CIFO Project Report

Solve Sudoku with a Genetic Algorithm

Daniel Philippi (m20200733@novaims.unl.pt)
Mario Rodríguez Ibáñez (m20200668@novaims.unl.pt)
Doris Macean (m20200609@novaims.unl.pt)

We decided to take on the challenge of implementing a Genetic Algorithm (GA) in order to solve a Sudoku puzzle. In this project we focus on the standard 9x9 Sudoku board. Sudoku is known to have a rigid fitness landscape with many local optima. As such, the main challenge will be to identify and implement configurations and operators to efficiently escape those local optima and converge to a perfect solution.

Code can be found on GitHub

1 Framework

In order to test as many configurations as possible, we developed a framework based on the Charles library. The framework expects a configuration file, or a grid of configurations, and automatically performs the evolution of the GA over a determined number of epochs. The results are then saved for later analysis.

Configuration object

The config object is a python dictionary that allows for the selection of different solution representations, GA operators, evolution parameters and additional features. One run over a config corresponds to a unique run id (run_id) and can be iterated “epoch” times to produce statistically significant results. We also implemented grid search, where every parameter can be expressed as a list of configuration options, performing every possible combination as a grid run, and allocated to a certain run_id. Here, every config inside that grid will be assigned to a grid search id (gs_id) underneath the same run_id.

Custom Experiment Tracker

The custom-built experiment tracker stores information from every generation, epoch and run. On the top level it populates an overview csv file on the granularity of one run_id or one gs_id. It manages the incrementation of ids and holds information about the specific configuration and evaluation metrics. In addition, on every run a new folder will be created. The folder holds the history of both fitness and diversity over each generation; per epoch and as mean over all epochs. Also, it stores the configuration as a json file, that allows for a convenient rerun of a certain configuration.

Evaluation criteria

Each configuration will be evaluated based on the mean and standard deviation across the best fitness achieved in each epoch. These metrics will serve as the main evaluation criteria. Furthermore, we will look at the fitness and diversity history over the generations, to understand the convergence behavior of a certain configuration. Finally, we track the total duration of a run over all epochs.

Sudoku class

The Sudoku class is able to generate random puzzles of varying difficulty. It starts at 25%, 50% or 75% of empty fields and iteratively adds fields from the original solution until the puzzle remains with one unique solution. It is based on a set of rules to generate admissible Sudoku solutions and on a deterministic solver. The solution obtained by the deterministic solver will not be used in evolving the GA and only serves as reference point for final evaluation. Furthermore, the Sudoku class supports pretty printing of a Sudoku board of various types, including the option to highlight initial positions and the ground truth in a solution generated by the GA.

For comparability purposes we generated one puzzle and its deterministic solution for each of the three difficulty levels. All experiments will be based on one of those three available puzzles. The difficulty level can be selected in the configuration. Each of those three puzzles available vary in proportion of empty fields ranging from 25% at the lowest difficulty, over 46% to 57% at the highest difficulty level and have only one unique perfect solution.

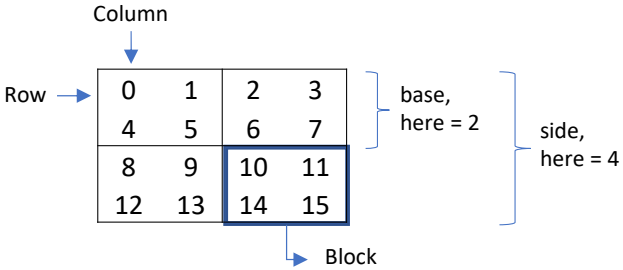
Our approach is to use the highest difficulty to identify the best configuration, assuming this to also be the best for the other puzzles.

2 Implementations

Different ways to index a Sudoku board

A Sudoku board can be represented in different ways. Figure 1 shows the different types of representations that were used in this project. We implemented functions, that dependent on the base size automatically calculate the indexes of those types, in order to perform actions on the subsets specific to each one. For example, considering the block representation, the first group consists of the positions [0, 1, 4, 5] in a board of base size of 2.

