Hybrid GA Sudoku solver

Created by Fareed Hanna Sokar

INDEX

Hybrid GA Sudoku solver

1. Problem:

1.1. Problem description:

Sudoku is a game of 9 sub-squares table (9x9 2d array) and every sub-square is in the size of 3x3, in the start of the game a few numbers is given.
We need to fill numbers between 1-9 according to certain rules (4 rules in particular):

a. Each row must have one of each number – **and only one.**
b. Each column must have one of each number.
c. Each sub-square must have one of each number.
d. The given numbers (clues/starter numbers) must remain the same – **in other words we mustn't change their place or values.**

1.2. Important values:

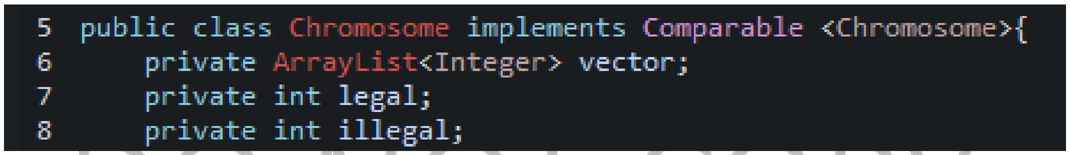
a. **Minimum number of given clues:** according to "Wikipedia" for a sudoku game to have a solution (and only one) we need at least **17** clues.
b. **Mutation rate:** we need a mutation rate which will allow us to have a variety of solutions (possible ones), in case we reached a local maximum/minimum we can cross to a global one. We are talking about a sudoku game with only one possible solution so will have a lot of local points. I chose the rate of **12%.**
c. **Cross-over points:** I wanted to use the cross-over method to enforce the "forth rule" (explained at) so I used my cross-over points to tell me **where in my "chromosome" each row ends.**
d. **Population size:** a small size will help with fast calculations but because of my random initializing values of this population we may not reach a fitting base vector and choosing a really large number will slow the algorithm. So, I picked a fitting number for an average PC processing unit which is **500**.

2. Solution: (Implemented in JAVA).

At first, I thought of using a regular GA with random (random numbers with redundancy) crossover and mutation functions however using such method eventually will reach a solution but after a lot of time, and we know in our world “Time” is the coin of “High-Tech”.

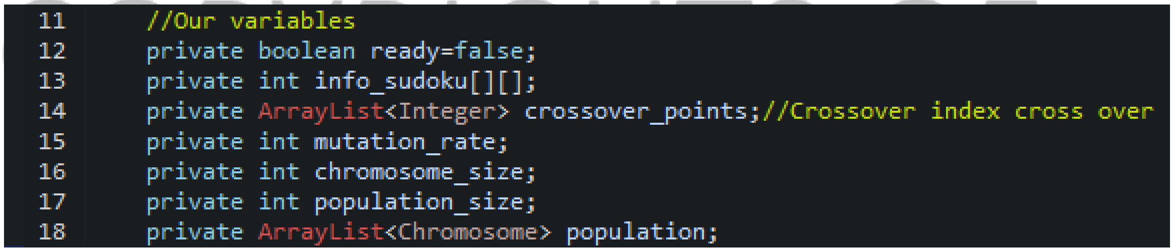
So, I created my algorithm to enforce each one of the sudoku rules:

a. “The given numbers must remain the same” – to insure the “rightness” of this rule I created my workspace in a way in which a Class by the name “Chromosome” created with an ArrayList<Integer> with the changeable values within it.



```java
public class Chromosome implements Comparable <Chromosome>{
    private ArrayList<Integer> vector;
    private int legal;
    private int illegal;
```

and to make sure my given numbers are saved, I used a 2d array (9x9) within the “Population” class and the given numbers are shown in their location and the other changeable location a zero is shown (**int info_sudoku[][]**).

