Computational Intelligence for Optimization

Sudoku Solver Using Genetic Algorithms

Alexandre Marques	20230976
Cathal O'Dwyer	20230986
Leon Debatin	20230549
Zenan Chen	20230996

Group:	Sudoku Slayers
Link:	https://github.com/LeonDebatin/CIFO-Sudoku-Solver

NOVA Information Management School

Instituto Superior de Estatística e Gestão de Informação

Universidade Nova de Lisboa

June, 2024

Table of Contents

1 Introduction

The main objective of this project was the development of an effective Sudoku solver using Genetic Algorithms (GAs) implemented in basic Python only. A standard Sudoku puzzle consists of a 9×9 grid divided into nine 3×3 boxes where the goal is to fill in the grid with digits ranging from $1-9$ such that each row, column and box contain each digit once. We aimed to develop an algorithm capable of solving Sudokus of any size (except prime numbers, as the boxes would be equal to columns/rows).

Genetic Algorithms are particularly well suited for solving optimization problems in vast and complex solution spaces, inspired by the process of natural selection. In order to solve Sudoku puzzles of different difficulty levels, we systematically evaluated various configurations of genetic operators and selection algorithms to find the best performing combination by running and analysing a random search. Using the resulting configuration, we attempted to solve multiple puzzles taking over 30 samples each, first for 9×9 grids, then for 6×6 and 12×12 grids.

2 Genetic Architecture of Individuals

An Individual is initialized with a Sudoku-Object of our self-built Sudoku-Class, which forms the foundation of each Individual. A Sudoku-Object takes a template, which is a quadratic list of lists with the given numbers and 0s as placeholders for the empty cells, as the input. To make our solution applicable for sudokus of any size, the Sudoku-Class calculates box size based on the total size by finding the multiplicators with the smallest sum. For example, a Sudoku puzzle with a size of 18, has two possible box sizes, 2x9 and 3x6, we always choose the one with the smallest sum.

When initializing an Individual it will fill the 0s in the template with random numbers from the possible entries, resulting in a list of lists containing numbers in the range from 1 to the size of the Sudoku-Template. We ensure that each cell only takes numbers from the possible entries to avoid an unnecessarily large search space for our Individuals and therefore being more efficient and avoiding some unplausible local optima.

We choose a list of lists representation because it imitates the structure of an ordinary Sudoku puzzle, allowing us to deploy our manipulation functions effectively. You could argue that a single list or a string implementation might be more efficient regarding resulting running times, but functionality-wise there shouldn't be a difference. Because it's commonly used [1] we thought about n-bit-encoding methods to encode each number within the Sudoku-Grid but couldn't think of any advantages through doing so.

To calculate the fitness of an Individual we count the missing numbers that are not represented within each row, column and box and added them up, which is equivalent to counting the duplicates. This way we ended up working with a minimization problem, where we have a solved puzzle when the fitness function is 0. In the case of a 9×9 grid the maximum number of conflicts for each row, column and box is eight. Therefore, the worst possible solution theoretically possible (just the same number in each cell) would lead to a fitness of $3 \times 9 \times 8 = 216$.

We considered different fitness functions, such as calculating the expected sum for each row, column and box but decided against it as we found the duplicate count more relevant and directly tied to the definition of a solved Sudoku.

3 Selection Algorithms

Selection methods identify which individuals from the current population will be chosen to produce the next generation. Orienting on the paper [1] we decided on three such methods: Fitness Proportionate Selection (FPS), Tournament Selection, and Boltzmann Selection. These selection methods are implemented in the 'selection.py' script.

FPS selection works by assigning each individual a probability of selection based on its fitness score relative to the other members within the same population. Lower fitness scores are more desired, as they indicate less conflicts. We transform the fitness scores to ensure individuals with fewer conflicts have a higher probability of being selected. This method works the same was as a roulette wheel, each slice is proportional to the probability of the individual being selected, offering higher chances of selection to the individuals with better solutions.

For Tournament Selection, the '*selection.py*' script randomly selects a subset of individuals from the population. The *tournament size* is given in percentage, whereas the absolute amount will be calculated based on the population size. Within this subset, the individual with the best fitness score (fewest conflicts) is chosen to be a parent. This process keeps repeating itself until the required number of parents are chosen. Tournament Selection takes place within a population and ensures that fitter individuals have a greater chance of being selected while maintain diversity within the population.

Boltzmann Selection uses a temperature parameter that gradually decreases overtime, this is controlled by the cooling rate. Initially, the Boltzmann method selects a wide range of individuals, known as the exploration phase, but as temperature decreases, individuals with a better fitness are more frequently selection. This method balances exploration and exploitation, thus permitting an initial wider solution followed by gradually shifting focus on to the better solutions. The balance between these two can be shifted by altering the cooling rate, thus influencing the balance of exploration and exploitation over time.

We also implemented elitism, where the worst $x\%$ of children are replaced with the best $x\%$ of the current population before replacing the current population with the children.

