COMPUTATIONAL INTELLIGENCE FOR OPTIMIZATION

Mestrado em Estatística e Gestão de Informação

MASTER DEGREE PROGRAM IN DATA SCIENCE AND ADVANCED ANALYTICS

Sudoku Solver

Afonso Anjos, 20230527

Filipe Pereira, 20230445

João Machado, 20230426

Group: Group1

Link: https://github.com/filipedp25/cifo

June, 2024

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

Selection, Mutation and Crossover

In addition to the selection methods, mutation operators and crossover operators on the charles library, we also implemented new functions to try to further enhance the performance of our GA. As this problem is a minimization one, when explaining the functions implemented, we will be describing them when set to minimization.

All mutation and crossover operators including the ones from charles receive a “fixed_mask” argument so it can identify the immutable cells of the Sudoku puzzle.

New selection methods implemented

rank_based_selection - This function sorts the population's individuals based on their fitness values in descending order, then computes the total rank (sum of all ranks) of the sorted population, where each individual's rank corresponds to its position in the sorted list. A random value 'r' is generated within the range from 0 to the total rank. The function iterates through the sorted population, accumulating the rank sum as it progresses. When the cumulative rank sum exceeds 'r', the function returns the individual at the current iteration position.

stochastic_universal_sampling - This function inverts the fitness values to handle minimization as maximization. It calculates the total inverted fitness, determines the pointer distances similarly, and proceeds to select individuals based on the inverted fitness values. This ensures that individuals with lower fitness (which are represented by higher inverted fitness values) have a higher chance of being selected. At last, it returns a randomly chosen individual from the selected set based on the pointers.

New mutation operators implemented

scramble_mutation - This function identifies the indices of the positions in the representation that are not fixed (i.e., can be mutated). If there are fewer than two mutable positions, indicating that there isn't enough flexibility for mutation, the function returns the original individual without performing any mutation. If there are at least two mutable positions, the function randomly selects two distinct indices from the list of mutable indices. It then extracts the segment between these two indices from the copy of the individual's representation and shuffles it, scrambling the order of elements within that segment and returns the mutated individual.

subgrid_swap_mutation - This function randomly selects two distinct subgrid indices from the range 0 to 8, representing the 9 subgrids in sudoku puzzle. For each subgrid, it calculates the indices of the cells within that subgrid and shuffles them independently. Then, it iterates through the shuffled cell indices of the two subgrids simultaneously, swapping the values of corresponding cells if both cells are mutable (i.e., not fixed according to the fixed_mask). This swap operation exchanges values between two randomly selected subgrids and returns the mutated individual.

New crossover operators implemented

two_point_crossover - This function randomly selects two distinct crossover points within the representation size. These points divide the representations into three segments: before the first crossover point, between the two crossover points, and after the second crossover point. The function creates two offspring representations by combining segments from the two parent representations. After generating the initial offspring representations, the function iterates through each position in the representations. If a position is marked as fixed in the fixed_mask, indicating that it should not be modified, the corresponding value from the respective parent representation is preserved in both offspring representations. Finally, the function returns two new individual objects, each initialized with one of the modified offspring representations.

uniform_crossover - This function, for each position in the representations, checks if the position is mutable (not fixed according to the fixed_mask) and if a random value generated falls within the crossover rate probability, if both conditions are met, a crossover is performed by swapping the values at that position between the two offspring representations. After iterating through all positions, the function returns two new individuals.

block_crossover - This function generates a list of block indices representing the 3x3 subgrids in the sudoku. Next, the function randomly selects one block from the list of block indices. For each cell within the selected block, the function checks if the cell is mutable (not fixed according to the fixed_mask), if so, it swaps the values at the corresponding positions between the two offspring representations and finally, returns two new individuals.

Main algorithm and fitness function

Our primary goal was to determine the best combination of parameters and operators to achieve the lowest fitness value, since this meant that the closer the fitness was to zero, the closer our GA was to completely solve the Sudoku. To be able to do this, we implemented a series of functions, with the main function as the core and, to ensure reproducibility, we set a seed value for the random number generators. This allowed us to consistently replicate the results across different runs, making it easier to compare the performance and finding what we believe to be the optimal solution for various Sudoku levels.

To do this, we started by manually defining an initial Sudoku puzzle, which was relatively challenging, where zeros represented empty cells that the GA had to be able to fill them. The fixed values were extracted, and a "mask" was created to indicate which cells were pre-filled and would remain unchanged during the algorithm's execution. This ensured that the genetic operations only modified the empty cells and did not replace any of the fixed ones.

