COMPUTATIONAL INTELLIGENCE FOR OPTIMIZATION

USING GENETIC ALGORITHMS FOR THE RESOLUTION OF SUDOKU PUZZLES

António Cymbron (r20181059), **Duarte Redinha** (r20181066), **Maria João Marques** (r20181119)
MASTER DEGREE PROGRAM IN DATA SCIENCE AND ADVANCED ANALYTICS, WITH A SPECIALIZATION IN DATA SCIENCE
NOVA INFORMATION MANAGEMENT SCHOOL

I. CONTEXT & INTRODUCTION:

In this project, we will use genetic algorithms to solve Sudoku puzzles completely.

To carry out our project, we created 6 files. In the first one - *puzzles.py* -, puzzles were imported into a Python file as lists of 81 elements, where elements that are not given at the beginning of the puzzle are imported as 0s. It is also in this same Python file that we choose which of the 4 puzzles that we have at our disposal (1 easy, 1 medium, 1 hard and 1 very hard) we wanted to solve. This list of 81 elements was then transformed into a 9 by 9 NumPy array (where each of the 9 arrays represented each of the 9 lines of the puzzle), this being the representation of our original puzzle and also the individuals that we later created. We chose this form of representation because we believe that it will be useful not only to be able to generate valid lines when creating the individuals, but also for when we perform the operations of *Crossover* (where we will select entire lines from both parents and perform the crossover between them) and *Mutation* (where we will select entire lines in each of the offspring and carry out the mutation within those lines). The second file - *sudoku.py* -, was where we solved the problem and defined the parameters that we used in the resolution. The third one - *charles_file.py* -, was where we created all the classes (*Population*, *Individual* and *Original*) and functions that we needed. The fourth - *selection.py* -, was where we implemented 3 *Selection* algorithms (*fitness proportionate selection*; *tournament* and *ranking*). The fifth - *crossover.py* -, was where we implemented 2 types of C*rossover* (*cycle* and *partially matched*). Finally, in the sixth one - *mutation.py* -, was where we implemented 2 types of *Mutation* (*swap* and *inversion*). All the above-mentioned processes will be explored and explained further in this report.

The division of the labour in our group was made as fairly as possible for every member, in a way that research was carried out by all, as well as code construction, tests to the solution in hands, as well as the creation of the document in-hands. The code repository for this project can be found at https://github.com/mariammarques/GAsSudoku.

II. METHODOLOGY:

Next, we created a Population with size *N*. The Population class received the size of the population we wanted to create, the original puzzle, if we are dealing with a maximization or a minimization problem. As we are going to mention later on, when talking about the fitness function, we tackled this problem as being a maximization problem. However, minimization was also implemented, with the aim of making the code as abstract as possible. Additionally, this class in hands also received the Tournament size, which is a variable that would only be used if the *Selection* method chosen is the Tournament, naturally. It is important to note that if the Tournament size is not specified, the default value that will be used is 0.20, which is a value that, through the research we carried out, was considered adequate.

When we created a new Population, the first step we took was to assess what are the legal values that each cell of the puzzle can receive. So, for each of the 81 positions available on the *Sudoku*, we kept a list of possible values to be entered in that cell. If we were dealing with one of the cells given to us at the beginning of the puzzle, that same value would be the only legal value for the cell in question; in the remaining cells, the possible values were the values that are not repeated in the same row, column and grid of the same cell. After we defined the legal values for each position,

we created the *N* individuals from the Population 1 by 1. This way, we iterated through the 81 cells, randomly selecting for each one a value among the legal values for that same position (in the case of a cell filled in the original puzzle, the only legal value is the value that was in that same cell, right from the start). In addition to this entire process, when creating our individuals, we would always ensure that the individual's lines are legal (i.e., they do not have repeated numbers). Finally, when we had the *N* individuals of the Population created, we calculated the fitness of each one of them.

