# Sudoku Solver

CIFO 20/21 Project

Bruno Belo | Tomás Amaro | Tomás Santos | Vasco Pestana

r20170735 | r2016660 | r20170734 | r20170803

# 1. Introduction

Based on the evolutionary theory of Charles Darwin, the genetic algorithms reproduce in a computational way the natural selection, where the fittest individuals are selected for reproduction in order to produce offspring's for the next generation.

This project aims to explore and apply concepts of Genetic Algorithm optimization, in the design and development of the same in order to solve Sudoku puzzles with different levels of difficulty.

For this we use some of the various types of operators taught in class by teachers and some others implemented by us, which we think fit our problem, after some research by our group.

# 2. Genetic Algorithm

Regarding to solving Sudoku boards, we're talking about a maximization problem. For our Genetic Algorithm, we used 3 Selection, 2 Crossovers, and 3 Mutations as we are going to enter in further detail.

## 2.1 Selection

In this phase the algorithm selects the fittest individuals and then passes the genes of the selected individuals to next generation. Parent selection is critical to the convergence rate of the genetic algorithm as good parents drive individuals to fitter solutions.

We used these 3 known selection methods:

- **FPS (Fitness Proportionate Selection):** One of the most popular ways of parent selection, here every individual as a chance to become parent with a probably proportional to its fitness. So fitter individuals have better change to mate and propagate their features to the next generation.
- **Rank Selection:** Mostly used when individuals have close fitness values. This leads to every individual have almost the same probability of being selected as parent. Therefore, lead to a loss of pressure towards fitter individuals, making the Genetic Algorithm to

make poor selections. So, in the rank selection we remove the term fitness value from the equation, but every individual is ranked based on its fitness. The first ones in the rank are preferred to be selected comparing to the last ones.

- **Tournament:** Also very popular as it can work with negative fitness values too. In this selection method we basically choose random individuals from a random population, and we put them 'fighting' each other as the best into the selected become a parent.

## 2.2 Crossover

As we know, the Crossover function works as the sexual reproduction. Two parents are chosen in order to proceed the crossover of their genes to produce offspring's, introducing diversity.

We implemented the following:

- **Single Point Crossover:** Randomly attributing a crossover point, each parent gives their genes to their respective offspring until that point, in other words, each row keeps their first x values, where x is the crossover point. Beyond the crossover point, the offspring receives the genes from the other parent, however, we couldn't have duplicated numbers since that is one rule of Sudoku. In those cases, for the index where the value would be repeated, the offspring's inherit the value from respective parent instead.
- **Cycle Crossover:** Randomly attributing two crossover points, the first between 0 and 8, and the second between 1 and 9. After that, we created two child's with the same length of the parents but filled with 0's, and the crossover will only stop when there's no 0's in both offspring's. While the values are different, from both parents for the same index, the offspring's inherit the values from their respective parents in that index. When the values for the same index are equal, we increased the cycle, and the offspring's inherit the values from the other parent for that index.

## 2.3 Mutation

Mutation is the genetic operator responsible for ensuring genetic diversity from generation to generation. Although in our work we work a lot with some randomness throughout the various phases, which in itself already generates some diversity, but it is in the mutation that is guaranteed that the population is not stagnant, that is, it guarantees the diversity of the population.

It consists of randomly choosing 2 elements from the population and randomly generating a number between 0 and 1, and if the generated number is smaller than the selection rate, the element with higher fitness will be selected, otherwise the weaker element will be selected.

We implemented 2 know methods:

- **Swap Mutation**: It consists of a random choice of 2 elements belonging to the same line, also chosen at random, to swap positions with each other. With this it is necessary to always take care to verify that, first, both positions are free to make the exchange and then verify that this same random exchange of positions does not violate the rules for the solution of the game, more specifically, that there is no repetition of numbers in the same row or column.

- **Inversion Mutation:** This type of mutation is based on the random selection of 2 elements from the same row and repeating the verification logic of elements in the rows, in order to verify that both positions are vacant and that there is no repetition of elements in the rows and columns, invert the order of all elements belonging to the range of positions between the two randomly selected elements.
- **Scramble Mutation:** This mutation is very similar to inversion mutation, the only difference between scramble and inversion mutation is that instead of inverting the order of the elements within the defined interval, these same elements are randomly shuffled.

As for the mutation rate, a base value is assigned with which the operator is started, but if the population stagnates or does not guarantee the supposed diversity for 80 generations, the mutation rate value is multiplied based on the fraction of generations already covered and the total possible number of generations.

