NOVA IMS Information Management School

SUDOKU SOLVER

Computational Intelligence for Optimization Project

Master Degree in Data Science and Advanced Analytics

Group S

Catarina Candeias, m20200656

Filipa Alves, m20210662

Helena Oliveira, r20181121

Maria Almeida, m20210611

GITHUB: https://github.com/helenado/Sudoku_CIFO.git

INTRODUCTION

History speaks for itself: throughout the centuries, humans have encountered new challenges and solving them was part of their daily tasks. Nonetheless, not all challenges were unpredictable or undesirable. Sudoku is known as a great game for being capable of testing our abilities when it comes to problem solving and reasoning thinking. Having said that, the aim of this challenge is very simple: a 9x9 matrix is given, with some values already provided that cannot be changed, and the purpose is to fill the empty boxes with the correct values, from 1 to 9. Those values cannot be repeated either by rows, columns, or boxes, and this is what makes this game so special and interesting. In this project, although sudoku puzzles are mainly solved for leisure by human brains, Genetic Algorithms will provide an efficient and optimized way of resolving them.

REPRESENTATION

As stated previously, the matrix of a Sudoku puzzle is 9x9, and, consequently, each box within the global matrix is a 3x3 format.

Three levels of difficulty were created (*initial_puzzle.py*) to evaluate how the genetic algorithm would perform in distinct situations, extracted from the website https://sudoku.com/, where it is possible to generate distinct sudoku puzzles. Each level of difficulty is categorized as “easy”, “medium” and “hard”, according to the number of pre-defined values, as well as their positions. For the sake of simplicity, the representation created was a list with all the information regarding the Sudoku puzzle. This means that the first 9 digits indicate the first row of the main matrix, and the following 9 values represent the second row, and so on. In this way, it is possible to have the entirety of the main matrix in a simple list. The zeros indicate the empty spaces, whose objective is to be replaced with the correct value automatically through our genetic algorithm implementation.

DEFINITION OF THE FITNESS

For the fitness function, the first thing to define was if it would be a maximization or minimization problem. In this project, it was decided that a maximization problem would be used, so the fitness function was defined in a way in which rows, columns or boxes would be penalized if they had repeated values, since the goal of sudoku is to have rows, columns and boxes with only one instance of each number from 1 to 9. This was done recurring to a function that counts the unique elements by rows, columns, and boxes. Thus, for each puzzle, it is desired that every row has 9 unique values, which makes the fitness function of a “perfect” puzzle in terms of rows equal to 81 (9 lines x 9 cells per row). Following this logic for the columns and boxes, the defined fitness function reaches its maximum when equal to 243 (81 x 3 levels – rows, columns, and boxes).

SELECTION AND GENETIC OPERATORS/METHODOLOGY

Genetic operators provide the basic search mechanisms in genetic algorithms and are used to create new solutions based on existing ones in the population[1]. In this project, all the pre-defined values of the initial sudoku puzzle are kept throughout the process, which means that all the operators (*charles* folder) created afterwards had into consideration that the indexes that were filled with a number different from 0 in the initial puzzle could not be changed.

[1] Garzelli, A., Capobianco, L., & Nencini, F. (2008). Fusion of multispectral and panchromatic images as an optimisation problem. *Image Fusion*, 223–250. https://doi.org/10.1016/b978-0-12-372529-5.00005-6

1. Selection

Selection allows the algorithm to choose between solutions, so that genetic operators can then be applied. Three methods were used (*selection.py* file), where the additional implementation of the rank selection and the minimization approach on all selection methods were reinforced.

The first selection algorithm executed was *Tournament*, which aims to randomly pick *k* individuals from the population and select the best one according to their fitness. In this method, a different approach was conducted. Theoretically, it may be possible to select the same individual for the same tournament (selection with replacement). However, the results were noticeably better when making sure the same individual could not take part into the same tournament, which can be explained by a gain of diversity (of fitness values) in the set of individuals in each tournament. In fact, it does not seem intuitive to have the possibility to allow competition between the same individual, so the convergence is not completely efficient.

