Design

The solution will be represented as a list containing the 9 'boxes' in a sudoku grid. Each box will also be represented as a list, although behave similar to a set, and will contain the numbers 1-9. All lists will be read from top-to-bottom and left-to-right. The solution space will therefore be a list containing 9 sets of the numbers 1-9 where their fitness value is 0.
This representation, of storing the grid as a list of its boxes, allows for the rows and columns to be extracted easier and therefore provide an efficient calculation of the fitness of each solution.

The fitness function will determine how many duplicates exist in each row and each column of the filled sudoku grid, a total of 18 checks. The boxes will not need to be checked as they will remain duplicate-free throughout their creation, mutation, and crossover.
The fitness value will be calculated by determining the number of unique elements in the list and subtracting that from 9 (the expected value). The final value will be the sum of the individual fitness values of each row and column. I chose to represent the fitness as such, compared to for example the number of conflicting pairs, as it is a much easier and simpler value to calculate and understand and should help to increase the speed of the algorithm.

The crossover operator for the chosen representation will create a child from two parents and the child will 'inherit' a box from either parent, chosen randomly. The random inheritance of blocks aims to introduce more randomness and remove as much potential bias from the results with the intention of diverting from a local minima. The crossover has been represented in this manner, instead of swapping rows/columns of boxes, as it increases the potential that adjacent blocks will be different, providing a greater chance to potentially move away from a bad solution (local minima).

The mutation operator for this chosen representation will involve swapping pairs of values within their respective box. This will prevent any duplicates from appearing within the box and ensure that there exist the correct values within the grid (i.e. 9 1's, 9 2's etc), as opposed to swapping an individual value with a different one (i.e. 1 becomes a 7). The mutation will also only swap elements that do not exist in the initial grid, ensuring that the known values remain in the same place throughout and greatly improve the efficiency of the algorithm.
Each box will have a 25% chance of mutating, resulting in 2 swaps, on average, for each solution. This mutation percentage ensures that each mutation of a solution will have a higher chance of producing a new permutation to be used as a future parent.

The population will be initialised by creating some number of individual random 'solutions'. This number (population size) and a file that contains the initial grid will be provided by the user. Each individual 'solution' will be created by first determining what values are missing from each 'box' before randomly assigning each value to a remaining position. This helps to reduce the number of possible permutations that can be created and therefore improve the overall efficiency of the algorithm.

The parent solutions are selected from the best 50% of solutions in the current population. Once the child solutions have been generated, the new population is generated from the best of the parent solutions and the best 50% of child solutions. The cutting of the child solutions aims to reduce the potential of encountering a local minima by allowing some worse solutions to exist where they would originally not, in the situations where a child solution would replace it.
Due to the large number of local minima that exist within a sudoku puzzle, it is important to retain some form of population diversity. In theory, this should also be accompanied by a slower convergence, to avoid the local minima, however in practice this greatly affected the efficiency of the algorithm. Through initial trials I had discovered that the less greedy selection and replacement, although better

avoiding local minima, took far longer to arrive at the desired solution compared to the greedier convergence in my algorithm, as I valued arriving at a correct solution faster a more desirable characteristic to aim for.

The program will terminate after a solution has been found or until it has reached 250 generations or until 5 minutes (300 seconds) has passed. The generation cap is to ensure that the program terminates if it does get stuck in some local minima, and the time cap is to prevent the larger populations from being in a local minima for those 250 generations. Earlier tests showed that, in those cases, this could last for up to 30 minutes. For a similar reason, the algorithm stops once a solution has been found to prevent excess computation.

Experiments

Grid 1

Population Size	Measurement	Run 1	Run 2	Run 3	Run 4	Run 5	Average
10	Number of Generations	58	81	99	53	39	66
	Best Fitness	0	0	0	0	0	0
	Time Taken /s	0.21	0.27	0.30	0.23	0.16	0.234
100	Number of Generations	16	15	14	11	16	14.4
	Best Fitness	0	0	0	0	0	0
	Time Taken /s	0.68	0.66	0.61	0.47	0.70	0.624
1000	Number of Generations	8	11	7	7	10	8.6
	Best Fitness	0	0	0	0	0	0
	Time Taken /s	4.06	5.14	3.32	3.34	4.91	4.154
10000	Number of Generations	4	5	5	9	7	6
	Best Fitness	0	0	0	0	0	0
	Time Taken /s	22.37	26.84	27.10	46.94	36.88	32.026