Regarding the fitness function, we thought of different solutions for it. Thus, possible implementations for the fitness function would be: the count of rows, columns and unique grids (which can then sum, multiply, or take the average of the 3 values - this would be a maximization problem); the count of non-unique rows, columns and grids (that would be able to perform the same 3 operations – however, this case would be a minimization problem); or a combination of the two previous ideas, where we could, for example, divide the first by the second one, adding to the second a constant, to avoid the case where we would be dividing by 0 (cases where we had found the solution). With all this in mind, we defined the fitness function as the product of the variables *column_sum* and *grid_sum*. The first step is to iterate through the 9 lines and, within each, we would count how many times we had found each of the numbers from 1 to 9. The ideal scenario would be the one where we find each number only once, as we would thus have all 9 numbers in the line and we would not have any repeated numbers. The variable *row_sum*, that starts this process as 0, is calculated as the sum of itself with $\frac{1}{\frac{len(set(row\_count))}{9}}$. So, in the ideal scenario presented above, the $set(row\_count)$ would be {1} and, in each iteration through the rows, we would add 0.1111111111111111 to the *row_sum*. In this way, at the end of the 9 rows, we would have that *row_sum* would be equal to 1. This reasoning is then repeated for the columns and for the grids. At the end of this process, the fitness of each individual is calculated. If the *row_sum*, *column_sum* and *grid_sum* are 1, the fitness of the individual will also be 1, and we will have found our solution. Otherwise, fitness would then be the product of *column_sum* and *grid_sum*, due to the fact that, in the creation of the individuals, we forced the solutions to not have duplicate values in the row, so that the *row_sum* was always 1.

Bearing this in mind, it is clear that we were faced with a maximization problem, since the better the solution, the greater the value of its fitness. Once we found an individual with a fitness of 1, we could finally say that we had found the solution to the puzzle in question.

After we had our original Population, we resorted to the *Evolve* function, which receives the number of generations, the parents' select algorithm, the elements related to the *Crossover* and *Mutation* (the operator of each one and the probabilities of both performing *Crossover* and of carrying out *Mutation*), and whether we wanted to carry out *Elitism* or not (and if so, of what kind).

In our project, we implemented 3 types of *Elitism*: simple, of percentage *k*%, and of size *K*. In simple *Elitism*, the individual with the best fitness of the previous Population is passed to the new one, and the individual with the worst fitness is deleted from it. In *k*% percentage *Elitism*, we are given a number between 0 and 1, and we pass the *k*% best individuals from the previous Population to the new one, and we eliminate the *k*% worst individuals from the new Population. Finally, in *K*-size *Elitism*, we receive a *K* – being *K* a number greater than 1 and less than the Population size -, and we pass the *K* best individuals from the previous Population to the new one, eliminating the *K* worst individuals from the new Population.

We then use the select algorithm that was passed in the *Evolve* function, to find the 2 parents. As mentioned before, we implemented 3 selection algorithms, both for maximization and minimization. After we found the 2 parents, we generated a random number between 0 and 1. If this same number is smaller than the probability of performing *Crossover*, we will perform the type of *Crossover* chosen between the 2 parents. For the 2 types of *Crossover* that we implemented, it is

carried out between lines (we randomly chose *m* lines from each of the parents, and performed *Crossover* between them. So, for example, the 3rd line of the 1st and 2nd offsprings will be the result of the *Crossover* chosen between the 3rd line of the parent 1 and the 3rd line of parent 2). Then, we generated once more a random number between 0 and 1, and if it was smaller than the probability of mutating, we would carry out the chosen type of *Mutation* on the 1st offspring. This process will be carried out once more for the 2nd offspring. For the 2 types of *Mutation* that we implemented, they are carried out considering the lines of the puzzle, meaning that we randomly choose *m* lines of the offspring, and carried out the *Mutation* within each of them. After this step, we appended the new Population of the 1st offspring and, if we still have space, of the 2nd offspring. When we finally have the new Population with the same size *N* that we defined at the beginning, we would calculate the fitness of the individuals of the new Population. Afterwards, we applied the chosen type of *Elitism* (or we don't apply it at all, if the user desires it so). At the end of each generation, we always printed the individual with the best fitness.

