Sudoku Genetic Algorithm Solver

Computational Intelligence for Optimization

GitHub repository: https://github.com/aspoderosas/Project-CIFO.git

Group Name: As Poderosas

May 2024

Group Members:

Carolina Jacobson, 20230594

Catarina Reis, 20230981

João Gonçalves, 20230560

Prof. Leonardo Vanneschi | Prof. Berfin Sakallioglu

NOVA Information Management School

Instituto Superior de Estatística e Gestão de Informação

1. Introduction

In this project, we explore the use of Genetic Algorithms (GAs) to develop an efficient Sudoku solver. For this study, we implemented three selection methods (roulette wheel, tournament, and rank selection), three mutation operators (one for each of the following: row, column and block swap mutation), and three crossover operators (row, single column and single point crossover). Then, we conducted a thorough analysis of the algorithm's performance using different parameters and combinations of these methods.

Sudoku is a logic-based number-placement puzzle, consisting of a 9x9 grid divided into nine 3x3 sub-grids. The challenge lies in filling the grid so that each column, each row, and each sub-grid contains all digits from 1 to 9 without repetition.

Regarding the division of labor in our group, all members contributed equally to research, coding, testing, and writing the report.

2. Representation

The representation of the problem is crucial for the algorithm's effectiveness. Our Sudoku puzzle is represented as a 9x9 grid. Each cell in the grid can either be pre-filled with a number from 1 to 9 or left empty. This initial grid serves as the template for our genetic algorithm, with the goal of filling in the empty cells while ensuring that each row, each column, and each 3x3 sub-grid contains all digits from 1 to 9 without any repetition.

To initialize the population of potential solutions, we ensured that each individual in the population is generated by taking the initial Sudoku grid and filling the empty cells with random numbers from 1 to 9, keeping the original filled cells fixed. If no specific Sudoku puzzle is provided, we create a new randomized valid Sudoku board. The generator starts by generating a complete solution and then removes numbers randomly, ensuring that the remaining puzzle has a unique solution. This generated grid is then used to create the initial population of individuals.

3. Fitness function

The fitness function is a critical component of the genetic algorithm, as it evaluates how close a given solution is to being a valid Sudoku solution. In our Sudoku solver, the fitness function measures the validity of the Sudoku grid by identifying conflicts in rows, columns, and 3x3 sub-grids.

To determine the fitness of an individual solution, the function checks each row, column, and 3x3 sub-grid in the Sudoku grid. For each row, a set is used to track numbers that have already appeared. If a number appears more than once, it is counted as a conflict, and the fitness penalty increases by one for each duplicate.

This process is repeated for columns and 3x3 sub-grids, with conflicts counted and penalties incremented similarly.

Mathematically, the fitness function can be defined as follows:

Let r_i be in the i-th row, c_i in the i-th column, and b_i in the i-th 3x3 sub-grid of the Sudoku grid. The total number of conflicts C is the sum of row conflicts, column conflicts, and sub-grid conflicts:

$$C = \sum_{i=1}^{9} conflits(r_i) + \sum_{i=1}^{9} conflits(c_i) + \sum_{i=1}^{9} conflits(b_i)$$

where $conflits(x)$ is the number of duplicate numbers in x.

The normalized fitness score is then calculated using the formula:

$$normalized_{fitness} = 1 - \left(\frac{C}{\max\limits_{conflits}} \right)$$

where $\max\limits_{conflicts} = 3 \times 9 \times 9 = 243$

(since each of the 81 cells can potentially conflict in its row, column, and sub-grid).

This fitness function guides the algorithm towards valid Sudoku solutions by favouring individuals with fewer conflicts. We are dealing with a maximization problem, as better solutions correspond to higher fitness values. When an individual achieves a fitness score of 1, we can confidently say that we have found a solution to the puzzle.

4. Selection methods

In our study, we implemented three selection methods: Fitness Proportional Selection (roulette wheel), Tournament Selection, and Rank-Based Selection. These methods influence genetic diversity and convergence. Fitness proportional selection assigns probabilities based on fitness, while tournament picks the best individuals from a random subset of a certain size. Rank-Based selection ranks individuals and assigns selection probabilities accordingly. Each method provides a unique strategy for selecting individuals for reproduction.

5. Genetic operators

Regarding the crossover methods, we implemented a row, single column and single point crossover operators.

The Row Crossover function takes two parent Sudoku grids as input. A crossover point is randomly chosen within the range of rows, excluding the first and the last rows, corresponding to the index 0 and 8, respectively, which ensures that at least one row before and after the crossover point is preserved. From the crossover point to the end of the grid, rows from one parent are swapped with the corresponding rows from the other

parent. This operation ensures that the new children receive complete rows from their parents (Figure 1 - Row Crossover Illustration).

The Single Column Crossover performs a crossover on a single column chosen randomly. The function creates two children by copying the parent grid, then a random column index (from 0 to 8) is selected as the crossover point. For each row in the Sudoku grid (from 0 to 8), the values in the chosen column are swapped between the two children if the corresponding cell in the template Sudoku grid is empty (i.e., its value is 0). This ensures that only swappable columns based on the template are swapped. The resulting children grids inherit complete columns from their parents while respecting the constraints of the template Sudoku grid (Figure 2 - Single Column Crossover Illustration).

The Single Point Crossover selects a random column index between 1 and 8 as the crossover point. Two new puzzles are created as copies of the parent 1 and parent 2. For each row in the grids, it checks if the cell at the crossover point is mutable, i.e. if its value is 0. If it's mutable, it swaps the segments of the rows after the crossover point between the two children's grids. Finally, the function returns the two new offspring Sudoku grids.

Regarding mutation operations, we tested row, column, and block swap mutations, applied based on a probabilistic mutation rate. In row mutation, two values in a row are swapped; column and block mutations work similarly by swapping values within columns and 3x3 sub-grids, respectively. It's important to enhance the importance of mutation rate: higher rates increase diversity but can disrupt good solutions, while lower rates may cause premature convergence. For easy puzzles, a low mutation rate is preferable; for difficult puzzles, a higher rate helps explore more solutions.

