Sudoku Solver

Diogo Barros, 20230555; Diogo Roiçado, 20230557; Pedro Guedes, 20230569; Rodrigo Rocha, 20230593

GitHub Repository

Our folder with the code implementation can be found on: CIFO_PROJECT

Problem Statement and Representation

This project investigates the use of Genetic Algorithms (GAs) for solving Sudoku puzzles, a complex combinatorial optimization problem. GAs mimic natural selection to evolve solutions over generations. We represent the Sudoku grid as a two-dimensional array, where everyone is a potential solution. A fitness function measures solution quality by counting rule violations in rows, columns, and sub grids. Custom genetic operators—selection, crossover, and mutation—efficiently explore the constrained Sudoku search space.

To simplify explanations, the example of the 9x9 grid was used, however the implemented code accepts any grid size, provided it is a square of an integer. Individuals are evaluated based on how closely they adhere to Sudoku's constraints, with the goal of minimizing errors measured by the fitness function.

Division of labour

Diogo Roiçado and Pedro Guedes have developed the concept of the individual and population in the context of Sudoku, including the implementation of operator functions. Meanwhile, Diogo Barros and Rodrigo Rocha have implemented the grid search for parameter optimization and conducted tests on various combinations of operators. Finally, each member had an equally important role in the construction of the report.

Initial Population Setup

To start, we define an initial individual by creating an array of shape *grid_size* * *grid_size* and testing it across various difficulty levels (easy, medium, hard, extreme) to test the algorithm's capabilities and understand how different operator combination's function. Initially, the unfilled cells, represented as zeros, are randomly filled with numbers within the valid range (1 to *grid_size*). This method ensures diverse initial solutions, essential for effective exploration. Additionally, we establish a fixed mask, a binary matrix allowing changes only to initially unfilled cells, ensuring adherence to the initial problem constraints.

To improve initial solutions, we implemented a method called constraint propagation with fallback, which examines each cell's row, column, and sub grid to ensure no rule violations. If multiple numbers are valid, it randomly chooses one of them to fill the cell. If no valid number is found, a random number from 1 to the *grid_size* is assigned. This blend of constraint checking, and random assignment enhances initial solution quality while maintaining diversity.

The Population class initializes a population of potential solutions using the constraint propagation method to define initial representations. By starting with a diverse population, the algorithm avoids local optima and increases the likelihood of finding a correct and optimal Sudoku solution. Over multiple generations, the algorithm favours individuals with fewer errors, gradually converging to a valid solution. This highlights the importance of a well-constructed initial population, adjusted for the evolutionary search process and significantly influencing the algorithm's overall performance and success.

Fitness Function Definition

In our project, we designed and experimented with different fitness functions to evaluate and evolve potential solutions to Sudoku puzzles using Genetic Algorithms (GAs). The fitness function quantifies how close a solution is to meeting the desired criteria. We explored two fitness functions: the standard fitness function and a fitness sharing function.

The standard fitness function counts rule violations in rows, columns, and sub grids. It initializes a fitness variable as 0, sums the range from 1 to *grid_size* subtracted by unique non-zero numbers, and treats unfilled cells as errors. This provides a clear measure of how far a solution is from being valid. Unfilled cells occur only in the first generation, as they are replaced with random numbers afterward.

The fitness sharing function maintains diversity by reducing the fitness of similar individuals. It calculates the Hamming distance between individuals, normalises the distances, and applies a sharing function to penalise similar solutions, increasing their fitness and to “reward” more diverse individuals, decreasing their fitness. This encourages exploration of a broader range of solutions and prevents premature convergence to suboptimal solutions.

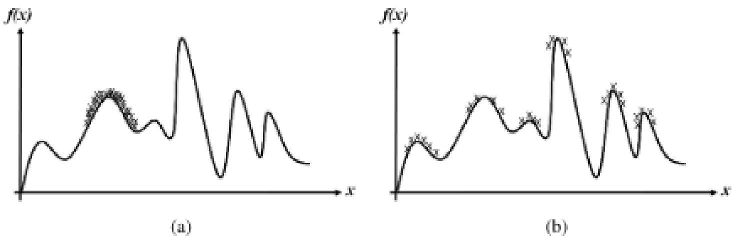


Figure 1 - Converge without/with Fitness Sharing

We chose the standard fitness function for its simplicity and effectiveness. It was easier to implement, provided a direct measure of solution quality, and led to quicker convergence to valid solutions. The standard fitness function was also easier to understand, allowed for straightforward analysis, and required less computational power and time. While fitness sharing maintains diversity, it was more complex and less efficient in our experiments.

