Prescriptive Analytics

Solving Sudoku with Linear Programming

Group 10
Ta-Wei Chien, Kai-An Lee, Eric Luong, Jeffrey Monticelli, Akshay Nayak

Summary
Sudoku puzzles offer an engaging challenge that combines logic, strategy, and pattern recognition. We used Gurobi to help us rapidly solve puzzles for standard sudoku and 6 other variants using linear programming.

Introduction
The history of sudoku spans hundreds of years; its origins can be traced back to the 13th century, when its predecessor “Magic Squares” was studied by mathematicians. Magic squares differ from sudoku because they contain double-digit numbers and require arithmetic rather than logic to solve, but the two share key characteristics: each row, column, and subsquare add up to the same number.

In the modern era, sudoku started to gain some limited popularity once it was published by freelance puzzle-maker Howard Garns under the name “Number Place,” but it was not until 1986 when Japanese puzzle company Nikoli tweaked the rules and gave it the name “Sudoku” (“single number” in Japanese), that it started to flourish internationally.

Sudoku has been recognized for its cognitive benefits, such as improving memory, concentration, and problem-solving skills. Studies have shown that senior adults who regularly perform puzzles have “younger,” more agile brains than those who do not.

Rules of Sudoku
A standard Sudoku puzzle consists of a 9x9 grid, divided into nine 3x3 subgrids. Some cells are pre-filled with numbers from 1 to 9, serving as clues for the puzzle. The objective is to fill all the empty cells in the grid such that every row, column, and 3x3 subgrid contains all of the digits from 1 to 9 exactly once.

Due to its popularity and the flexibility of its rules, sudoku has had numerous variants created that add or tweak the rules in interesting ways. We will use Gurobi to help us solve a standard sudoku grid, then make adjustments with additional rules as necessary for the variants.

Standard Sudoku
Using Gurobi, the goal is to find a solution(s) that satisfies the five constraints. For the sudoku puzzle, the objective function is arbitrary since it doesn't have a specific goal to maximize or minimize. The decision variables are binary, meaning they can only have a value of 0 or 1.

There are several constraints that need to be satisfied for Standard Sudoku:

The *Cell Constraint* ensures that each cell in the Sudoku grid contains exactly one number. It is expressed as a summation of binary variables associated with each cell and all possible numbers (from 1 to 9), where the sum must equal 1.

The *Row Constraint* guarantees that each number appears only once in each row. For every row and number, the summation of binary variables related to each cell in that row must equal 1.

The *Column Constraint* ensures that each number appears only once in each column. It involves summing up binary variables associated with each cell in a specific column and a particular number, with the sum required to be 1.

The *Subgrid Constraint* guarantees that each number appears only once in each 3x3 subgrid of the Sudoku grid. By dividing the grid into nine subgrids, the constraint is expressed as a summation of binary variables associated with each cell in a particular subgrid and number, with the sum required to be 1.

The *Pre-filled Constraint* accounts for any initial numbers already filled in the Sudoku puzzle. It sets the corresponding binary variable to 1 to indicate that the value is fixed and cannot be changed.

By formulating the Sudoku puzzle as an optimization problem with these constraints, Gurobi efficiently finds a solution(s) that satisfies all the rules of Sudoku.

Additions to the Standard Rules

To make the sudoku puzzles more interesting, we added some additional rules beyond the standard sudoku set. These rules necessitated additional Gurobi constraints.

Additional Constraints for Diagonal Sudoku:

The *Main Diagonal Constraint* states that for each value (k) from 1 to 9, the sum of the variables (X[i, i, k]) for i ranging from 0 to 8 must be equal to 1. This means that each number from 1 to 9 appears only once along the main diagonal of the Sudoku grid. In other words, no two cells on the main diagonal can have the same value.

The *Secondary Diagonal Constraint*, on the other hand, enforces a similar rule for the secondary diagonal. For each value (k) from 1 to 9, the sum of the variables (X[i, 8-i, k]) for i ranging from 0 to 8 must be equal to 1. This constraint ensures that each number from 1 to 9 appears only once along the secondary diagonal of the Sudoku grid, preventing any duplication of values.