6. Algorithm

Finally, we implemented the genetic algorithm for solving Sudoku. This algorithm iteratively improves a population of potential solutions over a specified number of generations. The process starts by initializing a population based on the given Sudoku puzzle. Each generation follows these steps: First, the fitness of each individual is assessed to determine how close they are to a valid Sudoku solution. Then, two parents are selected using a specified selection method, and they undergo a crossover operation to produce two offspring. Each offspring is then mutated using a specified mutation method and type. The new population replaces the old one, and its fitness is reassessed. The best fitness value of each generation is recorded to track progress.

After the specified number of generations, the best solution from the final population is identified and returned, along with the fitness values recorded over time. This approach ensures that the population evolves towards better solutions, leveraging the combined effects of selection, crossover, and mutation. We will experiment with different parameters and method combinations to optimize the algorithm's performance and effectiveness.

7. Experimental results

Regarding the experimental results, we decided to test 27 different combinations of selection, crossover and mutation methods. Since these combinations yielded good fitness results, we concluded that it was not crucial to perform an exhaustive grid search.

For our testing we will have in consideration three puzzles with different levels of difficulty (Figure 5 - Sudoku levels): easy, medium and difficult. The fitness of this puzzles are approximately 0.94, 0.48 and 0.37037, respectively. Our goal is to reach a fitness of 1, representing a solved puzzle. We applied our algorithm using different combinations of selection methods and genetic operations, as well as different hyperparameters: population size (500, 1000), tournament size (100, 250), and mutation rate (0.1, 1). Our visualizations show 150 generations because we observed that the best models converged within this range.

1: Population Size = 500, Tournament Size = 100, Mutation Rate = 1

We started by considering a population size of 500, a tournament size of 100, and a mutation rate of 1. For the easy puzzle, the best results were given by the tournament selection, showing the highest fitness scores and an early convergence, while the other two selection methods showed a decreased of fitness across generations. According to the high mutation rate, this result makes sense because the tournament method ensures that only the fittest individuals contribute to the next generation, lowering the negative impact of a large number of mutations in easy, almost solved, puzzles.

For the easy (Figure 6), medium (Figure 7) and difficult puzzles (Figure 8), the best models were the ones using tournament selection and row swap mutation. For the easy and medium puzzles, the single column crossover worked better, but for the difficult puzzle the single point crossover was the one with best results.

2: Population Size = 1000, Tournament Size = 100, Mutation Rate = 1

Testing with a larger population size of 1000 (Figure 9, Figure 10, Figure 11), the results showed that performance was barely changed compared to the previous value of 500 for the population size. The best model remained the tournament selection with single point crossover and block swap mutation.

3: Population Size = 500, Tournament Size = 100, Mutation Rate = 1

Given that our results indicate tournament selection as the most effective so far, we will now focus exclusively on this selection technique. Since the increase of population size didn't show a significant impact, from now on we considered a population size of 500. We will test the tournament size of 250, compared to the previous size of 100.

For the easy puzzle (Figure 12), the majority of configurations achieved a fitness score above 0.95 after approximately 20 generations. The model 9, a combination of single point crossover, and block swap mutation, achieved the highest fitness score of slightly above 0.98. It demonstrated quick convergence, reaching high fitness scores, and maintained a stable performance with minimal fluctuations after reaching its peak fitness. The model 7, incorporating single point crossover and row swap mutation, reached a fitness score of

approximately 0.95. While it displayed slower convergence compared to models 8 and 9, it maintained a consistent performance with minimal fluctuations after achieving its maximum fitness score after 80 generations.

For the medium difficulty puzzle (Figure 13), model 5, with single column crossover and column swap mutation, achieved the highest fitness score of approximately 0.89, demonstrating quick convergence, and maintained stable performance with minimal fluctuations.

4: Population Size = 500, Tournament Size = 100, Mutation Rate = 0.1

After testing the models with the maximum mutation rate, we will decrease it to 0.1. This reduction aims to achieve more refined and stable solutions by reducing disruptive mutations, allowing smoother and more effective convergence. For the easy puzzles (Figure 15), we clearly noticed a decrease on speed convergence, around less than 6 generations, and stability. The model that stood out was the 19, with a combination of single point crossover and block swap mutation.

8. Best model approach

According to our experiments, the optimal configuration combines tournament selection, single point crossover, and row swap mutation with a population size of 500, tournament size of 100, and mutation rate of 1. This setup consistently achieved rapid convergence to high fitness scores for easy, medium, and difficult puzzles, maintaining stability after reaching peak fitness.

9. Conclusion

After analyzing the three puzzles of varying difficulties, we found that our algorithm consistently produced better solutions and fitness scores compared to the initial attempts. This demonstrates the algorithm's ability to effectively solve the puzzles. We can confidently state that our project achieved good results, affirming that the use of Genetic Algorithms for solving Sudoku puzzles is a viable and effective approach. However, we believe that it would be important, with more time and resources, an even deeper exploration of different combinations of Selection, Crossover and Mutation methods, such as partially matched crossover and inversion mutation, and also a more robust statistical validation, increasing the number of times each combination is tested. Thus, it is possible that the speed of convergence of the genetic algorithm can be improved, by exploring more deeply and for a longer time possible combinations of parameters of the genetic algorithm, which may also lead to more difficult puzzles (or even Sudoku puzzles larger than normal size, with 81 cells) to be solved by the algorithm. Based on all the analysis and conclusions obtained so far, we can say that different operators affect the convergence of the Genetic Algorithm, with Crossover and Mutation types that stand out in terms of performance, as previously explained. The inclusion of Elitism would also positively impact the performance of the Genetic Algorithm.

