NOVA IMS Information Management School

OPTIMIZATION FOR COMPUTATIONAL INTELLIGENCE

Sudoku Solver using Genetic Algorithms

Spring Semester 2021/2022

Ana Luís, 20210671 | Carolina Machado, 20210676

Francisco Calha, 20210673 | Sara Arana, 20210672

Link for GitHub repository: https://github.com/carolinamagee/SudokuSolver.git

1. Introduction

This project aims at creating a Sudoku solver using Genetic Algorithms, and it was developed throughout the course of Optimization for Computational intelligence. Our goal is to further extend the Charles Library developed in class, so that it can applied to Sudoku problems, and hopefully solve them completely. We will explain in detail all implementations made and the reasoning behind them. Namely, the representation chosen, the fitness function and type of problem (minimization or maximization), the selection, crossover and mutation methods applied, and how we got to our best model.

2. Problem Definition

A Sudoku consists in a 9x9 matrix, where each cell corresponds to a number from 1 to 9. It is divided in rows, columns, and 3x3 grids, in which each of them there can be no repetition of numbers. So, in a Sudoku solution, we have a total of 81 numbers, where each number from 1 to 9 appears exactly 9 times. In addition, an initial Sudoku solution has already some cells filled with some numbers of the final solution (the number of cells filled depends on the problem's difficulty), and we decided to take into consideration these fixed values.

In our library we have 4 different initial solutions, each one accounting for a different level of difficulty: very easy, easy, intermediate, and hard. For testing and parameter tuning purposes, we used only the easy solution. In the end, when the best model is defined, we will run it for all difficulty levels and see if their behavior is similar.

3. Representation

The Representation we chose for the individuals consists of a list with 9 lists, each containing 9 numbers: essentially, a list with 81 numbers. Notice that, each list inside the list represents one row in the sudoku 9x9 matrix.

Although we are filling the 0 values (empty) in the initial solution by a random number between 1 and 9, we have two constraints in our representation. First, since the Sudoku's initial solution comes with some fixed values, we kept them in their right positions, to facilitate the learning process. We defined the possible values by going to each grid in the initial solution and seeing if the number is different than 0, if so, we remove that number for the list of possible values to add in each cell. The second constraint is the following: by grid, i.e., by each 3x3 matrix, there can be no duplicate values. By doing so, our model will save time in dealing with duplicate values on grids and can focus more on the duplicates in the rows and columns.

4. Fitness

To choose our fitness function, we decided to work with a minimization problem, in which we counted the duplicate values in all rows and columns. Since our crossover methods were not altered, but we still had the initial fixed values in the initial solutions, we decided to penalize the individual by increasing its fitness whenever a value was different from the pre-existing fixed values. We multiplied each value that is different from the fixed value in the initial solution by 10, that is, if the fixed value is the same in the individual, do nothing, otherwise add a penalization of 10. If this sanction of multiplying the fixed value by 10 was not implemented, what could have happened would be cases where the final solution complied with no repetitions in rows, columns, and grids, however, it was not the actual solution of the initial solution. Notice that we did not account for the repetitions in each 3x3 matrix, because of the constraint we added to our representation (it is built in such a way that each grid only has unique values).

From here, we tried two different fitness functions, one where we added all components mentioned previously[1], and another where we multiplied them[2].

[1] **Fitness = duplicates by row + duplicates by column + 10 * fixed value**

[2] **Fitness = (duplicates by row) * (duplicates by column) * (10 * fixed value)**

In the fitness function (1), the fitness of the global optimum is 0. Notice that for fitness (2), the values were only multiplied for cases where they were higher than 0 (otherwise the fitness would be 0, even in cases where duplicate numbers existed). In addition, the fitness function (2), the fitness of the global optimum would be 1, however we added a constraint for the cases where there were no duplicate values in neither the rows and columns, and the fixed values were in their corresponding positions. For the cases where this happens, the fitness of (2) would also be 0. We did this, because in cases where there was only 1 duplicate (in either a column or a row), 2 duplicates (1 in column and 1 in row), and no duplicates, the fitness value would be the same – 1. By adding this constraint, we are able to determine more easily when the best individual is able to obtain the correct solution of the sudoku.