Genetic Algorithm Design

Our approach leverages a structured Genetic Algorithm to solve Sudoku puzzles, incorporating essential GA components such as population initialization, fitness evaluation, selection, crossover, and mutation. This systematic process enables the algorithm to iteratively evolve towards optimal solutions, effectively navigating the search space of Sudoku constraints. Below is the detailed pseudo-code for the Genetic Algorithm implementation.

```
1.  Initialization of the Population - Initialize a population P of N individuals (applying constraint propagation
    with fallback)
2.  Evolutionary Loop for Each Generation
    2.1.  Evaluation of Each Individual's Fitness
    2.2.  Create an empty population P’
    2.3.  Elitism (it will be analysed forward if it is applied or not)
    2.4.  Repeat until P’ contains N individuals
          2.4.1.  Select two individuals from P with a selection algorithm
          2.4.2.  Apply crossover to the selected individuals
          2.4.3.  Mutate the resulting individuals with a certain probability
          2.4.4.  Insert the resulting individuals to P’
    2.5.  P := P’
3.  Return the best individuals in P
```

Selection Operators

Chooses individuals (parents) from the current population generation to create offspring to be added into the next generation. The chosen algorithm can largely influence the rate of convergence and the quality of solutions depending on the approach taken: ***tournament_sel, roulette_wheel_sel, rank_sel*** and ***random_sel*** (randomly selects two elements from the population).

Crossover Operators

Crossover functions are designed to combine genetic information from two parent individuals to generate new offspring, essential for exploring new genetic variations. Here’s a brief overview of each method used:

single_point_xo, **cycle_xo**, ***swap_xo***: Selects a random crossover point and swaps rows between parent grids beyond this point, introducing significant changes, ***subgrid_xo***: Focuses on sub grids within the Sudoku grid, randomly selecting and swapping sub grids between parents to maintain overall structure.

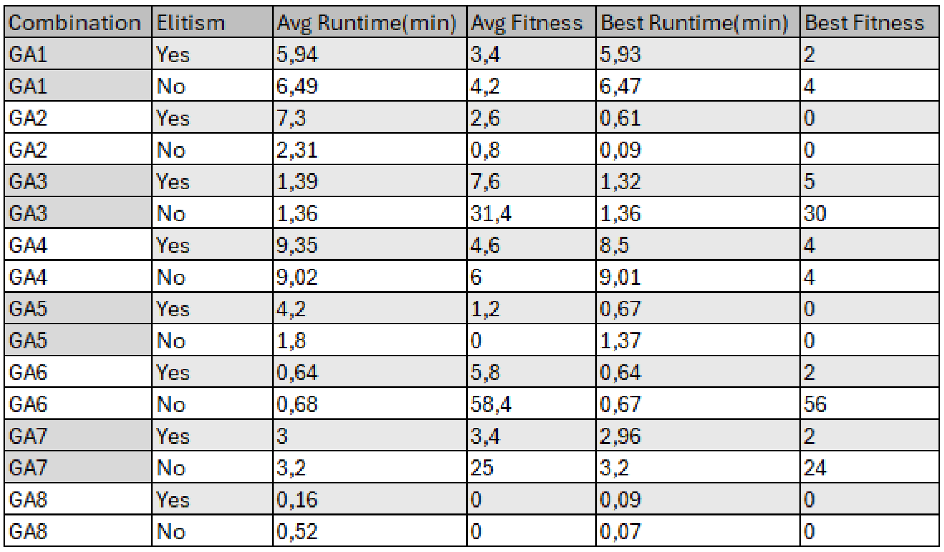
Mutation Operators

Mutation operations introduce diversity and help avoid local minima in the population. Here's a brief overview of each method used: ***single_mutation***: Selects a random mutable cell and assigns it a new random value, ***random_mutation***: Iterates through all mutable cells, assigning them random values between 1 and the grid size, ***subgrid_mutation***: Operates within sub grids, assigning random values to empty cells that respect sub grid constraints, ***valid_mutation***: Assigns values to mutable cells that do not violate Sudoku constraints by considering valid options within rows, columns, and sub grids, ***trial_mutation***: Similar to valid mutation, but only considers the original unfilled Sudoku values, ensuring new values respect initial constraints.