Appendix

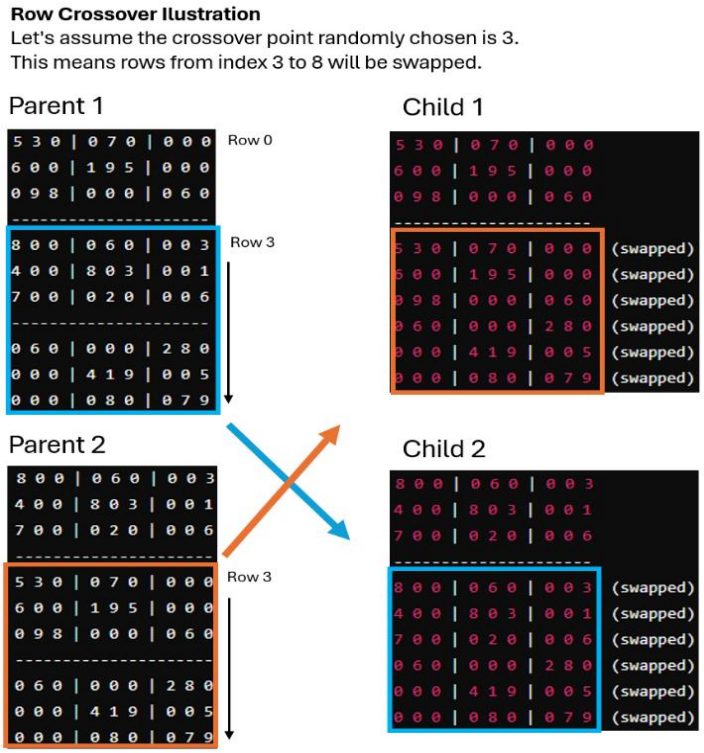


Figure 1 - Row Crossover Illustration

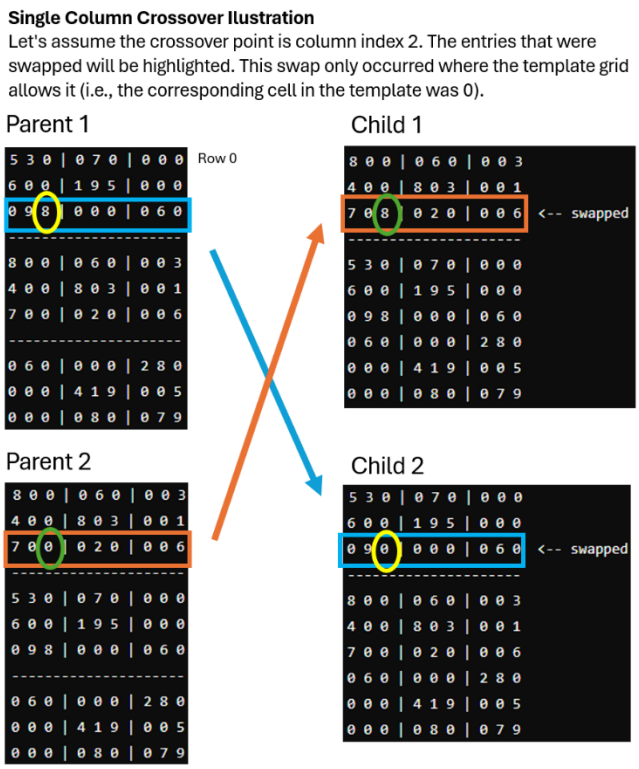


Figure 2 - Single Column Crossover Illustration

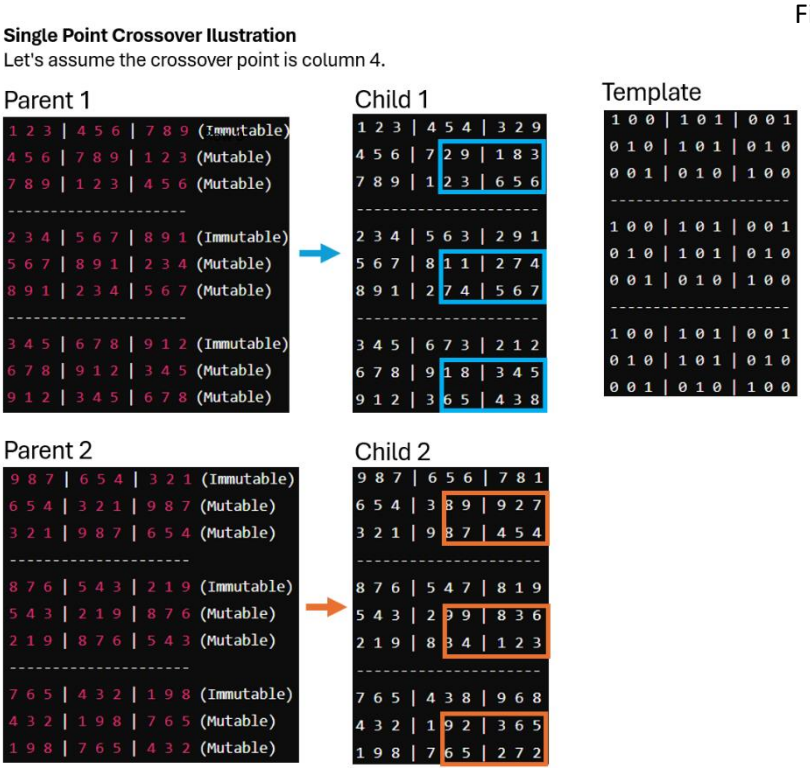


Figure 3 - Single Point Crossover Illustration

Single Point Crossover Ilustration

Let's assume the random row chosen is 0, and the empty cells in the template for that row are at indices 2, 3, 5, 6, 7, 8. If the mutation swaps the values at indices 2 and 5.

