NOVA IMS Information Management School

Solving Sudoku puzzles by using genetic algorithms

Computational Intelligence for Optimization - Summer Term 2021

Group: Sudoku Samurai

Filipe Coelho
(m20200580@novaims.unl.pt)

Franz Michael Frank
(m20200618@novaims.unl.pt)

1 Introduction

In the 1980s, the number puzzle Sudoku, which originated from Japan, became more and more popular, nowadays it is known all over the world. The rules are fairly simple: A 9x9 matrix is given, which contains several empty fields as well as some already fixed digits, so-called clues. The empty fields must be filled in, whereby only numbers between 1 and 9 are allowed. The condition here is that neither in a row nor in a column a number may occur twice. This also applies to the 9 3x3 so-called blocks, of which the entire matrix consists. The goal of this project is to develop a genetic algorithm that generates suitable solutions for such a Sudoku puzzle. First, a representation of the problem is designed, whereby especially the implementation of the fitness function plays a crucial role. Afterwards, an optimization of the algorithm using the best selection, crossover and mutation methods as well as other optimized parameters is attempted in order to completely solve as many Sudoku puzzles as possible.

2 Representation of the problem

2.1 Creating a Sudoku puzzle

The *create_sudoku()* function within the *sudoku_generator.py* file creates a Sudoku puzzle of a specified level of difficulty. The first part of this function creates a random but valid 9x9 Sudoku matrix, which is represented by a list of lists at first. Subsequently, the function removes random fields from the matrix, by setting them to 0. The number of deleted fields depends on the given difficulty: “easy” Sudokus will contain 30 clues, “medium” ones will contain 25 and in “hard” Sudokus there will be 20 remaining clues. In reality, there is no single correct definition of the difficulty. Instead, there are several approaches for defining the difficulty, which are usually more complex. But for this project, it is reasonable to create Sudoku puzzles in the described straightforward manner, as this will result in Sudoku matrices within the respective levels of difficulty that are more similar. This allows a better comparison of different genetic algorithms applied to various puzzles of a single difficulty class. For this reason, also the respective number of clues is fixed, rather than variable. Instead of returning the created Sudoku matrix as a list of lists, the function will return it as a flattened one-dimensional list for reasons of simplicity. Thus, the function finally returns a list of a fixed length of 81, with digits from 0 to 9, whereby the zeros indicate that the corresponding fields of the Sudoku matrix are empty, and numbers from 1 to 9 represent fixed fields. Furthermore, the first 9 values of the list represent the first row of the Sudoku matrix, the values of the positions 9-17 represent the second row and so on.

The second function of the *sudoku_generator.py* file is *display_sudoku()*. It prints a Sudoku puzzle given as a list in a proper way, similar to the common Sudoku layout. It can display both, either an unsolved puzzle or an unsolved puzzle together with a solution.

2.2 Initializing the population

For the purpose of creating an initial population for a genetic algorithm, the function *initialize_population()* within the file *initialize_population.py* was created. The function requires 2 arguments, the desired

population size as well as the wanted level of difficulty. The process starts by calling the *create_sudoku()* function with the given difficulty. It is important to mention that there will only be one single instance of a Sudoku puzzle. Subsequently, a specific number of possible solutions, equal to the given population size, will be initialized randomly. Thereby, for every 0 within the previously created Sudoku matrix, a random number between 1 and 9 will be created and added to a list. This list represents one instance of the population. As every number can be created several times, this initializing process is done with replacement. The function returns a dictionary consisting of 2 key value pairs: the value for the key "population" is a list that contains all the randomly created solutions as one-dimensional lists, the value for the key "sudoku" is a simple one-dimensional list that represents the created Sudoku puzzle. It is important to mention that while the latter list will always have a length of 81, the lists within the list for the key "population" will only contain values for the fields that are not fixed, so have a length smaller than 81, varying according to the amount of clues given in the puzzle. This list of lists is used as the population for the genetic algorithm problem. Thereby, a simple population representation is provided, as the individual instances are simple one-dimensional lists, that don't contain clues, which means that all of the elements of the list can be modified later on by a genetic algorithm.

2.3 Fitness function

The *evaluate()* function within the *main.py* file represents the fitness function for the genetic algorithms. To calculate the fitness score of a population instance, the instance must be passed together with the corresponding unsolved Sudoku puzzle. The function will create a Sudoku matrix with the clues form the puzzle and the solutions provided by a population instance. For the first part of this project, the fitness score will be calculated as follows: For every row, the fitness score will be increased by the number of unique numbers within that row. If that row consists of unique numbers only, it gets one extra point on top of that. That means that each of the 9 rows of the Sudoku can get a maximum score of 10. Likewise, the fitness score is increased similarly for each column and block of the Sudoku matrix. Hence, the maximum fitness score that a population instance can get is 27x10 = 270, which indicates that it is a valid solution for the given Sudoku puzzle.

