Group 6 Assignment Report

An Evolutionary Algorithm for solving Sudokus

Ben Bekir Ertugrul Eduard Iorga Haoran Liu
Moritz Schlitter

October 9, 2025

1 Introduction

Traditionally, a Sudoku is a logic based number placement puzzle. The objective of the puzzle is to fill a most commonly 9×9 sized grid with digits so that each column, row and 3×3 subgrid contain all of the digits from 1 to 9.

Solving such a puzzle programmatically falls into the category of search and optimization problems. These types of problems can be approached in different ways. One possible approach is depicted by evolutionary algorithms or in this case more specifically, genetic algorithms (GA)[1].

Inspired by the process of natural selection, GAs use biologically inspired operators such as selection, crossover and mutation to generate solutions to optimization and search problems[2].

This paper studies the efficiency of solving Sudoku puzzles with GA approaches. The objective is to compare the performance of different implementations of GAs. These approaches will additionally be compared to the naive search algorithm depth-first search (DFS). The puzzles explored will be of different difficulties for regular 9x9 grids and bigger 16x16 and 25x25 grids. This way the paper analyzes whether GAs are a viable option for efficiently solving Sudoku boards compared to naive search algorithms as the complexity of the puzzle increases.

This document is organized in the following way. Chapter one describes the objective of this study. Chapter two describes the problem of solving Sudoku puzzles and the motivation of using evolutionary algorithm approaches to solve this type of problem. Chapter three details the algorithm used in this study. The following chapter lays out the setup of the experiments run in this study. Lastly, chapter five presents and analyses the results of this paper.

2 Sudoku Puzzle Problem

2.1 Problem Description

Sudoku is a Japanese logical game that is played on a 9×9 grid, that are further divided into 3×3 subgrids. The objective of the game is to fill the grid with digits from 1 to 9, ensuring that each row, column, and subgrid contains each digit exactly once[3]. The puzzle starts with some cells already filled in, and the player must use logic and respect the base rule of the game in order to finish the puzzle.

Sudoku puzzles can vary in difficulty based on how many numbers they start with, the arrangement of these numbers and even the varying size of the Sudoku. They can go beyond the standard 9×9 grid, for example to 16×16 or 25×25 that we also use to test our algorithm on. Thus, we can think of the Sudoku problem as a graph coloring problem if it were to be expressed in a mathematical context. The 9×9 grid can be seen as graph that has 81 vertices, namely one vertex for each cell. Each vertex can be labeled with an ordered pair (x,y), where x and y are integers between 1 and 9 [4]. Two distinct vertices (x1,y1) and (x2,y2) are connected by an edge if and only if they are in the same row, column or subgrid:

- $x1 = x2$, which translates to same row
- $y1 = y2$, which translates to same column
- $\lfloor (x1-1)/3 \rfloor = \lfloor (x2-1)/3 \rfloor$ and $\lfloor (y1-1)/3 \rfloor = \lfloor (y2-1)/3 \rfloor$ which translates as same subgrid

As previously mentioned, the puzzle is completed when all the vertices have an integer between 1 and 9 assigned to them, in such a way that vertices that are joined by an edge don't have the same number assigned to them.

2.2 Evolutionary Approach

While Sudoku has a deterministic solution, evolutionary algorithms can be used in order to compare their performance with more traditional approaches such as the depth-first search algorithm. The paper *Evolutionary Algorithms and Sudoku* managed to implement an evolutionary algorithm that could very efficiently solve easy Sudoku puzzles[5]. The downside of that algorithm it that it is not as efficient when solving medium or hard puzzles. The way they approached the algorithm was to test constraints on different sets of spaces, namely the Hamming space and the Row Swap space. The drawn conclusions were that even though the Hamming space had a smoother fitness landscape, in the end the Row Swap space was more efficient and produced more optimal results.

Another approach was taken in the paper *Solving sudoku's by evolutionary algorithms with pre-processing*, where pre-processing was involved[6]. The pre-processing involved filling some sure numbers in the Sudoku through different methods, such as the naked single method, hidden single method, full house and lone rangers. The best method depends on on the type of puzzle, since each method performed differently depending on the case. After the pre-processing was done, different evolutionary algorithms were tested, including a genetic algorithm that uses crossover and mutation operators, a genetic algorithm that uses an ant colony optimization and one that firstly pre-processes the board and then applies the said genetic algorithm. The fitness function used in this paper is made of three components: the first one minimizes missing digits into rows and columns, the second one uses a sort of aging penalty in order to prevent the same best individual persisting over generations. Finally, the third one penalizes violations where a candidate digit conflicts with a given clue in its row or column. The conclusion they arrived at was that solving the Sudoku puzzle is faster when pre-processing is involved and that a hybrid algorithm between genetic algorithm and ant colony optimization is the most efficient with every type of Sudoku puzzle.

Lastly, the paper that influenced our approach the most was *Solving, rating and generating sudoku puzzles with GA*, which uses the genetic algorithm with Darwinian evolution as its inspiration[3]. This implies that each individual in the population is tested against a fitness function, that further decides whether the individual should be removed from the population or used as a parent to get closer to the desired result. New individuals are created through the use of crossover and mutation. Mutations is restricted to sub-blocks, using swap mutation as the primary operator. Further on, it was observed in the paper that using a cataclysmic mutation, specifically a restart of the population, every 2000 generations brought the best results when no solution was found. Similar to our project, the authors experimented with different fitness functions, but ended up using a function that counts missing or duplicate digits in each row and column, assigning penalties. The optimal value of the fitness function is 0. Furthermore, the authors introduced an aging penalty, adding +1 to the fitness of the best solution if it remains unchanged. The conclusion the paper arrived at is that the genetic algorithm could solve Sudoku puzzles efficiently, and even if there are better performing algorithms, they sometimes would fail to solve puzzles that the genetic algorithm could solve.

The motivation for our project is straightforward. We want to explore if a streamlined evolutionary algorithm with carefully chosen operators can achieve better performances than naive search algorithms, such as the depth-first search algorithm. As previously mentioned, our main inspiration for the evolutionary algorithm was the paper *Solving, rating and generating sudoku puzzles with GA*, thus we implemented an evolutionary algorithm that uses mutation and crossover operators while also taking an elitist approach regarding the selection of the parents[3]. Additionally, we also want to investigate the scalability of our evolutionary algorithm, thus we also tested it on a 16×16 and 25×25 puzzles. Evolutionary algorithms are well suited for this type of problem because they can handle large solution spaces effectively, balance exploration and exploitation, and adapt by integrating problem-specific heuristics. We believe that the Sudoku problem is a great choice for analyzing all the key aspects of evolutionary algorithms.

