Playing Sudoku with Optimization

Gabe Chapel

Abstract

In this report we describe how Sudoku problems can be solved using linear programming (LP). The development of the LP is discussed along with the resulting performance of the solver.

1 Introduction

Sudoku often presents itself in the daily newspaper and magazines as a way for readers to challenge themselves as they drink their morning coffee. What most people do not realize when they play Sudoku, though, is that they are actively performing optimization in their head, trying to optimize the placement of each number so that it satisfies a list of set rules, or constraints. Linear programming is a tool used for solving complex optimization problems that do not have a closed form solution and are, otherwise, too difficult to solve and Sudoku problems often fit this description. Thus, we formulate a general Sudoku solver as an LP and use a convex modeling system called *cvx* to solve it.

The most common variant of Sudoku uses a 9×9 grid, in which a player places numbers 1-9 to satisfy the following set of requirements:

- Every column must contain one, and only one, of each number
- Every row must contain one, and only one, of each number
- Every block must contain one, and only one, of each number
- Every grid cell must contain one, and only one, number
- The clues must remain unchanged

Blocks, also called boxes or regions, are the 3×3 subgrids that compose the grid. The clues are numbers initially filled in that determine the difficulty of the puzzle and give the player a starting point.[2] The player must fill in the empty cells, using these clues, as illustrated in the figures below.

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

Figure 1: Unsolved Sudoku with Clues

The blue numbers indicate the player's input to solve the puzzle and the blocks are separated by bolded lines. Notice that there are no repeated numbers in any rows, columns, or blocks.

To determine which numbers to input into which cells, the requirements can be transformed into a set of constraints to formulate an LP. The following sections describe this formulation and display the performance of the Sudoku solver.

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

Figure 2: Solved Sudoku

2 LP formulation

To formulate the LP, we begin with a simplified Sudoku problem, using a 4×4 grid with 2×2 blocks, as shown in the figure below. The clues are also provided.

		4	
1			
			3
	1		

Figure 3: Unsolved 4×4 Sudoku with Clues

Since Sudoku requires an input of numbers 1-9, or 1-4 in this simplified example, the problem involves integer optimization, which we do not have the tools to solve directly. Thus, we must find a way to represent the integer entries without constraining them to actually being integers 1-4. To do this, we create a binary representation for every cell, where each cell is provided with a four-element column vector. This vector, $\boldsymbol{x}_i$, contains either a 0 or a 1, where a 1 indicates which integer, 1-4, is in the cell. This is shown more clearly below.

$$\begin{aligned} \boldsymbol{x}_i &= \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^T \quad \text{entry} = 1 \\ \boldsymbol{x}_i &= \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}^T, \quad \text{entry} = 2 \\ \boldsymbol{x}_i &= \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^T, \quad \text{entry} = 3 \\ \boldsymbol{x}_i &= \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T, \quad \text{entry} = 4 \end{aligned} \tag{1}$$

The subscript $i = 1, 2, \ldots, N$ indicates the cell index, where N is the total number of cells in the grid. For this problem, we use the index convention as illustrated in the figure below.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure 4: Grid Cell Index Convention

With an unsolved Sudoku problem, the vectors $\boldsymbol{x}_i$ are unknown, so we can use them to build an array of variables $\boldsymbol{x}$ for all of the cells.

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_N \end{bmatrix} \tag{2}$$

This array is 64×1, since there are 16 cells that can have entries 1-4.

Although this does not completely eliminate the integer optimization issue, it reduces it to a point we can typically handle using norms. The objective of the optimization is to minimize the number of 1's in $\boldsymbol{x}$ while still satisfying the requirements introduced previously. This means we also aim to reduce as many variables in the array as we can to zero, which is a characteristic often provided by the 1-norm. Thus, the LP is designed to minimize $||\boldsymbol{x}||_1$ with the given constraints, which are derived in the following subsections. Note that all of these are equality constraints

