Minimum-Edit Broken Sudoku Optimization with ILP

A Gurobi-Based ILP Solution

Upamanyu Kashyap
Department of Electrical & Computer Engineering
University of California, Davis
Davis, California
ukashyap@ucdavis.edu

Kavin Rajasekaran
Department of Electrical & Computer Engineering
University of California, Davis
Davis, California
krajasekaran@ucdavis.edu

ABSTRACT

In this project, we present a Binary-Integer Linear Programming (BILP) model and formulation to repair broken Sudoku puzzles with the minimum number of edits to the given skeleton puzzle. We use an existing standard BILP formulation for solving Sudokus from Andrew Bartlett & Amy Langville. The objective function of the BILP is to minimize the number of edits to the given Sudoku puzzle. This formulation generalizes to any size Sudoku puzzle (nxn). For notation purposes, we refer to the "order" of a sudoku as $N = \sqrt{n},\ e.g.\ order\ 3\ (\text{"}normal\ Sudoku\text{"}) = 9x9$.

A "Broken" Sudoku describes a puzzle with some number of givens that is *unsolvable*. This happens when there are two or more equal numbers in the same row, column, or submatrix. This does not necessarily need to be explicit. In certain cases, the set of givens implicitly require rule violations as the puzzle is solved.

INTRODUCTION

Initial Formulation

The formulation from the Bartlett & Langville paper is as follows:

$x_{ijk} = \{1,\ if\ element\ (i,j)\ in\ \ nxn\ Sudoku\ matrix\ contains\ k;$

$0,\ otherwise$

$min\ 0^T x,\ such\ that:$

$\sum_{i=1}^{n} x_{ijk} = 1,\ j = 1{:}n,\ k = 1{:}n\ (only\ one\ k\ in\ each\ column)$

$\sum_{j=1}^{n} x_{ijk} = 1,\ i = 1{:}n,\ k = 1{:}n\ (only\ one\ k\ in\ each\ row)$

$\sum_{j=mq-m+1}^{mq} \sum_{i=mp-m+1}^{mp} x_{ijk} = 1,\ k = 1{:}n,\ p = 1{:}m,\ q = 1{:}m$

$(only\ one\ instance\ of\ k\ in\ each\ submatrix)$

$\sum_{k=1}^{n} x_{ijk} = 1,\ i = 1{:}n,\ j = 1{:}n\ (every\ position\ in\ matrix\ filled)$

$x_{ijk} = 1\ \forall (i,j,k) \in G\ (given\ elements\ G\ are\ set\ to\ \text{"}on\text{"})$

$x_{ijk} = \{0,1\}\ (each\ x\ is\ a\ binary\ value)$

This framework provides us a robust way to solve any sudoku puzzle given a certain number of known values in the matrix.

Minimizing Edits (Proposed Method)

We now encode the edit minimization into BILP as follows:

Let's call the original grid A_{ij}, which could be "broken," or unsolvable.

Adding a "cost" variable $c_{ijk}\ (if\ cell\ (i,j)\ is\ set\ to\ k)$

If $A_{ij}\ had\ an\ original\ value\ a \neq 0,$

- $c_{ijk} = 0\ iff\ k = a\ (no\ change\ to\ original\ cell)$
- $c_{ijk} = 1\ if\ k \neq a\ (the\ value\ in\ A_{ij}\ is\ changed)$

If $A_{ij}\ was\ empty\ (i.e.\ A_{ij} = 0),$

- $c_{ijk} = 0\ (Don't\ add\ a\ cost\ in\ the\ process\ of\ solving\ the\ puzzle)$

Therefore, our objective function is now:

$minimize\ \sum_{i,j,k}^{n} c_{ijk} * x_{ijk}$

Or, to put in plainly, minimize the number of instances where a cell with a given value $k \neq 0$ is edited.

METHOD

Data

Sudoku puzzles are inherently difficult to generate. The total number of possible filled grids of order 3 Sudokus is approximately $6.67 * 10^{21}$. If symmetries (rotation, relabeling of digits) are taken into account, this is roughly cut down to $5.47 * 10^{9}$. This complexity only increases for all higher orders N. We would like to challenge our optimization model for some higher values of N, but the ability to do so is constrained by computational time. For the purpose of this report, we will display results for up to order 6 Sudokus.