Operators Combination

For our project, we explored various combinations of crossover, mutation, and selection methods to find the most effective approach for solving Sudoku puzzles using Genetic Algorithms. We excluded rank selection due to inefficiency and time consumption. **GA1**: **Single Point Crossover**, **Single Mutation** and **Tournament Selection,** providing efficient gene mixing, minimal diversity, and strong competitive pressure. **GA2**: **Cycle Crossover**, **Valid Mutation,** and **Roulette Wheel Selection**, maintaining sub-block integrity, ensuring valid mutations, and balancing selection pressure based on fitness. **GA3**: **Swap Crossover, Sub grid Mutation** and **Random Selection**, maintaining positional integrity, focusing on sub grids, and maintaining dissimilarity. **GA4**: **Sub grid Crossover, Trial Mutation** and **Random Selection**, targeting sub grid cohesion, introducing new genetic material, and ensuring broad exploration. **GA5**: **Single Point Crossover, Random Mutation** and **Rank Selection**, introducing high diversity and preventing premature convergence. **GA6**: **Cycle Crossover, Sub grid Mutation** and **Tournament Selection**, maintaining gene order and providing strong competitive pressure. **GA7**: **Swap Crossover, Single Mutation,** and **Roulette Wheel Selection**, maintaining positional integrity, introducing diversity, and allowing high-fitness individuals to dominate. **GA8**: **Single Point Crossover**, **Valid Mutation**, and **Random Selection**, focusing on sub grid cohesion, ensuring solution validity, and maintaining diversity. These combinations were designed to explore genetic diversity and selection pressure, aiming to find the most suitable approach for solving Sudoku puzzles effectively.

To understand which one was the most suitable for our project, we applied them all, five times each, with a population of 5000 and 100 generations for the medium level sudoku with probabilities of 0.85 crossover, 0.25 mutation and 0.05 elitism as they seemed a solid initial combination. As we can see in the figure below, the best combination was GA8, as it had the best average fitness and runtime. Hence, it is the one we employed for further analysis.



Combination	Elitism	Avg Runtime(min)	Avg Fitness	Best Runtime(min)	Best Fitness
GA1	Yes	5,94	3,4	5,93	2
GA1	No	6,49	4,2	6,47	4
GA2	Yes	7,3	2,6	0,61	0
GA2	No	2,31	0,8	0,09	0
GA3	Yes	1,39	7,6	1,32	5
GA3	No	1,36	31,4	1,36	30
GA4	Yes	9,35	4,6	8,5	4
GA4	No	9,02	6	9,01	4
GA5	Yes	4,2	1,2	0,67	0
GA5	No	1,8	0	1,37	0
GA6	Yes	0,64	5,8	0,64	2
GA6	No	0,68	58,4	0,67	56
GA7	Yes	3	3,4	2,96	2
GA7	No	3,2	25	3,2	24
GA8	Yes	0,16	0	0,09	0
GA8	No	0,52	0	0,07	0