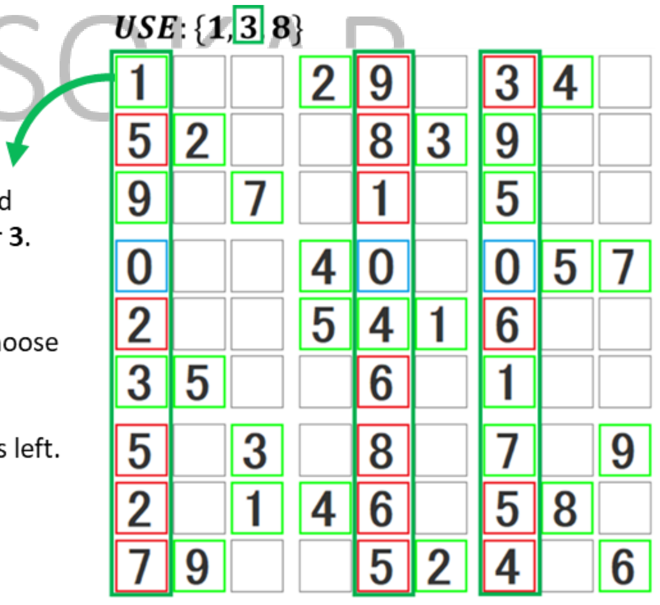


```java
    //Our variables
    private boolean ready=false;
    private int info_sudoku[][];
    private ArrayList<Integer> crossover_points;//Crossover index cross over
    private int mutation_rate;
    private int chromosome_size;
    private int population_size;
    private ArrayList<Chromosome> population;
```

b. “Each row must have one of each number”- when initializing our population, I made sure that the rows of each Chromosome is according to this rule, when filling random values in each row I made sure no redundancy of numbers in the same row.

c. “Each column must have one of each number” – I used the mutation function to impower this rule in case a mutation “Accuried” :
Algorithm -
First pick a random number [1,8] – the number of rows which it’s items will be rearranged – I gave it the symbol ∂ , we pick random rows as the number ∂ without redundancy – used a Boolean array in the size of 9 – now we do the following steps for each row (PS: after changing a row the next iteration will be with the new changed chromosome after changing the row):
A. we go through the row changeable values and check if there is the same value in the column:
A.A. if true then save the number in a structure and change the value in the chromosome to zero.
A.B. if false keep it and don’t change.
B. we go back to the start of the row and check every place a zero is shown:
B.A. if there is a number which we can for sure put it there then remove it from the structure and put it in the row (not shown in this column but shown in the other zero columns).
C. after doing so there may be some values left so choose randomly one of the zero columns which we can put that value there and after doing so if there is values left without legal location fill them randomly.

Example:



- In this example in the first column and the third according to our rules, we can’t put the **number 3**. however in the second column we can.

- The **number 8** we can but it in both thus we choose randomly in one of them.

- The **number 1** we put it in the location which is left.

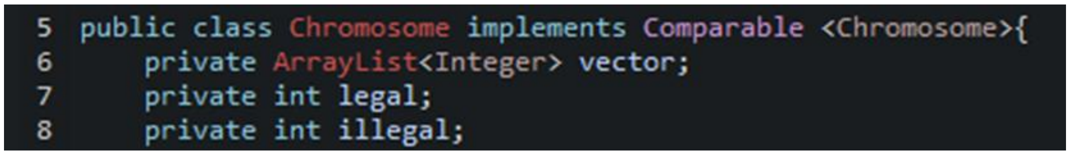
d. “Each sub-square must have one of each number”- I enforced this rule using my cross-over function in a way where we use a “Base Chromosome” and one of the two parents we chose (usually the **“Base Chromosome”** is the one with the best **fitness** function):

First pick a random number [1,8] – the number of rows which it’s items will be rearranged – I gave it the symbol ∂ , we pick random rows as the number ∂ without redundancy, now we use the **“Base Chromosome”** to build the offspring with changing the choosing rows to the ones of the parent while doing some changes – the change will be the same as in the mutation function but sub-square wise.

3. Details about the Algorithm:

3.1. The Fitness function:

in each chromosome there is two variables:



```
public class Chromosome implements Comparable <Chromosome>{
    private ArrayList<Integer> vector;
    private int legal;
    private int illegal;
```

the variable illegal is initialized to zero and for each item (cell) in the chromosome we add a point for each one of the rules 1-3 which is not satisfied, when an item (cell) satisfy the three rules we add a point to the legal variable, the fitness function will be:

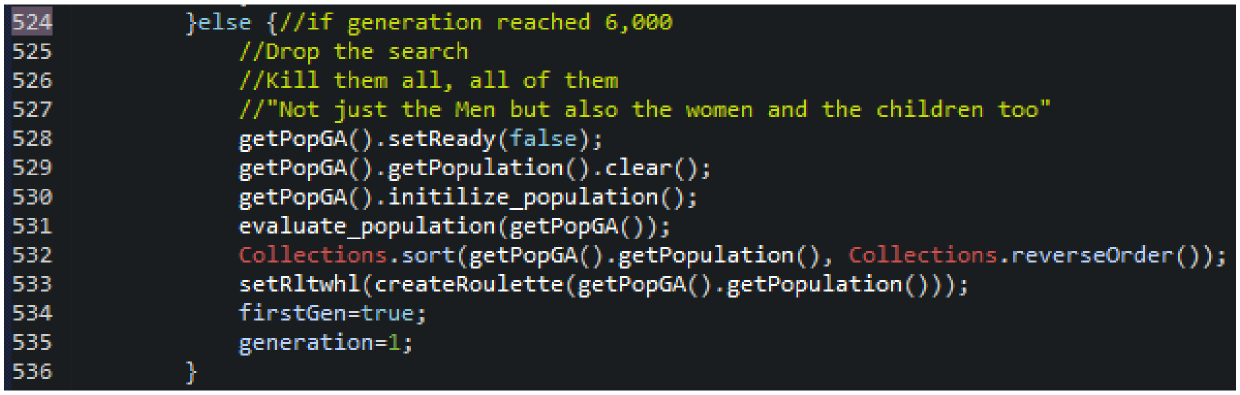
$$fitness(x) = x.legal - x.illegal$$

3.2. When to "kill" our population?

to get the solution to a sudoku we need a really big number of possible solution (Chromosomes in our population) I assumed that a regular PC can make $3 * 10^6$ calculations, I did not want a really big number because it will slow the whole total process (a lot of personal computers cannot endure such big numbers). And taking small group will not ensure us to get a solution as fast as possible (unless we had a stroke of luck).

we chose our population size to be 500, thus we need to kill our population after $\frac{3*10^6}{500} = \mathbf{6000}$ **generations.**

if we didn't reach a solution after 6000 generations we eliminate our current population and initialize it all over again, also with each time we check the status of our population if we are really near an answer or not if so we try to change the base to a different one.



```
        }else {//if generation reached 6,000
            //Drop the search
            //Kill them all, all of them
            //"Not just the Men but also the women and the children too"
            getPopGA().setReady(false);
            getPopGA().getPopulation().clear();
            getPopGA().initilize_population();
            evaluate_population(getPopGA());
            Collections.sort(getPopGA().getPopulation(), Collections.reverseOrder());
            setRltwhl(createRoulette(getPopGA().getPopulation()));
            firstGen=true;
            generation=1;
        }
```

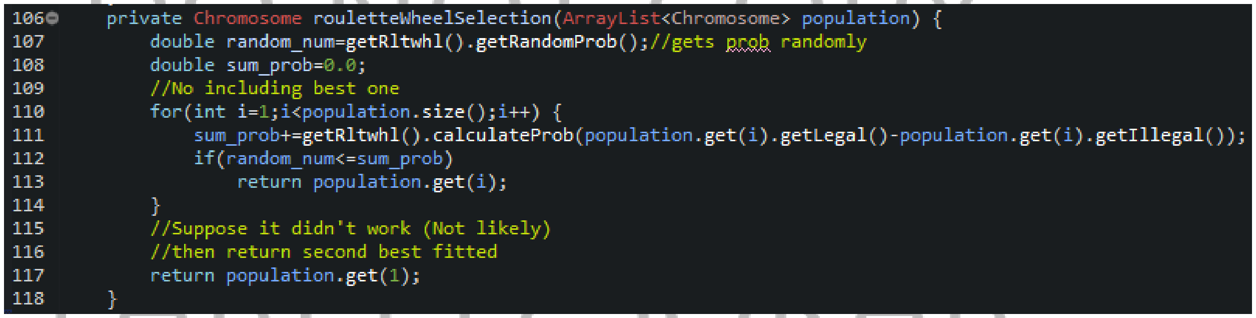
3.3. Building our "Roulette Wheel":

First I sum up all of our population fitness (but the best fittest chromosome) and also save the minimum and the maximum fitness. Building the roulette wheel will be the same as regular one but the only difference will be that the fitness function can have three values positive, negative and zero, thus calculating the probability for a chromosome will be as following:

$$\text{calculation}\ (\mathrm{X}) = \begin{cases} \dfrac{\mathrm{x} - \text{Min fitness} - 1}{\text{Sum} - ((|\text{Population}| - 1) * \text{Min fitness})}, \text{Min fitness} < 0 \\ \dfrac{\mathrm{x} + \text{Min fitness}}{\text{Sum} + ((|\text{Population}| - 1) * \text{Min fitness})}, \text{Min fitness} > 0 \\ \dfrac{\mathrm{x} + 1}{\text{Sum} + ((|\text{Population}| - 1) * 1)}, \text{Min fitness} = 0 \end{cases}$$

this way the sum of the probabilities are 1.0 like it should, and each chromosome has a probability different from zero.

