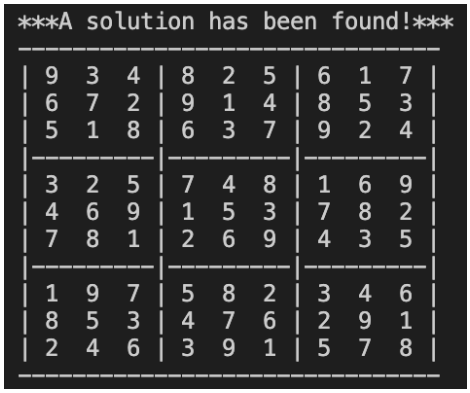
An Approach Based on Metaheuristics Algorithms for the solution of the Sudoku problem

A case for Genetic Algorithms and Simulated Annealing

Tommaso Sacchetti



Abstract. Artificial Intelligence algorithms can be applied to many different areas, each one of those with its particular necessities, that the algorithm has to consider and be built on. In this project two algorithms are going to be applied in the proccess of solution-finding for the Sudoku problem. In particolar the considered algorithms use metaheuristic strategies to find optimized solutions, and are genetic algorithms and simulated annealing. Since the Sudoku problem is not an optimization problem, it's an uncommon problem in which metaheuristic is used: Suokus only need one solution, and any optimization beside it wouldn't be accepted as feasible solution. It's interesting though how it's possible to use the above algorithms even outside of their common usage.

1 Introduction

Sudoku is a logic-based japanes game in which the player places the digits on a board, tipically a 9x9 grid, in order to fill it so that each column, each row and each square, that in case of a 9x9 board will be nine 3x3 squares, contains all the digits from the lowest to the highest. The puzzle is usually given with a partially complete grid that vinculates the moves and sets the difficulty for the puzzle. Sudoku is mathematically an NP-complete problem, and since it only requires one solution, it's not an optimization problem but a feasbility one. For this reason, a metaheuristic approach could not be the best way to solve a

Sudoku but is for sure an interesting and different from the ordinary application for this family of algorithms. From this prior knowledge of the problem it could be possible to formulate some hypotesis that will then be verified. It is in fact expected that the simulated annealing algorithm will be a better solution since it has a statistical nature that practically makes it an enchanced brute-force approach, that is believed to be more adapt than the evolutive approach of the genetic algorithm.

2 Algorithms Overview

2.1 Genetic Algorithm

A genetic algorithm is a metaheuristic inspired process to find optimal solutions to a given problem. It's an evolutionary algorithm therefore it takes inspiration from natural processes, in particolar emulating the evolution of a population in order to find, each generation, better elements, that will bring to a global optimum. It usually consists in creating a population of feasible solutions of the problem, checking their fitness, which is the quality indicator of a given solution, make the individuals reproduce, usually the best ones, substitute the old generation with the new one and then repeat the process until the solution with the wanted fitness is generated.

2.2 Simulated Annealing

Simulated annealing, on the other hand, is a probabilistic technique that takes insipiration from the annealing process, which is a metallurgy technique aimed at creating stronger metals by slowly and constantly decreasing the temperature in a controlled environment to alter the physical properties of the metal. In the same way, the algorithm has an internal "temperature" and after creating a random solution, it looks for a new state, which is a modification of the current element, and checks its quality with a function similar to the fitness function of genetic algorithms. The new solution can be accepted or not, depending on the temperature: with the temperature slowly decreasing, the probability of accepting solutions with a worse fitness than the current one decreases, hence slowly improving the found solutions overtime but keeping, especially at the beginning, a search space that is practically global. The slow convergence and big search space are needed to avoid the algorithm getting stuck too early into local optima.

3 Genetic Algorithm

In the implementation of the genetic algorithm, some design choices were taken in order to improve and model it to the Sudoku problem. Three different selection processes were used: an elitist selection, a tournament selection and a tournament selection that makes compete elements close to each others with a given quantity of possible neighbours to choose from.

The fitness function is the number of conflicts inside of the sudoku hence the "distance" of a proposed solution from the correct one.

The genes are encoded as the list of all numbers, from the top left corner to the bottom left corner of the grid. It's a flat vector that can be used and transformed into a bidimensional 9x9 vector, where every single line is separated.