3 Genetic Algorithm

This chapter describes the implementations of two algorithms used to solve Sudoku puzzles. It starts out by briefly introducing depth-first search as the baseline measure and then goes on to explain how each of the genetic algorithms work.

3.1 Baseline

Depth-first search (DFS) is used as a deterministic baseline. It iteratively scans the board for the first empty cell, attempts candidate values 1. . . n, and backtracks to try alternative values if no candidate leads to a solution. This approach ensures a solution (given that the puzzle is solvable) and is simple to implement, which makes it a reliable reference for correctness and for measuring runtimes.

Algorithm 1 shows the implementation of the DFS algorithm in pseudo-code.

Algorithm 1 Depth-first search algorithm

```
Require: matrix
Ensure: true if solved, false otherwise
  $n \leftarrow$ length(matrix)
  for $i \leftarrow 1$ to $n$ do
      for $j \leftarrow 1$ to $n$ do
          if matrix$[i, j] = 0$ then
              for $num \leftarrow 1$ to $n$ do
                  if is_valid_move(matrix, $i$, $j$, $num$) then
                      matrix$[i, j] \leftarrow num$
                      if DFS(matrix) then
                          return true
                      end if
                      matrix$[i, j] \leftarrow 0$
                  end if
              end for
              return false
          end if
      end for
  end for
  return true
```

3.2 Implementations

This section presents and compares two genetic algorithms designed to solve Sudoku puzzles. Each implementation varies in its approach to selection, crossover, mutation strategies, and fitness evaluation.

The first GA implementation originated from initial project discussions and represents our first practical attempt to apply standard genetic operators to Sudoku. The second implementation builds on that design and performs better in general, which is why we use it as the representative GA in the following chapters.

Nevertheless, we decided not to omit the first GA from this paper, because it represents the iterative approach to finding an optimized solution, and it shows which algorithms tend to not work as good for Sudoku compared to the algorithms in the latter implementation.

The following sections document these algorithms and use the Sudoku board shown in Figure 1 as a reference for visualizing the selection and mutation step.

	2						3	1
7					3			
			1	4		2	9	
	5	2	7	6	4		1	8
	6	3		1	2	7	5	9
	7	8				4		
2			3	7				5
	1					9		
5	4			8	1			

Figure 1: Initial Sudoku puzzle

3.2.1 Implementation 1

Algorithm 2 initializes a population by randomly filling empty cells and then iteratively evolves it using selection, per-cell crossover and mutation for up to a fixed number of generations. Most offspring are produced by top parents of the previous generation. Then, the population is replaced by the offspring, and eventually the best individual is returned.

A detailed description of how the fitness function, crossover and mutation work will be given afterwards.

Algorithm 2 Genetic Algorithm 1

```
Require: $initial\_sudoku$, $population\_size$, $mutation\_rate$, $max\_generations$
Ensure: Best individual found
  $population \leftarrow$ PopulateRandomly($initial\_sudoku$, $population\_size$)
  $generation \leftarrow 0$
  while $generation < max\_generations$ do
      $population \leftarrow$ SortByFitness($population$)                ▷ descending order
      if Fitness(first($population$)) $= max\_fitness$ then
          return first($population$)
      end if
      $next\_generation \leftarrow \emptyset$
      for $i \leftarrow 1$ to $population\_size \times 0.8$ do
          $parent_1, parent_2 \leftarrow$ Sample(top 20% of $population$)
          $child \leftarrow$ Crossover($parent_1$, $parent_2$)
          $child \leftarrow$ Mutate($child$, $mutation\_rate$, $initial\_sudoku$)
          $next\_generation \leftarrow next\_generation \cup \{child\}$
      end for
      for $i \leftarrow 1$ to $population\_size \times 0.2$ do
          $parent_1, parent_2 \leftarrow$ Sample($population$)
          $child \leftarrow$ Crossover($parent_1$, $parent_2$)
          $child \leftarrow$ Mutate($child$, $mutation\_rate$, $initial\_sudoku$)
          $next\_generation \leftarrow next\_generation \cup \{child\}$
      end for
      $population \leftarrow next\_generation$
      $generation \leftarrow generation + 1$
  end while
  $population \leftarrow$ SortByFitness($population$)                    ▷ descending order
  return first($population$)
```

Fitness function Let R_i, C_j and B_b denote the sets of distinct, non-zero digits present in row i, column j, and block b, respectively. Using zero-based indices ($i, j \in \{0, \dots, 8\}$) the block index is

$$b \;=\; 3\left\lfloor \frac{i}{3} \right\rfloor \;+\; \left\lfloor \frac{j}{3} \right\rfloor.$$

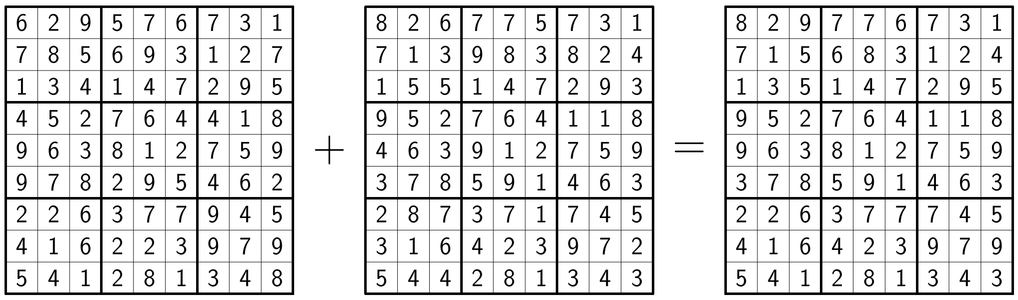
The fitness of a board is the total number of distinct digits across all rows, columns and 3×3 blocks:

$$F \;=\; \sum_{i=0}^{8} |R_i| \;+\; \sum_{j=0}^{8} |C_j| \;+\; \sum_{b=0}^{8} |B_b|.$$

This definition results in a maximum fitness score of 243 for 9×9 boards.

Our fitness function is inspired by prior work that sums distinct digits per row and column [7]; we extend that idea by also counting distinct digits in each 3×3 block to better reflect Sudoku constraints.

Crossover A new individual is created by crossing over two parents. The crossover process simply iterates over all the tiles that were not given in the initial puzzle and takes the value for that tile from either parent to copy it to the child.