In order not to get stuck in a solution, we created a condition that checked if, after verifying that in *z* times – being *z* 10% of the number of generations previously defined -, the fitness of the best individual of the generation had not improved, the whole process would be restarted, reinitializing the original Population. Also, in the cases where a solution was found, the algorithm stops, and the solution is returned. In order to be able to generate useful graphs in order to easily draw conclusions, as the ones presented later in this document, we will always keep the fitness of the best individual of each generation, even when we reset the Population.

III. CONFIGURATION TESTS & DISCUSSION:

We started by trying to solve the easy puzzle, with different combinations of Selection algorithms and *Crossover* and *Mutation* Operators, always with a Population of 50 individuals, for 100 generations, with a *Crossover* probability of 0.9, *Mutation* probability of 0.15 and Simple *Elitism*. However, we quickly realized that due to the fact that in this puzzle many values are already filled in by origin, and also due to the way we initialize the individuals, seeing what the legal values for each cell are, the solution to the puzzle was always found with the 1st Population. So, after running 10 times each of the 12 possible combinations (*Selection* algorithm + *Crossover* type + *Mutation* type), we realized that we would be spoiling precious time if we kept running more times for each combination. Thus, through the easy puzzle, it was not possible for us to draw in-depth conclusions about the functioning of our program and the impact on the search for a solution of the various parameters that constitute the Genetic Algorithm. Therefore, we proceed to the puzzles of greater difficulty, in order to try to extract valuable insights. This way, more in-depth conclusions and comparisons will not be presented for the easy puzzle.

With this in mind, we started to try to solve the medium puzzle, with different combinations of *Selection*, *Crossover* and *Mutation* methods, always with a Population of 100 individuals, for 200 generations, with a *Crossover* probability of 0.9, *Mutation* probability of 0.15 and Simple *Elitism*. For each combination, in order to have statistically more robust and reliable results, we ran 50 times. Firstly, we could clearly see that the *Tournament* Selection is the worst of the 3 *Selection* algorithms we tested, presenting longer times and needing more generations to find a solution. When it comes to *Crossover*, *Partially Matched Crossover* clearly presents better results, and the same happens with *Inversion*, for the *Mutation* operator. So, to see the effect of *Elitism* when solving the puzzle, we decided to test the most promising combinations - 4 and 12, which can be seen in Table 1, with other types of *Elitism*, and even without *Elitism*. We also tested the 7th combination with a *Tournament* size of 0.5 and no *Elitism*, since according to the Prof. Leonardo Vanneschi's booklet, “if no elitism is used, the choice of tournament selection may allow us to avoid evaluating the fitness of all

individuals in the population at each generation". Therefore, it is expected that the algorithm will be faster. However, as we increased the *Tournament* size, to increase the probability of the best individuals surviving, the time gains may not be as considerable.

