

# Artificial Intelligence and Applications

## ECM2423 Coursework

2022

### Contents

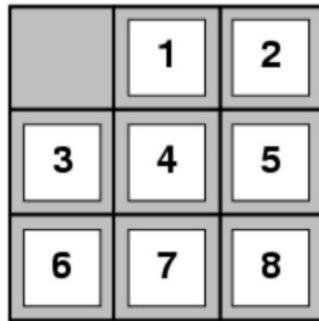
<b>1</b>	<b>Question 1</b>	<b>2</b>
1.1	Framing the 8-Puzzle Problem as a Search Problem . . . . .	2
1.2	Solving the 8-Puzzle using A* . . . . .	3
1.2.1	Outline of the A* Algorithm . . . . .	3
1.2.2	Heuristic Functions . . . . .	3
1.2.3	Implementation . . . . .	4
1.2.4	Comparison of the two Heuristics . . . . .	4
1.3	Generic Solution for the 8-Puzzle using A* Algorithm . . . . .	6
<b>2</b>	<b>Question 2</b>	<b>7</b>
2.1	Creating an Evolutionary Algorithm . . . . .	7
2.1.1	Design . . . . .	7
2.1.2	Implementation . . . . .	8
2.2	Analysis . . . . .	18
2.2.1	Population Size . . . . .	18
2.2.2	Solving the Grids . . . . .	20
2.2.3	Extra Experimentation . . . . .	22

# 1 Question 1

## 1.1 Framing the 8-Puzzle Problem as a Search Problem

### Context of the 8-Puzzle

The 8-Puzzle is a game, where the objective is to reorder the tiles in the board to resemble the goal. This is achieved by sliding tiles into an empty space to in effect move where the empty space is located on the board. Typically the goal for the 8 Puzzle looks like the following.



	1	2
3	4	5
6	7	8

Figure 1: Goal State

### Context of the 8-Puzzle

The objective of this project is to create an efficient search algorithm to find the solution to any initial state of the 8-Puzzle. To do this we will be using the A\* algorithm and two different admissible heuristic functions which will produce the best first solution to the puzzle.

### 8-Puzzle as a State Space Search

To view the problem as a search algorithm, we must view the problem as a graph and by solving the problem we are searching this graph. The state space of the problem is a directed graph of possible moves from each state to its children states and so forth. The nodes of the graph are each state (position of tiles) of the puzzle and the arcs of the graph are the actions and legal moves that can be made. These are moving the space: up, down, left or right depending on its starting position.

## 1.2 Solving the 8-Puzzle using A\*

### 1.2.1 Outline of the A\* Algorithm

The A\* search is the most widely known form of best-first search. It evaluates nodes by combining  $g(n)$ , the cost to reach the node, and  $f(n)$  the cost to get from the node to the goal.

$$f(n) = g(n) + h(n)$$

Since we evaluate the route through the nodes using the cost to reach the node and the cost to get from the node to the goal, it means that  $f(n)$  gives the cost of the cheapest solution through  $n$ . Therefore provided that the heuristic function is admissible and consistent, the A\* search is both complete and optimal.

For the heuristic to be admissible means that  $h(n)$  must never overestimate the cost to reach the goal. For the heuristic to be consistent (also known as monotonicity) means that for every node  $n$  and every successor  $n'$  of  $n$  generated by any action  $a$ , the estimated cost of reaching the goal from  $n$  is no greater than the step cost of getting to  $n'$  plus the estimated cost of reaching the goal from  $n'$ .

### 1.2.2 Heuristic Functions

Two admissible heuristic functions for the 8-Puzzle problem are Manhattan Distance and Misplaced Tiles.

The Misplaces Tiles heuristic function outputs the number of tiles which are not in the correct location. It does this by comparing each tile's location in the state to its location in the goal. If they are the same then the tile is in the correct location else the outputted value is increased by one.

I chose to use Misplaced Tiles as a heuristic function due to how it is a simple-to-implement heuristic algorithm which provides insight into how far away the state is from its goal.