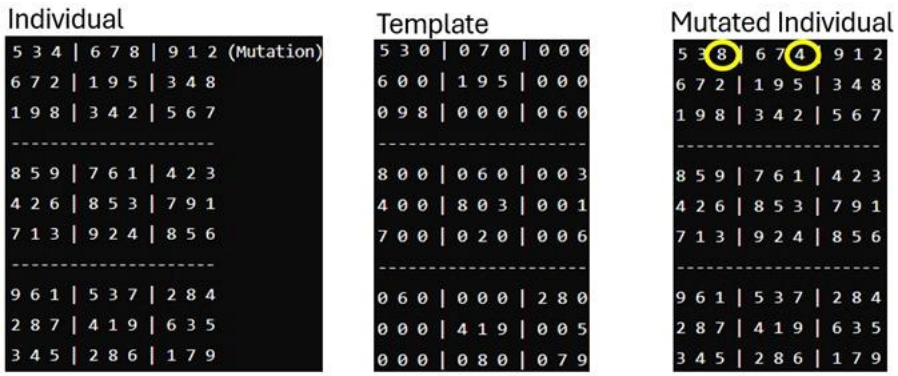


Figure 4 - Swap Mutation

```
easy_sudoku = [
    [2, 6, 4, 0, 7, 3, 0, 5, 1],
    [9, 7, 5, 2, 1, 6, 4, 8, 3],
    [3, 8, 1, 5, 4, 0, 6, 0, 2],
    [5, 2, 9, 6, 3, 7, 1, 4, 8],
    [4, 1, 3, 0, 8, 5, 7, 2, 6],
    [6, 9, 7, 1, 2, 4, 8, 3, 5],
    [7, 0, 8, 3, 0, 2, 1, 6, 4],
    [1, 3, 2, 4, 6, 8, 5, 9, 0],
    [8, 4, 6, 7, 5, 0, 2, 3, 9]
]
The fitness of the puzzle is: 0.9423868312757202
```

Easy difficulty puzzle

```
medium_sudoku = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]
The fitness of the puzzle is: 0.4814814814814815
```

Medium difficulty puzzle

```
difficult_sudoku = [
    [8, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 3, 6, 0, 0, 0, 0, 0],
    [0, 7, 0, 0, 9, 0, 2, 0, 0],
    [0, 5, 0, 0, 0, 7, 0, 0, 0],
    [0, 0, 0, 0, 4, 5, 7, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 3, 0],
    [0, 0, 1, 0, 0, 0, 0, 6, 8],
    [0, 0, 8, 5, 0, 0, 0, 1, 0],
    [0, 9, 0, 0, 0, 0, 4, 0, 0]
]
The fitness of the puzzle is: 0.37037037037037035
```

Hard difficulty puzzle

Figure 5 - Sudoku levels

Figure 6 - Easy difficulty puzzle [pop_size = 500, generation_size = 150, tournament_size = 100 , mutation_rate=1]

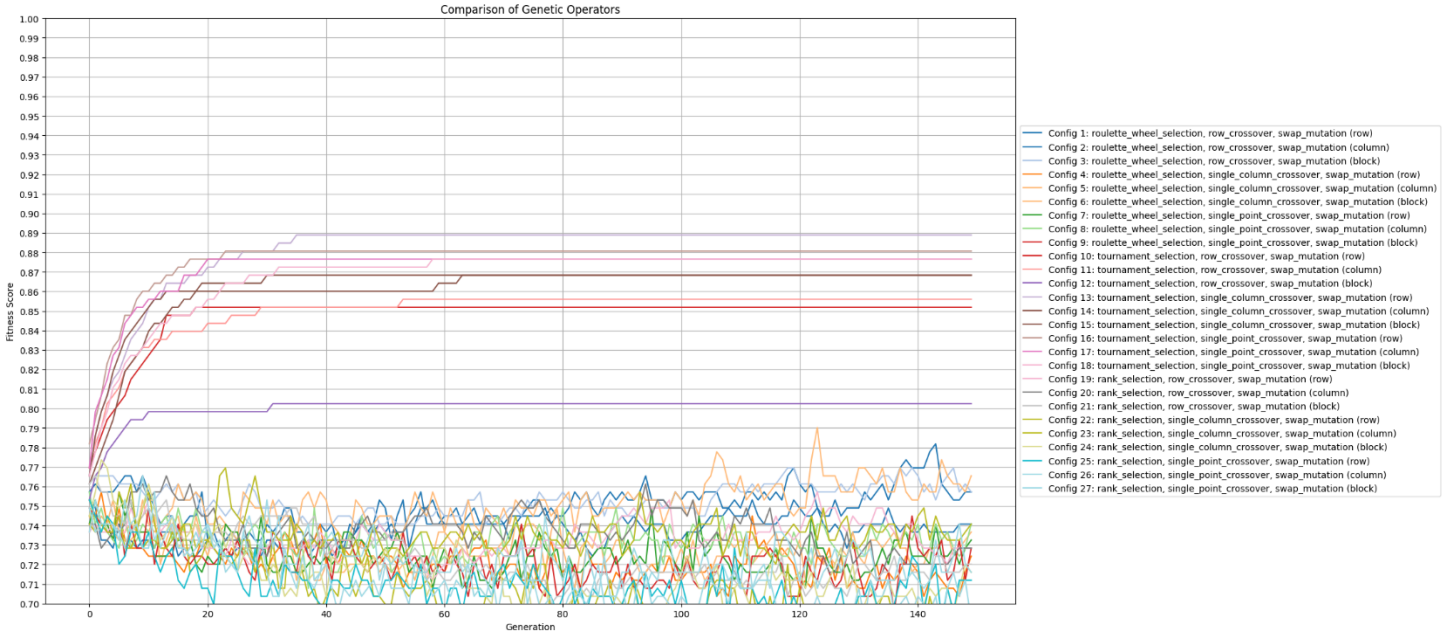


Figure 7 - Medium difficulty puzzle [pop_size = 500, generation_size = 150, tournament_size = 100 , mutation_rate=1]

