CIFO Project Report | Sudoku Solver

Pedro Ribeiro (R20181114@novaims.unl.pt), João Ribeiro (m20201026@novaims.unl.pt), Patrik Pop (m20210678@novaims.unl.pt)

Abstract -

For our project we chose to tackle the problem that aimed at using a genetic algorithm (GA) to solve a sudoku puzzle. In our iteration of this project we will be attempting to solve a 9*9 version of the puzzle, which is the most common variety of the game. From our initial research most Genetic Algorithms prove ineffective at solving this type of problems, so we will be attempting a variety of configurations to try to obtain the global optimum.

Our code can be found in our **git repository**.

Keywords - Genetic algorithm · Sudoku puzzles · Optimization

1- Introduction

The sudoku is a logic-based combinatorial number-placement puzzle which was created in 1979. Genetic Algorithms is a metaheuristic inspired by the work of Charles Darwin, the theory of evolution, which tries to tackle problems by applying operators such as mutation, crossover and selection. In this project we aim to combine both, solving a Sudoku using a GA and presenting the results of our implementation.

Throughout the whole implementation our objective was not only to solve the problem but also provide a high level of abstraction in order to make it as customizable as possible and be able to provide us with a higher array of data to analyze.

2 - Implementation

For the GA to be successful we had to define our implementation.

Sudokus: Our sudokus were taken from the website sudoku.com and their difficulty ranks from easy to medium, hard, expert and evil. What changes between them is the amount of pre-filled values making the harder ones have less of them.

Representation: Our sudoku would take the 9*9 boards, in an array of 81 numbers which had the unsolved values initialized at 0 and only some cells with the correct values. We would then use the sudokuGame class to transform the array into a matrix.

Population: The population would be an aggregate of all the possible sudoku solutions at a given generation. In the population class we fill the population with the individuals and search which values need to be replaced based on the length of the puzzle.

Selection: In this class we initialize our selection methods mainly the tournament selection and the fitness proportionate selection**.** The first takes a randomly selected group of individuals to output the one with the best fitness and the second one works in a roulette fashion where each individual is assigned a probability of being selected based on their fitness value.

Crossover: Our main crossover method is known as the one point crossover which we adapted to work in a matrix.

We pick two individuals from our population and choose one row to split the sudoku. To exemplify, the generated sudoku will be made by first parent values until the selected row and from there on it will have the second parent values.

Nevertheless, despite employing the aforementioned approach throughout the project, our implementation possesses a level of abstraction that allows the crossover to be done over N points instead of just one by changing the value of argument number_of_cutout_points in the crossover function.

Mutation: Mutations are the operator used to maintain genetic diversity between the generations. The percentage of mutations along our generations will be defined as a parameter which will have a certain dynamism. To test our problem, we will develop 3 types of mutations:

Swap will change row elements position pairwise verifying that we are not causing duplications in the rows columns and blocks.

Scramble picks two points in a row and shuffles all the values between those points.

Inversion operates like Scramble but instead of shuffling just reverts the order of the elements selected.

Fitness Function: The fitness function is what would dictate the success of our GA. Our solution to the function was to create a method that for each row, collum and block would search for each individual value from 1 to 9, when a value is not found it is represented in an array with a 0 and, when it's found, it is represented with a 1, we would then take all of the arrays and checks the uniqueness of all the arrays, if they are all similar it means that all our intended values are present and therefore the function rewards the individual, if they are not equal it means that some of the rows, columns or blocks have missing or repeated values and the fitness gets penalized for it.

Early Stopping: Our first implementations of the algorithm used an early stopping feature that upon receiving no improvements on the values of the fitness function, over a predetermined number of generations, ditched the entire population and re-started it with new individuals. This implementation was very successful at providing a solution for the sudokus, however, it did not allow us to generate accurate insights on how good our parameter selection was. To make our analysis more accurate we ended up ditching this re-seeding method and just chose to stop the run in its entirety. Additionally when we reach a solution our algorithm stops running, therefore the data on the final generations might become unreliable due to not all of the runs reaching it.

3 - Results

For a baseline all of our runs will be performed using a population of 1000 individuals over a maximum of 125 generations, over a total of 30 runs which was the value indicated by professor Vanneschi, we will be using a mutation rate of 15% but this value increases by 5% per 5 generations where the fitness function stays stale, this aims at helping us escape local optimum. As for the crossover it has a rate of 90%. Since our problem is a maximization one our goal is to reach our maximum possible value which is 1.

As we discussed in the implementation we implemented re-seeding which discards the entire population when it becomes stale, this makes our success rate jump from the 50 to 70 % range to a much more confident 100%. Consequently we will deactivate this feature when comparing runs because otherwise we would only have the running time to analyze.

