Math 110B Project 1 Report

Haodong Hu, Hao Zheng, Shanglin Li, Kaihao Zhang

August 30, 2021

1 Introduction

Our team firstly tried to use GA (Genetic Algorithm) to solve Sudoku since it is easy to understand and not such an abstract idea compared to the LP (Linear Programming). We did successfully finish the code of using GA to solve Sudoku, but it takes a too long time to solve 1000 quizzes. Later, we have found out the example on the Kaggle only uses 0.4 seconds to solve 16 quizzes with a high success rate, which is highly efficient. Therefore we decide to redo the code for using LIP (Linear Integer Programming) to solve Sudoku. Our team will focus on the LIP method to solve LIP with some of our achievements of the GA method.

2 GA Algorithm Report

2.1 Overview of Genetic Algorithm[7]

In this project, we have reached an agreement to include Genetic Algorithm (GA) as one of our computation and optimization algorithms. Therefore, before getting into the formal steps and codes of this method, it will be helpful to introduce the basic concepts of GA as well as the aspects which GA can help us with.

The algorithms of GA reflect the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. Such process can be visually represented by the image below[10].

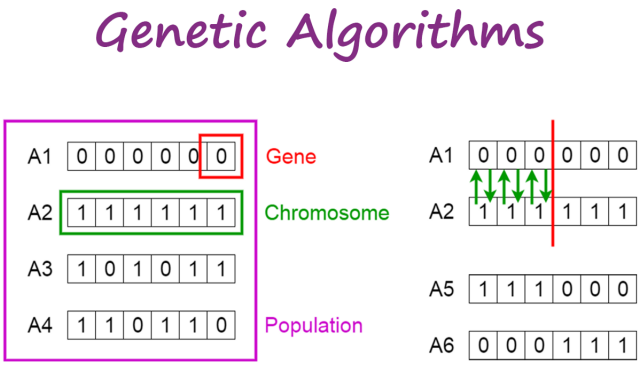

Figure 1: Image source: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

There are five phases considered in Genetic Algorithms. The first one is **initial population**, and the process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve. An individual is characterized by a set of parameters (variables) known as Genes.

Genes are joined into a string to form a Chromosome (solution). In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s), and in this case we say that we encode the genes in a chromosome. The second phase is **fitness function**, and the fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its **fitness score**. Next phase is **Selection**, and the idea of selection phase is to select the fittest individuals and let them pass their genes to the next generation. Two pairs of individuals (parents) are selected based on their fitness scores. Individuals with **high fitness** have more chance to be selected for reproduction. Moreover, we have a phase called **Crossover**, which 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. The following graphs will show how offspring are created and how new offspring are added to the selected population.

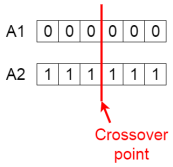

Figure 2: Crossover point is set to be 3 in this case

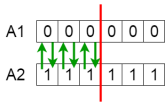

Figure 3: **Offspring** are created by exchanging the genes of parents among themselves until the crossover point is reached

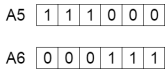

Figure 4: The new offspring are added to the population

The final phase of Genetic Algorithms is **Mutation**. In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped.

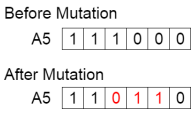

Figure 5: Mutation occurs to maintain diversity within the population and prevent premature convergence

In next section, we will go over the basic steps of Genetic Algorithms in detail and explain the results obtained based on the four given datasets.

2.2 Basic Steps

As we have mentioned in previous section that in GA, we have a pool or a population of possible solutions to the given problem. These solutions then undergo recombination and mutation (like in natural genetics),

producing new children, and the process is repeated over various generations. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield more "fitter" individuals.