To see which fitness function obtains better results, we decided to run the algorithm, for both functions, 10 times for each selection method (since selection considers the fitness values). Afterwards, we proceed to compute the average to see which one is better. Notice that, since the algorithm is very slow for both Ranking and Fitness Proportionate selection methods, we are running them for 200 individuals and 400 generation; later, we will increase both parameters. We concluded that the average fitness value, running the algorithm 10 times for each selection method (30 times total), for fitness (1) is 7.5 and for fitness (2) is approximately 7.57. However, these fitness functions behave differently, so we decided to also keep track of the duplicate values. The average number of duplicates for fitness (1) is 7.5 and for fitness (2) is approximately 6.87. From the obtained results, we decided to choose the fitness function (2) for our problem, since it led to less duplicates, on average.

One pattern of the results obtained using function (2) is that is, in almost all results, the total number of duplicates is in either the columns or the rows. This happens because having duplicates in both leads to a greater fitness value, and it also might be reason for why the results were slightly better using this approach.

5. Selection Methods

Three selection methods were explored: **Ranking Selection**, **Fitness Proportionate Selection**, and **Tournament Selection**. Since the Selection Methods are not problem dependent, we did not alter the algorithms done in the Charles Library. Nevertheless, all methods were adapted to both minimization and maximization problems.

For the Ranking Selection method, our model was taking a lot of time to sort the individuals according to their fitness values. So, to account for that problem, we implemented a quick sort algorithm in our selection script, and it proved to increase a bit the speed of the algorithm. However, both Fitness Proportionate selection and Ranking selection were still much slower than Tournament. In addition, in the Fitness Proportionate selection algorithm, we obtained better results using the tournament size at 3, rather than the default value 5.

6. Crossover Methods

Two crossover methods were explored: **Cycle Crossover** and **Partially Mapped Crossover**. For the Crossover methods, we decided to apply it per grid, i.e., by each 3x3 matrix composed by only unique values from 1 to 9. For both methods applied, we transformed the initial representation into a list of 9 lists, each of the 9 lists corresponding to a grid. For example:

Initial Parent 1:

[[1, 4, 8,1, 5, 9,4 ,5 ,6],
[5, 2, 9,3, 2, 7,2, 7, 3],
[7, 6, 3,4, 8, 6,8, 1, 9],
[9, 1, 5,1, 3, 5,5, 6, 8],
[7, 8, 3,4, 7, 9,2, 1, 7],
[4, 6, 2,8, 6, 2,3, 9, 4],
[2, 7, 6,1, 4, 8,5, 3, 1],
[3, 8, 9,2, 3, 5,6, 2, 9],
[5, 1, 4,6, 9, 7,8, 4, 7]]

Initial Parent 2:

[[1, 6, 8,1, 2, 3,4, 1, 6],
[4, 2, 5,4, 5, 6,2, 7, 3],
[9, 7, 3,7, 8, 9,5, 8, 9],
[9, 1, 3,4, 3, 5,1, 2, 3],
[5, 7, 6,1, 9, 2,5, 7, 4],
[4, 8, 2,8, 6, 7,8, 9, 6],
[7, 2, 6,3, 4, 8,5, 7, 1],
[3, 8, 9,2, 5, 6,6, 2, 9],
[5, 1, 4,9, 1, 7,8, 3, 4]]

Accessing the 9 list values to the grid's values:

Parent 1: [[1,4,8,5,2,9,7,6,3], [1,5,9,3,2,7,4,8,6], [4,5,6,2,7,3,8,1,9], [9,1,5,7,8,3,4,6,2], [1,3,5,4,7,9,8,6,2], [5,6,8,1,7,2,3,9,4], [2,7,6,3,8,9,5,1,4], [1,4,8,2,3,5,6,9,7], [5,3,1,6,2,9,8,4,7]]

Parent 2: [[1,6,8,4,2,5,9,7,3], [1,2,3,4,5,6,7,8,9], [4,1,6,2,7,3,5,8,9], [9,1,3,5,7,6,4,8,2], [4,3,5,1,9,2,8,6,7], [1,2,3,5,7,4,8,9,6], [7,2,6,3,8,9,5,1,4], [3,4,8,2,5,6,9,1,7], [5,7,1,6,2,9,8,3,4]]

