Computational Intelligence for Optimization Project

MASTER DEGREE PROGRAM IN DATA SCIENCE AND ADVANCED ANALYTICS

Sudoku Solver

Group Members - 1

António Fonseca number: r20181154

João Carvalho, number: r20181122

05, 2022

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

INDEX

1 INTRODUCTION

Throughout the history of artificial intelligence, it has been used to test whether or not it is capable of solving tasks that for humans are considered fairly easy. One of those tasks is trying to solve simple games such as the Sudoku, the snake game, etc. Our group chose to develop a Sudoku Solver using a Genetic Algorithm. Several selection methods, mutation and crossover operators were tested in order to understand which techniques were the optimal to use in the solver. The code can be found here.

2 METHODOLOGY

2.1 IMPLEMENTATION

Regarding the creation of the sudoku game we started by defining the fixed numbers that will never change during the solving problem and defining a valid sample whose length summed with the number of fixed numbers previously defined is equal to the length of a sudoku game. Afterwards we defined a function that for each row it inserts in each NAN value a random number from the valid sample and removes that number from it. Having these functions created we can than generate the rows, columns, and squares which will then be used to calculate the fitness of the individual.

Concerning the **fitness function**, our initial objective was to implement a code that would penalize the individual when it had repetitive numbers in the same row, column or square. Given this objective, we

opted to develop a minimization problem and, therefore, create a function that would increase the fitness of the individual whenever it would find an error. Thus, we started by evaluating for each row the number of values that were repeated. In order to penalize rows that had many equal numbers, we decided to exponentially count the errors and add to the fitness rows score. It was created for the columns and squares a similar code, and the final fitness of the individual is the sum of all three separated fitnesses (rows, columns and squares). Moreover, it was developed another fitness function that would increase the fitness if a number (allocated to a NAN spot) from the initial sudoku's fixed numbers was equal to a fixed number in the same row, column or square. These two functions can be used separately or together. This last implementation would penalize in a greater scale the individual, since, from our point of view, a certainly wrong number (impossible number) is worse than a possibly wrong number (wrong number). These 2 concepts will be used further.

Additionally, it was created a function that defined the individual, a single sudoku, that it would always have the fixed numbers previously defined. Also, it was developed a function that generates an initial population of sudokus (individuals), and a function that its main goal is to calculate the fitness sharing coefficient of an individual with the others in order to later be used for preserving different individuals (to preserve diversity in the population).

Moreover, it was created a function that returns the indexes of a given sudoku which are incorrect. This was achieved by calculating in each row, column and square the number of repetitive numbers. It was also developed a similar function though, it is more specific since it only returns the indexes that are impossible to be right since they are in conflict with the fixed numbers previously defined in a given row, column or square.

2.1.1 Crossover Operators

Two crossover operators were developed. The first one, which is called *point_crossover* in the jupyter notebook, is a One-point crossover and, therefore, a point in both parents' rows is chosen randomly and all the numbers to the right of the crossover point are swapped between them.

The second crossover developed is a bit more complex since it only swaps a pre-defined number of changes in the parents' rows, and it never changes the fixed numbers defined in the beginning of the problem. Furthermore, it is capable of setting the values to be changed to be only the wrong ones or even the impossible ones (numbers which are in a direct conflict with the fixed numbers). We concluded, right in the start of the project that this crossover is more efficient than the One-point crossover.

2.1.2 Mutation Operator

Regarding the mutation operator, a single function was developed. The basis of this function is fairly similar to the second crossover created since not only it is capable of only mutate the indexes that are not fixed numbers, but it can also be tuned to only mutate the numbers which are wrong or even impossible to be right (numbers that have a direct conflict with the fixed numbers). This mutation replaces the old numbers with random numbers from 1 to 9 into the chosen positions.

2.1.3 Selection Methods

As for the selection methods, it was developed three different approaches. *Tournament Selection* which selects the best individual based on a tournament size. It is important to note that it was used fitness sharing instead of fitness. *Ranking Selection* that goes through all individual fitnesses and sorts/ranks them. The individual ranked number one is chosen. The last method is the *Fitness proportionate selection*, also known as roulette wheel selection. It also uses fitness sharing instead of fitness.

