Computational Intelligence for Optimization Project

MASTER DEGREE PROGRAM IN DATA SCIENCE AND ADVANCED ANALYTICS

Implementation of Genetic Algorithms to evolve solutions to a Sudoku problem

Strawberry Shortcake group

Andriani Kakoulli, number: 20230484

Eugénia Rosário, number: 20220598

Kenza Boukhris, number: 20230604

Raquel Mendes, number: 20230596

GitHub link: https://github.com/raky55/StrawberryShortcake

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

Table of Contents

Introduction

This work aims to evolve solutions to the sudoku problem, using genetic algorithms (GAs). The sudoku consists of a logic-based combinatorial number-placement puzzle where the player has to fill a 9x9 grid with digits of a range from 1 to 9 such that each row, column, and 3x3 subgrid contains all nine digits, without repeating. The challenge varies based on how many numbers are initially given and their placement. To address this problem, we have implemented a solution in Python, employing genetic algorithms to explore potential solutions iteratively and improve them over time.

The field of optimization and search offers valuable tools for solving complex puzzles like Sudoku. GAs are a prominent method within evolutionary computing that can effectively simulate natural selection and stand out in exploring vast solution spaces. These algorithms are highly adaptable, handling discrete and non-continuous problems, enabling them to uncover multiple viable solutions across diverse and complex scenarios. This adaptability allows GAs to efficiently navigate challenging landscapes, consistently finding optimal or near-optimal solutions.

Initialization

In genetic algorithms, the initialization step is one of higher importance, given that it sets the foundation for the subsequent evolutionary process. This stage consists of defining an initial population of potential solutions, from which the algorithm will iteratively improve, to find an optimal or near-optimal solution.

The particular case of solving the sudoku puzzles through the use of GAs is no exception, so we started by creating a set of acceptable solutions for the same, resorting to a function on Python that has a base Sudoku grid as a starting point to set the framework for generating initial solutions. This grid is already filled with some numbers, which are placed in specific, fixed spots that can never be changed, according to the puzzle's given conditions. Each individual in the initial population is a strategic result of the attempt to fill the blank spots left in the base grid; which is not a random event, but instead has to comply with the already mentioned Sudoku's rules.

The first thing the algorithm does is to identify the empty positions on the grid. Secondly, it calculates which numbers are not already present in each row, varying from 1 to 9 - these are the permissible values per row. Finally, it shuffles the determined permissible values and randomly assigns them to the available positions on the grid. This process ensures that each solution is unique, enriching the genetic pool and broadening the search space. Such diversity is essential for the genetic algorithm to explore and optimize solutions more effectively, preventing premature convergence on suboptimal outcomes.

Fitness function

In the context of GAs, the fitness function plays a crucial role by evaluating how effective each individual (or solution) is at meeting the optimization problem requirements, in our case the puzzle's. In other words, the main objective is to quantify the closeness of a given solution to the ideal solution. Our fitness function, built specifically for the Sudoku problem, aims at this primarily by assessing the number of conflicts present.

A conflict in a Sudoku puzzle occurs when a number appears more than once in any row, column, or 3x3 subgrid (blocks or boxes). Each duplicate within these constraints contributes to a higher conflict score, indicating a lower fitness level. The goal is to minimize these conflicts, with a perfect score being zero conflicts, which means the Sudoku is solved correctly.

Our algorithm operates in a way that first identifies the rows and columns conflicts. It calculates the number of unique numbers present for each row and column and subtracts this count from 9 (the total number of unique numbers expected in a row or column). The result is the number of conflicts (missing unique numbers) for that row or column. Then the total conflict score for rows and columns is accumulated to provide a part of the overall fitness score. After completing this task, the algorithm looks for subgrid conflicts, dividing the grid into nine 3x3 subgrids. The function evaluates each subgrid separately and for each subgrid, it extracts all the numbers and counts the unique numbers present. Similar to rows and columns, the number of unique numbers in each subgrid is subtracted from 9 to determine the conflicts in that subgrid. These conflicts are then added to the total conflict score from the rows and columns.

The overall fitness of an individual is the sum of all the conflict scores from the rows, columns, and subgrids. A lower total conflict score signifies a higher fitness level, as it indicates fewer rule violations in the Sudoku grid, being 0 the ideal solution to reach.

Genetic Operators

These critical mechanisms used in GAs help drive the evolution of solutions toward optimality. These operators include selection, crossover, and mutation. Each plays a distinct role in generating diversity and improving fitness within the population.

When it comes to selection, we want to make sure that the algorithm is able to focus its evolutionary efforts on the most promising solutions. For that reason, three different methods have been explored to determine their effectiveness in picking individuals with lower conflict scores, hence propagating the traits of the solutions with better fitness into future generations.

Ranking Selection

The first method was ranking selection. As the name suggests, it is based on the ranked scores of population fitness, selecting only the top performers. In our case, the subset of individuals with the lowest occurrence of conflicts, indicating closer proximity to an ideal solution.