The Manhattan Distance heuristic function outputs the sum of Manhattan Distances of each tile from its current position in the state to its position in the goal. The Manhattan Distance is the number of moves to get a tile from its position in the current state to its position in the goal.

I chose to use Manhattan Distance as a heuristic function due to how it is easy to relate the number of moves to get a tile into the correct location, to making moves sliding tiles in the puzzle. Furthermore, when calculating Manhattan Distance all the movements are either up, down, left or right. The same

as for when sliding a tile. So it is more relative than say using Euclidean Distance, which calculates using the length of a line segment between two vectors (a diagonal straight line) .

Manhattan Distance and Misplaced Tiles are both admissible heuristic functions because, as described above, for a heuristic to be admissible it must not overestimate the number of moves to solve the problem, in this case the 8-Puzzle and since you can only slide one tile at a time and in only one of four possible directions. The optimal scenario for each of these tiles is that it has a clear unobstructed path to its goal, or desired location in the board which would mean the move has a Manhattan Distance of 1. The rest of the states for a pair of tiles is sub-optimal, meaning that it will take more moves than 1 move (a greater Manhattan Distance than 1) to get the tile to the correct place. Therefore, the heuristic never over-estimates and is admissible.

### 1.2.3 Implementation

The following is an implementation of the A\* algorithm using the heuristic functions described above. The system uses the same initial board described in `/src/Q1/Standard/puzzle.txt` which can also be seen below in figure 2. The goal for the system is the same as described in figure 1.

7	2	4
5		6
8	3	1

Figure 2: Initial State for Standard

To run the program, navigate to the file directory `/src/Q1/Standard` and run the command `'py main.py'` in command prompt or your chosen terminal.

### 1.2.4 Comparison of the two Heuristics

Having completed both the implementation using the Manhattan Distance and the Misplaced Tiles we can compare the efficiency and performance of both methods.

Both heuristic functions find a path of moves from the initial state to the final state however the Manhattan Distance does this far quicker compared to

Misplaced Tiles. On Average it takes less than a second for the A\* Algorithm using Manhattan Distance to find the solution whereas when using Misplaced Tiles it takes on average between twenty and thirty seconds to find the same solution. This implies that as a heuristic function Manhattan Distance is a better heuristic function for finding the correct path to the goal state. We can explain this by looking at how both heuristic functions calculate their values.

We can see that Manhattan Distance is a better heuristic because with every misplaced tile in a state, that tile is at least one move away according to Manhattan Distance. Therefore the count of misplaced tile is always less than or equal to the sum of Manhattan distances. This means that for every tile in the state, unless the tile is in the correct position or only one move away from its correct position, the Manhattan distance is greater than one and therefore shows more detail on the how close to the goal state the current state is.

### 1.3 Generic Solution for the 8-Puzzle using A\* Algorithm

The following is a general version of the A\* Algorithm to solve a generic version of the 8-Puzzle where the user can input any start and goal state. However, this system cannot solve any generic pair of configurations due to how it is possible to have unsolvable configurations.

By testing a number of different configurations, I have discovered that it is not possible to solve an instance of the 8-Puzzle where the number of pairs of tiles which are early is odd. For a pair of tiles to be early means that the values on the tiles are in reverse order of how they appear in the goal.

Take for example the initial state below on the left in figure 3 which is attempting to reach the same goal described in figure 1. As we can see the pairs (2, 1) and (8, 7) are early. Hence the number of early pairs is two (even) and therefore this initial state is solvable.

Similarly, the initial state below on the right in figure 3 which is once again attempting to reach the goal described in figure 1. As the pair (5, 4) are early the number of early pairs is one (odd) and therefore the initial state is unsolvable.

To account for this, when the user enters an initial state and goal state the algorithm carries out a check to ensure the given initial state is solvable.

	2	1
3	4	5
6	8	7

	1	2
3	5	4
6	7	8

Figure 3: A solvable (left) and unsolvable (right) Initial State

To run the program, navigate to the file directory `/src/Q1/Generic` and run the command `'py main.py'`. You will then be prompt to enter an initial state and a final state before the algorithm will start. These states should be entered with 3 digits on a row and a single space between each column. If the entered initial state is unsolvable then a error message will be displayed.