3.3.1. Selection function: like a regular selection function however we do not take the base in account.



```
    private Chromosome rouletteWheelSelection(ArrayList<Chromosome> population) {
        double random_num=getRltwhl().getRandomProb();//gets prob randomly
        double sum_prob=0.0;
        //No including best one
        for(int i=1;i<population.size();i++) {
            sum_prob+=getRltwhl().calculateProb(population.get(i).getLegal()-population.get(i).getIllegal());
            if(random_num<=sum_prob)
                return population.get(i);
        }
        //Suppose it didn't work (Not likely)
        //then return second best fitted
        return population.get(1);
    }
```

3.4. Pseudo code-Algorithm:

```
A. Validation of inputs.
B. Initializing of our population according to 2.B.
C. Evaluate the population.
D. Do (do-while loop):
    D.A. Build our Roulette Wheel using current population.
    D.B. if not first time check the status of the population compared to the
         best solution.
         D.B.A. if the population is not going near the solution (the rules
                explained in 3.2.) then kill and start over.
         D.B.B. Continue normally  or change the base.
    D.C. For loop [population-size/2]-times:
         D.C.A. Choose two parents using the roulette wheel make sure they
                are not the same parents.
         D.C.B. Do with each parent the cross-over function in 2.D.
         D.C.C. Check for both offspring if a mutation occurred if so use the
                mutation function in 2.C.
         D.C.D. Evaluate the two new chromosomes.
    D.D. Now we have a pool of 2*population-size, first we rearrange our
         population from the chromosome with the greatest fitness to the
         least and we choose the best "population-size" chromosomes to be
         our new population.
    D.E. change the base chromosome to the new best fit chromosome.
E. While best fitness chromosome is not equal to the size of the
chromosome then go to D.
```

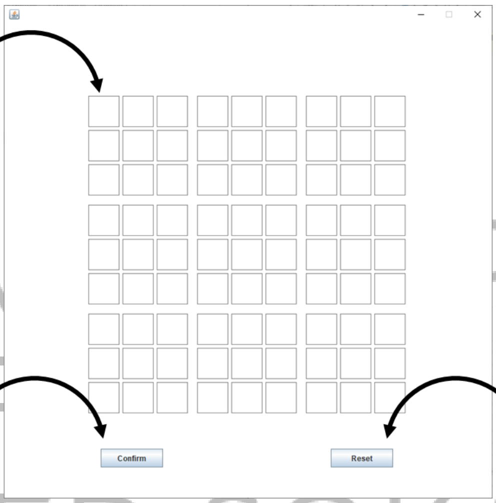
4. How to run:

4.1.

Click the runnable JAR file to start the App.

4.2.

To fill in values click on one of the cells.

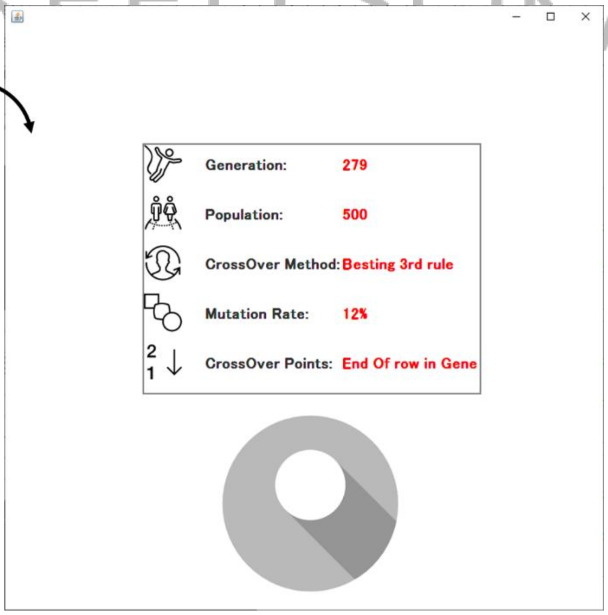


After entering all values click on Confirm start the process.