2.1.4 Run Function

Finally, it was defined the Sudoku Solver function. Several attributes can be modified: the initial sudoku and its fixed numbers, the population size, the valid sample, the selection method, the optimization problem (min or max), the maximum generations of the algorithm, the probability allocated for the crossover and mutation operators and whether it is used elitism or not. In case the problem finds a local optimum and cannot evolve from it after 5 generations, the algorithm proceeds to produce a generation using more disruptive genetic operators so that it doesn´t get stuck in a local optimum and the population stops evolving. This disruptive genetic operators could be a bigger mutation probability, a larger set of values to be mutated or crossovered from each individual or even a more restrict set of values to be changed (like the impossible indexes).

3 RESULTS

In order to ensure the quality of the results all selected configurations were ran several times, with a fixed set of parameters: population size: 50, crossover probability: 80, Elitism=True, max_generations=90.

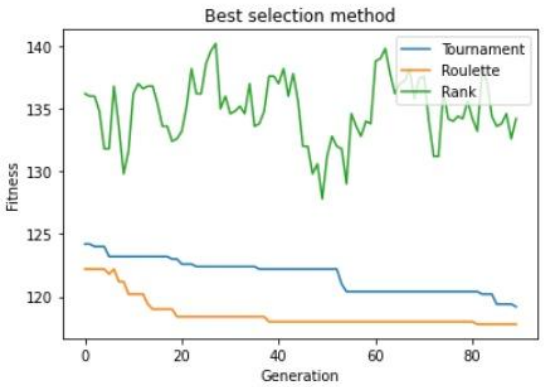

Figure 1 - Best Selection Method

The first batch of selected configurations was performed to only check which selection method was best without considering wrong and impossible numbers' functions. It is clear that both Roulette and Tournament methods better than the Rank method, having the first one a slight edge over the tournament selection. Nevertheless, this genetic algorithms were all very weak since the final fitness is still very high.

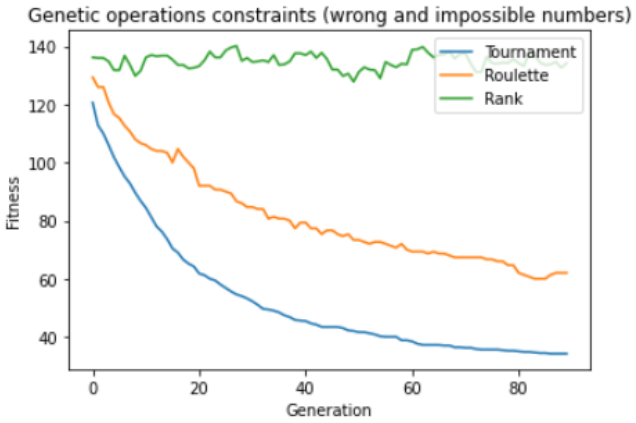

Figure 2 - Best Selection Method considering the wrong and impossible numbers

As we develop the project, we notice that the algorithm is having troubles evolving since it is constantly jumping between solutions without a straight-forward path. For that reason, we decided to implement the concept of wrong and impossible numbers which would help the genetic operators, transforming the population in a more efficient manner. Since theoretically it made sense that having into account a priori the wrong and impossible numbers of each row, column, and square, we decided to use those functions previously created and then, determine the best selection method. It was only at this point that we start to use fitness sharing for the selection methods. In fact, the results started to get better and as it is shown in figure 2 Tournament is the best selection method. Therefore, next configurations will only use tournament selection.

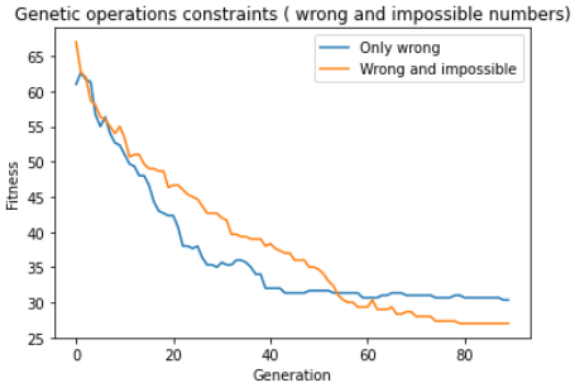

Figure 3 - Wrong numbers vs Wrong and Impossible numbers

As we were not certain whether mutate/crossover only the wrong numbers' indexes was sufficient or we should also use the impossible numbers function we decide to put that into test and concluded that even though it is a slight improvement, combining the two functions was the best. From the graphic we conclude that the using the impossible indexes, the algorithm is a bit slower but in the end it is capable of going deeper and finding a better solution. This is easily explained, since changing the impossible indexes can worsen the current fitness on the short time horizon (since those numbers were a good fit for the current solutions), but it allows the future solutions to get even better since those impossible numbers would be installed in the population and then it would not be possible to go as deep in the search for the global optimum.