When it came to solving our easiest puzzle, which contained around 54% of the values pre-filled, our algorithm had a 100% success rate, even when the re-seeding was turned off, with an average running time of 14 seconds per run, not needing to get past the 30th generation to solve the puzzle. For this reason we will be performing all other runs on the medium difficulty unless specified otherwise.

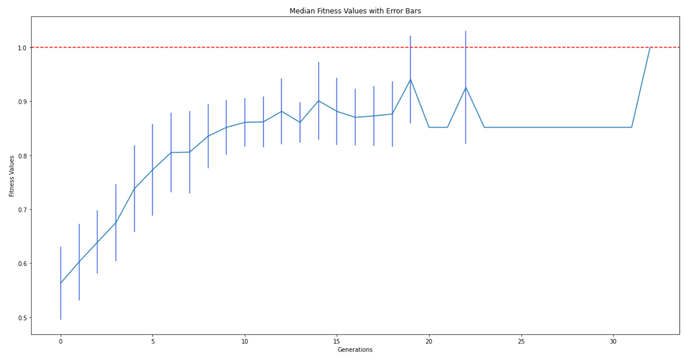

Fig 1 - Fitness landscape of the Easiest Sudoku.

Our first analysis is on the selection method we wanted to compare how efficient tournament selection was compared to fitness proportionate selection. We will be performing this for our easiest puzzle.

The results are all output with the swap mutation which will be discussed later.

Selection Method	Success Rate %	Average Mean Time in seconds per Run
FPS	73.3	36.16
Tournament	76.7	36.66

Although the difference in time is nonexistent the difference in the success rate made us keep using the tournament method in the following runs. It is important to notice that the difference in success rate might be due to the low number of runs but due to time constrictions we will accept these results.

When it came to analyzing our mutations we implemented 3 options, a row swap that changed the position of two values that had not been initialized by the sudoku, an inversion swap that reversed the order of a random segment of a row and a scramble mutation, which as the name implies randomized the position of a segment of a row.

These were the results.

Mutation	Success Rate %	Average Mean Time in seconds per Run
Swap	73.3	42.19
Scramble	0	72.92
Inversion	0	62.76

The difference in the success rate between types of mutations implemented can be explained by the implementation we did. The swap mutation verifies if it’s creating a valid mutation (not creating duplicates on columns and blocks), while the scramble and inversion ones don't have this type of validation, which often lead to a break in the sudoku logic.

4 - Final Solution

The configuration that was the most successful, used the swap mutation, tournament selection, elitism set to true and re-seeding set to true as well. As for the remaining parameters we still used a population of 100 over 125 generations with a mutation rate of 15 that increases by 5% over 5 stale generations, with a crossover rate of 90%.

If we, for a second, turn off the reseeding so that we can better analyze the success rate, this is how our solution fared out with the hard, expert and evil difficulty sudokus.

