Assignment 1 Report

Name: Swaminathan Sivaraman
SBU ID: 110951180
Course: Artificial Intelligence CSE 537
University: Stony Brook University

Section 1.
Heuristics - Describe the two heuristics you used for A*. Show why they are admissible (or consistent).

The two heuristics I used are,

1) The Misplaced Tiles Heuristic
2) The Manhattan Distance Heuristic

Admissibility for the Misplaced Tiles Heuristic:

This heuristic simply calculates the number of misplaced tiles in our start state. This is an admissible (and consistent) heuristic because each tile that is not in its final position *has* to be moved at least once. And since we can't swap two numbered-tiles (i.e., we can only move a tile to a gap position), we can increment 1 cost for each misplaced tile individually. It turns out that this is a very poor heuristic though, since it minimizes the cost too much.

Admissibility for the Manhattan Distance Heuristic:

This heuristic calculates the minimum number of moves it would take for each misplaced tile to reach its goal position. This is again admissible because each tile can reach its goal position only by moving through the board and it definitely has to at least make the number of moves from its start position to its goal position (Again, this can be calculated individually for each tile since a tile can be swapped only with a gap and not with another numbered tile). This is a heuristic that performs far better than the Misplaced Tiles Heuristic.

Better Heuristics:

But even the Manhattan Distance Heuristic doesn't use all the knowledge we have about the puzzle. For example, if two tiles are on the same row and their goal positions are also on the same row, but the tiles' order is opposite to that of the goal positions' order, then one of the tiles has to go around the other, thereby adding at least 2 more moves to our estimated cost. This heuristic is called the Linear Conflict Heuristic (I haven't implemented this though). There are even better heuristics like the Pattern Database one but that involves pre-computing results.

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 15puzzle?

My actual implementation itself didn't run into any issues, but this is due to the large amounts of memory available on modern computers and we can show why A* will take up a lot of memory by looking at the explored and frontier sizes for solved puzzles. The frontier and explored lists will be small for smaller solutions, so let us take a large solution as an example.

Start state: (The solution for this takes 46 moves)
```
,10,4,11
3,9,14,2
15,13,12,7
5,1,6,8
```

If we run A* on this puzzle, this is what we have after the puzzle has been solved,
```
Frontier size 201540
Explored size 208053
```

Clearly this is too much for a 46-move solution and as the number of moves grow for bigger nxn puzzles, these sizes will grow exponentially and we will run out of memory even on modern computers.

The number of nodes expanded will be b^d, where b is the branching factor and d is the solution depth. Better heuristics will have a smaller branching factor but it will still be exponential in solution depth.

For the 15 puzzle, the maximum depth is 80 moves (Source). For a very bad or worst-case heuristic, if our branching factor is 3 (4 possible moves minus the one that will take the state back to the previous state), we will expand 3^80 nodes (~1.47808829414e+38). Even with better heuristics, this number will not reduce that much, as the exponent d will be the same.

This is happening because we are exploring every possible node at each depth level, while really only one (or a few) solution exists. Memory-bounded A* algorithms seek to find out the solution directly, trading off time for doing it.

Section 3.

Memory-bounded algorithm - Describe your memory bounded search algorithm. Is this algorithm complete? Is it optimal? Give a brief complexity analysis. This doesn't have to be rigorous but clear enough and correct.

I used the Iterative-Deepening A* or IDA* algorithm. It is a depth-first search algorithm. But instead of doing a complete depth-first search, it sets a threshold and only does a search for that threshold. If it doesn't find a goal, it increases the threshold and restarts the search from scratch. The threshold here will be in terms of f(n). The algorithm first sets the threshold to f(n) at the root node. It then expands nodes whose f(n) values are lesser than or equal to this threshold in a recursive depth-first manner, and looks for a goal. Whenever it sees a node with a greater f(n) value, it saves that value before rejecting that node and path. If it couldn't find the goal, it selects the minimum value from these saved values and sets that as the threshold and restarts the search (Note that the algorithm doesn't maintain explored/frontier lists).

Completeness:
This algorithm is complete, because if a goal is present, it will eventually be found.

Optimality:
This algorithm is definitely optimal. The f(n) at the root node (threshold) will specify the minimum number of steps to the solution, or rather what the heuristic thinks is the perfect solution. So, the algorithm will explore all paths with depths equal to this threshold (obviously considering only paths where f(n) is less than or equal to the threshold). If we couldn't find the goal, we'll update the next smallest minimum from the leaf nodes as the new threshold. On each cycle, we'll find out what the minimum possible solution is, and explore all valid paths whose depth is equal to this solution and see if the goal is at the end of those paths. So, we will never find a non-optimal goal node (whose depth will be higher than an optimal goal node) before an optimal goal node.

Complexity:
Space Complexity:
As can be seen, the algorithm doesn't make use of a lot of extra space. If d is the solution depth, we will need O(d) space. Note that we can do a pairwise comparison as we go through the graph to find the next minimum and don't need to maintain a list of next possible values and then choose one.

Time Complexity:
IDA* will take more time than A* but the complexity itself will not change. For branching factor b, at the final solution depth, the time taken will be O(b^d). At each lower depth, the time taken will be O(b^(d-1)), O(b^(d-2)) and so on up to O(b). It can be seen that the first term dominates all other terms (Big O notation rules), and the time complexity will be O(b^d).

Section 4.
A table describing the performance of your A* implementation 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.

All samples below are 4x4 puzzles solved using the normal A* algorithm with the Misplaced Tiles and Manhattan Distance heuristics. The nodes explored is the sum of both the frontier and the explored lists when the goal was found.

Sample No#	Solution Depth	Time - Misplaced Tiles (ms)	Time - Manhattan Distance (ms)	Nodes explored - Misplaced Tiles	Nodes explored - Manhattan Distance
1	8	1.193	1.449	35	21
2	5	0.444	0.443	14	14
3	21	234.207	16.356	8878	573
4	15	4.26	1.227	76	37
5	15	6.902	4.62	254	104
6	16	12.345	2.832	410	101
7	9	0.944	1.041	29	26
8	20	133.643	3.56	4380	127
9	19	78.926	5.168	2834	181
10	9	0.915	3.232	30	27
11	13	11.318	1.076	280	35
12	16	23.715	2.369	702	80
13	23	1985.679	31.863	74512	1101
14	8	3.095	0.742	32	21
15	25	2139.677	114.348	80081	3822
16	18	12.551	1.178	351	40
17	7	2.737	0.58	19	14
18	15	11.567	1.925	293	54
19	22	92.33	4.308	3174	143
20	8	0.69	0.606	18	18