Grid 2

Population Size	Measurement	Run 1	Run 2	Run 3	Run 4	Run 5	Average
10	Number of Generations	250*	133	250*	250*	183	213.2
	Best Fitness	6	0	4	4	0	2.8
	Time Taken /s	0.70	0.40	0.66	0.71	0.53	0.660
100	Number of Generations	44	250*	40	105	48	97.4
	Best Fitness	0	2	0	0	0	0.4
	Time Taken /s	1.90	6.79	1.68	3.79	2.06	3.244
1000	Number of Generations	24	25	26	29	23	25.4
	Best Fitness	0	0	0	0	0	0
	Time Taken /s	10.99	11.31	12.18	13.72	10.52	11.744
10000	Number of Generations	19	22	21	22	22	21.2
	Best Fitness	0	0	0	0	0	0
	Time Taken /s	94.67	110.79	102.24	107.91	108.37	104.796

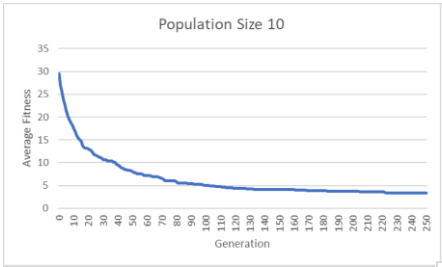
Grid 3

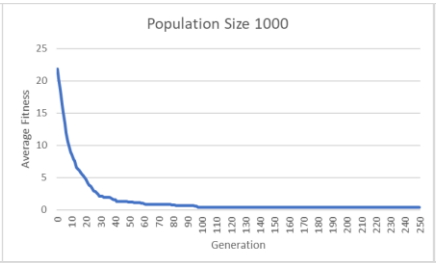
Population Size	Measurement	Run 1	Run 2	Run 3	Run 4	Run 5	Average
10	Number of Generations	250*	250*	250*	250*	250*	250*
	Best Fitness	6	7	6	7	11	7.4
	Time Taken /s	0.72	0.75	0.71	0.74	0.69	0.722
100	Number of Generations	250*	250*	250*	250*	250*	250*
	Best Fitness	4	7	2	2	4	3.8

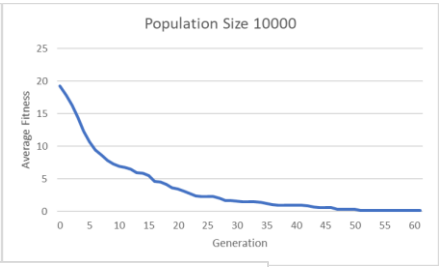
	Time Taken /s	8.49	6.98	7.05	8.95	7.35	7.764
1000	Number of Generations	38	250*	250*	97	61	139.2
	Best Fitness	0	2	4	0	0	1.2
	Time Taken /s	17.16	85.90	93.87	43.11	27.84	53.576
10000	Number of Generations	44	61	51	43	47	42.6
	Best Fitness	0	2	0	0	0	0.4
	Time Taken /s	222.10	304.22**	300.15**	244.41	265.71	267.318

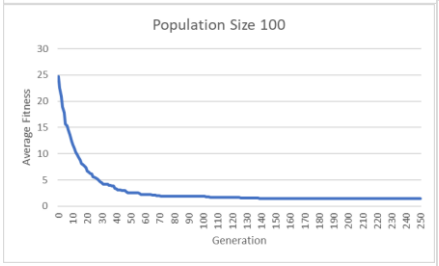
* - The generation cap was reached ** - The time cap was reached

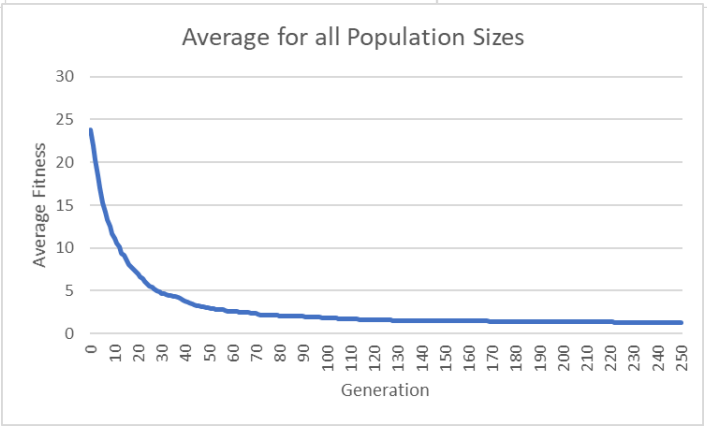
Below are graphical representations of the averages for each population size.