Click on Reset will delete all values.

4.3.

Wait until the process is finished.



Generation: 279

Population: 500

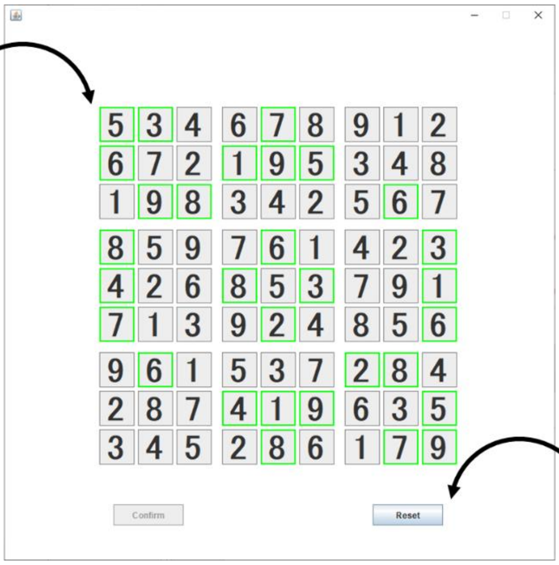
CrossOver Method: Besting 3rd rule

Mutation Rate: 12%

CrossOver Points: End Of row in Gene

4.4.

Final solution

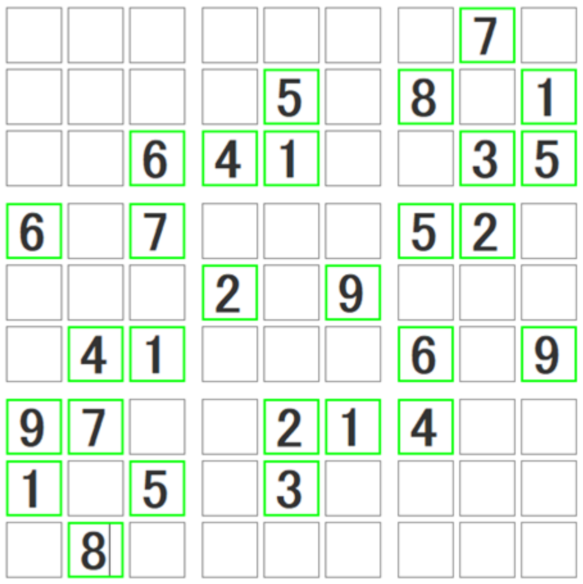


5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

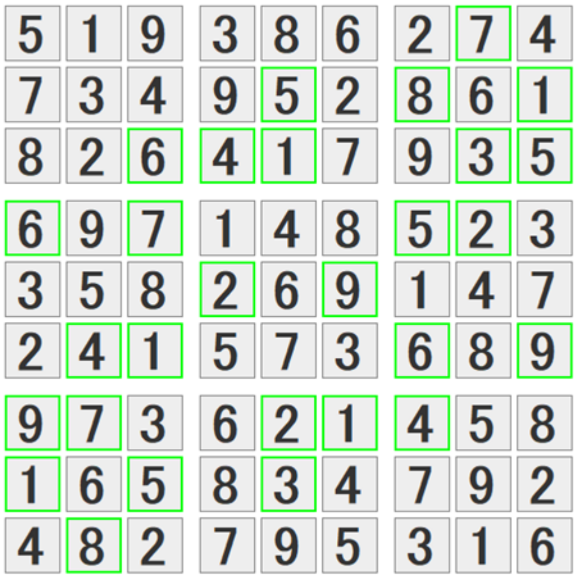
Click on reset will start a new empty sudoku.

5. Examples:

Easy: After 5576 Generations we got an answer – Time: 48 Seconds

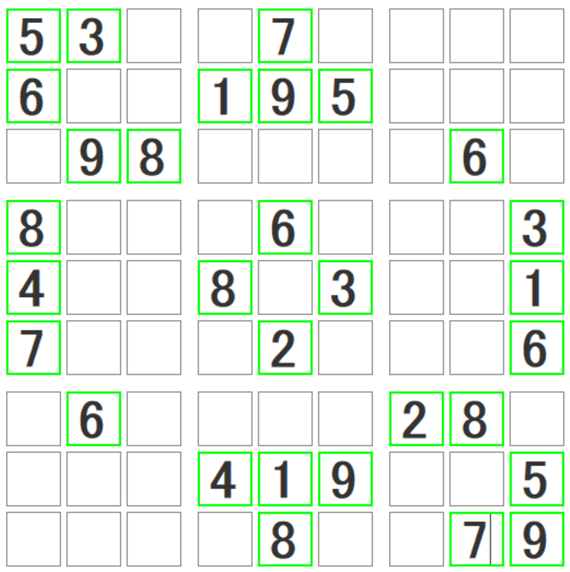


							7	
				5		8		1
		6	4	1			3	5
6		7				5	2	
			2		9			
	4	1				6		9
9	7			2	1	4		
1		5		3				
	8							