## 2 Question 2

### 2.1 Creating an Evolutionary Algorithm

#### 2.1.1 Design

##### Definitions

A Sudoku puzzle is where the objective is to fill a 9x9 grid with digits so that each column, each row and each of the nine 3x3 sub grids that compose the grid all contain the digits from 1 to 9, from an initially partially completed grid. A state is a filled Suduko board which is one of a number of boards within the population. A gene is a list of digits in a row which were not specified in the initial board and have been assigned either randomly when creating a population or by a genetic operator.

##### Design Process

**Solution Space and Representation:** The solution space of the evolutionary algorithm is the set of all possible genomes. In the case of a Sudoku board a genome is a array of integers ranging from 1 to 9. The Sudoku board is formed from nine of such arrays forming a two-dimensional array.

**Fitness Function:** The fitness function for the Evolutionary Algorithm will be a procedure which calculates the number of duplicates in every row, column and square in a board.

**Crossover Operator:** In the Evolutionary Algorithm, a crossover can take place at any point on a gene of a row. For a 9x9 Sudoku grid there are 8 possible places where crossover can take place. Once a crossover place is chosen, then the start of parent one up to this position is combined with the end of parent two from this point. This process is carried out for every row to produce a new child grid made from a combination of parent one and two's genes.

**Mutation Operator:** Mutation causes two elements of a sub grid to swap positions. When mutation occurs, a random number of the sub grids in a state are affected therefore it is possible for up to all nine sub grids to have elements swapped. All fixed and predetermined elements cannot be swapped out of position.

**Initialisation:** For the initial solution of the Sodoku grid, a random combination of the digits from 1 to 9 are filled in each gene. This means that when the grid is initially filled, as a minimum each row of the grid will contain no duplicate numbers. The values which were predefined are fixed and cannot be changed. This is done by randomly selecting a digit for each cell from a list of digits which are not in the row.

**Selection and Replacement:** The entire population is sorted based upon each state's fitness value calculated by the fitness function. This results in the population's fittest states being at the start of the list and the weakest being at the end. When selecting states to be parents, the fitter functions are favoured however there is a finite probability of any state being selected so there is a chance that even low ranked solutions can be chosen. These probability weights are calculated using the following formula.

$$\text{Probability of State Selection} = \frac{\text{State Fitness}}{\sum_{n=0}^{\text{population size}-1} \text{State}_n \text{ Fitness}}$$

**Termination Criterion:** For the evolution algorithm to terminate, it must either find a solution or reach the maximum number of restarts. For the algorithm to find a solutions, means that a state is found in a population that has a Fitness of 0, therefore it has no duplicates in any rows, columns or sub grids and the Sudoku board is solved. Otherwise if the algorithm reaches local minimums for the max number of restarts it is allowed then it will display the state with the best fitness found from any of the restarts and their respective generations.

### 2.1.2 Implementation

The following section is an implementation of the Evolutionary Algorithm in Python to solve the Sudoku problem. Using this implementation, I have run experiments for the three Sudoku boards specified (see figures 4, 5, 6 below) using population sizes of 10, 100, 1000 and 10000. Each experiment was run 5 times and the results of average performance are shown below. All of the experiments were run with the following parameters constant:

- MUTATION\_RATE = 0.025
- RESTART\_THRESHOLD = 100
- MAX\_RESTARTS = 5

I also investigated the affect of changing the selection rate. Therefore for each of the grids I have created three tables, each with the selection rate being either 0.2, 0.5 or 0.8. This has allowed me to analyse and see the effects of selection rate.

### Running the System

To run the system navigate to /src/Q2/ then run then run the command 'py main.py'. This will run the program with the parameters set in main.py as constants and using the filename that the user enters.