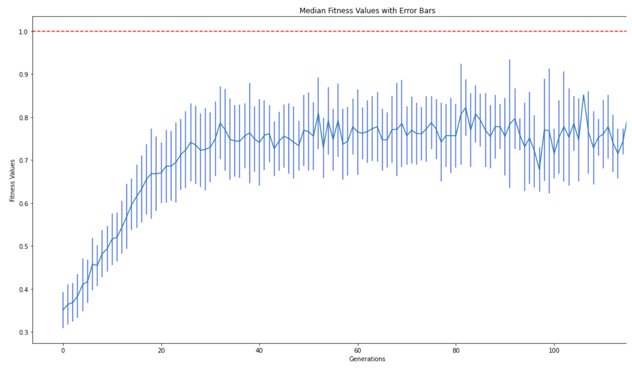

Fig 2 - Fitness landscape of the Hard Sudoku

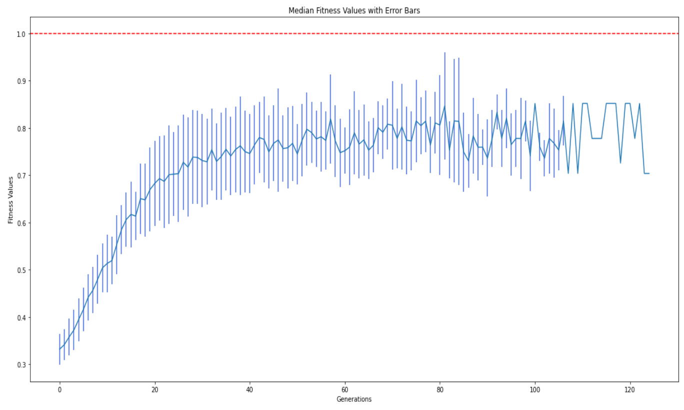

Fig 3 - Fitness landscape of the Expert Sudoku

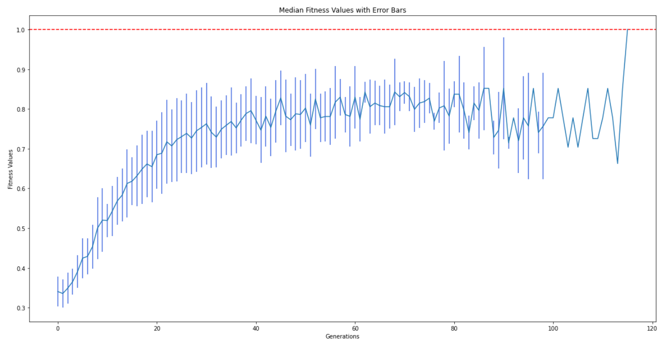

Fig 4 - Fitness landscape of the Evil Sudoku

As we can notice the fitness landscapes were very similar for all of the puzzles. The success rate was 67, 77 and 87% with an average running time of 42 seconds.

We then performed a final implementation where we turned the reseeding on and achieved a success rate of 93% while keeping the average running time at 41 seconds. The dip in fitness in the later generations is the result of the reseeding, since it's more likely for the last generations to be replaced, it is expected that the fitness returns to the initial values which results in the odd landscape that we have.

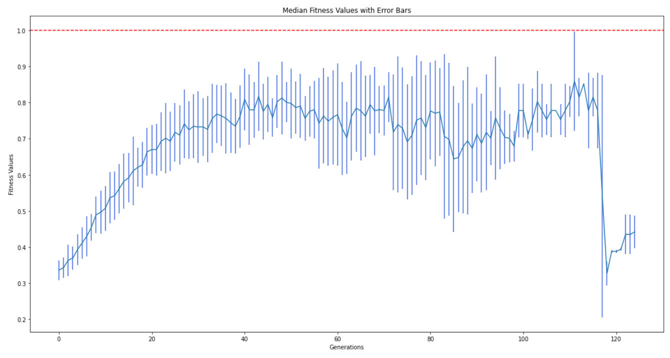

Fig 5 - Fitness landscape of the Expert Sudoku With Reseeding

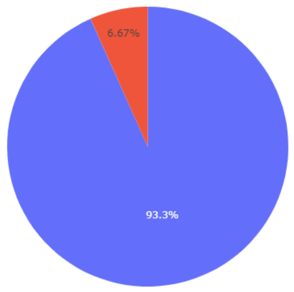

Fig 6 - Success Rate of the Expert Sudoku With Reseeding (1 - Success / 0 - Failure)

4 - Discussion

One of the things that limited our analysis, specifically of the fitness landscape, was the full stopping of the algorithms once the solution was found. Although it might not have been the optimal implementation it did save us a lot of processing time.

Our implementation was made with success and mind first and a variety of implementations second, and as such, our alternatives, for example for the mutation, were implemented after we already had a good solution. This caused the results of the mutations that weren't swap to be extremely poor.

Other aspects that could be improved on our project was the implementation of a smarter fitness function. Our fitness did not consider deviations from the original sudoku and this was probably what made the inverse and scramble mutation fail so drastically.

The results on the success of the algorithm for the harder puzzles were also incredibly surprising, but they most likely came from the low number of runs, and the high value of individuals each population had, which most likely gave the populations enough genetic variety for it to be very successful for each of the sudokus.

5 - Conclusion

In this project we implemented a genetic algorithm-based sudoku solver. Our solution aimed at solving the most common problem that genetic algorithms face, which is getting stuck on local optima. For this reason our implementation had a reseeding function which restarted the population once it got stale. This proved to be an incredible solution and allowed us to reach our global maximum of 1 very reliably.

The failure of the other combinations of Parameters most likely came from the way we implemented our fitness function and from the constrictions of the sudoku itself.

6 - Division of Labor

The initial skeleton of the code was developed by João, once we had an initial implementation Patrick and Pedro worked to add more features and increase the efficiency of the code. As for the report and the analysis it was written by Pedro, but always with the input of all students.

7 - References

[1] Introduction to Genetic Algorithms: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

[2] A novel hybrid genetic algorithm for solving Sudoku puzzles, by Xiu Qin Deng and Yong Da Li
https://www.math.uci.edu/~brusso/DengLiOptLett2013.pdf