Table 1 *– Combination Tests for Finetuning the Sudoku Solver Genetic Algorithm*

# Combination	Puzzle	Pop_Size	Gens	Co_P	Mu_P	Elitism	Selection	Crossover	Mutation	Avg_Time	Avg_Gen
1	Medium	100	200	0,9	0,15	1	FPS	Cycle	Swap	27,72	100,1
2	Medium	100	200	0,9	0,15	1	FPS	Cycle	Inversion	38,43	79,9
3	Medium	100	200	0,9	0,15	1	FPS	PMX	Swap	14,45	49,7
4	Medium	100	200	0,9	0,15	1	FPS	PMX	Inversion	11,58	37,5
5	Medium	100	200	0,9	0,15	1	Tournament	Cycle	Swap	41,69	158,8
6	Medium	100	200	0,9	0,15	1	Tournament	Cycle	Inversion	69,15	274,9
7	Medium	100	200	0,9	0,15	1	Tournament	PMX	Swap	24,06	91,9
8	Medium	100	200	0,9	0,15	1	Tournament	PMX	Inversion	28,87	99,3
9	Medium	100	200	0,9	0,15	1	Ranking	Cycle	Swap	28,8	110
10	Medium	100	200	0,9	0,15	1	Ranking	Cycle	Inversion	16,69	60
11	Medium	100	200	0,9	0,15	1	Ranking	PMX	Swap	13,4	44,8
12	Medium	100	200	0,9	0,15	1	Ranking	PMX	Inversion	8,4	19,7
13	Medium	100	200	0,9	0,15	0,1	FPS	PMX	Inversion	16,25	29,8
14	Medium	100	200	0,9	0,15	5	FPS	PMX	Inversion	14,84	58,5
15	Medium	100	200	0,9	0,15	-	FPS	PMX	Inversion	16,82	81,5
16	Medium	100	200	0,9	0,15	0,1	Ranking	PMX	Inversion	7,71	18,8
17	Medium	100	200	0,9	0,15	5	Ranking	PMX	Inversion	9,79	31,6
18	Medium	100	200	0,9	0,15	-	Ranking	PMX	Inversion	7,6	21,6
19	Medium	100	200	0,9	0,15	-	Tournament (t_size = 0.5)	PMX	Swap	47,57	195,9
20	Medium	100	200	0,8	0,15	0,1	FPS	PMX	Inversion	13,09	48,2
21	Medium	100	200	0,9	0,1	0,1	FPS	PMX	Inversion	11,39	40,2
22	Medium	100	200	0,8	0,15	0,1	Ranking	PMX	Inversion	9,16	29,2
23	**Medium**	**100**	**200**	**0,9**	**0,1**	**0,1**	**Ranking**	**PMX**	**Inversion**	**5,84**	**11,1**
24	Hard	250	500	0,9	0,15	1	FPS	Cycle	Swap	18,64	97,7
25	Hard	250	500	0,9	0,15	1	FPS	Cycle	Inversion	16,92	84,6
26	Hard	250	500	0,9	0,15	1	FPS	PMX	Swap	13,98	76,8
27	Hard	250	500	0,9	0,15	1	FPS	PMX	Inversion	15,98	92,8
28	Hard	250	500	0,9	0,15	1	Tournament	Cycle	Swap	21,64	104,7
29	Hard	250	500	0,9	0,15	1	Tournament	Cycle	Inversion	61,05	300,8
30	Hard	250	500	0,9	0,15	1	Tournament	PMX	Swap	14,99	89,4
31	Hard	250	500	0,9	0,15	1	Tournament	PMX	Inversion	28,37	144,8
32	Hard	250	500	0,9	0,15	1	Ranking	Cycle	Swap	19,2	60
33	Hard	250	500	0,9	0,15	1	Ranking	Cycle	Inversion	23,9	90,4
34	Hard	250	500	0,9	0,15	1	Ranking	PMX	Swap	14,7	54,8
35	Hard	250	500	0,9	0,15	1	Ranking	PMX	Inversion	18,38	64,8
36	Hard	250	500	0,9	0,15	0,1	FPS	PMX	Swap	18,6	79,6
37	Hard	250	500	0,9	0,15	5	FPS	PMX	Swap	22,39	92,8
38	Hard	250	500	0,9	0,15	-	FPS	PMX	Swap	65,24	289,3
39	Hard	250	500	0,9	0,15	0,1	Ranking	PMX	Swap	13,57	56,6
40	Hard	250	500	0,9	0,15	5	Ranking	PMX	Swap	8,61	22,2
41	Hard	250	500	0,9	0,15	-	Ranking	PMX	Swap	12,41	57,7
42	**Hard**	**250**	**500**	**0,9**	**0,15**	**0,1**	**Ranking**	**PMX**	**Inversion**	**14,89**	**15,5**
43	Hard	250	500	0,9	0,15	5	Ranking	PMX	Inversion	15,43	61
44	Hard	250	500	0,9	0,15	-	Ranking	PMX	Inversion	20,16	80,9
45	Hard	250	500	0,8	0,15	0,1	FPS	PMX	Swap	11,16	55,3
46	Hard	250	500	0,8	0,1	0,1	FPS	PMX	Swap	14,38	57,3
47	Hard	250	500	0,8	0,15	5	Ranking	PMX	Swap	14,53	48,9
48	Hard	250	500	0,9	0,1	5	Ranking	PMX	Swap	16,61	53,2
49	Hard	250	500	0,8	0,15	0,1	Ranking	PMX	Inversion	9,23	33,4
50	Hard	250	500	0,9	0,1	0,1	Ranking	PMX	Inversion	10,57	36,7
51	**Very Hard**	**500**	**1000**	**0,9**	**0,15**	**0,1**	**Ranking**	**PMX**	**Inversion**	**740,39**	**1313,11**
52	Very Hard	500	1000	0,9	0,15	5	Ranking	PMX	Swap	1010,96	1981,83