Additional Constraints for Symmetrical Sudoku:

The *Transpose Constraint* enforces a relationship between pairs of cells in the puzzle grid. It states that if a specific number is placed in a particular column, row, and position, then the same number must also be present in the corresponding row, column, and position. This

constraint essentially reflects a mirror-like symmetry, where the entries of the sudoku grid maintain a symmetrical relationship across the main diagonal.

The *Symmetry Constraint* is an additional rule in Sudoku that establishes a symmetrical relationship between pairs of cells in the grid. It stipulates that if a certain number is placed in a given column, row, and position, then the same number must appear in the symmetrically opposite row, column, and position. This constraint introduces a form of bilateral symmetry.

Additional Constraints for Non-consecutive Sudoku:

The *No Consecutive in Row* constraint prevents consecutive numbers from appearing in the same row. For each combination of indices (i, j, k), where i represents the column, j represents the row, and k represents the number, the constraint ensures that if a number (k) is assigned to cell (i, j), then the following cell (i + 1, j) cannot have the number (k+1). This constraint guarantees that consecutive numbers are not placed horizontally in the same row.

The *No Consecutive in Column* constraint ensures that consecutive numbers do not appear in the same column. For each combination of indices (i, j, k), the constraint states that if a number (k) is assigned to cell (i, j), then the cell (i, j + 1) cannot have the number (k+1). This constraint prevents consecutive numbers from being vertically placed in the same column.

The *No Consecutive in Row Backward* constraint considers the backward direction of the rows. It guarantees that consecutive numbers are not placed in the same row when moving backward (from right to left). For each combination of indices (i, j, k), where i represents the column, j represents the row, and k represents the number, the constraint ensures that if a number (k) is assigned to cell (i, j), then the preceding cell (i - 1, j) cannot have the number (k+1). This constraint prevents consecutive numbers from being horizontally placed in the same row when moving backward.

The No Consecutive in Column Backward constraint considers the backward direction of the columns. It ensures that consecutive numbers are not placed in the same column when moving backward (from bottom to top). For each combination of indices (i, j, k), the constraint states that if a number (k) is assigned to cell (i, j), then the preceding cell (i, j - 1) cannot have the number (k+1). This constraint prevents consecutive numbers from being vertically placed in the same column when moving backward

Sudoku Variants

(1) 3D Sudoku:

Utilizing Gurobi, the aim is to discover solutions that comply with the eight constraints of 3D Sudoku. As with standard Sudoku, the objective function in the case of 3D Sudoku is arbitrary since it does not seek to maximize or minimize a specific quantity. The decision variables are binary, implying they can only take the values of 0 or 1.

Here are the constraints that need to be satisfied for 3D Sudoku:

The Grid Cell Constraint ascertains that each cell in the 3D Sudoku grid contains exactly one number. This constraint is manifested as a summation of binary variables related to each cell and all potential numbers (from 1 to the grid size), where the sum must equal 1.

The Row Constraint guarantees that each number appears only once in each row. For every row, depth layer, and number, the summation of binary variables connected to each cell in that row must be 1.

The Column Constraint ensures that each number appears only once in each column. This involves summing up binary variables related to each cell in a specific column, a particular depth layer, and a specific number, with the sum required to be 1.

The Depth Constraint ensures that each number appears only once in each depth layer. This involves summing up binary variables associated with each cell in a specific row, a particular column, and a specific number within each depth layer, with the sum required to be 1.

The 3D Diagonal Constraints ensure that each number appears only once in each of the four 3D diagonals running from the top-left-front to bottom-right-back, top-right-front to bottom-left-back, top-left-back to bottom-right-front, and top-right-back to bottom-left-front. This involves summing up binary variables related to each cell along a specific 3D diagonal and a specific number, with the sum required to be 1.