Afterwards, we had to create our fitness function that evaluated the quality of a Sudoku solution by counting rule violations in the grid. This function worked by checking for each row, column, and 3x3 subgrid, how many times each number appears. If a number appeared more than once, the excess occurrences were duplicates. These duplicates were then summed to calculate the fitness score, which represented the total number of duplicates across all rows, columns, and subgrids. Naturally, as previously stated, a lower fitness score represented a better GA solution for the Sudoku problem.

We then created a function that represented an initial solution that was generated by filling the empty cells of the puzzle with random numbers while keeping the fixed cells unchanged in order to give us a starting point for the GA to begin its search for an optimal solution.

After having created the necessary functions we implemented the main function, which organized the whole GA. Our goal here was to determine the optimal configuration, so we ran multiple experiments with different combinations of selection methods, crossover operators, and mutation operators that we had previously created both from the Charles library and the ones cited previously. Each combination was evaluated based on the fitness of the solutions and the results were analyzed to identify the optimal configuration.

The random seed was set, and an initial population of candidate solutions generated. Each candidate solution, or individual, was evaluated using the fitness function to have a baseline fitness value for the population.

Then, the population was set up with pre-chosen parameters that defined the size, optimization goal (minimization), solution size, valid set of values, and whether repetition of values was allowed. After that, the evolution process started running for a specified number of generations and during each generation, the population underwent through the different selection, crossover and mutation operators.

These genetic operations generated new offspring, which were then evaluated and integrated into the population. Elitism was also incorporated into the process to ensure that the best individuals from the latest generation were preserved and carried over to the next generation. Thus, maintaining the quality of solutions over successive generations. Throughout the evolution process, the best individual (solution) was tracked, and if a better solution was found, it was updated. By the end of the specified number of generations, the best individual with the lowest fitness value was identified as the optimal solution. This individual represented the Sudoku solution with the fewest errors, evaluated by the fitness function.

A grid search was then created in a second main function, to fine-tune the GA parameters, this time with predefined GA operators. This way we would also be able to find the best combination of parameters and the individual best combinations of operators that we had previously found. This was all tested for multiple different Sudoku levels as will be presented below.

The parameter grid used for the grid search included a population size of 500, 400 generations, crossover probabilities of 0.8, 0.9, and 0.95, mutation probabilities of 0.05, 0.1, and 0.15, and elitism enabled. This grid search will be used on all configurations tested throughout the three Sudoku difficulties.

Results

Even though the stochastic universal sampling selection method gave us one of the best fitness in the *run_experiments* function, due to time constraints, we didn't test it because it was too computationally expensive.

Easy Level

To determine the effectiveness of our GA on an easy-level Sudoku puzzle, we used the *tournament_sel, uniform_crossover,* and *swap_mutation*. We decided to proceed with only one grid search combination because we observed that the fitness score reached 0 very quickly, indicating that we had already found a perfect solution with no rule violations. Therefore, further configurations were unnecessary.

For our initial setup, we generated an easy-level Sudoku puzzle and extracted the fixed values and their positions to create a mask. This mask ensured that fixed cells remained unchanged during the GA operations.

Following this, we performed the grid search using the specified selection, crossover, and mutation operators, the optimal parameters identified a crossover probability of 0.8, a mutation probability of 0.05.

Early stopping occurred at generation 16 with a fitness value of 0, indicating a perfect solution. The optimal parameters consistently resulted in a perfect solution across multiple runs.

Medium Level

For the medium difficulty Sudoku puzzle, we continued using the *tournament_sel, uniform_crossover,* and *swap_mutation*. Additionally, we tested the following combinations: *fps, single_point_xo, swap_mutation*; *fps, uniform_crossover, subgrid_swap_mutation*; and *tournament_sel, two_point_crossover, swap_mutation*.

All of our configurations managed to find a perfect solution for the medium-level Sudoku puzzle.

Tournament Selection, Uniform Crossover, Swap Mutation

The optimal parameters identified for this configuration are a crossover probability of 0.8, a mutation probability of 0.05. The GA consistently improved the fitness score across generations, with early stopping at generation 203.

FPS, Single Point Crossover, Swap Mutation

The optimal parameters identified for this configuration were a crossover probability of 0.9, a mutation probability of 0.15. The GA consistently improved the fitness score across generations, with early stopping at generation 341.

FPS, Uniform Crossover, Subgrid Swap Mutation

The optimal parameters identified for this configuration were a crossover probability of 0.8, a mutation probability of 0.15. The GA consistently improved the fitness score across generations, with early stopping at generation 266.

Tournament Selection, Two Point Crossover, Swap Mutation

The optimal parameters identified for this configuration were a crossover probability of 0.95, a mutation probability of 0.05. The GA consistently improved the fitness score across generations, with early stopping at generation 108.

Hard Level

For the hard difficulty Sudoku puzzle, we used exactly the same configuration of methods and operators as we did in the medium puzzle.