Questions

Which population size was the best?

When compare the different population sizes I determined the factors that make a population size 'good' which are the average time to compute a solution, the number of generations required, and the consistency in which it returns a correct solution (fitness value of 0).

In general, the smaller population sizes were able to compute a larger number of generations in a shorter period, however they were not very consistent in returning a correct solution, only managing to return a close solution in the more difficult grids.

Conversely, the larger population sizes were able to return a correct solution very consistently and only generate a smaller number of generations. However, they took a significant amount of time longer to compute.

For example, across all experiments, the population size of **10** returned a correct solution 47% of the time and, of those instances, required an average of 92 generations to do so. However, it took, on average, 0.54 seconds to process.

The population size of **10000**, on the other hand, returned a correct solution 93% of the time and required an average of 23 generations to do so, but it took an average of 134.71 seconds to compute that result.

It can therefore be deduced that the better population size would be somewhere between these to ensure a high level of success whilst lowering the computation time required. Therefore, I conclude that, from my experiments, the best population size is **1000** as it has a high success rate (87%), low number of created generations (28), and computes an answer in a reasonable amount of time (23.16 seconds).

What do you think is the reasoning for the above answer?
One of the biggest issues with the smaller population sizes is that it struggled, especially in the more difficult grids, to consistently produce a correct solution. I believe the reason for this is that they are more likely to get stuck on, once they encounter, some local minima. This would be due to the smaller number of solutions in each generation and, if they all mutate towards a certain solution, then it is much easier for them to all become very similar and get stuck.
Further, they can store a smaller amount of the total permutations at any one time, making it less likely to start at or move towards the correct solution quickly.
The bigger issue with the larger population was timing, as it took a significantly longer amount of time to compute. This would be due to the sheer number of potential solutions it would need to create and evaluate for each generation. However, due to the vast size of each generation, it becomes much harder for it to become stuck on some local minima.

Which grid was the easiest and which was the hardest to solve?
From the experiments, it can be determined that the easiest grid to solve was Grid 1 as it was able to return a correct solution more consistently, for these experiments, 100%.
Conversely, it can also be determined that Grid 3 was the hardest grid to solve as it had the lowest success rate of the three, for these experiments, 40%.

What do you think is the reasoning for the above answer?
I believe that Grid 1 was the easiest to solve as it contained the most clues, 50. Further, the clues that were provided were more spread out, allowing for fewer potential permutations to generate and thus a lower risk of being stuck on some local minima and a higher probability of creating an initial 'solution' that is close to the correct solution.
Conversely, I believe that Grid 3 was the hardest to solve because it contained the fewest clues, 29, and this meant that there were more potential permutations of the entire solution to consider.
The clues were also not as well distributed as Grid 1 as some rows/columns/boxes contain very little, if not any, clues, meaning that a significantly larger amount of permutations that can be generated by the algorithm for just a single box. This therefore increases the number of local minima in the solution space, due to the lack of information provided in the clues, and thus increase the potential for the algorithm to be caught in one.

What further experiments do you think would be useful and why?
In order to get a better understanding of how well the evolutionary algorithm performs against grids of different difficulty, further experiments would be required to be run against many different grids. The difficulty of these grids can be estimated from the number of clues they have and the distribution of these clues, or by using an established rating system, and would allow for the performance of the algorithm to be mapped against grid difficulty to better evaluate its performance.
It should also be noted, that to better draw conclusions from these results, more experiments should be undertaken with the current parameters to deduce a more reliable average and more clearly highlight anomalous results. This would also help in averaging out the random factors involved with the algorithm, such as the creation of each solution and the creation of offspring.
Further, a wider range of population sizes should be explored, such as 1000000, to ensure that the general pattern for results remains the same, that, for the larger populations, the time required increases, but the number of generations required and the success rate increases, and the opposite for the smaller populations.
Finally, it would be interesting to analyse the performance of the algorithm on grids where there is no unique solution, i.e. there is more than one possible solution, to inspect how it deals with multiple correct solutions, and thus testing the algorithm against all possible types of sudoku grid.