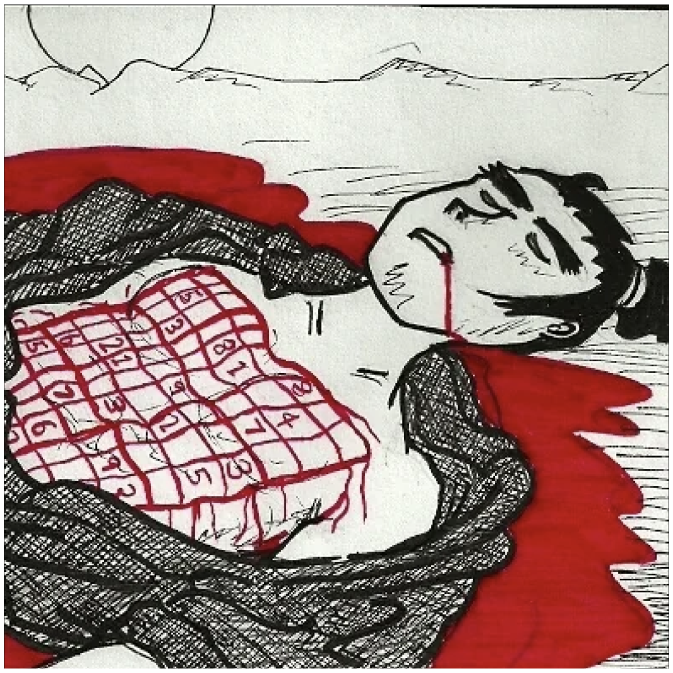

Commits sudoku,
Silent merges, lines align —
Code in perfect form.

Solving Sudoku puzzles with Genetic Algorithms

Anton Kutsenko	*ID: 20230765*
Mateus Amaral	*ID: 20230595*
Maximillian Laechelin	*ID: 20230979*
Diogo Parreira	*ID: 20230758*

Github project link: https://github.com/amarmate/CIFO24

1 Introduction

Sudoku is a world-famous puzzle game, although it was only invented and became popular in the late 20th century. The idea is to complete a partially filled board of size NxN (usually 9x9) in a specific manner, such that N numbers are used N times and never repeated in any row, column or box (squared region of the board with N cells). In the classic version, Sudoku has only one solution that can be achieved iteratively using logical operations.

In that project, we will try to develop a Genetic Algorithm (GA) that will be able to solve Sudoku after several generations of its initial population. We will focus on the implementation of multiple types of selection, mutation and crossover algorithms that can fit the Sudoku problem. Also, different additional techniques to improve GA implementation will be considered, such as elitism and fitness-sharing. Finally, algorithms will be evaluated using statistical tests based on several executions.

2 GA implementation

Our GA implementation is a separate library that contains two main classes: Sudoku and Population, as well as several separate functions that serve specific purposes. The Sudoku class is analogous to what is usually called Individual - one sudoku board, an element of the Population, which is defined by Population class - a set of identical boards. All the boards in the Population have the same predefined fixed elements, which cannot be changed on any iteration of the algorithm optimisation. All the other elements of the board differ for Individuals in Population, but the distribution of the numbers, valid for the Sudoku board, is preserved during initialisation. The other, more detailed properties of both classes will be covered later.

As for any optimisation problem, we need to define the fitness and the representation of any Individual. Firstly, we define the number of conflicts inside a row, column of box as the subtraction of all the numbers that can be put in each one of those (in our case, 9 different numbers) from the unique numbers there are inside each. From then on, the fitness is defined simply as the sum of row, column and box conflicts.

$$\text{Fitness} = \sum_{row=1}^{n} (N - unique_i) + \sum_{column=1}^{n} (N - unique_i) + \sum_{box=1}^{n} (N - unique_i) \tag{1}$$

This definition of fitness makes this a minimization problem, the objective being to achieve a fitness of 0, where all the numbers are correctly placed.

The representation of the board is defined in two possible way. This was useful for different crossovers. The first type of representation is a full board of sudoku as a NxN array, particularly useful for tabular crossover and row-wise cycle crossover. The second, and the simpler representation, is 1D array of all swappable (not fixed) positions of the Sudoku board, going line by line from the top left to the bottom right corner. That representation has one major drawback and one advantage. It does not preserve the structure of the sudoku board, being a one-dimensional array of the same shape for all kinds of sudoku boards: no matter if missing elements are located next to each other or have other values in between. Nontheless, at the same time, this array is easier and faster to manipulate on the code level as there are no fixed elements and the shape is flat.