From here, each crossover operation was done for each list in the same position:

Offspring 1: [[new grid from position 1], [new grid from position 2], [new grid from position 3], [new grid from position 4],

[new grid from position 5], [new grid from position 6], [new grid from position 7], [new grid from position 8], [new grid from position 9]]

Offspring 2: [[new grid from position 1], [new grid from position 2], [new grid from position 3], [new grid from position 4],

[new grid from position 5], [new grid from position 6], [new grid from position 7], [new grid from position 8], [new grid from position 9]]

Afterwards, the solutions were transformed into their initial format, where each list corresponded to a row (the inverse process is done). By using this approach, we are not dealing with repetitions in each grid, so the algorithms do not need to be altered. In addition, the defined fixed values, from the initial solution, were not taken into consideration when applying these methods – we allowed them individuals to not respect the possible values in each position. However, having a number in a certain position different than the one existing in the initial solution – substituting a fixed value by another –, is heavily penalized in the fitness value of that individual. As such, and because we used Elitism (which will be explained later on), the fixed values' problem was not something we had to worry about in the end, by looking at the results obtained.

After running the algorithm several times, the only alteration made regarding the two crossover methods used, was the one related to the Partially Matched/Mapped crossover. This method was not obtaining very good results, when comparing them to the ones obtained using Cycle crossover. We figured it would be because when the segment chosen was too small, we were switching only one or two numbers, hence, there was not enough diversity for the model to learn. As such, we defined the segment's length to always be 4, and the results improved significantly.

In addition, we did not apply the Single Point Crossover method in our problem because for cases where the position of the numbers matter, this method tends to destroy good individuals. Nevertheless, the algorithm was modified for our problem representation and we even ran the model a few times using this method and the results were not favorable.

7. Mutation Methods

Two mutation methods were explored: **Swap Mutation** and **Inverse Mutation**.

The application of both methods faced the same reasoning as the crossover methods (i.e., rows were switched to 3x3 matrices, where there were no repetitions of numbers, and then switched back to rows). However, we have 9 possible grids where the mutation operations can occur. Due to mutation being related to disruption of individuals, and we want less of it, we decided to randomly select one grid and apply the method – either Swap or Inverse – only to the selected 3x3 grid.

For example, having as parent the solution shown previously,

Initial Parent 1: [[1, 4, 8,1, 5, 9,4 ,5 ,6],
[5, 2, 9,3, 2, 7,2, 7, 3],
[7, 6, 3,4, 8, 6,8, 1, 9],
[9, 1, 5,1, 3, 5,5, 6, 8],
[7, 8, 3,4, 7, 9,2, 1, 7],
[4, 6, 2,8, 6, 2,3, 9, 4],
[2, 7, 6,1, 4, 8,5, 3, 1],
[3, 8, 9,2, 3, 5,6, 2, 9],
[5, 1, 4,6, 9, 7,8, 4, 7]]

If the selected grid is the one in position 5, the methods would be applied to the following sequence of numbers: [1,3,5,4,7,9,8,6,2].

8. Comparing different Models

We applied Elitism to our model and the results were better with Elitism than without it. We also experimented generating a probability of applying Elitism, but the outcome was the same – convergence is more likely when we use Elitism. In addition, we defined that from one generation to another, the worst 10 individuals should be replaced by the 10 bests from the previous generation. The number of individuals to be replaced was set as a parameter, 'n_elite', for the user to choose.

We also notice that our model often got 'stuck' in a certain fitness value. To overcome this issue and facilitate and/or accelerate convergence, we added another parameter called 'max_same_fitness', which defines the number of generations the model is allowed to be stuck in a certain fitness value. For instance, if this parameter is set at 100, after 100 generations of obtaining the best individual with the same fitness, the population would be renewed – i.e., the algorithm would generate new individuals all over again, as it did in the beginning of the algorithm.

Another parameter we added to the evolve function is for the user to define the fitness value of the global optimum ('global_optimum'). This implementation was done so that the algorithm stops when it finds the final solution.