The implementation is done with Python 3 and the usage of the following libraries:

- `math`
- `mathplotlib`
- `numpy`
- `random`

3.1 Implementation

The whole project is based on four different files: `genome.py, evolution.py, sudoku.py` and `main.py` The implementation is built to be used with different Sudoku problems sizes, just by changing the value of the `DIM` variable. To semplify, in this paper, a sudoku 9x9 (`DIM=9`) with 3x3 blocks, is going to be considered.

genome.py The genome.py file takes care of all the logics inside of the genes and is used to singularly create individuals that share the same functions. The genes of an individual are represented as a vector of numbers. Given the nature of the problem, a key factor is to represent the genes of the indviduals as permutation of each digit, repeated nine times, since it's a 9x9 sudoku and the number of times each digit appears needs to be the same.

So it's key to the whole process that every operation inside of the genes gives as a result a permutation of the initial ones.

Inside of the constructor there is the variable `spawn_chances` that contains the chances that a gene has to select it's spawn method. Since Sudoku is not a trivial application of a genetic algorithm the crossover chance of an individual is set as 20%, the chances for the copy and for a fresh new individual are 10% while the chances of the the usage of the swap as an operator are 60%, since it's the operator that is most likely useful for the solution finding process.

The spawn method is used to choose the reproduction technique for two individuals, based on the chances stored in `spawn_chances`.

The functions `copy()` and `fresh()` respectively copy the gene on which the method is invoked or shuffles every element inside it with the usage of the `random()` function.

The `swap()` function swaps two random element inside of an individual while the `crossover()` function creates a new individual, transmitting the digits that the parents have in common and choosing the transmittor of every other digit, which is the genetic code of the individual, giving a 50% probability of transmission to each parent.

evolution.py The file `evolution.py` contains all the logic of the evolutionary algorithm, it's based on the `genome` class for the operation between individuals. It creates a population of a given size, stored in the variable `POP_SIZE` and starts the evolutive process.

The constructor needs the fitness function written ad-hoc for the problem that is going to be used to evaluate the single individuals and the first genome which will be used to build the population.

Thanks to the function `eden_state()` a fresh population is created based on the first genome and the `_check_best()` function is called to evaluate the best individual's genome and the best relative fitness value of the current population.

The `evolve()` function is the main component of the implementation and uses all of the previously cited elements and functions to actually evolve the population to find the correct solution. It is basically a cycle that creates new individuals and modifies the population until either an external stop signal is sent or the optimal solution is found. In every iteration, a new element is created and substituted to an element existing inside of the population. In this way the count of individuals remains always the same. For this reason this is not a traditional genetic algorithm method, where a new generation of child substitutes partially or globally the previous generation but rather every single child is taken care individually. This choice comes from the necessity of lowering the implementation complexity and with the awareness that the general result is practically unvaried. Three selection functions are implemented in order to evaluate the results of different approaches.

`elite_selection()` chooses the two best individuals of the population and couples them. Since this approach is not considered a valid approach for the solution of a Sudoku, it was modified, giving a 75% chance for the selection of the two best individuals and a 25% chance for the coupling of the best individual with a random one inside of the population. This choice was made in order to keep a slower convergence towards the solution but as the evaluations will show later, this was not enough to avoid local optima.

`tourney_selection()` is the implementation of the tournament selection that selects a predetermined quantity of individuals, that can eventually be modified by changing the value of the variable `TOURNEY_SIZE` and is set to 3 as empirically is the size that works best and offers a minor chance of getting stuck inside of local optima. The function, once individuals are randomly chosen,

orders them and the two best individuals generate a new child that is going to subsitute the worst competitor of the tournament.

`tourney_selection_local()` does exactly what `tourney_selection()` does but chooses individuals that are close in the population vector, making convergence slower and increasing the probablity of finding the optimal solution and not a local minimum. The idea of this implementation comes from a blog post [1] in which an implementation of genetic algorithms applied to Sudokus is shown and since with all the other selection methods it wasn't possible to find a correct solution, it was decided to try this method and, as of now, it's the only selection method that made possible to find a correct solution.

sudoku.py `sudoku.py` contains at a more abstract level all the logics needed to the genetic algorithm to evaluate the particular case of the Sudoku. In particolar it contains the fitness function used by the evolution process to evaluate all the individuals.

