Sudoku Solver: genetic algorithm implementation

Group 7:
Guilherme Guerra, 20230754
Lizaveta Barysionak, 20220667
Cadmo Diogo, 20211886
Oséias Beu, 20230524

Division of labor

Genetic representation of the problem – Guilherme Guerra, Lizaveta Barysionak
Code review, report – Cadmo Diogo, Oséias Beu

Introduction

Solving Sudoku puzzles presents an interesting challenge for genetic algorithms (GAs). Sudoku requires filling a 9x9 grid so each row, column, and 3x3 sub-grid contain all digits from 1 to 9 without repetition. While traditional methods use backtracking and constraint propagation, GAs offer a heuristic approach to efficiently search large solution spaces.

In this project, we implemented a GA to solve Sudoku puzzles of varying difficulty. Our GA uses a population of candidate solutions, each represented as a Sudoku board. The algorithm evolves this population through selection, crossover, and mutation, guided by a fitness function that measures the board's closeness to a valid solution.

The report represents methodology, analysis and results. To see the code follow the link **https://github.com/nebezgreshniy/Sudoku_Solver_-CI4O-Group_7.git**

Genetic representation

In our Sudoku solver the problem is represented in a way that allows for the application of genetic operations such as selection, crossover, and mutation.

We chose to represent the Sudoku board as a 9x9 numpy array because it provides a clear and intuitive way to manipulate the board and access its elements. The board representation is straightforward for operations such as crossover and mutation, which involve swapping or rearranging rows, columns, or sub-grids. Additionally, numpy arrays are efficient for numerical computations and allow easy implementation of fitness evaluation functions.

1. Initial Board Representation

A direct encoding method was used, where each candidate solution is represented as a 9x9 grid directly corresponding to the Sudoku puzzle's structure. This grid serves as the initial input to the algorithm.

2. Individual Representation

Each candidate solution (individual) in the population is represented as an object of the Individual class, which contains a 9x9 grid (board). The empty cells (zeros) in the initial board are filled with random numbers between 1 and 9 to create a complete board. This is done in the fill_zeros method.

3. Fitness Function

The fitness function evaluates how close a board is to a valid Sudoku solution. It counts the unique numbers in each row, column, and 3x3 block. The maximum fitness score is 243, which indicates a completely valid board with no repetitions.

4. Genetic Operators

Crossover Operators

- Single-Point Crossover: Two parent individuals are selected, and a single crossover point is chosen. The offspring are created by combining the genes from the parents at the crossover point.
- Uniform Crossover: Each gene in the offspring is randomly chosen from one of the two parents.

Mutation Operators

- Swap Mutation: Two random positions on the board are chosen, and their values are swapped.
- Scramble Mutation: A subset of the board is randomly shuffled.
- Inversion Mutation: A subset of the board is reversed.

Selection Methods

- Tournament Selection: A subset of the population is randomly chosen, and the individual with the highest fitness in this subset is selected as a parent.
- Rank Selection: Individuals are ranked based on their fitness, and selection is based on these ranks.

5. Genetic Algorithm Execution

The genetic algorithm manages the population and applies genetic operators iteratively to evolve the population towards an optimal solution.

Design of the Fitness Function

The fitness function is designed to evaluate the uniqueness of numbers in each row, column, and 3x3 sub-grid. It calculates the total number of unique numbers across all rows, columns, and sub-grids, summing these values to get the fitness score. A fully correct Sudoku solution should have a fitness score of 243 (81 unique numbers in rows, 81 in columns, and 81 in sub-grids).

Current implementation focuses solely on the uniqueness of numbers in rows, columns, and sub-grids.

We did consider alternative fitness functions, such as penalizing repeated numbers or using a different weighting scheme for rows, columns, and sub-grids. However, the chosen fitness function is simple and directly correlates with the correctness of the Sudoku puzzle, making it effective for this problem.

Experimental Setup

The configurations tested include various combinations of:

- Crossover Strategies: Single-point, uniform, cycle crossover.
- Mutation Strategies: Swap, scramble, inversion mutation.
- Selection Methods: Tournament selection, rank selection.
- Elitism: Inclusion or exclusion with varying elite sizes.

The configurations were evaluated based on the fitness score of the best individual after a fixed number of generations and the consistency of achieving high fitness scores.

Several configurations were tested, including different population sizes, mutation rates, crossover strategies, mutation strategies, and selection methods. The chosen configuration included:

- Population size: 300
- Mutation rate: 0.2
- Crossover strategy: single_point
- Mutation strategy: swap
- Selection method: tournament with tournament size 8
- Elitism: False (with an elite size of 15 when elitism was tested)

Best Configuration Determination:

- The "best" configuration was determined based on the ability to consistently reach a fitness score of 243 within a reasonable number of generations across multiple runs.

- Performance was evaluated by running the GA multiple times for each configuration and recording the number of generations required to reach the optimal solution.

