Group 6 Assignment Report

An Evolutionary Algorithm for solving Sudokus

Ben Bekir Ertugrul Eduard Iorga Haoran Liu
Moritz Schlitter

October 6, 2025

1 Introduction

The recommended software for typesetting assignment reports is LaTeX. It will allow you to prepare high-quality documents, especially in the area of Computer Science. This document can serve as a template for reports. Each section begins with brief instructions in red text. All the instructions in red, as well as the dummy text, should be removed in the final version to submit. The LaTeX source of this file includes examples of using the most needed commands and environments. You can find plenty of other examples with explanations in many web forums and discussion groups on the Internet. The easiest way to edit your report is to use `https://www.overleaf.com/`. Overleaf does not require any setup on your computer, and it is free to create an account.

The book *Writing for Computer Science* [**?**] is a useful assistance on how to write properly and present your work when it comes to Computer Science topics. It is a strong recommendation to follow its guidelines and limit the usage of AI tools to generate text. Keep in mind that the examiner is an expert in Evolutionary Computation and therefore, any false information generated by an AI tool is easily notable. Such case may lead to failing the assignment.

The introduction should briefly introduce the assignment and its purpose.

—

- Context - Introduce Sudoku and Genetic algorithm very shortly

- This paper studies the efficiency of solving Sudoku puzzles with evolutionary algorithm approaches

- Different difficulties for 9x9, increased sizes

- Additionally, we want to test whether increasing the size of the Sudoku board leads to better performance of evolutionary algorithms compared to naive search algorithms like DFS.

- The performance is compared between evolutionary algorithms of different complexities as well as to Deapth-First Search - a naive optimization algorithm

- The objective is to draw a conclusion on which approach solves Sudoku more efficiently, both for regular 9x9 Sudoku boards as well as for bigger boards (16x16 and 25x25).

- The document is structured as follows: Section 1: Introduction and goal of the paper. Section 2: Describes the problem the paper solves in detail (Solving Sudoku) + motivation of the evolutionary approach used. Section 3: Describes the algorithm in detail. Section 4: Describes the experiments run with the algorithm. Section 5: Presents and analyses the results. Section 6: Conclusion.

Traditionally, a Sudoku is a logic based number placement puzzle. The objective of the puzzle is to fill a most commonly 9x9 sized grid with digits so that each column, row and 3x3 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, since they can go beyond the standard 9×9 grid, similar to the 25×25 grid we used to test our algorithm with. 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 as same row
- $y1 = y2$, which translates as 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. As it is mentioned in[5], they managed to implement an evolutionary algorithm that could solve very efficiently easy Sudoku puzzles. The downside of their algorithm, was that it wasn't as efficient when it came down to solving medium or hard puzzles. The way, they approached the algorithm [5] 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 [6], where pre-processing was involved. 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 way in which a method was selected to be used was based on the type of puzzle, since each method has different efficency depending on the case. After the pre-processing was done, different evolutionary algorithms were tested, such as a genetic algorithm that uses crossover and mutation operators, another one that firstly pre-processes the board and then applies the said genetic algorithm and finally even a genetic algorithm that uses an ant colony optimization.The fitness function used in this paper is made of three components: 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 to was that solving the sudoku puzzle was faster when pre-processing was involved and that a hybrid algorithm between genetic algorithm and ant colony optimization was the most efficient with every type of sudoku puzzle.

Lastly, the paper that influenced our approach the most was [3], which uses the genetic algorithm with Darwinian evolution as its inspiration. 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 generation brought the best results when no solution was found. Similar to our project, the authours experimented with different fitness functions, but ended up using a function that counts missing or duplicate digits in each row and column, assigning penalties. This being said, the optimal value of the fitness function would be 0. On top of that, the authors introduced an aging penalty, adding +1 to the fitness of the best solution if it remains unchanged. The conclusion the paper arrived to was that the genetic algorithm could solve sudoku puzzles quite efficiently, and even if there were better performing algorithms, they sometimes would fail at puzzles that the genetic algorithm could solve.