Besides all the functions not described here that are used as helper functions to operate inside the vector that represents the sudoku, the main element of the file is the fitness function `sudoku_fitness()` that calculates the fitness by summing all the conflicts inside of the proposed solution. Furthermore, in order to keep it's original numbers fixed as in the puzzle proposal, when a conflict in those number is found, it's weighted 10 times more than a normal conflict.

main.py `main.py` starts the program, creates a base genome from which the evolution will start by creating `DIM` rows with `DIM` ranging from 0 to `DIM`, so no duplicates are created and the solution is for sure a permutation of those elements. After the genome is created, the main function starts the evolution with `target_fitness() = 0`.

3.2 Evaluation of Genetic Algorithm

The algorithm has been evaluated in lots of the possible permutation of parameters and selection functions. The algorithm is able to complete the Sudoku puzzle only using a tournament selection with the local choice of participants, while if an elite selection or the tournament selection without local participants is selected, the algorithm gets stuck on local minima.

The following charts show three runs with three different selection methods. For the elite selection and tournament selection with no local population, data has been truncated for size reason, but the programs ran for about 30 minutes each and there were no improvements. It is possible to see that both tournament selection algorithms converge with the same speed, while the elite selection has a very fast convergence hence there is a very low probability that a global optima is going to be found.

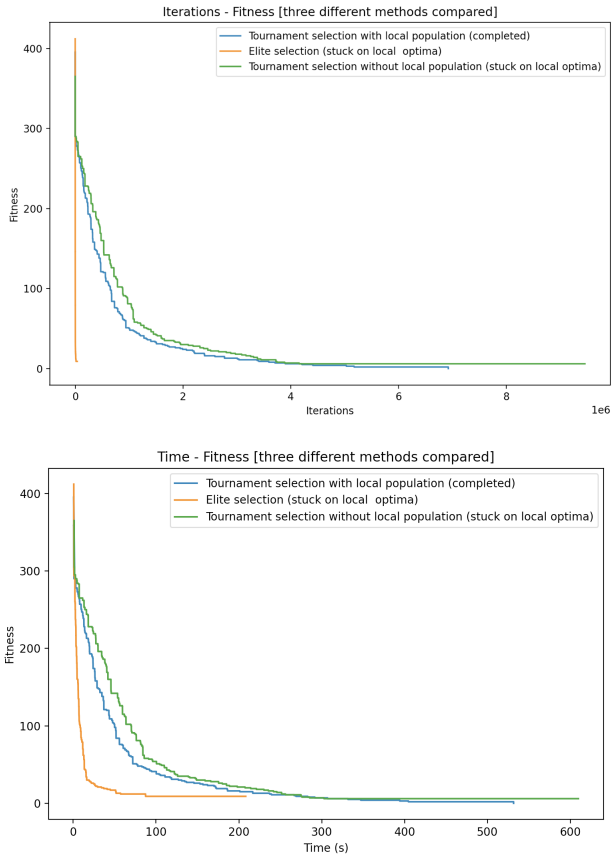


Fig. 1: Comparison of three different selection method, only the tournament selection with the local population could complete the puzzle

In order to improve the elite selection, the `spawn_chances` have been changed, giving the crossover a probability of 50% but the result has been the same, it got stuck in a local optima with a very fast convergence.

In conclusion the genetic algorithm, for how it has been implemented, can solve the Sudoku problems only if individuals reproduce with individuals in an interval of population of a predetermined size with a tournament selection. This method however requires a very big quantity of iterations, in the best case analyzed 6926673: hence a lot of computation power and a long period of time. Probably a finer tuning of the parameters and a right combination of the selection methods, eventually used all together in the same instance of program and dinamically selected during runtime, could improve by a lot the current result.

4 Simulated Annealing

As for the simulated annealing implementation, the semplicity of the project is extremely higher and the whole implementation is done in just one file: `sudoku.py`.

The implementation is done with Python 3 and the usage of the following libraries:

- `math`
- `mathplotlib`
- `numpy`
- `random`
- `statistics`

4.1 Implementation

sudoku.py This file contains the complete logic of the program. It first initializes the variables containing the puzzles at the beginning and contains various functions used by the main part. The individual Sudoku is encoded in the same way of the genetic algorithm implementation, as a `DIMxDIM` vector that is also used as a flat vector that contains all the numbers starting from the top left corner to the bottom right corner of the Sudoku grid.

Besides from the helper Sudoku functions used to manipulate data, that are similar to the genetic algorithm ones and not of particular interest for the purpose of this document, `sudoku.py` contains various interesting functions that dissect and divide the logic of the simulated annealing algorithm.

