Project Report

Implement of Generating & Solving Sudoku Game Through AI Methods

Han Wu: wu.han1@husky.neu.edu
Jianyu Hu: hu.jiany@husky.neu.edu

Abstract

This project is built by HAN WU and JIANYU HU. The project contains generating the Sudoku Game & Problem and different artificial game solving algorithms. What we can see in this project is a complete process of generating and solving Sudoku Game.

I. Introduction and Background

Sudoku is a logic-based combinatorial number-placement puzzle. The objective is to generate a 9*9 matrix as a sudoku matrix which has some blanks. It can be divided into 9 sub-matrixes (3*3, 3^2=9). Each sub-matrix can only have number from 1 to 9 once, at the same time each column and each row can only exist one number once (1-9). The input of the algorithm should be a 9*9 incomplete matrix containing blanks (0). The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution. Completed games are always an example of a Latin square which include an additional constraint on the contents of individual regions.

This project is solving the Sudoku problem we generated through an efficient and smart method. The output would be a complete matrix satisfying the game rule. The running cost would be shown as well. The data we input can be a seed matrix which doesn't need to satisfy the Sudoku rule, but the interesting point is we will generate the actual (solution exists) Sudoku matrix for the algorithm.

We chose this topic because we found the existing sudoku solutions are limited to several types. People prefer to use general Only-One Method, Greedy DFS/BFS or few Back Tracing as the guide to solve the problem. However, according to our research, if the sudoku are more complicated (different number of blanks) these solutions would be a waste of time and space. In all, we would like to use the Artificial Intelligence we learned to improve the algorithm, at the same time, we also want to try some new ideas related to AI as one direction to solve the problem

II. Related Work

Exclusion method: Use the number to find the only fillable space in the unit, called exclusion method.

For example, the number 9 has appeared in the first row and the second row, then the 9 in the third row must not be in the palace grid (small squares of 9 grids) that appeared in the first 2 rows, so only A 9 in the third row of a square will appear. If there is only one vacant row in that square, then 9 will be determined. This algorithm is different from our algorithm, at this time, there may be more than one vacant row, column, and square. These two methods are actually solved using different dimensions, one from the grid dimension and one from the digital dimension, which are complementary to each other. After one method cannot continue to solve, you can try the second method, it is possible to solve.

we choose the number 1 to explain: For the number 1, traverse each grid of each row. If there is no 1 in the row, then we will check the possible numbers of the remaining grids. If only one of the remaining squares may be 1, then 1 is confirmed.

Unique method: Use the grid to find the unique fillable number, called the unique method, and the unique fillable number in the grid is called the only residual solution. For example, 8 of 9 rows in a row are filled, and the remaining one is undoubtedly determined. My approach here is to traverse the

spaces in turn, that is, each unsolved grid. Check the rows, columns, and small squares where it is located, and remove the impossible numbers. For example, there are already 2, 5, 7 in the row, then this grid will definitely not be one of these 3 numbers. After those culling, there is only one possible number grid, remove operation, and update the matrix.

III. Problem Formulation and Solving

1. Problem Constraints

Constraint 1. Each line uses 1 to 9 once, no limit on positions

Constraint 2. Each column uses number 1 to 9 once, and the position is not limited

Constraint 3. Every 3 × 3 small palace uses 1 to 9 in any position

2. Depth_first_search, Back tracing and Pruning

2.1 Algorithm design

Obliviously, Sudoku can be seen as a CSP problem that is why we choose the back tracing, for our opinion is find the solution to the terminal state(all the blanks are filled) so DFS or BFS both work, but consider the cost we choose DFS + Back Tracing. The algorithm would be traverse all the blanks through DFS and check whether it meets the Sudoku Rule, if traverse a not-real Sudoku, trace back. After that we will apply the pruning algorithm with the back tracing to improve the efficiency. However, if we use pure Back Tracing when traverse a blank cell, it would try numbers from 1-9 in turn and each time you fill in a number in the blank box, you need to judge whether the current Sudoku is valid. This is a waste of time. Therefore, we prefer to improve the algorithm with Pruning to prove the game constraint (Determine whether the row or column to which the blank cell belongs and the n*n N-square cell already appear in this number. If it has already occurred, skip it directly) at first, this will decrease the constraint domain which means we will save a lot of time cost.

2.2 Algorithm Implement

Original Algorithm:

1. Set the mark as true where there is a number

2. Loop through the matrix where there is no mark true, that is the places that need to be filled in. If the current is 0, judge the current nine-grid grid, and then assign the number from 1 into the blank, and check whether the number assigned being used once in the row, column, and sub matrix; if the depth traversal is true return true. If it is not satisfied, the next number will be used to be assigned, until 1-9 is judged to be unsatisfied, it will return false, and it will be traced back to the previous layer. If there is no 0 at present, it means that they are filled and meet the unique conditions, end.