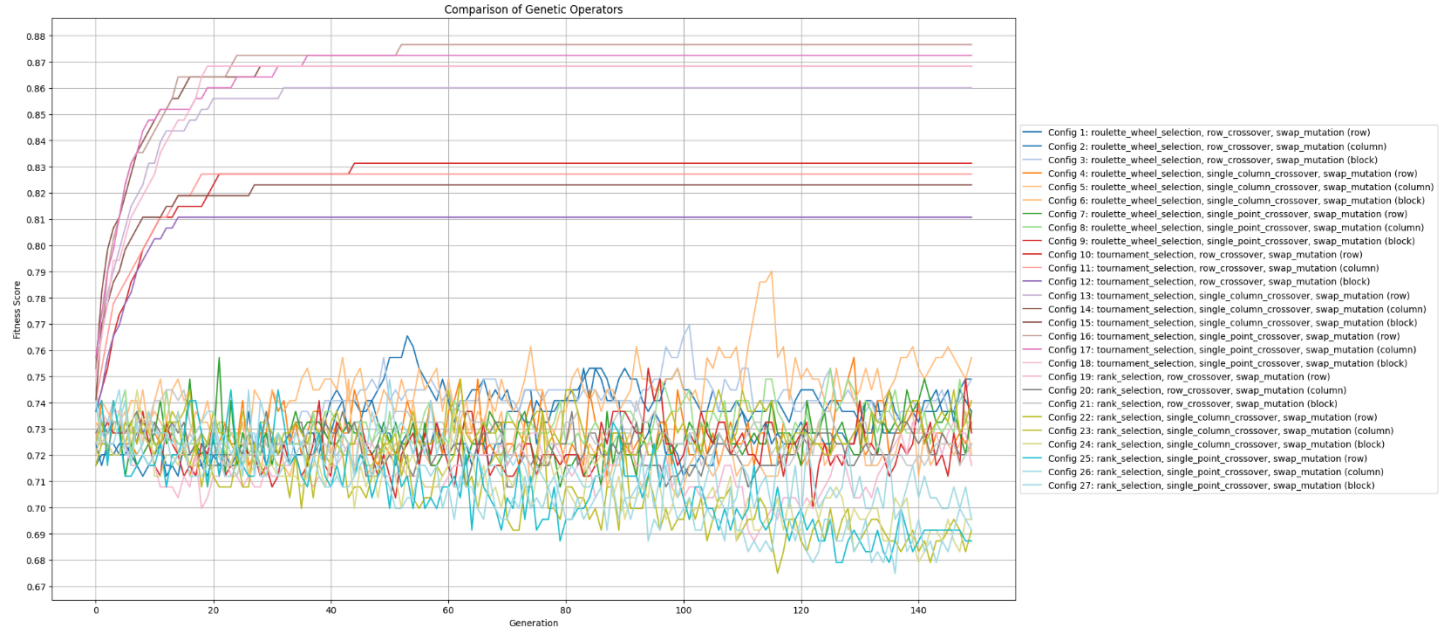
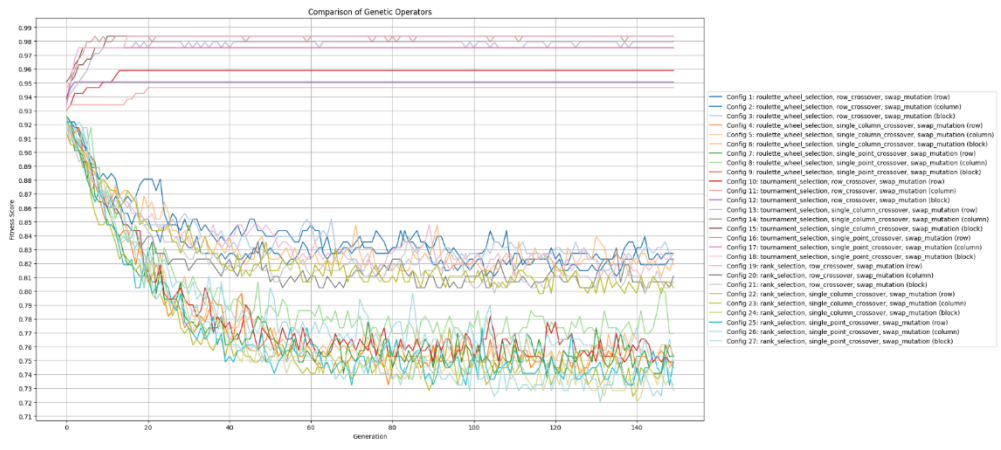


Figure 8 - Hard difficulty puzzle [pop_size = 500, generation_size = 150, tournament_size = 100 , mutation_rate=1]



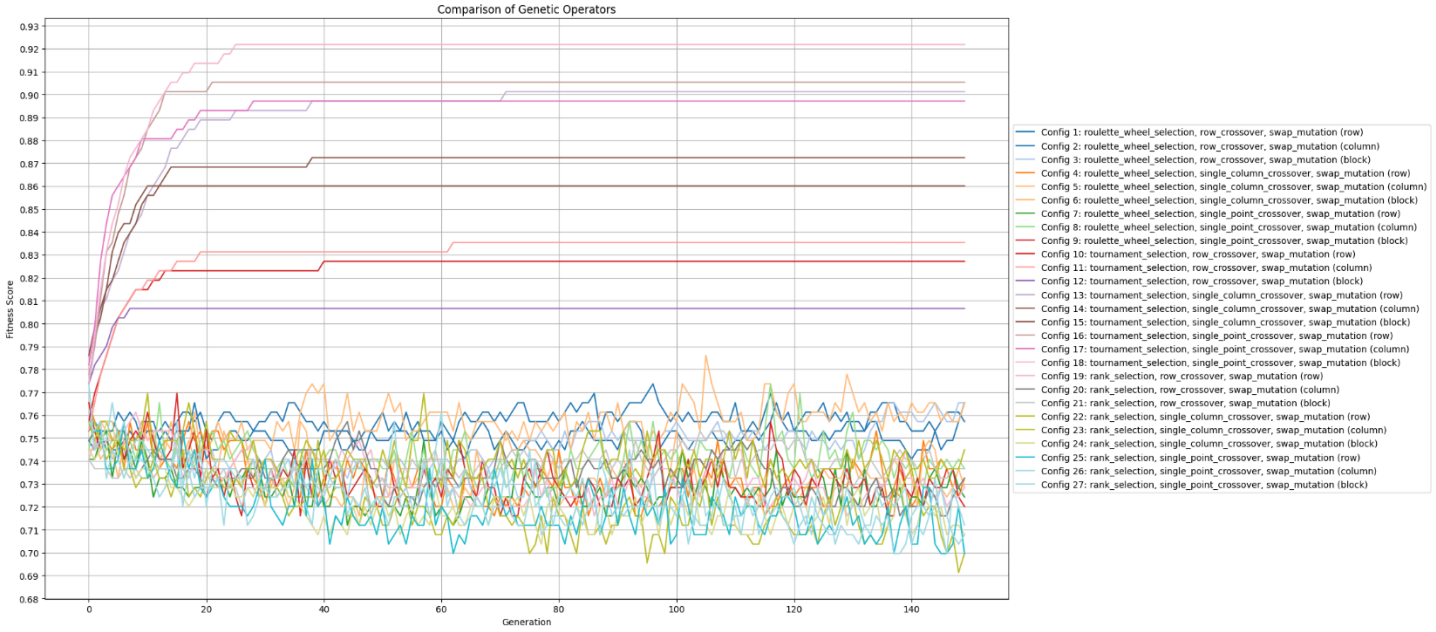


Figure 10 - Medium difficulty puzzle [pop_size = 1000, generation_size = 150, tournament_size = 100 , mutation_rate=1]