4 Genetic Operators

4.1 Crossover

Crossover is a genetic operator used to vary the programming of a chromosome from one generation to the next. It mimics sexual reproduction; two individuals are selected from the mating pool at random to 'crossover' and produce offspring. Our implementations builds upon well-known crossover techniques such as, Single-Point Crossover, Multi-point Crossover and Uniform Crossover. In Single-Point Crossover, one randomly selects a point in the genomes of parents where genetic material is exchanged. For the Sudoku grids one can do this from three perspectives: rows, columns and boxes. This is why we developed a "Mixed Single-Point Crossover" method that with $1/3$ chance each does a Single-Point Crossover regarding one of those perspectives, whereas the Crossover-Point is selected randomly as well. The implementation allows different variations of reproduction, thereby improving recombination potential.

The "Mixed Multi-Point Crossover" is also based on the same $1/3$ chance principle but allows for multiple crossover points. Since our algorithm is designed to solve Sudokus of any size, we didn't want to define a specific number of crossover points but let the size influence it. Therefore, after each row, column or box there is a coin flip determining if there will be a crossover point or not, making the amount and position of crossover points determined by the puzzle size and chance.

In implementing Uniform Crossover, we decided having a $50/50$ chance for each cell of one child to receive the cell from one parent or another, with the other child taking the corresponding opposite. We considered applying this to the row, column or box perspective but decided that it would be too similar to the Mixed Multi-Point Crossover algorithm and therefore discarded the idea.

Besides those algorithms we didn't see any senseful Crossover methods for the Sudoku problem given our initialization in [1] but came up with another idea. From observing nature, we noted that it doesn't restrict itself to one single crossover or mutation method but that they all can happen within a population. This led us to create "A bit of All Crossover" where for each pair of parents, one of the previously presented algorithms is selected at random.

4.2 Mutation Operators

Mutation operators introduce simple random adjustments to individuals that ensures genetic diversity and preventing premature convergence. Our implementations can be found in the '*mutation.py*' script, each with the aim of increasing genetic variability and maintaining the robustness of the population.

We implemented a Flip Mutation, based on the concept of flipping a number (1 to 0, 0 to 1) when working with binary representations. We adapted this idea for the Sudoku problem, where we replace the number in a cell with a randomly selected number from the possible entries (if there is only one number possible, it remains the same).

Another mutation technique we call Exchange Mutation, this involves switching the numbers of two cells within a row, column or box if the possible entries allow it. Since we face the three perspectives, we implemented a "Mixed Exchange Mutation" ensuring variability.

Following the argument that in nature, different mutation variants impacting individuals from the same population, we also applied an "A bit of All Mutation", which with a 50% chance, either performs a Flip Mutation or a Mixed Exchange Mutation to further enable exploration and genetic diversity.

4.3 Evolutionary Process

The evolution process integrates the application of selection, crossover and mutation together to iteratively improve the population over generations. Our genetic algorithm achieves this by using populations that are iteratively improved with selection crossover and mutation, therefore exploring the solution space and producing optimal or near-optimal Sudoku solutions.

5 Finding 'the best' Configuration

In order to find 'the best' configuration for solving the Sudoku puzzles using Genetic Algorithms, we conducted a Random Search with 200 different configurations. Each configuration was tested on Sudoku puzzles of five difficulty levels, ranging from easy to ultimate, resulting in a total of 1000 samples. We applied some restrictions when setting up the random search. Based on initial experiments, we observed that the FPS selection method struggles to converge without elitism, so our random search included elitism when FPS was randomly selected. The search space of possible values for the *mutation rate*, *crossover rate elitism rate* etc. was restricted based on literature recommendations [2]. The *mutation rate*, *tournament size*, and *elitism size* ranged from 1 to 19% each, *crossover rates* from 70% to 100%, and *population sizes* up to 5000. For Boltzmann Selection we allowed temperatures from 0.1 to 100 and *cooling rates* from 0.9 to 1.01, we considered a *heating rate* of 1.01, knowing it goes against the theoretical principal of exploration and exploitation, because we saw in an original experimental phase that the Boltzmann selection gets stuck in local optima quite easily, and therefore increasing the temperature in the later generations might lead to more diverse selections and therefore helps getting out of local optima. Furthermore, we applied an *early stopping* method that stops the solving algorithm when the *best fitness* didn't decrease for the last 150 generations to save computational time.

Analysing the results of the random search, we focused on configurations that were able to solve the most Sudoku puzzles. Seven configurations solved three Sudokus, but no configuration was able to solve more than three. From inspecting these successful configurations, we found that all of them used FPS as the selection method, with population sizes of 5000, 1000 or 500. Besides these factors there were no other visible similarities. This analysis led us to conclude that using FPS and having a large sample size are crucial factors for success.