Algorithm Improvement:

Using three boolean two-dimensional arrays l [] [], c [] [], s [] [],

l [i] [j] indicates whether the number j + 1 has appeared in the row i

c [i] [j] indicates whether the number j + 1 has appeared in the column i

s [i] [j] indicates whether the number j + 1 has appeared in the number i 3x3 nine-square sub matrix

And every time a blank cell is encountered, before trying the numbers 1-9 in sequence, determine whether the number has already appeared in the row, column and the 3x3 nine-grid cell the blank cell belongs. If it has appeared, skip it (pruning) directly. It can ensure that the Sudoku after each entry of the number is valid, no need to judge.

3. Genetic Algorithms

3.1 Algorithm Design

The second algorithm we want to design is based on Genetic Algorithms which is an interesting method to solve Sudoku puzzles. Genetic algorithm is a search heuristic that uses and implements the basic concepts of Charles Darwin's theory of natural evolution. First, we need to create a vector of random solutions (in Sudoku puzzles, the vector should be integers from 1-9). This is called initial populations. Second step pick a few solutions and rank them according to fitness then replace the worst solution with a new solution, which is either a copy of the best solution or an entirely new randomized solution, a mutation (perturbation) of the best solution, even a cross between the two best solutions. Moreover, fitness is a measure of how good a solution is, lower meaning better, thus we use the fitness function to give the individual

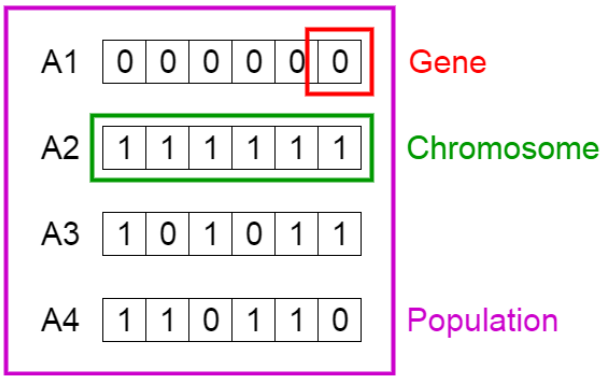
solution a score. In the problem of Sudoku, the lower the fitness function the better is the solution. Third step is to check if you have a new global best fitness, if so, store the solution. Finally, if too many iterations go by without improvement, the entire population could be stuck in a local minimum. If so, throw away all solutions and start over at the first step. All above is the overview of genetic algorithms, for details we need to design functions for fitness, crossover, mutation, selection (best and random). Genetic algorithm is a brand-new algorithm we will study out of the course, so it will be an interesting experience to use it solve sudoku puzzles.

3.2 Algorithm Implement

First, the pseudocode lists as following:

START
Generate the initial population
Compute fitness
REPEAT
    Selection
    Crossover
    Mutation
    Compute fitness
UNTIL population has converged
STOP

For the initial population, it means the process of solving a Sudoku puzzle begins with generating an initial set of candidate solutions. Each individual is a solution to the problem you want to solve, and each individual in the population is characterized by a set of parameters which we call genes, such as initial configuration of the Sudoku grid, a generation, which represents current state of the individual in one dimensional array, and the fitness value of the individual. Genes are joined into a string to form a Chromosome (solution), the figure following is a clear example explains relationships between genes, chromosome, and population.



Population, Chromosomes and Genes

The first generation was initialized by assigning a random number to the cells of the Sudoku grid, which were absent in the initial problem.

Then we talk about the fitness function, the fitness function determines how fit an individual is. It gives a fitness score to individuals. In the problem of Sudoku, the lower the fitness function the better is the individual. We calculate the following values for each cell in the generation:

rowError: determines the number of repeated values in the same row

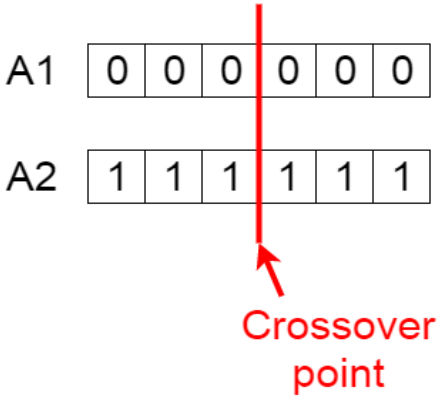
colError: determines the number of repeated values in the same column

boxError: determines the number of repeated values in the same box (where box means on of the smaller squares of the Sudoku grid)

For each error in any of the cases the fitness value of the generation is incremented by one. Moreover, in the case where the prefilled cells of the initial problem do not match with the cell values of the current generation, 1000 will be added to the fitness value in order to completely ignore that state in further iterations.

