NOVA IMS Information Management School

COMPUTATIONAL INTELLIGENCE FOR OPTIMIZATION

2020-21

SOLVING SUDOKU PROBLEMS USING GENETIC ALGORITHMS

Bugginators Group

Ana Carrelha m20200631
Inês Melo m20200624
Inês Roque m20200644
Ricardo Nunes m20200611

Introduction

This project was developed as a part of the Computational Intelligence for Optimization course in the Data Science and Advanced Analytics Masters.
Our purpose with this project is to develop and use Genetic Algorithms to solve sudoku problems. This report will explain how we applied our knowledge about GAs and optimisation problems to provide a good solution to this problem. We will describe our implementation approach and justify our decisions, more specifically about the choices related to the problem's representation, fitness function and selection, crossover, and mutation methods.
In the end, we will present the results obtained and select the best approach found.

The Problem

Given that we are dealing with a sudoku problem, we had to follow the sudoku rules. By this, we mean that each row, column, and 3x3 square must have nine unique digits from 1 to 9. Also, when starting a sudoku, we always have a 9x9 grid with some cells already filled (we will refer to these as fixed values). Therefore, our implementation follows this condition where these cells' values cannot be changed.
Our implementation is based on the Charles library developed during classes.

Representation

The individual's representation is a list with a length of 81 (where each element represents one cell of the 9x9 grid).

The only requirement to create an individual is to define the fixed values. This requirement is achieved by defining a dictionary where the keys are the indexes of the list, representing the position of the fixed values in the grid and the dictionary's values being the cell's value.

To fill the empty cells, we randomly assign digits from 1 to 9. However, this assignment is conditioned by the set of possible values. By this, we mean that we know each solution can only have nine repetitions of each digit. Therefore, the fixed values already count as repetitions of their respective digits.

Fitness

Our fitness function considers the number of unique digits in each row, column and 3x3 square. This means that the optimal individual has a fitness value of 243 (9 x 9 x 3). Therefore, our implementation considers sudoku as a maximisation problem.

Selection Methods

Our selection methods are the same that we implemented in the Charles library (Fitness Proportionate, Tournament and Rank). In our understanding, there was no need to modify these methods due to their generalisation capabilities.

Crossover Methods

One small change we did differently from the practical classes is that all crossovers and mutations return an individual in order to keep the dictionary with the fixed values instead of returning only the representation.

1. Single point crossover

The implementation stays the same since all individuals have the same fixed values. This allows to apply single point crossover without changing the fixed values.

Consider the following example of parents where the fixed values are marked in grey:

- Parent1 = [1 1 1 2 3 3 4]
- Parent2 = [4 1 1 3 2 3 1]

Now consider the crossover point is index 3. This will generate the following offspring:

- Offspring1 = [1 1 1 | 3 2 3 1]
- Offspring2 = [4 1 1 | 2 3 3 4]

As said, the fixed values remain in place. However, this crossover method does not guarantee the offspring will have each value repeated exactly nine times.

2. Cycle Crossover

Since every individual shows the same digit nine times, we had to adapt our approach to consider this fact and to avoid ending the algorithm when both parents have the same value in the cycle.

We created a dictionary to save the positions of each digit in the individual's list. With this dictionary, we could search between both parents without stopping the cycle in the first occurrence of a digit. For a better understanding, we present a shorter example with individuals of length 7:

Parent1 = [1 1 1 2 3 3 4]
Parent2 = [4 1 1 3 2 3 1]

In this example, we start in the first position of Parent1 with value 1, and then we look at the value at the same position in Parent2, which is 4. After this, we check the index where that value is in Parent1, that is 6, and we look again to the value in Parent2 in that index, that is 1. Here is the big difference from the practical class implementation: instead of going again to index 0 since it is the first value 1, we check if there is another value 1, which there is with index 1, and continue the cycle until there is no new digit to "visit".

We have the following cycle in Parent1:

Cycle = [1 1 1 _ _ _ 4]

Therefore, we end up with the following offspring:

Offspring1 = [1 1 1 3 2 3 4]
Offspring2 = [4 1 1 2 3 3 1]

Notice this implementation maintains the fixed values in the same position. Following the given example, the value 1 with index 1 in both parents is a fixed value, and we see that it remains in the same position in the offspring.

3. Arithmetic Crossover

This crossover method is very similar to the practical classes' implementation. The only difference is that it returns a rounded version of the values.

Since all values from both parents are between 1 and 9, the offspring will have values belonging to [1,9]. This method also maintains the fixed values in place. For example, considering the value 4 as a fixed value in both parents, we have Offspring1 = $4*\alpha + (1- \alpha)*4 = 4$.

However, this crossover method does not guarantee the offspring will have each value repeated exactly nine times.

Mutation Methods

1. Swap Mutation

This mutation consists of selecting two random indexes that do not represent a fixed value (this is, that do not appear in the fixed values dictionary) and mutate them. It maintains the fixed values in place, and each individual has nine repetitions of each value.

2. Inversion Mutation

This mutation is very similar to the one implemented in practical classes. The difference is that we do not invert fixed values. For example, where fixed values are marked with blue:

[1 1 1 2 3 3 4] is inverted to [4 1 3 2 1 3 1]

Therefore, it maintains the fixed values in place, and each individual has nine repetitions of each value.

3. Element Wise Mutation

To implement this mutation method, we took inspiration from the mutation examples given in theoretical classes. If an individual experiences mutation with a probability p, then x=round(len(individual)*p) values will experience mutation. Notice that only non-fixed values are possibly selected for mutation. Then the selected values will be randomly assigned among the corresponding indexes.