Tournament Selection, Uniform Crossover, Swap Mutation

The optimal parameters identified for this configuration were a crossover probability of 0.8, a mutation probability of 0.1 and achieved a best fitness score of 2.

FPS, Single Point Crossover, Swap Mutation

The optimal parameters identified for this combination were a crossover probability of 0.9, a mutation probability of 0.15. Despite testing various configurations, the GA consistently achieved a best fitness score of 8.

FPS, Uniform Crossover, Subgrid Swap Mutation

The optimal parameters identified for this combination were a crossover probability of 0.8, a mutation probability of 0.05. Despite testing various configurations, the GA consistently achieved a best fitness score of 8. Most configurations, regardless of mutation probability or crossover probability, did not improve beyond a fitness score of 8.

Tournament Selection, Two Point Crossover, Swap Mutation

The optimal parameters for this combination were identified as follows: a crossover probability of 0.9, a mutation probability of 0.15. Despite testing various configurations, the genetic algorithm (GA) consistently achieved a best fitness score of 5.

Plot analysis

Across various genetic algorithm (GA) configurations tested, we observed common trends in fitness convergence and population diversity.

In FPS (Fitness Proportionate Selection), convergence is steady, showing a gradual improvement in fitness over generations. The population steadily moves towards optimal solutions without sudden changes, indicating a balanced exploration and exploitation. However, there is a risk of not converging due to the proportional nature of selection, which can sometimes maintain suboptimal solutions for too long.

In Tournament Selection, convergence is rapid, with fitness values improving quickly in the initial generations. This method efficiently selects the best individuals, accelerating the optimization process. However, there is a risk of getting stuck in a local optima due to sudden convergence, potentially limiting the exploration of other promising areas in the solution space.

With Swap Mutation, population diversity starts high and continuously decreases. This mutation operator maintains a high initial variability, allowing extensive exploration of the solution space, but diversity gradually diminishes as the population converges on optimal solutions.

In contrast, Subgrid Swap Mutation sees population diversity starting low and increasing rapidly after several generations. Initially, the population lacks variability, but as the mutation operator introduces new combinations, diversity quickly rises, enhancing the exploration of new solutions.

These trends are consistent across different configurations and are reflected in the respective plots, demonstrating the distinct characteristics and behaviors of each selection and mutation method within the GA framework.

Conclusion

In our effort to tackle the Sudoku puzzle, our chosen solution was the configuration combining Tournament Selection, Uniform Crossover, and Swap Mutation. Parameters were fine-tuned for performance: a population size of 500 individuals, 400 generations for thorough exploration, and a balance between exploration and exploitation with a crossover probability of 0.8 and a mutation probability of 0.1. Furthermore, integrating an elitism strategy ensured that the fittest solutions persisted across generations, fostering continued improvement. This configuration yielded a remarkable fitness score of 2 for the hard-level Sudoku puzzle.

This was our best solution, but it could have been different. As we have seen in the plots, the FPS method takes longer to converge, indicating that while it maintains diversity and steadily improves, it might not achieve the same rapid optimization as Tournament Selection. Therefore, the effectiveness of the solution is highly dependent on the number of generations chosen.

Future work

Several strategies could enhance the genetic algorithm's performance and robustness for solving. Increasing the number of generations and the population size, while computationally expensive, could lead to better solutions. Developing a new fitness function might offer more nuanced assessments of solution quality. Additionally, testing the algorithm without elitism could reveal its impact on convergence and solution quality. Exploring these areas could yield more effective configurations for solving Sudoku puzzles.