At a later stage of the project, a second fitness function is implemented and chosen to be the final one. This fitness function is quite similar to the first one, however, this one doesn't award extra points for completed rows, columns or blocks. Thus, the maximum fitness score for this function is 27x9 = 243. Since, of course, the aim is always to achieve the highest possible fitness score, the task at hand represents a maximization problem.

3 Development of the setup

Altogether, 3 selection (*fps, rank, tournament*), 3 crossover (*cycle_co, pmx_co, single_point_co*) and 3 mutation (*distinct_mutation, inversion_mutation, swap_mutation*) operators have been tested through all 27 (3x3x3) possible combinations of

them. 8 out of these 9 methods are from the original *Charles* library, while the mutation method *distinct_mutation* has been newly created. This function mutates an individual by randomly choosing a mutation point and subsequently exchanging its value with that of another random point. This happens under the condition that the new value with which the initially selected value is exchanged is neither the same as the initial one nor as the four nearest neighbors, which are the 2 values before and after, of the initial one.

In total, each of the 27 different configurations was performed 75 times each, with 25 runs for the difficulty levels easy, medium and hard respectively. With each new attempt, a new Sudoku of the respective difficulty level was generated. Thereby the obtained results became more meaningful, because they do not depend on the special peculiarities of a single individual Sudoku. Populations of size 500 only were created in each case, for reasons of time efficiency. Therefore, the achieved fitness values couldn't be expected to approach 270 at this point, but a reasonable benchmark for intercomparison between the configurations could be achieved. In order to obtain comparable results, all configurations were performed with the same parameter settings (*gens = 100; co_p = 0.7; mu_p = 0.2; elitism = True*).

The results of this comparison are illustrated in Figure 1. There were 2 configurations that outperformed the others across all difficulty levels:

- selection: *tournament*;
  crossover: *pmx_co*;
  mutation: *swap_mutation*
- selection: *tournament*;
  crossover: *single_point_co*;
  mutation: *swap_mutation*

Furthermore, the first configuration represents the 2nd most time efficient one, while the second one ranks 7th overall in terms of time efficiency.

Top 5: All difficulty levels combined

Configuration	Average Fitness
tournament, pmx_co, swap_mutation	249.65
tournament, single_point_co, swap_mutation	249.30
tournament, cycle_co, swap_mutation	247.95
rank, pmx_co, swap_mutation	245.70
tournament, pmx_co, distinct_mutation	245.00

Top 5: Difficulty level easy

Configuration	Average Fitness
tournament, single_point_co, swap_mutation	250.84
tournament, pmx_co, swap_mutation	250.16
tournament, cycle_co, swap_mutation	249.12
rank, pmx_co, swap_mutation	248.84
rank, single_point_co, swap_mutation	248.00

Top 5: Difficulty level medium

Configuration	Average Fitness
tournament, single_point_co, swap_mutation	249.20
tournament, cycle_co, swap_mutation	248.80
tournament, pmx_co, swap_mutation	248.12
rank, pmx_co, swap_mutation	245.08
rank, pmx_co, distinct_mutation	244.36

Top 5: Difficulty level hard

Configuration	Average Fitness
tournament, pmx_co, swap_mutation	250.68
tournament, single_point_co, swap_mutation	247.84
tournament, cycle_co, swap_mutation	245.92
tournament, pmx_co, distinct_mutation	245.88
tournament, cycle_co, distinct_mutation	243.24

Figure 1: Best performing configurations

In addition, the configuration *tournament, cycle_co, swap_mutation* has also performed well in terms of the fitness score, but it has the major drawback of being very time consuming. Overall, the configurations with the operator *cycle_co* take 99 times as much time as those with *pmx_co* and 59 times as much as those with *single_point_co*. In Figure 2, the average fitness scores can be obtained for each of the 9 different selection, crossover and mutation operators individually. So, for example the first blue selection bar shows the average fitness score of all the 9 different configurations that include the *rank* selection function. Among the 9 different methods there is only one which did not work at all for the given problem, that is the selection method *fps*. Therefore, it can be concluded that *fps* selection is not capable of handling the problem at hand.