Population	Run	Best End Fitness	Average End Fitness	Solved	Comments
10	1	29	28.8	N	Would reach local minima very quickly as didn't take long for the population to be filled with all the same values.
	2	30			
	3	30			
	4	29			
	5	26			
100	1	15	14.4	N	By the time it has reached the end of the MAX_GENERATIONS (50) all states in the population have the same fitness.
	2	12			
	3	15			
	4	18			
	5	12			
1000	1	2	2.8	N	Reaches local minima within 20 generations of each restart where necessary. By end of each restart it reaches the stage where the best state in the population has the same fitness as the worst state.
	2	3			
	3	3			
	4	4			
	5	2			
10,000	1	0	0	Y	Solved all the boards without any need for restart and withing 25 generations. The algorithm was still slow to run, however solved everytime.
	2	0			
	3	0			
	4	0			
	5	0			

Figure 4: Results of investigating Grid1.ss using a selection rate of 20%.

Population	Run	Best End Fitness	Average End Fitness	Solved	Comments
10	1	30	27.2	N	The algorithm runs quickly as expected however was unsuccessful at solving any attempt at the sudoku board.
	2	20			
	3	27			
	4	31			
	5	29			
100	1	12	10.6	N	The algorithm ran quickly however reached a point where the fitness of all the values in the population were the same (local minima) and was unable to jump out.
	2	12			
	3	10			
	4	9			
	5	10			
1000	1	2	2.8	Y	The algorithm was able to solve the problem, it would often reach a local minima very close to the goal however then be unable to jump out but with restarts it was able to reach the goal.
	2	5			
	3	3			
	4	0			
	5	4			
10,000	1	0	0	Y	The algorithm initially increases in fitness value due to the large selection rate however after approx 9 generations it begins to linearly decrease as expected. I believe this is because due to the selection rate selecting more than just the optimum set of parent states from the previous generation. All the runs solved within 3 restarts. Very slow.
	2	0			
	3	0			
	4	0			
	5	0			

Figure 5: Results of investigating Grid1.ss using a selection rate of 50%.

Population	Run	Best End Fitness	Average End Fitness	Solved	Comments
10	1	28	28.4	N	By having a large selection rate it meant that there was no real linear movement increasing or decreasing the fitness of each generation and rather it fluctuated randomly around the initial fitness.
	2	27			
	3	29			
	4	31			
	5	27			
100	1	19	18.6	N	It was clear over a number of generations that the fitness of the populations was increasing however between specific generations this was not always clear. I believe this is due to how as 80% of the previous population could be used to make children, there was a much larger possibility of there being unfit parents used to reproduce.
	2	14			
	3	21			
	4	20			
	5	19			
1000	1	16	15.4	N	By using a higher select rate, the EA was able to decrease the general fitness and improve in general however was never able to solve or reach a fitness of under 14. I believe this is due to how the new children could be made from a greater range of states with worse fitnesses', hence the probability of having parents with a good fitness was less.
	2	14			
	3	15			
	4	16			
	5	16			
10,000	1	17	14	N	By using a higher select rate the program was not able to decrease enough to solve or even achieve a fitness under 12. From looking at the populations, even when the best state has a fitness of 12 the worst had a fitness of 46, so the average fitness was far too high to make it probable to ever reach a solution.
	2	13			
	3	15			
	4	12			
	5	13			

Figure 6: Results of investigating Grid1.ss using a selection rate of 80%.

Population	Run	Best End Fitness	Average End Fitness	Solved	Comments
10	1	19	16.8	N	With this small selection rate the fitness of the population didn't drastically change. I believe this is because a lot of the board is completed using predetermined values, and therefore the smaller population tends to contains similar states.
	2	17			
	3	17			
	4	16			
	5	15			
100	1	8	4.6	N	Similar to with a population of 10 it did not take long to reach a local minimum, where all the states in the population had the same value, but because of the low starting fitness of the board it was able to get reasonably close to solving.
	2	5			
	3	4			
	4	2			
	5	4			
1000	1	0	0	Y	This was the fasted population for this grid and was able to solve everytime within at most two restarts. The best fitness of the population was a lot smaller than the worst fitness for the same population.
	2	0			
	3	0			
	4	0			
	5	0			
10,000	1	0	0	Y	Unexpectedly the best fitness for each generation actually gets worse for the first few generations before rapidly decreasing to solve within the first twenty generations.
	2	0			
	3	0			
	4	0			
	5	0			

Figure 7: Results of investigating Grid2.ss using a selection rate of 20%.

