SOLVING SUDOKU WITH GENETIC ALGORITHMS

COMPUTATIONAL INTELLIGENCE FOR OPTIMIZATION
NOVA IMS
2021/2022

Carolina Rodrigues | m20211298

Elena Nozal | m20210989

Elsa Camuamba | m20210992

Manuel Pedro | m20210999

1. Introduction

The aim of this project is to implement Genetic Algorithms (GAs), applied to a Sudoku problem in order to obtain the best solution. To do so, we have designed a fitness function and other appropriate genetic operators to reach an optimal solution (e.g. crossover, mutation and selection).

The main objective for the Sudoku problem is to obtain an optimal solution and, in the best case scenario, find the solution to different levels: easy, medium and difficult. The code implemented in this project can be accessed in the **GitHub repository** through this ***link***.

2. Representation

A puzzle is represented in a matrix of 9x9, in which the empty cells are represented by zeros. Objects of the class Individual, when initialized, solve the puzzle in the following manner:

- A list of lists is created by iterating over all of the cells of the puzzle, and for each cell that has a zero, a list of possible digits is created - digits from 1-9 that are not present in the corresponding row, column and block.
- For each row, the algorithm iterates over each cell with value zero, and randomly selects a digit from the corresponding list of possibilities.
- The process above will continue looping while the row does not contain 9 unique digits, belonging from 1-9.

We have seen that by forcing the puzzle to avoid repetitions in rows, we obtained better results and it was easier to implement GA operators such as cycle crossover and inversion mutation. Therefore this was the approach we kept regarding representation.

Additionally, individuals have a solution key attribute to facilitate genetic operations, which consists of a list of lists, only containing the row-wise elements to be filled (positions of the zeros in the puzzle)

3. Fitness Function

The sudoku solver could easily be a maximization or minimization problem, however we decide to look at it as a maximization one. As so, we designed a fitness function that calculates the fitness of each solution based on the sum of the unique values for all the rows, columns and blocks. Therefore, a perfect solution of a sudoku should have a fitness of 243. Regarding other approaches considered, we thought of multiplying instead of doing the sum of the number of unique numbers in each row, column and block. Also, computing the average of the sum of unique values of the grid or the average of the multiplication. However at the end we saw fit to keep the sum of the total unique numbers as the results were easier to read and understand.

Worth to mention that, keeping in mind the abstraction of the code, we have also implemented a fitness function for minimization. This code works as the maximization one, but instead of counting the unique values, it counts the non-unique values per row, column and block. Therefore, a perfect fitness for a solved sudoku is equal to 0.

4. Genetic Operators

4.1. Selection

Three selection methods, namely fitness proportional selection (or roulette wheel), ranking selection, and tournament selection were used. Although our Sudoku problem was approached as a maximization problem, the overall implementation for the selection methods are defined for both minimization and maximization problems. Below, we discuss in more detail the implementations with regards to minimization for fitness proportional selection (FPS) and the full implementation for ranking. Since tournament selection was fully explored during the practical classes, no additional modifications were needed.

Fitness Proportional (Roulette Wheel) Selection: To apply FPS to a minimization problem, the probability of selecting an individual is defined based on the inverse of its fitness value [1]. Using a roulette wheel analogy, the following steps were taken:

- Calculate the sum of the inverse of the fitness of each individual, 'total_fitness';
- Generate a random number from a uniform distribution in the interval [0, total_fitness], 'spin';
- Loop through the population by adding each (inverse) fitness to the variable 'position'. The individual for which 'position' exceeds 'spin' is selected.

Ranking Selection: As alluded to earlier, FPS is highly influenced by the fitness value [2]. In ranking selection, individuals are assigned selection probabilities on the basis of their rank, and not their fitness. As such, the implementation consists of the following steps:

- Sort the population according to their fitness value, 'sorted_fitnesses', from worst to best fitness value. This step is adjusted for minimization by sorting in the reverse order (from best to worst fitness value);
- Compute the sum of the fitnesses, 'total_fitness', using the Gauss summation formula
- Calculate the ranking scores for each individual, 'ranked_scores';
- Generate a random number from a uniform distribution in the interval [0, sum(ranked_scores)], 'spin';
- Loop through the ranking scores and retrieve both the 'index' and the corresponding rank score, 'rank'. The ranks are each added to the variable 'position' and the 'index' is used to access and select the fitness for which 'position' exceeds 'spin'.