Out of the 1000 samples, 70 led to a solved puzzle, involving 48 out of the 200 configurations. To further refine our configuration, we analysed each parameter combination (*population size*, *elitism percentage*, *crossover method*, *crossover rate*, *mutation method*, and *mutation rate*) to identify the most effective combinations. For a population size of 5000 and an elitism proportion of 0.13, the 'A Bit of All Crossover' method with a crossover rate of 0.9, and the 'A Bit of All mutation' method with a mutation rate of 0.05 were the most successful.

Since the larger population sizes generally yielded better results, we also ran five samples with the configuration using 10,000 individuals. However, this did not result in improved outcomes, suggesting a sweet spot at 5000, beyond which the quality of the solution doesn't justify the required additional computational effort [2]. Given the sample size and comparing it to all the possible configurations there is no way of knowing we found 'the best' configuration and there is always the possibility to run more tests. We are confident to say we identified one of the best configurations with the potential to solve a wide range of Sudoku puzzles.

6 Solving Attempts

Using the optimal configuration identified from the random search, we conducted our own experiment trying to solve the Sudoku puzzles with varying difficulties for the 9×9 grid, taking 33 samples for each difficulty level. Furthermore, we made an adaption to the solving algorithm, since we saw that we often get stuck in local optima we try to escape it by increasing the mutation rate for 5 epochs to 0.25 if the fitness did not improve for 50 generations. Additionally, we removed early stopping, allowing each solving attempt to run for the full 250 generations if a solution was not found prior.

Given this set up we achieved the following success rate for the 9×9 Sudoku puzzles: the easy puzzles were solved 30 out of 33 times, normal puzzles 25 out of 33 times, hard puzzles 28 out of 33 times, evil puzzles 10 out of 33 times and ultimate puzzles only 1 out of 33 times. Analysing the plots in the appendix for the evil and ultimate puzzles, one can see that the median value often gets stuck at a local optimum with a fitness of 2, indicating our approach of increasing the mutation rate in order to escape local optima was not effective most of the times.

To better understand our results, we examined the complexity scores, which represent the number of possible configurations given the allowed entries (calculated by multiplying the possible entries for each cell). We found that the evil Puzzle has a complexity factor 100 times greater than that of an easy one and the ultimate has factor $300\,000$ greater. This suggests that as the complexity of the puzzles increases, the number of local optima also increases, leading to many unsolved puzzles for the evil and ultimate difficulties.

Our goal was to develop a Sudoku solver capable of solving puzzles of any size, we therefore extended our experiments to 6×6 and 12×12 size girds, testing each for the difficulties easy, normal and hard. For the 6x6 puzzles, we were able to solve all difficulties within 25 generations for each of the 33 trials. This success can be attributed to the lower complexity scores of the 6×6 puzzles compared to the 9×9 puzzles.

For the 12×12 Sudokus, we were only able to take 13 samples for each difficulty due to long running times associated. The easy puzzles were solved every time, the normal puzzles 11 out of 13 times and the hard puzzle was not solved at all. Comparing the complexity scores from the 12×12 puzzles with the 9×9 puzzles, we find that the normal 12×12 has a complexity score factor 100 times greater than the ultimate of the 9×9 puzzles.

Despite this, we solved the ultimate 9×9 Sudoku only 1 out of 33 times and the normal 12×12 Sudoku 11 out of 13 times. We assume the reason for this is that even though there are way more possible representations for the 12×12, the proportion of unviable solutions is higher, leading to their exclusion through selective pressure. The real problem is the number of local optima, which we seem to have more of in the 9×9 ultimate puzzle.

7 Conclusion

In this project, we successfully developed a Sudoku solver capable of solving puzzles of various sizes using Genetic Algorithms. To find ‘the best’ configuration, we ran and analysed a random search and did some additional tests afterwards, leading to a configuration that was able to solve many Sudokus of different sizes and difficulty levels. The primary challenge we faced was dealing with high complexity puzzles, which often led the algorithm to get stuck in local optima. Our adaptive mutation strategy helped mitigate this issue to some extent, but further improvements are possible.

If we were to redo the project, we would consider choosing a single list representation and apply parallelization to achieve faster running times and therefore run a larger sample for the random search. Additionally, exploring more selection methods such as Rank or Stochastic Universal Selection could yield better results. Different initialization strategies that ensure each box is already solved within itself and therefore your crossover and mutation methods only evolve around reshuffling the order within each box, which could be worth further inspecting. Finally, conducting separate random searches for each Sudoku Size could optimize configurations more effectively for different puzzle dimensions.