Population	Run	0	Average End Fitness	Solved	Comments
10	1	11	15	N	After approx twenty five generations it reached a local minimum where the best and worst fitness in every generation were the same.
	2	17			
	3	16			
	4	18			
	5	13			
100	1	4	5.6	N	The best states always came close to solving but became stuck in a local minimum where all the states in multiple generations had the same fitness so was very unlikely to find the solution ever.
	2	3			
	3	3			
	4	4			
	5	4			
1000	1	0	0	Y	In every case apart from one the EA solved within 30 generations without any resets however in one case requires three resets.
	2	0			
	3	0			
	4	0			
	5	0			
10,000	1	0	0	Y	With the larger population, the first population had a better initial fitness however as with Grid1, it first had a slight increase in fitness before it started decreasing however would always solve within twenty generations.
	2	0			
	3	0			
	4	0			
	5	0			

Figure 8: Results of investigating Grid2.ss using a selection rate of 50%.

Population	Run	Best End Fitness	Average End Fitness	Solved	Comments
10	1	15	17.4	N	It did not take long for the restarts of each run to reach a local minima where all the fitness values were the same. Hence it restrected how low the values could get but, due to the low initial fitness they were able to get sun 20 values.
	2	17			
	3	19			
	4	19			
	5	17			
100	1	9	9.2	N	It was clear that this was a much more effective population size than 10 as it was rare for the generations to have populations with the same fitness values for all states but the decrease in fitness was in a weak negative correlation.
	2	9			
	3	9			
	4	11			
	5	8			
1000	1	7	8.2	N	As seen before with when using 0.8 selection rate there was not a clear decrease in fitness, the populations would sometimes increase as well as decrease but none the less in general there was a general decrease.
	2	10			
	3	8			
	4	9			
	5	7			
10,000	1	8	8.2	N	For the initial generations of every restart and run the fitness only increased in average for each state. Once again as seen before it was then unable to ceate optimal children so couldnt solve and the fitness did not decrease linearly rather in a more dynamic manor.
	2	7			
	3	6			
	4	7			
	5	13			

Figure 9: Results of investigating Grid2.ss using a selection rate of 80%.

Population	Run	Best End Fitness	Average End Fitness	Solved	Comments
10	1	41	37.4	N	The fitness did not typically make much progress on solving the grid, for this set of parameters the Evolutionary teded to have a minute effect on the fitness.
	2	40			
	3	35			
	4	35			
	5	36			
100	1	23	25	N	The larger population helped the EA to start making improvements to the fitness however did not take long to reach a local minimim where all the fitness of every state wre the same.
	2	26			
	3	28			
	4	25			
	5	23			
1000	1	9	8.4	N	The larger population worked much better with te selection rate and from the start there was a linear decrease in fitness before it would become stuck in a local minima and even with restards was unable to jump out.
	2	7			
	3	8			
	4	9			
	5	9			
10,000	1	9	6.6	N	Similar to other large populations with this selection rate, the initial generations are negatively affected and fitness increases before it then starts to linearly decrease but then it does not manage to solve.
	2	6			
	3	7			
	4	5			
	5	6			

Figure 10: Results of investigating Grid3.ss using a selection rate of 20%.

Population	Run	Best End Fitness	Average End Fitness	Solved	Comments
10	1	39	36.8	N	Once again when using such a small population, the EA is not very effective and the majority of the time the algorithm fluctuates around the initial fitness rather than decreasing.
	2	35			
	3	38			
	4	38			
	5	34			
100	1	22	21	N	The population is big enough that the EA has an effect and over the generations the fitness value decreases however it often gets caught in local minima.
	2	20			
	3	23			
	4	19			
	5	21			
1000	1	9	7.8	N	When running these tests there is a clear linear decrease in the fitness value however it appears to reach the termination criteria before solving or reaching a local minima. The rate of decrease is not enough to reach a solution in 50 generations.
	2	7			
	3	8			
	4	8			
	5	7			
10,000	1	11	10	N	Runs very slowly. The fitness value increases initially as the first generations are formed, before starting to decrease. This appears to be a pattern with this population size and selection rate.
	2	9			
	3	12			
	4	8			
	5	10			

Figure 11: Results of investigating Grid3.ss using a selection rate of 50%.