Firstly, we created a helper matrix which is a list of lists containing all possible values that a Sudoku[2] problem can have. Then, we initialized a **population** (or the randomly generated possible number to each slot) of size 200 randomly using the helper and calculated the **fitness** of each candidate and sorted them. After the fitness scores are generated and recorded, we utilized the sorted list and took the first 120 (having greater fitness) as elites(More close to the correct answer). Elites would be directly used as the next generation since they have very close to the correct answer.

For the rest of the 80 possibilities, we have randomly selected two candidates from the population and performed crossover among them, and we call such step **mutation**. Then, we repeat all of the above steps (new generation → test fitness → divide the first 120 groups and the rest of 80 groups → Crossover and mutate) until we obtain a fitness value of 1, which indicates that we have successfully located a solution to the given Sudoku problem.

Now, for the given dataset "Small 1.csv", we set the numerical value of population to be 200 and generation to be 1500. Below is the result obtained from implementing GA.

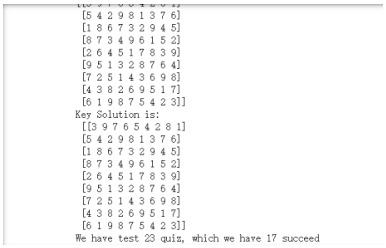

Figure 6: First try of the dataset "Small 1.csv"

2.3 Useful Python Commands for GA

- **isRow**: Checks whether there is a duplicate of a fixed/given value in a row.
- **isCol**: Checks whether there is a duplicate of a fixed/given value in a column.
- **isBlock**: Checks whether there is a duplicate of a fixed/given value in a 3×3 block.
- **Crossover**: Creates two new child candidates by crossing over parent genes.

2.4 Success Rate Report for GA

According to the GA algorithm, the success rate would be relatively low unless we enlarge the population and generation parameters as large as 200 and 1500, respectively. On the other hand, the GA algorithm becomes extremely time consuming as we enlarge these two parameters (data set part A and part B).

3 Linear Integer Programming Algorithms

3.1 Overview of Linear Integer Programming[3]

The method of Linear Integer Programming, also know as LIP, is largely utilized in terms of solving linear equations and inequalities. These systems often have many possible solutions and Linear programming is a set of mathematical and computational tools that allows users to locate a particular solution to this system that corresponds to the maximum or minimum of some other linear functions. Linear programming is a fundamental optimization technique that's been used for decades in science and math intensive fields. It's precise, relatively fast, and suitable for a range of practical applications.

The basic method for solving linear programming problems is called the **simplex method**, which has several variants. Another popular approach is the interior-point method. On the other hand, **Mixed-integer linear programming** problems are solved with more complex and computationally intensive methods like the **branch-and-bound** method[9], which uses linear programming under the hood. Some variants of this method are the **branch-and-cut** method[9], which involves the use of cutting planes, and the branch-and-price method[6].

For example, if we are given the following linear programming problem:

$$
\begin{aligned}
\text{maximize} \quad & z = x + 2y \\
\text{subject to:} \quad & 2x + y \leq 20 \\
& -4x + 5y \leq 10 \\
& -x + 2y \geq -2 \\
& x \geq 0 \\
& y \geq 0
\end{aligned}
$$

Figure 7: Find x and y such that the red, blue, and yellow inequalities, as well as the inequalities $x \geq 0$ and $y \geq 0$, are satisfied. The value of z should also be maximized

The independent variables we need to find, in this case x and y, are called the **decision variables**. The function of the decision variables to be maximized or minimized, in this case z, is called the **objective function** or the **cost function**. The inequalities we need to satisfy are called the **inequality constraints**. We can also have equations among the constraints called **equality constraints**. Moreover, we can visualize the problem in the following way[4].

