Sudoku Solving with Linear Optimization

Max Ptasiewicz

April 30, 2022

1 Introduction

Sudoku is a game which begins on a 9 by 9 grid, with several digits 1 through 9 placed in the grid. The goal is to fill in the grid such that each row, each column, and each 3 by 3 box in the grid contain exactly the digits 1 through 9, without doubles or missing digits. This is generally done by hand, using the given clues to eliminate possibilities from each box. We wanted to skip this part, as it takes effort and brainpower. Our goal is to create a solver that can take the puzzle clues and output a solved puzzle. There are several ways to do this. The simplest idea is a fully brute force solution, where every possible combination of numbers is tried in the puzzle, until the one that works with the clues is determined. This takes approximately 1×10^{25} iterations and checks, which already exceeds the maximum number of iterations MATLAB will do in a single loop. Additionally, the amount of computational power required to do this is immense, and would take several hours, making this option the least efficient. There is another algorithm that uses a backtracking method, which is a slightly smarter brute force technique. This will try a 1 in the first empty box, and then treat that as a clue and solve the puzzle from there, which would then assume the first empty box is a 1, and go until it gets to the end. If a number is found to break the rules, it is deemed that the box initially tested must not be a 1, and then it is changed to a 2, and the process is repeated. This method is much more efficient than a pure brute force solution, but does not use any optimization techniques. However this method does guarantee a solution if the given clues are valid. The technique I will be using to solve sudoku puzzles is an optimization technique that involves including all of the rules of sudoku as linear constraints, and letting a linear programming solver work from there. This method is extremely fast, depending on the solver used, as it only has to solve one linear programming problem. The main issue with it is that it will not guarantee a solution in its current state, as the hardest of puzzles will not allow the solver to converge to the correct solution. A solution will happen, but it will break the rules of sudoku and will not be a valid solution. This is only true of the hardest puzzles out there; for most puzzles easier than this it will solve correctly.

Before we begin, there are a few vocabulary terms used in this paper that will be helpful to define beforehand:

- Row $\rightarrow$ A single row of the puzzle that spans all of the sub-boxes, highlighted in brown.
- Column $\rightarrow$ A single column of the puzzle that spans all of the sub-boxes, highlighted in blue.
- Box $\rightarrow$ A single box of the puzzle that represents a ninth of the total puzzle, highlighted in orange.
- Cell $\rightarrow$ A single box of the puzzle, which contains 81 cells, highlighted in pink.
- Clue $\rightarrow$ A given digit in the puzzle, used to deduce the unknown digits, drawn in black.
- Guess $\rightarrow$ A guessed digit in the puzzle, guessed from the given clues in the puzzle, drawn in green.
- N $\rightarrow$ The size of a puzzle, in this paper either a 4 or a 9.

