SUDOKU Solver with Linear Programming

Introduction

Our project, “sudoku solver”, is intended to implement a solver that can solve a given 9 x 9 sudoku problem using optimization method. For the simplicity of our implementation, each sudoku problem is represented by a string of length 81, in which each element, in order, represents a cell of 9 x 9 sudoku problem. In this string, a positive integer represents a cell that is given a number as a clue, and 0 represents an empty cell that need to be filled in using our solver.

There are plenty of methods that are invented and posted online to solve sudoku, and the simplest method is using Backtracking algorithm. Others including Genetic Algorithm and Linear Programming. Our solver uses Linear Programming, a constrained optimization method. The reason we choose Linear Programming is that we can perfectly represent a given sudoku problem as linear constraints of a linear problem.

Given a string representing a sudoku problem, our solver encode each element in a 1 x 9 binary array, for example, 1 -> [1,0,0,0,0,0,0,0,0]. Therefore, get a 1 x 729 binary array representing the whole problem. Then we generate the constraints of sudoku from the following requirements:

(1) each column has no repeated number;
(2) each row has no repeated number;
(3) each 3 x 3 box has no repeated number;
(4) each cell should be filled in with an integer between 1 and 9 inclusively;
(5) each cell that is given as a clue should not be changed;

These requirements need to be implemented in the form of an equation of linear combination, then we can solve it with a Weighted LP1 described in the paper. The following is the mathematical expressions of the above constraints.

No repeated numbers: $\underbrace{(I_{9*9}0_{9*92}I_{9*9}0_{9*92}\ldots\ldots I_{9*9}0_{9*92})}_{9}x = 1_{9*1}$

1st Roll $\underbrace{(I_{9*9}I_{9*9}\ldots\ldots I_{9*9}}_{9}\,0_{9*648})x = 1_{9*1}$

1st Box $(I_{9*9}I_{9*9}I_{9*9}0_{9*54}I_{9*9}I_{9*9}I_{9*9}0_{9*54}I_{9*9}I_{9*9}I_{9*9}0_{9*540})x = 1_{9*1}$

1st Cell $\underbrace{(1\ldots\ldots1}_{9}\,\underbrace{0\ldots\ldots0)}_{720}x = 1$

Clue $\underbrace{(0\ldots\ldots0}_{144}\ 010000000\ \underbrace{0\ldots\ldots0)}_{576}x = 1$

We found that Linear Programming is a better algorithm to solve this problem becau se Linear Programming helps in simplicity and productive management of an organiza tion which yields better outcomes. Also, it improves the quality of decision so th at we can theoretically get higher accurate recovery rate.

Basic algorithm: We decide to choose Weighted LP1 as our main algorithm to solve the linear programming problem. We set $x_{ori} = x_1 - x_2$, and construct a new variable by combining x_1 and x_2 (which will be the solution for WP1) to take charge of all entr ies in x that is greater than 0. Meanwhile, we add weight to each element by multi plying a matrix W=diag(w_1, w_2, …, w_n) with

$$w_k = \frac{1}{|x_k|^{i-1} + \epsilon}, \; 0 < \epsilon < 1$$

since our variable has 729 * 2 = 1458 elements, we reconstruct W = [W,W.T]. Also, we decide to set epsilon equal to 0.5 after several times of attempts because it g ives the highest success rate. Below is the pseudocode for the LP1 algorithm.[1]

```
(Weighted LP1) Solving the Weighted l1-Norm minimization problem (WP_1) by (LP1)
Input: $L = 10, \hat{x} = 0, \check{x} = 0, x_{ori} = \hat{x} - \check{x}, tol = 1*10^{-10}$
 For i = 1: L
$W = diag\left(\frac{1}{|x_{ori}| + \epsilon}\right);$
Based on the method (LP1), $x_{new} = \hat{x} - \check{x}$ is obtained by solving the following linear programming problem:
        $min\ W\begin{bmatrix}\hat{x}\\ \check{x}\end{bmatrix}, s.t.\ A\begin{bmatrix}\hat{x}\\ \check{x}\end{bmatrix} = b, \begin{bmatrix}\hat{x}\\ \check{x}\end{bmatrix} \in C;$
 $if\ \|x_{ori} - x_{new}\| < tol:$
      break;
   else
      $x_{ori} = x_{new}$
   END
   END
   Output: $x_{new}$
```

Citations

Sinkhorn is an entropic regularization of optimal transport algorithm. It was sugg ested that can be applied into sudoku problem. We were able to find a recent notes written by Gabriel Peyré and Marco Cuturi[2]. The notes describes Sinkhorn on a theoretical level. While Nat Timmis from Google AI research team[4] provides an elegant TensorFlow implementation of skinhorn for 1D optimal transport. This implementation is originally proposed by Marco Cuturi, Olivier Teboul and Jean-Philippe Vert in a recent research paper[3].