The motivation for our project is quite straightforward. We want to explore if a streamlined evolutionary algorithm with carefully chosen operators can achieve better perfromances than already existing and efficient search algorithms, such as the depth-first search algorithm. As previously mentioned our main inspiration for the evolutionary algorithm was the paper [3], thus we implemented an evolutionary algorithm that uses mutation and crossover operators, while also taking an elitist approach regarding the selection of the parents. On top of that, we also wanted to investigate the scalability of our evolutionary algorithm, thus we also tested it on a 16×16 puzzle. Evolutionary algorithms are well suited for this type of problem, because they can handle large solution spaces effectively, balance exploration and explotation, and adapt by integrating problem-specific heuristics. All being considered, we believe that the Sudoku problem is a great choice for analyzing all those 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 is 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 ← length(matrix)
for i ← 1 to n do
    for j ← 1 to n do
        if matrix[i, j] = 0 then
            for num ← 1 to n do
                if is_valid_move(matrix, i, j, num) then
                    matrix[i, j] ← num
                    if DFS(matrix) then
                        return true
                    end if
                    matrix[i, j] ← 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 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 ← PopulateRandomly(initial_sudoku, population_size)
  generation ← 0
  while generation < max_generations do
      population ← SortByFitness(population)                ▷ descending order
      if Fitness(first(population)) = max_fitness then
          return first(population)
      end if
      next_generation ← ∅
      for i ← 1 to population_size × 0.8 do
          parent_1, parent_2 ← Sample(top 20% of population)
          child ← Crossover(parent_1, parent_2)
          child ← Mutate(child, mutation_rate, initial_sudoku)
          next_generation ← next_generation ∪ {child}
      end for
      for i ← 1 to population_size × 0.2 do
          parent_1, parent_2 ← Sample(population)
          child ← Crossover(parent_1, parent_2)
          child ← Mutate(child, mutation_rate, initial_sudoku)
          next_generation ← next_generation ∪ {child}
      end for
      population ← next_generation
      generation ← generation + 1
  end while
  population ← 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 ($i, j \in \{0, \ldots, 8\}$) 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.

Crossover A new individual is creating 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 of a new individual, it 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 increase 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; 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.

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). Duplicate numbers, as a result of the crossover step, can now only emerge in columns or subgrids, which is an improvement compared to Implementation 1.

Mutation For the mutation, we iterate all rows of an individual and swap two random tiles that both haven't been given in the initial sudoku. We do this with the probability of *mutation_rate* for each row, once again ensuring the uniqueness of numbers in each row.

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

The mutation of the individual that resulted from the previous crossover is shown in figure 5. For this example, the mutation rate was set to 20%, leading to mutations in row 1 and row 5.

Problem The second genetic algorithm also often converges to local optima and loses diversity, which limits further improvement. We added adaptive mutation rates and replaced most of the population when fitness stagnated across multiple generations. These measures helped but did not completely eliminate local optima problems. Furthermore, scalability is 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 section describes the setup of experiments [?]:

- Provide the details of the hardware and software that you used.
- Describe the steps you carried out during your experiments.
- Detail the data you used for the evaluation of your algorithm.

This section describes the setup of the experiments run to study the efficiency of solving Sudoku puzzles with GA approaches.

The test 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[7].

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 are 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 1000 times for a maximum of 10.000 generations for Sudoku puzzles of easy, medium and hard difficulty of 9x9 boards. Because of the drastic jump in complexity with 16x16 and 25x25 boards, the GA is only run ? times for these puzzles. For the evaluation of the performance the following properties of the experiment are tracked:

1. Number of successfully solved boards
2. Execution time
3. Number of average generations
4. Average execution time per Sudoku puzzle
5. Average generations per second

For failed runs the following properties are tracked:

1. Number of generations stuck at local minimum
2. Best fitness achieved
3. Average violations
4. Generations without improvement
5. Average generations without improvement

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 properties of the experiment are tracked:

1. Number of successfully solved boards
2. Execution time
3. Average execution time per Sudoku puzzle

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 9x9 grids, the easy difficulty gives between 40-50 numbers. Medium and hard puzzles have 30-39 and 25-29 givens respectively.