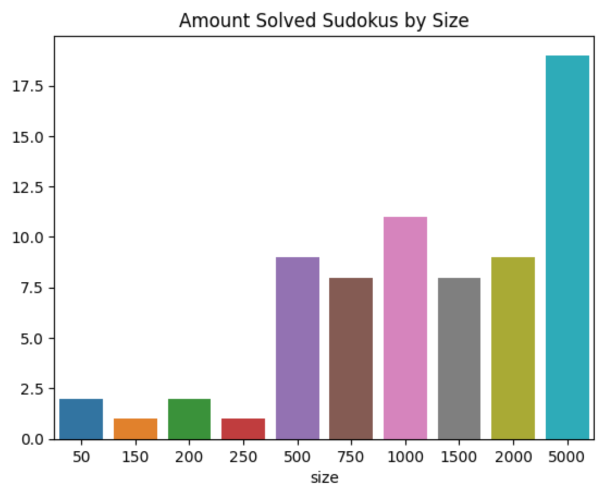
8 References

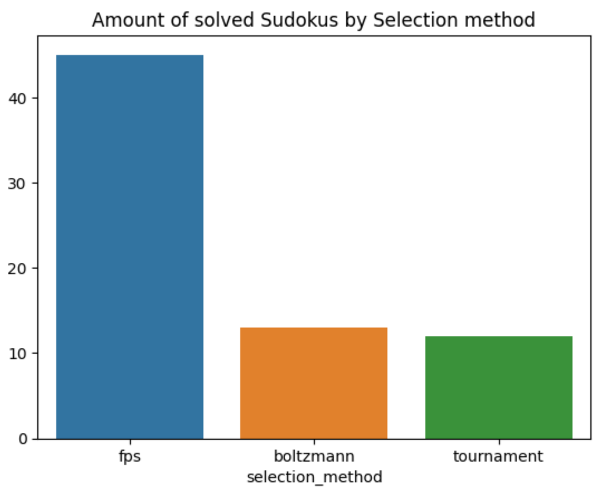
[1] S. Katoch, S. S. Chauhan und V. Kumar, „A review on genetic algorithm: past, present and future,“ *Multimedia Tools and Applications,* 2021.

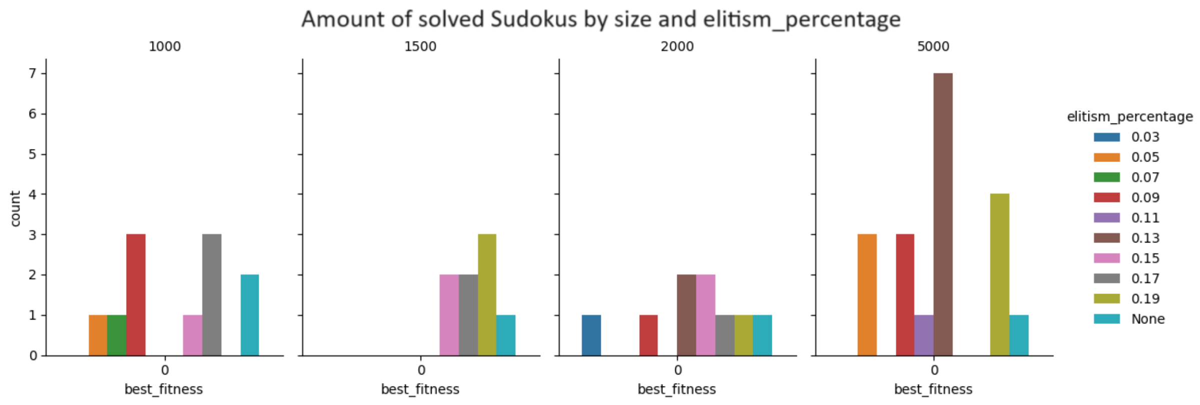
[2] L. Vanneschi und S. Silva, Lectures on Intelligent Systems, Lisbon, Portugal: Springer, 2023.