While the main inspiration and the resource of our project is the paper "A Warm Restart Strategy for Solving Sudoku by Sparse Optimization Methods" [1] written by Yuchao Tang, Zhenggang Wu and Chuanxi Zhu. In the paper, they propose a way to transforming the sudoku board into binary variables that can be applied in math equations. Also a linear programming method is applied in solving the sudoku problem. Since we failed to implement sinkhorn, this paper greatly helps us implement a working algorithm which also has a quite good result.

The most popular randomized algorithm for Sudoku game is the GA (genetic algorithm). This algorithm is studied in the paper[7]. The genetic algorithm requires the fitness function instead of the objective function, and there are some ideas to solve the sudoku by genetic algorithm[8][9], However, from the general idea of genetic algorithm, this method for solving sudoku problem is super inefficient because it takes a long time to solve one problem (one generation of genetic algorithm is long enough, and there are usually 5 generations to solve a sudoku). Therefore, it will be impossible for us to test this algorithm against our dataset (more than 1000 problems).

We mainly referred to the pseudo code in the paper. For the main algorithm, we used one of the functions written in SciPy library[5] to solve for the weighted linear minimization problem, we also referred other official documents about Weighted Linear Programming to help us implement our own weighted lp1 solver. During the process of reading and writing files part, we referred to some previous codes that use pandas[6] to help us get this job done easily. The main algorithm that we use is discussed above (Problem Statement::basic_algorithm)

Reference and previous work search:

The first step is of course to find the best algorithm for our Sudoku problem. We split and head for different kinds of solutions. Chenhao did some research on geometric algorithm. Yingyi and Jason went for sinkhorn, and find some useful notes about optimal transport algorithm[2][3]. Yingyi also found a Tensorflow implementation of sinkhorn in 1D as a reference code[4]. While Luobin found a paper solving sudoku with linear programming is the one that we used as our final algorithm[1]. Since we failed on sinkhorn, Luobin also explained the linear programming methods to the rest of us and split the job in coding part.

Sudoku basic problem setting:

Sudoku itself is quite easy. We represent it as a 2-dimension list. The real probl em here is the constraints. After discussion, we set the constraints into followin g parts: column(aka, must have 1-9 in one column), row(same as column), box(that i s the 3*3 square), clue(those numbers that are already given at the beginning of t he game) and cell(we need to make sure that there is no 0s in the board). Luobin s et up the Sudoku class and finished two most confusing constraints: cell and clue. Chenhao finished row constraints and column constraints. All these constraints are referred to the section 2-1 in the paper[1]. While Jason and Yingyi helped finishi ng rest constraints and some helper functions after they stop working on sinkhorn.

<u>Algorithm first try with sinkhorn:</u>
Yingyi and Jason did the first try with sinkhorn. The reference is from the codes written by Google research team implemented with tensorflow. Originally, Yingyi an d Jason intended to transfer the code using numpy since Yingyi and Jason are both not familiar with tensorflow. While some of the definitions in tensorflow are not easy to rewrite in numpy, the progress of implementation was quite slow and had lo ts of errors when debugging. While Luobin found a relatively easier approach and a lready had some accomplishments, this method is dropped and both Yingyi and Jason started to help with the current one.

<u>Main algorithm Linear Programming coding:</u>
Basic algorithm: Luobin implemented the basic weighted linear programming algorit hm. This is our key algorithm. This implementation exactly follows the first weigh ted linear programming algorithm in the paper (We also mentioned it in problem sta tement). The key function in this algorithm is to use SciPy library[5] since it’ s a little bit hard to start it from scratch. For all the hyper parameters, we use t he numbers that are provided in the paper because it gives the best results.

Detective function: This is the function that helps checking if the sudoku game i s already solved or not. It is used the main solver every time we use a solver in the loop. It returns a new board status and a status telling if there are any unso lved row/column/box constraint in the game board. Yingyi finished the detective fu nction.

Main solver and helper functions: This is the main implementation for solving the sudoku game. Chenhao set up the main frame of the implementation. Luobin and Jason finished all of the helper functions that is needed in this section. As stated in the paper, we will need new solver after three tries with the original solver. Ja son finished the fourth solver. While he also tried to implement a fifth solver, b ut didn’ t work out.

Plots and clean up: Jason and Yingyi helped with the plots and all the cleanup of the codes. Some of the file reading codes are referred to other people’ s work[6].