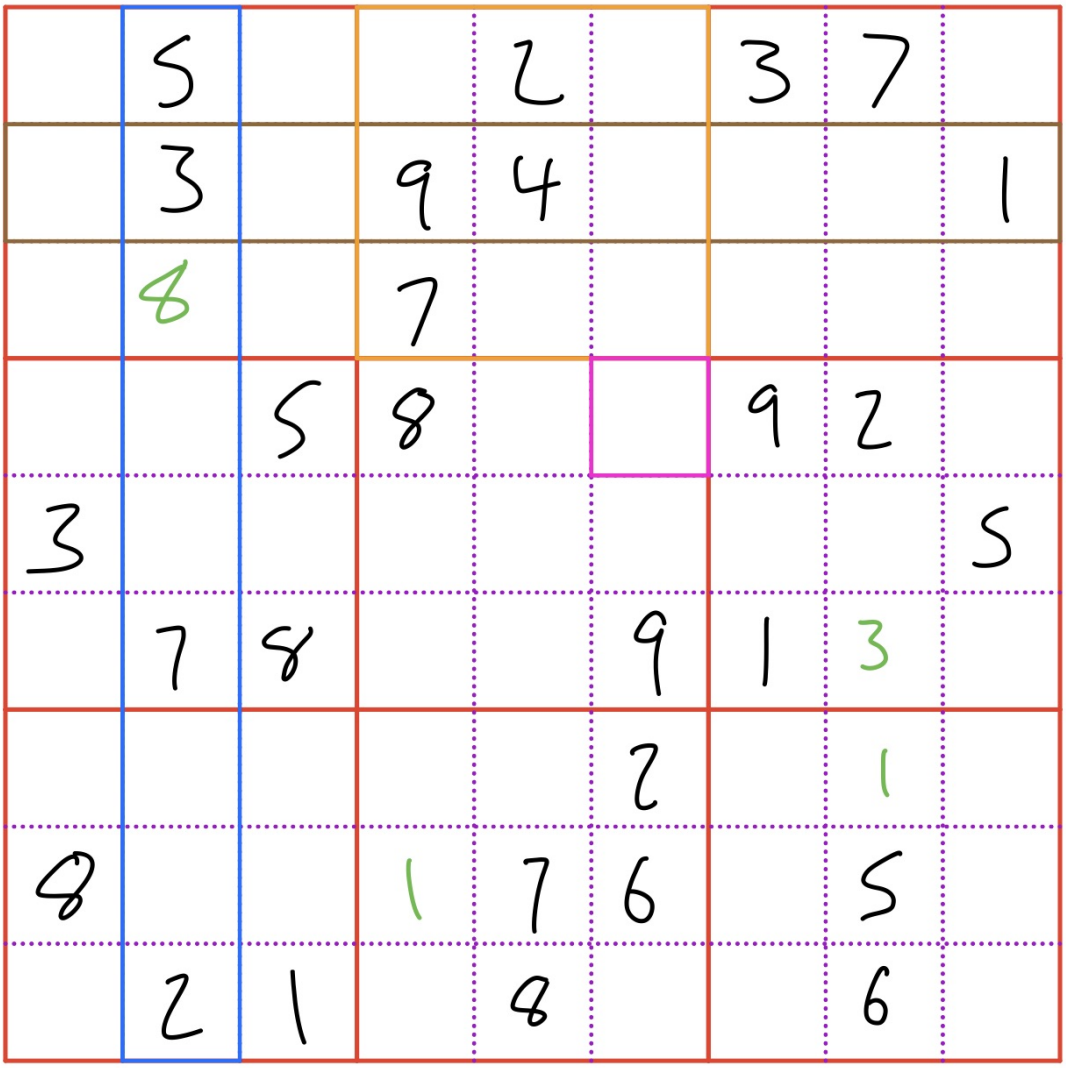

Figure 1: Example sudoku puzzle

2 Formulation

The formulation of this problem essentially involves programming the rules of sudoku as linear constraints. There are really five main types of constraints, but two of those constraints have three subsets within them. The five main types are that each number placed in any cell must be an integer, that the digits in all rows, columns, and boxes must sum up to 45, there must be 9 unique digits in each row, column, and box, a constraint that outlines where the clues are and what effect

they have on the puzzle, and finally there must be only one digit placed in any cell. This problem was formulated entirely using a small 4 by 4 puzzle, in order to more easily visualize the constraints and be able to count each constraint more accurately. This small puzzle also helped to see the pattern that was a part of each constraint.

- First, the integer constraint. There are several integer optimization solvers, but the goal of this project was to implement the integer constraint without actually using an integer solver. Let's use the small test case as an example: There are four possibilities for each cell, a 1, 2, 3, or a 4. Instead of trying to calculate a 1, 2, 3, or a 4, we decide to assign each possibility for a single cell a binary value, either 1 or 0. There are then essentially four binary values for each cell, which expands our total number of unknowns from 16, in the 16 cells in a puzzle, to 64, with now four binary variables for each cell. This means that for any solved cell, the solution will contain a set of three zeros and a single one. For example, if there were a 4 placed in the third cell in the first row, that would say that variables x_9, x_{10}, x_{11} are all zeros, and x_{12} would be a 1. This can then be expanded to a 9 by 9 puzzle, where each of 81 cells would have 9 variables assigned to it, for a total of 729 variables in the final solution.
- Second, the sum of each row, column and box must add up to 45, or 10 in the small test case.
  - Each row summing up to 10 was the problem tackled first. This turned out to be simpler than the other summing constraints, just based on the layout of the variables and location of the rows. It turns out that the first 16 variables of the 64 corresponded to the first row, the second 16 corresponded to the second row, and down the line. Then the trick of having a set of binary variables summing up to 10 had to be implemented. This was done through the following equations:

$$\begin{aligned} 1(x_1) + 2(x_2) + 3(x_3) + 4(x_4) + \\ 1(x_5) + 2(x_6) + 3(x_7) + 4(x_8) + \\ 1(x_9) + 2(x_{10}) + 3(x_{11}) + 4(x_{12}) + \\ 1(x_{13}) + 2(x_{14}) + 3(x_{15}) + 4(x_{16}) = 10 \end{aligned} \quad (1)$$

    This is then repeated with the next three sets of 16 variables, and then scaled up to a 9 by 9 puzzle. This works, as every time there is a binary quadruple for a single cell, three of the values will be zero, and one will be a one, and the one that is not a zero will be multiplied by the value corresponding to its number, and adding the correct value to the sum of the row.
  - The second summation constraint tackled was the column sum. This was fairly similar to the row constraint, the only difference was the variables grouped together in each equation. This is due to the order of variables starting with the entire top row, and then going to the second row, and so forth through the puzzle. This time the counting was different, so that every fourth binary quadruple would be part of the same row. The resulting equation is as follows:

$$\begin{aligned} 1(x_1) + 2(x_2) + 3(x_3) + 4(x_4) + \\ 1(x_{17}) + 2(x_{18}) + 3(x_{19}) + 4(x_{20}) + \\ 1(x_{33}) + 2(x_{34}) + 3(x_{35}) + 4(x_{36}) + \\ 1(x_{49}) + 2(x_{50}) + 3(x_{51}) + 4(x_{52}) = 10 \end{aligned} \quad (2)$$

This is again repeated for the next binary quadruples, just as the row constraints were. Similarly, this is scaled up to a 9 by 9 puzzle, and instead of skipping to the variable after the 16th, (N^2), we skip to the variable after the 81st, to make sure each column starts in the correct place.

- The final and by far the hardest of the summation constraints is that of the boxes. Essentially the entirety of the final matrix had to be planned out initially in order to match up the correct cells with each other. This results in the following equation, although it does not display much of a pattern:

$$\begin{aligned} 1(x_1) + 2(x_2) + 3(x_3) + 4(x_4) + \\ 1(x_5) + 2(x_6) + 3(x_7) + 4(x_8) + \\ 1(x_{17}) + 2(x_{18}) + 3(x_{19}) + 4(x_{20}) + \\ 1(x_{21}) + 2(x_{22}) + 3(x_{23}) + 4(x_{24}) = 10 \end{aligned} \quad (3)$$

This does not help much with realizing the pattern, the best way to visualize it is to use a representative matrix. The unit used to form the matrix is something I will call "short", and represents a single row of a box. For a 4 by 4, this would be 1 2 3 4 1 2 3 4, and for a 9 by 9, this would be the digits 1 through 9, laid out next to each other three times instead of twice, as there are three cells in a row of a box. For the 4 by 4, the final matrix is as follows:

$$\begin{bmatrix} short & 0 & short & 0 & 0 & 0 & 0 & 0 \\ 0 & short & 0 & short & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & short & 0 & short & 0 \\ 0 & 0 & 0 & 0 & 0 & short & 0 & short \end{bmatrix}$$

And for the 9 by 9, the top left ninth of the matrix is as follows:

$$\begin{bmatrix} short & 0 & 0 & short & 0 & 0 & short & 0 & 0 \\ 0 & short & 0 & 0 & short & 0 & 0 & short & 0 \\ 0 & 0 & short & 0 & 0 & short & 0 & 0 & short \end{bmatrix}$$

This matrix is then tiled along the diagonal of a larger matrix, as if it were an identity matrix, and it is tiled a total of three times for the 9 by 9, as opposed to twice for the 4 by 4. Then the only task was creating this matrix, which was just done in MATLAB, using code shown at the end. The hardest part was making this work with both a 4 by 4 and a 9 by 9, but was accomplished by relating every hardcoded number as a function of N, or the size of the puzzle (4 or 9).