Population	Run	Best End Fitness	Average End Fitness	Solved	Comments
10	1	40	38	N	The fitness values initially got worse before becoming stuck in a local minima similar to the initial best fitness. This is similar to other population sizes with the other grid and selection weights.
	2	39			
	3	38			
	4	34			
	5	39			
100	1	31	31.8	N	In this test, the fitness values initially got worse then began to improve however the resolution of this was it only typically just got below the initial value fitness which it started with.
	2	33			
	3	35			
	4	29			
	5	31			
1000	1	33	30.2	N	With such a large selection rate the child generations typically did not always achieve improvements on their parents, therefore the functions struggled to decrease especially as the board did not have many predetermined.
	2	28			
	3	30			
	4	29			
	5	31			
10,000	1	15	14.4	N	Runs very slowly and does not perform expected from a large population. Due to the large population size and large selection rate there is no linear decrease in the fitness value and rather a fluctuation which slowly decreases over numerous generations.
	2	14			
	3	16			
	4	12			
	5	15			

Figure 12: Results of investigating Grid3.ss using a selection rate of 80%.

## 2.2 Analysis

Having developed the evolutionary Algorithm and run 180 tests using a variety of parameters, I am now able to analyse the algorithm in full and determine the following results.

### 2.2.1 Population Size

The Population size specified before running the evolutionary algorithm makes a huge difference to the probability of finding a solution as well as the time it requires to run each generation. When determining the best starting population size, from all of the data it is clear to see that when using a larger population, the best fitness produced is typically lower than a smaller population size. Also when the grid does solve, it does not need any restarts to find the solution (depending on the complexity of the puzzle). Taking figure 14 for example, (where selection rate is 0.5 hence centered to avoid bias towards any population size) it is clear to see that for all three grid files when the population is increased, the best fitness decreases until consequently a population of one thousand is sufficient to solve grid 2 and a population of ten thousand for grids 1 and 2.

It is important however to mention the results of changing the selection rate in relation to population size, as using a large population with a large selection rate does still perform better than a small population with a large selection rate but the performance is hindered drastically. As discussed in the results tables (see figures 4 - 12), a large selection rate combined with a large population seemed first to negatively affect the fitness before beginning to act as we expect from an Evolutionary Algorithm. I believe this is because when initialising the board, the algorithm ensures that as a minimum all the rows do not contain any duplicates, then when the new generations are made using crossover operators and mutation operators, as the selection rate is so high the rows become populated with duplicates and therefore until the Evolutionary Algorithm reaches a population where the entire selection percentage starts to improve the fitness values of the states do not decrease. This can be seen in figures 13 through 15 where as the selection rate increases no matter the population size, with a large selection rate the performance is drastically hindered.

The reason behind these results comes from how filling the initial board is a random process. Therefore when we only take a population size of 10, the probability of all ten states having good starting fitness, (but with enough variation so that when you produce children in each generation the fitness increases all while maintaining enough mutation to ensure all the states do not become the same) is very small. The maths behind this can be seen, so with a population of 10, we would expect 0.25 states to mutate, which in effect is 0. So this is why we reach a local minima so soon. Whereas when working with a population size of 10,000 the probability is far greater that there will be fitter states and also dramatically less fit states which can separate them apart to allow the best to be selected. This paired with how we would expect 250 mutations every generation

means that we are able to avoid the local minima and therefore, as we saw, the algorithm is very likely (depending on the initial board) to solve.