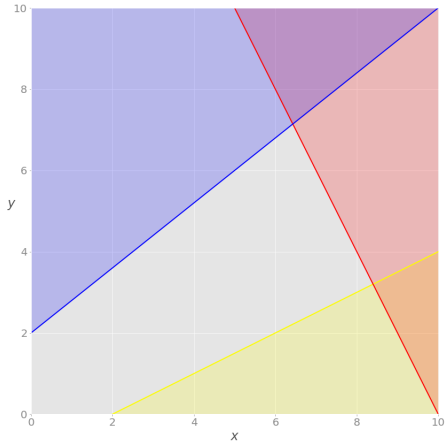

Figure 8: The red line represents the function $2x + y = 20$, and the red area above it shows where the red inequality is not satisfied. Similarly, the blue line is the function $-4x + 5y = 10$, and the blue area is forbidden because it violates the blue inequality. The yellow line is $-x + 2y = 2$, and the yellow area below it is where the yellow inequality is not valid.

If we disregard the red, blue, and yellow areas, only the gray area remains. Each point of the gray area satisfies all constraints and is a potential solution to the problem. This area is called the **feasible region**, and its points are **feasible solutions**. In this case, there's an infinite number of feasible solutions. We want to maximize z. The feasible solution that corresponds to maximal z is the optimal solution. If you were trying to minimize the objective function instead, then the optimal solution would correspond to its feasible minimum. Now, we expand the problem by adding an equality constraint in green.

$$
\begin{aligned}
\text{maximize} \quad & z = x + 2y \\
\text{subject to:} \quad & 2x + y \leq 20 \\
& -4x + 5y \leq 10 \\
& -x + 2y \geq -2 \\
& -x + 5y = 15 \\
& x \geq 0 \\
& y \geq 0
\end{aligned}
$$

Figure 9: New equation is labeled as green in the graph

Then, we are able to visualize the quation by adding it to previously generated graph.

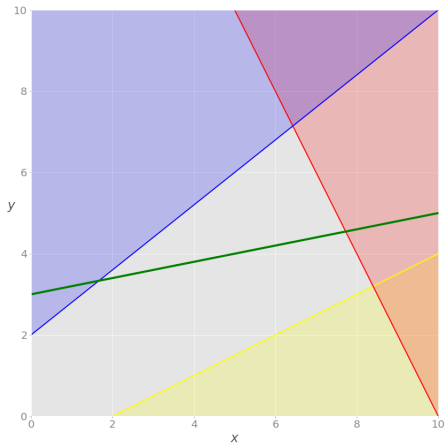

Figure 10: New line of equation is added to the original graph

The solution now must satisfy the green equality, so the feasible region is not the entire gray area anymore. It's the part of the green line passing through the gray area from the intersection point with the blue line to the intersection point with the red line. The latter point is the solution.

3.2 Implementation of LIP with the package PuLP

First of all, we need to install install the package **PuLP**, and to do that, we use the following code in Python:

```
 1 !pip install pulp
```

```
Collecting pulp
  Downloading PuLP-2.5.0-py3-none-any.whl (41.2 MB)
     |████████████████████████████████| 41.2 MB 65 kB/s
Installing collected packages: pulp
Successfully installed pulp-2.5.0
```

Figure 11: This code will download the latest version of PuLP and automatically check if the local computer has such package

Moreover, we might need to run pulptest or sudo pulptest to enable the default solvers for PuLP when the local computer is Mac, and the code is:

```
$ pulptest
```

Figure 12: This code will run pulptest or sudo pulptest to enable the default solvers for PuLP

Then, we defined a function called varname(i, j, k), which will return a string in the format $x_{i,j,k}$. The crange(a, b) function will return a range object for the closed interval $[a, b]$, and we set up a Sudoku puzzle solver class **Sudoku**. The dimensions of a Sudoku puzzle are defined by the size of a puzzle block. Each block has m rows and n columns. Since each block has $m \times n$ cells, the total possible number of values a cell can have is $N = m \times n$, and since each row or column of the puzzle must have exactly one occurrence of each value in $1 : N$, both the height and width of the puzzle are equal to N. This also implies that the puzzle has n blocks on the vertical direction and m blocks on the horizontal direction. This solver indexes the puzzle cells with pairs of indices (i, j) which are assigned as done for matrix entries, i.e., $(i, j) = (1, 1)$ represents the top-left corner cell of the puzzle. The index i represents the cell row and hence increases downwards, while j represents the cell column and hence increases to the right.

