Evolutionary Algorithm for Sudoku Puzzles

Report

# Terminology

Square: little 3-by-3 square (Sudoku consists of 9 squares)

Fixed number: number given in the initial Sudoku

# 1 Solution Space and Solution Representation

## Solution Space

First I thought about an incrementing solution space. So the Sudoku fulfils the no duplicates rule all the time. Therefore, only valid incomplete Sudokus would be searched and the search space would therefore be very small. But I reckoned the mutation and crossover operators would be very hard to define. So I chose the following solution space:

The solution space just consists of totally filled in Sudokus. At first they are created using each number between 1 and 9 in each row exactly once. So there are only duplicates in the columns and the squares. This brings down the fitness at the beginning in order to save time.

## Solution Representation

My Sudoku is represented as a 9 by 9 numpy array. This is really handy because I can invoke practical functions like transpose() on it, which makes the code much nicer and cleaner.

# 2 Fitness Function

My fitness function counts the number of duplicate numbers in each column, row and square and sums them all up. So the smaller the fitness value is, the better the individual solves the puzzle.

I also included the difference to 45 in each column, row and little square. But my experiments did not show improvements because of this. So I excluded it again in order to make the important fitness function more performant.

# 3 Crossover Operator

First I tried single point, double point and multipoint crossover. Out of these, multipoint crossover worked the best. So at each not fixed number with 50% probability it chose the father’s or the mother’s number.

Then I tried another method which yielded better results: At the beginning of crossover it chooses randomly between row, column and square. Suppose it chose row, then for each row in the child it takes either the father’s or the mother’s with 50% probability.

# 4 Mutation Operator

First of all, I experimented with just mutating random numbers according to the mutation rate. But the probability for the algorithm to find the correct number to mutate is extremely small. So I changed it to the following:

First of all, I choose randomly if I want to mutate in rows, columns or squares. Suppose it chooses rows. Excluding the fixed numbers, it randomly swaps two of the remaining numbers in this row according to the mutation rate. The same can happen analogous to columns or squares.

If the fitness level is below a certain boundary (6 in my case) I am swapping duplicates which will be explained in the following:

If two duplicates in two rows are detected and two of these four numbers are in the same column it will swap these two in order to eliminate the duplicates. The same procedures can be applied for duplicates in columns respectively.

# 5 Termination criterion

My genetic algorithm terminates when the fitness level reached 0. That means the puzzle has been solved successfully.

If after a specified time no solution has been found, the algorithm terminates as well. This additional criterion has been added in order to terminate eventually, so that the algorithm does not run forever. Moreover, we do not wish a solution to take such a long time to be computed. But a problem of using the time as a termination criterion is that the algorithm could produce an entirely different amount of generations on a different machine.

At first I had a maximum number of generations as a second termination criterion but as the time used for a generation highly depends on the population size, this was not a very well suited measure for the experiments.

# 6 Code

See online hand in.

# 7 Results of the Experiments

For Details (Console Output) see Appendix.

# 8 Analysis of the Experiments

## a) What population size was best?

Population size 1000 was the best.

## b) What are the reasons for that?

10 and 100 were too small to provide enough diversity needed to come up with a solution. 10000 was too big, because only very few generations could be tried in the available time.

I think there are some similarities of a genetic algorithm with a small population size to depth-first search and one with a large population to breath-first search. Because one with a large size has a large diversity and searches lots of individuals at the same time. In contrary the one with a small size tries one small population and tries this one all the way down to a dead end.

## c) Which grid was the easiest and which the hardest to solve?

Grid1 was the hardest and Grid3 the easiest to solve.

## d) What are the reasons for that?

In Grid1 there are the least fixed numbers which opens up much more possibilities to fill in the remaining numbers.

## e) What further experiments do you think it may be useful to do and why?

The biggest difficulty I faced was the fast convergence towards local optima. And unfortunately there seem to exist lots of local optima. So I could get a nearly solved Sudoku after less than 50 generations (using population size 500) but it would just not find a solution. I tried to solve this using the following measures:

Tournament selection: Enhances the chance of individuals with bad fitness level to be selected into the mating pool. This should ensure a greater diversity.

Dead End Detection: After a given number of consecutive times where the best individual has the same fitness, I suppose that a dead end has been reached (local optimum). This knowledge is used for the following three measures:

Supermutation: If a dead end is reached, I apply a big so called “supermutation” (having a much bigger mutation rate) to all of the individuals in the population. The hope of this measure is, to get out of a local optimum and converge to another (hopefully global) optimum in the landscape.