Figure 2 - Combinations Operators Results

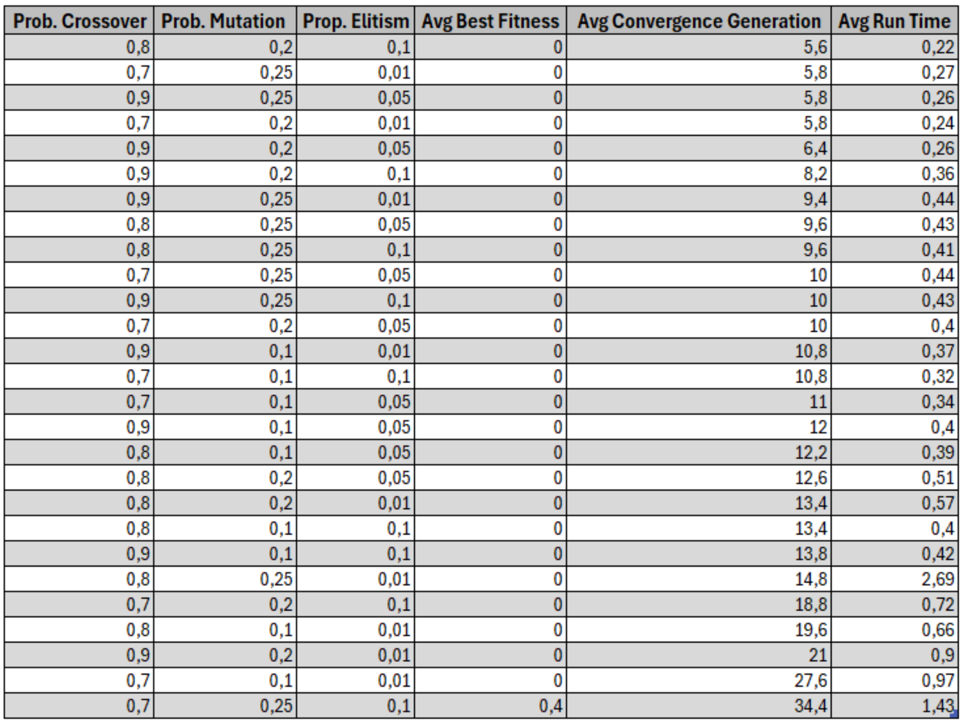
Combinations Analyses

The analysis of GA combinations reveals that different genetic operators and the use of elitism significantly impact the performance of the genetic algorithm. Elitism generally improved average fitness without significantly increasing runtime, as observed in GA1, GA3, GA4, GA6, GA7 and GA8. The absence of elitism in GA3, GA6, and GA7 led to poor fitness results, emphasising its importance for maintaining high-quality solutions. Different operator combinations also affected convergence, with GA8 showing the best performance, suggesting that optimal operator selection is crucial for efficiently solving Sudoku puzzles. Overall, incorporating elitism and choosing the right combination of operators are key to enhancing the genetic algorithm's effectiveness.

Grid Search over Parameters

In order to determine the optimal parameters for GA8 to use in the algorithm, we conducted a manual grid search over some parameters. On crossover, a higher crossover probability promotes exploration of diversity by increasing the likelihood of recombination. We tested the values 0.7, 0.8, 0.85, and 0.9. Additionally, we evaluated the mutation probability, adhering to the genetic perspective where high mutation rates are uncommon. Therefore, we limited our search space to the values 0.10, 0.15, 0.2, and 0.25. Furthermore, we set a parameter called elitism proportion with the possible values of 0.01, 0.05, 0.1 to determine the proportion of individuals from the previous population that would be retained for the next generations. This was done to preserve the best individuals and ensure better convergence of our algorithm.

For each combination of parameters, we ran the algorithm five times with a population of 12500 and 125 generations and calculated the average fitness, average convergence generation, and average runtime across these runs. The results can be seen in the figure below having columns for crossover probability, mutation probability, elitism proportion, average fitness, average convergence generation and average runtime. The best combination found was Crossover probability = 0.8, mutation probability = 0.2 and elitism = 0.1.



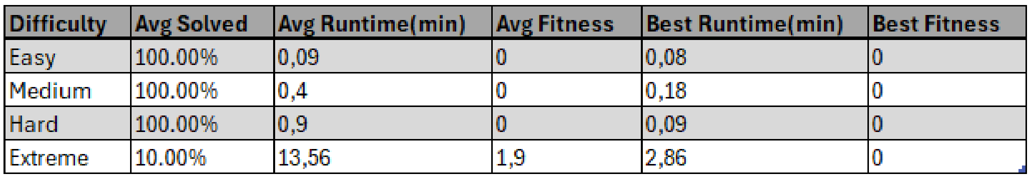
Prob. Crossover	Prob. Mutation	Prop. Elitism	Avg Best Fitness	Avg Convergence Generation	Avg Run Time
0,8	0,2	0,1	0	5,6	0,22
0,7	0,25	0,01	0	5,8	0,27
0,9	0,25	0,05	0	5,8	0,26
0,7	0,2	0,01	0	5,8	0,24
0,9	0,2	0,05	0	6,4	0,26
0,9	0,2	0,1	0	8,2	0,36
0,9	0,25	0,01	0	9,4	0,44
0,8	0,25	0,05	0	9,6	0,43
0,8	0,25	0,1	0	9,6	0,41
0,7	0,25	0,05	0	10	0,44
0,9	0,25	0,1	0	10	0,43
0,7	0,2	0,05	0	10	0,4
0,9	0,1	0,01	0	10,8	0,37
0,7	0,1	0,1	0	10,8	0,32
0,7	0,1	0,05	0	11	0,34
0,9	0,1	0,05	0	12	0,4
0,8	0,1	0,05	0	12,2	0,39
0,8	0,2	0,05	0	12,6	0,51
0,8	0,2	0,01	0	13,4	0,57
0,8	0,1	0,1	0	13,4	0,4
0,9	0,1	0,1	0	13,8	0,42
0,8	0,25	0,01	0	14,8	2,69
0,7	0,2	0,1	0	18,8	0,72
0,8	0,1	0,01	0	19,6	0,66
0,9	0,2	0,01	0	21	0,9
0,7	0,1	0,01	0	27,6	0,97
0,7	0,25	0,1	0,4	34,4	1,43