To assess the configurations that work best together, we run several combinations 10 times. The following table has all the combinations experimented.

pop_size = 500, gens = 500, co_p = 0.9, mu_p = 0.1, elitism = True, n_elite = 10, max_same_fitness = 89													
Different Combinations	**Nr. Of Runs**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average Fitness Value**	**Average Total Fitness**
Cycle Crossover + Swap Mutation	**Tournament Selection**	6	6	2	4	6	6	5	4	4	6	4,9	5,10
	Ranking Selection	4	4	2	7	7	5	7	6	2	6	5	
	Fitness Proportionate Selection	2	6	6	8	7	5	4	6	4	6	5,4	
Cycle Crossover + Inversion Mutation	**Tournament Selection**	7	16	8	5	4	6	7	4	8	7	7,2	7,43
	Ranking Selection	5	8	7	4	5	10	4	6	5	6	6	
	Fitness Proportionate Selection	11	7	5	10	10	10	6	12	10	10	9,1	
Partially Mapped Crossover + Swap Mutation	**Tournament Selection**	2	6	5	7	2	2	4	4	4	4	4	4,73
	Ranking Selection	6	4	3	4	4	4	5	4	5	4	4,3	
	Fitness Proportionate Selection	8	8	6	4	6	6	4	7	2	8	5,9	
Partially Mapped Crossover + Inversion Mutation	**Tournament Selection**	6	6	4	2	4	2	4	4	2	3	3,7	5,03
	Ranking Selection	12	4	4	6	6	12	3	7	4	8	6,6	
	Fitness Proportionate Selection	4	2	4	7	5	4	7	4	4	7	4,8	

On a first instance, by looking at the average fitness value of all combinations, we can see that for each 'group' (one 'group' being one possible crossover and mutation combination), the Tournament selection method behaved better than the other two selection methods. This behavior was expected since it is a method through which we are actually choosing the best parents, so the children (offsprings) are expected to be better. Regarding the other two selection methods, Fitness Proportionate Selection is better than Ranking Selection, theoretically. However, in our problem, this pattern is not verified. The poor performance of Fitness Proportionate Selection might be happening because the fitness values are always very close to each other. Comparing the Crossover methods, in theory, there is no better method between Cycle and Partially Mapped Crossover. But, according to our problem definition, fitness function, and representation, the Partially Mapped method proved to work better. Regarding the combination of Partially Mapped Crossover with either Swap or Inversion Mutation, there is not significant evidence that allows us to conclude that one is better than the other. However, since from using the combination of Partially Mapped Crossover, Inversion Mutation, and Tournament Selection we obtained the lowest fitness value out of all combinations, we will continue with it.

Now, we will experiment several values for the crossover and mutation probabilities in the best model defined previously.

Partially Mapped Crossover + Inversion Mutation + Tournament Selection												
pop_size = 500, gens = 500, elitism = True, n_elite = 10, max_same_fitness = 89												
Different Combinations	**Nr. Of Runs**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average Fitness Value**
co_p = 0.95, mu_p = 0.05	**Tournament Selection**	4	3	6	4	4	0	4	5	3	4	3,7
co_p = 0.9, mu_p = 0.1		6	6	4	2	4	2	4	4	2	3	3,7
co_p = 0.85, mu_p = 0.15		6	2	4	7	4	4	4	6	4	8	4,9
co_p = 0.8, mu_p = 0.2		2	4	6	9	5	3	5	6	5	3	4,8

We can see that the combination of co_p = 0.9 and mu_p = 0.1 and the combination co_p = 0.95 and mu_p = 0.05 obtain the same average fitness value. Since we were able to obtain a 0 in the case of 0.95/0.05 probabilities, we will stick to this one.

9. Applying the Best Model

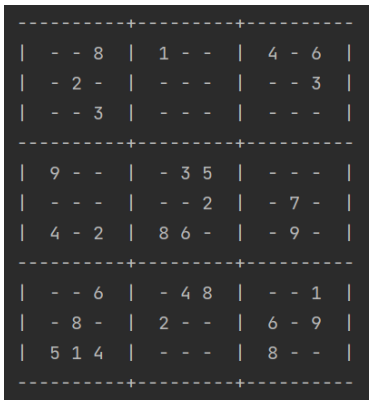
Now that our best model is defined, we are going to run it 10 times for each of the four level of difficulty Sudokus we have in our data. For this step, we will increase both the number of individuals and generations, to 1000 and 600 respectively. This approach will, hopefully, increase the performance of our model.

