Noah Foster

ITCS 6150 Project 1 Report

Problem Formulation

The goal of this project is to create a program which will solve the 8 Puzzle Problem by implementing the A* search algorithm, while supporting two separate heuristics, and allowing user input of the puzzle to be solved. The puzzles themselves can be represented by 2D arrays, so I chose to design the program to work with those structures. The program needs to use hashmaps in order to stay efficient, as it needs to check for unsolvable problems by keeping track of the branches it has been down, and lists are too slow for this task. It also needs to use an efficient form of priority queue, such as a heap, so that it can choose from the many possible paths efficiently. I chose to write the program in Python, specifically Python3, which has a built-in dictionary structure which uses hashmapping, and a module called heapq that enables an efficient priority queue.

Operators (Functions and Classes)

Helper Functions:

To make working with 2D arrays easier, I created several functions for performing tasks relating to the arrays.

The first one, findNum, takes the array, and a number which you wish to find inside of it, and returns the index of said number. This function is used both in the findSwaps helper, and the manhattanDistance heuristic function.

The second helper, swapSpaces, takes an array, and two sets of indexes, copies the array, and swaps the values of the indexes in the new array. It goes ahead and makes a new array because we need to keep the old array for our use case.

The third helper, copy2dArray, was used in swapSpaces, and anywhere else we need to copy 2D arrays, and performs the job of copying not only the outside array, but also the inner arrays, so that we do not interfere with old arrays that are used in the storage of the path.

The fourth helper, printArray, is used to print out the 2D array in such a way that it looks similar to the way you would draw the puzzle. This function is used in the next helper, which prints the path array.

The fifth helper, printPath, is used to print out the path that the function finds for solving the puzzle. It takes an array of puzzle states and prints them with an arrow below every state but the last for user readability.

The sixth helper, findSwaps, takes the 2D array, and finds all the possible swaps. It does so by finding the 0 in the array, and uses simple knowledge of where in the puzzle it is to determine all the possible tile movements.

Heuristic Function Implementations:

The two Heuristic functions need to take the same two inputs, a puzzle array, and the goal array. This way, they can be easily substituted between the two final functions meant to use them. The outOfPlace function implements the Out of Place heuristic, while the manhattanDistance function implements the Sum of Manhattan Distances heuristic. The outOfPlace function simply counts the number of elements which are not equal between the two arrays. The manhattanDistance function uses the helper findNum to find each number in the goal array and count the Manhattan distance between the desired position and the current position. These heuristics are described in the ‘Heuristic Functions (h-cost)’ section of this report.

The Step Class

The Step Class is an Object used to store information about a given branch of the A* tree. It stores the array representing the position of the puzzle at that branch, the g-cost so far to get down that branch, and the path taken to get to that branch. It is important that we store the path information here so that when we get to the end of the program, we can print it for the user. G-cost is described in the ‘g-cost’ section of this report.

The A* implementation function

The A* implementation function, AStar, takes in the puzzle array, the goal array, and the desired h-cost function. For our purposes, it automatically prints either the path or “Solution Not Found” and the nodes generated and expanded counts. In order to find the solution, it uses several important data structures. It utilizes a heapq heap in order to store the frontier. This allows us to efficiently sort the frontier by the f-cost of each branch. It also utilizes a python dictionary to implement the ‘visited’ list as a hashmap. This allows the program to efficiently check if it has explored a possible arrangement of the puzzle before.

In the case of the 8 puzzle problem, and with admissible heuristics, A* will always find the best solution first. This means we just use a while loop to detect when we find a solution by checking the h-cost of the nodes we expand. If during the while loop, it detects that the frontier is empty, we have visited all possible rearrangements of the puzzle and the algorithm has failed to find a solution. In this case, we break the loop and print a message indicating that we could not find a solution. In the case that a solution is found, we use printPath to print the path to the solution, and print information on the number of nodes generated and expanded.

The User Interaction Code

I originally coded the program using an ipython notebook, which I included in the submission in case you were interested in quickly running the six input/output cases that I detail in the ‘Sample Input/Output Cases’ section of this report. This last portion of code makes the program usable by a user as a .py file that can accept any input puzzle. Since the program has no restrictions on the size of the arrays, it can technically run on an 8 puzzle-style puzzle of any size. Because of this, the user interaction code asks the user for the size of the puzzle, and then ensures that they input the correct size of array. It asks for the puzzle and the goal states by asking for each individual row of the puzzle and goal. It then asks whether they want to use the Out of Place Heuristic, Manhattan Distance Heuristic, or both, and runs the AStar function using the respective heuristic/s.

g-cost

g-cost in the 8 Puzzle Problem is simply the number of tiles shifted in reaching a solution. Because of this, we can just increment g-cost each time we place a new branch in the frontier and store it as a part of the node object that represents the branch of the A* tree. This is why g is a part of the ‘Step’ object defined earlier in the ‘Operators’ section of this report.