`numberOfErrors()` calculates all of the conflicts inside of a sudoku solution, and it shares the structure and the objective of the previous genetic algorithm fitness function `sudoku_fitness()`.

The initial Sudoku contains zeroes for unknown values and the numbers of the puzzle. A mask is made with all zeros except ones in the place of the known numbers, so that is always possible know in which position digits can be modified and in which not.

The initial puzzle is filled by `randomlyFillRows()` that fills each row with random numbers and each time a number is added, the validity of that number is checked in order to keep the values inside of the row a permutation of each number ranging from zero to `DIM`.

After the Sudoku is completely built, a sigma value is calculated by `calculateInitialSigma()`, which in this example is equal to the standard deviation of the errors of n randomly generated states where n is the dimension of the Sudoku. The demonstration of this value and the reason why it's the optimal for the Sudoku problem are illustrated on the paper "Metaheuristics can solve Sudoku puzzles" [2]. The decrease factor is set to 0.99 as empirically is the decrease factor that works best for this problem.

After the setup of the initial variables, the program starts a cycle that continues until a solution is found. Inside of every iteration there is a second cycle

that iterates a number of times which is predetermined and that is the number of free spots in the original, empty, Sudoku puzzle. Inside of every iteration, a new state is found thanks to the `chooseNewState()` function which simply randomly flips two modifiable values inside of the proposed solution, calculates the cost of the new solution and then, after rho is computed with `costDifference` and `sigma`, it chooses whether the new solution is a valid one or not and returns either the new Sudoku with the flipped values or the old Sudoku.

If the new solution is not the optimal one, sigma decreases of `decreaseFactor`. Since this approach has a high probability of getting stuck in a local optima, a function that counts for how many iterations there hasn't been an improvement is implemented. A counter `stuckCount` is incremented every iteration where the score is major then the previous one, and after eighty times in a row that this condition is true, sigma is incremented by one hence the whole algorithm restarts.

4.2 Evaluation

The simulated annealing has been evaluated by changing the cooling rate and, since the success rate of the simulated annealing algorithm was much higher than the evolutive one, it was tested with a 16x16 sudoku problem, but it didn't succeed in solving it.

In the following chart three different runs are displayed. Since simulated annealing is a probability based method, the time and iterations required to solve the same problem, can drastically change in a random way. In fact, as shown in the chart, those three runs of the program have a very different trend.

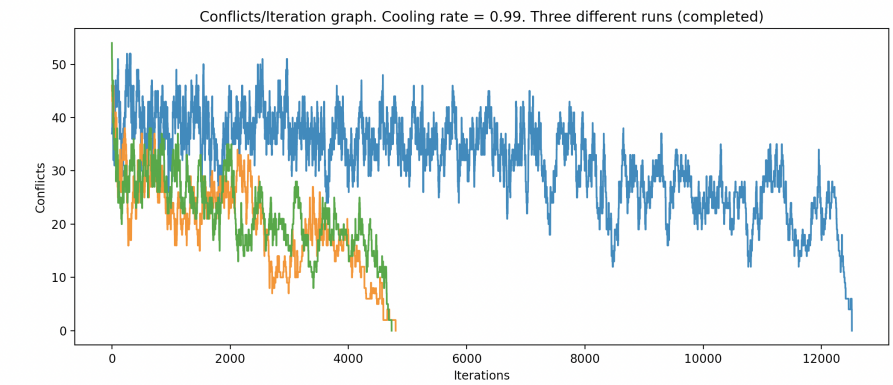


Fig. 2: Comparison of three runs of the simulated annealing algorithm with the same Sudoku problem to solve

A cooling rate of 0.95 has been tried and since convergence is a lot faster, it's easier to end up in local optima. Thanks to the programmed restart after a predefined number of iterations, in the following chart is possible to see how

two different runs with the same cooling rate can drastically change. In the first case the algorithm found a solution in the first iteration, while for the second case the algorithm needed five restarts.