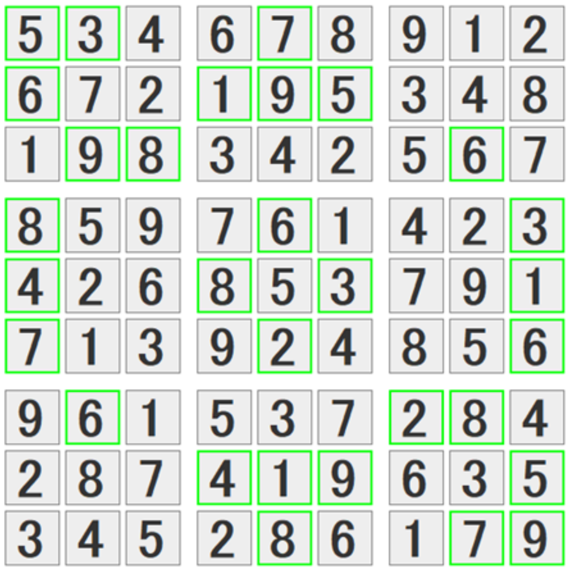


5	1	9	3	8	6	2	7	4
7	3	4	9	5	2	8	6	1
8	2	6	4	1	7	9	3	5
6	9	7	1	4	8	5	2	3
3	5	8	2	6	9	1	4	7
2	4	1	5	7	3	6	8	9
9	7	3	6	2	1	4	5	8
1	6	5	8	3	4	7	9	2
4	8	2	7	9	5	3	1	6

Medium: After 386 Generations we got an answer – Time: ~10 Seconds

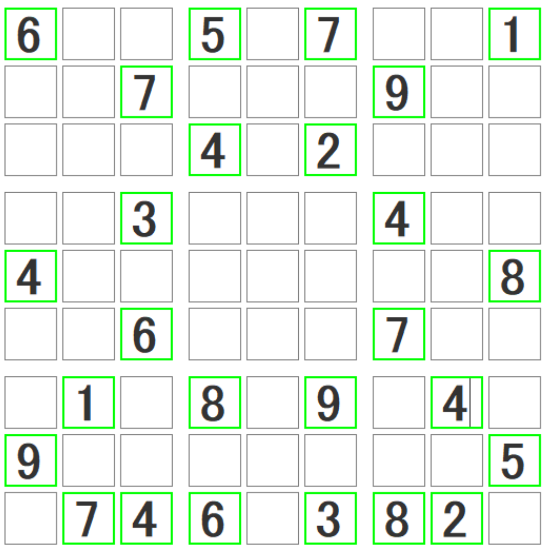


5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

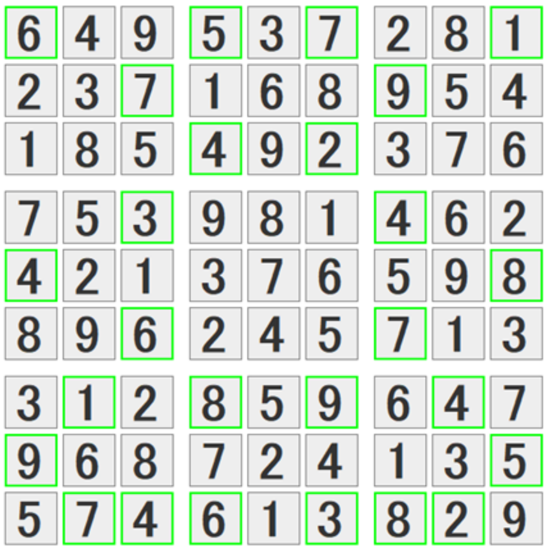


5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Hard: After 107222 Generations we got an answer – Time: 17 Minutes and 45 Seconds



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



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