Experience in coding

Problem setting:

Initially, we set up the Sudoku problem by just using several functions that were written separately. The whole file looks like a mess and we faced several errors when we tried to compile the codes. Then we decided to create a class for the Sudo ku problem and define all the helper functions within the class, all the parameter s are stored in a class variable which makes more sense. This also helps reduce th e number of variables when dealing with solver functions in later codes.

While the first challenge is transforming the equation in the paper into codes. Th e representations in the paper is not easy to read and understand. It is not writt en in a very common way. We did lots of scratches trying to make sure that we unde rstand each and every equation correctly. Especially Yingyi, it took him quite a long time figuring out what the matrix looks like.

Second thing is, setting the constraints. There are lots of constraints in this pr oblem. At the beginning, we have no idea which one is relatively easy and which on e is hard and confusing. We randomly decomposite the tasks and of course, some of us got stuck. As described in the previous section, we have to flatten the board a ccording to the paper. When writing the code, it took us a long time figuring out the correct index number for the loop because finding index on a flattened matrix could be quite confusing, it violates common practice. While row constraint is the easiest one, the rest are time-consuming, especially the “clue constraints” beca use this constraint changes in every game.

Algorithm:

At the beginning, we tried to use sinkhorn as our main algorithm. We find some oth er people’ s implementation of sinkhorn that is written in tensorflow. We tried to translate into numpy since we are not quite familiar with tensorflow, but the prog ress was slow. Then we decided to use Linear Programming. We refer to the pseudo c ode in one of the papers. We tried to build the algorithm from scratches but we fo und that it’ s better to use scipy library (scipy.optimize.linprog) to help us set up the equation. We also ran into some index error during testing. Luckily, the co re part of the algorithm is quite short compared to the constraints, it’ s much ea sier to debug.

Solver:

There are a few steps following the order - first solver, second solver, third sol ver and so on…

As the first solver is implemented using the weighted LP1, if the first solver pro duces wrong solutions (at least one row, column or 3 x 3 grid contains duplicate n umbers, then another solver is used to improve the strategy to get the correct sol ution.

The following solver will delete repeated numbers in the Sudoku puzzle, and then k eep the remaining numbers as a new sudoku puzzle input. If the solver successfully solves the puzzle, then it will return the result. Otherwise, go to the next solve r with the same process (delete repeated numbers and so on). If the third solver s till doesn’ t improve the result, we go to the next successive solver by deleting repeated number, removing original clue numbers, select one number and add it to t he puzzle. If solved successfully, return the result. Otherwise, return the Sudoku is unsolvable.

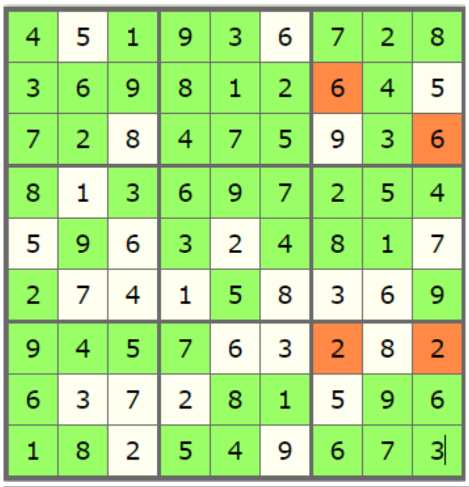
The solver is originally written in a while loop to check the status of the board. While in the paper, it suggests that after approximately 3 loops, we need to reorg anize the status of the board (aka, we consider it as a new sudoku game, just with more prompts) and then send it back to detective function.

Because till this end, we have several possible solutions. We tried to come up wit h an advanced strategy that could improve the performance of the existing solver, other than only delete the repeated cells, we delete all of the cells except the d efault clue, and choose some cells from the previous solutions that we come up, th en put these cells back in the sudoku as new clue, and finally solve for the new s udoku problem (Fourth solver). The fifth solver use previous 4 solutions to pick n ew clues.

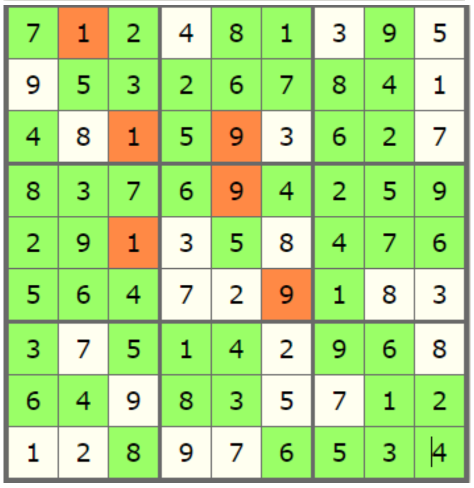
Results

Below are the sample results we exemplified from our test (two success cases and t wo failure cases).