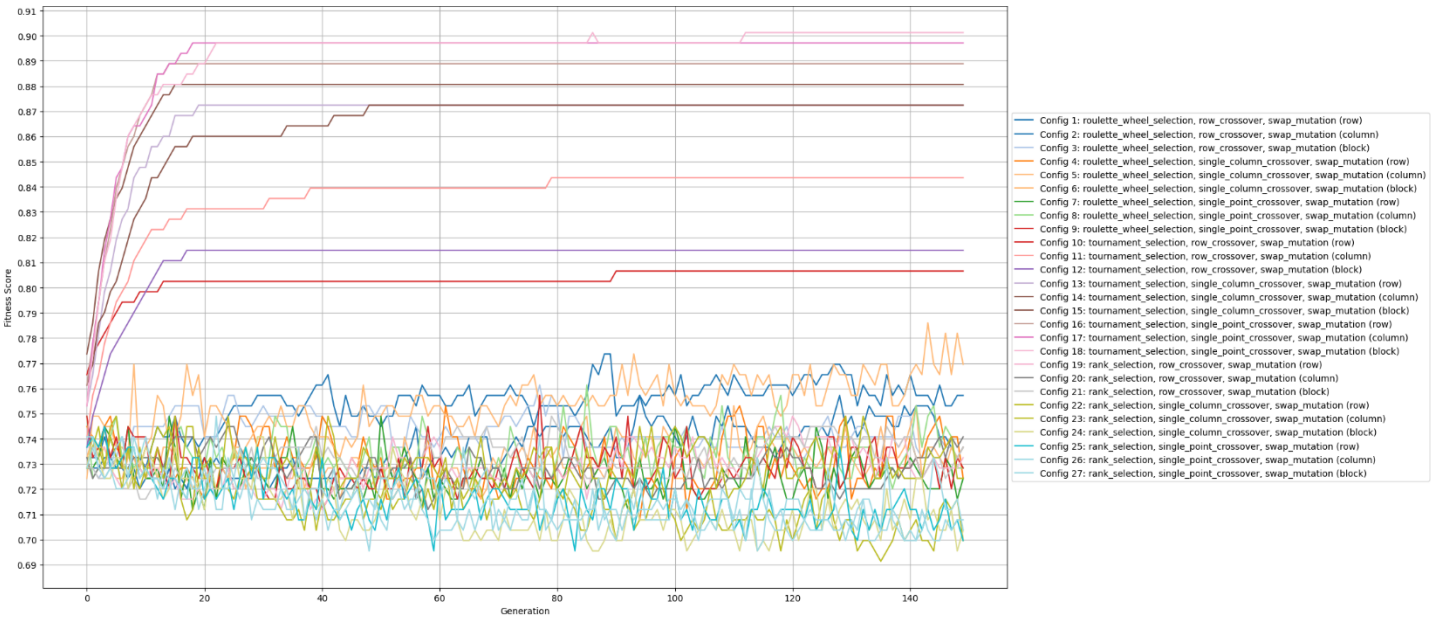


Figure 11 - Hard difficulty puzzle [pop_size = 1000, generation_size = 150, tournament_size = 100 , mutation_rate=1]

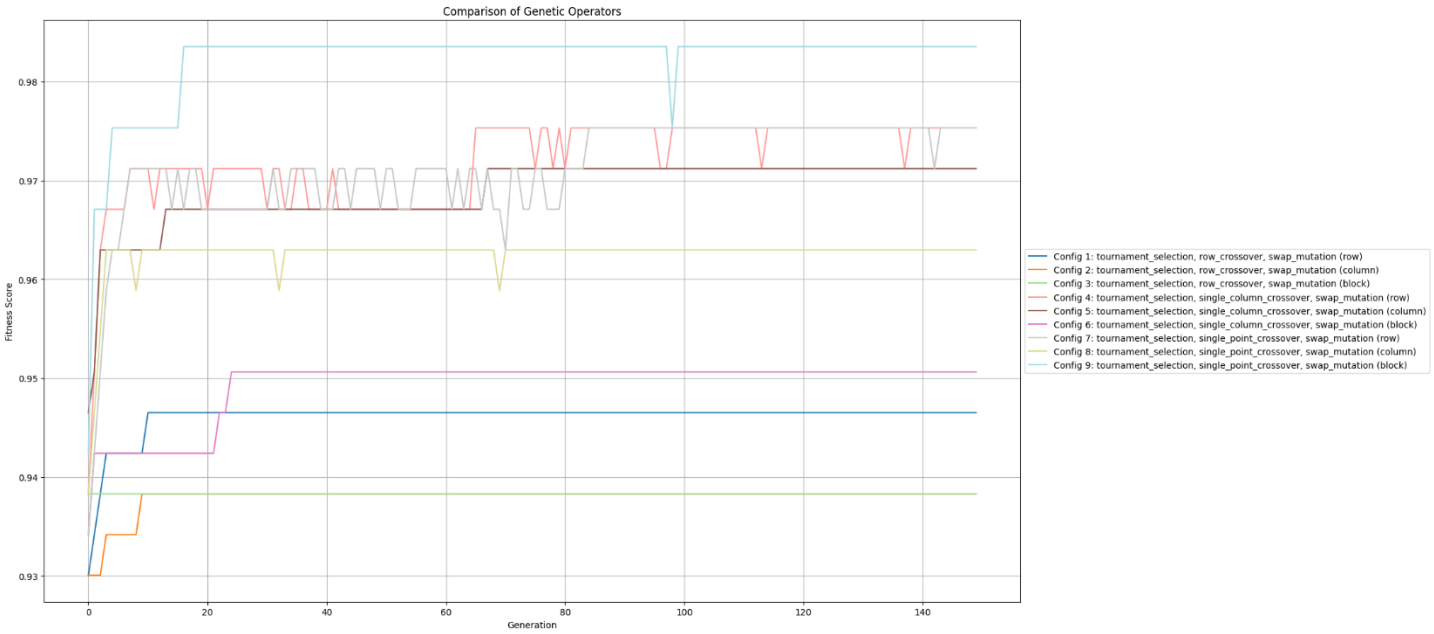


Figure 12 - Easy difficulty puzzle [pop_size = 500, generation_size = 150, tournament_size = 250 , mutation_rate=1]