2.1 Clues ($\boldsymbol{A}_{clues}\boldsymbol{x} = \boldsymbol{b}_{clues}$)

First, the clues must be defined so that they do not get overwritten. To do this, we designate rows in $\boldsymbol{x}$ for the clues by, essentially, setting it equal to an array $\boldsymbol{b}_{clues}$ that contains the locations of the clues. The array $\boldsymbol{b}_{clues}$ is 64×1 and is formatted like $\boldsymbol{x}$, in that every four rows represents a cell in the grid and each row indicates the entry to the cell. For example, the grid shown in Figure 4 has clues in cells 3, 5, 12, and 14 with respective entries 4, 1, 3, and 1. The locations of these clues in $\boldsymbol{b}_{clues}$ and $\boldsymbol{x}$ can be defined by

$$l = 4 * (i - 1) + j \tag{3}$$

where l is the index in $\boldsymbol{b}_{clues}$ and $\boldsymbol{x}$, i is the cell index, and j is the entry in the cell. This results in $\boldsymbol{b}_{clues}(12) = \boldsymbol{b}_{clues}(17) = \boldsymbol{b}_{clues}(48) = \boldsymbol{b}_{clues}(53) = 1$ and, therefor, $\boldsymbol{x}(12) = \boldsymbol{x}(17) = \boldsymbol{x}(48) = \boldsymbol{x}(53) = 1$ for the example. To define these clues in x, we build a matrix $\boldsymbol{A}_{clues}$ in which the diagonal has 1's in the same rows as $\boldsymbol{b}_{clues}$ and $\boldsymbol{x}$, so that

$$\boldsymbol{A}_{clues}\boldsymbol{x} = \boldsymbol{b}_{clues} \tag{4}$$

2.2 Cells ($\boldsymbol{A}_{cells}\boldsymbol{x} = \boldsymbol{b}_{cells}$)

Although it seems obvious, a constraint must be established to ensure that each cell has one, and only one, entry of an integer 1-4. To do this, we set the sum of the elements of each $\boldsymbol{x}_i$ to 1, so each cell only has either a 1, 2, 3, or 4. This can be shown by creating a 16×64 matrix $\boldsymbol{A}_{cells}$ in which the

rows represent the possible entries into an individual cell and the columns indicate the index of the entries in $\boldsymbol{x}$.

$$\boldsymbol{A}_{cells} = \begin{bmatrix} 1&1&1&1&0&0&0&0&0&0&0&0&\dots&0&0&0&0 \\ 0&0&0&0&1&1&1&1&0&0&0&0&\dots&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&1&1&1&\dots&0&0&0&0 \\ & & & & \vdots & \vdots & & & & & & & \ddots & & \vdots & \vdots & \\ 0&0&0&0&0&0&0&0&0&0&0&0&\dots&1&1&1&1 \end{bmatrix} \tag{5}$$

This is used with a 16×1 array of 1's, $\boldsymbol{b}_{clues}$ to define the cell constraint,

$$\boldsymbol{A}_{cells}\boldsymbol{x} = \boldsymbol{b}_{cells} \tag{6}$$

2.3 Columns ($\boldsymbol{A}_{cols}\boldsymbol{x} = \boldsymbol{b}_{cols}$)

To ensure that there is only one of each number 1-4 in every column, we set the sum of $\boldsymbol{x}_i$ for each column to $\mathbf{1}$. For the first column in this example, we write this as,

$$\boldsymbol{x}_1 + \boldsymbol{x}_5 + \boldsymbol{x}_9 + \boldsymbol{x}_{13} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}^T \tag{7}$$

since the first column has cell indices 1, 5, 9, and 13, as shown in Figure 2. This is the same for the other three columns in the grid. To formulate this as an equality constraint involving $\boldsymbol{x}$, we define a 16×64 matrix $\boldsymbol{A}_{cols}$ where the rows represent each possible entry, 1-4, for an individual column in the grid and the columns represent the location of the binary index in $\boldsymbol{x}$. To sum the possible entries in a column, $\boldsymbol{A}_{cols}$ follows the pattern as such:

$$\boldsymbol{A}_{cols} = \begin{bmatrix} I_{16} & I_{16} & I_{16} & I_{16} \end{bmatrix} \tag{8}$$

where I_{16} is a 16×16 identity matrix.

