A* Implementation of 8-Puzzle Problem

Question 1.1

The 8-puzzle can be represented in a graph and solved by searching the graph. Each node in the graph contains the current tile positions of the puzzle and has up to four child nodes (moving the blank tile left, right, up, down). The graph is built by generating the children of nodes in the graph, and once the graph contains a node containing the goal state, the graph can be searched for a path from the root node (the starting position of the tiles) to the goal state. Therefore, giving a sequence of moves in order to solve the puzzle.

States: Integer location of the tiles, with the blank tile represented as a '_'

Operators: The blank tile can be moved Left, Right, Up, Down

Goal test: The goal state given

Path cost: 1 per move

Question 1.2

1.

The A* algorithm is a search algorithm that finds the shortest path between the start state and goal state. This is done with 3 values:

g : the cost of moving from the initial state to the current state.

h : a heuristic value, which is an estimate of the cost to get from the current state to the goal state. This should always be generated by an admissible heuristic function, meaning the generated value is always an underestimate. An admissible heuristic results in the final solution being optimal.

f : the sum of the values of g and h for the current state

The A* algorithm begins at the start node and considers all adjacent nodes in the graph. The adjacent node with the smallest f value is picked and its child nodes are created and added to the graph. Now all the unvisited nodes are checked again, the smallest f value picked, and this process repeats until the node picked is the goal. In order to avoid an infinite loop between nodes, the algorithm also makes sure a node with the same value (e.g. same layout of tiles in the 8-puzzle) is never visited twice.

2.

Manhattan Distance - This is the sum of the distances of each tile from its goal position. This is admissible as the Manhattan distance for each tile is the minimum number of moves required for the tile to get from its current position to its goal position. It ignores the presence of other tiles that get in the way for this minimum number of moves to be possible so is therefore always an underestimate, making it an admissible function.

Misplaces Tiles – The number of tiles not in the same position as the goal state. This is admissible as each out of place tile must move at least once in order to get to the goal state.

I chose these two particular heuristic functions as they have different ranges of possible values. Misplaced tiles had a heuristic value of integers between 0 and 8 inclusive, whereas Manhattan distance has a much larger range of possible values as each tile can be up to 4 Manhattan distance units away. I hope this will result in different performance of the two algorithms.

3.

Prerequisites – python must be installed as well as NumPy. This can be installed in command line with:

```
pip install numpy
```

The program can then be run by entering the directory the program is in and entering the following into the command line:

```
python 8-puzzle.py
```

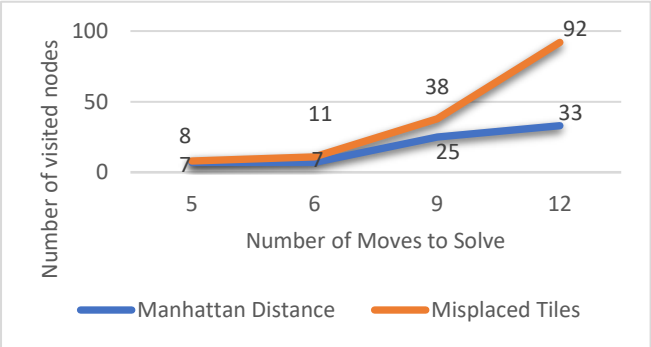
The starting grid in this program takes 17 moves to complete:

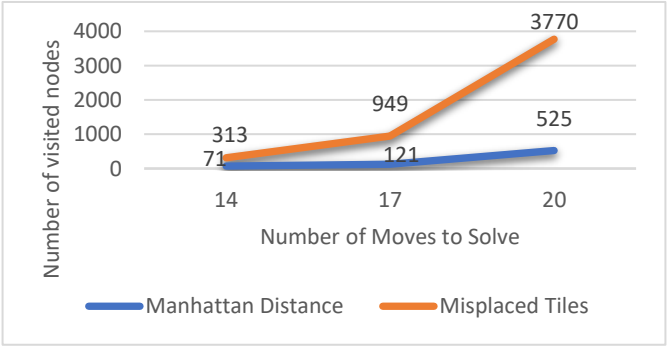
Start grid:	Goal Grid:
4 1 2	1 2 3
_ 8 7	4 5 6
6 3 5	7 8 _

The program will run and once complete show the moves needed to solve the puzzle from the starting grid to the goal grid entered into command line.