In order to synthesize broken Sudoku puzzles, we employ a few methods:

1. Making trivial edits to break known-solvable puzzles (i.e. adding a duplicate digit in the same row, column or sub-matrix).
    a. This creates a reference point for us to confirm that the algorithm works, by solving a puzzle with a known number of required edits.
2. Finding incorrectly solved puzzles through community forums (Reddit etc.)
    a. We found a few puzzles that were almost completely filled in, but done incorrectly. These puzzles were originally solvable, but left the user in a position where they could no longer progress. We took these puzzles and loaded them with all digits (given, and user-inputted), to see how the algorithm would handle these.
3. Using Generative AI (especially for larger order puzzles)
    a. Given the increased complexity of higher order Sudokus and their relative lack of popularity, creating puzzles is a difficult task and we employ AI to help us create these.

We are running three types of tests for each order of complexity. Up till now, we have loosely used the word "trivial." We will take the time to define it now: *For our purposes, trivial refers to a type of edit that is easily observable to a human user (i.e. replacing or deleting duplicate numbers).*

The complexity of solving Sudokus varies from puzzle to puzzle. This also holds true for fixing broken puzzles (not all broken puzzles are easily resolved). Thus, we have come up with three different tests that we will apply to each order of complexity:

1. Non-Trivial (Implicitly Broken) Problems
    a. Non-Trivial problems consist of a set of givens in a grid that aren't immediately obvious in how they are broken. There are no immediately observable basic Sudoku rules violations in the grid, yet attempting to solve them will not yield a valid solution.
2. Trivial Problems
    a. These are problems where, as described above in the definition of trivial, errors in the puzzle grid are immediately observable to humans because they violate a rule of Sudoku (row/column/sub-matrix duplicates).
3. "Chaotic" Problems
    a. We define chaotic problems as those where each line of the grid contains numbers 1-n. For a 9x9 puzzle, this would mean each row of the puzzle is simply the digits 1-9. These problems require a significant number of edits for them to become playable puzzles. We quantify the solve time and edits for each of these in increasing order as well.

Important: All our test cases for Trivial & Non-Trivial problems use puzzles that only require one edit. This is a choice we made in order to make the quantification of solver runtime more consistent, as a "random" number of edits would make it harder to compare solver runtimes for more complex puzzles.

Algorithm

Gurobi is used to implement the ILP minimization model. The sudokus are imported as an array of constants, as shown in Figure 1. Next, the code verifies that the grid is uniform and square.

```
[0,2,4, 7,9,5, 0,0,3],
[1,0,3, 0,0,0, 0,0,4],
[0,7,5, 4,0,6, 0,0,2],

[0,0,0, 0,0,0, 2,0,6],
[7,5,0, 0,0,0, 1,0,9],
[0,4,8, 0,6,9, 0,0,0],

[0,9,0, 0,0,0, 0,2,0],
[5,0,2, 0,0,0, 0,0,0],
[0,0,0, 0,3,0, 0,0,0]
```

Figure 1: An example of an order 3 non-trivial sudoku imported into the code.

The construction of the Gurobi model involves creating the Gurobi object, and then importing the constraints, variables and the objective.
It first creates a dictionary that has the grid itself, the size of the grid, and the Gurobi solver object.

```
def create_sudoku_model(grid, name="SudokuMinEdit"):
    n, m = validate_grid(grid)

    solver = gp.Model(name)

    # Store everything in a dictionary so we can pass it around easily
    original_grid_copy = []
    for row in grid:
        new_row = []
        for val in row:
            new_row.append(val)
        original_grid_copy.append(new_row)

    data = {
        "original_grid": original_grid_copy,
        "n": n,
        "m": m,
        "solver": solver,
    }

    setup_variables(data)
    setup_constraints(data)
    setup_objective(data)

    solver.update()
    return data
```

Figure 2: create_sudoku_model() function. Takes the original grid as an input and outputs a dictionary "data" which contains all constraints/variables.

The script then runs "setup_variables()," which takes the dictionary input from the Gurobi model and creates the variables x_{ijk} & c_{ij}.