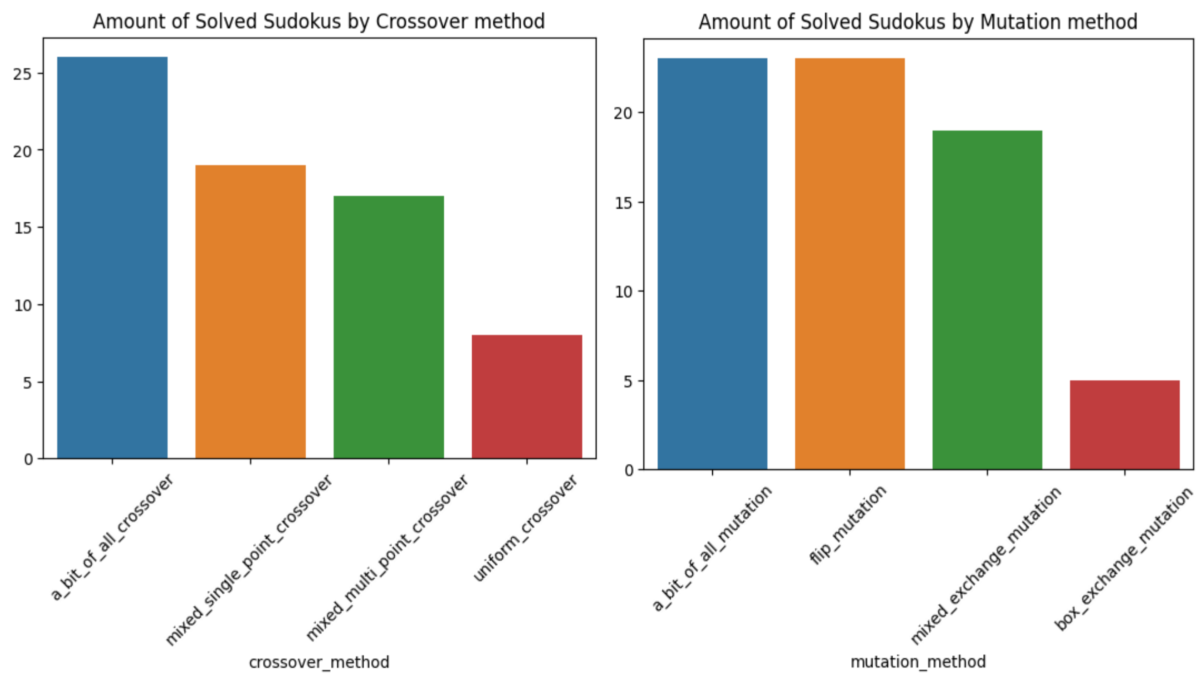
9 Appendix

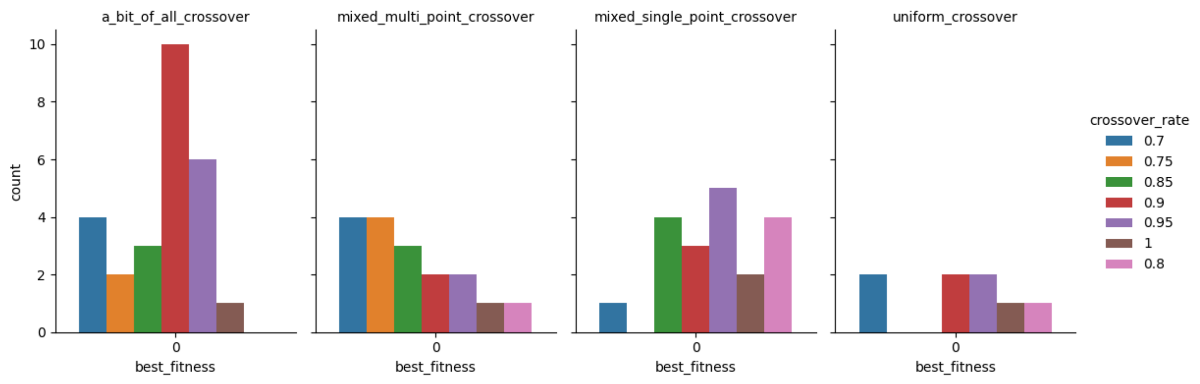
Visualizations of the random search:

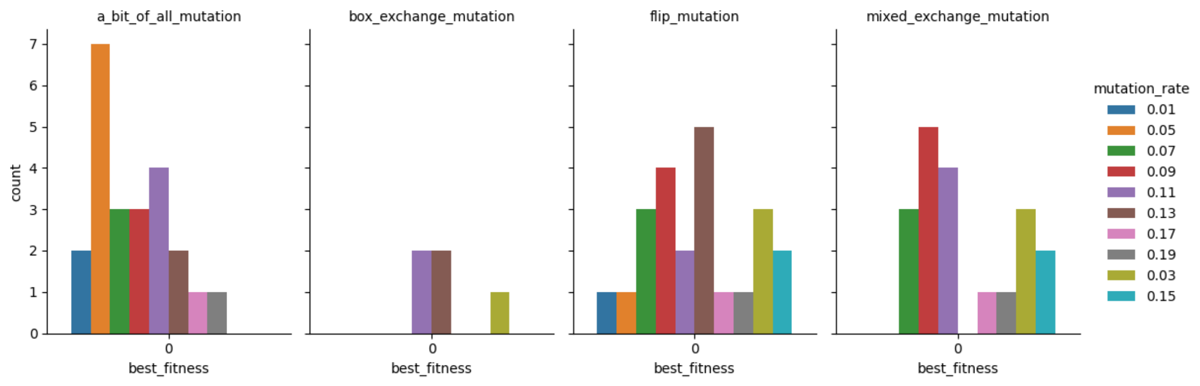




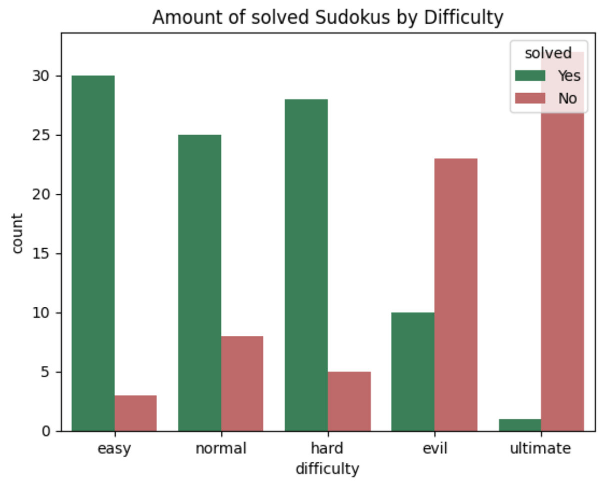


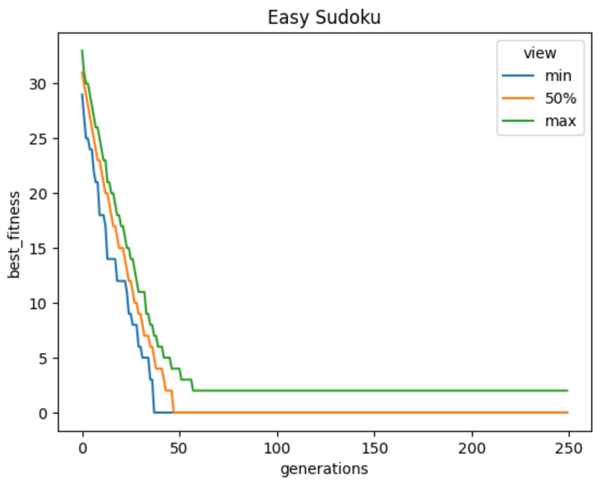

Amount of Solved Sudokus by Crossover method
25
20
15
10
5
0
a_bit_of_all_crossover
mixed_single_point_crossover
mixed_multi_point_crossover
uniform_crossover
crossover_method
Amount of Solved Sudokus by Mutation method
20
15
10
5
0
a_bit_of_all_mutation
flip_mutation
mixed_exchange_mutation
box_exchange_mutation
mutation_method

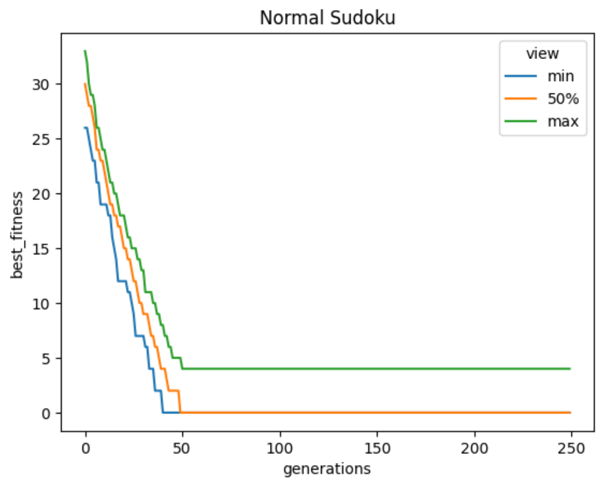

a_bit_of_all_crossover
mixed_multi_point_crossover
mixed_single_point_crossover
uniform_crossover
count
10
8
6
4
2
0
0
best_fitness
crossover_rate
0.7
0.75
0.85
0.9
0.95
1
0.8

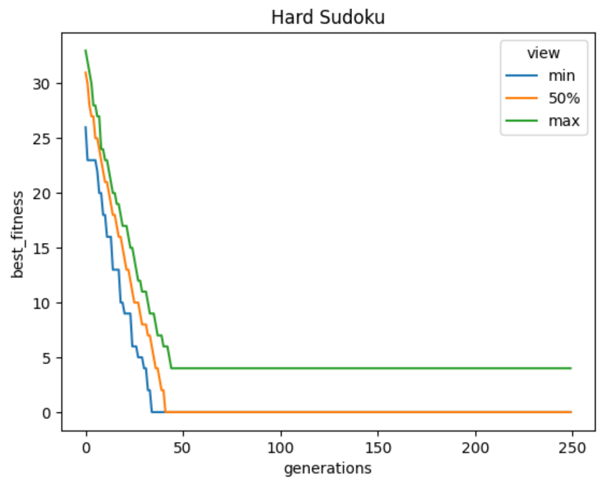

a_bit_of_all_mutation
box_exchange_mutation
flip_mutation
mixed_exchange_mutation
count
7
6
5
4
3
2
1
0
0
best_fitness
mutation_rate
0.01
0.05
0.07
0.09
0.11
0.13
0.17
0.19
0.03
0.15

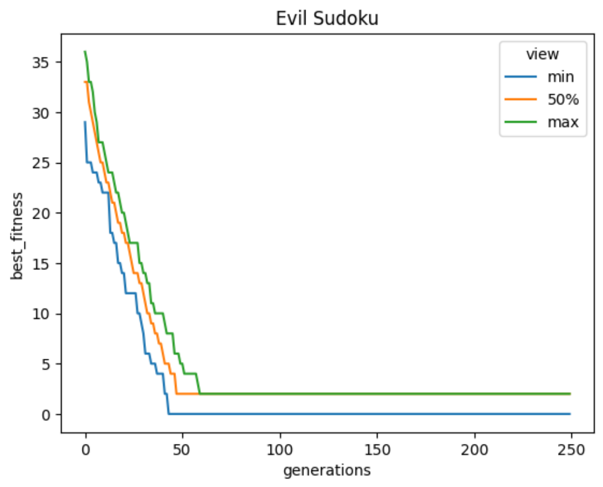
Solving attempts 9x9:

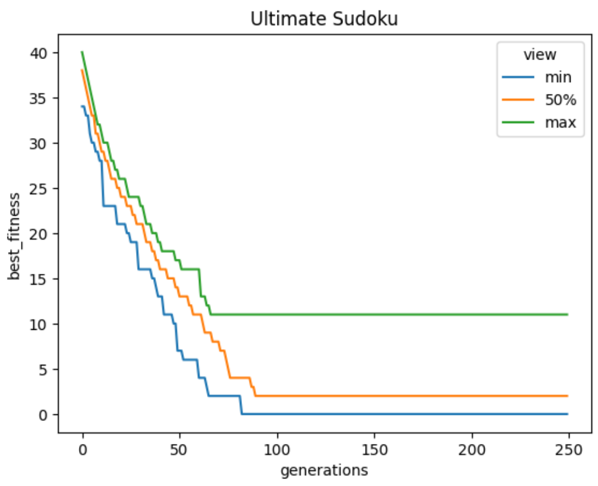




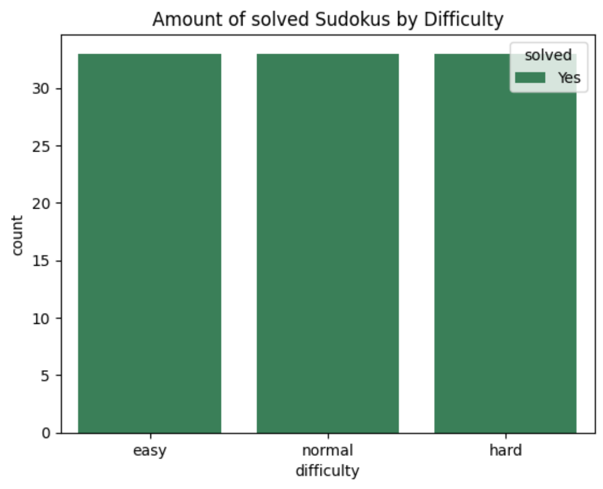


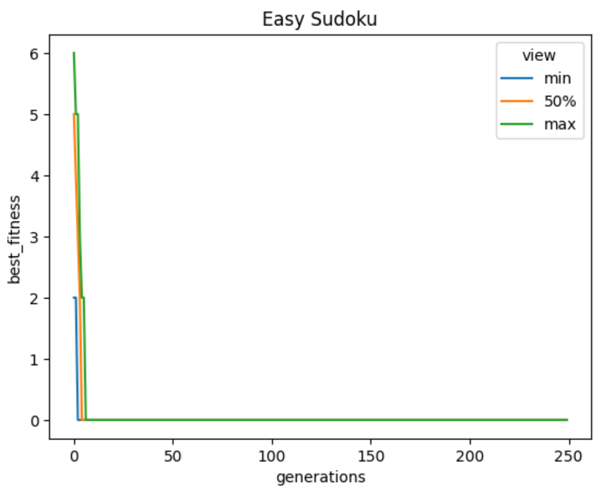


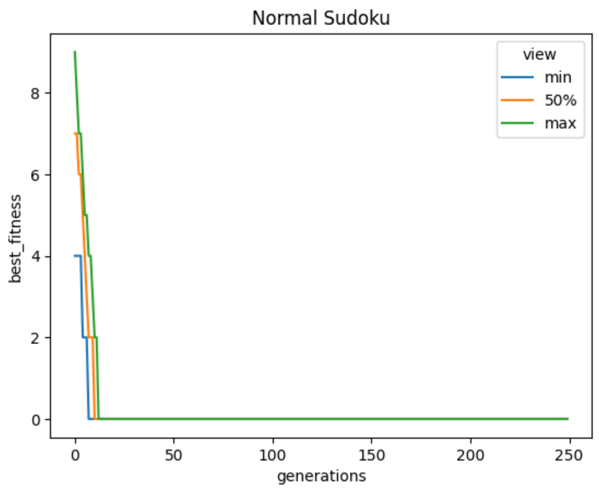


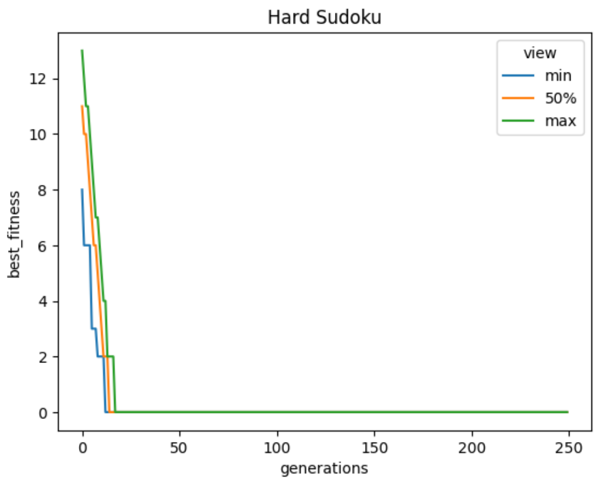


Solving attempts 6x6:

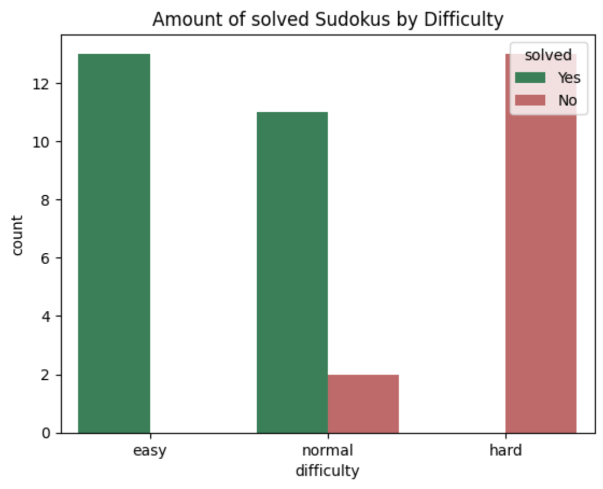


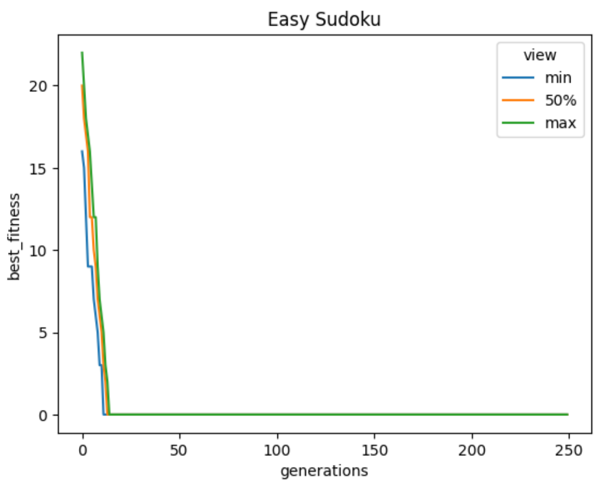


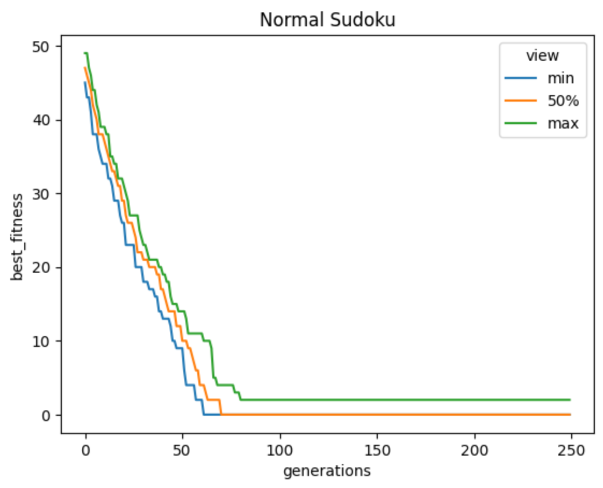


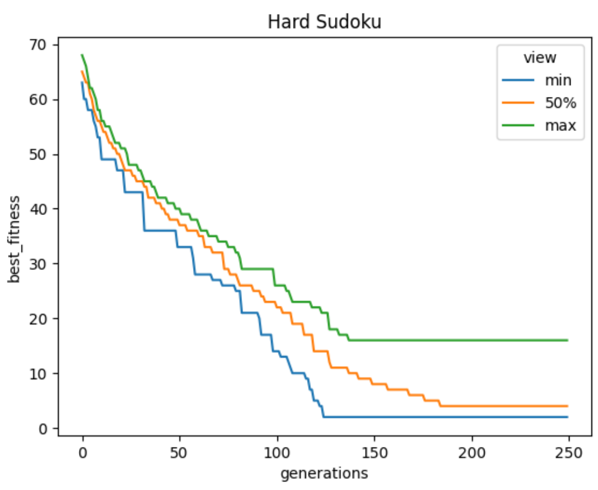


Solving attempts 12x12:









Complexities Score:

	9x9	6x6	12x12
easy	27732638834255462400000	2519424	17832200896512
normal	5990249988199179878400	604661760	128930309981131694327149363200000
hard	1919044442020254842880 0	88159684608	670124079286592951200246501736448000 0000000000000
evil	221861110674043699200000 00	-	-
ultimate	969325983275608524718080000 000	-	-

<u>Division of work:</u>

We the authors contributed equally to the development of this work.