4.

The Manhattan distance and misplaced tile heuristics both end up in giving the same solution in the same number of steps, as this is the optimal solution. However, the Manhattan distance heuristic runs faster as it takes less iterations of selecting the next node. This results in less visited nodes and a smaller generated graph for the Manhattan distance compared to the misplaced tiles. The difference can be shown in the following graphs:





As seen in the graphs (split into two graphs for ease of viewing), misplaced tiles results in more visited nodes before finding a solution, and this gets worse with harder problems. For example, in my test data, a problem in which the solution requires 17 moves needed 121 nodes to be visited with the misplaced tiles heuristic and 949 visited with Manhattan distance. This may be due to the Manhattan distance having more of a range of values generated by the heuristic function, whereas misplaced tiles the function is limited to an integer between 0 and 8, so many nodes may share the same heuristic value, making it harder to find the correct path.

Problems used in graphs:

Number of moves to solve	Start grid	End grid
5	1,2,3,8,_,4,7,6,5	1,3,4,8,6,2,7,_,5
6	1,3,4,8,_,5,7,2,6	1,2,3,8,_,4,7,6,5
9	1,2,3,8,_,4,7,6,5	2,8,1,_,4,3,7,6,5
12	3,2,8,4,5,1,6,7,_	_,1,2,3,4,5,6,7,8
14	2,3,1,7,_,8,6,5,4	1,2,3,8,_,4,7,6,5
17	4,1,2,_,8,7,6,3,5	1,2,3,4,5,6,7,8,_
20	2,6,1,7,8,3,5,4,_	1,2,3,4,5,6,7,8,_

Note: the example puzzle given in the coursework specification takes 26 moves and visits 4357 nodes with the Manhattan distance but takes too long to run for misplaced tiles.

Question 1.3

Code for general implementation using the Manhattan Distance heuristic is in 8-PuzzleGeneral.py. To run the program, enter the following into command line:

```
python 8-puzzleGeneral.py
```

When prompted enter the start and goal grid in the following format:

If the starting grid is:

Then enter 7,2,4,5,_,6,8,3,1 when prompted by the program.

Note: More complex problems may take a while to run. Example solutions and their complexity are given above in Q 1.2.4.

The program will run and once complete show the moves needed to solve the puzzle from the starting grid to the goal grid entered into command line.

There are some pairs that are unsolvable. However, the code manages this by checking no configuration is used twice in the graph and once every possible branch has been tried (the open list becomes empty), the code will exit the loop and display to the user that the problem is unsolvable.

Evolutionary Algorithm to Solve Sudoku

Question 2.1.1

In my implementation, I chose to have a solution space in which each current layout of the grid in the population is represented as a list of 81. I chose this over a 2d array as it is faster to iterate through so made my code much faster for the larger populations. The solution space I chose is the grid list with the best fitness in the population for each generation, which is a positive integer up to 243 (this is explained later). A general solution is the grid with the best fitness in the population, and an optimal solution is one with a fitness of 243.

In my implementation, I chose a fitness function that counts each unique number for each row, column, and sub grid of 9. This will end in a number between 1 and 9 for each part, with 9 being where every number is different in each section, and 1 if each section only has one number repeated 9 times. Since there are 9 rows, 9 columns and 9 sub grids, the maximum value is 243 (3x9x9), and this occurs when the sudoku board is correct (no repeating numbers in any row, column, or sub grid).

