Sudoku Puzzle Solving using Genetic Algorithms
CIFO Project Report

João Loureiro, student nº 202221219

https://github.com/newJL/CIFO_Project_Sudoku_JL.git

1. Introduction

The objective of this project is to implement a Genetic Algorithm (GA) to solve Sudoku puzzles efficiently. The developed GA algorithms are designed to solve any randomly generated, valid and solvable Sudoku puzzle, applicable to any 9x9 Sudoku puzzle rather than just a single specific one.

Sudoku, a combinatorial number-placement puzzle, presents a unique challenge due to its constraints: each number from 1 to 9 must appear exactly once in each row, column, and 3x3 sub-grid. This project explores various components of GAs to optimize the Sudoku-solving process, including the use of different selection approaches and experiments with various mutation and crossover operators. The project tests these configurations by running multiple experiments, analyzing and comparing the results.

2. Representation choice

The representation chosen for the Sudoku puzzle is a 9x9 grid where each cell can hold a number from 1 to 9. This grid-based representation is intuitive and directly corresponds to the standard Sudoku puzzle format, making it easy to implement and manipulate within the GA framework. This representation was chosen mainly due to constraint handling *i.e.* making it easy to *e.g.* check the validity of a solution using row and column operators and flexibility for genetic operators. The grid-based representation supports a variety of genetic operations. For example, crossover operations can exchange entire rows or columns between parent solutions, while mutation operations can change the value of a single cell or swap values between cells. These operations are easier to conceptualize and implement with a grid structure.

While the 9x9 grid representation was ultimately selected, other representations were considered during the design phase. One such alternative was a flat array representation which was discarded due to unnecessary added complexity in constraint checking and overall bad vibes.

3. Fitness function

3.1- Design

The fitness function in this genetic algorithm is designed to evaluate the closeness of a given Sudoku solution to being a valid and complete Sudoku puzzle. Higher scores indicate better solutions. Here's a breakdown of how this fitness function works:

The score increases from zero as the solution becomes closer to a valid Sudoku solution. The function first checks each row of the Sudoku grid. For each row, it counts the number of unique numbers present, as a correct Sudoku row should contain every number from 1 to 9 exactly once. The function also counts the number of empty cells in the row; these are penalized because a complete Sudoku solution should have no empty cells. The difference between the number of unique numbers and the number of empty cells is added to the fitness score, increasing the score when there are more unique numbers and fewer empty cells in a row. Similarly, the function examines each column of the Sudoku grid, counting the unique numbers and the empty cells in each column. Just like with rows, the difference between these counts is added to the fitness score. The function also evaluates each of the nine 3x3 sub-grids within the Sudoku grid, counting the unique numbers and the empty cells within each box while adding the difference to the fitness score.

The maximum fitness score in this Sudoku genetic algorithm is 243, calculated as follows: 81 points each from having all unique numbers in 9 rows, 9 columns, and 9 sub-grids (boxes). Each valid row, column, and box contributes 9 points, totaling 243 (9x9x3 = 243).

3.2. Why This Fitness Function Was Chosen

The fitness function effectively penalizes any row, column, or box that has duplicate numbers or empty cells. This guides the genetic algorithm towards finding solutions that are both complete and correct, as a valid Sudoku puzzle should have no duplicates and no empty cells. As the genetic algorithm iterates, even small improvements in the number of unique numbers or the reduction of empty cells will lead to an increase in the fitness score although how fast or slow this happens will depend on the characteristics of the genetic operators used.

To prevent the algorithm from getting stuck in a local minimum, additional mechanisms were implemented in the code. For instance, the algorithm stops if the fitness score has not improved for a set number of generations.

4. Configuration Benchmarking- Experimentation and Findings

4.1. Evaluating Criteria and Results

In this project, several configurations of a base genetic algorithm using the same fitness function were tried. The primary configurations included variations in selection approaches, mutation operators, and crossover operators. Stemming from these primary configurations, other parameters were tested such as population number, maximum number of generations and tournament size, however given that these parameters remained relatively unchanged throughout configurations, we will be focusing on comparing the performance between the various configurations. These were the main tested:

- Selection Approaches: Tournament Selection, Roulette Wheel Selection
- Mutation Operators: Swap Mutation, Random Resetting
- Crossover Operators: Uniform Crossover, One-point Crossover, Two-point Crossover

Each configuration was assessed based on four key performance metrics over 20 runs: computational cost (average time per run in seconds), the number of times the maximum fitness score (243) was achieved, the average number of generations required to find the solution, and the average fitness value per run. The results can be verified on figure 1 below.