When each Sudoku element is created, an initial board is passed, which contains the givens of the puzzle, as well as the blanks. The blanks are represented as zeros, and for our specific problem it is best to have the positions filled. Additionally, to reach the solution as fast as possible its best to keep the distributions of the sudoku puzzle balanced, meaning we have the same count of numbers for each of them (in this case: $\text{counts}_{number} = 9,$ for $number \in \{1, \ldots, 9\}$). To achieve this, we implemented a function to fill in the board, while keeping the distribution. Moreover, we devised another type of initialization which respects the within-row distribution, i.e., $\text{counts}_{number,row} = 1,$ for $number, row \in \{1, \ldots, n\}, \{1, \ldots, n\}$. This is done to permit doing special crossovers like the row wise cycle crossover, which will be covered later.

A detailed analysis of the Population class is needed to explain several parameters. Our Population implementation has classical parameters, such as size and arguments to create each Individual. However, there is a slight difference in evolution implementation. We treat the Population as an evolving instance itself, meaning that we perform all the steps of the algorithm on each iteration inside Population replacing existing sudoku boards with new ones after each generation. That is why we define selection, crossover and mutation as methods of the Population.

Going forward, we will focus on the main parameters of the algorithm and their specifications related to the Sudoku problem.

2.1 Selection types

We implemented two basic selection algorithms for the given task: roulette wheel (fitness-proportionate) selection and tournament selection. These selection algorithms offer the possibility of using fitness-sharing for the diversification of the population. Fitness-sharing is implemented with two possible specifications of the sharing coefficient:

1. **Normalized Inverted Sum of Hamming Distances**:

$$f'_i = \frac{f_i}{\sum_{j=1}^{N} \left(1 - \frac{d_H(x_i, x_j)}{L}\right)} \tag{2}$$

where N is the population size, x_i is the representation of individual i, and x_j is the representation of individual j, L is the length of representation, f_i is the fitness of individual i

2. **MinMax Scaled Sum of Hamming Distances in the power of -1**:

$$f'_i = f_i \frac{\left(\sum_{j=1}^{N} d_H(x_i, x_j) - \min d_H\right)}{(\max d_H - \min d_H)} \tag{3}$$

where N is the population size, x_i is the representation of individual i, and x_j is the representation of individual j, f_i is the fitness of individual i.

2.2 Crossover types

We tried 6 different crossovers that can be used with sudoku problems. Some of them are simple crossovers, that can be used for any type of optimisation problem: single-point and multi-point crossovers. These two crossovers are used with a flat array of swappable numbers on the sudoku board. There are two more crossovers that work with the same representation: cycle and special crossovers.

Cycle crossover adapted for Sudoku has the same idea behind it as the one for TSP - to preserve the distribution of numbers. We go on a cycle considering the numbers themselves, not their indices. However, there is an important modification - some numbers will be repeated. To solve this issue on every iteration of the cycle, we do the following steps: If the next number we are looking for is repeated multiple times, we randomly pick one position and go there. On the next iteration of the loop, we will not consider this position again. Every position can be visited only once to avoid infinite loops.

Another crossover that we created is the **special crossover**. This is comparable to the single-point crossover, but it differs in that it keeps the distribution of the numbers. It starts by generating a crossover point, copying the first part of the P1 to O1, and the second part of the P1 to O2. Then it finds the distribution of the numbers in the second part of P1 (which are missing in O1), and goes to P2 to get them sequentially. This sequence is then copied to O1, making O1 complete. The rest of the numbers from P2 are later copied to O2, finishing O2 as well.

For the following two crossover types, the full boards was used, because the crossover techniques heavily rely on its structure to operate. The first was **Row-wise cycle crossover**, which is the same cycle crossover as explained above with one special peculiarity: it performs cycles within rows, meaning that it considers iteratively rows of the board and performs cycles on non-fixed elements of each row of two parents. At the end, a new board is constructed from these changed rows. The main power of this crossover (and its weakness at the same time) is that it will work very well if each row is initialised with correct proportions of numbers and the mutation procedure doesn't alter that fact.

The last crossover we used is **single-point tabular crossover**. It works as a single-point crossover but with rows and columns of the board. On each iteration, we randomly choose the cutting position to split the board into two parts (vertically or horizontally) and then switch these parts of the two parental boards. There are modes to do it vertically, horizontally or randomly on each iteration.

2.3 Mutation types

As for mutations,5 types were implemented. these can be divided into two groups: change-based and swap-based.

Change-based mutations are subdivided into change and change-smart mutations. **Simple change mutation** involves the replacement of one or several non-fixed numbers of the board with a random number from 1 to N (in most cases N=9), depending on the board size. This mutation doesn't preserve the initial distribution of the board which can be crucial for some types of crossovers. **Change-smart mutation** uses the same principle, but it switches the number to one that isn't available on the row, column and box. This results necessarily in the reduction of conflicts, which in turn improves the fitness (Δfitness $= -3$). In the case where no number is available at all, it just uses the simple change mutation.