To initialise my population, I chose to fill in each row with unique numbers from 1-9, without overwriting the starting values in the grid. This makes the algorithm start with all rows to be correct (no repeating values) and decreases the search space needed to solve the puzzle, allowing it to be solved faster. I repeat this until I have filled in a grid for each member of the population size.

From here, I split the population into 4 categories based on their fitness values. The first being the top 10%, second being the next best 20%, then the next best 30% and finally the worst 40%.

40% of the previous generation make it through the next generation and these are chosen by first adding all those in the first category (top 10%). From here, it then chooses the rest, with nodes in the second category having a 50% chance of being selected, nodes in the third category having a 30% chance of being selected and the fourth category having a 20% chance. This makes sure that the best nodes in the population always remain, and the other selected nodes tend to be those with the higher fitness, but also allowing some worse nodes through to maintain population variety so it does not converge too fast at a local maximum.

A crossover function is then used to rebuild the population to its initial size from the nodes that were selected in the previous generation. To maintain population variety, I used three crossover functions that all have the same probability of occurring (only one used at a time though). They are all row based as the population is initialised with rows containing all numbers from 1-9, so by only using row-based functions the rows stay correct. These three functions take two nodes that remained in the population from the previous generation and performs one of the following:

- Select a random row and swap them from the two nodes to create two new child nodes.
- Select a random row and swap this row and all rows below it with the other node, again creating two child nodes (e.g., if 4 is selected, rows 4,5,6,7,8,&9 will be swapped between the two parents).
- Go through each row and randomly decide whether to swap it with the other node or not, generating 2 child nodes.

This is repeated until the total population reaches the selected population size.

In addition, for each new node created, there is a 70% chance of a mutation. This is purposely set high as the mutation does not change much in the grid and through experimentation led to a correct

solution in less generations. The mutation function chosen is again row based to preserve the rows containing all numbers from 1-9 and picks two numbers in a single row at random and swaps them (cannot swap those already in the starting puzzle).

The algorithm terminates when a node is generated that has a fitness of 243, as this occurs when there are no repetitions of a number in each row, column, or sub grid of 9. This is when a sudoku board is completed correctly.

Question 2.1.2

Code is in sodoku.py, and a video of how to run the program is also provided, called running_sodoku.

In order to compare the results of each population size and starting puzzle, I have chosen to run the program for 1 minute best fitness function is in the population is for each population and grid combination. This is because some populations get stuck more easily at local maxima and never fully converge to the optimal solution.

The program prints the best fitness for each generation, with the generation number.

Tables showing best fitness in population after 1 minute (best fitness is 243, when solved):

Grid1:

POPULATION SIZE:	RUN 1	RUN 2	RUN 3	RUN 4	RUN 5	AVERAGE
10	236	239	237	235	238	237
100	236	237	241	241	237	238.4
1000	241	239	241	241	241	240.6
10,000	243	241	243	241	243	242.2

Grid2:

POPULATION SIZE:	RUN 1	RUN 2	RUN 3	RUN 4	RUN 5	AVERAGE
10	239	239	237	237	240	238.4
100	241	237	240	241	239	239.6
1000	240	241	241	243	241	241.2
10,000	243	243	243	243	243	243

Grid3:

POPULATION SIZE:	RUN 1	RUN 2	RUN 3	RUN 4	RUN 5	AVERAGE
10	233	235	235	235	237	235
100	235	236	237	236	237	236.2
1000	235	339	237	237	235	236.6
10,000	237	239	239	237	237	237.8