Regarding *Elitism*, we found that the one in which we used the best 10% of individuals showed excellent results, requiring fewer generations to reach a solution. We then tested combinations 13 and 16 with a *Crossover* probability of 0.8, instead of 0.9, to see the impact of this percentage when solving the problem, concluding that 0.9 was better. Thus, we tested combinations 13 and 16 with a *Mutation* probability of 0.1, to see the impact of this percentage when solving the problem. At the end of all these tests, we concluded that the best possible combination for the medium difficulty puzzle is the *Ranking Selection*, with *Partially Matched Crossover* and *Inversion*,

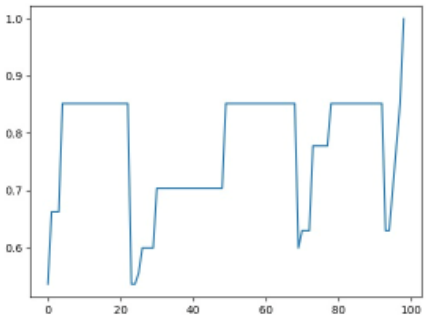
with 10% *Elitism*, *Crossover* probability of 0.9 and *Mutation* probability of 0.1, since it is the parameterization that requires less time and generations to find the solution to the puzzle.

As for the hard puzzle, we performed the same iterative method to draw conclusions. However, due to the puzzle's increasing difficulty, we have increased the Population size to 250 and the number of generations to 500, in order to explore a wider range of possible solutions. For each combination, in order to have statistically more robust and reliable results, we ran 50 times. As with the medium puzzle, we observed that *Tournament* Selection is the least efficient in finding the solution to the puzzle. As for the *Crossover*, just like with the medium puzzle, the *Partially Matched Crossover* shows a better performance. However, the big difference lies in the fact that, for the puzzle of this difficulty, the *Swap Mutation* presents a better performance, both in terms of time and in terms of the number of generations needed to find the solution to the puzzle. Then, to understand the effect of *Elitism* in solving this more difficult puzzle, we decided to test the most promising combinations – 26, 34 and 35, with other types of *Elitism* and even without it. Thus, regarding this parameter, as had happened with the medium difficulty puzzle, the *Elitism* in which we used 10% of the best individuals showed better results. However, it is important to note that, when using the *Ranking* Selection combination, with *Partially Matched Crossover* and *Swap Mutation*, using *Elitism* with the 5 best individuals showed more satisfactory results. We then tested parameterizations 36, 40 and 42 with a *Crossover* probability of 0.8, instead of 0.9, to understand the impact of this percentage when solving the problem, having concluded that in most cases, as in the medium puzzle, 0.9 was better. Finally, we tested combinations 45, 40 and 42 with a *Mutation* probability of 0.1, having concluded that, unlike the medium difficulty puzzle, in this case 0.15 was better. After all these tests, we concluded that the best possible combination for the hard difficulty puzzle is *Ranking* Selection, with *Partially Matched Crossover* and *Inversion Mutation*, with 10% *Elitism*, 0.9 *Crossover* probability and 0.15 *Mutation* probability (being this the only parameter that differs from the best solution for the medium puzzle). We could also have assumed that the best solution was 40, but even though it takes less time to find a solution, it needs more generations to find it.