The second selection algorithm implemented was *Fitness Proportionate Selection*, also named *Roulette Wheel* (*fps* function), which is a method that allows for a faster convergence but sacrifices diversity in the population. Taking into consideration that the Sudoku problem has many local optima, this approach might not be the best to perform (Results and Discussion section).

Lastly, *Ranking Selection* method was applied (*rank* function), ranking individuals from worst to best, based on their fitness values. The probability of a solution being chosen is directly proportional to its position in the ranking.

2. Crossover

Crossover operators are used in genetic algorithms to combine existing solutions into new solutions, being also classified as conservation operators. In this project, seven different conservation operators were implemented (*crossover.py* file), however, only three of them were useful and indeed suitable to apply to this project, adding the fact that those three crossover operators were developed from scratch.

The first crossover applied was *In Common Crossover* (*in_common_co* function), which was created with the purpose of retaining the pre-defined values given by the initial sudoku puzzle and swapping randomly 9 commonly missing indexes between two individuals.

Afterward, a new crossover function was developed, *In Common Crossover Probability* (*in_common_prob_co* function), whose baseline is the same as the previous one, but the difference resides in the fact that, in this case, a single probability, defined by the user, establishes if the crossover is performed in each common index. The established threshold was 0.7.

Finally, the last crossover deployed was *Swap Elements Crossover* (*swap_elements_co* function), which is the most complex so far. This crossover has the unique detail of being capable of randomly selecting in which dimension it is going to operate. Basically, it decides if it wants to perform crossover on rows, columns or boxes and, afterwards, it takes two unique individuals to perform the swapping task in the corresponding dimension. In this way, the values already pre-defined are maintained in the initial sudoku. In practical terms, it is assumed that, in case the first row in one individual is swapped with the same row of another individual, for example, the numbers originally filled are not modified.

Nonetheless, although the remaining four crossovers tried (*crossover.py* file) were not deployed, it is important to highlight that our specific problem has the particularity of not allowing the modification of the original pre-defined values, as this is the essence of the game. For this reason, the use of *Single Point Crossover*, *Cycle Crossover*, *Partially Matched/Mapped Crossover* and *Arithmetic Crossover* would not have had the impact desired for the optimization of the problem. Furthermore, it is probable that by using the

Standard Crossover, for example, some good quality individuals would be destroyed due to the position problem of this crossover and since good solutions must have the initial numbers present.

3. Mutation

Mutation operator allows the algorithm to create and maintain genetic diversity, and, for this reason, it is also well-known as an innovation operator. With that being said, four innovation methods were created (*mutation.py* file). All the mutation methods developed in class are present as well, although they are not appropriate for the sudoku problem, and, therefore, were not applied. *Binary Mutation* only considers individuals defined by 0s and 1s, which clearly is not applicable to this problem. *Inversion Mutation* and the original *Swap Mutation* were not used, since they do not guarantee that original values in the initial problem are not changed. However, a personalized version of the latter was created, named *Mutation Swap*, which will be described further.

The first mutation method applied was *Mutation Sample* (*mutation_sample* function), which randomly chooses the elements of the individual to be changed, although the number of elements to modify is defined *a priori* by the user. Then, for each of the elements chosen, a random number between 1 and 9 is generated.

Secondly, the next innovation method employed was *Mutation Probability* (*mutation_prob* function), which uses a probability, provided by the user, to decide how likely it is that this mutation is applied for each missing index, being replaced by a random number between 1 and 9.

Then, *Mutation Swap* method was implemented (*mutation_swap* function), which randomly chooses in which dimension the mutation will be performed: by row, column, or box. After selecting it, a random element of the puzzle is picked (e.g., first row, third column, fifth box), and consequently, two random indexes are chosen to swap between themselves (obviously, between the ones that can be changed).