Indexes of a Sudoku 4x4 board:



Types of index representations

flat_board:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

by_row:	by_col:	by_block:
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]	[[0, 4, 8, 12], [1, 5, 9, 13], [2, 6, 10, 14], [3, 7, 11, 15]]	[[0, 1, 4, 5], [2, 3, 6, 7], [8, 9, 12, 13], [10, 11, 14, 15]]

Figure 1: Indices and their groups in the sudoku structure

Representations

Early in the project we decided to use the flat board as the natural representation of a Sudoku individual. This makes it similar to the TSP problem, allowing for the easy refactoring of most of its operators. When using more specific operators that are based on the original row-wise matrix shape, the representation can be modified on demand. Based on the flat shape, we implemented four different operators to generate the individuals. While the first two were refactored from the original Charles library, the second two are designed specifically for the Sudoku board.

Draw with replacement

The type “with replacement” randomly draws digits from the array [1…9] and appends them to the flat board, until the desired length has been reached (here: 9x9=81). This allows for solutions that have too few and/or too many numbers of a certain digit. Thus, producing individuals with high variance, but at risk of possibly very unfit ones.

Permutation without replacement

The second type of representation is a random permutation of the array populated with nine times each digit from [1..9]. In this case, every individual is guaranteed to have the correct number of different digits at the cost of some loss in diversity.

Maintain initial positions

The two prior mentioned representations may produce individuals in which the initial positions do not match the puzzle that has to be solved. In order to produce more fit individuals from the start, we implemented another representation. It follows the idea of permutation without replacement, only now freezing the initial positions, limiting the permutations to the remaining ones. It by design guarantees no duplicates inside each row.

Random mix

To combine the advantages of high diversity with the ability to maintain the original positions, we implemented a fourth representation. Here, for every individual, the representation will be chosen at random. The resulting first generations consists partially of individuals that match the initial positions of the puzzle, and total random ones.

Fitness function

The fitness function is designed to penalize two different types of errors in a solution. The first type counts the number of duplicates in either rows, columns or blocks. It uses the different index representations explained in Figure 1 to group the cells of the board and count the number of duplicates within each of those groups. The second error type penalizes deviations from the initial puzzle. It looks up the values at the initial positions in the individual and compares them with the ones from the initial puzzle. Every time those values do not match, the total error count will be incremented by one. As we consider the second error type more severe, we penalize it

with a ratio of 10:1 compared to the first error type. Finally, all the error counts are summed up into the total number or errors, which serves as the fitness value. This way, the best fitness possible has a value of 0, without any duplicate in any of the shapes and with all fields matching the initial positions of the puzzle to solve. Consequently, we deal with a minimization problem, with a known global optimum of 0.

Note that by design, certain types of representation cannot produce an error of type 2. Thus, the fitness of the initial population will be much better. We will observe how far the GA is able to evolve equally fit offspring over the generations, starting with the different types of individual representations.

Minimization vs. Maximization

The code was developed and tested to be suitable to work with both, maximization, and minimization problems.

Operators

We tried multiple operators implemented in the Charles library, but some of them were not applicable to our problem or were not compatible with some of the implemented representations. For example, the arithmetic crossover cannot be used in our problem, as we want a sequence of digits as a solution and not real numbers. Similarly, the binary mutation cannot be implemented as we are not working with any binary representation. Cycle or partially matched crossovers are problematic with our representations as they expect sequences of unique values as input, and when they are fed repeated values, it can lead to infinite loops.

Selection

We implemented two different selection operators. First, we refactored the tournament selection from the Charles library. Second, we implemented a fitness proportionate selection operator, suitable for minimization problems. Here, we use NumPy's choice method, which allows for the addition of weights to each element in the list of choices. The weights are calculated based on the proportion of an individual's fitness in regard to the total fitness, then normalized between [0,1] and reversed.

Crossover and mutation

We refactored certain operators from Charles. Namely, swap- and inversion-mutation and single point crossover. Additionally, we implemented a partially match crossover operator following the algorithm explained in chapter 3.5 from the booklet. In a rather different manner, we created two new Sudoku specific crossover operators, "cycle_by_row_co" and "partially_match_by_row_co". These two operators are based on the cycle crossover and partially matched crossover, with the difference of not being applied to the whole sequence of digits but instead to the corresponding rows of the sudoku board. They first load the flatten board as a list of the different rows of the sudoku and then use either cycle or partially matched crossover on every single row. It is important to note that these operators, like the rest, are only applied with a certain "crossover probability", but once they are used, every row is affected.