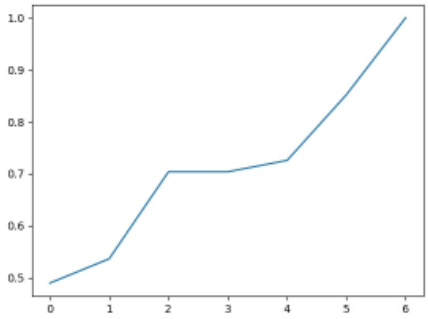
Bearing in mind the conclusions obtained so far and knowing that for the very hard puzzle the search for a solution would take much longer and the computational resources could not be sufficient, we chose not to resort to the same iterative process that we had used until then. Thus, we extrapolated to the resolution of this puzzle the conclusions that we have drawn so far from the less difficult puzzles. Thus, in order to obtain statistically more robust and reliable results, we ran 2 combinations of parameters 25 times (we reduced this number, as we were aware that the search for a solution would be a very time-consuming process in a limited time to develop this *Sudoku* Solver), for a population of size 500 and for 1000 generations. The 2 combinations of parameters considered for the tests were based on the conclusions obtained so far, and were *Ranking* Selection, with *Partially Matched Crossover* and *Inversion Mutation*, with a *Crossover* probability of 0.9, a *Mutation* probability of 0.15 and *Elitism* of 10%; and *Ranking* Selection, with *Partially Matched Crossover* and *Swap Mutation*, with a *Crossover* probability of 0.9, *Mutation* probability of 0.15 and *Elitism* using the 5 best individuals.

IV. CONCLUSION:

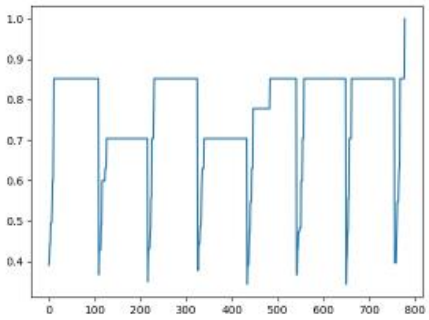
Below, you can see the graphs that show the evolution of the fitness of the best individual of each generation, over the generations, for the medium, hard and very hard puzzles (using the configurations 23, 42 and 51, from Table 1, to solve each puzzle, respectively).



Graph 1 *– Fitness Evolution Throughout Generations for a Medium Difficulty Sudoku Puzzle*



Graph 2 *– Fitness Evolution Throughout Generations for a Hard Difficulty Sudoku Puzzle*



Graph 3 *– Fitness Evolution Throughout Generations for a Very Hard Difficulty Sudoku Puzzle*

After successfully solving all the puzzles, of different difficulties, we can confidently say that we obtained good results for the project we chose and that the use of Genetic Algorithms for solving *Sudoku* puzzles is a choice that presents acceptable results. However, we believe that it would be important, with more time and resources, an even deeper exploration of different combinations of *Selection*, *Crossover* and *Mutation* types, and also a more robust statistical validation, increasing the number of times each combination is tested. Thus, it is possible that the speed of convergence of the genetic algorithm can be improved, by exploring more deeply and for a longer time possible combinations of parameters of the genetic algorithm, which may also lead to more difficult puzzles (or even *Sudoku* puzzles larger than normal size, with 81 cells) to be solved by the algorithm. Based on all the analysis and conclusions obtained so far, we can say that different operators affect the convergence of the Genetic Algorithm, with *Crossover* and *Mutation* types that stand out in terms of performance, as previously explained. The inclusion of *Elitism* positively impacts the performance of the Genetic Algorithm, and, in general, when we resorted to *Elitism* with the best 10% of individuals, the results were excellent.

V. REFERENCES:

Vanneschi, L. (n.d.). Computational Intelligence for Optimization. In Lectures in Intelligent Systems. Springer.

D, D. (2019, November 9). Solving Sudoku puzzles with Genetic Algorithm. Road to ML. https://nidragedd.github.io/sudoku-genetics/.

Deng, X. Q., & Li, Y. D. (2011). A novel hybrid genetic algorithm for solving Sudoku puzzles. Optimization Letters, 7(2), 241–257. https://doi.org/10.1007/s11590-011-0413-0.

Jacobs, C. T. (2022, March 23). Overview. GitHub. https://github.com/ctjacobs/sudoku-genetic-algorithm.

Weiss, J. (2009). Genetic Algorithms and Sudoku. http://micsymposium.org/mics_2009_proceedings/mics2009_submission_66.pdf.