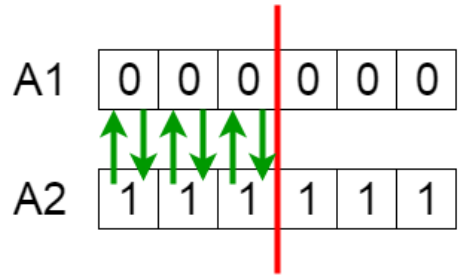
For operations in the loop of pseudocode, Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes.

For example, consider the crossover point to be 3 as shown below:



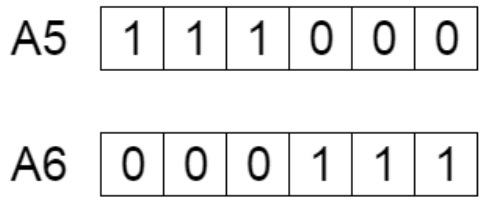
Crossover point

Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached.



Exchanging genes among parents

The new offspring are added to the population.



New offspring

Mutation means, each individual in the population is subject to a mutation. The algorithm randomly selects a cell in the generation to which it assigns a new random valid value.

Selection phase is to select the fittest individuals and let them pass their genes to the next generation. It appears after the stages of crossover and mutation. The selection is done in two different methods:

Best selection: where the algorithm chooses the individual with the lowest fitness function and repopulates the whole population with that individual.

Random selection: where the individual is being chosen randomly depending on its fitness function, the fitness function of the worst individual and the sum of the differences between the worst fitness function and fitness function of every individual in the current population.

The program continuous to iterate until the fitness function of the best selection is not zero in which case that generation is chosen as the solution of the Sudoku problem.

IV. Experiments

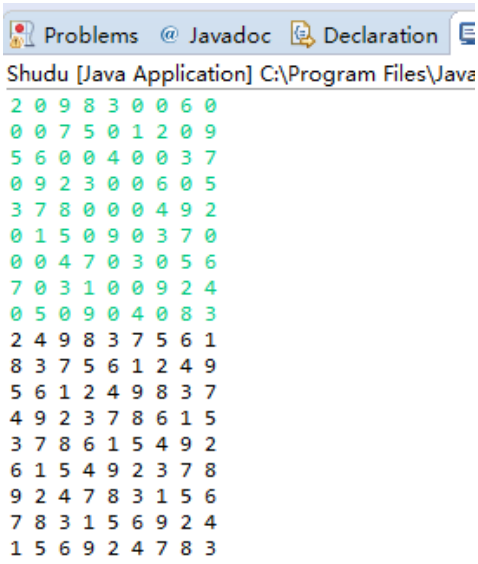
First, we run the program which can generate a random Sudoku puzzle to get the initial problem.

```
<terminated> Generate [Java Application] C:\Program Files\
Initial matrix:
 9 7 8  3 1 2  6 4 5
 3 1 2  6 4 5  9 7 8
 6 4 5  9 7 8  3 1 2

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

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

Array Generated:
 2 7 4 6 5 1 3 8 9
Randomized matrix:
 2 4 9  8 3 7  5 6 1
 8 3 7  5 6 1  2 4 9
 5 6 1  2 4 9  8 3 7

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

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

Sudoku Generated:
 2 0 9  8 3 0  0 6 0
 0 0 7  5 0 1  2 0 9
 5 6 0  0 4 0  0 3 7

 0 9 2  3 0 0  6 0 5
 3 7 8  0 0 0  4 9 2
 0 1 5  0 9 0  3 7 0

 0 0 4  7 0 3  0 5 6
 7 0 3  1 0 0  9 2 4
 0 5 0  9 0 4  0 8 3
```

As the figure shows, we use a matrix which satisfy constraints for Sudoku puzzle as a seed matrix. Through the seed matrix, we can generate a new randomized matrix which is the solution for Sudoku puzzle. Then we erase some value in matrix, use zero instead to get a Sudoku problem.

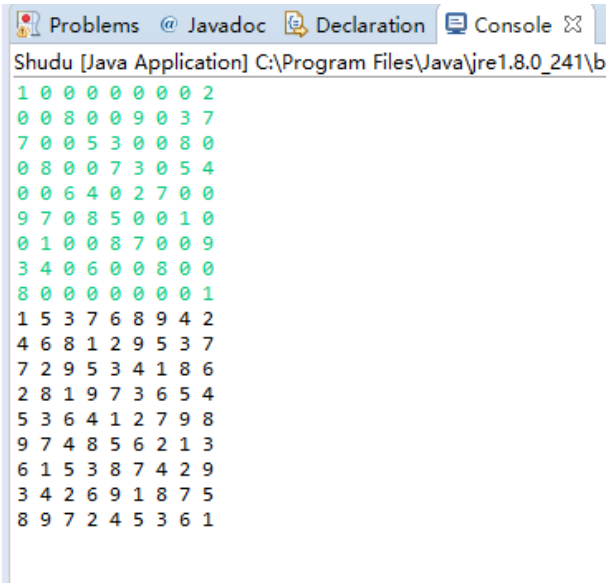
Then we will implement the first algorithm to solve this Sudoku puzzle.