The Pre-filled Constraint caters for any initial numbers already filled in the 3D Sudoku puzzle. It assigns the corresponding binary variable to 1 to denote that the value is fixed and cannot be modified.

By formulating the 3D Sudoku puzzle as an optimization problem with these constraints, Gurobi efficiently uncovers a solution(s) that aligns with all the rules of 3D Sudoku.

(2) Killer Sudoku

Introduction to Killer Sudoku:

Killer Sudoku, a variant of the classic Sudoku puzzle, provides an additional challenge by including cages within the grid. Each cage, depicted by dotted lines, contains a set number that indicates the sum of all numbers within that cage. Like traditional Sudoku, each row, column, and region must contain the numbers 1 through 9 exactly once, and no number in a cage can be repeated. The combination of these constraints makes Killer Sudoku a challenging and engaging puzzle that requires both logic and arithmetic skills to solve.

Problem Statement:

The problem is to develop an efficient approach to solve Killer Sudoku puzzles, integrating logical deduction and arithmetic reasoning.

The main objective is to devise a method that can accurately fill a 9x9 grid, ensuring that each row, column, and region contains a unique set of numbers. Further, the sum of the numbers in each cage must equal the set number given in the top-left corner of the cage. This approach should use logical deduction techniques to identify the correct placement of numbers based on these constraints. It should consider the interdependencies between sections of the grid and the arithmetic constraints of the cages, enabling systematic exploration of possibilities while promoting efficient deduction strategies.

The proposed approach to address this challenge involves the use of constraint satisfaction algorithms. These algorithms enforce the uniqueness of numbers within rows, columns, regions, and cages. Also, they ensure that the sum of numbers in each cage matches the given sum. The goal is to strike a balance between systematic exploration and logical deduction, accommodating Killer Sudoku puzzles of varying difficulties.

The successful development of an effective Killer Sudoku solver will provide puzzle enthusiasts with a valuable tool for solving these numeric puzzles. It will enhance their puzzle-solving experience and foster their analytical thinking. Therefore, we focus on proposing and evaluating a solving approach that seamlessly combines logical deduction and arithmetic analysis, enabling efficient and accurate solving of Killer Sudoku puzzles.

Constraints:

Cell Constraint: Each cell in the Sudoku grid should contain precisely one number from 1 to 9. The corresponding code that ensures this constraint is:

```
“for cell_x in range(1,10):
   for cell_y in range(1,10):
      m.addConstr(sum(grid[cell_x-1][cell_y-1]) == 1.)”
```

This sum operation ensures that the binary variable for any given cell, represented by cell_x and cell_y, will be 1, indicating a single number assignment to each cell.

Row, Column, and Box Constraints: The Sudoku rules dictate that every number from 1 to 9 must appear once and only once in each row, column, and 3x3 box.
The following code enforces these constraints:

```
“for cells in sudoku:
   for i in range(1,10):
      m.addConstr(sum(grid[x-1][y-1][i-1] for (x,y) in cells) >= 1.)”
```

This operation makes sure that the sum of the binary variables for a given number i across all cells in the current group is at least 1, guaranteeing that the number i appears in every cell group.

Cage Constraint: Specific to Killer Sudoku, the sum of the numbers in each cage (a group of one or more cells) must be equivalent to a specified value.
The code enforcing this is:

```
“for cells, sum_ in killer_sudoku:
   m.addConstr(sum(i*grid[x-1][y-1][i-1] for i in range(1,10) for (x,y) in cells) == sum_)”
```

Here, the sum of the products of each number i and its respective binary variable across all cells within a cage should be equal to the expected sum. This constraint is applied for every cage, ensuring the solution adheres to the cage sums outlined in the puzzle.

Key Findings
Completing this project has reminded us of the importance of optimizing your code. We realized early on that we could make our coding process more efficient by using soft coding. Instead of using fixed sizes for our grids, we used the lengths of our rows and columns. This made our code more robust as it didn’t need to be adjusted to account for sudoku grids of different dimensions.