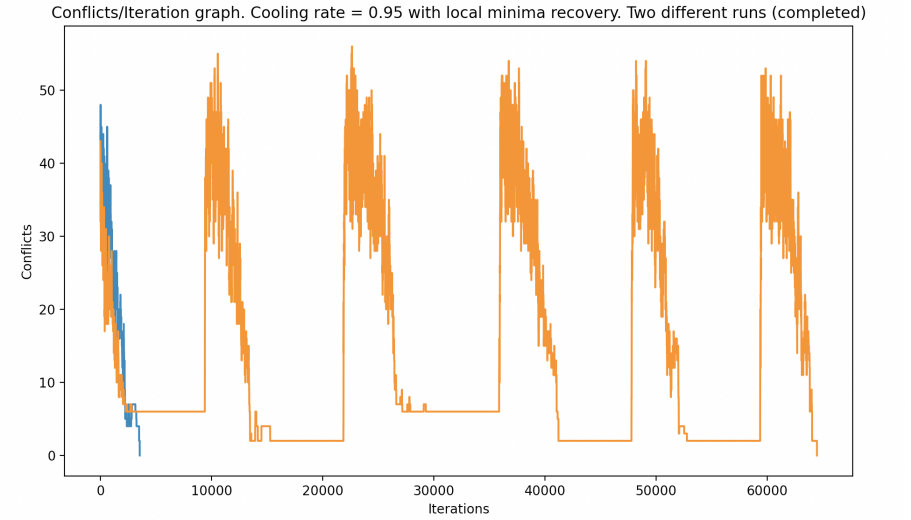


Fig. 3: Comparison of three runs of the simulated annealing algorithm with the same Sudoku problem to solve

A 16x16 Sudoku problem has been tried too but the algorithm got stuck multiple times in the same local optima.

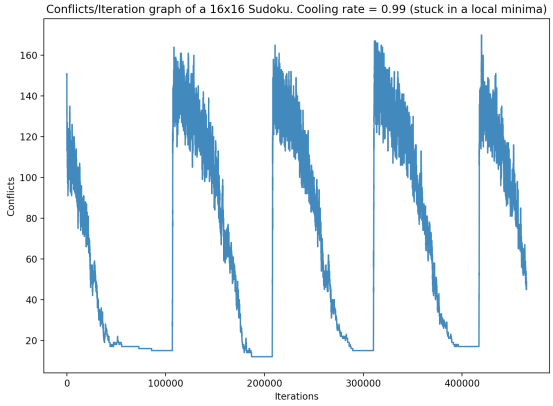


Fig. 4: Comparison of three runs of the simulated annealing algorithm with the same Sudoku problem to solve

5 Conclusion

In conclusion after extensively testing both algorithms, it's trivial that the simulated annealing process, as expected, works much better than the genetic algorithm. Even if there is room for improvement for the genetic algorithm, by tuning the parameters and trying to mix various selection method, the differences in performance and reliability from the simulated annealing algorithm is incredibly big.

A chart of comparison of two completed runtimes with solution found for the same Sudoku is shown underneath. The fitness parameter in this case is not representative since there are some slight differences in how the algorithms calculate the fitness but the time and iterations parameters are indicative and show a very big difference. The difference in performance of the two algorithms could be due to the nature of the two methods. A probabilistic approach in the solution of the Sudoku is much better than an evolutive one. It's necessary to point out that the evolutive process, in order to solve the Sudoku problem, needed a modification chance higher than the crossover, so it's possible to say that without a probabilistic focused tuning, a genetic algorithm couldn't solve a Sudoku.

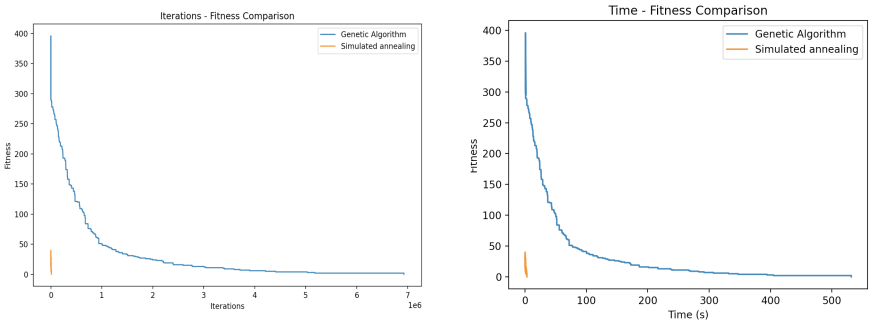


Fig. 5: Comparison of three runs of the simulated annealing algorithm with the same Sudoku problem to solve

References

1. http://fakeguido.blogspot.com/2010/05/solving-sudoku-with-genetic-algorithms.html
2. Lewis, Rhydian. (2007). Metaheuristics can solve Sudoku puzzles. J. Heuristics.