6	2	9	5	7	6	7	3	1
7	8	5	6	9	3	1	2	7
1	3	4	1	4	7	2	9	5
4	5	2	7	6	4	4	1	8
9	6	3	8	1	2	7	5	9
9	7	8	2	9	5	4	6	2
2	2	6	3	7	7	9	4	5
4	1	6	2	2	3	9	7	9
5	4	1	2	8	1	3	4	8

$+$

8	2	6	7	7	5	7	3	1
7	1	3	9	8	3	8	2	4
1	5	5	1	4	7	2	9	3
9	5	2	7	6	4	1	1	8
4	6	3	9	1	2	7	5	9
3	7	8	5	9	1	4	6	3
2	8	7	3	7	1	7	4	5
3	1	6	4	2	3	9	7	2
5	4	4	2	8	1	3	4	3

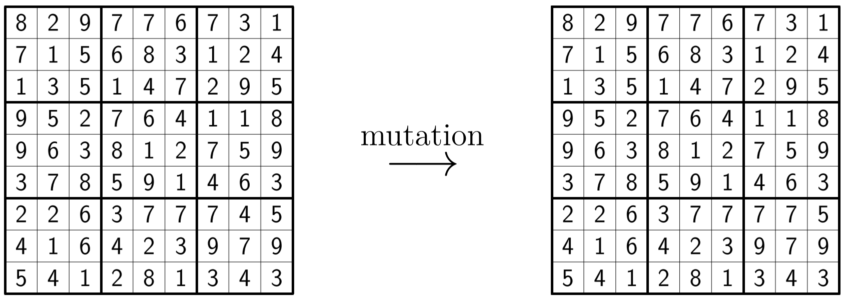
$=$

8	2	9	7	7	6	7	3	1
7	1	5	6	8	3	1	2	4
1	3	5	1	4	7	2	9	5
9	5	2	7	6	4	1	1	8
9	6	3	8	1	2	7	5	9
3	7	8	5	9	1	4	6	3
2	2	6	3	7	7	7	4	5
4	1	6	4	2	3	9	7	9
5	4	1	2	8	1	3	4	3

Figure 2: Exemplary selection step according to Implementation 1

In the figure above, two parents (left and middle) of the current generation form an individual of the next generation (right). One area for improvement is evident in row 7: Parent 1 contains two occurrences of the digit 7 and Parent 2 contains three. The offspring also has three 7s, while it would have been possible to generate a row with all distinct digits, namely **286371945**.

Mutation After the creation, the new individual undergoes up to *mutation_amount* mutations. A mutation means that a random tile changes its value. If that tile, however, happens to be a given tile from the initial puzzle, it is not altered.



8	2	9	7	7	6	7	3	1
7	1	5	6	8	3	1	2	4
1	3	5	1	4	7	2	9	5
9	5	2	7	6	4	1	1	8
9	6	3	8	1	2	7	5	9
3	7	8	5	9	1	4	6	3
2	2	6	3	7	7	7	4	5
4	1	6	4	2	3	9	7	9
5	4	1	2	8	1	3	4	3

$\xrightarrow{\text{mutation}}$

8	2	9	7	7	6	7	3	1
7	1	5	6	8	3	1	2	4
1	3	5	1	4	7	2	9	5
9	5	2	7	6	4	1	1	8
9	6	3	8	1	2	7	5	9
3	7	8	5	9	1	4	6	3
2	2	6	3	7	7	7	7	5
4	1	6	4	2	3	9	7	9
5	4	1	2	8	1	3	4	3

Figure 3: Exemplary mutation step according to Implementation 1

Figure 3 illustrates a mutation step with $mutation\_amount = 2$. Meanwhile, only a single tile (row 7, column 7) was actually modified to have the value 7. The other candidate position coincided with a given cell and therefore could not be changed. This example highlights a weakness of the implementation: mutations can be wasted when selected positions are immutable, and a single mutation may decrease fitness significantly by introducing additional conflicts (the number of 7's increased to 4 in the respective row, to 2 in the respective column, and to 3 in the respective grid).

Problem This implementation leads to a situation where the diversity of the population decreases rapidly, resulting in convergence to suboptimal solutions. On many runs, the algorithm creates more than 100,000 generations before finding a valid solution. To counter this problem, we have tried to adjust the mutation rate and the population size as well as introducing new random individuals in each generation. However, these adjustments only led to marginal improvements in performance.

3.2.2 Implementation 2

Algorithm 3 creates a population with unique rows and improves it using elitism, tournament selection, row-wise crossover and swap-based mutation. Each generation the population is evaluated, the best *elite_size* individuals are carried over unchanged, and the remaining offspring are produced from parents chosen by tournament selection. Children copy whole rows from parents to preserve row validity and then undergo mutations that swap non-given tiles within rows. The algorithm terminates when a perfect solution is found or the maximum number of generations is reached.

A description of the fitness function, crossover, and mutation will once again be given afterwards.

Algorithm 3 Genetic Algorithm 2

```
Require: $population\_size$, $elite\_size$, $max\_generations$
Ensure: Best individual found
  $population \leftarrow$ PopulateWithUniqueRows($population\_size$)
  $best\_individual \leftarrow$ arbitrary element of $population$
  for $generation \leftarrow 1$ to $max\_generations$ do
      for all $ind \in population$ do
          ComputeFitness($ind$)
      end for
      $population \leftarrow$ SortByFitness($population$)          ▷ ascending order
      if Fitness(first($population$)) $<$ Fitness($best\_individual$) then
          $best\_individual \leftarrow$ first($population$)
      end if
      if Fitness($best\_individual$) $= 0$ then
          return $best\_individual$
      end if
      $next\_generation \leftarrow$ Top($elite\_size$, $population$)
      $parents \leftarrow$ TournamentSelection($population$)
      for $i \leftarrow 1$ to $population\_size - elite\_size$ step 2 do
          $parent_1 \leftarrow parents[i]$
          $parent_2 \leftarrow parents[i+1]$
          $child_1 \leftarrow$ Crossover($parent_1, parent_2$)
          $next\_generation \leftarrow next\_generation \cup \{child_1\}$
          if $|next\_generation| < population\_size$ then
              $child_2 \leftarrow$ Crossover($parent_2, parent_1$)
              $next\_generation \leftarrow next\_generation \cup \{child_2\}$
          end if
      end for
      $population \leftarrow next\_generation$
  end for
  return $best\_individual$
```

Fitness function Let C_j and B_b denote the sets of distinct, non-zero digits in column j and block b, respectively, and let n be the board size. The fitness function counts the number of duplicates in each column, and 3x3 subgrid:

$$F = \sum_{j=0}^{n-1}\Big(n - |C_j|\Big) + \sum_{b=0}^{n-1}\Big(n - |B_b|\Big).$$

A value $F = 0$ indicates a valid solution. This formulation is similar to that of the previous work , which also iterates over columns and sub-blocks but assigns a value of 0 when a unit is valid and 1 when it contains any violation[8]. By contrast, our fitness function counts the number of duplicate digits per column and block, giving a more fine-grained penalty (e.g. a column with three duplicated entries contributes 3 rather than 1).