Blacklist: In order to ensure that the algorithm does not get stuck in the same dead ends all the time, I am conducting a blacklist. So every time a dead end is reached, the best individual is stored in the blacklist. From this point of time this individual will not exist any more and will not be created anymore.

Retrievable Genetic algorithm: This is the simple version of the approach with supermutation and blacklist. So here if a dead end is reached I just start over. I discarded this idea in favour of the supermutation and blacklist approach because I hope to get to the result faster, as the algorithm only has to start from a high fitness level at the start.

Presolving: This method tries to fill in numbers at the start which are obvious. So if at a specific point in the Sudoku only one number is possible, presolving writes this number in there and adds it to the fixed numbers. This helps to reduce the search space, as there are less remaining numbers which have to be guessed by the genetic algorithm. Grid3 can even be solved just using presolving.

Evolutionary algorithms are normally used in fields where there is very little known about the problem. In Sudoku solving though there exists a lot of knowledge. In fact, most Sudokus can be solved deterministically in a very short time. Therefore, a genetic algorithm is not very well suited for this task. Because this presolving algorithm is really very basic I thought it was all right to include it, even though it already solves grid3. Otherwise if more advanced methods were used in presolving, it would render the evolutionary algorithm redundant. In my evolve() method it is very easy to turn off presolving, namely by setting the boolean parameter at the end to False.

Another idea which came to my mind was to implement a Multipopulation Genetic Algorithm. So several populations are evolving parallel to each other and then get merged together after some time. This would help to cover the whole landscape and minimize the risk of getting stuck in local optima.

I did some performance optimizations but there sure is a lot more to do. Also because of my restrictions regarding computer power I had to abort the algorithm after a relatively small amount of generations. If the necessary equipment was available, it would definitely be worth trying lots of more generations.

# Sources

1. Crossover (<https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)>, 02.03.2017)
2. Diversity (<http://www.ijetae.com/files/Volume2Issue5/IJETAE_0512_10.pdf>, 26.02.2017)
3. Local Minima (<https://www.researchgate.net/post/How_to_overcome_strong_local_minima_in_Genetic_Algorithm>, 01.03.2017)
4. Mutation (<https://en.wikipedia.org/wiki/Mutation_(genetic_algorithm)>, 02.03.2017)

# Appendix

## Experiment Output

The Termination time per run was: 1200 seconds

==================== Experiment Summary ====================

========== Population Size: 10 ==========

===== Grid 3 =====

Average Number of Generations: 1.0, Algorithm running for: 0.093s, Number of solved Sudokus: 5/5

===== Grid 2 =====

Average Number of Generations: 6397.8, Algorithm running for: 6036.141s, Number of solved Sudokus: 0/5

===== Grid 1 =====

Average Number of Generations: 2460.2, Algorithm running for: 6055.021s, Number of solved Sudokus: 0/5

========== Population Size: 100 ==========

===== Grid 3 =====

Average Number of Generations: 1.0, Algorithm running for: 34.5s, Number of solved Sudokus: 5/5

===== Grid 2 =====

Average Number of Generations: 1950.4, Algorithm running for: 6029.482s, Number of solved Sudokus: 0/5

===== Grid 1 =====

Average Number of Generations: 1576.0, Algorithm running for: 6039.656s, Number of solved Sudokus: 0/5

========== Population Size: 1000 ==========

===== Grid 3 =====

Average Number of Generations: 1.0, Algorithm running for: 50.771s, Number of solved Sudokus: 5/5

===== Grid 2 =====

Average Number of Generations: 168.2, Algorithm running for: 5498.885s, Number of solved Sudokus: 2/5

===== Grid 1 =====

Average Number of Generations: 160.0, Algorithm running for: 5782.045s, Number of solved Sudokus: 1/5

========== Population Size: 10000 ==========

===== Grid 3 =====

Average Number of Generations: 1.0, Algorithm running for: 69.645s, Number of solved Sudokus: 5/5

===== Grid 2 =====

Average Number of Generations: 126.0, Algorithm running for: 6034.896s, Number of solved Sudokus: 0/5

===== Grid 1 =====

Average Number of Generations: 131.4, Algorithm running for: 6074.625s, Number of solved Sudokus: 0/5

==================== End of Experiment ====================