Appendix

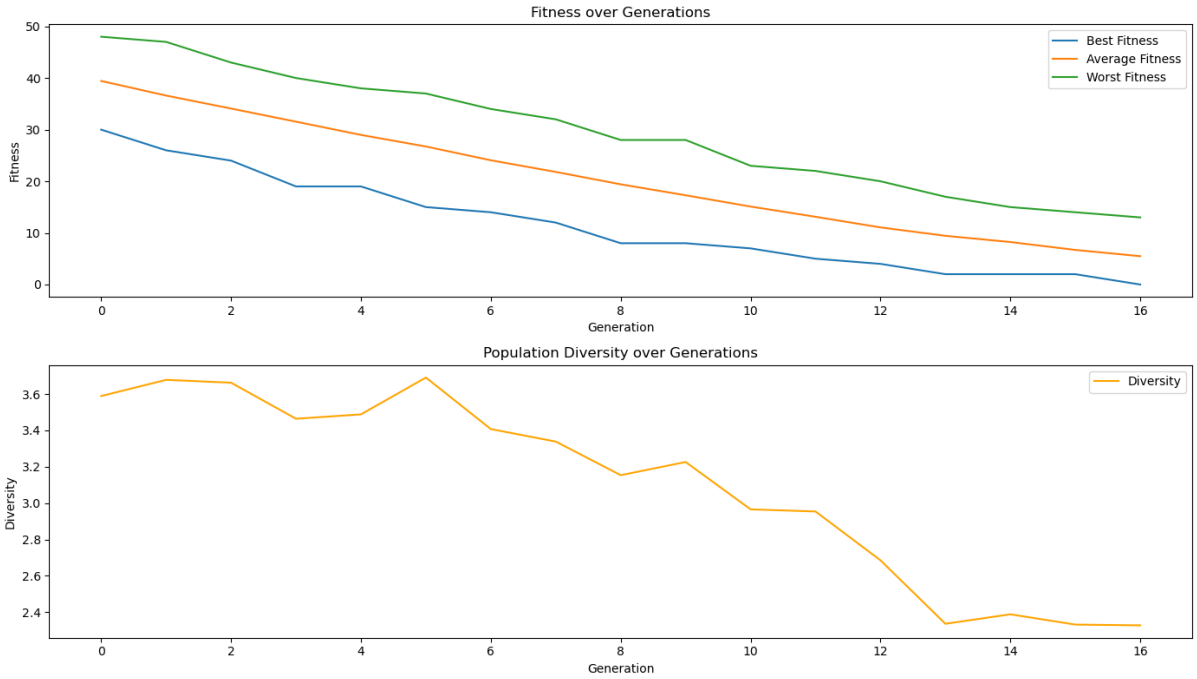

Fig. 1 - Easy level with tournament Selection, uniform crossover, swap mutation plots

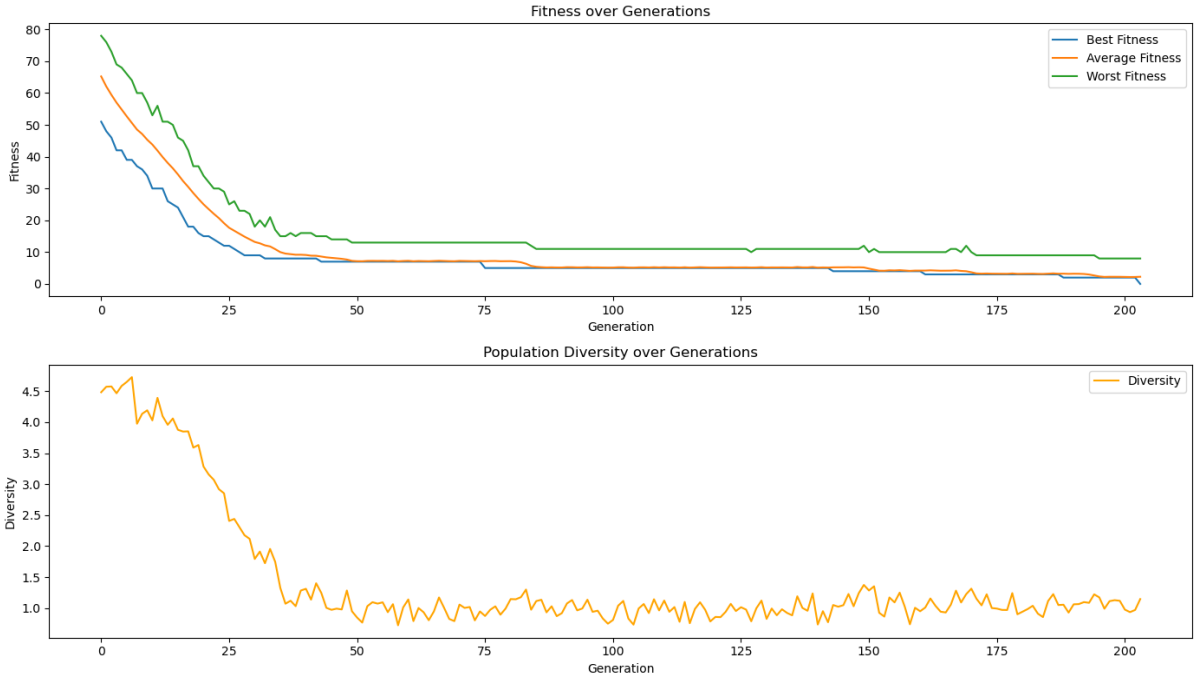

Fig. 2 - Medium level with tournament selection, uniform crossover, swap mutation configuration plots

