Nikoli

Japanese Logic Puzzles, Modelled and Solved

Vamshi Jandhyala

Vamshi Jandhyala
London

Contents

1 Kakuro 1

2 Sudoku 9

3 Skyscrapers 16

4 Kakurasu 23

5 Takuzu 29

6 KenKen 36

7 Flow Free 42

8 Squares Sudoku 48

9 Slitherlink 53

10 Hashiwokakero 60

11 Akari 67

12 Shikaku 74

13 Nurikabe 80

14 Masyu 87

15 Hitori 95

16 Fillomino 102

17 Heyawake 109

18 *Shakashaka* 116

19 *Marupeke* 125

20 *Walls* 131

21 *L-Panel* 137

22 *BlockNumber* 143

23 *Searchlights* 148

24 *Numbrix* 153

25 *3-in-a-Row* 159

Chapter 1

Kakuro

Kakuro is a crossword for arithmeticians. A grid of black and white cells awaits; the white cells are to receive digits between 1 and 9. Where the standard crossword demands a word, Kakuro demands a sum. Each horizontal or vertical run of white cells must add to the small number written in the adjacent clue cell, and no digit may repeat inside a run.

The puzzle was published by Maki Kaji's Nikoli in 1980 under the name *Kasan Kurosu*, a contraction of the Japanese *kasan* ("addition") and the English-borrowed *kurosu* ("cross"). Nikoli shortened this to *Kakuro* six years later, and the puzzle made its way into the English-speaking world in the early 2000s, often sharing newspaper column inches with Sudoku. The two puzzles share a parent (Dell Magazines' *Cross Sums*, 1966) and a logical substrate: both are constraint-satisfaction problems over the alphabet $\{1, 2, \dots, 9\}$ with *all-different* constraints on subsets of cells.

Small Kakuros yield to pencil and paper. Large ones, and the sub-class in which clue cells carry three- or four-cell sums, yield fastest to a constraint solver. This chapter does both: a hand-solvable instance to anchor the rules, then a general 0/1 integer-programming model that a dozen lines of Python turn into a solver for any Kakuro you can encode.

Rules and a small instance

A Kakuro puzzle is an $m \times n$ grid of cells, each of which is one of three kinds. A *block cell* is uniformly shaded and carries no clue. A *clue cell* is shaded and holds up to two sums: an *across sum* in the lower-left triangle referring to the horizontal run of white cells to its right, and a *down sum* in the upper-right triangle referring to the vertical run of white cells below. A *white cell* is to be filled with a digit from 1 to 9.

Two constraints govern each run.

1. The digits in the run add to the sum declared by its clue.
2. No digit is used twice inside the run.

A digit may, of course, appear in many runs of a single puzzle; the all-different constraint applies strictly inside each run, never across the whole grid.

Figure 1.1 gives a 4×4 instance with six white cells, reduced to the minimum that still exhibits all the clue geometries. The two column sums on the top row govern the two columns of white cells below them, and each row clue on the left governs the white cells to its right.

One constraint is instantly visible. The clue 7↓ at the top of column 3 governs a single white cell, so that cell must be 7. That cell sits in the row whose across clue is 15, forcing its neighbour to be 8; and so on. Any human solver following these forced implications to their conclusion will arrive at the digits later recovered by the solver in Figure 1.2.

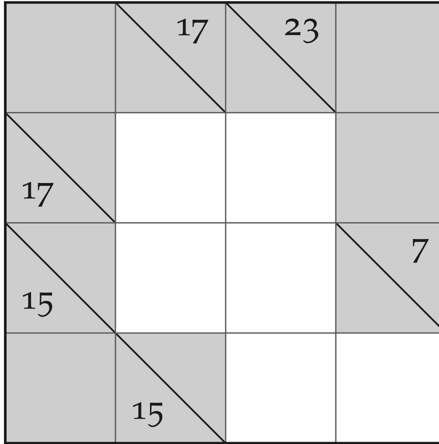

Figure 1.1: *A small Kakuro. Two column clues at the top of columns* 1 *and* 2*, three row clues on the left edge, and an isolated column clue at the top of column* 3 *govern six white cells.*

The programming model

Write $\mathcal{C}$ for the set of white cells and $\mathcal{R}$ for the set of runs. A run $r \in \mathcal{R}$ is a maximal horizontal or vertical sequence of white cells governed by a single clue; denote its target sum by s_r and its cell set by $\mathcal{C}_r \subseteq \mathcal{C}$. Each white cell belongs to exactly one across-run and exactly one down-run.