Figure 3 - Grid Search Results

Experimental Setup and Results

After obtaining the optimal parameters, we evaluated the effectiveness of our genetic algorithm in solving Sudoku puzzles by conducting a series of experiments across different levels of difficulty, namely, Easy, Medium, Hard and Extreme. These categories reflect the typical gradations in Sudoku puzzle complexity, determined by the number and distribution of pre-filled cells.

The results are shown in the table below. To ensure consistent results, we ran each grid 10 times with a population of 50000 and 200 generations for each one of them.



Difficulty	Avg Solved	Avg Runtime(min)	Avg Fitness	Best Runtime(min)	Best Fitness
Easy	100.00%	0,09	0	0,08	0
Medium	100.00%	0,4	0	0,18	0
Hard	100.00%	0,9	0	0,09	0
Extreme	10.00%	13,56	1,9	2,86	0

Figure 4 - Difficult Levels Results

The easy and medium levels were effortlessly solved by the algorithm with little to no runtime. For the hard level, it was noticeable that the algorithm performs equally well as in the previous levels due to its robust combination of genetic operators and efficient initial population setup. The accompanying graph illustrates the fitness progression during one of the runs for the hard level. As shown, the algorithm starts with a relatively low fitness and rapidly converges to an optimal solution, demonstrating its capability to handle more challenging puzzles efficiently.

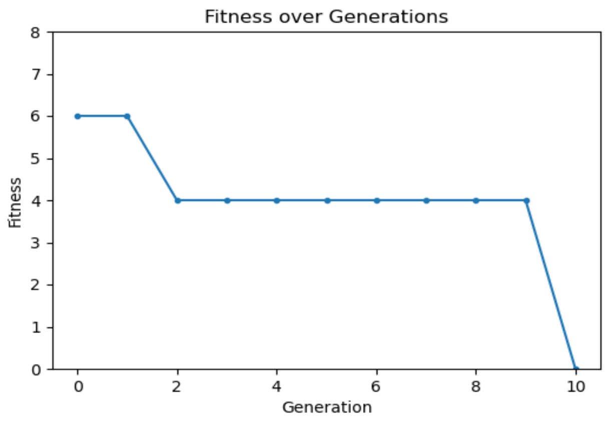
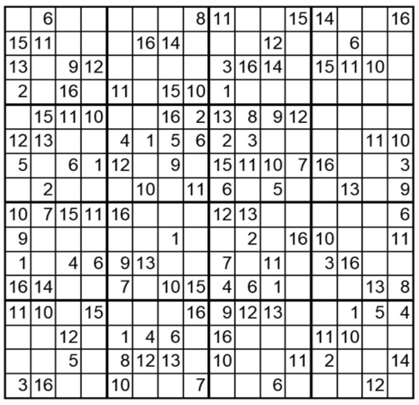


Figure 5 - Fitness Plot over Generations - Hard Level

As we move to the extreme level, the algorithm begins to encounter difficulties. Often, it gets stuck with a fitness of 2, meaning that the current best grid has two errors. However, it is important to mention that due to the combinatorial nature of Sudoku puzzles, having such low fitness is not correlated to being close to the final solution since there can be many ways to rearrange numbers while maintaining the same fitness value. And so, the algorithm converges to a suboptimal solution, leading to longer runtimes as it frequently runs through all generations to reach the generational termination condition.