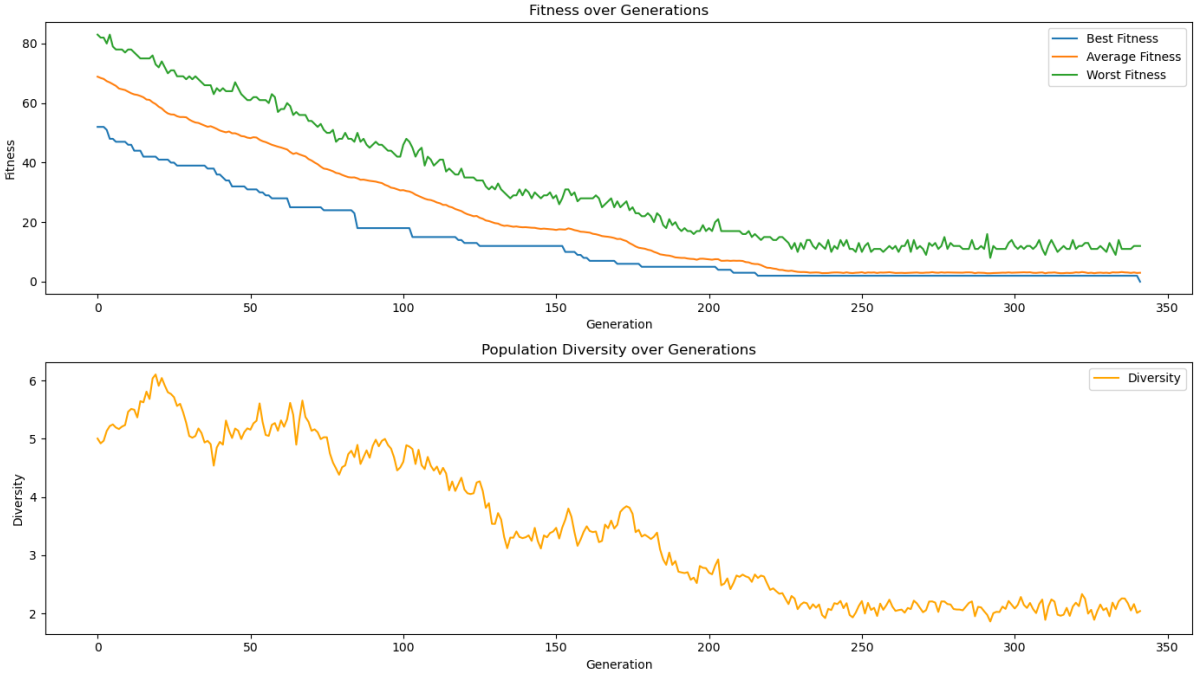

Fig. 3 - Medium level with fps, single point crossover, swap mutation configuration plots