Crossover The crossover step is done by iterating through the rows of the sudoku board and copying each row from either parent. This ensures that the children will still respect the uniqueness of numbers in each row.

9	2	4	6	7	8	5	3	1
7	9	1	5	2	3	8	6	4
8	3	5	1	4	6	2	9	7
3	5	2	7	6	4	9	1	8
4	6	3	8	1	2	7	5	9
6	7	8	5	3	9	4	1	2
2	9	6	3	7	4	1	8	5
3	1	7	2	6	5	9	8	4
5	4	9	6	8	1	3	2	7

$+$

8	2	4	7	9	6	5	3	1
7	9	1	5	2	3	6	8	4
3	6	7	1	4	8	2	9	5
3	5	2	7	6	4	9	1	8
8	6	3	4	1	2	7	5	9
1	7	8	5	3	9	4	2	6
2	9	6	3	7	4	1	8	5
3	1	7	8	5	6	9	4	2
5	4	6	9	8	1	3	2	7

$=$

8	2	4	7	9	6	5	3	1
7	9	1	5	2	3	6	8	4
8	3	5	1	4	6	2	9	7
3	5	2	7	6	4	9	1	8
8	6	3	4	1	2	7	5	9
1	7	8	5	3	9	4	2	6
2	9	6	3	7	4	1	8	5
3	1	7	2	6	5	9	8	4
5	4	6	9	8	1	3	2	7

Figure 4: Exemplary selection step according to Implementation 2

In Figure 4, it is easy to see that the very first row is taken from the first parent (left), while row 8 is copied from the second parent (center). As a result of the crossover step, duplicate numbers can now only emerge in columns or subgrids, which is an improvement compared to Implementation 1.

Mutation For the mutation, we iterate all rows and select two non-given tiles in that row to swap their values with probability *mutation_rate*. This row-wise swap was inspired by swap-mutations within subgrids and is also consistent with other work that performs swap mutations inside random rows to ensure row constraints are preserved [8, 7, 3]. The cited paper additionally applies a reinitialisation mutation step, but in our implementation we only adopt the row-swap component of the mutation logic.

To increase diversity when the fitness stagnates over *generations_stuck* generations, we also introduce an adaptive mutation rate which increases the mutation rate temporarily until progress resumes. The mutation rate is capped at a maximum of 0.3, which alignes with the value used in related work [7].

8	2	4	7	9	6	5	3	1
7	9	1	5	2	3	6	8	4
8	3	5	1	4	6	2	9	7
3	5	2	7	6	4	9	1	8
8	6	3	4	1	2	7	5	9
1	7	8	5	3	9	4	2	6
2	9	6	3	7	4	1	8	5
3	1	7	2	6	5	9	8	4
5	4	6	9	8	1	3	2	7

$\xrightarrow{\text{mutation}}$

8	2	4	6	9	7	5	3	1
7	9	1	5	2	3	6	8	4
8	3	5	1	4	6	2	9	7
3	5	2	7	6	4	9	1	8
4	6	3	8	1	2	7	5	9
1	7	8	5	3	9	4	2	6
2	9	6	3	7	4	1	8	5
3	1	7	2	6	5	9	8	4
5	4	6	9	8	1	3	2	7

Figure 5: Exemplary mutation step according to Implementation 2

Figure 5 shows a mutation step with a rate of 20%, leading to swap-mutations in row 1 and row 5.

Problem The second genetic algorithm also often converges to local optima and loses diversity, limiting further improvement. To overcome stagnation we reinitialised a large portion of the population after the fitness has stopped decreasing which is related to the cataclysmic mutation/restart procedure discussed in *Solving, rating and generating sudoku puzzles with GA*[3]. This strategy often restored search progress but did not fully eliminate premature convergence. Furthermore, scalability remains an issue: larger boards increase the combinatorial search space and therefore require proportionally larger populations and/or more generations to retain a comparable probability of finding a solution.

Despite these limitations, Implementation 2 performs much better in practice than Implementation 1, but the problems with premature convergence apply to both implementations and to genetic algorithms in general.

3.3 Solution representation

The solution representation is an $n \times n$ matrix where each entry is a number in the range from 1 to n. The solution is valid if and only if every number in a row, a column, and a respective block is unique.

4 Experimental part

This chapter describes the setup of the experiments run to study the efficiency of solving Sudoku puzzles with Implementation 2.

The tests will be run on a Macbook pro M1 max.

4.1 Choosing hyperparameters

Chapter 3 discusses the algorithm used for the experiments in detail. However, before the algorithm can be run, the hyperparameters for ***population_size*** and ***mutation_rate*** need to be chosen. The solution quality of a stochastic algorithm strongly depends on choosing the correct hyperparameters. Previous studies show that configuring hyperparameters of GAs using Bayesian Optimization leads to significantly better results than choosing them at random while keeping the computational time low[9].

Using Bayesian Optimization, the optimal values for the hyperparamters ***population_size*** and ***mutation_rate*** of the GA are evaluated as follows. First, a search space is defined. It specifies the lower and upper bounds of the hyperparameters. Next, an *objective function* tries to solve a Sudoku using the GA. It measures solving time and the final fitness. These measurements calculate a score. An *optimize* method tries to minimize the score by fitting a Gaussian Process model to predict performance across the parameter space. It iteratively chooses new parameter sets that are either promising or not well explored. This process is repeated for a set number of iterations. The result is the best found population size and mutation rate.