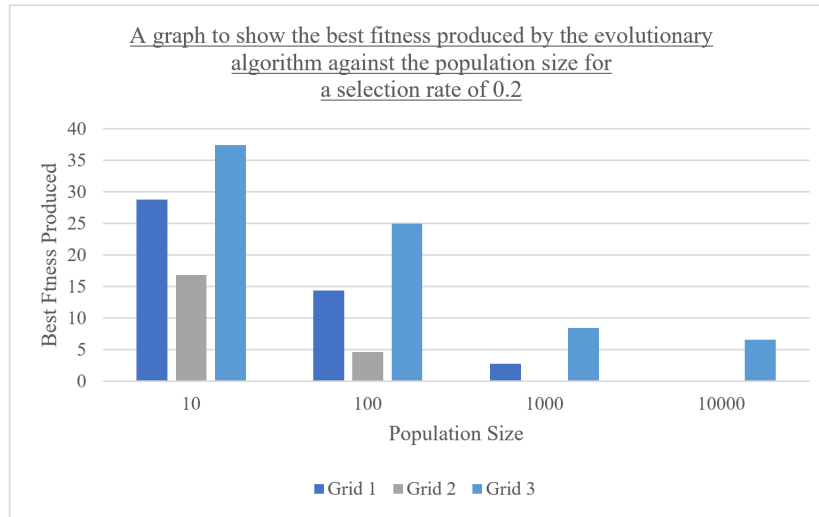


Figure 13: Graph displaying the performance results using a 0.2 selection rate.

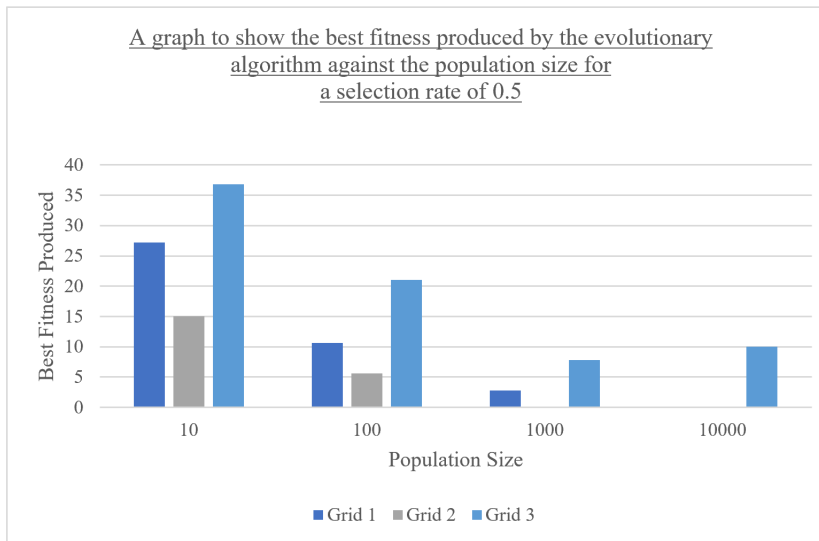


Figure 14: Graph displaying the performance results using a 0.5 selection rate.

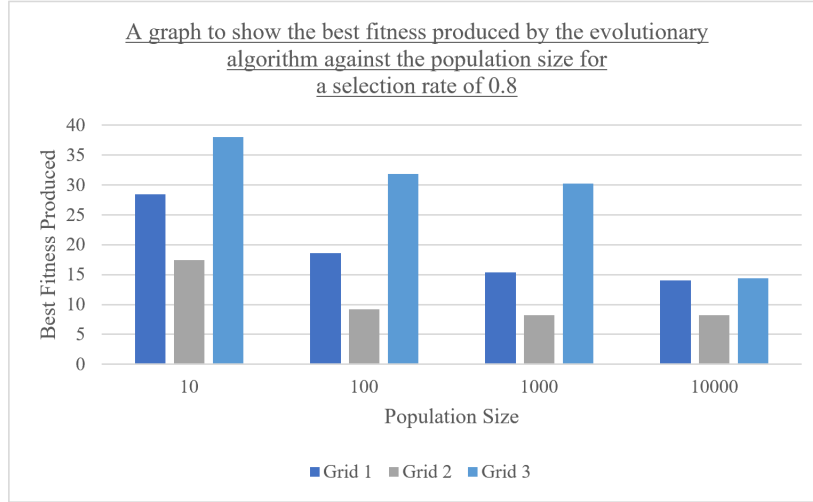


Figure 15: Graph displaying the performance results using a 0.8 selection rate.