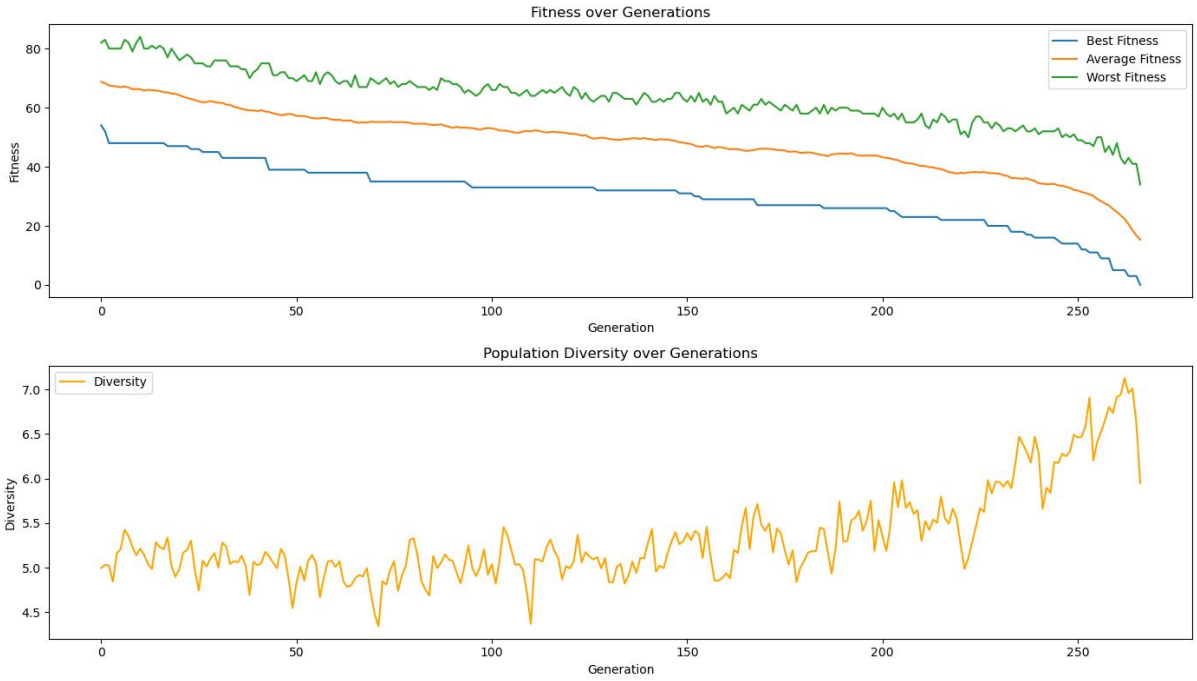

Fig. 4 - Medium level with fps, uniform crossover, subgrid swap mutation

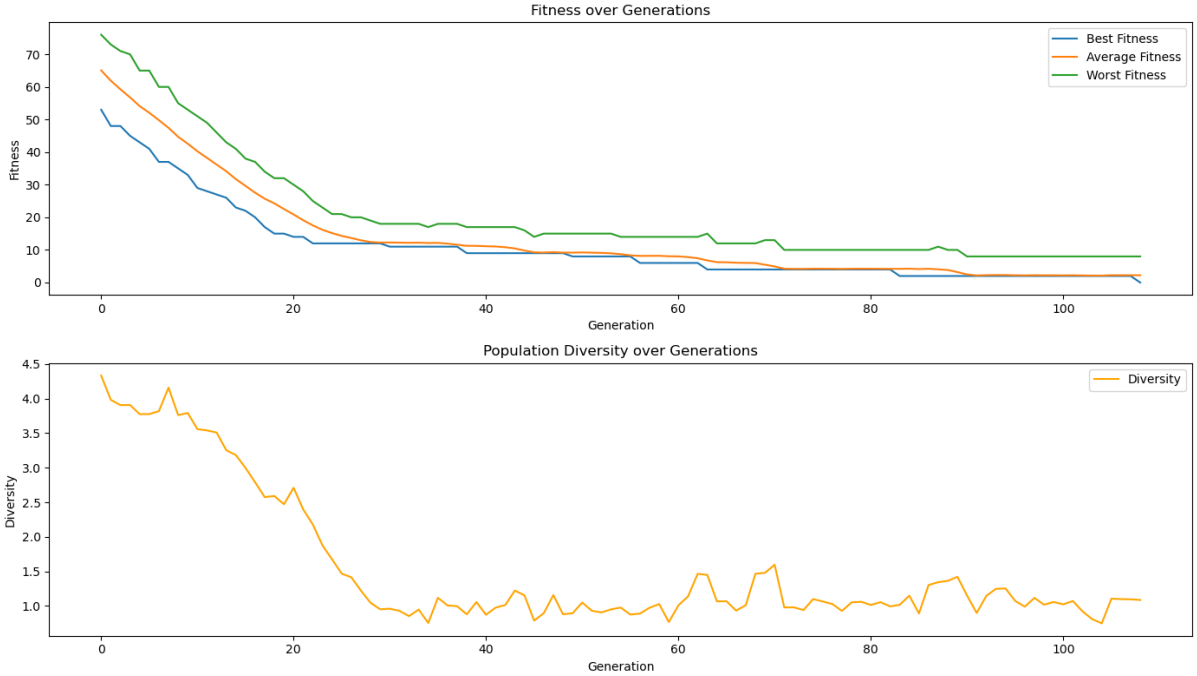

Fig. 5 - Medium level with tournament selection, two point crossover, swap mutation