4.2 Running the experiment

The GA is run around 1000, 400 and 100 times for a maximum of 100.000 generations for 9×9 Sudoku puzzles of easy, medium and hard difficulty. Because of the drastic jump in complexity with 16×16 and 25×25 boards, the GA is run even less times for these puzzles. For the evaluation of the performance the following statistic of the algorithm are tracked:

1. Number of successfully solved boards
2. Execution time
3. Generations
4. Solution Quality

To compare the performance of the GA, a DFS algorithm is run on Sudoku boards of the same complexity. For the evaluation of the performance the following statistic of the algorithm are tracked:

1. Number of successfully solved boards
2. Execution time

4.3 Sudoku boards used in the experiment

Because of the high number of Sudoku boards needed to accurately evaluate the performance of the stochastic algorithm, the Sudoku grids are generated randomly as part of the experiment. The boards are generated in the following way. First, a solved board is generated using the DFS algorithm. Next, a random number is removed from the grid. The DFS algorithm tries to solve the puzzle again. If it is still solvable, repeat the previous steps until the requirement of given numbers is met for the different complexities of the Sudoku boards. For 9×9 grids, the easy difficulty gives between 40-50 numbers. Medium and hard puzzles have 30-39 and 25-29 givens respectively. The DFS algorithm works well for generating the boards because it has a very low run time on 9×9 grids. Additionally, it is a complete algorithm, meaning a solution will be found if it exists.

Because of the increased complexity, run time and reduced success rate of solving 16×16 and 25×25 puzzles, the algorithm will be run only on three 16×16 boards and one 25×25 board taken from [10, 11].

5 Results and Analysis

This chapter presents the performance of the GA on the boards of different complexities.

5.1 Bayesian optimization results

As mentioned in chapter 4, Bayesian Optimization is used to find the optimal population size and mutation rate for the GA. The results are shown in Figure 6.

5.1.1 Optimization progress

Chart one of Figure 6 shows that the difference between the best and the worst score is minimal, meaning that the GA is stable and the hyperparameters are selected well.

5.1.2 Population size vs Performance

From chart two of Figure 6 we can see a population size of around 100 and 900 can find solutions in a shorter time while the score is more discrete at population size 900. Thus, the average time of solving puzzles at population size 900 is costlier than at population size 150.

5.1.3 Mutation rate vs Performance

Chart three of Figure 6 presents that the mutation rate between 0.05 and 0.15 could often times not find a solution. When mutation rate reaches over 0.20, the number of failure cases decreased and no solution events happened less frequently.

5.1.4 Parameter space exploration

Chart four of Figure 6 shows that the best population size is 142 and the best mutation rate is 0.230. This result is based on the best score shown in the first chart.

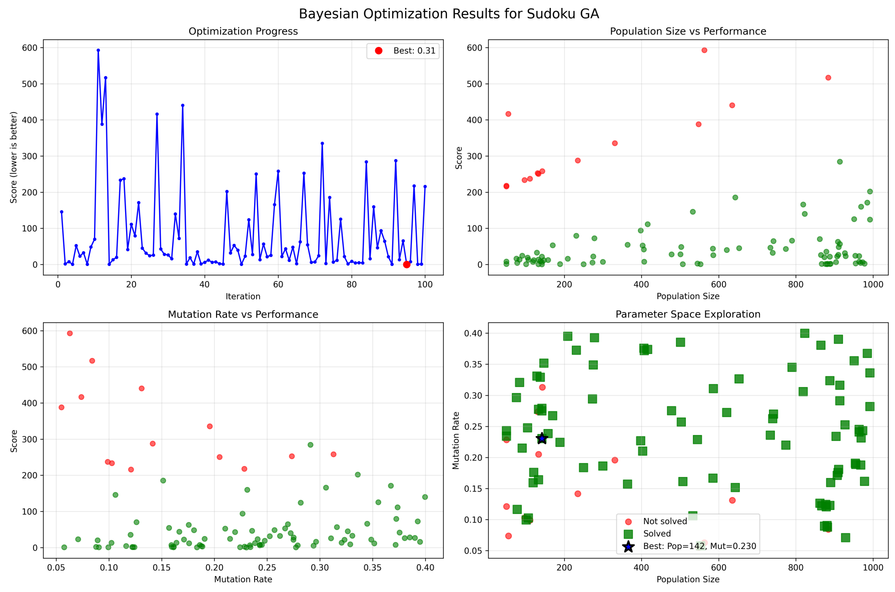


Figure 6: Bayesian optimization results.

5.2 Performance of the GA

We test easy, medium and hard difficulty puzzles in the same setting, but as mentioned in chapter 4 it is costly to test each difficulty 1000 times. Thus, we only test the easy difficulty for 1000 times, the medium one for around 400 times, and the hard one for 100 times.

5.2.1 Easy difficulty

The charts below shows the performance of the GA with easy difficulty puzzles. For every single board a solution was found by the GA. In over 500 times we find a solution in less than 0.5 seconds. Only around $\frac{1}{20}$ of the solutions are reached very slow which is more than 20s. Additionally, the average execution time is 5s, which is a good performance. Moreover, the execution time and generation have linear relationship which means that if the algorithm needs more generations to find the solution, the more execution time it will cost.

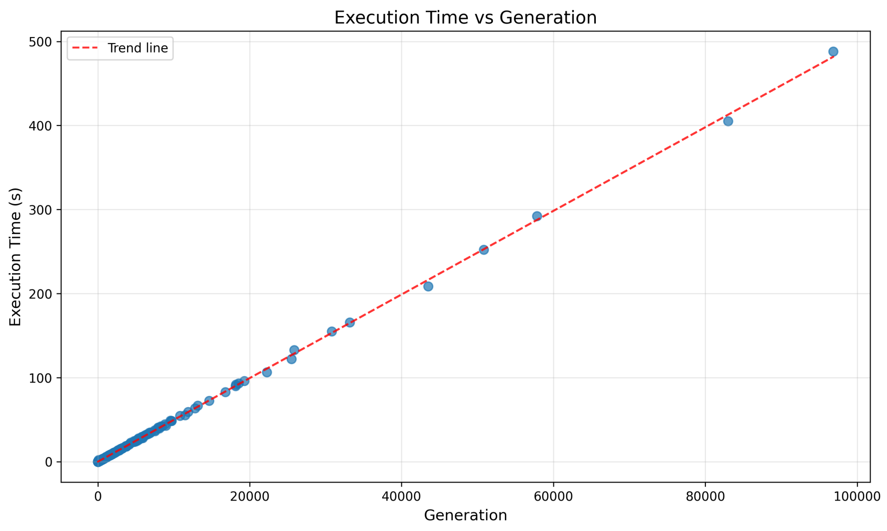


Figure 7: Generation vs execution time for easy difficulty puzzles.

