Assignment 1

Sliding tiles puzzle

Name: Harsha Chandwani

SBU ID: 111481387

Section 1: *Heuristics – Describe the two heuristics you used for A*. Show why they are consistent and why h1 dominates h2.*

Solution: I have used the following two heuristics:

a) Manhattan Distance Heuristic

b) Misplaced Tiles Heuristic

a) Manhattan Distance Heuristic

This heuristic is calculated by taking the sum of the minimum distance of each tile position from the goal tile position. This heuristic is admissible because it does not overestimate the number of moves required to solve a problem. Every tile will have to be moved by at least the number of cells between itself and the correct position..

b) Misplaced Tiles Heuristic:

This heuristic is the total number of misplaced tiles. Like Manhattan Distance, this heuristic is also an admissible heuristic because it does not overestimate the number of moves required to solve a problem. It is quite obvious that the total number of moves required to reach the goal state of the puzzle will be at least the number of misplaced tiles, meaning they have to be moved at least once. The total cost of reaching the goal state has to be **at least** the Misplaced tiles heuristic.

Why H1(Manhattan Distance) dominates H2(Misplaced tiles heuristic)

The approximation given by the H1 is actually better than the one given by H2. With H2, we know if a tile is at it's ideal position or not, but we have no way of knowing how far it is from its ideal position in a solved puzzle. H2 treats all the tiles equally, a tile which is just a cell away from the goal state is treated the same as the one which is away from the goal state. Hence, we can say that H1 provides a better idea about which states to explore next.

Section 2. *Memory issue with A* -- Describe the memory issue you ran into when running A*. Why does this happen? How much memory do you need to solve the 15-puzzle?*

Solution: I did not come across any memory issues while running A* on my computer. However, with a few observations, we can conclude that A* consumes a lot of memory as it stores the explored and the frontier nodes in the memory.

From the Table 1 :A* Performance(see last answer), following are the observations for a few puzzles.

Sr. no	Depth	Explored nodes
1.	16	248
2.	20	238
3.	22	8817
4.	22	2127
5.	28	15679
6.	24	4966

the number of explored nodes is way too high when we look at the depth of the solution. Obviously, these numbers would increase exponentially if we increase the size of the puzzle.

The reason why we see this happening is because we explore every node at each depth level.

The number of nodes expanded would be b^d , where b is the branching factor and d is the depth of the solution. Nodes expanded will be exponential to the solution depth.

How much memory do you need to solve the 15-puzzle?

As per https://en.wikipedia.org/wiki/15_puzzle , the depth for the optimal 15 puzzle solution would be 80. So, since the branching factor in our case is 3,(actual number of moves possible is 4, but we do not take into consideration the reverse of the action that got us to a new state, otherwise we will be back to the previous state). So we will need the memory to store the expanded 3^80 nodes in this case.

Section 3: *Memory-bounded algorithm – Describe your memory bounded search algorithm. How does this address the memory issue with A* graph search. Is this algorithm complete? Is it optimal? Give a brief complexity analysis. This analysis doesn't have to be rigorous but clear enough and correct.*

Solution. The memory bounded search algorithm used is Iterative Deepening A* search. It works like this. It starts with an initial cutoff on the f value and does not consider a branch for expansion if its f cost is greater than the current set cutoff on the f value. The fcutoff intially is set to the f value of the start state. For the children of the the start state, the f value would be greater than this cutoff, so the cutoff is updated to the next higher cutoff amongst the children of the start node.Thus, this cutoff increases for every iteration of the algorithm and for each iteration, it is the miniAmum of the f values that exceeded the currently set f value cutoff.

How does this address the memory issue with A graph search.*

This algorithm does not maintain the list of explored nodes or the nodes in the frontier, which accounts for a major reason why A* consumes a lot of memory.

Is this algorithm complete?

Yes, if a solution exists, if is definitely going to find it.

Is it optimal?

Yes, the algorithm is optimal. This is because, at every iteration we update the cutoff to the next minimum f value of the nodes at the next level. It will never be the case that the we will find a goal node whose depth will be higher than the actual goal node.

Time complexity:

Though the time taken by the IDA* is more than the A*, the complexity remains similar to A*, i.e. O(b^d). The time complexity is decided by the number of iterations of the search.

Space complexity:

The space complexity will be directly proportional to the solution depth. If the depth is d, the space complexity is O(d).

4. A table describing the performance of your A and memory-bounded implementations on a randomly drawn set of 20 solvable puzzles. You should tabulate the number of states explored, time (in milliseconds) to solve the problem, and the depth at which the solution was found for both heuristics.*

Solution: The first 10 observations are for a 3x3 puzzle size and the remaining 10 observations are for 4x4 puzzle size. H1 represents the Manhattan Heuristic and H2 represents the Misplaced Tiles Heuristic.

Table 1: A* Performance

Sr.no	Nodes explored (H1)	Time(ms) (H1)	Nodes Explored (H2)	Time(ms) (H2)	Depth
1	7	0.41103	4	0.31995	4
2	42	1.73091	76	2.20894	12
3	15	0.67210	30	1.06430	10
4	43	1.54685	208	5.63621	14
5	40	1.53613	35	1.17540	10
6	248	7.98797	423	11.81530	16
7	47	1.93166	210	5.73039	14
8	82	2.89821	104	4.52494	12
9	7	0.47779	7	0.37956	6
10	12	3.06415	23	0.78392	10
11	238	11.58094	2597	96.56763	20
12	74	3.60178	11	0.57220	10
13	8817	413.65242	23055	932.10864	22
14	16	0.90980	6	0.45084	6
15	2127	96.69971	13070	507.68899	22
16	29	1.46579	6	0.40102	6
17	15679	714.03145	256930	11040.7147	28
18	232	11.24620	163	5.99479	16
19	2192	132.33304	19118	978.11889	22
20	4966	244.28892	28902	1210.07394	24

Table 2: IDA* Performance

Sr.no	Nodes explored (H1)	Time(ms) (H1)	Nodes Explored (H2)	Time(ms) (H1)	Depth
1	53	1.01399	8	0.24890	4

2	383	6.19387	479	6.05845	12
3	149	2.43782	141	1.84273	10
4	258	4.24885	651	8.14008	14
5	427	6.78944	208	2.52079	10
6	3567	56.37836	2924	35.27212	16
7	537	8.73398	677	8.40163	14
8	1137	18.17035	574	7.01737	12
9	42	0.79846	11	0.28705	6
10	64	1.13177	55	1.35993	10
11	1550	34.09385	10995	157.34291	20
12	744	16.11852	22	0.48017	10
13	128503	2803.96842	157008	2257.33923	22
14	162	3.52191	11	0.32138	6
15	22787	499.86195	80867	1169.80075	22
16	215	4.64129	12	0.39505	6
17	226465	5003.61967	1241563	18989.4816	28
18	2054	47.96743	761	17.28439	16
19	18129	473.79612	80254	1687.28399	22
20	51262	1200.25634	132961	2203.73296	24