expected, heuristic function 2 showed to be more valuable than heuristic function 1. This is because h2 yielded a more accurate distance from the goal state. That is, h1 only accounted for misplaced tiles, while h2 accounted for the distance each tile was from its goal position. Consider A*TS. Since its search cost was capped at 800,000 while testing, it only produced 204 solutions with h1. Although, with h2 this was improved by producing 4.4 times more solutions. Moreover, with h1 it produced solutions of max depth 21 but with h2 it produced solutions with max depth 29. Similarly, A*GS performed better with h2. Both h1 and h2 with A*GS yielded 1000/1000 solutions for the cases tested, but on average h1 used 11.5 times more nodes than h2. Furthermore, average runtime with h1 was 3.947 seconds, while with h2 it was 0.029 seconds. It is undeniable that h2 is best candidate heuristic function.

Challenges and Constraints

Due to time and memory constraints, effective testing of A*TS was not possible, especially with h1. With A*TS, I found that my system ran out of memory after generating the 7,607,262th node. This is probably due to its larger branching factor as compared to that of A*GS. Furthermore, I found that solutions with search cost greater than 800,000 nodes took significantly more time to calculate.

Tables of Averages

A* Tree Search averages by depth. #DIV/0! means the value does not exist.

Depth	Search Cost H1	Runtime H1 (ms)	Solution Count H1	Search Cost H2	Runtime H2 (ms)	Solution Count H2
7	7	0	1	7	0	1
8	#DIV/0!	#DIV/0!	0	#DIV/0!	#DIV/0!	0
9	118.5	1	2	19.5	0	2
10	84	0	1	16	0	1
11	1225.5	5.5	2	186	0.5	2
12	1346	5.666666667	6	202.6666667	0.666666667	6
13	5319	22	4	2599.5	11.5	4
14	9049.25	39.5	4	1464.25	6.5	4
15	21646.6875	113.5625	16	3063.8125	13.5	16
16	56732.75	300.7916667	24	5262.208333	22.41666667	24
17	126795.1034	657.2758621	29	8963.517241	37.89655172	29
18	237162.3276	1229.758621	58	15565.38983	83.28813559	59
19	434080.1471	2188.235294	34	33504.59259	170.6666667	54
20	633021.3636	3194.363636	22	38415.1875	199.4479167	96
21	575409	2815	1	73557.94	371.22	100
22	#DIV/0!	#DIV/0!	0	71773.44444	371.8205128	117
23	#DIV/0!	#DIV/0!	0	127997.3462	660.3365385	104
24	#DIV/0!	#DIV/0!	0	170560.5077	865.1230769	130
25	#DIV/0!	#DIV/0!	0	179644.6615	916	65
26	#DIV/0!	#DIV/0!	0	206248.5536	1053.017857	56
27	#DIV/0!	#DIV/0!	0	213326.8824	1117.764706	17
28	#DIV/0!	#DIV/0!	0	237898.9	1245.7	10
29	#DIV/0!	#DIV/0!	0	99954	544.5	2

A* Graph Search averages by depth. #DIV/0! means the value does not exist.

Depth	Search Cost H1	Runtime H1 (ms)	Solution Count H1	Search Cost H2	Runtime H2 (ms)	Solution Count H2
7	7	0	1	7	0	1
8	#DIV/0!	#DIV/0!	0	#DIV/0!	#DIV/0!	0
9	31	0	2	16.5	0	2
10	43	1	1	16	0	1
11	69.5	0.5	2	31	0	2
12	100	0.666666667	6	36.33333333	0.166667	6
13	142.5	1	4	43.5	0.25	4
14	218.25	1	4	61	0.25	4
15	372.625	2.8125	16	105.125	0.3125	16
16	502.4583333	4	24	108.6363636	0.636364	22
17	880	9.172413793	29	161.1481481	0.888889	27
18	1213.706897	15.79310345	58	213.9482759	1.310345	58
19	2063.12963	40.03703704	54	300.745098	2.137255	51
20	3179.278351	93.15463918	97	371.4137931	2.62069	87
21	4598.613861	197.1386139	101	539.90625	4.59375	96
22	7608.158333	563.5583333	120	688.078125	6.515625	128
23	10878.18018	1171.216216	111	973.2678571	11.33929	112
24	17383.30556	2993.590278	144	1375.451389	19.70139	144
25	24185.11494	5755.494253	87	1983.573034	38.97753	89
26	37055.28916	13205.40964	83	2510.807229	63.36145	83
27	44277.58621	19265.72414	29	3934.2	167.7714	35
28	61761.61905	36183.28571	21	4704.64	226.64	25
29	82609.5	61760.16667	6	7433.857143	544.5714	7

Conclusion

Despite the constraints, it is obvious that A*GS is the optimal algorithm for the 8-puzzle problem. Moreover, Manhattan distance is the better heuristic. Together they make a fast and powerful search algorithm with a very small branching factor that can find a solution of depth 29 within a fraction of a second.