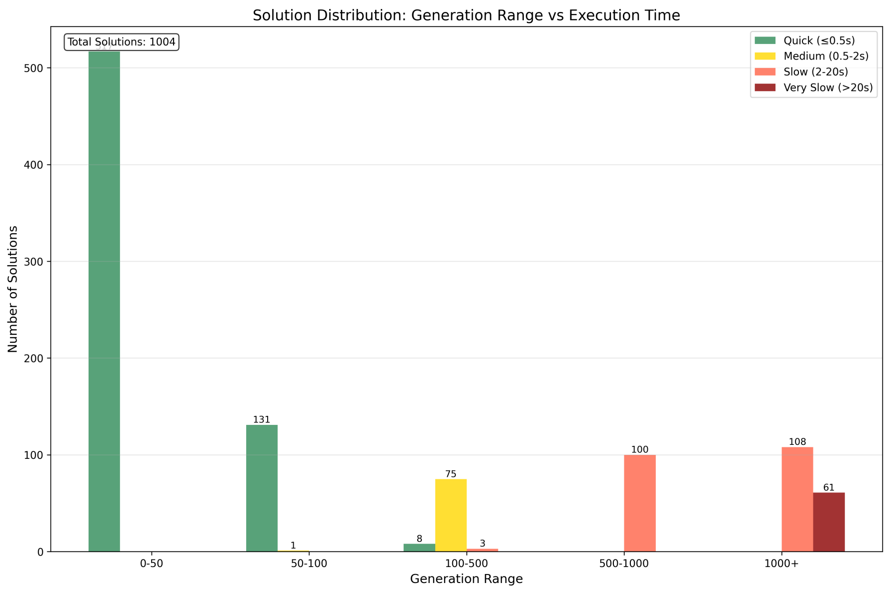


Figure 8: Generation and execution time distribution for easy difficulty puzzles.

Easy Difficulty Statistical Analysis Table 1 presents a comprehensive statistical analysis of the genetic algorithm performance on easy difficulty Sudoku puzzles over 1000 runs.

Table 1: Statistical Analysis of GA Performance on Easy Difficulty Puzzles

Metric	**Value**
Total Runs	1000
Successful Runs	1000 (100%)
Failed Runs	0 (0%)
Execution Time	
Average	5.99s
Range	0.03s – 488.11s
Successful Runs (Avg)	5.99s
Failed Runs (Avg)	0s
Generations	
Average	1202.4
Range	6 – 96862
Generations per Second	204.55
Solution Quality	
Average Violations	0.0
Failed Runs Violations	0 (constant)

5.2.2 Medium difficulty

The charts below shows the performance of the GA with medium difficulty. At 91,4%, a solution for most of the boards could be found. The time cost of finding a solution at quick and medium time only plays a small role in the total cases. Additionally, the proportion of the getting solution in very slow is 60%, which is much higher compared to easy difficulty. Meanwhile, the execution time and generation still remain a linear relationship.

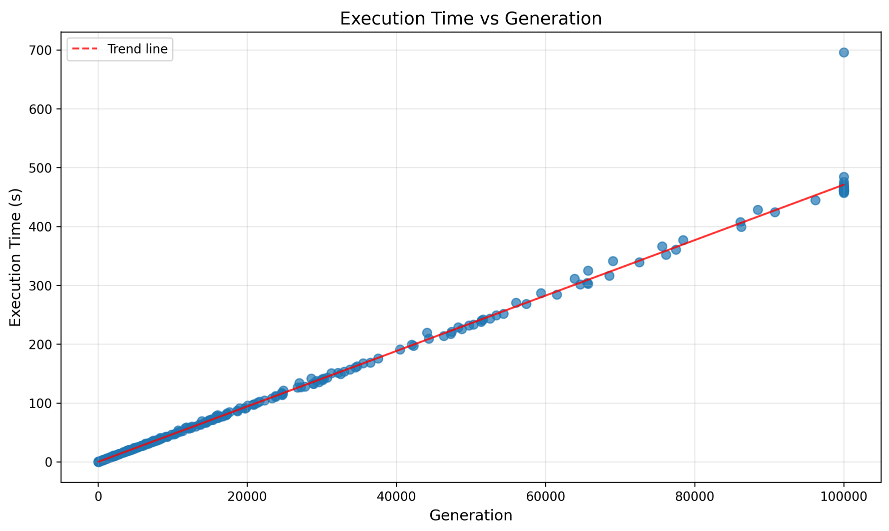


Figure 9: Generation vs execution time for medium difficulty puzzles.

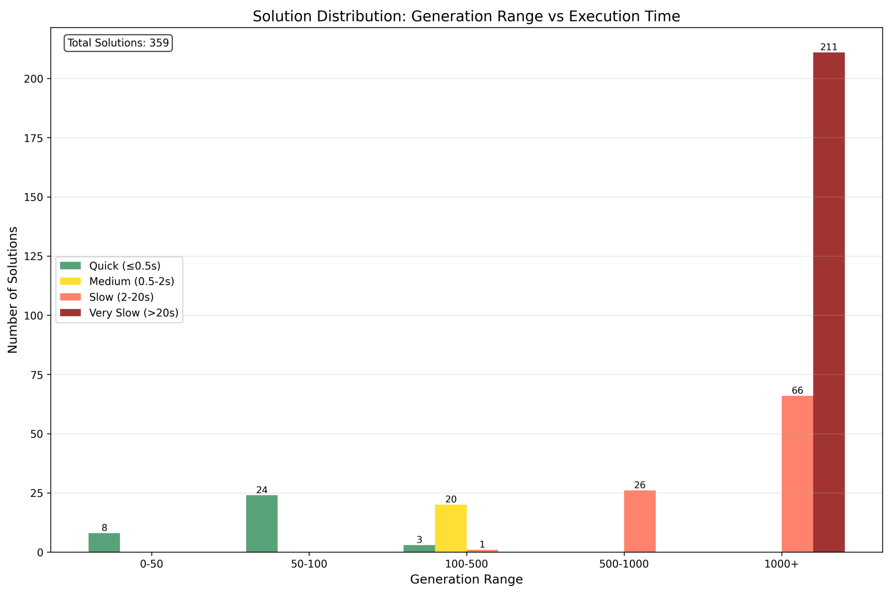


Figure 10: Generation and execution time distribution for medium difficulty puzzles.

Medium Difficulty Statistical Analysis Table 2 presents a comprehensive statistical analysis of the genetic algorithm performance on medium difficulty Sudoku puzzles over 300 runs.

Table 2: Statistical Analysis of GA Performance on Medium Difficulty Puzzles

Metric	**Value**
Total Runs	359
Successful Runs	328 (91.4%)
Failed Runs	31 (8.6%)
Execution Time	
Average	102.05s
Range	0.13s – 695.95s
Successful Runs (Avg)	67.17s
Failed Runs (Avg)	471.15s
Generations	
Average	21681.9
Range	30 – 100,000
Generations per Second	213.95
Solution Quality	
Average Violations	0.2
Failed Runs Violations	2.0 (constant)

5.2.3 Hard difficulty

The charts below shows the performance of the GA with hard difficulty. The number of solutions found decreased drastically. Only 45% of boards could be solved. The proportion of getting the solution in very slow is 97% which is much higher compared to easy and medium difficulty. Meanwhile, the execution time and generation still remain a linear relationship.