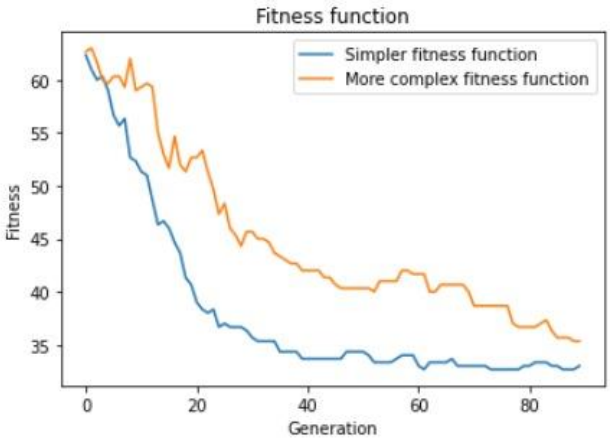

Figure 4 - Simple vs Complex fitness function

Another question raised was whether use both fitness functions or only use the first one as theoretically it made sense to use both. The "Simpler fitness function" calculates a sudoku´s fitness by only taking into account the wrong numbers while the "More complex fitness function" also takes into account the wrong numbers but also penalizes even further a sudoku if a wrong number is also an impossible number. As it shown by the graph, using a simpler fitness function produces better results.In the end, we have no doubts about the best parameters and choices to use for the final genetic algorithm: selection_method= 'tournament' (using fitness sharing), elitism=True, the fitness function is the simpler one, the genetic operator crossover and mutation work better using both the wrong and impossible indexes. Then we run this algorithm 10 times, with a population size = 125, max generations=125 to get a precise view of the performance of the algorithm. The results are displayed below.

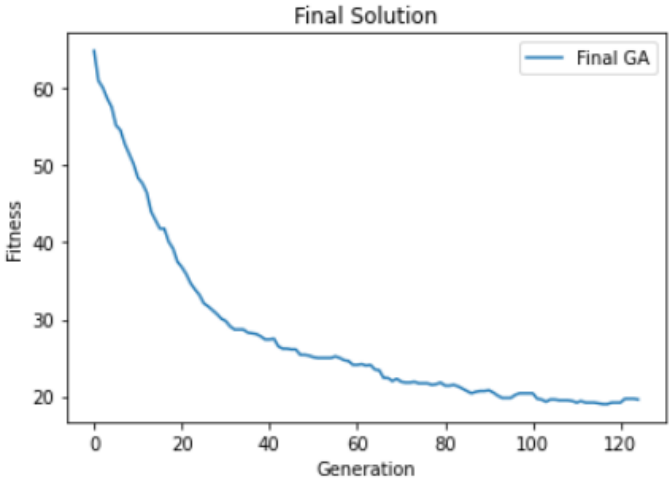

Figure 5 - Final results

As we can see the algorithm is constantly improving. In the final generations we can observe that the fitness starts to go up and down and this is due to the fact that we allow the algorithm to worsen the best fitness in an effort to make it jump from a local optimum to the global optimum. Nevertheless, we couldn´t find a global optimum, but maybe with a greater computational power it could be found using our algorithm.

4 CONCLUSION

In these project we were asked to develop a Sudoku Solver where the topics covered in the course would be applied. It was conducted several experiences, using different fitness functions, selection methods, different crossover and mutation operators which were applied using different parameters in order to adapt to the problem, to design the best genetic algorithm regarding both performance as well as computational effort and running time. We tried, explored and searched for several different approaches and made a huge effort to get the best genetic algorithm possible. And in the end, we consider that our final solution is a very efficient one since its fitness values, which were close to 20, on average, get very close to the global optimum which is 0.