For example, considering an individual with representation [1 5 6 2 2 3 4] (fixed values are represented in grey) and a mutation probability of 0.6, there will be round(7*0.6)=4 mutated values. From the non-fixed values, let us select four indexes, for example, 0, 3, 4, and 6. So we have the following individual [_ 5 6 _ _ 3 _], and 1, 2, 2, and 4 to be randomly assigned to those four empty spots. One possible outcome is [2 5 6 4 2 3 1].

This mutation maintains the fixed values in place, and each individual has nine repetitions of each value.

Statistical Analysis

To analyse the performance of our implementation, we decided to test several hypotheses to achieve our best solution to this problem further.

Comparing Algorithms

We implemented all possible combinations of selection, crossover and mutation methods with elitism to find the best one. For each combination, there were 35 runs of 150 generations with fixed crossover and mutation probabilities of 0.6 and 0.2, respectively. As said in theoretical classes, we calculated the average fitness per generation over the 35 runs for each combination.

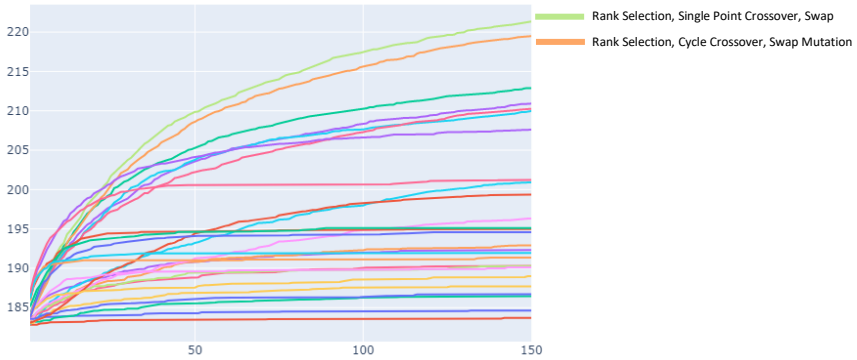

Figure 1- Average Fitness Value per Generation

The two best-performing algorithms were:

1) Rank Selection, Single Point Crossover, Swap Mutation
2) Rank Selection, Cycle Crossover, Swap Mutation

Combination	Average fitness value in last generation	Maximum fitness value	Success rate
1)	221	227	11
2)	219	227	6

Since no algorithm achieved the global optimum (fitness of 243), then we consider 223 (243 minus 20) a reasonable approximation to determine the success rate.

Finetune crossover and mutation probabilities

We tried different probability values ranging from 0.1 to 0.9 with a 0.1 step with the two combinations mentioned above. This returned 162 different algorithms.

We consider the algorithm with single point crossover with a probability of 0.9 and mutation probability of 0.4 as the best one. We made this decision since it has the highest average fitness value in the last generation (226) and a maximum fitness value of 232, which only differs one unit from the maximum fitness value achieved among all probabilities combinations, and a success rate of 2 considering 230 as a reasonable approximation. We decided to decrease the distance to the global optimum because we are finetuning the best combinations. If we assumed 223 as the reasonable approximation, this algorithm has the highest success rate.

Increase mutation probability and decrease crossover probability over generations

We tried to implement an increasing mutation probability and decreasing crossover probability over generations with 5%, 1%, and 0.1% rates. In any of these cases, the fitness values were worse than the algorithm with fixed probabilities.

Applying the best algorithm to solve different difficulty sudokus

To test the generalisation of our implementation, we applied it to three distinct sudoku problems with different levels of difficulty (easy, medium, and high). Doing this, we observed that the algorithm's behaviour was very similar across the different experiences. For that reason,

we consider that our implementation is well generalised and does not present significant differences between levels of difficulty.

Conclusion

Reaching the end of this project, there were some findings that we were not expecting.

Firstly, the algorithm considered the best did not achieve the global optimum. To be more precise, none of the possible algorithm combinations achieved it. We consider this finding an opportunity to enhance our project, which can possibly be accomplished by trying different fitness functions or different selection, crossover, and mutation methods.

Secondly, finding the algorithm with single point crossover as the best one was a surprise since it does not guarantee each digit's nine repetitions, which we thought would negatively affect the individuals' fitness values.

Also, we were expecting that the best algorithm would have a higher mutation probability than crossover probability. We thought this way because, from our point of view, changing values inside an individual would be better than changing values between two different individuals since the latter option might lead to repetition and lack of digits.

There are some changes and other methodologies we would like to implement to improve the algorithm performance.

One is to give more freedom to the algorithm in the mutation methods to randomly choose any value from the valid set without being restricted by the number of repetitions of each value. Single point crossover being the best crossover method made us think we should give the same freedom to the mutation methods to try to improve fitness.

Another change could be applying a grid search with the selection, crossover and mutation methods and multiple values for the crossover and mutation probabilities. This way, we would have more combinations and possibly achieve an algorithm with better performance. We did not apply this due to computational power restrictions.

References

Vanneschi, Leonardo. *Computational Intelligence for Optimisation.*

Montali, Dave. Charles Library from practical classes

Sudoku example generator website, sudoku.com/easy

Weiss, Dr. John M (2009). *Genetic Algorithms and Sudoku - micsymposium.org/mics_2009_proceedings/mics2009_submission_66.pdf*

Deng, Xiu Qin & Li, Youg Da (2011). *A novel hybrid genetic algorithm for solving Sudokupuzzles* - math.uci.edu/~brusso/DengLiOptLett2013.pdf