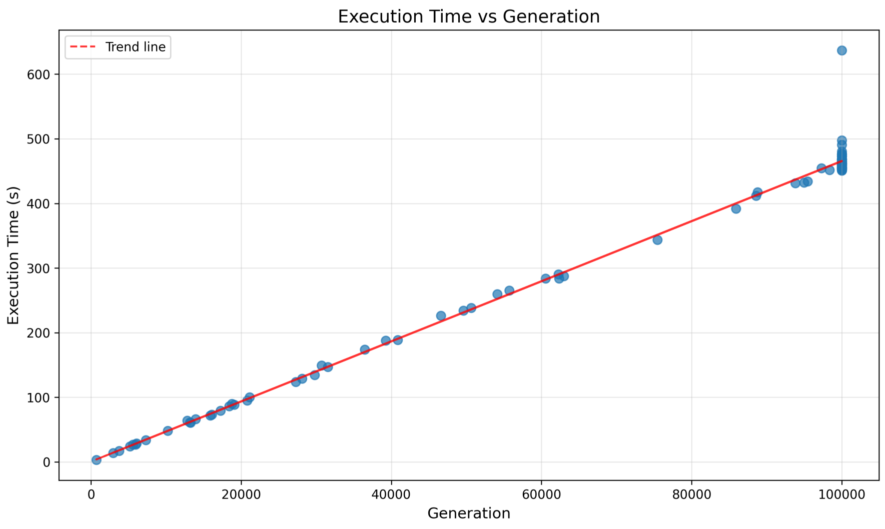


Figure 11: Generation vs execution time for hard difficulty puzzles.

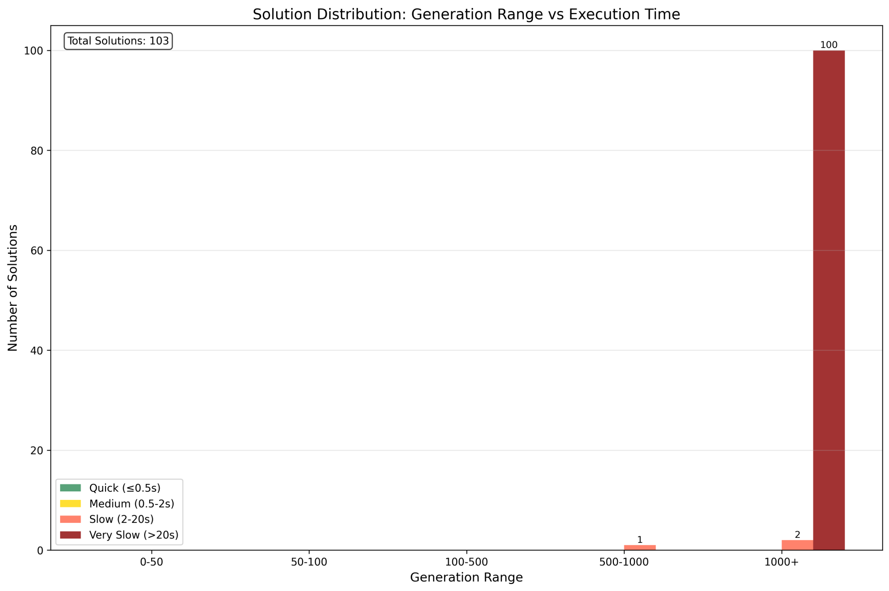


Figure 12: Generation and execution time distribution for hard difficulty puzzles.

Hard Difficulty Statistical Analysis Table 3 presents a comprehensive statistical analysis of the genetic algorithm performance on hard difficulty Sudoku puzzles over 100 runs.

Table 3: Statistical Analysis of GA Performance on Hard Difficulty Puzzles

Metric	**Value**
Total Runs	100
Successful Runs	45 (45.0%)
Failed Runs	55 (55.0%)
Execution Time	
Average	338.62s
Range	3.22s – 636.92s
Successful Runs (Avg)	183.11s
Failed Runs (Avg)	465.85s
Generations	
Average	72,763.8
Range	705 – 100,000
Generations per Second	215.18
Solution Quality	
Average Violations	1.1
Failed Runs Violations	2.0 (constant)

5.2.4 Performance Analysis

The results reveal significant challenges as difficulty increases the performance of the GA:

- The success rate decreases to 45% on the hard level, compared with 100% on the easy level and 91.4% on the medium level, which reflects the higher complexity of hard puzzles.
- Failed runs take 282.75s longer on average than successful runs (465.85s vs 183.11s) on hard level. However, the average successful run time compared with failed runs (67.17s vs 471.15s).
- All failed runs (100%) were stuck at local minima with exactly 2 violations
- Average of 85,181 generations without improvement in failed runs which is better than the medium level (91661.1).

5.3 Performance of DFS

We test the performace of the rule based algorithm with easy, medium and hard difficulty.

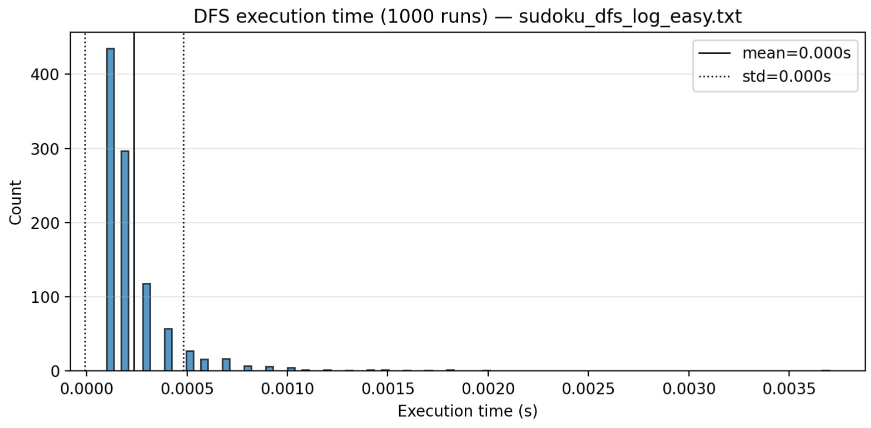


Figure 13: sudoku_dfs_log_easy_histogram.