Heuristic Functions (h-cost)

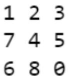
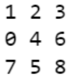
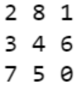
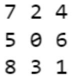
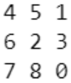
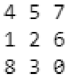
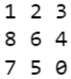
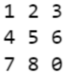
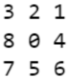
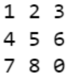
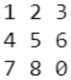
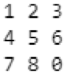
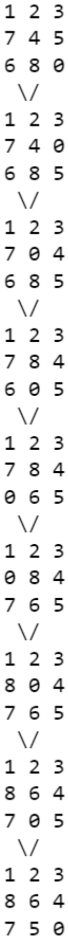
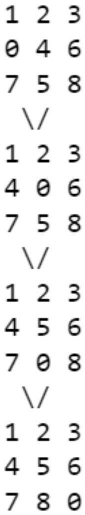
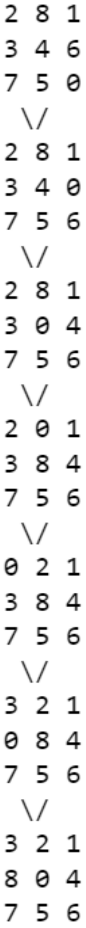
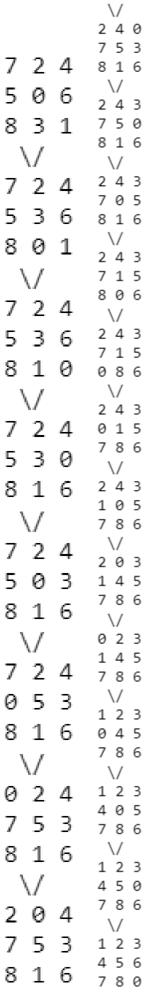
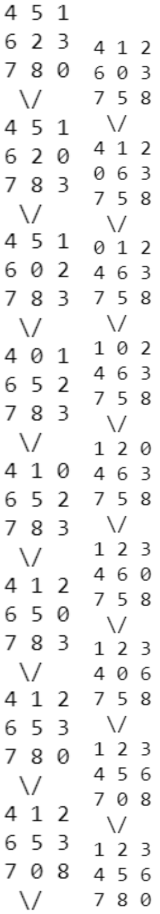
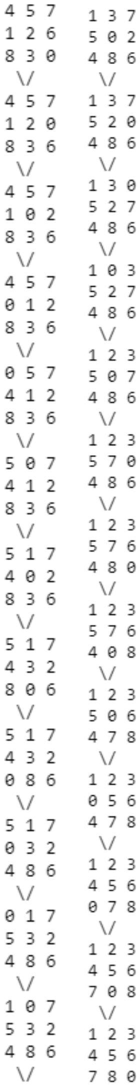
The h-cost in the A* puzzle is represented by the heuristic functions. The two heuristic functions that I used were the Out of Place heuristic and the Sum of Manhattan Distances heuristic. The Out of Place heuristic is simply the number of tiles which are not in the place that they are in the goal state. The Sum of Manhattan Distances heuristic is a bit more complicated. The Manhattan Distance of a tile is the number of spaces it is displaced horizontally plus the number of spaces it is displaced vertically. The Sum of Manhattan Distances heuristic, therefore, is the sum of the Manhattan distance of every tile in the puzzle.

Sample Input/Output Cases

The below tables summarize the results of my six test input/output cases by stating the puzzle, goal, heuristic used, Path to solution, nodes generated, and nodes expanded. I also tested the case of "No solution found" by inputting an unsolvable puzzle/goal combination. This combination was [[2,1,3],[4,5,6],[7,8,0]] with a goal of [[1,2,3],[4,5,6],[7,8,0]]. This puzzle is unsolvable as it uses an odd number of order inversions. In cases like these, the program always generates 181,439 nodes and expands 181,440 of them. This is because 181,440 is the number of ways you can arrange a puzzle only by sliding tiles, and the initial array is never generated by the program, leaving 'generated' as one lower than 'expanded'. In general, for the solvable cases, Manhattan distance finds the solution while generating and expanding fewer nodes than Out of Place. This is only false when the puzzles are exceptionally simple and well suited towards Out of Place, and then Manhattan Distance still generates and expands the same number of nodes as Out of Place.