Constraint-satisfaction formulation

Associate an integer variable $x_c \in \{1, 2, \dots, 9\}$ with each white cell $c \in \mathcal{C}$. The puzzle becomes

$$\sum_{c \in \mathcal{C}_r} x_c = s_r \qquad \text{for all } r \in \mathcal{R},$$

$$\text{alldifferent}(\{x_c : c \in \mathcal{C}_r\}) \qquad \text{for all } r \in \mathcal{R}.$$

The sum constraints are linear; the all-different constraints are global, but a CP solver handles them natively through

hall-interval reasoning. A Kakuro puzzle is well-posed when this system has exactly one integer solution.

Integer-programming formulation

For a plain ILP formulation, replace each cell variable by nine indicators $y_{c,d} \in \{0,1\}$ for $d \in \{1,\dots,9\}$, where $y_{c,d} = 1$ means cell c holds digit d. The model becomes

$$\sum_{d=1}^{9} y_{c,d} = 1 \qquad \text{(one digit per cell)}$$

$$\sum_{c \in C_r} \sum_{d=1}^{9} d \cdot y_{c,d} = s_r \qquad \text{(run sum)}$$

$$\sum_{c \in C_r} y_{c,d} \le 1 \qquad \forall d \qquad \text{(distinct within run).}$$

Either formulation works; the CP version is shorter to write and usually faster to solve. Google's OR-Tools CP-SAT backend, used below, exposes the integer model directly.

A solver in thirty lines

The following Python reads a Kakuro specified as a list of clues and a grid of cell coordinates, builds the CP-SAT model, and returns the digit assignment.

```
from ortools.sat.python import cp_model

def solve_kakuro(cells, runs):
    """Solve a Kakuro and return {cell: digit}.

    cells : white-cell coordinates (row, col).
    runs  : (sum, [cell, ...]) pairs, one per run.
    """
    model = cp_model.CpModel()
    x = {c: model.NewIntVar(1, 9, f"x{c}") for c in cells}
    for s, run_cells in runs:
        model.Add(sum(x[c] for c in run_cells) == s)
        model.AddAllDifferent([x[c] for c in run_cells])
    solver = cp_model.CpSolver()
    status = solver.Solve(model)
    if status not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        raise RuntimeError("no solution")
    return {c: solver.Value(x[c]) for c in cells}
```

The solver is content-free: every Kakuro-specific fact sits in the `cells` and `runs` arguments. For the instance in Figure 1.1, those arguments are

```
cells = [(1, 1), (1, 2), (2, 1), (2, 2), (3, 2), (3, 3)]
runs = [
    (17, [(1, 1), (2, 1)]),              # col 1  17 down
    (23, [(1, 2), (2, 2), (3, 2)]),      # col 2  23 down
    (7,  [(3, 3)]),                      # col 3   7 down
    (17, [(1, 1), (1, 2)]),              # row 1  17 across
    (15, [(2, 1), (2, 2)]),              # row 2  15 across
    (15, [(3, 2), (3, 3)]),              # row 3  15 across
]
print(solve_kakuro(cells, runs))
```

CP-SAT returns in a handful of milliseconds the assignment

$$(1,1)=8,\ (1,2)=9,\ (2,1)=9,\ (2,2)=6,\ (3,2)=8,\ (3,3)=7,$$

which Figure 1.2 renders back on the grid.

Figure 1.2: *The solution. Copper digits are the values recovered by CP-SAT; the only given constraint value is the clue 7↓, whose one-cell run forces the 7 in the lower-right corner before the solver does any real work.*

A larger instance

The toy puzzle is solved as much by inspection as by the solver. To exercise CP-SAT properly, Figure 1.3 gives a 10 × 10 Kakuro with 63 white cells and 42 runs, of typical published difficulty: many runs of length three or four, overlapping clue geometries, no obvious forced moves at the start. The same solver code, called on this instance, returns the unique assignment in Figure 1.4 after roughly 17 milliseconds of search. Enumeration confirms uniqueness: the solver, asked for all feasible assignments, finds exactly one.

■	38\	29\	■	10\	14\	■	21\	16\	14\
\14			\6			\16			
\16			5\13			13\23			
\26					12\16			33\	31\
\10				21\15					
\6			14\22				9\15		
\16						12\7			
■	17\	5\9			8\30				
\11				\3			\17		
\19				\9			\3		

Figure 1.3: *A 10 × 10 Kakuro of typical published difficulty. Sixty-three white cells and forty-two runs.*

What carries the search at this size is the all-different constraint's domain reasoning. The clue 38 ↓ over a five-cell run, for instance, instantly forces the digit set to $\{1, 4, 7, 8, 9\}$

	↓38	↓29		↓10	↓14		↓21	↓16	↓14
14→	8	6	6→	1	5	16→	1	7	8
16→	7	9	13→ ↓5	4	9	23→ ↓13	8	9	6
26→	9	8	4	5	16→ ↓12	7	9	↓33	↓31
10→	6	3	1	15→ ↓21	2	1	3	5	4
6→	5	1	22→ ↓14	8	9	5	15→ ↓9	8	7
16→	3	2	4	6	1	7→ ↓12	2	4	1
	↓17	9→ ↓5	2	7	30→ ↓8	8	7	6	9
11→	8	2	1	3→	2	1	17→	9	8
19→	9	3	7	9→	6	3	3→	1	2