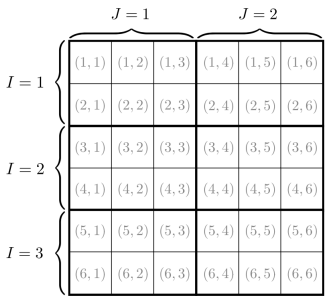

Figure 13: A Sudoku puzzle with blocks of size $m \times n = 2 \times 3$. The cell indices (i, j) are shown inside every cell, and the block indices (I, J) are shown on the left and top sides of the grid respectively. Both the height (number of rows) and width (number of columns) of the puzzle are equal to $N = m \times n = 6$. The puzzle has $n = 3$ blocks along the vertical direction and $m = 2$ blocks along the horizontal direction

Inside of the **Sudoku** class, we defined a function –init–$(self, m, n)$. This function initializes a solver for a Sudoku puzzle with block size $m \times n$. In general, solving a Sudoku puzzle is equivalent to solving an linear integer programming (LIP) problem, and this equivalence allows us to solve a Sudoku puzzle using any of the many freely available LIP solvers. A Sudoku puzzle is an $N \times N$ grid divided in blocks of size $m \times n$, i.e., each block contains m rows and n columns, which $N = m \times n$ (as mentioned above) since the number of cells in a block is the same as the number of rows/columns on the puzzles. The commonly known version of Sudoku is a 9×9 grid ($N = 9$) with 3×3 blocks ($m = n = 3$). At first, a cell can be either empty or contain an integer value in the interval $[1, N]$ and non-empty cells are fixed and can not be modified as the puzzle is solved. Here are the rules for solving the puzzle[1]:

- Each integer value $k \in [1, N]$ must appear exactly once in each row.
- Each integer value $k \in [1, N]$ must appear exactly once in each column.
- Each integer value $k \in [1, N]$ must appear exactly once in each block.

Each rule above individually implies that every cell of the puzzle will have a number assigned to it when the puzzle is solved, and that each row/column/block of a solved the puzzle will represent some permutation of the sequence $\{1, 2, \cdots, N\}$. Now, we define N^3 variables as follows: x_{ijk} is an integer variable which is restricted to be either 0 or 1, where 1 means that the value at cell (i, j) is equal to k and 0 means that the value at cell (i, j) is not k. Since each $k \in [1, N]$ must appear exactly once per row, we can express it as

$$\sum_{j=1}^{N} x_{ijk} = 1 \quad \text{for} \quad i, k = 1, 2, \cdots, N$$

In other words, for a fixed row i and a fixed $k \in [1, N]$, only a single x_{ijk} will be 1 on that row for $j = 1, 2, \cdots, N$. If the constraints above do not have any "$\leq$", we should always know that $x = a$ can be expressed as $a \leq x \leq a$, which is equivalent to the combination of $-x \leq -a$ and $x \leq a$ (any constraint can

be expressed as a pair of "less than or equal to" constraints like the ones we have in linear programming problems. In the second rule, it states that each $k \in [1, N]$ must appear exactly once per column, which can be expressed as

$$\sum_{i=1}^{N} x_{ijk} = 1 \quad \text{for} \quad j, k = 1, 2, \cdots, N$$

In order to simplify the third rule, which states that each $k \in [1, N]$ must appear exactly once per block, we begin by assigning pairs of indices (I, J) to each block in a similar way as we did in Figure 13: block (I, J) will contain cells with row indices $i = (I-1)m+1, \cdots, Im[5]$ and column indices $j = (J-1)n+1, \cdots, Jn$. Therefore, the third rule can be represented as:

$$\sum_{i=(I-1)m+1}^{Im} \sum_{j=(J-1)n+1}^{Jn} x_{ijk} = 1 \quad \text{for} \quad I = 1, 2, \cdots, n,\ J = 1, 2, \cdots, m,\ k = 1, 2, \cdots, N$$

