Suheyla Iyimaya (112006174)
Niranjan Balasubramanian
CSE 352
03/01/2020

Assignment #1: Search Report

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

   The **TileProblem** class defines the 8 or 15 tile puzzle problem. It has the variables start_state, goal_state, actions, puzzle_size, and methods transition_function, and goal_test.

   Start_state variable defines the initial state of the problem as an array of characters where indices 0-(8 or 15) have the numbers that are on each tile initially. The blank tile is represented as ‘0.’

   Goal_state variable is the state that we are trying to reach. For 8-puzzle problem our goal_state is [‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘0’] and for 15-puzzle problem our goal_state is [‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘10’, ‘11’, ‘12’, ‘13’, ‘14’, ‘15’, ‘0’].

   Actions variable represents the moves that can be done by the user, which are moving the tile up, down, left, and right. This is represented as an array of characters [‘U’, ‘D’, ‘L’, ‘R’].

   Puzzle_size variable is the integer 3 if our problem is 8-puzle and it is 4 if our problem is 15-puzzle.

   Transition_function method applies the given action to the given state and returns the resulting state.

   - If the provided action is ‘U’ (moving a tile up), then it swaps the positions of the blank tile (let’s say state[i]) with the position of the tile at state[i - 3] for 8-puzzle. For 15-puzzle, it swaps the positions of the blank index (let’s say state[i]) with the position of state[i - 4]
   - If the provided action is ‘D’ (moving a tile down), then it swaps the positions of the blank tile (let’s say state[i]) with the position of the tile at state[i + 3] for 8-puzzle. For 15-puzzle, it swaps the positions of the blank tile (let’s say state[i]) with the position of the tile at state[i + 4]
   - If the provided action is ‘L’ (moving a tile left), then it swaps the positions of the blank tile (let’s say state[i]) with the position of the tile at state[i - 1] for 8-puzzle and 15-puzzle.

- If the provided action is 'R' (moving a tile right), then it swaps the positions of the blank tile (let's say state[i]) with the position of the tile at state[i + 1] for 8-puzzle and 15-puzzle.

Goal_test method checks if the provided state (represented as a character array) is equal to the goal_state array by comparing each index of the two arrays.

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

   - In my implementation, h1 checks the number of misplaced tiles in the whole board while h2 checks the number of misplaced tiles in the board excluding the last 4 tiles. h1 always dominates h2 since h2 looks at a part of what h1 looks at. Meaning h1 contains h2. h1(n) is always greater than or equal to h2(n).
   - For example, if the state of our board was ['1', '5', '3', '4', '2', '6', '8', '7', '0'], h1 would give us 4 while h2 would give us 2 since it doesn't check the last 4 tiles.
   - Better heuristic is h1(), which gives higher h values to all the nodes, but still it's an underestimate of the true distance (to satisfy admissibility).
   - Both h1 and h2 are consistent since the estimated cost of a node is always less than or equal to the sum of the estimated cost of its successor and the step cost (in the 8 or 15 puzzle problem, the step cost is always 1).

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* uses memory that increases exponentially. As it explores the nodes, it saves them in in an explored array. This causes the complexity to increase exponentially. Solving 15-puzzle problem requires $O(b^{Ed})$ memory where b is the branching factor, d is the depth and E is error since we have constant cost, which is 1. There may be exponentially many nodes to explore which can make the space complexity an issue (saving all those nodes in explored causes us to use so much space).

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.**

   - Memory bounded algorithm has a space **complexity** of **O(bd)** where b is the branching factor and d is the depth. Since RBFS is a recursive function, we go to the depth of each branching factor without saving them in an explored array. Since A* keeps track of the explored nodes, it uses more memory. The memory of A* increases exponentially while RBFS uses linear space. This gives us an algorithm similar to A* with better space complexity.
   - RPFS is complete if infinity is sent as a starting f_limit. If when rbfs is first called with a low f_limit, it might not find the right solution since it won't be looking at

many states. Thus, we always first send infinity to our rbfs. This makes the algorithm **complete**.

- Similar to A*, RBFS is **optimal** depending on the heuristic function. When the algorithm is both admissible and consistent and has an efficient heuristic function, it is optimal. It is consistent when the estimated cost of a node is always less than or equal to the sum of estimated cost of its successor and step cost.

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.**

	Algorithm	Heuristic	Output	# States Explored	Time (ms)	Depth Solution is Found
puzzle1.txt	A*	h1	['R', 'R']	2	0.108	2
		h2	['R', 'R']	3	0.153	2
	Memory Bound	h1	['R', 'R']	2	0.164	2
		h2	['R', 'R']	2	0.155	2
puzzle2.txt	A*	h1	['U', 'R', 'D', 'D']	4	0.221	4
		h2	['U', 'R', 'D', 'D']	6	0.384	4
	Memory Bound	h1	['U', 'R', 'D', 'D']	4	0.263	4
		h2	['U', 'R', 'D', 'D']	4	0.272	4
puzzle3.txt	A*	h1	['L', 'U', 'U', 'R', 'D', 'L', 'D', 'R']	26	2.492	8
		h2	['L', 'U', 'U', 'R', 'D', 'L', 'D', 'R']	42	7.598	8
	Memory Bound	h1	['L', 'U', 'U', 'R', 'D', 'L', 'D', 'R']	60	2.615	8
		h2	['L', 'U', 'U', 'R', 'D', 'L', 'D', 'R']	206	7.88	8
puzzle4.txt	A*	h1	['L', 'L', 'D', 'R', 'R', 'R']	6	0.499	6
		h2	['L', 'L', 'D', 'R', 'R', 'R']	22	2.022	6
	Memory Bound	h1	['L', 'L', 'D', 'R', 'R', 'R']	6	0.565	6
		h2	['L', 'L', 'D', 'R', 'R', 'R']	68	3.713	6
puzzle5.txt	A*	h1	['R', 'D', 'R', 'U', 'L', 'D', 'R', 'R', 'D', 'D']	26	3.77	10
		h2	['R', 'D', 'R', 'U', 'L', 'D', 'R', 'R', 'D', 'D']	28	4.749	10
	Memory Bound	h1	['R', 'D', 'R', 'U', 'L', 'D', 'R', 'R', 'D', 'D']	38	2.612	10
		h2	['R', 'D', 'R', 'U', 'L', 'D', 'R', 'R', 'D', 'D']	39	2.523	10