Gurobi will then reference this through each iteration of the solver in order to populate the costs for each cell.

```
    for i in range(n):
        for j in range(n):
            original_val = original_grid[i][j]
            for k in range(1, n + 1):
                # Create a binary variable for cell (i,j) taking value k
                var_name = f"cell_{i}_{j}_{k}"
                cells[(i, j, k)] = solver.addVar(vtype=GRB.BINARY, name=var_name)

                # Determine cost
                if original_val == 0:
                    # It was blank, so filling it is "free" (no edit penalty)
                    costs[(i, j, k)] = 0.0
                elif k == original_val:
                    # Keeping the original value is free
                    costs[(i, j, k)] = 0.0
                else:
                    # Changing the value costs 1
                    costs[(i, j, k)] = 1.0

    data["cells"] = cells
    data["costs"] = costs
```

Figure 3: setup_variables() looping through each row and column of the original Sudoku grid in order to populate the cell values and cost values.

Next, the model imports the constraints for the solver using setup_constraints(). These constraints are the same BILP constraints defined by Bartlett & Langville.

```
    # Constraint 1: Each cell must have exactly one value
    for i in range(n):
        for j in range(n):
            # sum(cells[i,j,k] for all k) == 1
            solver.addConstr(
                gp.quicksum(cells[(i, j, k)] for k in range(1, n + 1)) == 1,
                name=f"one_value_{i}_{j}"
            )

    # Constraint 2: Each row must have each number exactly once
    for i in range(n):
        for k in range(1, n + 1):
            # sum(cells[i,j,k] for all j) == 1
            solver.addConstr(
                gp.quicksum(cells[(i, j, k)] for j in range(n)) == 1,
                name=f"row_{i}_val_{k}"
            )

    # Constraint 3: Each column must have each number exactly once
    for j in range(n):
        for k in range(1, n + 1):
            # sum(cells[i,j,k] for all i) == 1
            solver.addConstr(
                gp.quicksum(cells[(i, j, k)] for i in range(n)) == 1,
                name=f"col_{j}_val_{k}"
            )
```

Figure 4: First part of setup_constraints(), adding constraints for 1 entry per cell, unique instances of numbers for rows/columns.

```
    # Constraint 4: Each subgrid (block) must have each number exactly once
    for block_i in range(m):
        for block_j in range(m):
            # Determine range of rows and columns for this block
            row_start = block_i * m
            col_start = block_j * m

            for k in range(1, n + 1):
                # Sum over all cells in this block
                block_sum = gp.quicksum(
                    cells[(r, c, k)]
                    for r in range(row_start, row_start + m)
                    for c in range(col_start, col_start + m)
                )
                solver.addConstr(block_sum == 1, name=f"block_{block_i}_{block_j}_val_{k}")
```

Figure 5: Last part of setup_constraints(), adding requirements for each sub-matrix containing only one instance of each digit.

Finally, the model imports the solver objective. This objective is the one defined in our BILP formulation (minimize cost * cell value).

```python
def setup_objective(data):
    solver = data["solver"]
    cells = data["cells"]
    costs = data["costs"]

    # We want to minimize the total cost of our choices
    # Total Cost = sum( cost[i,j,k] * cell[i,j,k] )
    total_cost_expr = 0
    for key in cells:
        total_cost_expr += costs[key] * cells[key]

    solver.setObjective(total_cost_expr, GRB.MINIMIZE)
```

Figure 6: setup_objective() function that creates the main solver minimization objective

Now that all the solver parameters are set up, Gurobi begins optimizing the minimum edits to the input Sudoku.

RESULTS

Tests

Test 1: Non-Trivial Problem (9x9)

In figure 1, there are no explicit rules violations; No single number is repeated across a column, row, or sub-matrix. However, this is still impossible to solve. Just because the rules of Sudoku appear to be followed does not mean that the puzzle will be solvable, as when solving this, the user will create a contradictory region. For example, in rows 4-6 and columns 4-6, there is no set of numbers that will be able to solve the puzzle.

```
. 2 4 | 7 9 5 | . . 3
1 . . | . . . | . . 4
. 7 5 | 4 . 6 | . . 2
------------------------
. . . | . . . | 2 . 6
7 5 . | . . . | 1 . 9
. 4 8 | . 6 9 | . . .
------------------------
. 9 . | . . . | . 2 .
5 . 2 | . . . | . . .
. . . | . 3 . | . . .
```

Figure 7: Playable Sudoku (unchanged givens only; blanks elsewhere)