Since most LIP solvers allow bounds on the values which each x_{ijk} can take to be set directly, this last set of constraints often does not need to be specified in the same manner as the previous ones. We now have a complete LIP formulation of a Sudoku puzzle: our goal is to minimize the objective function $f(x_{111}, \cdots, x_{ijk}, \cdots, x_{NNN}) = 0$ subject to all constraints specified on above equations. After solving the LIP problem outlined above, the solution to the Sudoku puzzle can be constructed directly by placing, at each cell (i, j), the value k such that $x_{ijk} = 1$[8].

3.3 Useful commands for LIP

- m: The number of rows per puzzle block.
- n: The number of columns per puzzle block.
- set-cell-value($self, i, j, k$): Sets the value of cell (i, j) to k. This function will throw a **RuntimeError** exception if called after the puzzle has already been solved.
- get-cell-value($self, i, j$): Returns the value of cell (i, j) or **None** if the puzzle has not yet been solved.
- solve(self): Solves the puzzle and returns True if the puzzle is solvable, False otherwise.
- size(self): Returns the number of rows and columns in the puzzle.

3.4 LIP idea in Python

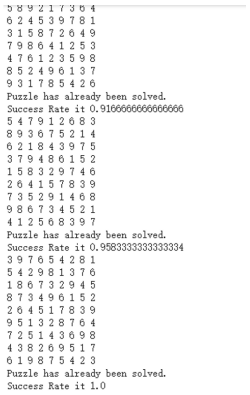

Figure 14: Result in Command window for LIP

In the solution, each 9×9 grid is the solution return from LIP and will be compared to the solution give in the "solution" column. If they are the same, it will increase the Success rate and store the result with

"Valid" into the new "CSV." file. Following are some explanations of the codes that we have implemented in Python.

```
# setup dictionary with all variables x_{i,j,k}
self.x = pulp.LpVariable.dict("%s",
                                                x_names,
                                                lowBound=0,
                                                upBound=1,
                                                cat=pulp.LpInteger)
self.sudoku_model += 0
```

Figure 15: Before set up the rule of the LIP. We need to setup the dictionary for i, j, k variables.

```
# constraint: k appears only once in each i
for i in crange(1, N):
     for k in crange(1, N):
          self.sudoku_model += sum([self.x[variable_name(i, j, k)]
                                                for j in crange(1, N)]) == 1

# constraint: k appears only once in each j
for j in crange(1, N):
     for k in crange(1, N):
          self.sudoku_model += sum([self.x[variable_name(i, j, k)]
                                                for i in crange(1, N)]) == 1

# contraint: k appears only once in each block (I,J)
for I in crange(1, n):
     for J in crange(1, m):

          i_low = (I - 1) * m + 1
          j_low = (J - 1) * n + 1
          block_i_values = range(i_low, i_low + m)
          block_j_values = range(j_low, j_low + n)

          for k in crange(1, N):
               self.sudoku_model += sum([self.x[variable_name(i, j, k)]
                                                for i in block_i_values
                                                for j in block_j_values]) == 1

# contraint: each cell (i,j) must have some value k
for i in crange(1, N):
     for j in crange(1, N):
          self.sudoku_model += sum([self.x[variable_name(i, j, k)]
                                                for k in crange(1, N)]) == 1
```

Figure 16: Here, we follow the rule of the Sudoku. Each row/column for number $1 : N$, each number only appear once. It is also same to each block in the $N \times N$ grid

```
def set_cell_value(self, i, j, k):
     """
     Input cell(i,j) to get the value of k
     """

     if self.sudoku_model.status != pulp.LpStatusNotSolved:
          self.flag = True
     self.sudoku_model += self.x[variable_name(i, j, k)] == 1

def get_cell_value(self, i, j):
     """
     Returns the value of cell (i,j) or failed
     """

     N = self.size()

     for k in crange(1, N):
          if self.x[variable_name(i, j, k)].value() == 1:
               return k
     return None
```

Figure 17: Define two functions in the Class Sudoku to figure at (i, j) what value of K will be suited. We will repeat to do for all the null slots in the $N \times N$ Sudoku to get the solution.