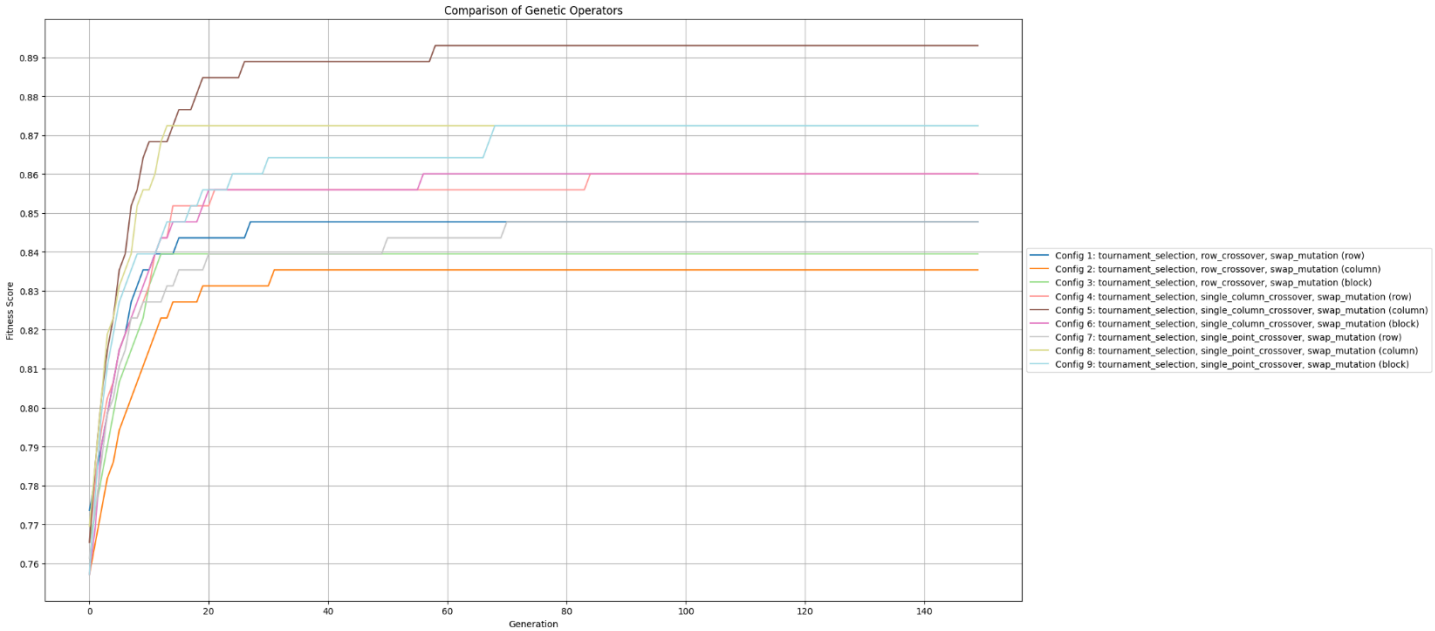


Figure 13 - Medium difficulty puzzle [pop_size = 500, generation_size = 150, tournament_size = 250 , mutation_rate=1]

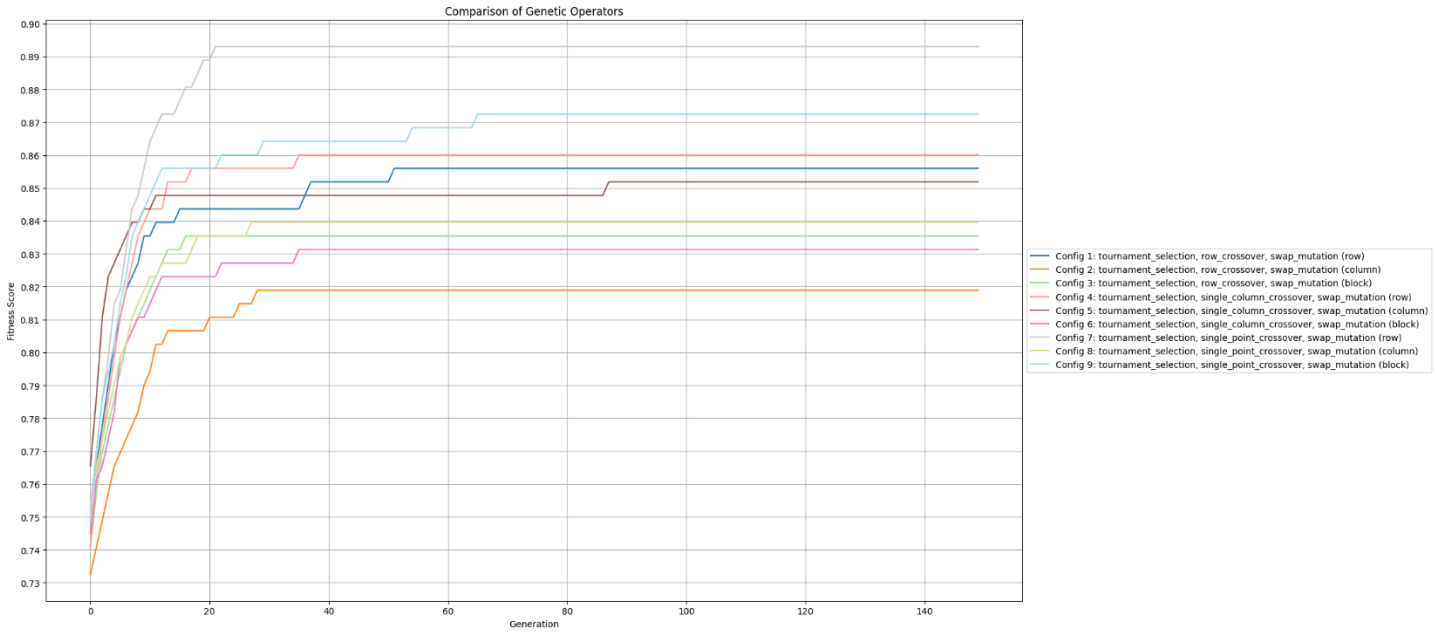


Figure 14 - Hard difficulty puzzle [pop_size = 500, generation_size = 150, tournament_size = 250 , mutation_rate=1]