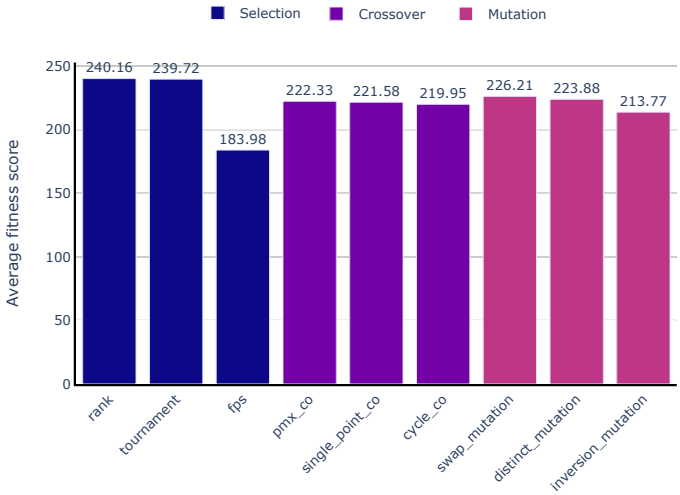

Figure 2: Average fitness score per method

Afterwards, the best 2 configurations were tested again under the above conditions, but with the change of *pop_size* to *5000* to get an even more accurate comparison. The configuration *tournament, pmx_co, swap_mutation* performed slightly better than the other one, with an average fitness score of 258.01 compared to 256.49. Therefore, this configuration was finally selected as the overall strongest one and from now on only this configuration was used.

Since now the final configuration for the operators was chosen, it could be tested whether the genetic algorithm performs better with the inclusion of elitism or without. For this purpose, the algorithm with *pop_size = 1000* was executed with *elitism = True* and *elitism = False*, respectively, with 75 rounds each, similarly to the approaches described above. The attempt with elitism was slightly better than the one without, with an average fitness score of 253.32 compared to 251.95. Therefore, it was decided to include elitism, as it had only a small but positive impact.

At this stage of the project, fitness scores close to 270 were already achieved with increased population size, but the maximum value and thus a complete solution for the Sudoku was only achieved extremely rarely. Therefore, the second fitness function with the maximum value of 243 was tested for the first time at this point. For both fitness functions, the algorithm was repeated 75 times, i.e. 25 times for each of the 3 difficulty levels. Besides the already defined selection, crossover and mutation operators, the parameter setup was as follows: *pop_size = 20000; gens = 100; co_p = 0.7; mu_p = 0.6; elitism = True.* The result of this comparison was quite clear and in favor of the fitness function with the maximum of 243. With this function, 5 (3x easy, 1x medium, 1x hard) of the 75 Sudokus were completely solved, while with the other function, only 2 (1x easy, 1x medium) were completely solved. Overall,

the better function with an average value of 238.89 reached about 98.31% of the maximum possible value of 243, while the inferior one only achieved 96.89% of 270. From this point on, only the fitness function with the maximum of 243 was used.

In order to find the optimal value for the parameter *co_p*, which represents the crossover probability, the following configuration was tested for the *co_p* values *0.6, 0.7, 0.8* and *0.9*: *pop_size = 2000; gens = 100; mu_p = 0.6; elitism = True.* As shown in Figure 3, the most appropriate value is *co_p = 0.8*.

Value of *co_p*	Average Fitness
0.8	236.31
0.9	235.93
0.7	235.53
0.6	233.75

Figure 3: Best performing co_p value

The most suitable value for the *mu_p*, which represents the mutation probability, was identified similarly to the above approach concerning the *co_p*. Therefore, the same configuration was used, with the only difference that this time the *co_p* was fix, with a value of *0.8*, while the *mu_p* value was variable. The results of this test can be seen in Figure 4.

Value of *mu_p*	Average Fitness
0.7	236.57
0.6	236.31
0.8	236.21
0.4	235.67
0.2	234.37

Figure 4: Best performing mu_p value

Finally, the final algorithm was performed in the following configuration: *pop_size = 200000; gens = 75; co_p = 0.8*, *mu_p = 0.7; elitism = True.* Of course, there is a tradeoff between time and performance, especially when choosing the population size. However, since the goal of this project is to completely solve Sudoku puzzles as frequently as possible, which is only the case when the fitness score of 243 is reached, the population size was chosen to be rather high. The parameter *gens* was reset to 75, because a first test of the configuration showed that the result does not improve after the 75th round. As a result, the genetic algorithm completely solved 60% of the easy, 48% of the medium, and 36% of the hard Sudokus. It is therefore obvious that, just as with humans, puzzles with a higher difficulty level are more difficult to solve for the algorithm. This is mainly because the individuals for more difficult levels are larger. In general, if the population size is set even higher, the algorithm can be expected to solve the puzzles even more frequently.

4 Conclusion

This project shows that through the use of genetic algorithms it is possible to find good solutions for Sudoku puzzles and often even to solve them completely. The *Charles* library offers all methods and operators that are necessary for this purpose. The implementation is abstract and could also be adapted to a minimization problem. However, there is still room for improvement, especially if the population size gets increased and thus the Sudokus would be solved even more frequently. Additionally, due to the already high success rate of the algorithm, for a desired application that completely solves Sudokus 100% of the time, it would be possible to restart the algorithm for a single Sudoku until the puzzle is fully solved.