3.5 Success Rate Report for LIP

According to the LIP algorithm, We run all data sets from A and B smoothly which return a **100** percent success rate for all the data. In particular for data in part B with a significant amount of data, we test our Success rate by random selection of 1000 data and also return a **100** percent success rate.

	quizzes	solutions	Result
0	8E+79	4.9E+80	Valid
1	9.01E+75	3.52E+80	Valid
2	9E+80	9.28E+80	Valid
3	1E+80	1.65E+80	Valid
4	6E+80	6.73E+80	Valid
5	2.14E+79	3.21E+80	Valid
6	2E+80	2.59E+80	Valid
7	4.71E+80	4.72E+80	Valid
8	2.11E+80	2.15E+80	Valid
9	4.03E+78	7.35E+80	Valid
10	5.03E+76	8.8E+80	Valid
11	4E+76	1.95E+80	Valid
12	3E+77	6.57E+80	Valid
13	3.1E+77	2.17E+80	Valid
14	6.09E+77	3.82E+80	Valid
15	8.04E+79	7.89E+80	Valid
16	4.38E+75	9.71E+80	Valid
17	6E+79	4.7E+80	Valid
18	5.04E+79	2.53E+80	Valid
19	9.06E+79	5.94E+80	Valid
20	9E+80	9.44E+80	Valid
21	8.02E+75	2.47E+80	Valid
22	9.1E+77	5.48E+80	Valid
23	3.97E+80	3.98E+80	Valid
Count of "valid" in column D			24
Count of quizzes			24
Success Rate			1

Figure 18: Results of “samll 1.csv” dataset

	quizzes	solutions	Result
0	9.4E+79	7.95E+80	Valid
1	9.42E+68	2.49E+80	Valid
2	7E+75	6.53E+80	Valid
3	5.25E+79	1.52E+80	Valid
4	9E+79	8.94E+80	Valid
5	5E+75	1.68E+80	Valid
6	5E+80	5.97E+80	Valid
7	7E+79	3.8E+80	Valid
1001	5E+74	1.39E+80	Valid
1002	7.6E+71	8.42E+80	Valid
1003	8.9E+75	4.26E+80	Valid
1004	3.08E+80	3.48E+80	Valid
1005	5E+75	3.3E+80	Valid
1006	1E+79	3.18E+80	Valid
1007	5.4E+71	1.63E+80	Valid
1008	7.6E+74	8.32E+80	Valid
1009	4E+72	5.28E+80	Valid
1010	2.5E+80	2.58E+80	Valid
Count of "valid" in column D			1011
Count of quizzes			1011
Success rate			1

Figure 19: Result of “small 2.csv” dataset

	quizzes	solutions	Result
0	5.08E+80	5.99E+80	Valid
1	7.08E+79	9.72E+80	Valid
2	1.09E+80	1.5E+80	Valid
3	4.03E+79	7.49E+80	Valid
4	6.9E+80	6.92E+80	Valid
5	3.81E+80	3.82E+80	Valid
6	5E+80	5.82E+80	Valid
7	9E+80	9.22E+80	Valid
8	4.7E+76	8.17E+80	Valid
992	2.06E+77	3.57E+80	Valid
993	1.4E+78	6.31E+80	Valid
994	3E+80	3.18E+80	Valid
995	2.01E+80	2.72E+80	Valid
996	5E+77	9.19E+80	Valid
997	3.85E+78	1.74E+80	Valid
998	7E+76	4.96E+80	Valid
999	9E+80	9.35E+80	Valid
Count of "valid" in column D			1000
Count of quizzes			1000
Success Rate			1