Lastly, there was the development of the *Mutation Swap All* method (*mutation_swap_all* function), which has the same logic as the method explained just before, but instead of swapping only two elements of the individual, the whole row, column, or box, is swapped, by applying a shuffling of the values located in the initial missing indexes of the individual.

RESULTS AND DISCUSSION

The aim of the present section is to describe not only the results that allowed the assessment of the performance of the algorithms adjusted and created, but also the optimal combinations of the distinct genetic operators explained previously.

At first glance, it is important to highlight the fact that each configuration was run 30 times to fairly compare different sets of configurations, due to the probabilistic nature of genetic algorithms. Also, 40 generations were performed for each run, as it seemed like a reasonable number, taking into consideration the computational effort of a higher one.

The most interesting results to be discussed are presented in the graph of Figure 1, which shows the comparison between distinct configurations considering the average best fitness value found (for each generation through the 30 runs).

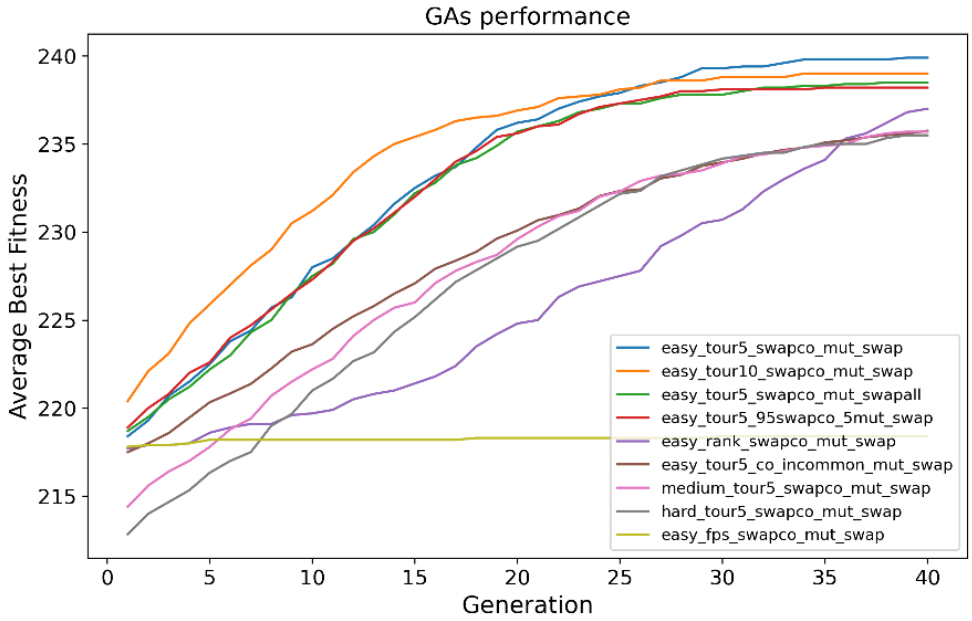

Figure 1 - Performance plot: Average best fitness for each generation of the configurations performed.

As it is possible to affirm by analysing the graph, the best configuration found was ***tour5_swapco_mut_swap***, since it is the one that reaches the greatest average best fitness value of all configurations. This configuration presents the following characteristics: *tournament size* equal to 5, *Swap Elements Crossover*, *Mutation Swap* with 0.9 probability of doing crossover and with 0.1 probability of doing mutation. It is important to note that the probability of doing crossover should be high and for mutation it should be low since this operator tends to be very destructive. Additionally, it is the one with the highest Success Rate, with 2 out of 10 runs. All the other configurations did not reach the global optimum in any of the runs, apart from *tour5_95swapco_5mut_swap*, which reached it in only one of the runs.

By evaluating each operator created and adopted as well the selection methos, the following conclusions were taken:

Regarding the comparison between crossovers, it is possible to conclude that the *co_incommon* performed worse than the *swap_co* method altered for the sudoku problem. In fact, the way the latter was defined seemed much more accurate and complex, so these results make sense.