(White = default, Green = correct, Red = wrong)
Failed Examples (compared with expected solution)

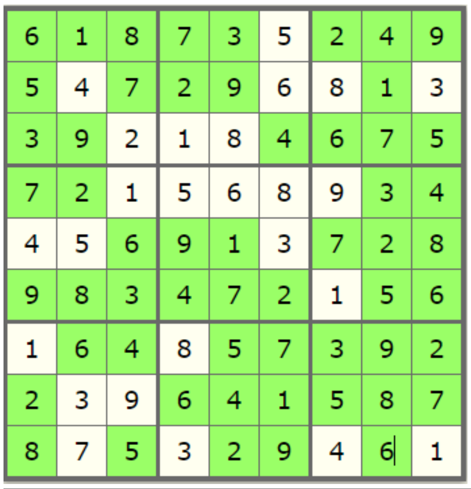


4	5	1	9	3	6	7	2	8
3	6	9	8	1	2	6	4	5
7	2	8	4	7	5	9	3	6
8	1	3	6	9	7	2	5	4
5	9	6	3	2	4	8	1	7
2	7	4	1	5	8	3	6	9
9	4	5	7	6	3	2	8	2
6	3	7	2	8	1	5	9	6
1	8	2	5	4	9	6	7	3

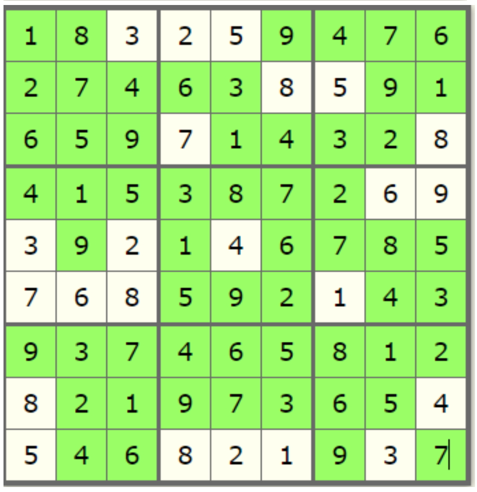


7	1	2	4	8	1	3	9	5
9	5	3	2	6	7	8	4	1
4	8	1	5	9	3	6	2	7
8	3	7	6	9	4	2	5	9
2	9	1	3	5	8	4	7	6
5	6	4	7	2	9	1	8	3
3	7	5	1	4	2	9	6	8
6	4	9	8	3	5	7	1	2
1	2	8	9	7	6	5	3	4

Succeeded Examples

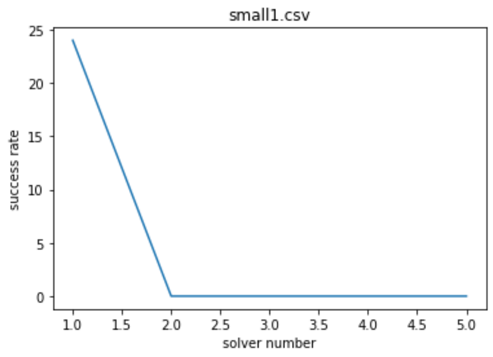


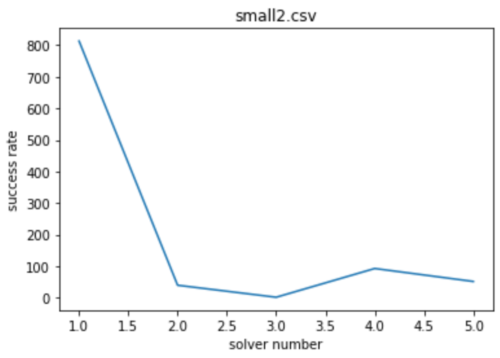
6	1	8	7	3	5	2	4	9
5	4	7	2	9	6	8	1	3
3	9	2	1	8	4	6	7	5
7	2	1	5	6	8	9	3	4
4	5	6	9	1	3	7	2	8
9	8	3	4	7	2	1	5	6
1	6	4	8	5	7	3	9	2
2	3	9	6	4	1	5	8	7
8	7	5	3	2	9	4	6	1



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

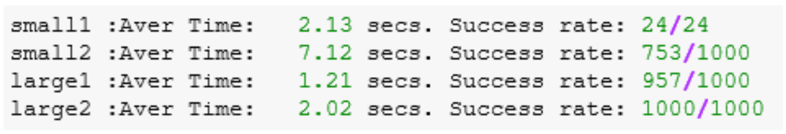
Below are two diagrams that record the number of sudoku solved by each solver, based on two datasets (small1,small2).