Swap-based mutations, unlike change-based, never use numbers outside the board, preserving the given distribution of numbers. We implemented simple swap, swap-row and swap-smart mutations in that group. The first one, **swap mutation**, involves a simple exchange of positions between 1 or more couple of numbers on the board. **Swap-row mutation** does the same but with a requirement to permutate numbers inside one random row of the board. If a chosen row has fewer couples of numbers to exchange positions than asked, then the mutation won't be performed. **Swap-smart mutation** is more similar to the swap mutation in that it is board-wise, but it first checks for the numbers that can be exchanged in order to maximize the fitness variation. This is done by looking at two positions in the sudoku board (e.g.: (2,2) and (8,5), filled with 1 and 7 respectively), which numbers they could take without generating conflicts (7, 1,9), and then if the numbers match, swap them (which in this case they do). This swap is much more computationally expensive in comparison to the other mutation methods we've described so far.

2.4 Extra parameters

Instead of vanilla elitism, where only the best individual is kept, in the elitism we implemented it is possible to specify the number of individuals to keep: from 0 (no elitism) to the size of the Population (replication without crossovers and mutation). However, the optimal number is always somewhere in the middle. To grasp the full picture of the ongoing process, we have also implemented methods to calculate and plot the genotypic and phenotypic diversity of the population. Furthermore, we created a specific method to keep the distribution of non-fixed numbers within the board after each step of the evolution. The idea is that for each board after crossover and mutation, the perfect distribution of the non-fixed numbers on the board is compared to the current one and redundant numbers are randomly replaced with missing numbers. But this approach was adding too much noise and didn't allow the algorithms to converge which is why it was removed from the scope of this report.

3 Evaluation and experimentation

We used an interative approach for the setup in our experimentation. On each iteration, we chose the best configuration switching one or several parameters of the GA and running 30 experiments with each. We did not go to the full grid search, since it was too computationally expensive. Step by step we chose the best crossover, mutation, mutation/crossover rates and so on, while keeping other parameters fixed. This approach did not guarantee that we would find the best configuration of the GA but this approach reduced the search space significantly. Also, we started running the algorithm on an easy board to find the best configuration. Afterwards, we experimented with the best algorithm on multiple boards of various difficulties. All the results are reported below. The plot below is not exhaustive, as we decided to use a tabular form. To see more detailed plots and convergence patterns, please refer to the experimentation notebook onGithub.

By default, we started with mutation probability of 0.1, crossover probability of 0.9, population size of 1000, 100 generations, tournament selection and fitness sharing with inverted hamming distances. After each parameter tested, we updated that parameter with the best value according to the chosen metrics.

The main metrics we considered was the average fitness of 30 runs and confidence intervals for it, the percentage of iterations where the board was solved and fast convergence. Welch t-test for equality of means was used to compare pairwise algorithms' mean fitnesses at the end of the run.

Starting with the crossover tests we ran two different initialisations of the board, as mentioned above, to see how it affected the performance. Also, we tested each crossover type with one mutation type that fits that crossover logically (just to reduce computational costs). In the Table 2 below we present the results - the percentage of solved Sudokus, mean fitness and confidence interval. The best parameterization will be highlighted in bold and we put * if the mean fitness after 30 runs for this configuration is significantly different from all the others according to the Welch t-test.

For further tests, we used single-point tabular crossover as fits better the Sudoku problem and works well without row-wise initialization. With full-board initialisation, its mean fitness is not statistically different from the means of all the other crossovers, but higher convergence speed than cycle crossover which has comparable mean fitness.

In the next step different mutations, except row-wise cycle were tested because it was defined for row-wise cycle crossover specifically. Results are shown in the Table 3. Change-smart mutation tended to converge faster, however, swap-smart leaded to slightly better mean fitness at the end.

Although the difference in means was not statistically significant. We will use change-smart further since we value the speed of convergence more in that case.

High mutation and xo rates worked the best and mean fitness is significantly better (Table 4). It was unexpected and a bit controversial to what we normally have when we use genetic algorithms. However, in our case it worked, because mutation was not only maintaining diversity and helping to explore search space, but being "smart" also helped to achieve stable convergence.

We also tried to experiment with selection types, results in Table 5. Tournament selection worked faster (converging after around 20 iterations), however, with the roulette wheel we solved Sudoku in all 30 cases after a maximum of 40 iterations. Welch's test tells us that the roulette wheel mean is significantly better than the tournament mean. However, it was no surprise since the roulette mean = 0 with a standard deviation of 0.