To evaluate the performance of our genetic algorithm on different grid sizes, we also tested a 16x16 Sudoku puzzle. The algorithm ran for 16 minutes and achieved a best fitness score of 10. This result indicates that while the algorithm is capable of handling larger grids, the time complexity increases significantly, leading to longer runtimes and more challenging convergence to optimal solutions. The graph on the right illustrates how the fitness improved over the generations, showcasing the algorithm's ability to progressively reduce errors, though achieving complete accuracy remains difficult in larger grids.



	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

Figure 6 - 16x16 Sudoku Puzzle

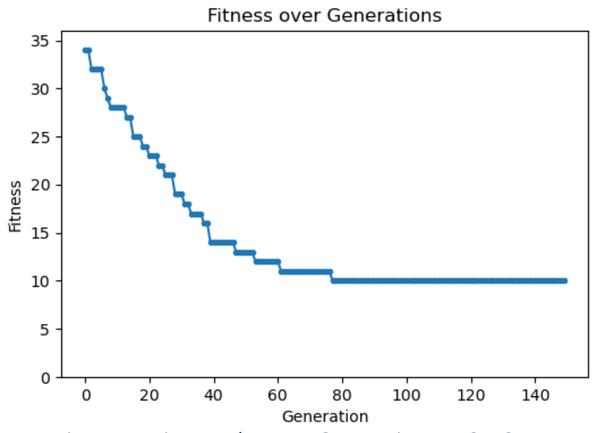


Figure 7 - Fitness Plot over Generations - 16x16

Further Work

As mentioned earlier, fitness sharing was a potential improvement for our algorithm. However, due to computational complexity when working with other operators, large populations, and numerous generations, we decided against its implementation. Given more time, this approach could have been reconsidered to enhance performance. Additionally, computational resources posed a challenge when attempting to scale our methods to larger grid sizes, such as 16x16 or even 25x25.

We could have explored different methods for initialising the population to further reduce the initial fitness scores. This includes experimenting with other fitness functions and incorporating different crossover, mutation, and selection operators.

One potential area for future exploration is integrating Sudoku-solving strategies into our genetic algorithm's operators and fitness functions. For example, implementing crossover operators that prioritise maintaining the integrity of rows, columns, or sub grids could ensure fewer rule violations during the crossover process. Developing mutation strategies that incorporate common Sudoku-solving techniques such as naked pairs, hidden singles, and box-line reductions could guide the mutation process by prioritising changes that are more likely to lead to a valid solution. Additionally, more advanced techniques like X-Wing, Swordfish, and XY-Wing could be considered. Enhancing the fitness function by incorporating heuristics from Sudoku-solving strategies, such as rewarding moves that align with solving techniques or penalising rule violations, can improve the algorithm's efficiency.

Conclusion

This project showcased the application of Genetic Algorithms (GAs) to solve Sudoku puzzles, effectively demonstrating GA's capability in combinatorial optimization tasks. We tested various operator combinations and identified that elitism significantly enhances performance. The GA8 combination, involving **Single Point Crossover**, **Valid Mutation**, and **Random Selection**, proved most effective, solving easy to hard puzzles efficiently, although extreme puzzles posed greater challenges, leading to suboptimal solutions and longer runtimes.

Overall, our approach successfully highlighted the importance of a well-constructed initial population and the impact of different genetic operators on convergence and solution quality. The project underscores the potential of GAs in solving complex optimization problems and offers insights into areas for further improvement, such as refining fitness functions and exploring more sophisticated genetic operators.

References

D, David. “Solving Sudoku Puzzles with Genetic Algorithm.” *Road to ML*, 9 Nov. 2019, nidragedd.github.io/sudoku-genetics/.

“Genetic-Algorithm/Sudoku_solver.ipynb at Main · MojTabaa4/Genetic-Algorithm.” *GitHub*, github.com/MojTabaa4/genetic-algorithm/blob/main/sudoku_solver.ipynb.

“Sudoku-Genetic-Algorithm/Sudoku.py at Master · Ctjacobs/Sudoku-Genetic-Algorithm.” *GitHub*, github.com/ctjacobs/sudoku-genetic-algorithm/blob/master/sudoku.py.

Vanneschi, Leonardo, and Sara Silva. *Lectures on Intelligent Systems*. *Natural Computing Series*, Springer Science+Business Media, 1 Jan. 2023. Accessed 2 June 2024.