Gabriello Lima 112803276 - Assignment 1 Report

1. **Problem Formulation - Describe how you define the problem in the TileProblem class.**

In the TileProblem class, I defined the problem as a list of lists, which boils down into a matrix. Each index of the list corresponds to a row and another list, and within that other list, each index corresponds to a column. It ends up looking something like this: [[1,2,3],[4,5,6],[7,8,0]] where 0 represents the position the blank space is supposed to maintain. I broke this class into two functions: solveTile, and constructInstance. solveTile returns a solved tile of the given size in the format mentioned above. This is useful for calculating Manhattan distance, and my other Heuristic — the number of misplaced tiles. On the other hand, constructInstance creates the instance of the input file in the format mentioned above.

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

In my Heuristics class, I defined two Heuristics, H1 and H2, which respectively correspond to Manhattan and the number of misplaced tiles. H1 will always dominate H2, because H1 calculates the number of rows and columns away each tile is from its respective solution row and column. On the other hand, H2 merely calculates how many tiles are misplaced. In that case, if there is 1 tile misplaced (H2), H1 will always be greater than or equal to 1, as that tile has to be at least one tile away from its proper position.

3. 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?

A* ends up using a decent amount of memory because of the lists it has to maintain, which exponentially increases with the depth of the search. For puzzle 4, I ended up using 14,676 MB of memory.

4. 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.

My algorithm behaves as follows: generate some children of the current state. Sort these children by their respective Heuristic costs, and recursively dive into each of those children. If there is an issue, F-Limit or otherwise, return false, indicating a failure. If all the children returned false, the F-Limit needs to be increased. We keep repeating this process of recursively diving, returning an issue, and increasing the F-Limit until a solution is found.

The algorithm is both complete and optimal. It is complete in the sense that, if there is a solution to be found, it will find it. As the F-Limit approaches infinity, RBFS will dive deeper and deeper until it finds the solution. As for optimality, specifically for memory, RBFS has a memory complexity of O(bd). Compared to the exponential nature of A*'s memory complexity, RBFS linear memory complexity is a major improvement.

5. A table describing the performance of your A* and memory-bounded implementations on the five test input files shipped as part of the assignment. You should include the output of your solver on the five test input files. You should tabulate the number of states explored, time (in milliseconds, you can use datetime to measure the time) to solve the problem, and the depth at which the solution was found for both heuristics.

H1	Time	# States	Depth
A* Puzzle 1	1	2	2
RBFS Puzzle 1	1	2	2
A* Puzzle 2	1	4	4
RBFS Puzzle 2	1	4	4
A* Puzzle 3	3	23	8
RBFS Puzzle 3	2	25	8
A* Puzzle 4	3	15	6
RBFS Puzzle 4	2	22	6
A* Puzzle 5	9	36	10
RBFS Puzzle 5	3	33	10

H2	Time	# States	Depth
A* Puzzle 1	1ms	2	2
RBFS Puzzle 1	1	2	2
A* Puzzle 2	1	4	4

RBFS Puzzle 2	1	4	4
A* Puzzle 3	1	67	8
RBFS Puzzle 3	2	60	8
A* Puzzle 4	1	6	6
RBFS Puzzle 4	1	6	10
A* Puzzle 5	3	26	10
RBFS Puzzle 5	2	16	10