Joshua Bicera

Professor Tang

CS 4200

07/23/2023

Project 1

Approach:

1. Data Structure: In this program I utilized two different data structures called PuzzleState and StateNode. The PuzzleState stores the current state of the puzzle board configuration along with the g(n) and h(n) values. The StateNode holds a PuzzleState object along with information of the parent node and the children nodes. With this approach I have two separate classes where I can manipulate just the board by itself when I need to and a wrapper class that can handle the state of the board as a node separately.
2. Random Puzzle Generator: In addition to the manual input puzzle boards I implemented a random puzzle generator for testing which shuffles a list of values 0-8 in an array and converts that array into a puzzle board and then puzzle state object. To check whether or not the generated board is solvable I included a testing condition where the puzzle board constantly gets generated again until it passes a testing function called isSolvable to count the number of inversions.
3. A*Search: The A* search algorithm was implemented using a heap queue or priority queue to store all nodes to explore and set to store nodes that have already been explored.

This allows the algorithm to pop off the puzzle state with the smallest f(n) value to explore first before going onto other nodes and revisit the same states.

4. User Input: To get user inputs I implemented a function called getValidIntInput() which takes in a singular input where the function will check and ask again if the value is either not a digit or not within a specified range such as 1 and 2 for options 1 or 2. I also had a getUserPuzzle() function that allows users to input their own 8-puzzle board by typing a long string that the function will then separate into numbers. This also implements input validation as it uses getValidInput() as well as statements checking if the input string is the right size.
5. Heuristics: The two heuristic functions were implemented to analyze the board variable of the PuzzleState object which is a 2D array. The first heuristic function only checks misplaced tiles so it iterates through the 2D array and counts the amount of times the value of the tile and the tile in the goal state. While the first counts the amount of misplaced tiles, the second heuristic calculates the distance of the misplaced tiles from their correct positions.

Heuristic Comparison:

1. Results

	Average Search Cost (nodes)	Average Search Cost (nodes)	Runtime (milliseconds)	Runtime (milliseconds)
d (20 Cases each)	$A * (h_1)$	$A * (h_2)$	$A * (h_1)$	$A * (h_2)$
4	9.2	8.8	2.9	2.9
8	31.4	19.0	12.95	13.96
12	133.5	44.4	53.89	19.97

	Average Search Cost (nodes)	Average Search Cost (nodes)	Runtime (milliseconds)	Runtime (milliseconds)
16	663.0	100.2	448.8	62.83
20	4079.4	315.4	12,305.76	210.0

2. Analysis:

   From the table above, the use of heuristic functions 1 and 2 with A* search for the 8-puzzle problem is very effective as it solves complex puzzle boards in roughly 12 seconds. For the differences between the approaches, the first heuristic function for simple boards (d<6) is shown to have similar runtimes and average search costs as the second heuristic function. As the complexity increases, however, the search cost and the runtime becomes drastically less efficient as by depth of 20 the first heuristic function has 10 times more average search cost and nearly 60 times longer runtime. This shows that the simplistic and direct approach of the first heuristic is effective for simple solutions but does not scale as well as the second heuristic.