João Bolas Soares
MIEET
n40651

# Evolutionary Computing

*"Sudoku solving Genetic Algorithm based on Latin Square"*

# Index

# Introduction

This report is about the project i made for the evolutionary computing subject where the purpose was to solve a Sudoku grid with NxN size through a pure Evolutionary Algorithm. The problem is based on the Latin Square problem which is a NP-Complete problem due to his growth in possibilities.

This Latin Square problems is about the position of several colours on the same square with out having the same colour on the same row or column.

| | | |
|---|---|---|
| A | B | C |
| C | A | B |
| B | C | A |

Because of this property we can compare this Latin Square problem to a more recent game of Sudoku, increasing the difficulty with the size of the grid. It's considered NP-Problem to find if a partially filled Latin Squared is solvable and considered GA-Hard to actually solve it with a Genetic Algorithm. This is because of the huge number of combinations possible and only a few are considered correct. To express this number of combinations in a Mathematical expression we can use this:

$$\prod_{k=1}^{n} (k!)^{n/k} \geq L(n) \geq \frac{(n!)^{2n}}{n^{n^2}}$$

So in a table we can easily relate the size n of the size of the side of the latin squares with the number of possibilities.

| n | all Latin squares of size n |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 12 |
| 4 | 576 |

| | |
|---|---|
| 5 | 161280 |
| 6 | 812851200 |
| 7 | 61479419904000 |
| 8 | 108776032459082956800 |
| 9 | 5524751496156892842531225600 |

Specifically in this case we are trying to solve a Sudoku with some fixed numbers, so we introduced the concept of difficulty incremented to this problem. If we start with an empty Sudoku matrix the algorithm will converge fast because it has no external rules to obey or if we have the starting matrix almost filled we will have less possibilities to reach the correct answer making the difficulty on this problem vary  with has many fixed numbers we have on the starting matrix.

Based on this perception of the problem we can say that it's more difficult to reach the correct answer with semi-solved starting matrix. For this reason the GA created will try to solve Sudokus created with one third of their matrix already solved. The entire GA was built from zero although some ideas were discussed with the Professor and some of my colleagues and later confirmed with an external paper with had done a similar work in this area.

On this algorithm we are using 1 fixed number for each subMatrix due to some problems creating a valid Sudoku Grid although if a sudoku grid with several elements was used as input the algorithm would find the answer.

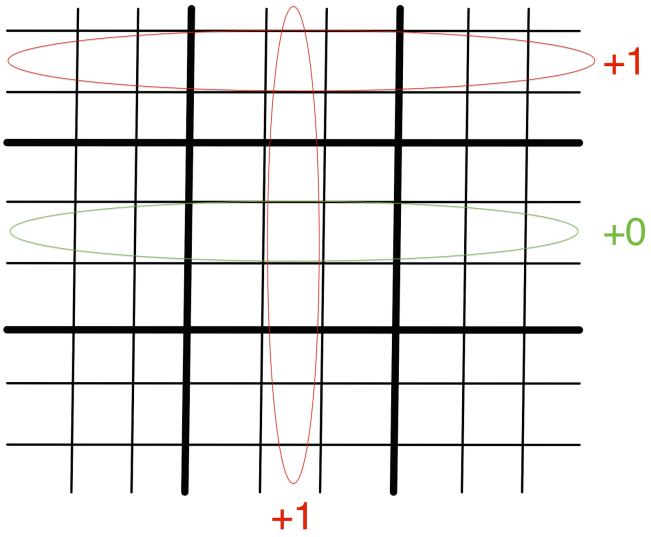
# Methodology

To build this GA very simple concepts were used has a selection based on best fitness , a crossover adapted to the idea of manipulating a matrix has the mutation and the number of times that a mutation applies to a structure.

This algorithm was built with an elitist influence due to the fact that when the algorithm starts to converge we don't want to lose the best answer to the huge possibilities we are working with.

## The Fitness

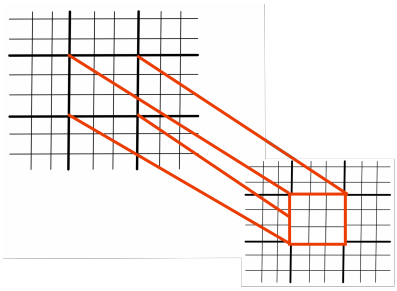
The fitness is usually the secret to the GA's.In this problem i considered the fitness the number of lines or columns wrong being this a minimizing problem, when fitness is "0" we have the correct answer to the sudoku Grid.




The green circles represent the contribution of the correct line to the fitness and the red circles represent the contribution of a wrong line to the fitness. This will make the fitness for a 9x9 Sudoku vary from 18 to 0 being 0 the best score possible.