## 3. Results

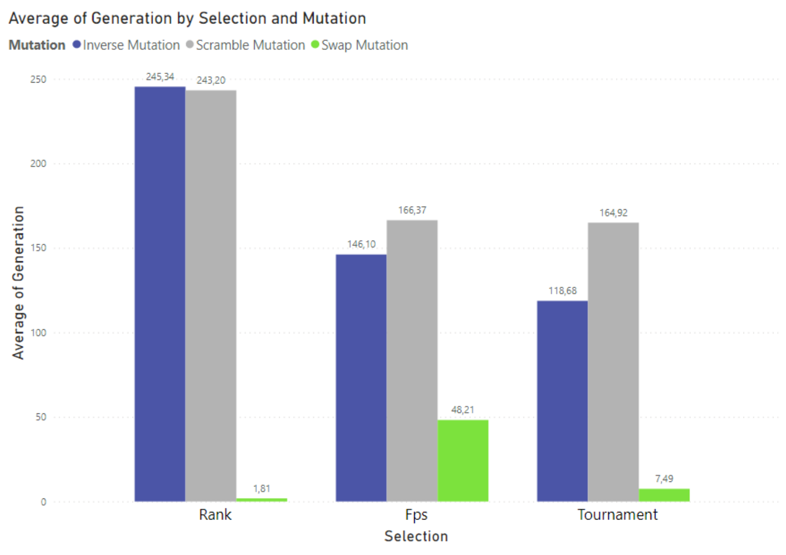

*Figure 1- Avg of Generation by Selection and Mutation*

In this plot it is possible to see that there is a big difference using the Swap Mutation compared to the other two options and for that we decided that this was the best Mutation to use.

Being the fattest one using the Rank Selection with an average of 1.81 generations per run and using the Tournament Selection we also got very positive results taking only approximately 8 generations to find a solution what makes these two options candidates to being chosen as the best Selection.