As we can observe, we only needed to change one number in this puzzle to make it valid: (2, 3): 3 is changed to a blank spot in order to make this puzzle solvable.

```
Repaired Sudoku (minimum edit solution)
8 2 4 | 7 9 5 | 6 1 3
1 6 9 | 2 8 3 | 5 7 4
3 7 5 | 4 1 6 | 8 9 2
------------------------
9 3 1 | 5 7 4 | 2 8 6
7 5 6 | 3 2 8 | 1 4 9
2 4 8 | 1 6 9 | 7 3 5
------------------------
6 9 3 | 8 5 1 | 4 2 7
5 1 2 | 9 4 7 | 3 6 8
4 8 7 | 6 3 2 | 9 5 1
```

Figure 8: One nonunique solution given the final minimal edit fixed sudoku puzzle

Test 2: Trivial 4x4 Broken Sudoku Puzzle

```
[0,6,0,0, 0,14,10,0, 13,5,15,0, 0,8,0,4],
[7,0,10,0, 16,5,6,0, 0,0,0,0, 15,0,11,1],
[0,0,0,3, 1,0,12,0, 6,0,0,0, 0,16,0,0],
[14,0,0,0, 3,11,0,0, 10,16,1,0, 0,0,0,0],

[0,0,7,0, 8,0,9,14, 16,0,12,0, 5,0,4,0],
[6,0,0,0, 0,0,7,0, 9,0,0,1, 0,0,8,13],
[0,5,0,0, 0,0,0,0, 15,0,13,0, 9,7,2,12],
[0,8,0,2, 0,13,1,12, 14,0,10,0, 0,15,0,0],

[0,0,3,0, 13,7,0,6, 2,8,16,4, 1,0,12,0],
[4,13,5,16, 12,1,0,10, 0,0,0,0, 0,0,3,0],
[12,1,0,16, 14,0,0,4, 0,13,0,0, 0,0,0,8],
[0,9,0,7, 0,3,0,0, 1,12,0,10, 4,13,0,0],

[0,0,0,0, 0,0,15,0, 8,1,3,16, 0,0,0,2],
[0,0,6,0, 0,16,14,5, 0,15,0,13, 8,0,0,0],
[1,16,0,8, 0,0,0,0, 0,10,14,2, 0,4,0,9],
[15,0,2,0, 0,8,13,1, 0,9,6,0, 0,0,10,0]
```

Figure 9: Example of 4x4 puzzle that's broken due to a repeat number in the same column

```
Playable Sudoku (unchanged givens only; blanks elsewhere)
 .  6  .  . |  . 14 10  . | 13  5 15  . |  .  8  .  4
 7  . 10  . | 16  5  6  . |  .  .  .  . | 15  . 11  1
 .  .  .  3 |  1  . 12  . |  6  .  .  . |  . 16  .  .
14  .  .  . |  3 11  .  . | 10 16  1  . |  .  .  .  .
-----------------------
 .  .  7  . |  8  .  9 14 | 16  . 12  . |  5  .  4  .
 6  .  .  . |  .  .  7  . |  9  .  .  1 |  .  .  8 13
 .  5  .  . |  .  .  .  . | 15  . 13  . |  9  7  2 12
 .  8  .  2 |  . 13  1 12 | 14  . 10  . |  . 15  .  .
-----------------------
 .  .  3  . | 13  7  .  6 |  2  8 16  4 |  1  . 12  .
 4 13  5 16 | 12  1  . 10 |  .  .  .  . |  .  .  3  .
12  1  .  . | 14  .  .  4 |  . 13  .  . |  .  .  .  8
 .  9  .  7 |  .  3  .  . |  1 12  . 10 |  4 13  .  .
-----------------------
 .  .  .  . |  .  . 15  . |  8  1  3 16 |  .  .  .  2
 .  .  6  . |  . 16 14  5 |  . 15  . 13 |  8  .  .  .
 1 16  .  8 |  .  .  .  . |  . 10 14  2 |  .  4  .  9
15  .  2  . |  .  8 13  1 |  .  9  6  . |  .  . 10  .
```

Figure 10: Playable 16x16 Sudoku (unchanged givens only; blanks elsewhere)

As we see, we only needed to change one number in this puzzle to make this a valid puzzle: (11, 4): 16 is changed to a blank spot in order to make this a valid solvable puzzle. This shows how our code is able to handle problems that are beyond the typical 9x9 Sudoku.

Test 3: "Chaotic" 5x5 Broken Sudoku Puzzle

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
```

*Figure 11: Broken Chaotic 25*25 Sudoku (unchanged givens only; blanks elsewhere)*

Here, we have a 25*25 puzzle that has the numbers 1-25 repeating through each column. This leaves us many conflicts to fix. Once we run the solver, we can observe that the number of minimum edits required to make this a valid puzzle is 600 edits.