Using a 16×1 array of 1's, $\boldsymbol{b}_{cols}$, the column constraint is written as,

$$\boldsymbol{A}_{cols}\boldsymbol{x} = \boldsymbol{b}_{cols} \tag{9}$$

2.4 Rows ($\boldsymbol{A}_{rows}\boldsymbol{x} = \boldsymbol{b}_{rows}$)

The row constraint is similar to the column constraint, where we set the sum of $\boldsymbol{x}_i$ for each row to $\mathbf{1}$, in order to ensure that there is one, and only one, of each number 1-4 in every row. For the first row in this example, this can be written as,

$$\boldsymbol{x}_1 + \boldsymbol{x}_2 + \boldsymbol{x}_3 + \boldsymbol{x}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}^T \tag{10}$$

since the first row contains indices 1, 2, 3, and 4, as shown in Figure 2. Like in the column constraint, we define a 16×64 matrix $\boldsymbol{A}_{rows}$ where the rows represent each possible entry, 1-4, for an individual row in the grid and the columns represent the location of the binary index in $\boldsymbol{x}$. This matrix is shown below, where I_4 is a 4×4 identity matrix for each cell in the Sudoku grid.

$$\boldsymbol{A}_{rows} = \begin{bmatrix} I_4 & I_4 & I_4 & I_4 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & I_4 & I_4 & I_4 & I_4 & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ & & \vdots & \vdots & & & & \ddots & & \vdots & \vdots & & \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & I_4 & I_4 & I_4 & I_4 \end{bmatrix} \tag{11}$$

Using this with a 16×1 array of 1's, $\boldsymbol{b}_{rows}$, the rows equality constraint can be defined as,

$$\boldsymbol{A}_{rows}\boldsymbol{x} = \boldsymbol{b}_{rows} \tag{12}$$

2.5 Blocks ($\boldsymbol{A}_{blocks}\boldsymbol{x} = \boldsymbol{b}_{blocks}$)

Lastly, every block must contain one, and only one, of each number 1-4, so we set the sum of $\boldsymbol{x}_i$ for each block to $\mathbf{1}$. For the first block, this is written as,

$$\boldsymbol{x}_1 + \boldsymbol{x}_2 + \boldsymbol{x}_5 + \boldsymbol{x}_6 = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}^T \tag{13}$$

since the first block contains indices 1, 2, 5, and 6 in this example. Like with the column and row constraints, we create a 16×64 matrix $\boldsymbol{A}_{blocks}$ to accomplish this, however, the formulation of the matrix is not as simple, due to the orientation of the blocks within the grid. The rows of the matrix represent the possible entry, 1-4, for each individual block and the columns indicate the location of the binary index in x, but the pattern of the matrix is not trivial because of the indices of each cell in the blocks. The pattern is illustrated below.

$$\boldsymbol{A}_{rows} = \begin{bmatrix} I_4 & I_4 & \mathbf{0} & \mathbf{0} & I_4 & I_4 & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I_4 & I_4 & \mathbf{0} & \mathbf{0} & I_4 & I_4 & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ & & \vdots & \vdots & & & & \ddots & & & & \vdots & \vdots & & & & \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & I_4 & I_4 & \mathbf{0} & \mathbf{0} & I_4 & I_4 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & I_4 & I_4 & \mathbf{0} & \mathbf{0} & I_4 & I_4 \end{bmatrix} \tag{14}$$

Using this with a 16×1 array of 1's, $\boldsymbol{b}_{blocks}$, the rows equality constraint can be defined.

$$\boldsymbol{A}_{blocks}\boldsymbol{x} = \boldsymbol{b}_{blocks} \tag{15}$$

2.6 Final LP

With all constraints defined, a final LP can be constructed as follows:

$$\begin{array}{ll} \text{minimize} & ||\boldsymbol{x}||_1 \\ \text{subject to} & \boldsymbol{A}_{clues}\boldsymbol{x} = \boldsymbol{b}_{clues} \\ & \boldsymbol{A}_{cells}\boldsymbol{x} = \boldsymbol{b}_{cells} \\ & \boldsymbol{A}_{cols}\boldsymbol{x} = \boldsymbol{b}_{cols} \\ & \boldsymbol{A}_{rows}\boldsymbol{x} = \boldsymbol{b}_{rows} \\ & \boldsymbol{A}_{blocks}\boldsymbol{x} = \boldsymbol{b}_{blocks} \end{array} \tag{16}$$

Notice, we do not include a constraint for $0 \leq \boldsymbol{x} \leq 1$ because the equality constraints already accomplish this. The LP can now be input into a solver or modeler to obtain the the optimal $\boldsymbol{x}$.