4.2. Crossover

For both crossover and mutation operators, the main aspect we had to pay attention to was that the initial input values of the sudoku could not be changed.

We implemented this method in a row by row approach, this is due to our selected approach to generate the first solutions. As discussed earlier, we guarantee non-repetition on a row level since by allowing crossover to be applied by column, block or even to the complete sequence of the puzzle would remove the certainty of uniqueness on a row level. As such, applying crossover on a row by row basis allows us to program the algorithm as a problem without replacement.

Partially matched crossover: in Partially matched crossover (or PMX), in order to create the offspring, we define a window inside of strings. For child 1, we look at every index of parent 1, and we copy the value if the index is not inside of the window or if it is outside and there is not already that index in child 1. Whenever we cannot copy from parent 1, we copy the value from parent 2. We do the same to child 2.

Cycle crossover: in cycle crossover, the offspring is created by mixing the parents' loci row-wise using the cycle method. For the first offspring, for each row, the first locci is taken from the first parent, then the index for the loci in index one of the second parent is inherited from the first parent, and so on, until this cycle can no longer continue, the next unfilled index is then inherited from the second parent, and a new cycle is created using the same logic as above, until all indexes are filled.

This operation is made using the solution keys of the parents, not the full representations, so that the pre-filled spots in the puzzle are not compromised.

4.3. Mutation

Swap Mutation: in our case, no initial input value of the sudoku can be modified. Therefore a *while loop* is implemented so that when the random indexes of a row are selected, it checks that it was not a fixed value. The rest of the code is as seen in class.

Inversion Mutation: here, we created a code where we take an object individual and the puzzle as the arguments, and we define a function that, first of all, uses the solution key of the individuals to get the number of rows in which it will repeat the process.

Following the above implementation, we used a *for loop* to randomly choose to mutate or not each row (having a probability of 60% of choosing to do it). If the mutation occurs, the function will take the position of 2 mutation points, saving the starting and ending positions of these points and assuming that the second point is after the first one. It will invert all the values inside the window between the two points, creating a new row. After looping over all rows of the solution key, a new individual is obtained.

As objects of the class individual required a representation input, the code includes a transformation, using the imputed puzzle and the mutated solution key to get the corresponding representation.

5. Configurations and results

First, we tried all possible combinations with a pop_size =50. Obtaining the following results:

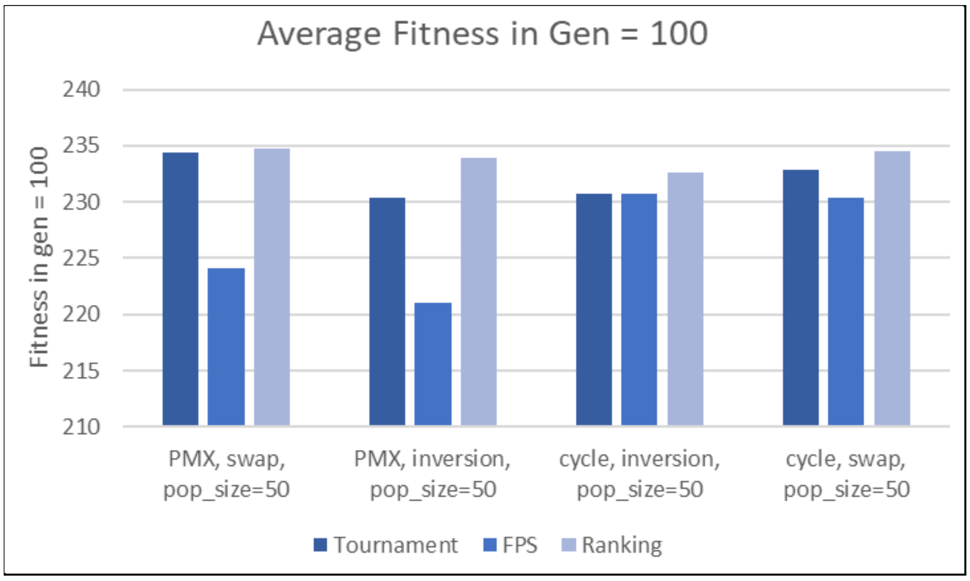

Figure 5.1. Average fitness for all possible combinations in the last generation, gen = 100.