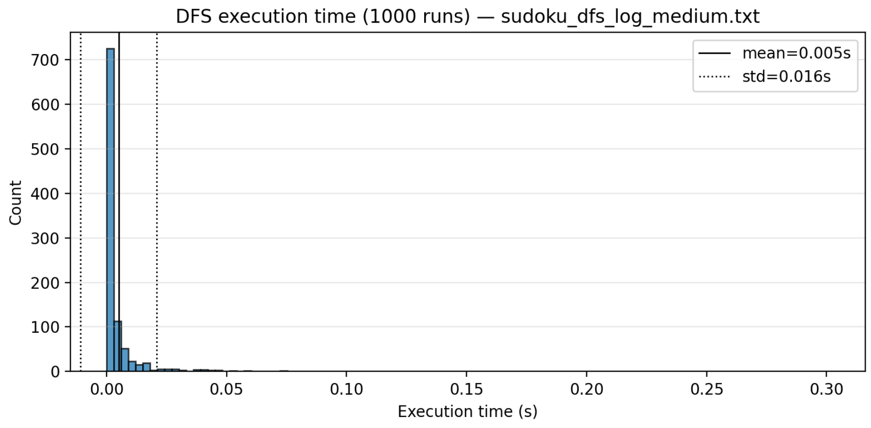


Figure 14: sudoku_dfs_log_medium_histogram.

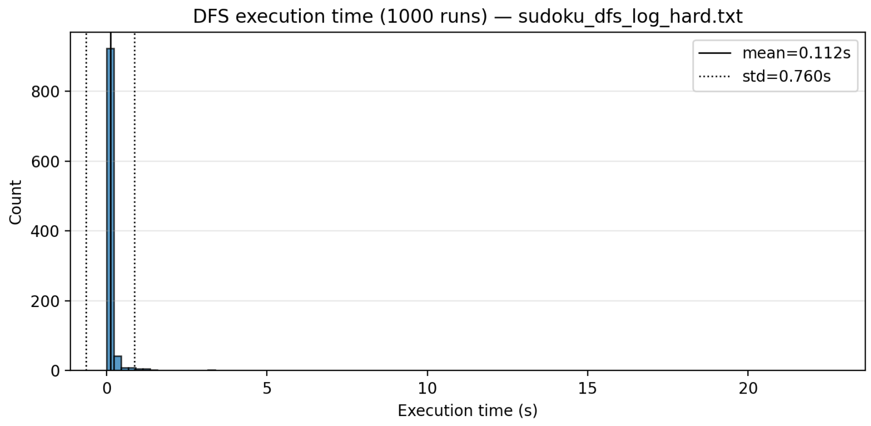


Figure 15: sudoku_dfs_log_hard_histogram.

The charts illustrate that no matter how hard the puzzle is, the rule based algorithm can solve it in a short time. Moreover, the execution time is very stable, only taking 0.001s-0.002s to solve each puzzle. It is immediately apparent that the DFS algorithm drastically outperforms the implemented GA algorithm.

6 Conclusions

This paper studied the efficiency of solving Sudoku puzzles with GA approaches of different complexities. The results show that the performance on 9x9 boards varies drastically. While most easy boards can be solved fast, the algorithm struggles on medium and hard boards. Only 10% of medium boards can be solved in under 2 second. For puzzles of hard difficulty, the algorithm did not achieve a single solution in under 2 seconds. Additionally, the success rate of the GA in finding a solution to the puzzle decreased with increasing complexity of the problem. While 100% of the easy boards were solved, only 91.4% and 45% of boards were solved of medium and hard complexity respectively.

The naive search algorithm DFS found solutions for every puzzle of every difficulty. Furthermore, the mean solving time of the boards of hard difficulty is 0.112 seconds, while easier puzzles were solved even faster.

These results show that while GAs can be used to solve Sudoku puzzles, they do not outperform naive search approaches. This is the case in the Sudoku context because there is only one optimal solution (solving the board without any duplicates). GAs are not optimal. They perform well in finding solutions that are "good enough". When solving Sudokus however, only the optimal solution is relevant. The algorithm finds local maxima quickly, but then gets stuck in those in many cases. The algorithm reacts by soft resetting the population. This is done by generating a new population and combining it with some of the worst performing individuals of the stuck generation. This helps the algorithm to get out of a local maximum, but it doesn't guarantee to find the optimal solution next. This is the reason why 55% of hard difficulty boards could not be solved in 100.000 generations.

This led to the decision of not evaluating the performance of the bigger 16×16 and 25×25 boards in this paper. It is expected that the performance of bigger grids is even worse, especially since the increase in complexity is far greater when increasing the size of the boards compared to reducing given numbers in a 9×9 puzzle.

Future research may consist of combining GAs with different optimization approaches. If a significant increase in performance on 9×9 boards can be achieved, these hybrid GAs may be suitable for solving bigger 16×16 and 25×25 Sudoku puzzles.

As shown in [**?**], evolutionary algorithms can solve Sudoku puzzles efficiently.

References

[1] "Sudoku - wikipedia." Accessed: 2025-10-01.

[2] "Genetic algorithm - wikipedia." Accessed: 2025-10-01.

[3] T. Mantere and J. Koljonen, "Solving, rating and generating sudoku puzzles with ga," *2007 IEEE Congress on Evolutionary Computation, CEC 2007*, pp. 1382–1389, 2007.

[4] "Mathematics - wikipedia."

[5] "Evolutionary algorithms and sudoku."

[6] P. R. Amil and T. Mantere, "Solving sudoku's by evolutionary algorithms with pre-processing," *Advances in Intelligent Systems and Computing*, vol. 837, pp. 3–15, 2019.

[7] Y. Sato, N. Hasegawa, and M. Sato, "Acceleration of genetic algorithms for sudoku solution on many-core processors," in *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '11, (New York, NY, USA), pp. 407–414, Association for Computing Machinery, 2011.

[8] C. Wang, B. Sun, K.-J. Du, J.-Y. Li, Z.-H. Zhan, S.-W. Jeon, H. Wang, and J. Zhang, "A novel evolutionary algorithm with column and sub-block local search for sudoku puzzles," *IEEE Transactions on Games*, vol. 16, no. 1, pp. 162–172, 2024.

[9] C. Rüther and J. Rieck, "A bayesian optimization approach for tuning a genetic algorithm solving practical-oriented pickup and delivery problems,"

[10] "Free 16x16 printable sudoku puzzles (pdf)." Accessed: 2025-10-06.

[11] "The daily printable 25 x 25 sudoku puzzle." Accessed: 2025-10-06.