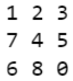
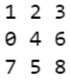
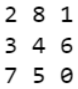
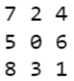
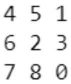
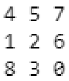
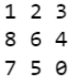
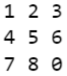
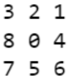
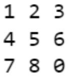
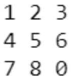
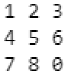
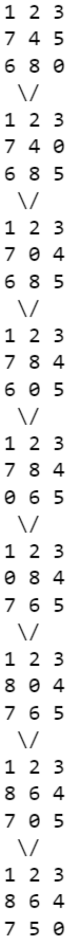
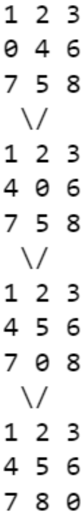
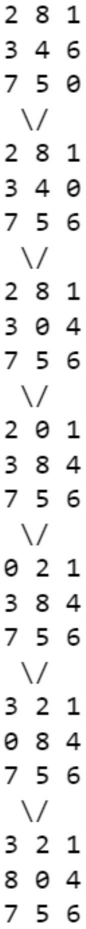
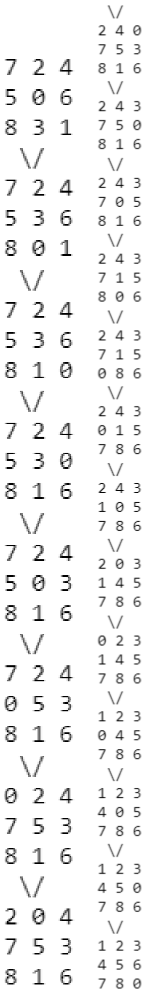
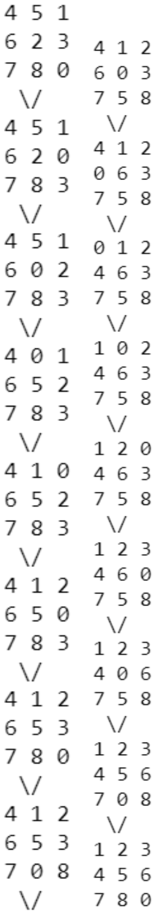
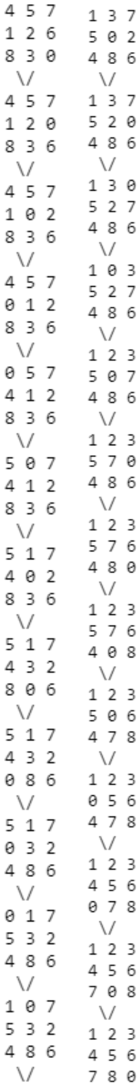
Noah Foster