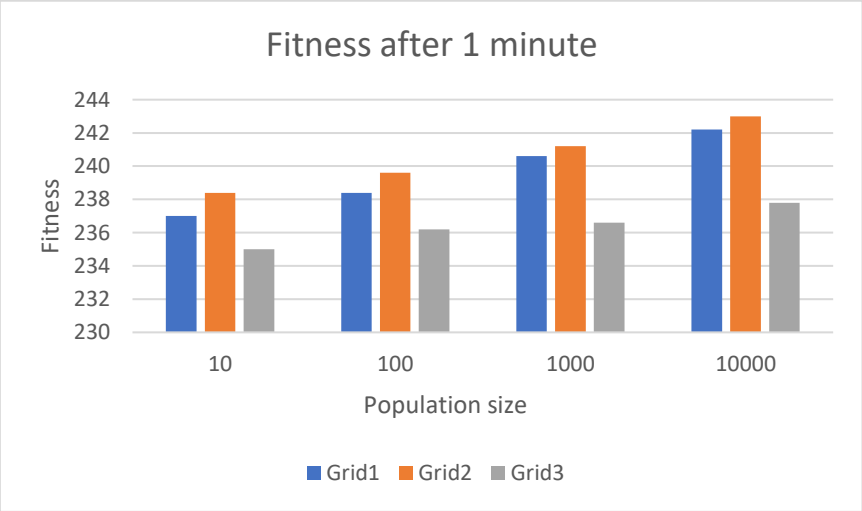
Furthermore, to allow for a deeper analysis, I also compared the time taken to get a complete solution for a population of 10,000, as this performed the best when in the previously shown tests.

Time taken to fully solve the grid with a population of 10,000:

GRID	RUN 1	RUN 2	RUN 3	RUN 4	RUN 5	AVERAGE
1	50.94s	67.92s	60.30s	49.99s	55.79s	56.99s
2	41.33s	45.16s	48.58s	42.45s	45.09s	44.52s
3	370.72s	350.32s	403.43s	547.23s	337.86s	401.91s

Question 2.2

1)



The best population size for the algorithm in my program is 10,000. This is shown in the first 3 tables and the graph above, in which the 10,000 population has the highest average fitness for all three grids after one minute. Furthermore, 10,000 often reached the optimal solution (fully correct sudoku board), where other population sizes struggled, converging at a local maximum, with this being much less common with the 10,000 population.

2) 10,000 running the fastest and overcoming local maximums is likely due the larger population allowing for more variance in the population. This is because more of the last generation continue into the next (in this case, 4000 continue to the next with 10000 population, but only 400 with the population of 1000). More variance in the population allows for the algorithm to have more options to search and does not narrow down its options too early, leading to premature convergence at a local maximum.

3) Grid 2 was the easiest to solve, having the highest averages across the board for all populations (as shown in the tables in question 2.and fastest time to solve with a population of 10,000. Grid 3 was the hardest to solve, with the lowest average fitness score for all 4 populations, as well as the longest time to solve in the 10,000 population.

4) Grid 2 being the easiest, and Grid 3 being the hardest to solve is most likely to the number of cells prefilled in for the puzzle, with the three grids having the following number of filled in cells:

Grid1 – 29 Grid2 – 33 Grid3 – 23

With more prefilled in cells, the algorithm has a smaller search space, as it has less cells to fill in and change with the crossover and mutation functions, resulting in less generations required to get a solution, and in turn leading to a faster running algorithm.

5) I tested my program with many different variations of mutation rate, selection processes, number of nodes to continue to next generation and different mutation functions. The algorithm defined in question 2.1.1 is the variation that converged to the optimal solution fastest in my testing. However, with more time I would like to experiment with using a hybrid algorithm. This is because evolutionary algorithms are good at getting a close to the solution, but struggle with prematurely converging at a local maximum. The use of a hybrid algorithm in which a genetic evolutionary algorithm is used to get close to a solution, and then a second algorithm such as an A* algorithm is used from this point to get to the optimal solution.

   In addition, I would like to experiment with different ways to initialise the population, such as instead of filling the rows with correct permutations, filling in each sub-grid of 9 with no repeats. This may help in puzzles where there are some rows with lots of prefilled cells and others with little to no prefilled cells as the sub-grids span multiple rows. These could be ran at the same time so the program can deal with different types of puzzles.