```
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . | 16  .  .  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  9  . |  . 12 13  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  .  5 |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  . 23  .  .
------------------------------
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  . 20 |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  . 22  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  . 14  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  . 15 |  .  .  .  .  . |  .  .  .  .  .
------------------------------
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  7  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . | 11  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  2  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  3  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  . 19  . |  .  .  .  . 25
------------------------------
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  . 17  .  .  . |  .  .  . 24  .
 1  .  .  .  . |  .  .  8  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  4  . |  .  .  .  . 10 |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
------------------------------
 .  .  .  .  . |  6  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  . 18  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . | 21  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
 .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  . |  .  .  .  .  .
```

*Figure 12: Example of the chaotic 25*25 puzzle fixed in order to be solvable*

We observe that the optimizer has turned a significant portion of the puzzle to be blank in order to make the puzzle solvable. Even a "chaotic" example such as this is able to finish in a reasonable amount of time.

Plotting Runtime vs. Sudoku Order

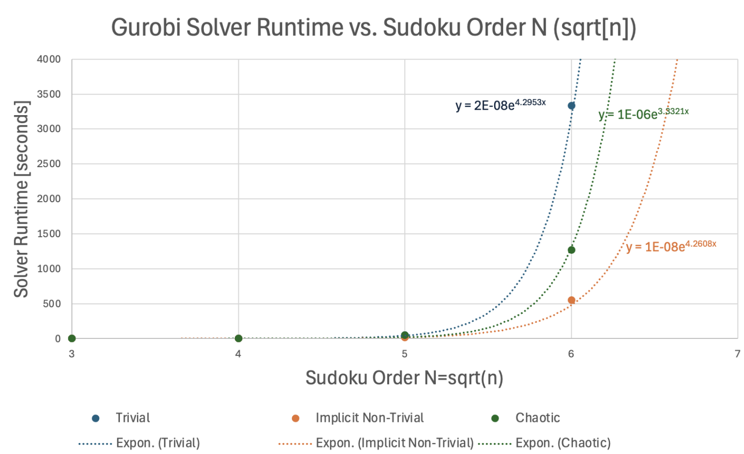

Figure 13: Plot of Gurobi solver runtime vs. the order of complexity of the puzzle

As we expected, the solver runtime increases exponentially with respect to increase in the order of the puzzle. This observation shows that the BILP minimization will take increasingly longer periods of time as the order of the Sudoku increases.

Table 1: Number of Edits Required to Solve Sudoku Puzzles			
Order	**Trivial**	**Non-Trivial**	**Chaotic**
3	1	1	72
4	1	1	240
5	1	1	600
6	1	1	1260

In Table 1, we show the number of edits required to solve each example puzzle we used in testing. As stated earlier, we decided to use puzzles where we know the number of edits required in order to quantify and compare the solver's performance in a more accurate manner.

Table 2: Gurobi Solver Runtime for Various Sudoku Puzzles			
Order	**Trivial [s]**	**Non-Trivial [s]**	**Chaotic [s]**
3	0.01	0.031	0.01
4	0.398	0.35	0.034
5	48.38	18.53	53.17
6	3,335.16	550.09	1,268.7

Table 2 shows the solver run times that are plotted on the graph above. It's apparent that there is an exponential trend on the increase in solve time as the order of the Sudoku increases. Order 7 and greater would likely require many hours of run time in order to produce a viable solution.

ACKNOWLEDGMENTS

Our work greatly relies on the previous work from Bartlett & Langville, who have provided the ILP basis formulation for solving Sudokus. We would also like to extend gratitude to Professor Soheil Ghiasi for his guidance and instruction throughout this course.

REFERENCES

[1] Bartlett, A. C., and Langville, A. N. 2006. *An Integer Programming Model for the Sudoku Problem*. College of Charleston, Charleston, SC, USA.

[2] SudokuWiki. 2025. Daily Sudoku. Retrieved December 7, 2025 from https://www.sudokuwiki.org/Daily_Sudoku

[3] Conroy, T. 2025. *Will We Ever Run Out of Sudoku Puzzles?* Britannica. Retrieved December 7, 2025 from https://www.britannica.com/story/will-we-ever-run-out-of-sudoku-puzzles