Crossover and mutation operators with "maintain_init_positions"

In combination with the representation "maintain_init_positions", it is not necessary to modify the individuals at the initial positions, as by design they are already correct. Thus, we implemented a generic function, that can be wrapped around any crossover or mutation operator. Here, instead of directly passing the entire individual, it will first remove the initial positions, then perform the actual operation and finally insert the initial positions back in. This way the initial positions will stay untouched, without the need of specifically changing any of the operator implementations themselves.

Further implementations

Elitism

We used the elitism implementation from Charles library. In the early experimenting stages, we realized that the usage of elitism lead to a smoother curve of the fitness of the best individual of each iteration. When not using elitism these fitness curves where very noisy and did not seem to converge well.

Diversity

To further understand the fitness evolution over iterations and how the different representations were affecting the algorithm, we implemented two diversity measures. Both are entropy measures. One of them is genotypic, meaning that it will calculate the entropy of the representation of the individuals in the population. However, this measure did not lead to any insight, other than the fact that almost always the "genome" of each

individual is unique. Thus, we implemented and used the phenotypic version, where we calculate the entropy of the fitness of the individuals in the population.

Fitness sharing

Given the scarce diversity within the individuals of the populations, we tried to compensate by implementing fitness sharing following the pseudo-code explained in chapter 3.5.1.1 of the booklet. It first calculates the pairwise Euclidean distance between all the individuals. Then, calculates a sharing coefficient based on how close each individual is to the rest of the population. Using this coefficient, it penalizes individuals who have many "close neighbors" and rewards the more distant individuals. This penalization and reward system consist of multiplying or dividing (depending on if it is a maximization or minimization problem) the fitness of each individual before applying the selection operator. To be able to compare the runs with and without fitness sharing, and for readability reasons, after the selection the fitness is recalculated with the initial fitness function (without fitness sharing).

Early stopping

In order to be able to select a high number of generations, while at the same time optimizing the runtime of an experiment, we implemented early stopping. We introduced the parameter "early_stopping_patience", which serves as a threshold value that defines after how many consecutive generations without fitness improvement to prematurely terminate the evolution. The counter will reset every time a generation achieved an improvement. This is especially useful at the early stages of experimenting, as we can discover a wide variety of different configurations and compare them with less computational effort, in order to identify the most promising ones.

3 Experiments

To ensure that the results are statistically valid all configuration runs discussed below have been repeated 30 times. All of the following results can be reconstructed with the "explore_results.ipynb" notebook, which is part of the git repo.

Set 1: Selection, population size and representation type

The first set of experiments consist of a total of 41 different configurations. The focus lies on how the choice of a selection operator (tournament and fps) and varying population sizes (100, 350, 600, 850) impact the ability to converge to a good solution. We tested those options in combination with different mutation operators and representations.

Ceteris paribus it can be shown that there is a clear tendency for the fitness to improve with increasing the population size, independent from the specific configuration. This is at the cost of largely increased runtime. For the selection algorithm, ceteris paribus shows a mixed result, with 11:10 wins for tournament selection. However, in direct comparison, the gains in mean fitness between the two selection options are always relatively small, with a mean difference of 0.33 and a standard deviation of 0.26. Consequently, the best choice of a selection algorithm varies over the context of further parameters, with only a small impact on the fitness.

As stated earlier, by design the representations "with_replacement" and "without_replacement" have a poorer fitness in the initial population than "maintain_init_puzzle". This line of experiments showed that increasing population size or the number of generations (all 41 configs stopped early within the set patience of 100 generations) is unlikely to close this gap.

Based on those findings, and in order to reduce complexity and the computational effort, we fixed the population size to 50 and selection to tournament for the following experiments.

Set 2: Standard operators with different mutation and crossover probabilities

In the 2nd set of experiments we compared 192 different configurations. Here, we take into consideration all the available types of representations, generic crossover (single point) and mutation (swap- and inversion) operators. We analyzed their behavior on different kinds of crossover and mutation probabilities, as well as with or without fitness sharing. The objective is to see whether the additional choice of parameters verifies the previous findings on the superiority of the representation of type "maintain_init_position".

