AI Solver Project

March 2022

1 Implement a heuristic search algorithm: A* and the 8-puzzle game

1.1 Describe how you would frame the 8-puzzle problem as a search problem.

Search Problems is the problem that requires finding the correct path from the start state to the goal state with minimal costs. To clearly represent an 8-puzzle game we need to define what properties this game has:

- States - lists of possible location of tiles/possible combination of puzzle
- Operators - moving right, left, up or down
- Goal test - from 1 to 8 (right to left & up to down)
- Path cost - 1 per move

Therefore I can say that an 8-puzzle game is a search problem where we construct possible movements (possible boards/grids of the puzzle) in a tree format (a tree that expands down with all possible moves, where the boards/grids are nodes). By using heuristic functions with the A* Algorithm we decide what is the best solution and constantly move towards the goal state.

1.2 Solve the 8-puzzle problem using A*.

1.2.1 In this question, you should first briefly outline the A* algorithm.

The A* is a search algorithm used in path findings. It is a combination of uniform cost search and best first search, which avoids expanding expensive paths. A* star uses admissible heuristics which is optimal as it never overestimates the path to goal. The evaluation function A* star uses for calculating distance is *f(n) = g(n) + h(n)* where g(n) = cost so far to reach n / h(n) = estimated cost from n to goal /f(n) = estimated total cost of path through n to goal.

1.2.2 Describe two admissible heuristic functions for the 8-puzzle problem...

The two admissible (because it will always find an optimal solution, and would not overestimate the cost of reaching a goal) heuristic functions that I have used to solve 8-puzzle problem are:

- *Manhattan Distance* - distance between 2 tiles measured along the axes of right angles. It is the sum of absolute values of differences between goal state (i, j) coordinates and current state (l, m) coordinates respectively, i.e. |i - l| + |j - m|
- *Misplaced Tiles* - how many tiles are in the goal position.

The reason behind this selection is to clearly see how efficiency varies in between relatively difficult (Manhattan Distance) and easy (Misplaced Tiles) methods of solving the 8 puzzle. Both of them are admissible heuristic that will always result in an optimal solution. The reason behind that is that the structure of an 8-puzzle game (3x3 matrix) is suitable for both heuristic and can follow the steps into finding the goal state. Other admissible heuristic that I found would not suit the algorithm (not Euclidean Distance due to the implementation of the puzzle itself require step by step actions, and not Chebyshev Distance because 0 can not move diagonally in the puzzle)

1.2.3 Then, you should implement two versions of the A* algorithm in Python...

I have implemented two versions of the A* algorithm Manhattan distances and Misplaced tiles, which can be chosen in the program, by imputing the related integer. The goal states that I have used is the same as the Specifications, which is [0,1,2,3,4,5,6,7,8]. But I have used [4,3,1,6,7,2,0,8,5] as the starting position. The are 11 steps until solution (start included), which the program prints out one-by-one. The algorithm is not done completely, it *only solves puzzles which do not require repeating steps.* It is a serious drawback, but I'm not sure how to resolve this issue.

1.2.4 Briefly discuss and compare the results given by A*...

In my case both of my algorithms gave the same number of generations, meaning the same path of solutions. Start h(n) was *12 for Manhattan Distance* and *8 for Misplaced Tiles.* In terms of time on most runs the Misplaced Tiles heuristic was better then Manhattan Distance by approximately *0.0002±0.00015 seconds.* I believe that due to the size of the problem Misplaced Tiles calculates it faster, since it requires less calculations in the code, but if the problem was bigger than Manhattan Distance would solve it quicker.

1.3 General solution of the 8-puzzle using A*.

I have imported the solve function from previous exercise into a new file, but also added an additional checker for the grids (since not all grids can be used to reach the goal state). It is done by finding the number of inversions, if the number is odd then the program will not be computed. It also offers user to enter the grids values one by one in the terminal to make it easier when formatting the integers. Works with Manhattan Distance heuristic.