3 Solving the LP

To simplify the process of solving the LP, we use a modeling system for convex programming called cvx.[1] This modeler reduces the need for reformatting LPs and supports the input of individual constraints, like those shown in Equation 16. Thus, we simply input this equation and let cvx's SeDuMi solver obtain the optimal solution. Although speed is not much a concern for a problem of this size, we use SeDuMi instead of SDPT3 because it is more time efficient.

4 Results

As stated previously, the results are obtained through a cvx modeler. The solution of the modeler, however, is still the cell entries in binary form, so we transform it into a 16×4 list of $\boldsymbol{x}_i$ vectors and search for nonzero terms.

Each row in this list,

$$\text{binaryArray} = \begin{bmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix} \tag{17}$$

represents a cell in the Sudoku grid and each column indicates the value to be entered into the cell. We use MATLAB's *find()* function to determine the index of the nonzero terms and, therefor, the cell entry. This is reshaped into the initial Sudoku matrix to present the solved puzzle. The solution we obtain for the 4×4 example is shown below with the blue numbers, again, representing the player's inputs and the black showing the initial clues.

2	*3*	4	*1*
1	*4*	*3*	*2*
4	*2*	*1*	3
3	1	*2*	*4*

Figure 5: Solved 4×4 Sudoku

As shown in the figure, the solution satisfies the constraints, where every row, column, and block includes one, and only one, of each number 1-4. The clues also remain the same as when introduced in Figure 4. Since this displays a functional solver for a simple 4×4 Sudoku, we adapt it to solve the typical 9×9 Sudoku. This is done by simply changing every 4 to 9, 16 to 81, and 64 to 729 in the LP formulation. This means that $\boldsymbol{x}$ is a 729×1 array. The performance can be shown by solving Sudoku problems at different difficulty levels, as illustrated below.

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

Figure 6: Medium Level Sudoku

The solutions to each of these problems satisfy the constraints and prove that the solver performs as expected. This, however, is only true up to a 9×9 Evil Difficulty problem.

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

Figure 7: Evil Level Sudoku

5 Limitations

When testing with a harder problem, denoted the *Hardest Sudoku Ever*, the solver fails to produce a solution with only 0's and 1's, though the solutions remain between 0 and 1. This means there are multiple nonzero values for each cell and the solver does not converge to an answer. We attempted to use the data and, instead of searching for indices of nonzero terms in binaryArray, found the maximum values in each row. Using these for the Sudoku inputs did not work, however, and resulted in repeats in rows, columns, and blocks. We also tried testing two larger 16×16 Sudoku puzzles, an *Easy* and a *Medium*, but found the same result. These are not displayed here, but can be found in the Limitations section of the code in the Appendix.

6 Conclusion

As Sudoku problems become more challenging, they begin to better fit the criteria for using optimization for solving. We presented a method with which to formulate Sudoku problems into LPs and, using cvx's SeDuMi solver, provided the resulting performance. Since the Sudoku constraints involve integer optimization, we developed a way to transform the problem into a binary problem and built constraints that aim to limit the solution to either a 0 or a 1. The optimization technique proved capable of solving a 9×9 problem with a difficulty level up to Evil. We observed that the technique failed, though, when attempting to solve the *Hardest Sudoku Ever* and larger 16×16 puzzles. This is because we do not actually use integer optimization even though the solution relies on the results being either 0 or 1, so when the constraints cannot force the results to be integers, the solver fails. To remedy this, we would need to use a more complex integer optimization tool and establish tighter constraints. When successful, however, we illustrated that Sudoku is actually an optimization problem that people often attempt in their heads and can be solved simply using linear programming.