## The CrossOver

The crossover was applied here as a translation of subGrids from one Sudoku Grid to another sudoku Grid. Only sub Grids on the same position can be changed avoiding nullifying the main sudoku grid keeping intact the game structure.

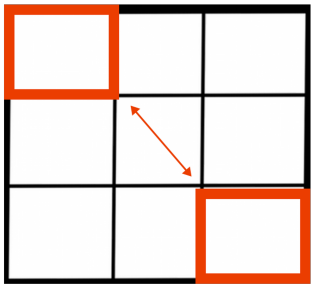


So by using this crossover we get a genetic variability across all generations making a smoother and faster convergence to the correct answer.

## The Mutation

In this algorithm we use the mutation has a drastic genetic modifier. The mutation itself in this algorithm is a change in two numbers inside a sub Grid, making it a controlled alteration although this alteration is random and the only possibility to the alteration does not happen is the number being changed with a fixed number.

Although this is a very basic process , this process can be applied multiple times on the same generation to the same sub Grid has it will be explained in the report.

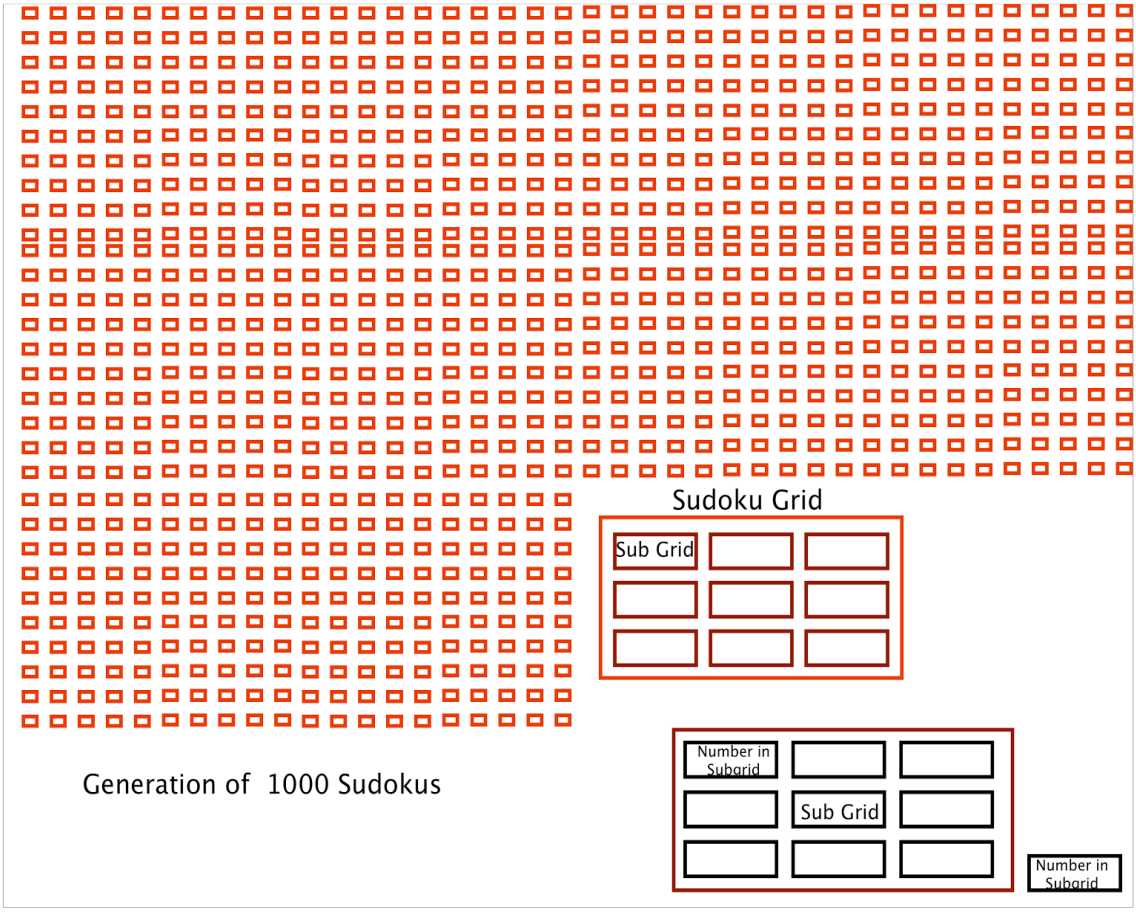


# Implementation

Knowing how all the rules of genetics are applied to this algorithm (crossover and mutation) it's time to explain the selection process and this was implemented.