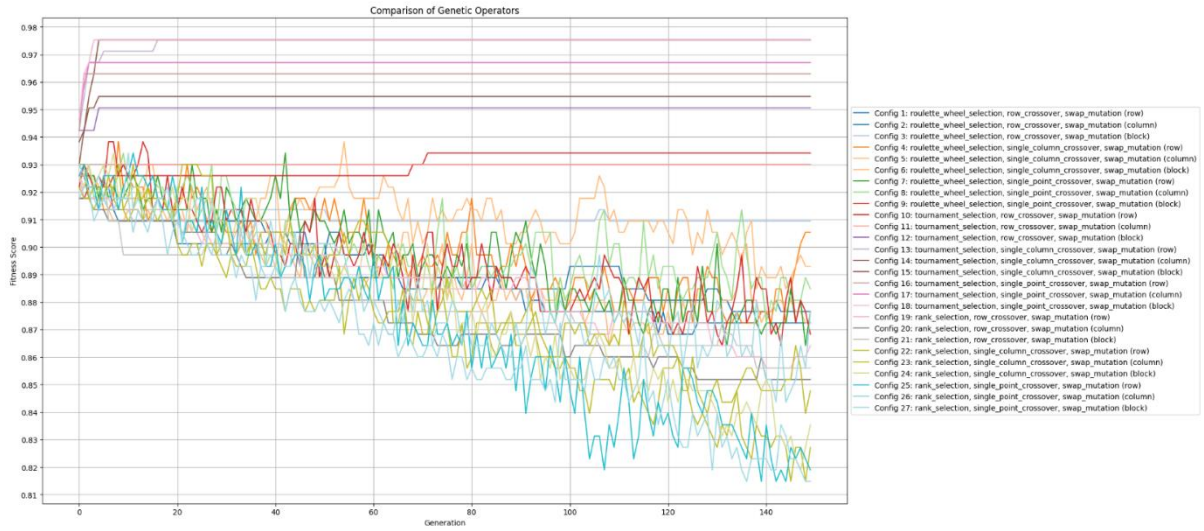


Figure 15 - Easy difficulty puzzle [pop_size = 500, generation_size = 150, tournament_size = 100 , mutation_rate=0.1]

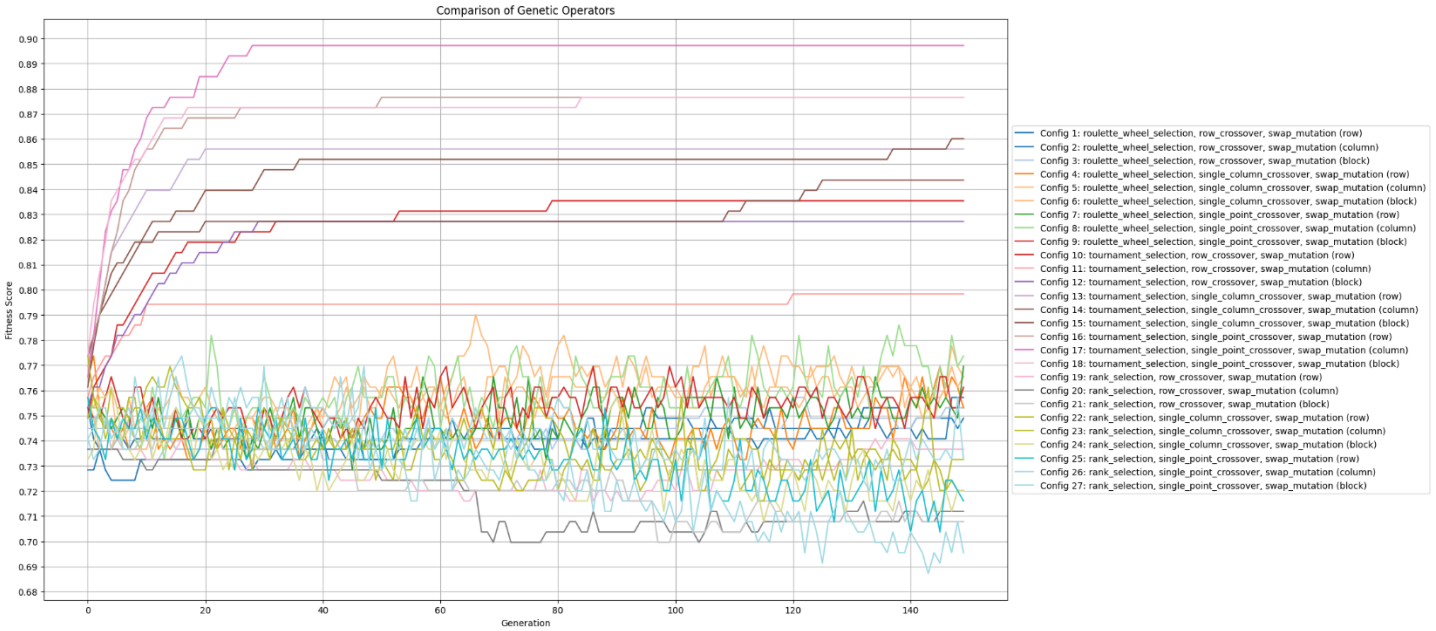


Figure 16 - Medium difficulty puzzle [pop_size = 500, generation_size = 150, tournament_size = 100 , mutation_rate=0.1]

Figure 17 - Hard difficulty puzzle [pop_size = 500, generation_size = 150, tournament_size = 100 , mutation_rate=0.1]