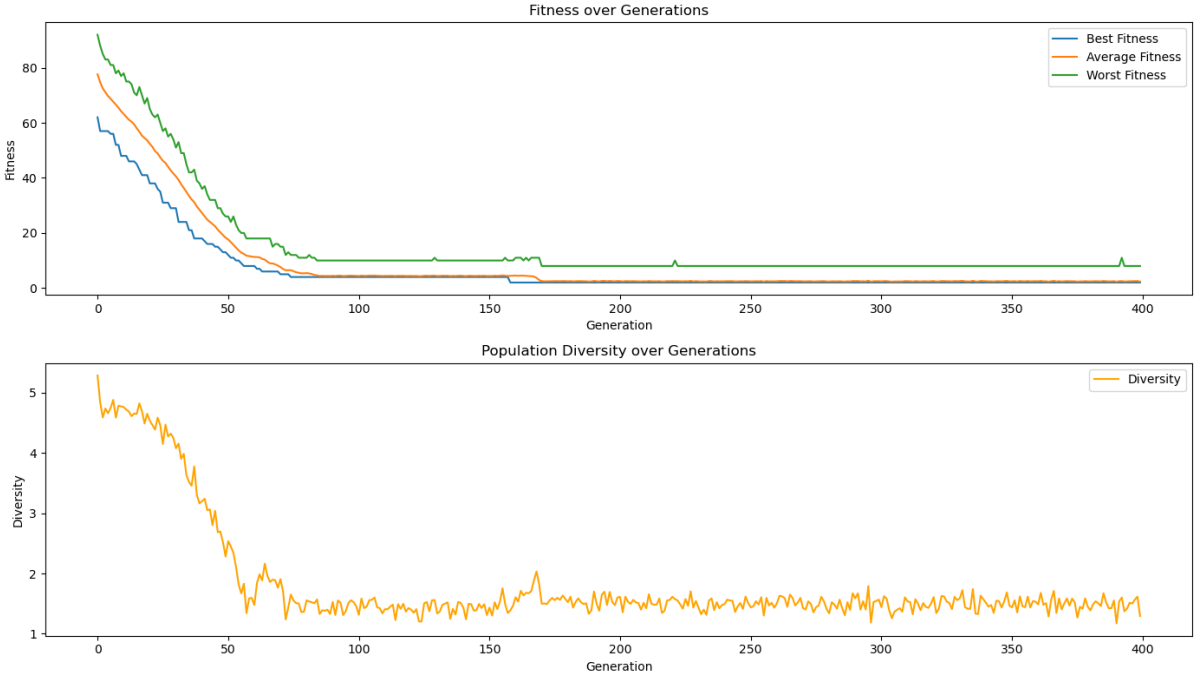

Fig. 6 - Hard level with fps, single point crossover, swap mutation configuration plots

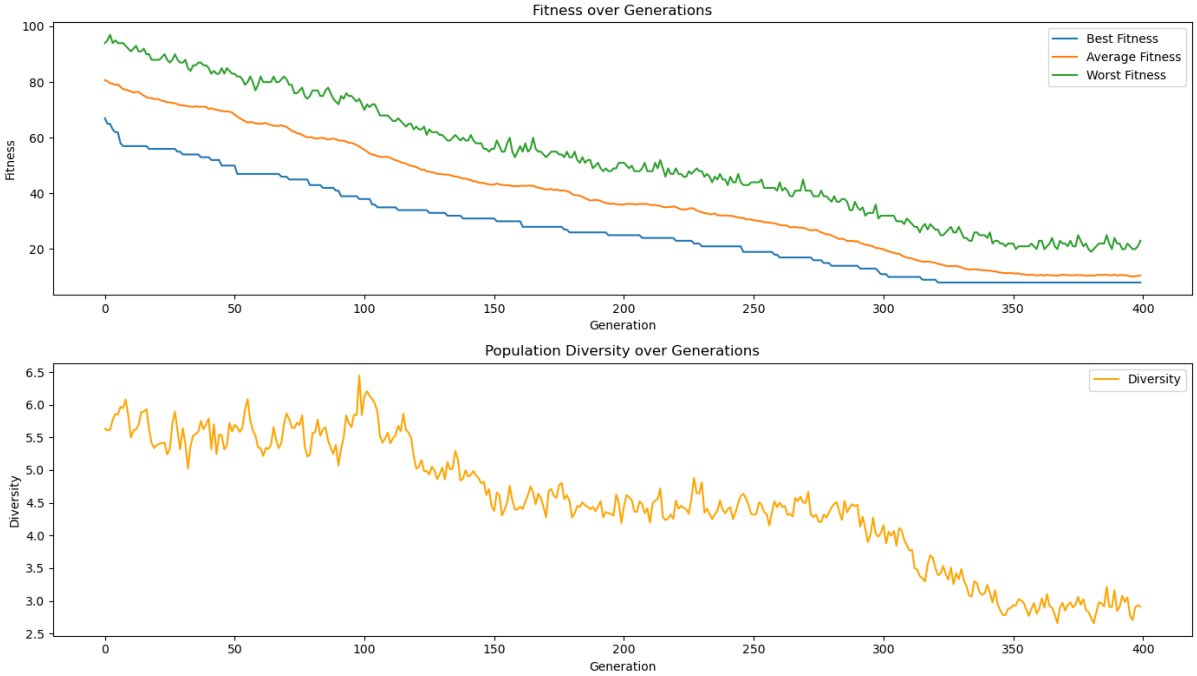

Fig. 7 - Hard level with fps, single point crossover, swap mutation configuration plots