Because of the increased complexity, run time and reduced success rate of solving 16x16 and 25x25 puzzles, the algorithm will be run fewer times only on three 16x16 boards taken from [8] and one 25x25 board taken from [9].

5 Results and Analysis

This section should present the obtained results and provide an insightful analysis of them. You can present the results using graphs, tables, or any other visualization method suits your purpose. Do not forget to include proper captions [?] in any of these illustration methods you use. You do not need to provide any execution details as they are already presented in Sec. 4.

A good practise would be to compare your algorithm with a simpler approach, such as (a) a naive method, (b) a Hill Climbing approach, or (c) a simple evolutionary algorithm. In the third case, you can use the simpler version of the algorithm you developed, i.e., the original algorithm without your modifications. In that case, you should briefly describe the comparing method(s) in Sec. 4. Alternatively, you can use some reference results derived from the repositories you found some benchmark instances.

To display tables, the `booktabs` package might be useful. For example, Table 1 shows how you should increase the size of n, when running your code. You can advice [?] to see a few examples of proper tables.

Table 1: Example of comparison the developed algorithm's results with the best ones from a repository.

Instance	**Optimum (Repository xyz)**	**EA**	**time (s)**
st70	678.597	677.109	0.67
ei176	545.387	544.369	1.16
kroA100	21285.443	21285.443	1.69
rd100	7910.396	7910.396	2.14
Pr136	96772	96770.924	7.11
Pr144	58537	58535.221	7.97
a280	2856.769	2856.769	33.47

You can use different illustration methods to present different aspects of your analysis. Figure 6 gives an example using the `pgfplots` package.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

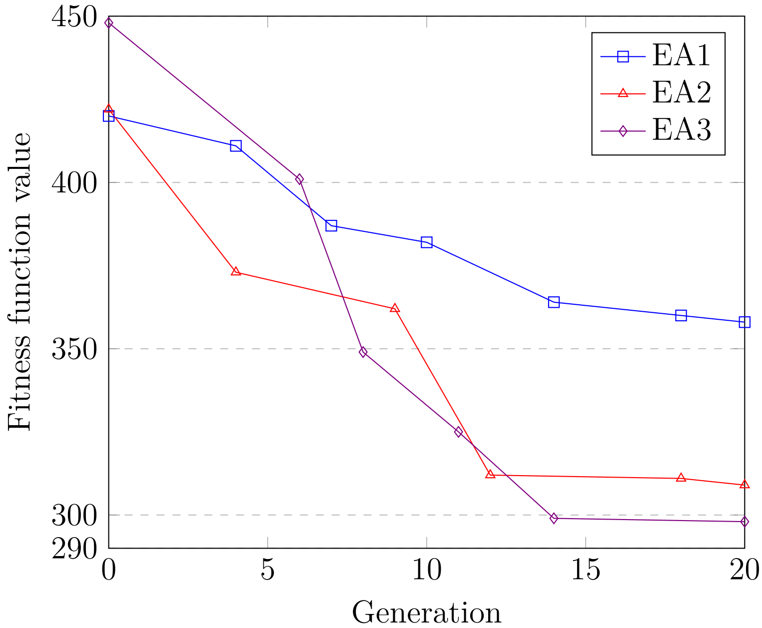

Figure 6: Example of convergence analysis.

6 Conclusions

In this section you should provide a concise summary of what has been done, the obtained results and some recommendations on how this study could be extended.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

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

References

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

[2] "Genetic algorithm - wikipedia."

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

[8] "Free 16x16 printable sudoku puzzles (pdf)."

[9] "The daily printable 25 x 25 sudoku puzzle."