In Figure 5.1., we observe that the highest average fitness is obtained with Ranking as a selection method for all different combinations, followed by Tournament. Both selection methods have similar performances and neither solves the very_easy sudoku level with the population size established at 50, probability of crossover fixed at 0.8 and a mutation rate of 0.2.

Consequently, we decided to explore the impact of increasing the size of the population keeping the rest of the parameters fixed. We then applied a population size of 500 to the four combinations that scored the highest fitness and obtained the following graph (Figure 5.2):

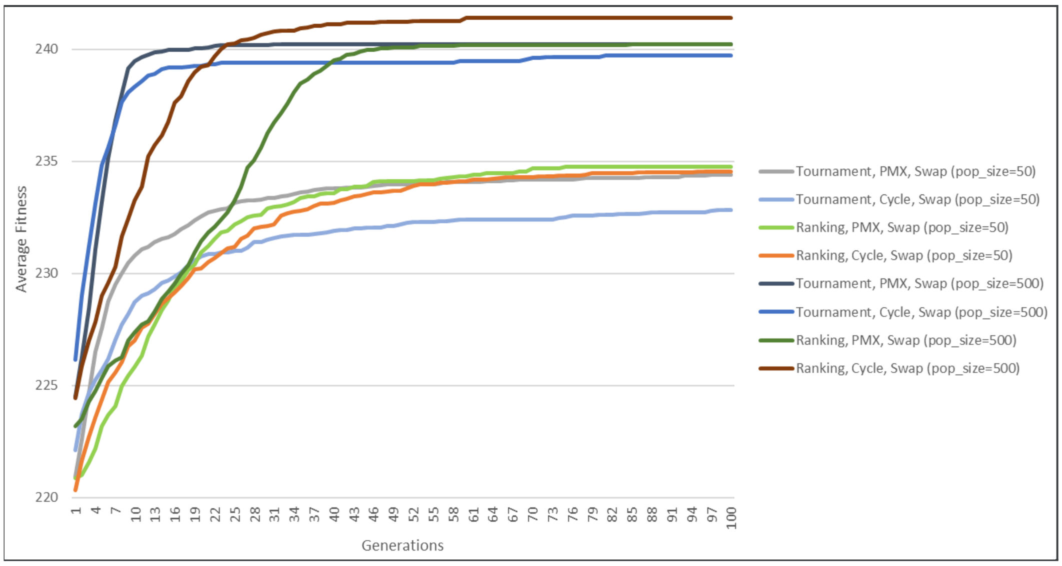

Figure 5.2. Evolution of the average fitness per generation for the four best approaches when size population is 50 and 500.

As we can see, with a larger population size the performance is slightly better. To evaluate the combinations, we compare the percentage the Sudoku is solved, the average of generations it takes to solve and the average fitness per generation. The results are shown in Table 5.1 below:

Configurations	Avg. fitness, gen= 100	Solved Sudoku	Avg. generations
Tournament, PMX and swap	240.23	23.3%	14
Tournament, cycle and swap	239.7	13.33%	26
Ranking, PMX and swap	240.23	20%	47
Ranking, cycle and swap	241.4	43.33%	26

Table 5.1. Parameters to decide which combination is the best approach for the combinations with pop_size equal to 500.

Finally, we attempted to solve the four levels of difficulty of the sudoku puzzle with the best combination found thus far: ranking, cycle and swap with pop_size = 500. The resulting graph is shown in Figure 5.3:

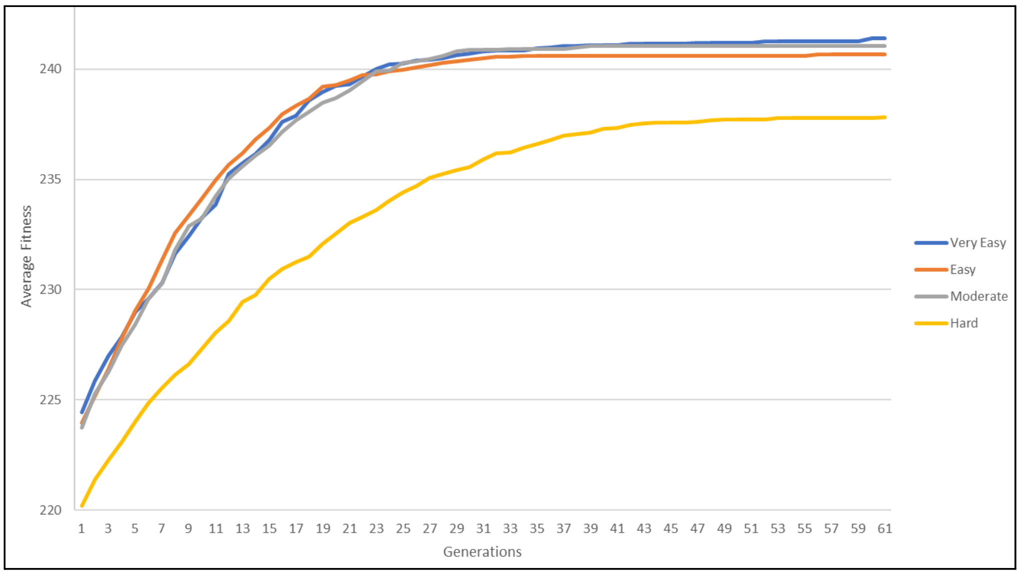

Figure 5.3. Average fitness per generation by level of difficulty of the Sudoku.

6. Conclusion

In this project, we successfully implemented different GA to a Sudoku: three selection algorithms, two crossover and two mutation operators. Multiple configurations were explored, being our best model: Ranking, cycle and swap. This combination is able to solve Sudoku in very easy mode 43% of the time in 26 generations on average, in easy mode 36,67% of the time in 24 generations on average and in moderate level, 33,33% of the time in 28 generations on average.

We noted that, as the difficulty level increases, the model takes more time to converge to a solution and is therefore less probable to solve the sudoku, Figure 6.1. This is due to how the fitness and representation of the Sudoku were defined. As the level of difficulty increases, the algorithm needs modifications in the representation and fitness declaration in order to converge to a solution and not get stuck in a local optimum.

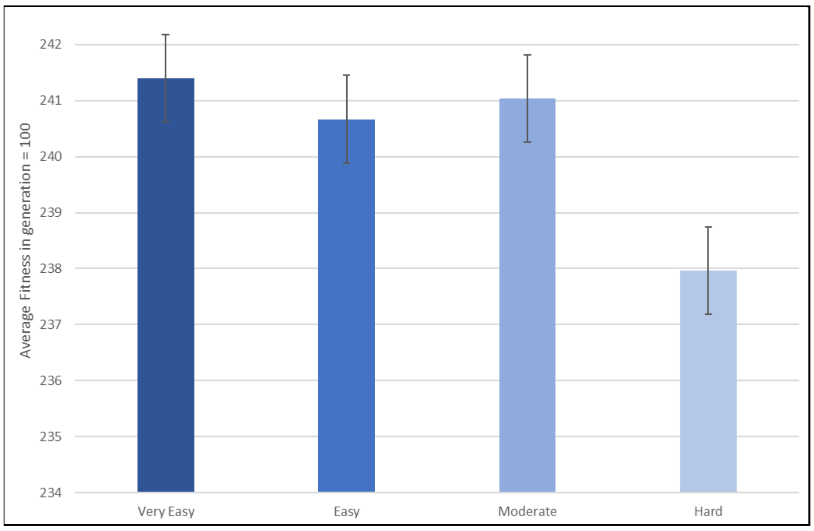

Figure 6.1. Average Fitness per level in generation equals to 100.

Furthermore, we have seen that different operators affect the convergence of our GA, such as the population size, the probability of crossover and mutation to occur and the different algorithms of selection, crossover and mutation as seen in section 5 above. We further note that all the results displayed have been carried out with elitism as the exclusion of elitism was seen to impact negatively our results (lower percentage of solved Sudoku, from 43% to 33%, and takes more generations to converge into the solution, from 26 on average to 40).

7. Future Works

Building upon the results obtained, for future studies, we would consider (i) evaluate whether altering the tournament size has any influence on the fitness values; (ii) for both the crossover and mutation rates, study what impact, if any, differing rates have on the fitness values; (iii) lastly, modify the fitness function and representation in order to solve a hard level sudoku.

Bibliography

[1][2]Vanneschi, Leonardo. "Computation Intelligence for Optimization". *Document extracted from the book "Lectures in Intelligent Systems".* https://elearning.novaims.unl.pt/pluginfile.php/110499/mod_resource/content/1/BOOKLET_CIFO_Leonardo_Vanneschi.pdf. Accessed on 17 May 2022.