Out of Place Heuristic

Puzzle	 1 2 3 7 4 5 6 8 0	 1 2 3 0 4 6 7 5 8	 2 8 1 3 4 6 7 5 0	 7 2 4 5 0 6 8 3 1	 4 5 1 6 2 3 7 8 0	 4 5 7 1 2 6 8 3 0
Goal	 1 2 3 8 6 4 7 5 0	 1 2 3 4 5 6 7 8 0	 3 2 1 8 0 4 7 5 6	 1 2 3 4 5 6 7 8 0	 1 2 3 4 5 6 7 8 0	 1 2 3 4 5 6 7 8 0
Heuristic	O.O.P	O.O.P	O.O.P	O.O.P	O.O.P	O.O.P
Path	 1 2 3 7 4 5 6 8 0 \/ 1 2 3 7 4 0 6 8 5 \/ 1 2 3 7 0 4 6 8 5 \/ 1 2 3 7 8 4 6 0 5 \/ 1 2 3 7 8 4 0 6 5 \/ 1 2 3 0 8 4 7 6 5 \/ 1 2 3 8 0 4 7 6 5 \/ 1 2 3 8 6 4 7 0 5 \/ 1 2 3 8 6 4 7 5 0	 1 2 3 0 4 6 7 5 8 \/ 1 2 3 4 0 6 7 5 8 \/ 1 2 3 4 5 6 7 0 8 \/ 1 2 3 4 5 6 7 8 0	 2 8 1 3 4 6 7 5 0 \/ 2 8 1 3 4 0 7 5 6 \/ 2 8 1 3 0 4 7 5 6 \/ 2 0 1 3 8 4 7 5 6 \/ 0 2 1 3 8 4 7 5 6 \/ 3 2 1 0 8 4 7 5 6 \/ 3 2 1 8 0 4 7 5 6	 7 2 4 5 0 6 8 3 1 \/ 7 2 4 5 3 6 8 0 1 \/ 7 2 4 5 3 6 8 1 0 \/ 7 2 4 5 3 0 8 1 6 \/ 7 2 4 5 0 3 8 1 6 \/ 7 2 4 0 5 3 8 1 6 \/ 0 2 4 7 5 3 8 1 6 \/ 2 0 4 7 5 3 8 1 6 \/ 2 4 0 7 5 3 8 1 6 \/ 2 4 3 7 5 0 8 1 6 \/ 2 4 3 7 0 5 8 1 6 \/ 2 4 3 7 1 5 8 0 6 \/ 2 4 3 7 1 5 0 8 6 \/ 2 4 3 0 1 5 7 8 6 \/ 2 4 3 1 0 5 7 8 6 \/ 2 0 3 1 4 5 7 8 6 \/ 0 2 3 1 4 5 7 8 6 \/ 1 2 3 0 4 5 7 8 6 \/ 1 2 3 4 0 5 7 8 6 \/ 1 2 3 4 5 0 7 8 6 \/ 1 2 3 4 5 6 7 8 0	 4 5 1 6 2 3 7 8 0 \/ 4 5 1 6 2 0 7 8 3 \/ 4 5 1 6 0 2 7 8 3 \/ 4 0 1 6 5 2 7 8 3 \/ 4 1 0 6 5 2 7 8 3 \/ 4 1 2 6 5 0 7 8 3 \/ 4 1 2 6 5 3 7 8 0 \/ 4 1 2 6 5 3 7 0 8 \/ 4 1 2 6 0 3 7 5 8 \/ 4 1 2 0 6 3 7 5 8 \/ 0 1 2 4 6 3 7 5 8 \/ 1 0 2 4 6 3 7 5 8 \/ 1 2 0 4 6 3 7 5 8 \/ 1 2 3 4 6 0 7 5 8 \/ 1 2 3 4 0 6 7 5 8 \/ 1 2 3 4 5 6 7 0 8 \/ 1 2 3 4 5 6 7 8 0	 4 5 7 1 2 6 8 3 0 \/ 4 5 7 1 2 0 8 3 6 \/ 4 5 7 1 0 2 8 3 6 \/ 4 5 7 0 1 2 8 3 6 \/ 0 5 7 4 1 2 8 3 6 \/ 5 0 7 4 1 2 8 3 6 \/ 5 1 7 4 0 2 8 3 6 \/ 5 1 7 4 3 2 8 0 6 \/ 5 1 7 4 3 2 0 8 6 \/ 5 1 7 0 3 2 4 8 6 \/ 0 1 7 5 3 2 4 8 6 \/ 1 0 7 5 3 2 4 8 6 \/ 1 3 7 5 0 2 4 8 6 \/ 1 3 7 5 2 0 4 8 6 \/ 1 3 0 5 2 7 4 8 6 \/ 1 0 3 5 2 7 4 8 6 \/ 1 2 3 5 0 7 4 8 6 \/ 1 2 3 5 7 0 4 8 6 \/ 1 2 3 5 7 6 4 8 0 \/ 1 2 3 5 7 6 4 0 8 \/ 1 2 3 5 0 6 4 7 8 \/ 1 2 3 0 5 6 4 7 8 \/ 1 2 3 4 5 6 0 7 8 \/ 1 2 3 4 5 6 7 0 8 \/ 1 2 3 4 5 6 7 8 0
Nodes Generated	44	9	17	5586	942	27597
Nodes Expanded	22	4	8	3813	584	18834