The algorithm was built on Java for better understanding making it more versatile. I have used 4 classes, a main class , a sudokuGrid class to build the grid, a subGrid class to which are components of the sudokuGrid, and finally a numberSubGrid class which make up the subGrid.

For organization the entire code was structured on ArrayList's. Since the generations until the numbers in the subGrid which means alot of objects being created and manipulated in the program. For an idea on how many objects we have in a generation i made a small comparison.




Creating more than 81000 objects per generation of 1000 Sudokus.

## Selection

For the implementation of selection i had to make some minor tweaks to the standard selection.

Because in this algorithm we are looking for a correct combination amongst 552475149615689284253122560 combinations i stated a elitism criteria where a percentage of the population would always saved from change making the algorithm maintain the best fitness even with high probabilities of mutations and crossovers.

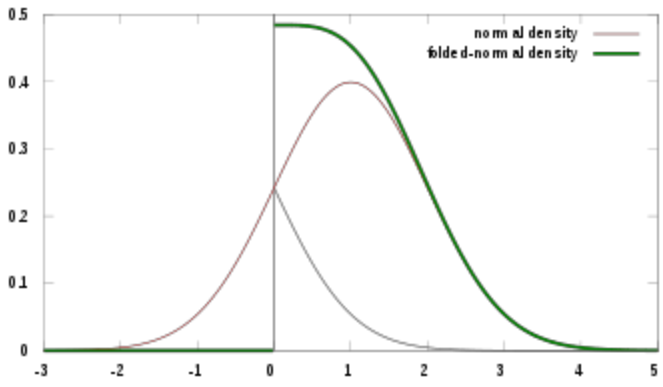
The best percentage i achieve through trial and error was 15% enhancing the convergence of the population with the other 85%. Although i felt the need to change this percentage when reaching for the perfect fitness combination so i implemented a change from 15% to 50% elitism when the algorithm reaches ¼ of the starting fitness, by making this i make some pressure on the algorithm much more focus on improving those 50% with mutations instead of looking for different solutions.

The elitist part of population is also privileged on the mutation with their own set of rules. For example a elite member goes through the same mutation process as a normal member of population but , he only replaces the elite member if his fitness is equal or inferior.

The selection itself starts from the first generation being entirely random, after that we sort the population by fitness to generate the next generation we make a crossover from the best half of the population alternating each of them and making them do crossovers with another random element of the population, with this method we create 25% of the next generation.

Using crossovers this way we can mix solutions with great fitness with solutions which are not that good and creating genetic variability.

After the crossovers we apply the mutations, mutations have a probability of being very drastic. A basic mutation was explained previously on this report but in the selection process the mutation algorithm implemented was more complex. The algorithm was based on a probability of mutating x number of Sub Grids y times. This probability is biased and its based on the folded normal Distribution adapted to discrete numbers to allow all the subgrids




mutate on the same generation and allowing them to mutate has many times has there are numbers inside them. This method brings increased genetic variability to the algorithm creating a bigger convergence when put together with the elitism factor.

The console should print several output's to let the user know how close the algorithm is to reach the final solution for example:

"Generation: 495 With the current Fitness of: 2.0 And Average Fitness: 10.033 The best Score till now was: 2"

And this prints should be followed by the answer when the "Current Fitness" reaches "0".

In the case of the algorithm does not find the right answer the algorithm will restart from a random population because the algorithm might have reached a local minimum that he can't leave by himself so he resets the population and begins to solve it again.