This algorithm uses DFS and backtracing while combining with pruning to improve the efficiency.

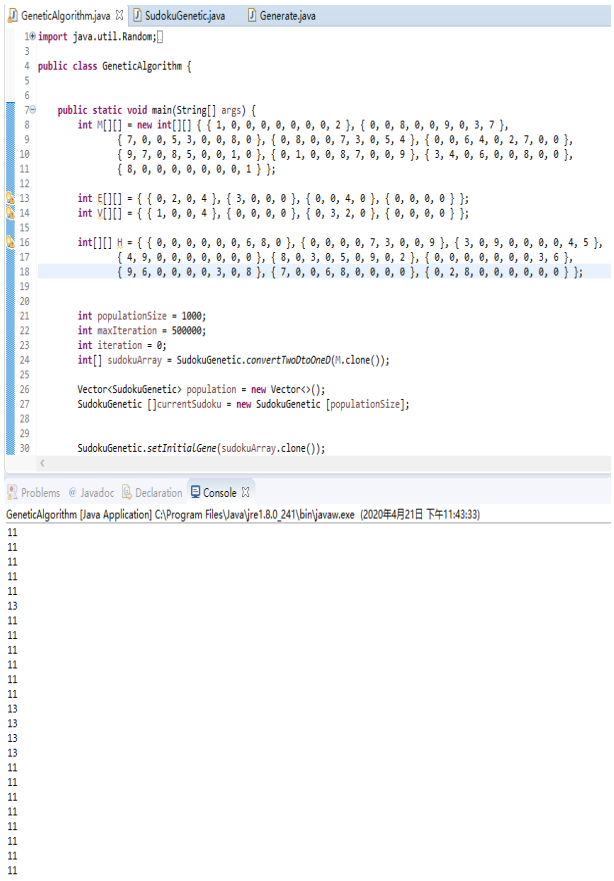


After entering the initial problem, solution was given out immediately.

Also, the program can solve more difficult Sudoku puzzle which generated by former codes at once.

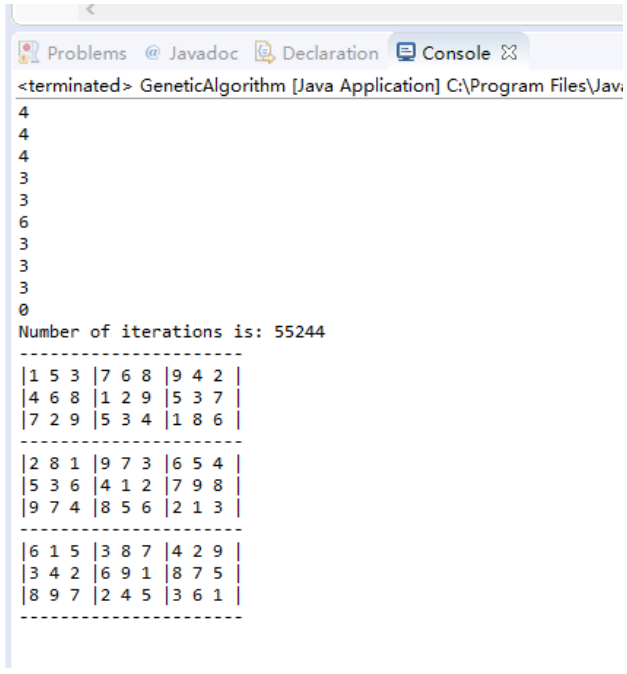


After testing the first algorithm, we continue to test genetic algorithm on solving Sudoku puzzles.