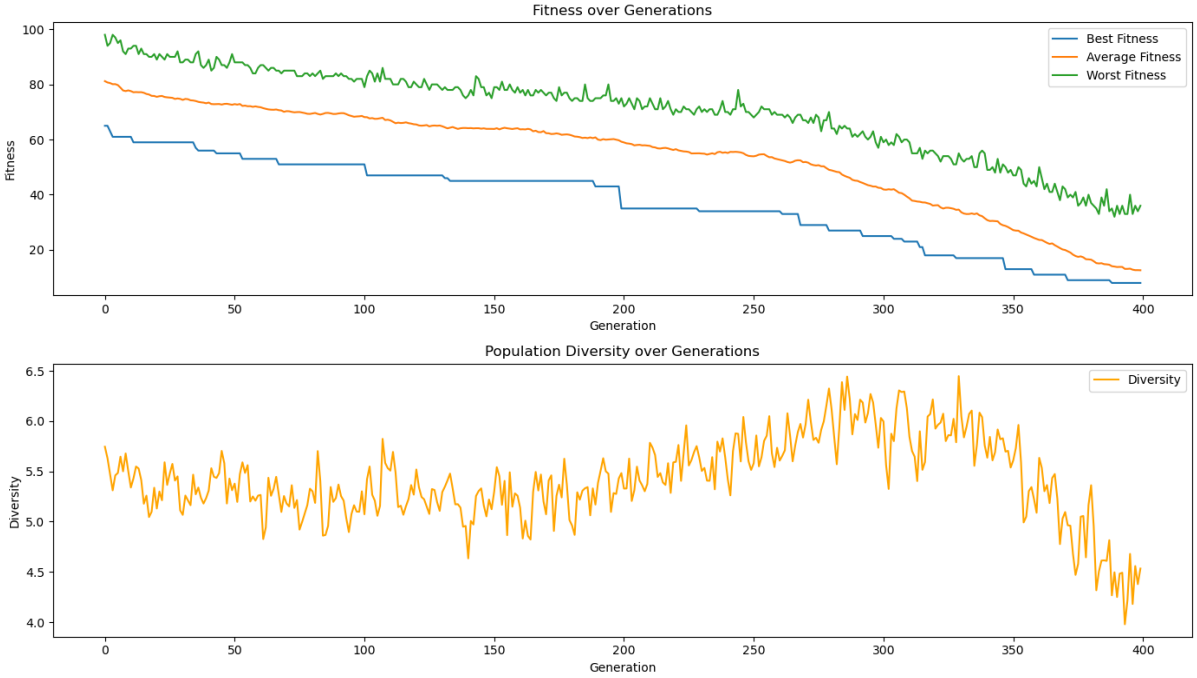

Fig. 8 - Hard level with fps, uniform crossover, subgrid swap mutation

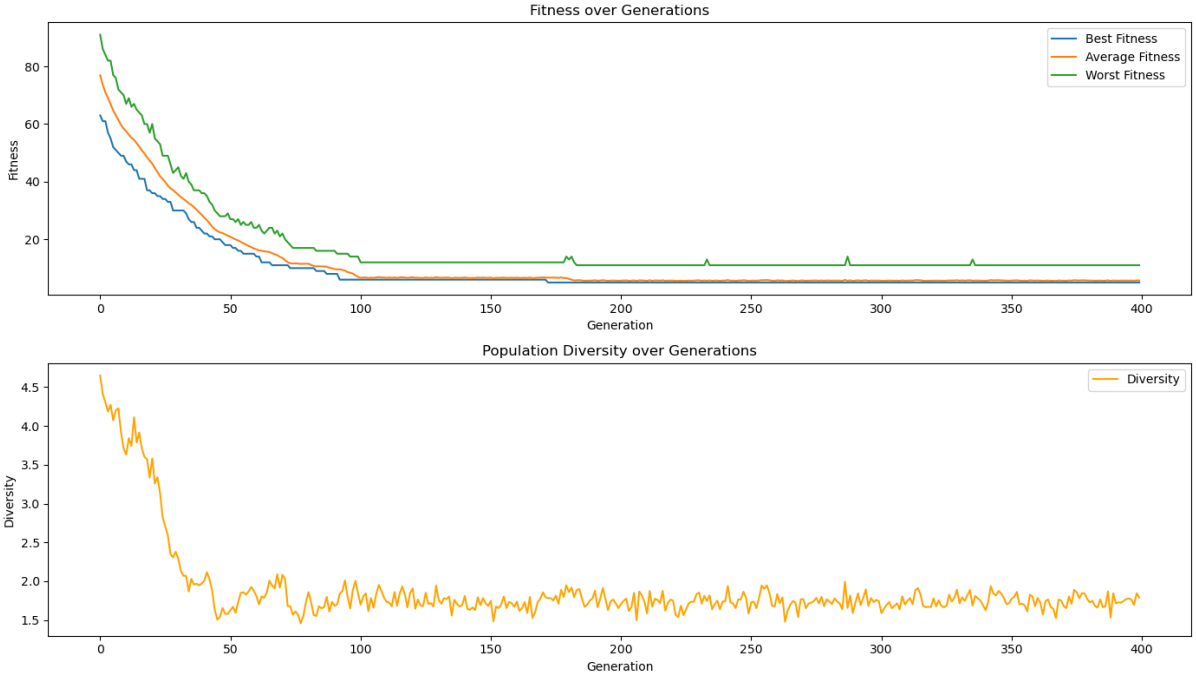

Fig. 9 - Hard level with tournament selection, two point crossover, swap mutation