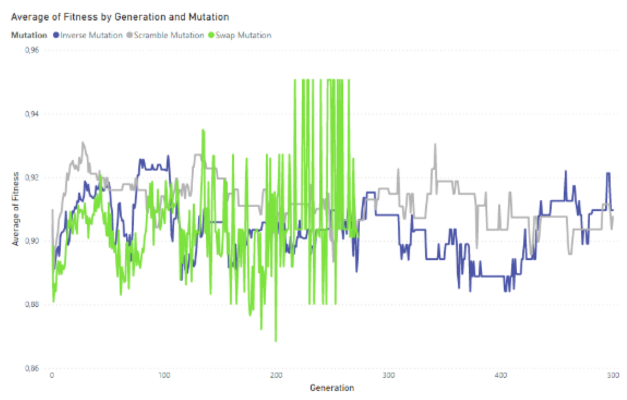

*Figure 2- Fps*

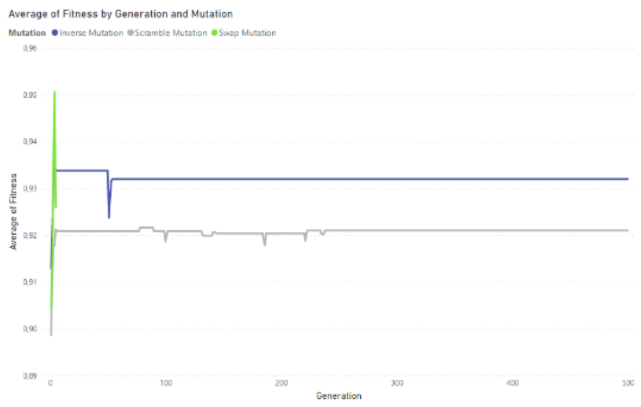

*Figure 3- Rank*

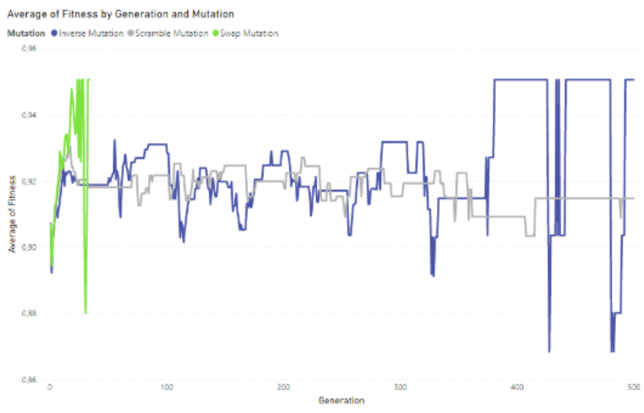

*Figure 4- Tournament*

These 3 graphics depict the development of fitness over the generations, and whenever the code finds a solution, it does not register it on the data frame, we can only observe how stagnant the algorithm becomes when it finds a "local optimum". Between the Tournament selection and the FPS selection, we can conclude that the FPS is faster to recover from an "optimum location". When it comes to Ranking Selection, it tends to remain stagnant in the same "optimum location" until the population is reseeded. Even though the development when we use the Ranking Selection may even be faster, in this specific situation where the objective is to find a perfect and unique solution (fitness = 1), it is not at all the best to be used. The behavior shown is too greedy, even if it was the one that proved to be the fastest when it is combined with Swap Mutation. In a real situation where the goal would be to achieve maximum consistency, we think it is not the most suitable. It is possible that the samples obtained do not demonstrate the reality, since the code was executed 100 times in each combination, however the random seeding solved the problem in generation 0 several times, before even any function was applied, so for this statistic only used about 80 runs of each combination.

For the reasons mentioned before we decided that the best parameters to use were the Swap Mutation, Tournament Selection with the Cycle Crossover, as we could not obtained enough results from the Single Crossover it not possible to compare results for different Crossovers.

# 4. Conclusion

The purpose of our project was to solve Sudoku boards through a Genetic Algorithm. With the help of the code developed in our CIFO classes and some research through the web, we built our code with 3 Selection, 2 Crossovers, and 3 Mutations. After testing and analysing the possible combinations we concluded that the best combination for solving the "easy" Sudoku, were the Swap Mutation, Tournament Selection with the Cycle Crossover.

# 5. Limitations

We faced some limitations towards our projects. Despite our efforts in performing all of the combinations, we could only get results for the Cycle Crossover, since Single Point Crossover took more time than we were expecting to run.

For the creation and development of our project we followed different implementations that were published and discussed in GitHub as well as the code provided by our teachers.

## Main Classes/Files:

- **Individual:** The class individual has as main objective to represent each individual in the population, in other words, which individual will represent a sudoku solution. It is also present in this class the function "update_fitness" which is responsible for the calculation of the fitness of all individuals.
  Being the fitness one of the most important part in a genetic algorithm project, we will describe the design behind it. It was applied in a way where it would be evaluated in two different ways and then summarized into the final fitness. First we will do the line_counter, where we will confirm the number of different number for each line, for each one we will add to line_counter (1/9)/9, if there is no duplicate numbers in any line counter will assume the value 1 ((1/9)/9)*81=1 .After confirming the lines we will see if each block of 3x3 also has no duplicate number, following the same approach where, if all numbers are unique in all blocks the block_counter will be 1 in the end to calculate the fitness we just need to multiply with each other, which will translate into a number between 0 and 1 where 1 is a solution for the sudoku.

- **Population:** Population is the representation of the group of individuals and will have as its main objective to make sure that the functions needed are applied to each individual when there is the need to apply the same function to the entire group as it is the case when we need to calculate the fitness of the entire population, also is responsible for the reseed.
- **Solver:** We can call solver the "brain" of the entire algorithm, it is responsible for calling all the functions and it happens in loop until we find a solution. In this class there is a big number of important steps as it is responsible for calling all the other functions and classes. Inside Solver class we decided to dynamize some static values, like the mutation rate, each time a generation finishes, and the two best individuals (based on their fitness) are the same, there is a variable called stale that increases, if stale reaches a certain value the mutation_rate starts increasing and increasing until the populations isn't stale anymore. Solver is responsible for calling all the different types of Selection, Crossover and Mutation methods applied based on the selected ones in the file "setup". Also, in this file there is a class called Fixed responsible for maintaining the original gid values and confirm if there is any duplicate value.
- **Setup:** Setup as the name suggests is where we sore all the possible specifications and parameters for all classes and functions. As it was not possible to store those values in their respective file, because it will origin circular imports or references, so this file is not importing any other, and it is being imported for all the other files instead. We can think of setup as being a database of parameters that can be edited in a simple and fast way.
- **Csv_saver:** Csv_saver was created in order to save the results obtained when we run the code in batch mode to test all the different combinations and compare the results obtained in each one. Those results are being added to a dictionary and inserted into the final dataframe at each generation, saving as an index the selected Selection, Mutation and Crossover method as well as the number of the run in question. In the end we used the function save_dataframe in order to transform our dataframe into a csv file for further analysis.
- **Main:** Main file has as is only objective to be executed and to run and call Solver. The only possible configuration is between the normal run or the batch run, when the setup variable Batch_run = False main will only call solver one time with the Selection, Mutation and Crossover functions specified in the setup file, if Batch_run = True main will call solve as many times as needed with all the different setup combinations.