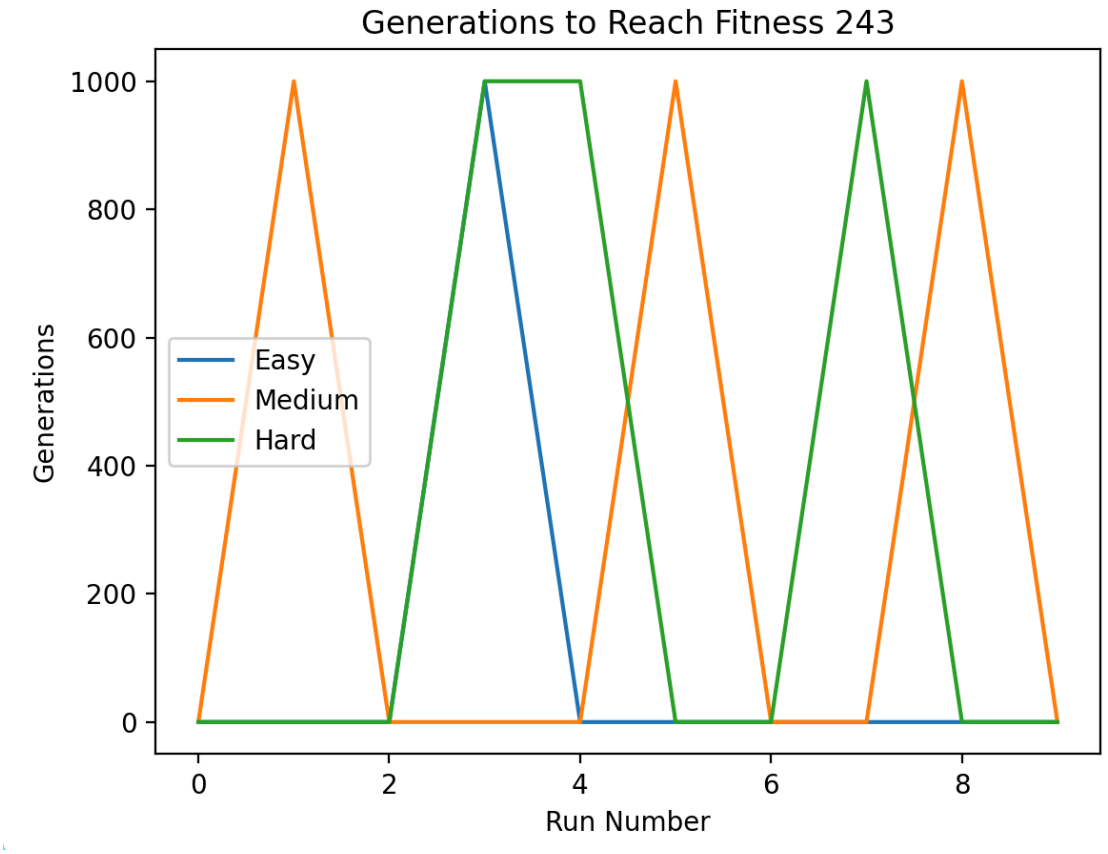

Fig.1 - Generations to Reach Fitness 243

The resulting plot (Fig.1) shows the number of generations it took to reach a fitness of 243 for each of the three difficulty levels (Easy, Medium, Hard) across multiple runs. For each difficulty level, there are 10 points corresponding to each run. The points indicate how many generations it took for the GA to find a solution with a fitness of 243 (a solved Sudoku puzzle) in each run.

The plot shows that the Genetic Algorithm consistently solves Easy Sudoku puzzles in fewer generations compared to Medium and Hard puzzles, with Medium puzzles exhibiting more variability and Hard puzzles consistently requiring more generations, reflecting their difficulty.

Ideally, for each difficulty level, the trend should show consistency in terms of the number of generations required to find a solution. If the plot shows significant variability or unexpected trends, it may indicate that certain configurations or parameters are affecting the GA's performance.

Affection on the convergence of GA

- Crossover Strategies: Single-point crossover, uniform crossover, and cycle crossover each have different impacts on how genetic material is mixed between parents. Single-point crossover was found to be effective in this context, but other strategies could also be explored further.
- Mutation Strategies: Swap mutation, scramble mutation, and inversion mutation each introduce genetic diversity differently. Swap mutation was chosen for its simplicity and effectiveness, but other strategies might offer better results depending on the problem specifics.
- Selection Methods: Tournament selection was used for its balance between exploration and exploitation. Rank selection was also tested but was found to be less effective in this context.

Elitism

Elitism was implemented but not used in the final configuration. When elitism was included, it ensured that the best individuals from the current generation were carried over to the next generation, preserving high-quality solutions.

- With elitism: Faster convergence but higher risk of premature convergence.
- Without elitism: Slower convergence but better exploration of the solution space.

Potential improvements

The genetic algorithm produced good results, often finding optimal solutions within a reasonable number of generations. However, there are several areas for potential improvement:

1. Parameter Tuning: Further fine-tuning of parameters such as mutation rate, population size, and selection pressure could improve performance.
2. Hybrid Approaches: Combining GA with other optimization techniques (e.g., simulated annealing, local search) could enhance performance and avoid local optima.
3. Adaptive Mechanisms: Implementing adaptive mutation rates or crossover probabilities that change during the run based on the population's performance could improve convergence.
4. Parallelization: Running the GA in parallel could speed up the process, allowing for larger population sizes and more extensive searches.