To improve speed we also needed to adjust population size, number of generations and elite size (Table 6). To make it a fair competition we looked for the absolute average time in seconds, not to amount generations, as we know that bigger populations would require less time to reach the solution. Welch's t-test results state that there was no statistically significant difference between the several best configurations fitness means, which is why we relied more on the speed of convergence. Population size = 500, number of generations = 200, elite size = 200 seems the best configuration. Only once it didn't solve the puzzle, however, 29 other times it was twice as fast as GA with a population of size 1000.

With all the parameters defined above we ran tests with 4 different boards: easy, medium, hard, expert (Results in the Table 1). The algorithm was able to solve each level of difficulty at least once, while an easy and medium board were solved in >50% of runs. Even with the hard and expert boards algorithm, in most cases, converged to a fitnesses around 2, meaning that it is stuck in a local optimum that almost solved the board.

Difficulty	easy	medium	hard	expert
% Solved	90%	70 %	26.67 %	3.33 %
Mean	0.27	0.77	1.57	2.3
95% CI	[-0.06, 0.59]	[0.29, 1.24]	[1.18, 1.96]	[1.97, 2.63]
Run/success (s)	24.36/15.58	44.82/20.02	83.18/22.57	105.36/23.46

Table 1: Performance comparison with different board difficulties

4 Conclusion

Sudoku is a problem that is not an easy task for genetic algorithms due to its unusual structure, which requires specific mutations and crossovers. We were able to build GA from scratch and optimise its parameters to solve even the hardest boards. However, there is still potential to build GAs with more exotic crossovers and mutations that might be able to fit Sudoku even better.

Division of labor: All the main decisions about fitness, representation, crossovers, mutations and evaluation pipeline were done all together. For the coding part, Maximillian was in charge of designing and building main methods of Sudoku and Population classes, including defining the board, plotting histories and displaying board, as well as all the attributes and evolution logic, Diogo was focused on fitness function implementation and the evaluation part, including plots and statistical tests. Mateus and Anton implemented selection, mutation and crossover algorithms for the Sudoku problem. Report writing was shared accordingly.

5 Appendix

Crossover Mutation		multi-point change-smart	single-point change-smart	cycle swap-smart	special swap-smart	s-p tabular change-smart	row cycle swap-row
Row-wise init	% Solved	20 %	30 %	16.67 %	6.67 %	20.0 %	**36.67 %**
	Mean	4.13	3.17	4.3	5.1	3.2	**2.5**
	95% CI	[3.12, 5.15]	[2.23, 4.1]	[3.33, 5.27]	[4.0, 6.2]	[2.35, 4.05]	**[1.69, 3.31]**
Full init	% Solved	6.67 %	6.67 %	16.67 %	0 %	**23.33 %**	0 %
	Mean	7.83	5.33	4.17	10.23	**3.97**	9.9
	95% CI	[6.28, 9.39]	[4.19, 6.48]	[3.1, 5.24]	[8.93, 11.54]	**[2.9, 5.03]**	[8.28, 11.52]

Table 2: Performance comparison of various crossovers

Mutation	swap	swap-smart	change	change-smart
% Solved	6.67 %	23.33 %	6.67 %	23.33 %
Mean	6.03	**3.9**	7.67	3.97
95% CI	[4.76, 7.31]	**[2.82, 4.98]**	[6.43, 8.9]	[2.9, 5.03]

Table 3: Performance comparison of various mutations

Mutation rate	0.1	0.5	0.9	0.1	**0.9** *
Crossover rate	0.9	0.5	0.1	0.1	**0.9**
% Solved	20 %	36.67 %	0 %	0 %	83.33 %
Mean	4.03	2.4	14.5	19.27	0.6
95% CI	[2.87, 5.2]	[1.54, 3.26]	[11.5, 17.5]	[17.16, 21.38]	[0.04, 1.16]

Table 4: Performance comparison of various mutation and crossover rates

Selection	**roulette** *	tournament
% Solved	100%	83.33 %
Mean	0	3.9
95% CI	[0.0, 0.0]	[0.04, 0.89]

Table 5: Performance comparison of various selection types

Population size	100	100	100	**1000**	**1000**	500	500
N generations	1000	1000	1000	**100**	**100**	200	200
Elite size	20	50	10	**100**	**300**	100	200
% Solved	33.33 %	13.33 %	56.67 %	**100 %**	**100 %**	93.33 %	96.67 %
Mean	2.5	3.83	1.87	**0**	**0**	0.2	0.07
95% CI	[1.64, 3.36]	[2.92, 4.75]	[0.96, 2.77]	**[0.0, 0.0]**	**[0.0, 0.0]**	[-0.07, 0.5]	[-0.1, 0.2]
Run/success (s)	63.41/9.57	85.55/13.05	46.49/14.07	22.8/22.8	20.5/20.5	13.3/**9.78**	**12**/10.2

Table 6: Performance comparison with various population and elite sizes and N generations