- Next, we get to the uniqueness constraint of each row, column, and box. This was actually significantly easier than the summation constraint, and was done for rows, columns, and boxes all at the same time. We start by thinking that there can be only a single one in the first row. That means that for each binary quadruple in the row, the binaries that represent each 1 must sum up to one, and the binaries that represent each 2 must sum up to one, and so on for each number. This constraint then consists of replacing each of the same digit with a 1, and everything else with zeros, for every unique digit in the puzzle. For the 4 by 4, this meant we were left with four 4 by 64 matrices from each initial 4 by 64 matrix that represented the

row constraints. This identical technique was applied to the columns and boxes, as all of the digits were in the correct place, and the matrices simply had to be searched through and new ones created using locations found in the original matrices. All of this could be easily scaled up to a 9 by 9 puzzle, again by defining all lengths of loops based on N.

- Now we have to create hard constraints based on the given clues in the puzzle. This was rather simple, as whenever a clue was found in the puzzle, that binary quadruple was determined, and then the equation set resulted in something as simple as this:

$$\begin{aligned} x_1 &= 0 \\ x_2 &= 0 \\ x_3 &= 1 \\ x_4 &= 0 \end{aligned} \tag{4}$$

  This would be set if the clue in cell 1 was a 3. This tells our solver more than just that cell 1 is a 3, it also says that it is not a 1, 2, or 4, by assigning those values to zero. The constraint is just an identity matrix with some of the ones on the diagonal made into zeros, corresponding to wherever a clue did not appear in the puzzle. The b vector for this was slightly harder, but only required that it be made alongside the identity matrix, consisting of the correct binary quadruple for whatever the clue was. This was then again scaled up to a 9 by 9 by basing all constants on N.

- Our final constraint is one slightly harder to see that it is required, but it is extremely necessary. We need to assure that only a single digit is placed in each cell, and this is done by saying the sum of each binary quadruple is 1. Again easily expanded to a 9 by 9.

The only thing left is to stack each A matrix and b vector created in the correct order, and then run it through a solver. The only real constraint, once the A and b had been created, is the equality constraint $Ax = b$. This was fed into the cvx solver mosek, and the results were re-interpreted into a solution matrix, representing the solver's final guess for the puzzle. Additionally, the cost function for this problem was the 1-norm of the solution vector, in order to calculate the most zeros in the final results.

3 Analysis

I decided to formulate my solver as two different functions, one that would take in the unsolved puzzle with zeros wherever unknown digits were, and the clues as part of that matrix and then it would output the solved matrix. The second function is just a printing function, where it would take in any grid, either a 4 by 4 or a 9 by 9, and print the puzzle in an organized format. This was then run on each of five puzzles; first the 4 by 4 on which the entire solver was developed, and second the 9 by 9 puzzles given in the project. The resulting solutions are shown below: (Keep in mind that the results are the outputs as printed by the printing function in MATLAB, so the formatting translation to LaTeX is a little strange. The goal of the printing function is to make the output puzzles look nice in the command window of MATLAB).

- First, the 4 by 4 puzzle on which everything was tested:

23	41
14	32
42	13
31	24

  This puzzle was completed correctly, as one would hope given how it led to the development of the entire algorithm.

- Second, the first 9 by 9 puzzle given, rated as an intermediate level:

159	628	374
732	945	681
684	731	592
415	863	927
396	217	845
278	459	136
567	392	418
843	176	259
921	584	763

  This puzzle was solved perfectly, confirmed both by inspection and through using an online sudoku solver. I did attempt to solve it by myself, but a few minutes in I got bored and wanted to do more code.