Roulette Wheel Selection (Fitness Proportionate Selection)

Secondly, we applied the roulette wheel selection. Since the sudoku problem is a minimization problem, the individuals are selected proportionally to the inverse of their fitness (1/fitness). The selection probability of an individual is determined by the inverse of its fitness score. The fitter the individual (the lower the conflict score), the higher the chances of being selected. However, even the least fit individual has a chance (however small) to be selected, which helps maintain the diversity of the population.

Tournament Selection

A number of individuals are randomly selected from the population. The individual with the lowest fitness (i.e. the best so far) in the selected subset is chosen. The above step is repeated until the number of selected individuals is fulfilled. The tournament selection strikes a compromise between applying selection pressure and preserving genetic variety.

Crossover Operator

The crossover operator takes two parent individuals and generates new individuals (offspring) by combining the genetic information of the parents, hopefully achieving the best combination of the parents' traits in the offspring. We chose to perform a single-point crossover by swapping rows:

A crossover point is randomly chosen within the Sudoku grid (chromosome). The offspring are then created by exchanging the corresponding segments of the parents' chromosomes at the crossover point. This operation combines good traits from the parents and creates potentially better solutions.

Mutation Operator

We chose to implement a swap mutation operator, with the goal of introducing random changes that would, therefore increase the genetic diversity within the population.

The function *mutate* randomly swaps values in a given offspring. Parallelly the *dynamic_mutation_rate* function decreases the mutation rate in each generation by adapting the mutation rate based on the progress of the algorithm, helping it find better and better solutions, without abandoning the exploration of different solutions.

Results of operators

Regarding the crossover and mutation operators, our choices proved to positively affect the convergence of the GA. Especially, the choice of implementing the dynamic mutation rate on each generation turned out to be very helpful for the faster convergence of the GA.

As for the selection operators, all three selectors were implemented and tested in the GA, though the conclusion was to use the ranking selection as it was the one to find the nearest solution to the global optimal in the least amount of time compared to the others. The tournament selection came second in terms of finding the optimal solution, but it could not be selected for implementation in the GA as it needed more time to select the individuals. The roulette wheel selection was the slowest and gave the worst results.

Extra Implementations

Elitism

The experimentation with elitism was a way to preserve the top-performing individuals across generations. The structure of the GA was maintained, with the extra touch of extracting the best individuals and replacing the worst ones with these top performers. Even though it sounded like a promising technique, it presented the drawback of landing on local minima. The conclusion was that the implementation of the GA with elitism led to a lack of diversity and thus, the algorithm converged prematurely. Therefore, it was not added in the final algorithm.

Keep the parents with better fitness than the offspring

Before replacing the population of parents with the population of offspring, a combined population of the two populations was created and evaluated in terms of fitness. The purpose of this tactic was to get the individuals with the best fitnesses from this combined population using the selection operator, having an upper bound of this selection of the initial population size. This implementation did not give the best results as it lacked diversity and, similarly to elitism, it got stuck to suboptimal solutions.

Reinitialize population

In trying to optimize the results of the GA, we employed the restart population strategy. This approach involved the reinitialization of the population at certain generations with the goal of introducing new individuals to the algorithm and thus overcoming the problem of premature convergence. The implementation of this idea was done by setting a parameter named *restart_inteval* which interrupted the smooth iteration of generations in four parts and independently created a new population of individuals.

This approach proved successful as it enabled the algorithm to escape local minima. Though, the implementation of this approach did not seem to significantly affect the convergence of the GA. Hence, we chose to proceed with our initial approach of the algorithm.

Final GA

After implementing and comparing our initial approach with the three aforementioned techniques, we reached to the conclusion that the final genetic algorithm, that would solve the sudoku problem in the most efficient way and in the least amount of time, was an algorithm that would use as selection operator the ranking selection, single-point crossover and mutation with dynamic mutation rates, and without elitism.

Conclusion

To summarize, this project is a practical example of a concept that came across very often during the classes - the no free lunch theorem. This is a fundamental concept in optimization that brings us the notion that it is impossible to find an algorithm able to universally perform better, for all optimization problems.

While building and analyzing the performance of our genetic algorithm, designed to solve Sudoku puzzles, we acknowledge the importance of each one of the genetic operators' methods to achieve the desired outcomes. Having algorithmic strategies - such as the three different selection methods employed, crossover, mutation -, aligned with the ability to combine and test different approaches - for instance, implementing elitism to preserve superior traits, dynamic mutation rates to balance exploration and exploitation - gave a significant contribution to the success of this work.

Through this work we were also able to witness the effectiveness of genetic algorithms to solve complex puzzles, such as Sudoku. And even though, as already mentioned, there is no super algorithm, with thoughtful adaptation and parameter tuning we can reach a highly effective solution to a specific task.

Division of labor

The successful completion of our project was the result of the collaborative contribution of each member of the strawberry shortcake group. A basic outline of how we divided the tasks code-wise was:

Mrs Bouhkris was responsible for the initialization of the code and functions, Mrs Kakoulli took over the optimization of the code until reaching the best fitness.

As for the report writing:

Mrs Rosário generated most of the report, with the significant contribution of Mrs Mendes, and some final adjustments by Mrs Kakoulli.

Overall, each member of the group brought her strengths to the table, leading to a well-crafted project, and our teamwork and dedication were instrumental in achieving our ‘best fitness’.