GA Results Summary

	GA Tested	Computational Cost (seconds)	Max Fitness Count	Generations to Solution (avg)	Fitness Value (avg)
0	Base	1.58	14	141.75	242.1
1	Roulette Wheel	1.16	16	249.15	242.5
2	Roulette Wheel (No Elitism)	0.99	0	349.0	219.6
3	Other Mutation Method	3.09	1	339.95	234.5
4	Uniform Crossover	3.21	3	301.7	236.7

Figure 1

The base configuration, used tournament selection, swap mutation and single and two point crossover. This crossover method works by applying **single-point crossover on a**

row level: the crossover process starts by selecting a random row index (row_index). This means that the offspring will inherit all rows from the first parent up to this row index and all rows from the second parent after this row index. **Two-point crossover, on a column level:** Within the row selected by row_index, the crossover further specifies a column index (col_index). The elements up to this column index in the selected row are inherited from the first parent, and the elements after this column index are inherited from the second parent. This introduces a second crossover point within the selected row.

The base configuration performed relatively well, attaining max fitness scores on 14 out of 20 solutions. It had a computational cost of 1.58 seconds per run and required an average of 142 generations to solve the puzzle, with an average fitness value of 242.10. The configuration using roulette wheel selection performed the best in terms of achieving the maximum fitness score, reaching it 16 times out of 20 runs. This approach had the lowest computational cost of 1.16 seconds per run and an average fitness value of 242.50. However, it required more generations (249.15) to find the solution compared to the base configuration.

Regarding the Roulette Wheel Selection (without Elitism), this configuration did not achieve the maximum fitness score in any of the 20 runs, highlighting the importance of elitism in maintaining high-quality solutions. It had the lowest computational cost of 0.99 seconds per run but required the highest number of generations (349) to converge, with a significantly lower average fitness value of 219.60. Using an alternative mutation method (random resetting), this configuration showed poor performance, achieving the maximum fitness score only once. It had a higher computational cost of 3.09 seconds per run and required 340 generations on average, with an average fitness value of 234.50. The GA configuration with uniform crossover achieved the maximum fitness score three times. It had the highest computational cost of 3.21 seconds per run and required 302 generations on average to find the solution, with an average fitness value of 236.70.

The results indicate that the **roulette wheel selection** approach, particularly with elitism, is highly effective in achieving high fitness scores efficiently. However, the **base configuration using tournament selection** also performed well, striking a balance between computational cost and the number of generations required. The importance of elitism was underscored by the poor performance of the roulette wheel selection without elitism. While the alternative mutation and uniform crossover configurations showed some potential, they generally required more computational resources and generations to converge to a solution.

4.2. Impact of operators and use of Elitism

Selection methods like tournament selection and roulette wheel selection with elitism provided effective convergence. In contrast, the absence of elitism led to poor

performance, highlighting its crucial role in preserving the best solutions. Mutation operators also played a vital role; swap mutation balanced convergence effectively, while random resetting introduced too much disruption, resulting in slower convergence. Regarding crossover operators, the crossover type from the base configuration facilitated steady convergence, whereas uniform crossover, although promoting diversity, required more time to settle on a solution, indicating a trade-off between maintaining genetic diversity and achieving quick convergence.

5. Improvements

The GA's produced valid Sudoku solutions in most runs, with the best configurations achieving optimal fitness values consistently. When revisiting this work in the future, the algorithm's performance could possibly be improved in the following ways:

- Dynamic Mutation Rate: Implementing a dynamic mutation rate that adapts based on the convergence behavior of the GA could enhance the exploration capabilities of the algorithm.
- Hybrid Approaches: Combining GA with local search techniques like Simulated Annealing or Hill Climbing could improve solution quality and convergence speed.
- Parallel Processing: Leveraging parallel processing to evaluate fitness functions and perform genetic operations could significantly reduce computation time.

6. Conclusion

This project successfully demonstrated the capability of Genetic Algorithms in solving Sudoku puzzles. Through experimentation with various selection methods, mutation operators, and crossover techniques, the most effective configurations were identified, leading to high-quality solutions. The results underscored the significance of operator selection and parameter tuning in optimizing GA performance.

This project not only showcased the potential of GAs in solving Sudoku puzzles but also cemented their application to a broader range of combinatorial optimization challenges.