Regarding the mutation operators, in particular the comparison between *Mutation Swap* and *Mutation Swap All*, respectively, *tour5_swapco_mut_swap (blue line) and tour5_swapco_mut_swapall (green line)*, it is possible to assess that during the first 20 generations, the results were fairly similar; nonetheless, as it gets closer to the final generation, a bigger difference was observed. This can be justified by the fact that *swapall* is more destructive than *swap*. The other mutation methods developed were also tried, however no satisfactory results were found.

In respect to the selection operators, more specifically *Tournament*, the selection pressure was established based on its size, 5 and 10. The results were similar, however, the configuration using a tournament size equal to 5 gave better results. Thus, we can conclude that although a higher selection pressure tends to increment the probability of the best individuals to survive, it is not giving the best outcomes, since the population should also be constituted by individuals with lower fitness, incrementing diversity. Furthermore, the higher the tournament size, the more it would take to run, as it needs to compute more fitness values. For *Fitness Proportionate Selection* (*fps*) and *Ranking Selection* (*rank*), the run time was indubitably higher than with *Tournament*. In fact, this can be explained by the fact that the first two referred selection methods analyse every individual's fitness in

order to select an individual, contrarily to *Tournament*, which only evaluates *k* individuals (Table 1). Therefore, *fps* and *rank* were performed only once each, resulting in worse outcomes, especially *fps,* as it has the disadvantage of loss of diversity, even though its convergence is faster than the others, reaching a best average fitness value of 220, a poor result.

Generations	Tournament	Rank	FPS
1	3.28	10.86	17.03
2	4.39	11.48	18.99
3	5.43	12.84	21.11
4	7.77	13.67	24.18
5	8.79	14.72	26.38

Table 1 - Time in seconds for the first 5 generations of the three selection methods (the full data can be visualized on the csv's files)

Thereafter, with the urge to improve the results on the *easy* problem, the probability of doing crossover was increased to 0.95 and the probability of performing mutation was decreased to 0.05, identified in *tour5_95swapco_5mut_swap* configuration. Nonetheless, it was not successful, as it did not perform better than the defined *tour5_swapco_mut_swap* configuration.

Finally, the best configuration encountered for the *easy* problem, *tour5_swapco_mut_swap,* was applied to *medium* and *hard* difficulties. Although the results decreased slightly, they were still relatively good, with an average best fitness value of 237. If more computational resources were available, different configurations would have been tried similarly to what was done for the *easy* difficulty.

Another aspect to consider when performing a genetic algorithm is the inclusion of elitism. The conclusion obtained was that when elitism was included, the best individual encountered in each generation was always better from generation to generation, which did not happen when elitism was not used.

CONCLUSION

This project was able to demonstrate how genetic algorithms can have a great influence in solving sudoku puzzles in an efficient and optimized way. The operators used and the combinations done returned satisfactory results. Still, there is room for improvement, such as the implementation of fitness sharing, in order to maintain diversity, and by trying different combinations of inputs, to find an even better fitness value than the one we obtained. Additionally, with the availability of more computational resources, it could be possible to experiment running the algorithms with more generations and maybe a larger population size, which can have an influence on the results *a posteriori*. Lastly, new experiments using other kind of puzzles, perhaps bigger ones, could be an interesting challenge to follow after this project.

WORK DIVISION

The entire project was done by all authors jointly, thus the efforts were equally distributed.

REFERENCES

[1] Garzelli, A., Capobianco, L., & Nencini, F. (2008). Fusion of multispectral and panchromatic images as an optimisation problem. *Image Fusion*, 223–250. https://doi.org/10.1016/b978-0-12-372529-5.00005-6

[2] *Play Free Sudoku online - solve web sudoku puzzles*. (n.d.). Sudoku.Com. https://sudoku.com/

[3] Vanneschi, L. (n.d.). *Computational Intelligence for Optimization*.