- Next, the first 9 by 9 evil puzzle given, which would be much harder to solve by hand:

269	718	435
415	329	678
387	465	192
934	682	751
528	197	364
671	534	829
756	241	983
193	856	247
842	973	516

  The evil puzzle was also solved perfectly, confirmed by an online solver, and in a similar time to the initial intermediate puzzle (not that this was a lot of time, about half a second per solve).

- Next, the second 9 by 9 evil puzzle given, just another test using a difficult puzzle:

693	418	572
754	392	816
281	576	934
549	183	627
327	964	158
816	725	349
972	631	485
438	259	761
165	847	293

  Again, the evil level puzzle was solved perfectly, confirmed by an online solver.

- Finally, the hardest sudoku puzzle ever created:

814	753	659
923	658	177
675	491	283
154	937	841
386	845	721
787	129	535
531	974	568
468	564	917
697	816	452

  As you can see, the hardest sudoku ever was ultimately too difficult for the solver to reconcile, which was expected.

4 Discussion

Each of the easier puzzles were expected to solve using this type of optimization solver, and I believe that was due to the coding of the rules of sudoku. If each rule of the puzzle is coded correctly, it should solve perfectly no matter what. The reason I expected the harder puzzle not to be solved is because it requires more than the basic rules of sudoku to solve. I am not privy to these techniques, as I am not a professional sudoku solver, nor do I believe those more complex techniques could be implemented using this optimization strategy, as I would assume they are conditional to specific situations during a puzzle solve. The error in the solution vector of the hardest puzzle ever yielded interesting results, in that the binary values were not only zeros and ones, but rather decimals between zero and one. These decimals presumably obeyed the constraints, as all of the constraints were based on only binary values being possible. This led to the largest value in a single binary set representing a cell being a decimal, and not representing the actual value the cell was supposed to have. This same error was noticed in development of the puzzle, and led to similar errors in the guesses. The hardest puzzle ever could easily be solved by either of the two brute force solutions discussed in the introduction, but that defeats the purpose of the optimization technique. Either way, the optimization technique does have the added benefit of being extremely fast, although the online solvers were of a similar speed, so I would not say that one should sacrifice the accuracy of

the backtracking method for a possible speed increase. This did however make for an interesting problem, and some extremely entertaining matrices to create, especially when trying to make the algorithm applicable to several sizes of puzzle. Overall, the project was a success, and the solver worked as expected.

5 Conclusion

As stated previously, the solver solved all of the puzzles that it was supposed to solve, and failed on the one that it should have failed on. There are two possibilities for how to remedy this; one being implementing the previously discussed more advanced sudoku techniques that only extremely talented solvers know, and the other could be using an integer solver. The integer solver would disallow the existence of the decimal values that would satisfy the constraints, but not make a solved puzzle. Additionally, the use of the 1-norm could be amended to using the 0-norm, which represents the solution with the most zeros in the final vector. This technique was actually tried with the hardest puzzle ever, but yielded even worse results. In the future, it would be interesting to try this solver on a larger puzzle, like a 16 by 16 or a 25 by 25, just to see what would happen. Furthermore, I would be curious to see what an integer solver could do with these puzzles, especially the hardest sudoku ever. I would think that it could somewhat rectify the issues in the non integer solver.

6 Code

This is the driver function that tested each of the puzzles and gave the results:

```
clc
clear
close all

% this is a driver script which calls my solver function using each of the
% test cases, and prints results to the command window

tic

test4x4 = [0 0 4 0;
           1 0 0 0;
           0 0 0 3;
           0 1 0 0];
MatrixInitial1 = [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
MatrixInitial2 = [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
MatrixInitial3 = [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];

% %Hardest Sudoku Ever
MatrixInitial4 = [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];

test1 = sudokusolver(test4x4);
test2 = sudokusolver(MatrixInitial1);
test3 = sudokusolver(MatrixInitial2);
test4 = sudokusolver(MatrixInitial3);
test5 = sudokusolver(MatrixInitial4);

fprintf('Solution of the 4x4 puzzle:\n')
printpuzzle(test1)
fprintf('Solution of the intermediate 9x9:\n')
printpuzzle(test2)
```

```
fprintf('Solution of the first evil 9x9:\n')
printpuzzle(test3)
fprintf('Solution of the second evil 9x9:\n')
printpuzzle(test4)
fprintf('Solution of the hardest 9x9 ever:\n')
printpuzzle(test5)

toc
```