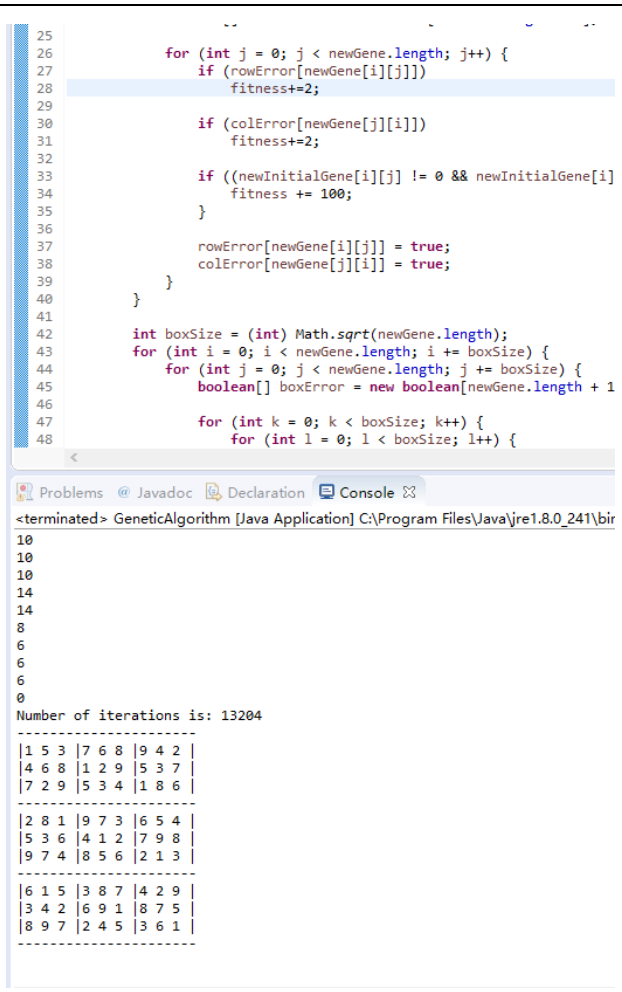


The figure shows initial problem we used as input M[][] and running iterations with printed fitness values.

Finally, we get the solution:



After changing fitness value added by 2 when error occurs:

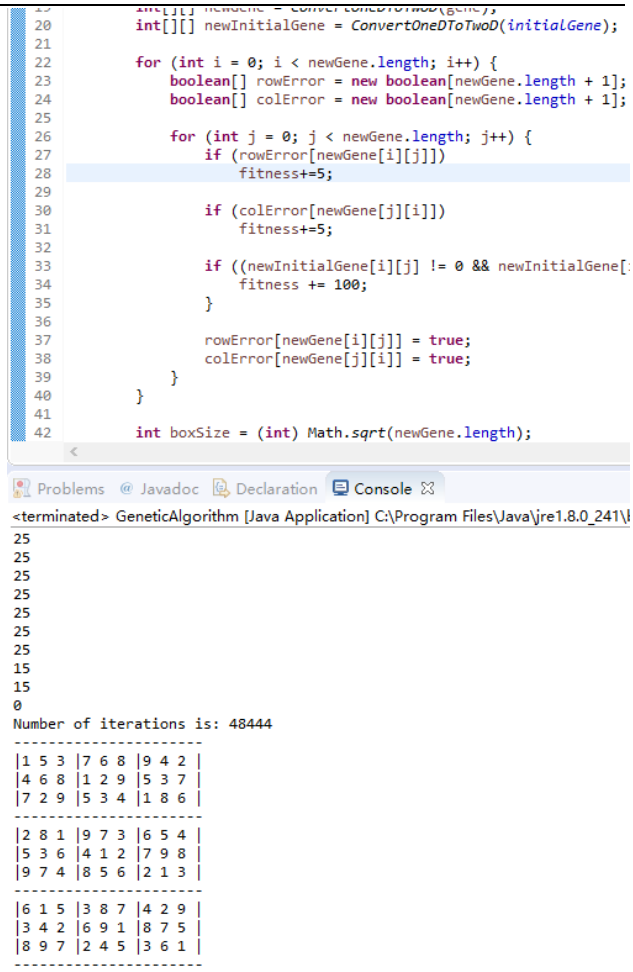


```
            for (int j = 0; j < newGene.length; j++) {
                if (rowError[newGene[i][j]])
                    fitness+=2;

                if (colError[newGene[j][i]])
                    fitness+=2;

                if ((newInitialGene[i][j] != 0 && newInitialGene[i]
                    fitness += 100;
                }

                rowError[newGene[i][j]] = true;
                colError[newGene[j][i]] = true;
            }
        }

        int boxSize = (int) Math.sqrt(newGene.length);
        for (int i = 0; i < newGene.length; i += boxSize) {
            for (int j = 0; j < newGene.length; j += boxSize) {
                boolean[] boxError = new boolean[newGene.length + 1

                for (int k = 0; k < boxSize; k++) {
                    for (int l = 0; l < boxSize; l++) {
```

Problems Javadoc Declaration Console

```
<terminated> GeneticAlgorithm [Java Application] C:\Program Files\Java\jre1.8.0_241\bir
10
10
10
14
14
8
6
6
6
0
Number of iterations is: 13204
---------------------
|1 5 3 |7 6 8 |9 4 2 |
|4 6 8 |1 2 9 |5 3 7 |
|7 2 9 |5 3 4 |1 8 6 |
---------------------
|2 8 1 |9 7 3 |6 5 4 |
|5 3 6 |4 1 2 |7 9 8 |
|9 7 4 |8 5 6 |2 1 3 |
---------------------
|6 1 5 |3 8 7 |4 2 9 |
|3 4 2 |6 9 1 |8 7 5 |
|8 9 7 |2 4 5 |3 6 1 |
---------------------
```

After changing fitness value added by 5 when error occurs:

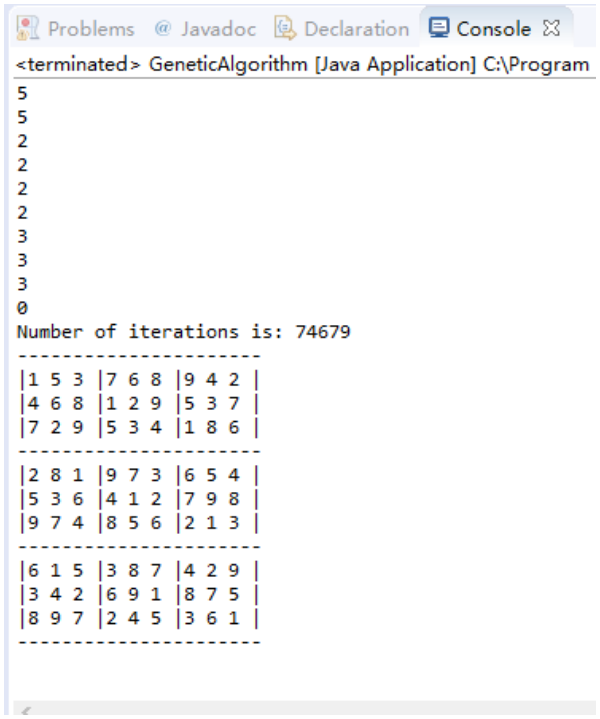


```
        int[][] newInitialGene = ConvertOneDToTwoD(initialGene);

        for (int i = 0; i < newGene.length; i++) {
            boolean[] rowError = new boolean[newGene.length + 1];
            boolean[] colError = new boolean[newGene.length + 1];

            for (int j = 0; j < newGene.length; j++) {
                if (rowError[newGene[i][j]])
                    fitness+=5;

                if (colError[newGene[j][i]])
                    fitness+=5;

                if ((newInitialGene[i][j] != 0 && newInitialGene[
                    fitness += 100;
                }

                rowError[newGene[i][j]] = true;
                colError[newGene[j][i]] = true;
            }
        }

        int boxSize = (int) Math.sqrt(newGene.length);
```

Problems Javadoc Declaration Console

```
<terminated> GeneticAlgorithm [Java Application] C:\Program Files\Java\jre1.8.0_241\
25
25
25
25
25
25
25
15
15
0
Number of iterations is: 48444
---------------------
|1 5 3 |7 6 8 |9 4 2 |
|4 6 8 |1 2 9 |5 3 7 |
|7 2 9 |5 3 4 |1 8 6 |
---------------------
|2 8 1 |9 7 3 |6 5 4 |
|5 3 6 |4 1 2 |7 9 8 |
|9 7 4 |8 5 6 |2 1 3 |
---------------------
|6 1 5 |3 8 7 |4 2 9 |
|3 4 2 |6 9 1 |8 7 5 |
|8 9 7 |2 4 5 |3 6 1 |
---------------------
```

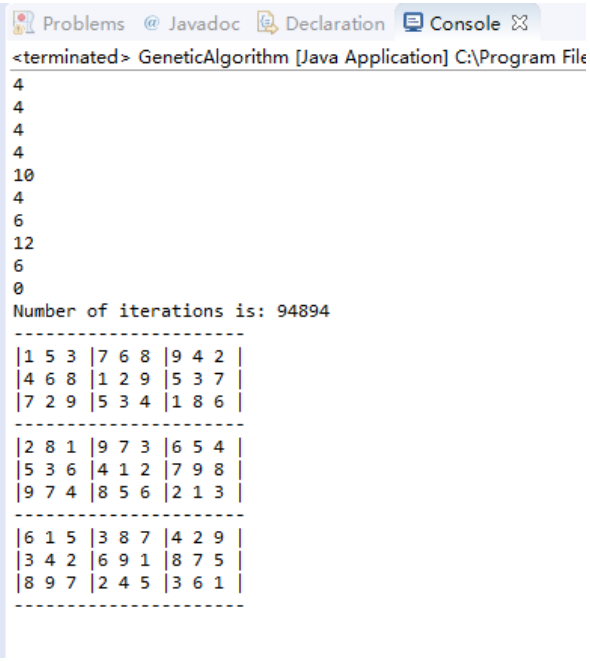
Change values to 500 when the prefilled cells of the initial problem do not match with the cell values of the current generation:



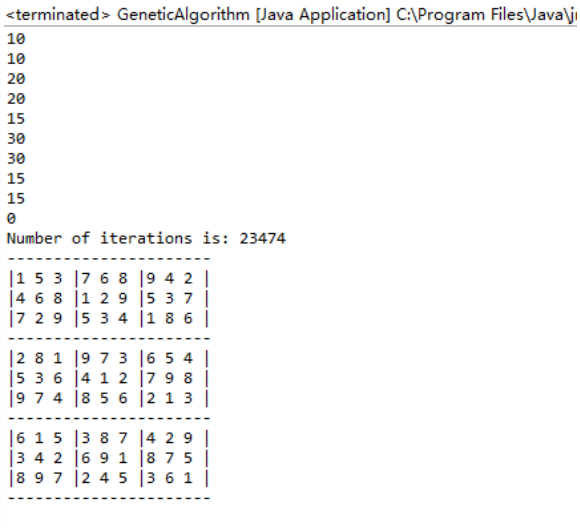
Problems Javadoc Declaration Console

```
<terminated> GeneticAlgorithm [Java Application] C:\Program
5
5
2
2
2
2
3
3
3
0
Number of iterations is: 74679
---------------------
|1 5 3 |7 6 8 |9 4 2 |
|4 6 8 |1 2 9 |5 3 7 |
|7 2 9 |5 3 4 |1 8 6 |
---------------------
|2 8 1 |9 7 3 |6 5 4 |
|5 3 6 |4 1 2 |7 9 8 |
|9 7 4 |8 5 6 |2 1 3 |
---------------------
|6 1 5 |3 8 7 |4 2 9 |
|3 4 2 |6 9 1 |8 7 5 |
|8 9 7 |2 4 5 |3 6 1 |
---------------------
```

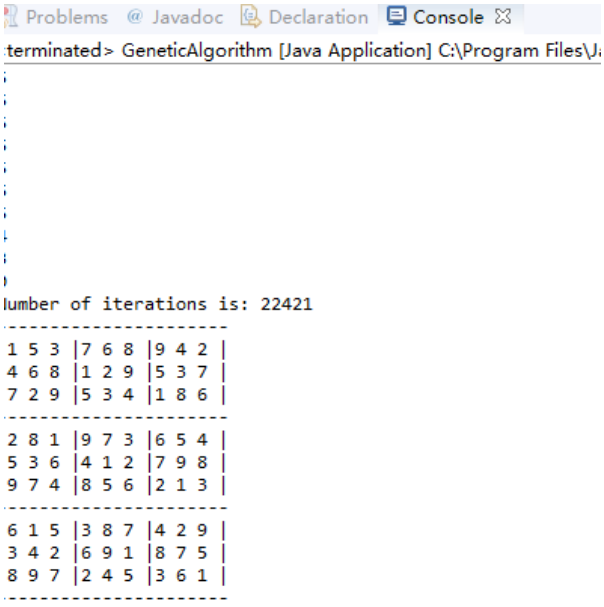
Then change fitness value added by 2 when error occurs:



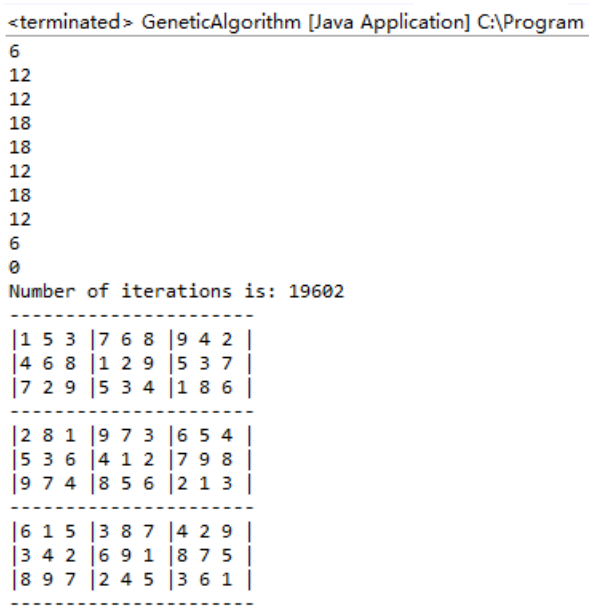
Then change fitness value added by 5 when error occurs:



Change values to 1000 when the prefilled cells of the initial problem do not match with the cell values of the current generation:



Then change fitness value added by 2 when error occurs:



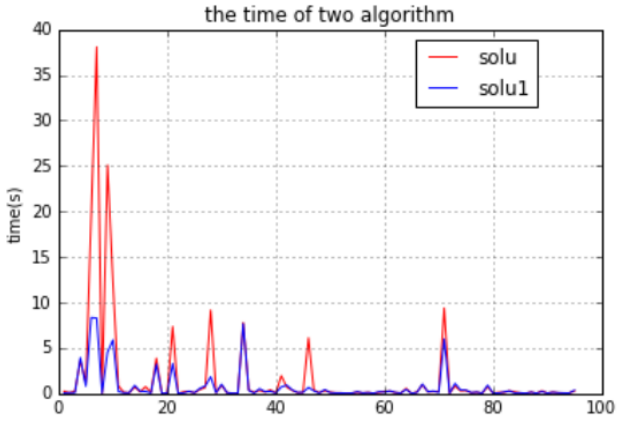
Then change fitness value added by 5 when error occurs:

```
Problems  Javadoc  Declaration  Console
<terminated> GeneticAlgorithm [Java Application] C:\Program
10
20
25
20
15
15
30
30
15
0
Number of iterations is: 35544
----------------------
|1 5 3 |7 6 8 |9 4 2 |
|4 6 8 |1 2 9 |5 3 7 |
|7 2 9 |5 3 4 |1 8 6 |
----------------------
|2 8 1 |9 7 3 |6 5 4 |
|5 3 6 |4 1 2 |7 9 8 |
|9 7 4 |8 5 6 |2 1 3 |
----------------------
|6 1 5 |3 8 7 |4 2 9 |
|3 4 2 |6 9 1 |8 7 5 |
|8 9 7 |2 4 5 |3 6 1 |
----------------------
```