Tournament Selection + Partially Mapped Crossover + Inversion Mutation											
pop_size = 1000, gens = 600, co_p = 0.95, mu_p = 0.05, elitism = True, n_elite = 10, max_same_fitness = 89											
Different Difficulty Levels	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average Fitness Value**
Very Easy Solution	2	2	2	3	2	2	0	6	4	2	2,5
Easy Solution	3	3	4	2	2	0	4	2	4	4	2,8
Intermediate Solution	4	5	4	4	4	4	4	6	4	4	4,3
Hard Solution	4	6	5	5	4	6	5	4	6	4	4,9

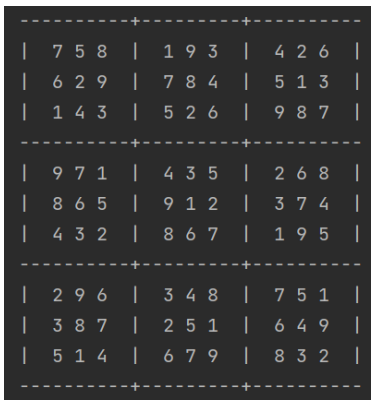
As expected, as the difficulty level increases, so does the average fitness value, i.e., the model is able to predict easy solutions better than harder ones.

The results obtained, by increasing the population size and the number of generations, were rather good, since we were able to actually find a solution for a ‘very easy’ sudoku and for an ‘easy’ sudoku.

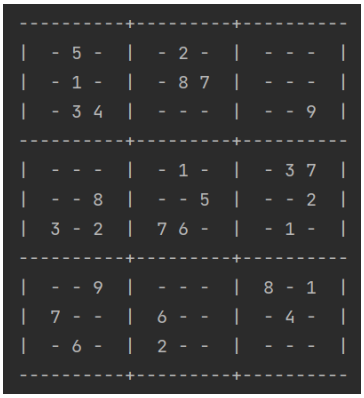
Very Easy Sudoku



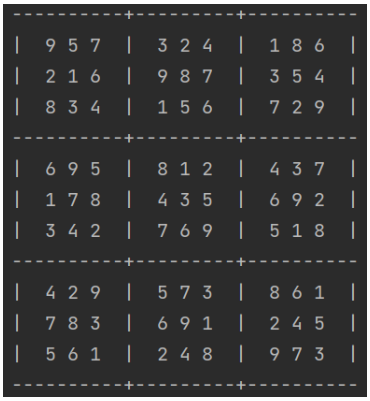
Very Easy Sudoku Solution



Easy Sudoku



Easy Sudoku Solution



10. Conclusion

In addition, all methods we implemented were done for both minimization and maximization problems, although our problem is a minimization one.

Overall, the outcome of our project is good, and we even got to the final solution (global optimum) in two out of the four difficulty levels we had in our ‘data’. However, there are steps we could have done to, possibly, improve the algorithm for all solutions, namely:

- Increase the number of individuals (population size) and the number of generations – theoretically, we already knew that when defining these two parameters the higher the better; however, we obtained significant performance improvements when we went from 200 individuals to 1000, and from 400 generations to 500. As such, it is plausible the inference that if we increased these numbers even further, better results would be obtained.

- A different fitness function – we tried two different fitness functions; however, more could have been explored. For example, we could have computed the average of duplicate values, and maintain the minimization problem rationale we have been developing. Or we could have taken another approach and count the unique values, turning the problem into a maximization one. These approaches were not tried due to time constraints.

- Fitness Sharing – what could be happening in our case could be loss of diversity at a certain point, so a possible solution would be to monitor variance or entropy to measure this loss and apply fitness sharing to maintain diversity in our population.

11. References

Vanneschi, Leonardo. Computational Intelligence for Optimization.

https://nidragedd.github.io/sudoku-genetics/

Genetic-Algorithm-based-Sudoku-Solver/GA_Sudoku_Solver.py at master · chinyan/Genetic-Algorithm-based-Sudoku-Solver. GitHub. (2022). Retrieved 29 May 2022, from https://github.com/chinyan/Genetic-Algorithm-based-Sudoku-Solver/blob/master/GA_Sudoku_Solver.py.