Here is the function that actually solves the puzzle, regardless of size, and outputs the solved puzzle in matrix form:

```
function [resultmat] = sudokusolver(puzzle)

N = size(puzzle,1);

% figure out number of clues for later
numclues = 0;
for v = 1:size(puzzle,1)
    for w = 1:size(puzzle,2)
        if puzzle(v,w) ~= 0
            numclues = numclues + 1;
        end
    end
end

% let's make some constraints

% rows

rowcon = zeros(N,N^3);
for i = N^2*(1:N) - N^2 + 1
    rowint = zeros(1,N^3);
    for j = N*(1:N) - N + 1
        rowint((i + j - 1):(i + j + N - 2)) = 1:N;
    end
    rowcon((i - 1 + N^2)/N^2,:) = rowint;
end

rowb = sum(1:N)*ones(N,1);

% columns

colcon = zeros(N,N^3);
for k = N*(1:N) - N + 1
```

```
    colint = zeros(1,N^2);
    colint(k:(k + N - 1)) = 1:N;
    colintlong = zeros(1,N^3);
    for l = N^2*(1:N) - N^2 + 1
        colintlong(l:(l + N^2 - 1)) = colint;
    end
    colcon((k - 1 + N)/N,:) = colintlong;
end

colb = sum(1:N)*ones(N,1);

% boxes

boxcon = zeros(N,N^3);
for m = 1:N
    boxintshort = zeros(1,sqrt(N^3));
    for n = N*(1:sqrt(N)) - N + 1
        boxintshort(n:(n + N - 1)) = 1:N;
    end
    boxintmid = zeros(sqrt(N),N^3/sqrt(N));
    for p = (1:sqrt(N))*sqrt(N)^3 - sqrt(N)^3 + 1
        for q = (1:sqrt(N))*N^2 - N^2 + 1
            boxintmid((p - 1 + sqrt(N)^3)/sqrt(N)^3,
            (p + q - 1):(p + q + length(boxintshort) - 2))
            = boxintshort;
        end
    end
    s = 1;
    for r = (1:sqrt(N))*sqrt(N) - (sqrt(N) - 1)
        boxcon(r:(r + sqrt(N) - 1),s:(s + N^3/sqrt(N) - 1)) = boxintmid;
        s = s + N^3/sqrt(N);
    end
end

boxb = sum(1:N)*ones(N,1);

% sum of each number set must be 1, can't have multiple numbers

sumnumcon = zeros(N^2,N^3);
for t = 1:N^2
    sumnumcon(t,(N*t - N + 1):(N*t)) = ones(1,N);
end

sumnumb = ones(N^2,1);

% row, column, and box uniqueness
```

```
rowunique = zeros(N^2,N^3);
colunique = zeros(N^2,N^3);
boxunique = zeros(N^2,N^3);

for ii = 1:N
    numuniquerow = zeros(N,N^3);
    numuniquecol = zeros(N,N^3);
    numuniquebox = zeros(N,N^3);
    for jj = 1:N
        for kk = 1:N^3
            if rowcon(jj,kk) == ii
                numuniquerow(jj,kk) = 1;
            end
            if colcon(jj,kk) == ii
                numuniquecol(jj,kk) = 1;
            end
            if boxcon(jj,kk) == ii
                numuniquebox(jj,kk) = 1;
            end
        end
    end
    rowunique((N*ii - N + 1):(N*ii),:) = numuniquerow;
    colunique((N*ii - N + 1):(N*ii),:) = numuniquecol;
    boxunique((N*ii - N + 1):(N*ii),:) = numuniquebox;
end

rowuniqueb = ones(N^2,1);
coluniqueb = ones(N^2,1);
boxuniqueb = ones(N^2,1);

% hard constraints based on clues

clueconextra = zeros(N^3);
cluebextra = zeros(N^3,1);

for x = 1:size(puzzle,1)
    for y = 1:size(puzzle,2)
        if puzzle(x,y) ~= 0
            colind = (N*x - N + y)*N - N + 1;
            clueconextra(colind:(colind + N - 1),
            colind:(colind + N - 1)) = eye(N);
            cluebint = zeros(N,1);
            for z = 1:N
                if puzzle(x,y) == z
                    cluebint(z) = 1;
```

```
                end
            end
            cluebextra(colind:(colind + N - 1)) = cluebint;
        end
    end
end

% remove the extra

cluecon = zeros(N*numclues,N^3);
clueb = zeros(N*numclues,1);
next = 0;
for a = 1:N^3
    yes = 0;
    for b = 1:N^3
        if clueconextra(a,b) == 1
            yes = 1;
        end
    end
    if yes == 1
        next = next + 1;
        cluecon(next,:) = clueconextra(a,:);
        clueb(next) = cluebextra(a);
    end
end

% build A and b for equality constraints
A = [rowcon;
     colcon;
     boxcon;
     sumnumcon;
     cluecon;
     rowunique;
     colunique;
     boxunique];
b = [rowb;
     colb;
     boxb;
     sumnumb;
     clueb;
     rowuniqueb;
     coluniqueb;
     boxuniqueb];

% use cvx and mosek
cvx_begin
```

```
    cvx_solver mosek
    variables xnums(N^3)
    minimize(norm(xnums,1))
    subject to
        A*xnums == b;
cvx_end

% guesser

guesses = zeros(N^2,1);
for c = (1:N^2)*N - N + 1
    possibles = xnums(c:(c + N - 1));
    guessval = 0;
    for d = 1:N
        if possibles(d) > guessval
            guessval = possibles(d);
            guessnum = d;
        end
    end
    guesses((c - 1 + N)/N) = guessnum;
end

resultmat = zeros(N);
guessnext = 0;
for f = 1:N
    for g = 1:N
        guessnext = guessnext + 1;
        resultmat(f,g) = guesses(guessnext);
    end
end

end
```