In “small1”, there are only 24 easy sudoku problems, and all of them only need to use weighted lp1 once to solve for the right answer. The second diagram is more interesting, and it proves the correctness of our improved strategy. From the diagram, most of the sudoku problems (80%) only need to use a weighted lp1 once to solve for the right answer. However, for the rest of the problems that need to use more solvers, we can tell from the diagram that the strategy from the paper (deletin g repeated cells) rarely improves the performance, while our strategy improved the performance by almost 18%.

Success rate against different dataset and its average runtime:



```
small1 :Aver Time:   2.13 secs. Success rate: 24/24
small2 :Aver Time:   7.12 secs. Success rate: 753/1000
large1 :Aver Time:   1.21 secs. Success rate: 957/1000
large2 :Aver Time:   2.02 secs. Success rate: 1000/1000
```

the average runtime and success-rate of small2 is obviously worse than the others because it contains problems that are considered harder than others, and this problem set is mainly used in our project to view the improvement of our strategy.

Baseline:

The result using only the strategy from Yuchao's paper [1]. Since we use different dataset than the paper, we tested their strategy against our own dataset, and it has 100%, 62.2%, 87.3%, 100% success rate relatively against "small1", "small2", "large1", "large2".

Improvement:

Success rate is shown above, and the improvement between each solver can also be easily seen from the diagrams above.

Conclusions

Instead of using normal linear programing, we chose Weighted LP1 to solve sudoku puzzles. By adding weights to each clue (cells that are not clue are initially zero), we can emphasize the influence of all clues without adding more constraints. Meanwhile, Weighted LP1 implicitly gives constraints priority so that we are much safer to allow the optimization method to change the clues because the weight will ensure that the change will be small enough.

Comparing to the strategy using only Weighted LP1 and a warm restart strategy, that keep deleting repeated cells and resolving the problem, our new strategy only keeps several cells that are the same during the process of warm strategy so that there will not be too many constraints for the new sudoku puzzles. Also, in warm restart strategy, all clue constraints are weighted so that it will be more likely to return the same wrong result that are provided in the previous loop. However, our strategy, even though not good enough to solve all problems, restricts the number of clues so that there will be less likely to include the wrong cells as our clue. Indeed, our strategy improves the performance of the whole sudoku solver by almost 20 percent.

Though our project is restricted to solve Sudoku puzzles, the Weighted LP1 we implemented and the strategy we improved can be applied to many problems that can be w ritten into linear minimization questions, such as Linear Assignment that we talke d about in class.

Bibliography

Group Members:
Luobin Wang
Yingyi Xu
Jason Huang
Chenhao Yang

References:

[1] Yuchao T., Zhenggang W. and Chuanxi Z.. "A Warm Restart Strategy for Solving Sudoku by
Sparse Optimization Methods". arXiv:1507.05995[math.OC]. https://arxiv.org/abs/1507.05995.
[2] Gabriel Peyré, Marco Cuturi. "Computational Optimal Transport". Chapter 4.2 Sinkhorn's
Algorithm and Its Convergence. Foundations and Trends in Machine Learning, vol. 11, no. 5-6,
pp. 355-607, 2019.
[3] Marco C., Olivier T., Jean-Philippe V.. "Differentiable Ranks and Sorting using Optimal
Transport". arXiv:1905.11885[math.OC]. https://arxiv.org/pdf/1905.11885.pdf. 2019.
[4] Google AI Research. "Differentiable Ranks and Sorting operators for Tensorflow".
https://github.com/nattimmis/GoogleAi/blob/999a7a7ce3af6051593b6e947fc251fbbb29bb23/
soft_sort/sinkhorn.py
[5] SciPy library, Linear programming.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html
[6] File reading with pandas. https://www.kaggle.com/gaz3113/sudoku-challenge-example-1
[7] Xiu Qin Deng. "A novel hybrid genetic algorithm for solving Sudoku puzzles".
Optim Lett (2013) 7: 241. https://doi.org/10.1007/s11590-011-0413-0.
[8] Hamidreza M., Sudoku solver by Genetic algorithm.
https://github.com/mahdavipanah/sudoku_genetic_python
[9] Dr. John M. Weiss. "Genetic Algorithms and Sudoku". MICS 2009.
https://github.com/mahdavipanah/sudoku_genetic_python/blob/master/paper.pdf
[10] Sudoku solver by Genetic algorithm. https://github.com/ctjacobs/sudoku-genetic-algorithm