A Code

A.1 Example Problems

```
%%%%%%%%%%%%%%%% EXAMPLE PROBLEMS %%%%%%%%%%%%%%%%%
% % % 4x4 Example
% MatrixInitial = [0 0 4 0;
%                  1 0 0 0;
%                  0 0 0 3;
%                  0 1 0 0];
% % %MEDIUM LEVEL
% MatrixInitial = [0 5 0 0 2 0 3 7 0;
%                  0 3 0 9 4 0 0 0 1;
%                  0 0 0 7 0 0 0 0 0;
%                  0 0 5 8 0 0 9 2 0;
%                  3 0 0 0 0 0 0 0 5;
%                  0 7 8 0 0 9 1 0 0;
%                  0 0 0 0 0 2 0 0 0;
```

```
%                   8 0 0 0 7 6 0 5 0;
%                   0 2 1 0 8 0 0 6 0];

% %EVIL LEVEL
% MatrixInitial = [0 6 9 7 0 0 4 3 0;
%                   0 1 0 0 0 0 0 7 0;
%                   3 0 0 0 0 5 0 0 2;
%                   0 3 0 0 0 0 0 0 1;
%                   0 0 0 0 9 0 0 0 0;
%                   6 0 0 0 0 0 0 2 0;
%                   7 0 0 2 0 0 0 0 3;
%                   0 9 0 0 0 0 0 4 0;
%                   0 4 2 0 0 3 5 1 0];

% %EVIL LEVEL
% MatrixInitial = [0 9 0 4 0 8 5 0 0;
%                   0 0 0 0 0 0 0 0 6;
%                   2 0 1 0 7 0 9 0 0;
%                   5 0 0 0 8 0 0 0 7;
%                   0 0 7 9 0 4 1 0 0;
%                   8 0 0 0 2 0 0 0 9;
%                   0 0 2 0 3 0 4 0 5;
%                   4 0 0 0 0 0 0 0 0;
%                   0 0 5 8 0 7 0 9 0];
```

A.2 Limitations

```
%%%%%%%%%%%% LIMITATIONS %%%%%%%%%%%%%%
% % 16x16 Easy Example %%DOES NOT WORK
% MatrixInitial = [ 0  0  3  6  0  8  0  0  5 14  0  9  0  0  0 15;
%                   0  1  2  7 11  0  0  6  0  8 12 16  5 14 13  9;
%                   9  0 14 13 15  1  2  7  0  4  3  0 10  8 12 16;
%                  16 10  8 12  9  0  0 13 15  0  2  7 11  4  3  6;
%                   6 11  4  3  0 10  8 12  9  5 14  0 15  0  0  7;
%                   7  0  1  0  0 11  4  3  0 10  0 12  9  5 14 13;
%                  13  9  5 14  0  0  0  2  6  0  0  0 16  0  0 12;
%                  12  0 10  8 13  9  5 14  7 15  1  2  6 11  4  3;
%                   3  0  0  4 12  0 10  0 13  9  5 14  0  0  1  0;
```

```
%                    2  7  0  0  0  6  0  4 12 16  0  0 13  9  5 14;
%                    0 13  0  0  2  7 15  1  0  6 11  4 12  0 10  8;
%                    0 12  0 10 14  0  9  5  2  7 15  1  3  6  0  4;
%                    4  0  6  0  0 12 16 10  0  0  9  0  0  0  0  1;
%                    1  2  7 15  4  3  6 11  8 12 16 10 14  0  9  0;
%                    0 14  0  9  1  2  0 15  4  0  6  0  8 12 16 10;
%                   10  8  0 16  5 14 13  0  0  2  0  0  4  3  6 11];

% % %16x16 Medium Example %%DOES NOT WORK
% MatrixInitial = [ 4  0  0  0  8  1  0  0  0 11 12  0  0  0 13  0;
%                    6  9  0 13  0  2  7  5  8  0  0 10 15 11  0 14;
%                    0 15 11 12  0  0  0  0  4  2  7  0  0  0 16 10 ;
%                    0  0  0 16 14  0  0 12  6  0  3 13  0  2  7  5;
%                    5  4  2  7 10  8  1 16  0 15 11  0  0  0  0 13;
%                   13  6  0  0  0  0  0  7  0  0  1  0  0 15  0 12;
%                    0 14 15  0  0  6  0  3  0  4  0  0 10  8  1 16;
%                    0  0  0  1 12  0  0  0  0  0  9  0  0  4  0  7;
%                    7  0  4  2 16 10  8  1 12  0 15 11 13  6  0  0;
%                    0 13  6  9  0  5  4  0 16  0  0  1 12 14  0  0;
%                   11  0  0  0  3 13  0  0  7  5  4  2 16 10  8  0;
%                    0  0  0  8  0 12  0 15  3  0  0  9  0  5  0  0;
%                    0  7  0  4  1  0 10  8 11  0 14  0  0  0  0  9;
%                    0  3  0  0  0  0  0  4  0  0  0  8  0  0 14  0;
%                    0 11 12  0  9  3 13  6  2  0  0  4  0 16 10  8;
%                    8  1 16  0  0  0  0 14  0  3 13  6  0  0  5  0];

% % %Hardest Sudoku Ever %%DOES NOT WORK
% MatrixInitial = [8 0 0 0 0 0 0 0 0;
%                   0 0 3 6 0 0 0 0 0;
%                   0 7 0 0 9 0 2 0 0;
%                   0 5 0 0 0 7 0 0 0;
%                   0 0 0 0 4 5 7 0 0;
%                   0 0 0 1 0 0 0 3 0;
%                   0 0 1 0 0 0 0 6 8;
%                   0 0 8 5 0 0 0 1 0;
%                   0 9 0 0 0 0 4 0 0];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.3 Constraints