Finally, this is the function that will print any 4 by 4, or 9 by 9 puzzle, or presumably a 16 by 16 puzzle:

```
function [] = printpuzzle(resultmat)

% all of this is a convoluted way of making a nice grid
N = size(resultmat,1);

fprintf('|')
for pp = 1:(N + sqrt(N) - 1)
    if pp < N + sqrt(N) - 1
        fprintf('--')
```

```
    else
        fprintf('-')
    end
end
fprintf('|')
fprintf('\n')
for ll = 1:sqrt(N)
    for mm = 1:sqrt(N)
        for nn = 1:sqrt(N)
            if nn == 1
                fprintf('|')
            end
            for oo = 1:sqrt(N)
                if nn ~= sqrt(N) || oo ~= sqrt(N)
                    fprintf('%d ',resultmat(sqrt(N)*ll - sqrt(N) + mm,
                    sqrt(N)*nn - sqrt(N) + oo))
                else
                    fprintf('%d',resultmat(sqrt(N)*ll - sqrt(N) + mm,
                    sqrt(N)*nn - sqrt(N) + oo))
                end
            end
            if nn < sqrt(N)
                fprintf('| ')
            else
                fprintf('|\n')
            end
        end
    end
    if ll < sqrt(N)
        fprintf('|')
    end
    for rr = 1:sqrt(N)
        if rr < sqrt(N) && ll < sqrt(N)
            for ss = 1:(sqrt(N) + 1)
                fprintf('--')
            end
        elseif ll < sqrt(N)
            for tt = 1:(sqrt(N) - 1)
                fprintf('--')
            end
        end
        if rr == sqrt(N) && ll < sqrt(N)
            fprintf('-|\n')
        end
    end
end
```

```
fprintf('|')
for qq = 1:(N + sqrt(N) - 1)
    if qq < N + sqrt(N) - 1
        fprintf('--')
    else
        fprintf('-')
    end
end
fprintf('|')
fprintf('\n')

end
```