### 2.2.2 Solving the Grids

Out of the three grid files given by the specification, my Evolutionary Algorithm was able to solve two of them and then came within a fitness value of 5 (5 duplications in the board for grid 3). It is clear which board was easiest to solve from the data in figure 15 above.

From figure 7 we can see that with Grid 2, the Evolutionary Algorithm was able to solve the grid every time for a population size of both 1000 and 10,000 and only needed at most 2 restarts for the smaller of the population sizes. Looking at figures 8 and 9 which describe the performance of the algorithm with larger selection rates, and solved when the selection rate was 0.5 in a similar way to 0.2 but with use of more restarts. It did not solve for a 0.8 selection rate but still performed better than the other graphs in a similar simulation. So to conclude this point Grid 2 was the easiest board to solve, with the initial state shown below in figure 16.

One key reason why Grid 2 was easier to solve is that it can be partially solved using predetermined values. As discussed earlier in my design process, before the Evolutionary Algorithm starts, the initial state is run through an algorithm which checks if any cells can be predetermined and if so fills the cell with the correct value. For Grid 2 the initial state has multiple cells filled in by this check and therefore the Evolutionary Algorithm is guaranteed to have a better fitness for each state in each generation. This can be seen in figure 16 which shows both the initial state and initial state after predetermined cells are filled.

The hardest grid to solve was Grid 3, which was never solved by the Evo-

lutionary Algorithm. Looking at figures 13, 14 and 15 we can see that even when the population increases the fitness value plateaus and does not seem to make any improvements. This is because, unlike Grid 2, the algorithm to fill any cells which can be predetermined was not successful in making any major developments so the evolutionary algorithm had to work from a largely unfilled initial grid. Figure 17 shows both the initial grid of Grid 3 and the Initial Grid once all predetermined values have been calculated.

		2				6	3	4
1		6				5	8	
		7	3			2	9	
	8	5			1			6
			7	5			2	3
		3					5	
3	1	4			2			
		9		8	4			
7	2			4				9

		2				6	3	4
1		6				5	8	7
		7	3	6		2	9	
	8	5			1	7	4	6
		1	7	5		9	2	3
		3				1	5	8
3	1	4			2	8		5
		9		8		4		2
7	2	8		4		3		9

Figure 16: The initial grid (left) and the initial grid with predetermined values (right) of Grid2.ss

		4		1			6	
9							3	
	5		7	9	6			
		2	5		4	9		
	8	3		6				
						6		7
			9		3		7	
		6					1	

		4		1			6	
9							3	
	5		7	9	6			
		2	5		4	9	8	
	8	3		6				
						6		7
			9		3		7	
		6					1	

Figure 17: The initial grid (left) and the initial grid with predetermined values (right) of Grid3.ss

### 2.2.3 Extra Experimentation

In the course of completing this coursework, I identified that additional experimentation into selection rates would be useful I have therefore carried out an investigation of selection rates and my findings and analysis of this are Incorporated into this report

The selection rate refers to the percentage of states from the previous generation that are use as a genetic pool for reproduction. Now the states are ordered based upon fitness value from lowest to highest so, a selection rate of 0.1 will only use the top 10% of states with the top 10% lowest fitness values whereas as selection rate of 0.8 will use the top 80% and therefore the top 80% of lowest fitness values. With a large selection rate the possibility of having not optimal parents increases but the possibility of a large change between generations also increases hence helps the algorithm to avoid becoming stuck in a local minima.

From my experimentation (shown in figures 4-12) we can conclude that, to achieve the optimal parameters to solve a Sudoku board, the selection rate needs to be inversely proportionate to the population size. Furthermore using a population size such a 10, you need to use a larger selection rate else the Evolutionary Algorithm reaches a minima very quickly and doesn't perform well decreasing the fitness value. With a large population such as 10,000 you need a small selection rate such as 0.1 else your gene pool contains too many states with a higher fitness than the optimal.