Ceteris paribus shows that in all scenarios “maintain_init_position” leads to the best mean fitness. The representation types “with_replacement” and “without_repalcement” are always clearly inferior, with a minimum distance of at least 270 fitness points and a standard deviation of 3. While “random_mix” performance is much closer to that of “maintain_init_position”, with mean difference in fitness of 1.22 and a standard deviation of 0.4, it is also always inferior.

Set 3: Find overall best configuration

The 3rd set of experiments builds on the configuration of the previous set. It limits the type of representation to “maintain_init_position” and adds two new types of crossover operators (“cycle_by_row_co” and “partially_match_by_row_co”), which by design only work together with this type of representation. This results to a total of 144 different configurations. After having ruled out several options in the previous sets of experiments, the main objective now is to identify the best configuration and to investigate the impact of fitness sharing.

Ceteris paribus over all varying configuration options, only shows a tendency for the mutation probability (mu_p). Here, the configurations with a mu_p of 0.8 or higher win by a ratio of 25:11.

Considering the top20 configurations, as shown in Figure 2, 80% of the top20 have a mu_p of 0.8 and above. Another finding is that 70% have one of the “by_row” crossover operators, indicating, that more Sudoku-specific operators are superior. Furthermore, with a ratio of 70%, inversion mutation tends to be the favorable operator. Regarding the use of fitness sharing there is a spilt decision.

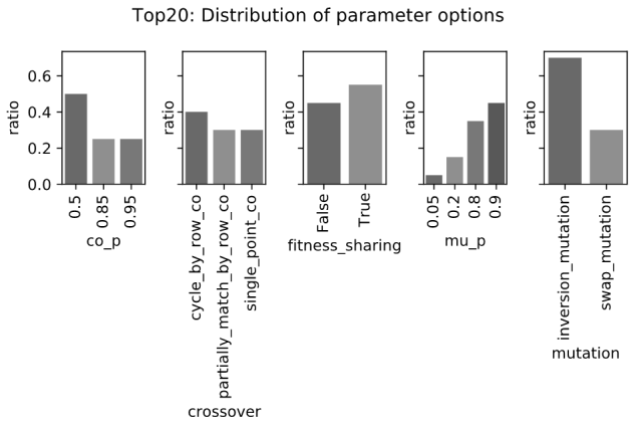


Figure 2: Distribution of top20 parameter options' of exp3

However, looking at the fitness comparison shown in Figure 3 none of the configurations are significantly better (within the range of 1xSD). Therefore, based on the current implementation it is not possible to identify a “best” configuration. The same applies when we decrease the level of difficulty to 2.

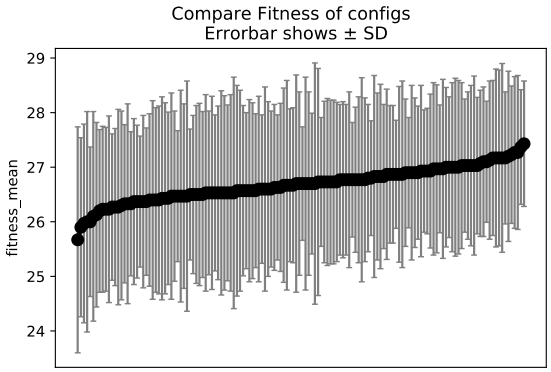


Figure 3: Fitness comparison over configs of exp3

4 Conclusion

After testing and evaluating nearly 400 configurations the best observed mean fitness is 26 at the difficulty level of 3, and 18 at the difficulty level of 2. Bearing in mind that by design, the fitness function counts the same error up to three times, these results are still far from perfect. As in all observed cases, the GA gets stuck in a local optimum. Also, there is no clear tendency towards a certain set of parameters, as multiple configurations lead to similar results, where the best fitness does not differ significantly.

With more computational effort it is possible to explore the space of options even further, possibly identifying better configurations. Testing different populations sizes, fitness sharing, and number of generations is especially costly and thus leaves room for further experiments. Possible further potential lies in understanding the context in which fitness sharing is beneficial to make use of this more efficiently. Also, we need to consider the possibility that the best solution is not yet included in our search space. Thus, identifying and implementing further operators and adding them to the search space can further improve performance.