Noah Foster

Manhattan Distance Heuristic

Puzzle	1 2 3 7 4 5 6 8 0	1 2 3 0 4 6 7 5 8	2 8 1 3 4 6 7 5 0	7 2 4 5 0 6 8 3 1	4 5 1 6 2 3 7 8 0	4 5 7 1 2 6 8 3 0
Goal	1 2 3 8 6 4 7 5 0	1 2 3 4 5 6 7 8 0	3 2 1 8 0 4 7 5 6	1 2 3 4 5 6 7 8 0	1 2 3 4 5 6 7 8 0	1 2 3 4 5 6 7 8 0
Heuristic	M.D	M.D	M.D	M.D	M.D	M.D
Path	1 2 3 7 4 5 6 8 0 \/ 1 2 3 7 4 0 6 8 5 \/ 1 2 3 7 0 4 6 8 5 \/ 1 2 3 7 8 4 6 0 5 \/ 1 2 3 7 8 4 0 6 5 \/ 1 2 3 0 8 4 7 6 5 \/ 1 2 3 8 0 4 7 6 5 \/ 1 2 3 8 6 4 7 0 5 \/ 1 2 3 8 6 4 7 5 0	1 2 3 0 4 6 7 5 8 \/ 1 2 3 4 0 6 7 5 8 \/ 1 2 3 4 5 6 7 0 8 \/ 1 2 3 4 5 6 7 8 0	2 8 1 3 4 6 7 5 0 \/ 2 8 1 3 4 0 7 5 6 \/ 2 8 1 3 0 4 7 5 6 \/ 2 0 1 3 8 4 7 5 6 \/ 0 2 1 3 8 4 7 5 6 \/ 3 2 1 0 8 4 7 5 6 \/ 3 2 1 8 0 4 7 5 6	7 2 4 5 0 6 8 3 1 \/ 7 2 4 5 3 6 8 0 1 \/ 7 2 4 5 3 6 8 1 0 \/ 7 2 4 5 3 0 8 1 6 \/ 7 2 4 5 0 3 8 1 6 \/ 7 2 4 0 5 3 8 1 6 \/ 0 2 4 7 5 3 8 1 6 \/ 2 0 4 7 5 3 8 1 6 \/ 2 4 0 7 5 3 8 1 6 \/ 2 4 3 7 5 0 8 1 6 \/ 2 4 3 7 0 5 8 1 6 \/ 2 4 3 7 1 5 8 0 6 \/ 2 4 3 7 1 5 0 8 6 \/ 2 4 3 0 1 5 7 8 6 \/ 2 4 3 1 0 5 7 8 6 \/ 2 0 3 1 4 5 7 8 6 \/ 0 2 3 1 4 5 7 8 6 \/ 1 2 3 0 4 5 7 8 6 \/ 1 2 3 4 0 5 7 8 6 \/ 1 2 3 4 5 0 7 8 6 \/ 1 2 3 4 5 6 7 8 0	4 5 1 6 2 3 7 8 0 \/ 4 5 1 6 2 0 7 8 3 \/ 4 5 1 6 0 2 7 8 3 \/ 4 0 1 6 5 2 7 8 3 \/ 4 1 0 6 5 2 7 8 3 \/ 4 1 2 6 5 0 7 8 3 \/ 4 1 2 6 5 3 7 8 0 \/ 4 1 2 6 5 3 7 0 8 \/ 4 1 2 6 0 3 7 5 8 \/ 4 1 2 0 6 3 7 5 8 \/ 0 1 2 4 6 3 7 5 8 \/ 1 0 2 4 6 3 7 5 8 \/ 1 2 0 4 6 3 7 5 8 \/ 1 2 3 4 6 0 7 5 8 \/ 1 2 3 4 0 6 7 5 8 \/ 1 2 3 4 5 6 7 0 8 \/ 1 2 3 4 5 6 7 8 0	4 5 7 1 2 6 8 3 0 \/ 4 5 7 1 2 0 8 3 6 \/ 4 5 7 1 0 2 8 3 6 \/ 4 5 7 0 1 2 8 3 6 \/ 0 5 7 4 1 2 8 3 6 \/ 5 0 7 4 1 2 8 3 6 \/ 5 1 7 4 0 2 8 3 6 \/ 5 1 7 4 3 2 8 0 6 \/ 5 1 7 4 3 2 0 8 6 \/ 5 1 7 0 3 2 4 8 6 \/ 0 1 7 5 3 2 4 8 6 \/ 1 0 7 5 3 2 4 8 6 \/ 1 3 7 5 0 2 4 8 6 \/ 1 3 7 5 2 0 4 8 6 \/ 1 3 0 5 2 7 4 8 6 \/ 1 0 3 5 2 7 4 8 6 \/ 1 2 3 5 0 7 4 8 6 \/ 1 2 3 5 7 0 4 8 6 \/ 1 2 3 5 7 6 4 8 0 \/ 1 2 3 5 7 6 4 0 8 \/ 1 2 3 5 0 6 4 7 8 \/ 1 2 3 0 5 6 4 7 8 \/ 1 2 3 4 5 6 0 7 8 \/ 1 2 3 4 5 6 7 0 8 \/ 1 2 3 4 5 6 7 8 0
Nodes Generated	19	9	15	452	230	3587
Nodes Expanded	10	4	7	283	137	2317