```
N=length(MatrixInitial);
NN = N*N; % Number Cells

%NOTE: each cell has 9 possible numbers that can
%      be chosen.  So the first 9 variables refer
%      to cell (1,1).  If for example the (1,1) cell
%      is a 4, than x(1:9)=[0 0 0 1 0 0 0 0 0] etc
NNN = N*N*N; %Total number of binary variables x

% Find clues: Aclues*x=bclues
ArrayInitial = reshape(MatrixInitial', 1, NN); % Reshape initial matrix to array
bClues = zeros(NNN, 1); % b array for the clues
AClues = zeros(NNN,NNN); % A matrix for clues
for i = 1:NN
cellVal = ArrayInitial(i);
if cellVal ~= 0
bClues((i-1)*N+cellVal) = 1;
AClues((i-1)*N+cellVal,(i-1)*N+cellVal) = 1;
end
end
arrayCheck = reshape(ArrayInitial, N, N)';

% Constrain one number in each space: ASpaces*x=bSpaces
bCells = ones(NN,1);
ACells = zeros(NN,NNN);
for i = 0:NN-1
ACells(i+1, N*i+1:N*(i+1)) = 1;
end

% Eliminate duplicates in columns: AColumns*x=bColumns
bColumns = ones(NN,1);
AColumns = [];
for i = 1:N
AColumns = [eye(NN) AColumns];
end
% Eliminate duplicates in rows: ARows*x=bRows
```

```
bRows = ones(NN,1);
for k = 0:N-1
for i = 1:N
for j = 0:N-1
ARows(i+N*k, j*N+i+NN*k) = 1;
end
end
end

% Eliminate duplicates in NxN blocks
bBlocks = ones(NN,1);
n = sqrt(N);
for l = 0:n-1
for h = 0:n-1
for i = 0:N-1
for j = 0:n-1
for k = 0:n-1
ABlocks(i+1+(h*N)+(l*n*N), N*k+1+(j*NN)+i+(h*n*N)+(l*n*NN)) = 1;
end
end
end
end
end
end
```

A.4 Solver

```
%% Use CVX to Solve LP
% Add cvx files to path
addpath /Users/gabrielchapel/Downloads/cvx
addpath /Users/gabrielchapel/Downloads/cvx/structures
addpath /Users/gabrielchapel/Downloads/cvx/lib
addpath /Users/gabrielchapel/Downloads/cvx/functions
addpath /Users/gabrielchapel/Downloads/cvx/commands
addpath /Users/gabrielchapel/Downloads/cvx/builtins
cvx_begin
cvx_precision low % Low precision for faster performance
variables x(NNN) % Set up variables for LP
minimize norm(x,1) % Linear program
```

```
subject to % Constraints
AClues * x == bClues;
ACells * x == bCells;
AColumns * x == bColumns;
ARows * x == bRows;
ABlocks * x == bBlocks;
cvx_end

for i = 0:NN-1
binaryArray(i+1,1:N) = x(i*N+1:(i+1)*N);
end
for i = 1:NN
ArrayFinal(i) = find(binaryArray(i,:));
end
MatrixFinal = reshape(ArrayFinal, N,N)'
```

References

[1] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx, March 2014.

[2] Shalom Ruben. Optimal design lecture, May 2018.