Figure 1.4: *The unique solution. Recovered by CP-SAT in about 17 ms on a laptop, with the all-different constraints doing the heavy lifting.*

or $\{2,4,6,8,9\}$ or one of a handful of other partitions of 38 into five distinct digits between 1 and 9; CP-SAT enumerates these at the front of the search and reuses them throughout. The two-cell, low-sum clues like $5 \rightarrow$ admit only $\{1,4\}$ or $\{2,3\}$, and propagate cell-by-cell.

Sources. Kakuro appeared in Nikoli's magazine *Puzzle Tsushin* in 1980 under the name *Kasan Kurosu*; Nikoli shortened it to *Kakuro* in 1986. The American ancestor is Dell Magazines' *Cross Sums*, which the puzzle designer Jacob Funk published in 1966 in *Dell Pencil Puzzles & Word Games*. Google's OR-Tools CP-SAT solver, released open-source in 2019, is used throughout this book; its constraint model is documented at

`developers.google.com/optimization`. The clue geometry in Figure 1.1 is original to this chapter, but the underlying aesthetic — diagonal clue cells, grey block fills, running sums on the outer margin — is Nikoli's.

Chapter 2

Sudoku

Sudoku is the Latin square with pedigree. Nine rows, nine columns, nine 3 × 3 boxes, and the same nine digits placed everywhere exactly once. The puzzle's commercial history is briefer than its logical lineage: Leonhard Euler studied Latin squares in the 1780s; Howard Garns, an American retired architect, published a reduced-size version called *Number Place* in Dell Magazines in 1979; and the Japanese publisher Nikoli picked it up in 1984, renaming it *Sudoku* (a contraction of *suuji wa dokushin ni kagiru*, "the digits must remain single") and imposing the rotational symmetry of clues that defines its modern visual rhythm.

Sudoku is, for our purposes, the canonical all-different puzzle. It has no sum clues, no length-variable runs, no visibility rules. The entire specification sits in three disjoint families of all-different constraints: rows, columns, and boxes. A CP-SAT model is a five-line affair. What makes individual puzzles hard is the amount of propagation a human needs to perform before a cell can be written in; what makes the solver fast is the aggregation of that propagation into a few standard constraint operators.

This chapter treats Sudoku twice over: once with a beginner's instance, then again with Arto Inkala's *AI Escargot* from 2006, at the time the hardest puzzle known. The solver code does not change between the two.

Rules and a small instance

A Sudoku grid is a 9×9 lattice of cells. A subset of cells carry *given* digits between 1 and 9; the solver's task is to fill every other cell with a digit between 1 and 9 subject to three constraints.

1. Each row contains each digit from 1 to 9 exactly once.
2. Each column contains each digit from 1 to 9 exactly once.
3. Each of the nine 3×3 boxes contains each digit from 1 to 9 exactly once.

A well-posed Sudoku has exactly one completion consistent with the givens. McGuire, Tugemann and Civario proved in 2012 that the minimum number of givens a uniquely solvable Sudoku can carry is seventeen. At seventeen the puzzle is unique but often delicate; most Nikoli-published puzzles sit between twenty and thirty.

Figure 2.1 is the standard beginner's example circulated in introductions to the puzzle, with thirty givens.

The programming model

Let $x_{r,c}$ denote the digit in row r and column c, with $r, c \in \{0, 1, \ldots, 8\}$. The givens $g_{r,c}$ fix $x_{r,c} = g_{r,c}$ for those cells; all other $x_{r,c}$ are variables with domain $\{1, 2, \ldots, 9\}$. The three families of all-different constraints are

$$
\begin{aligned}
\text{alldifferent}(\{x_{r,c} : c = 0, \ldots, 8\}) &\quad \text{for each row } r, \\
\text{alldifferent}(\{x_{r,c} : r = 0, \ldots, 8\}) &\quad \text{for each column } c, \\
\text{alldifferent}(\{x_{3b_r+i,\, 3b_c+j} : i, j = 0, 1, 2\}) &\quad \text{for each box } (b_r, b_c).
\end{aligned}
$$

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

Figure 2.1: *A beginner's Sudoku. Thirty given digits; the remaining fifty-one cells admit a unique completion.*

Twenty-seven constraints in total, each over nine variables. A 0/1 ILP formulation replaces $x_{r,c}$ with nine indicators $y_{r,c,d}$ and expresses each all-different as a pair of $\leq$ inequalities. The CP model is shorter to write and faster to solve.