V. Conclusion and Analysis

The experiments' results contain the Sudoku Problem automatically generated and the solutions got through different algorithms. We compared the algorithms under different Sudoku problems as well. (consider different N and different blanks to see whether these elements would affect the algorithms) We generated the time cost between our algorithms to see the actual difference/benefits after applying new elements into the algorithm. Considering the risk of not getting all the answers for the problem, we generate all the answers for the Sudoku Problem through lazy algorithm if it is necessary (there are not only one solution for the problem), but under the general condition, getting one real-actual solution is enough. According to our experiments, although the math logic solution work well in some specific scenes (the back tracing times are few), the algorithm designed based on the CSP problem works the best under all circumstances especially when the problem requires many back tracings (the solution are mostly number large), the genetic algorithm and several logical algorithms have their limitations on this problem. Besides, we found that the constraints in the Constraint satisfaction problem are quite significant, because in general, the pruning improvement is just changing the limitation considered order, the time cost has a great progress (compared to our origin algorithm pure DFS and back tracing), shown as the figure below.

The red line shows the original algorithm that only uses DFS and backtracing, the blue line shows improved algorithm which contains pruning. We have tested 100 different Sudoku puzzles generated and obviously in this figure, first 10 puzzles which has few back-tracing times show that pruning has huge advantages. With the increasement of back-tracing times, these two strategies seems have similar performance.



For the genetic algorithm, we are pretty sure that it can solve NP-complete problems effectively. However, when it comes to Sudoku, GA is ineffective in solving puzzles for there are thousands of iterations. So, we discuss why Sudoku seems easy by using backtracking algorithms but intractable for genetic algorithm. We tried the different fitness function with different parameters and found that they have few effects to control the iterations due to there are too many indeterministic variables during the genetic process. For example, mutations and crossover are totally randomized operations, so do random selections. Thus, these random processes do effect the iterations when we implement genetic algorithm.

Based on the algorithms we used and searched before, we can easily find that the algorithm we learned in related work such as Exclusion method can be seen using part of the hill climbing theory, with the solution processed the possible solution left are fewer, but when the solutions are open this kind of idea will not work as well.

In general, the backtracking method is a heuristic algorithm. From this point of view, it is very similar to the exhaustive method. The backtracking method could be thought as an organized exhaustive method. During the trial and error process, the search space is continuously reduced through the requirements of the title. This reduction is not a reduction of

one solution, but a large-scale search space Pruning makes the actual search space much smaller than the solution space of the problem, so the actual operating efficiency of the backtracking method is relatively high. That is why the back tracing and pruning has a better appearance in our project. However, we still need to see the shiny points of the genetic algorithm although it doesn't work well. The genetic algorithm simultaneously processes multiple individuals in the group, that is, simultaneously evaluates multiple solutions in the search space. Through the operation of the collection, the collection of rules and the knowledge base can be refined to achieve the purpose of advanced operation like machine learning. Through the operation of the tree structure, the best decision tree for classification can be obtained. It has better global search performance and is easy to parallelize. It does not require auxiliary information. Based on the experiments we can see only the fitness function is used to evaluate genetic individuals, and genetic operations are performed on this basis. It is not easy to fall into local optimum during the search process. Even if the defined fitness function is discontinuous, irregular, or noisy.

VI. Reference

1. *https://github.com/ctjacobs/sudoku-genetic-algorithm*
2. *Genetic Algorithms and Sudoku, Dr. John M. Weiss, Department of Mathematics and Computer Science, South Dakota School of Mines and Technology (SDSM&T),Rapid City, SD 57701-3995, MICS 2009*
3. *https://github.com/mahdavipanah/sudoku_genetic_python*
4. *https://www.geeksforgeeks.org/genetic-algorithms/*
5. *Sastry K., Goldberg D., Kendall G. (2005) Genetic Algorithms. In: Burke E.K., Kendall G. (eds) Search Methodologies. Springer, Boston, MA*
6. *https://blog.csdn.net/u010451580/article/details/51178225*
7. *https://zhuanlan.zhihu.com/p/33042667*
8. *https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6*