2 Evolutionary Algorithm.

2.1 Design and implement the Sudoku problem using Evolutionary algorithm.

2.1.1 In first problem design your evolutionary...

- (a) The template grids will be copied into the code to make it easier. I have implemented a minor algorithm to solve Sudoku's to then used the solved version for comparison with the mutated Sudoku's. Every new evolution will produce the best grid with new mutations, until it will be identical to the solved Sudoku. Every best grid will be printed into the terminal to clearly see what has been changed and how mutation works.
- (b) The rules of Sudoku game will act as limitation, therefore no repeating integers should be in any square, row or column. The number of total mistakes will be recorded and the grids that have the least mistakes will be selected to continue evolving.
- (c) Crossover operator will take 2 best grids out of the population and mix their rows in a specified order, so that the integers of the original puzzle stay in the correct parts of the mutated puzzle.
- (d) Mutation will fill the base grid that we have with randomly generated integers.
- (e) The template grids provided will act as the first individual of the population, so that all the new mutations are going to be alike the first one. By using recursion the evolution function will select the best grid and make new population with it, until the Sudoku is solved.
- (f) 2 selected grids will be the grid with the less mistakes in it (overall mistakes). After that they will become one (using crossover) and the replaced with the new population that will have different mutation perks.
- (g) The perfect criteria will be 0 mistakes in the Sudoku puzzle, and then if this rule is obeyed then the program will finish with the solved Sudoku.

2.1.2 Then, you should implement the evolutionary algorithm in Python...

This program is slightly different from the pseudo code evolutionary algorithm, but I believe that still it is an algorithm, that evolved depending on the mutations, therefore is an evolutionary algorithm. Instead of making population just once it makes it every time for the new best grid, that continues to keep all the mutations from the past generations. It is an important feature because it will massively affect the results provided:

Time Performance Table in seconds

Grid $\searrow$ *Pop.Size*	10	100	1000	10000
Grid 1	$0.0993 \approx 0.1$	$0.2944 \approx 0.29$	$1.8030 \approx 1.8$	$15.7873 \approx 15.8$
Grid 2	$0.0935 \approx 0.09$	$0.2863 \approx 0.29$	$1.9699 \approx 2$	$13.9361 \approx 14$
Grid 3	$0.1254 \approx 0.13$	$0.3305 \approx 0.33$	$2.1228 \approx 2.1$	$15.2324 \approx 14.2$

Amount of Generation Performance Table

Grid $\searrow$ *Pop.Size*	10	100	1000	10000
Grid 1	$18{,}8 \approx 19gen$	$9.4 \approx 9gen$	$8.2 \approx 8gen$	$7.4 \approx 7gen$
Grid 2	$15{,}4 \approx 15gen$	$9.2 \approx 9gen$	$8.2 \approx 8gen$	$6.4 \approx 6gen$
Grid 3	16 gen	11 gen	$9.8 \approx 10gen$	$6.8 \approx 7gen$

Overall Simplified Performance Table

Grid $\searrow$ *Pop.Size*	Overall Average Time	Overall Average Generations
Grid 1	4.497 sec	10.95 gen
Grid 2	4.095 sec	9.5 gen
Grid 3	4.19 sec	10.9 gen

2.2 Analysis the Sudoku problem using Evolutionary algorithm.

- Population 10 in terms of time performance and population 10000 in terms of number generations
- The reason behind this result is the algorithm design that I have used. Since it uses recursion and makes new population each time it's running, the compilation time of the program will increase with larger population. In terms of generations, bigger populations provide better mutations, therefore when choosing the new best grid my algorithm has more variety, meaning better mutations and less generations to go through
- Easiest - grid 2; Hardest - grid 1
- In both tables the easiest to solve was grid number 2, and I believe the main reason for that is the grid has the most integers filled in at the start - 33 (out of 3 grids). From the overall result of the data that I have

collected we can see that in terms of time and generation the first grid was the hardest one to solve, but the 3rd grid was quite close to being the hardest.

- Try to solve using a different approach (for example: trying solving Sudoku without recursion, or solving it without comparison to the solved Sudoku), to than compare the results and choose what perforce in a better way. This will help find the optimal evolutionary algorithm for solving a Sudoku puzzle. Another experiment that can take place is to let the user play with it and see how it will react when solving different Sudoku puzzles. That will bring up any errors that may be hidden.