A complete output should look like this:

```
Validating ... a 9by9 Sudoku Matrix
Validated
4 0 0  0 0 0  0 0 2
0 0 0  0 0 0  0 0 0
0 0 0  2 0 0  0 0 0

0 0 0  0 9 0  0 6 0
0 0 0  0 0 0  0 0 0
0 0 9  0 0 0  0 0 0

0 0 1  0 0 6  0 0 0
0 0 0  0 0 0  1 0 0
0 0 0  0 0 0  0 0 0

---------------------------------
Generation: 0 With the current Fitness of: 17.0 And Average Fitness: 17.948 The best Score till now was: 17
Generation: 1 With the current Fitness of: 16.0 And Average Fitness: 17.934 The best Score till now was: 16
Generation: 5 With the current Fitness of: 15.0 And Average Fitness: 17.832 The best Score till now was: 15
Generation: 12 With the current Fitness of: 14.0 And Average Fitness: 17.418 The best Score till now was: 14
Generation: 21 With the current Fitness of: 13.0 And Average Fitness: 16.629 The best Score till now was: 13
Generation: 34 With the current Fitness of: 12.0 And Average Fitness: 15.572 The best Score till now was: 12
Generation: 47 With the current Fitness of: 10.0 And Average Fitness: 14.814 The best Score till now was: 10
Generation: 86 With the current Fitness of: 9.0 And Average Fitness: 13.385 The best Score till now was: 9
Generation: 112 With the current Fitness of: 8.0 And Average Fitness: 12.921 The best Score till now was: 8
Generation: 166 With the current Fitness of: 7.0 And Average Fitness: 12.191 The best Score till now was: 7
Generation: 252 With the current Fitness of: 6.0 And Average Fitness: 11.774 The best Score till now was: 6
Generation: 3044 With the current Fitness of: 4.0 And Average Fitness: 11.309 The best Score till now was: 4
Generation: 5844 With the current Fitness of: 2.0 And Average Fitness: 9.989 The best Score till now was: 2
Generation: 11832 With the current Fitness of: 0.0 And Average Fitness: 3.743 The best Score till now was: 0
The Solution for the Starting 9by9 is the following, achieved in :11832 Generations
4 9 3  8 6 5  7 1 2
2 5 6  7 1 9  3 4 8
1 8 7  2 4 3  6 9 5

8 3 2  4 9 7  5 6 1
5 1 4  6 3 2  8 7 9
6 7 9  1 5 8  2 3 4

7 4 1  5 8 6  9 2 3
3 6 5  9 2 4  1 8 7
9 2 8  3 7 1  4 5 6
```

```
---------------------------------
```

## Algorithm Starting Parameters

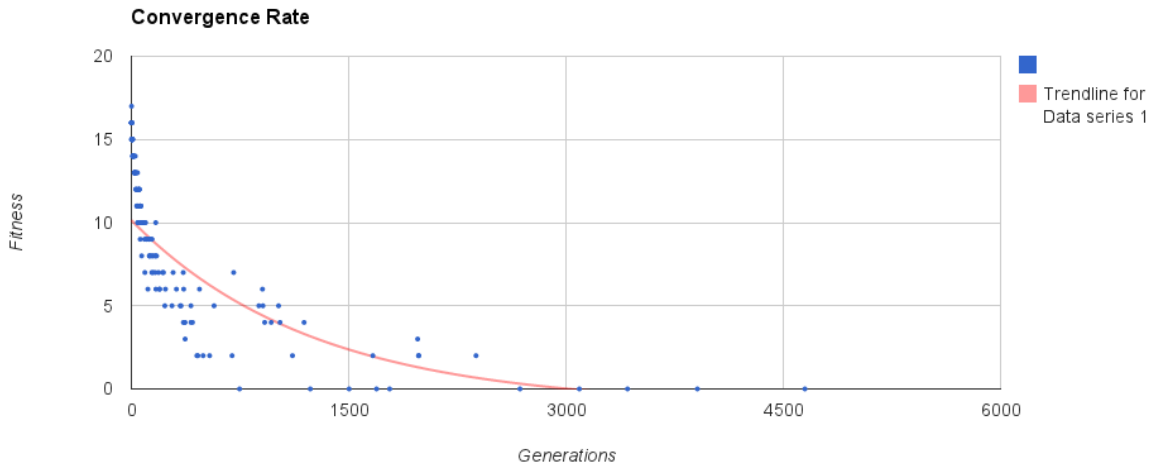
The algorithm can be tinkered for the user's preference for the size of the sudoku but the size of the sudoku must always be perfect squares. Making this a versatile NxN GA sudoku solver, although with the increase of the size of the sudoku matrix it's good practice to increase size of population to increase convergence and that is going to consume more memory. At some point the performance of the algorithm will be dependent of the resources available.

Number of generations is also a starting parameter default is 99000 because it's a number that is not going to be reached and the algorithm can work his way. But this parameter can be easily changed to the user suggestion.

And the "genThreshold" parameter which is the threshold which the user understands that the algorithm should reset after x generations with the same fitness(generally this is associated with local minimum).

## Result Analysis

Doing an analysis on the results we can see on the following chart that the algorithm usually converges before the 4000 generation we can see from the exponential trending line that the difficulty to achieve a "0" fitness is increased comparing to other fitness achievements.




Although this difficulty logically increases with a decreasing fitness the difficulty of reaching “0” is greater due to the fact of the last step is always from “2” to “0”. It’s not possible to simply have one wrong line or wrong column giving you a fitness of “1”. So to make this 2 points in fitness jump we have an increased difficulty in probability. To evaluate the performance of this algorithm i'm going to give it a score based on:

$$Score = Number\ Of\ Generations\ X\ Population\ X\ Number\ of\ times\ Fitness\ is\ calculated$$

Following this rule using my average number of generations i got 5948000.

I believe we have achieve success in calculating the Sudoku with a GA, getting the correct answer and being the algorithm capable of any NxN Sudoku solving.