Figure 20: Result of "large 1.csv" dataset

1		quizzes	solutions	Result
2	0	4.8E+77	2.97E+80	Valid
3	1	7E+79	4.79E+80	Valid
4	2	7.39E+79	8.74E+80	Valid
5	3	7E+79	3.76E+80	Valid
6	4	1.7E+80	1.77E+80	Valid
7	5	8.31E+78	5.68E+80	Valid
8	6	7E+79	3.76E+80	Valid
993	991	8.2E+79	7.82E+80	Valid
994	992	8E+79	5.84E+80	Valid
995	993	9.04E+76	6.48E+80	Valid
996	994	9.46E+80	9.46E+80	Valid
997	995	6.05E+80	6.45E+80	Valid
998	996	6E+80	6.58E+80	Valid
999	997	3.52E+78	7.44E+80	Valid
1000	998	5.09E+79	6.54E+80	Valid
1001	999	8.69E+77	2.54E+80	Valid
1002	Count of "valid" in column D			1000
1003	Count of quizzes			1000
1004	Success rate			1

Figure 21: Results of "large 2.csv" dataset

4 Comparison of GA and LIP with Summary

GA and LIP are based on totally different algorithms. GA is created on randomly generated numbers. But LIP is based on the LP-minimize. It decided the GA must spend more time than LIP. Also, the Success rate of GA will depend on the parameters such as the population and generation. Increase these parameters will increase the success rate, but also will increase the time spend on each quiz. Since its property of random, the success rate will be different for different times when you run it. It depends on "how lucky you are" when you run GA. But I believe, if we set up the parameters at a large number, it should be worked to solve all the Sudoku quiz. For the LIP, since it basically solving many LP to get the solution. And the module pulp gives the way to solve them. It is perfectly suited to the Sudoku with a very high success rate and solved in a very short time. Therefore LIP should be the best method in the optimization area to solve Sudoku.

References

[1] Prabhu Babu et al. "Linear systems, sparse solutions, and Sudoku". In: *IEEE Signal Processing Letters* 17.1 (2009), pp. 40–42.

[2] Xiu Qin Deng and Yong Da Li. "A novel hybrid genetic algorithm for solving Sudoku puzzles". In: *Optimization Letters* 7.2 (2013), pp. 241–257.

[3] Fred Glover. "Improved linear integer programming formulations of nonlinear integer problems". In: *Management Science* 22.4 (1975), pp. 455–460.

[4] Ellis L Johnson, George L Nemhauser, and Martin WP Savelsbergh. "Progress in linear programming-based algorithms for integer programming: An exposition". In: *Informs journal on computing* 12.1 (2000), pp. 2–23.

[5] Ellis L Johnson, George L Nemhauser, and Martin WP Savelsbergh. "Progress in linear programming-based algorithms for integer programming: An exposition". In: *Informs journal on computing* 12.1 (2000), pp. 2–23.

[6] Elias Khalil et al. "Learning to branch in mixed integer programming". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 1. 2016.

[7] Seyedali Mirjalili. "Genetic algorithm". In: *Evolutionary algorithms and neural networks*. Springer, 2019, pp. 43–55.

[8] Sherif Sherif, Bahaa Saleh, and Renato De Leone. "Binary image synthesis using mixed linear integer programming". In: *IEEE Transactions on Image Processing* 4.9 (1995), pp. 1252–1257.

[9] Jadranka Skorin-Kapov and Frieda Granot. "Non-linear integer programming: sensitivity analysis for branch and bound". In: *Operations Research Letters* 6.6 (1987), pp. 269–274.

[10] Ma Yong-Jie and Yun Wen-Xia. "Research progress of genetic algorithm [J]". In: *Application Research of Computers* 29.4 (2012), pp. 1201–1206.