Nikoli

Japanese Logic Puzzles, Modelled and Solved

Vamshi Jandhyala

Vamshi Jandhyala
London

Contents

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 $\{1, 2, \ldots, 9\}$ 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.

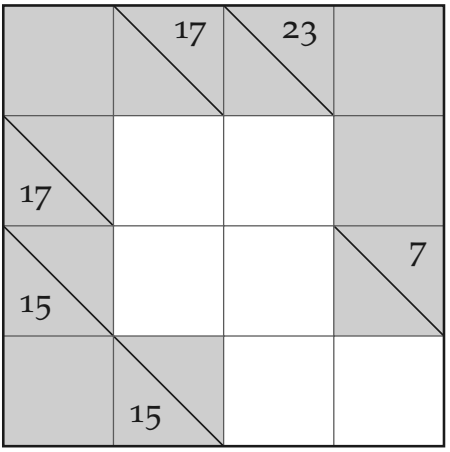

Figure 1.1: *A small Kakuro. Two column clues at the top of columns* 1 *and* 2*, three row clues on the left edge, and an isolated column clue at the top of column* 3 *govern six white cells.*

The programming model

Write 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 $C_r \subseteq C$. 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 $c \in C$. The puzzle becomes

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

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

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.

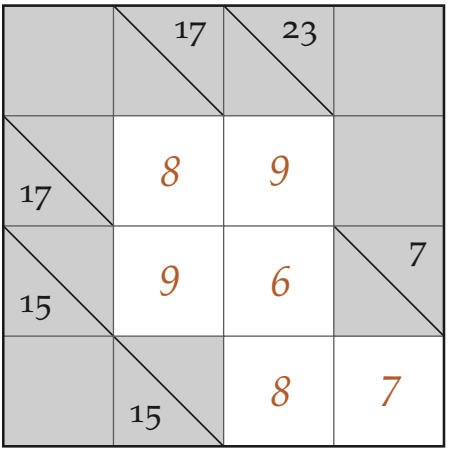

Figure 1.2: *The solution. Copper digits are the values recovered by CP-SAT; the only given constraint value is the clue 7↓, whose one-cell run forces the 7 in the lower-right corner before the solver does any real work.*

✠

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.

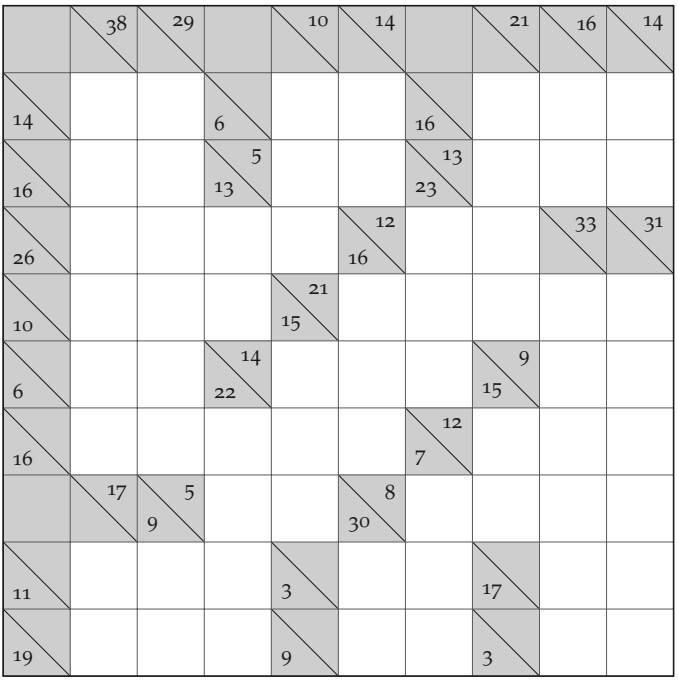

Figure 1.3: *A* 10×10 *Kakuro of typical published difficulty. Sixty-three white cells and forty-two runs.*

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	5 / 13	4	9	13 / 23	8	9	6
26	9	8	4	5	12 / 16	7	9	33	31
10	6	3	1	21 / 15	2	1	3	5	4
6	5	1	14 / 22	8	9	5	9 / 15	8	7
16	3	2	4	6	1	12 / 7	2	4	1
	17	5 / 9	2	7	8 / 30	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 $r, c \in \{0, 1, \dots, 8\}$. The givens $g_{r,c}$ fix $x_{r,c} = g_{r,c}$ for those cells; all other $x_{r,c}$ are variables with domain $\{1, 2, \dots, 9\}$. The three families of all-different constraints are

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

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.

A solver in twenty lines

```
from ortools.sat.python import cp_model

def solve_sudoku(givens):
    """Return a 9x9 completion of the puzzle."""
    m = cp_model.CpModel()
    x = [[m.NewIntVar(1, 9, f"x{r}{c}")
          for c in range(9)] for r in range(9)]
    for r in range(9):
        for c in range(9):
            if givens[r][c]:
                m.Add(x[r][c] == givens[r][c])
    for r in range(9):
        m.AddAllDifferent(x[r])
    for c in range(9):
        m.AddAllDifferent([x[r][c] for r in range(9)])
    for br in range(3):
        for bc in range(3):
            block = [x[br*3+i][bc*3+j]
                     for i in range(3) for j in range(3)]
            m.AddAllDifferent(block)
    s = cp_model.CpSolver()
    ok = s.Solve(m)
    if ok not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        raise RuntimeError("no solution")
    return [[s.Value(x[r][c])
             for c in range(9)] for r in range(9)]
```

The function recovers Figure 2.2 in about 15 milliseconds. No tactical code is written: no naked pairs, no hidden triples, no X-wings. CP-SAT's all-different propagator, implemented as a matching in a bipartite digit-cell graph, does every one of these inferences by construction.

A hard instance

Arto Inkala's *AI Escargot*, published in 2006, was for several years the hardest Sudoku known in the sense that it required the longest chain of logical inferences a human could be forced to follow to solve it. Computationally it is no harder than any

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

Figure 2.2: *The unique completion. Copper digits are the values returned by CP-SAT; the upright inkdeep digits are the originals.*

other unique puzzle: the same solver, the same code path, the same handful of milliseconds.

The gap between human and solver here is instructive. A human solves AI Escargot by maintaining a grid of candidate digits for every empty cell, eliminating candidates via row, column, and box exclusions, and tracing long conditional chains to contradiction ("if this cell is a seven, then that cell is a four, and then the box down-right has no place for a one"). CP-SAT does the same work algebraically. The all-different propagator, given the current domains, computes a maximum matching between cells and digits; any edge not in some maximum matching corresponds to a digit a cell cannot take, and is pruned. A linear-time update after each assignment suffices. In machine terms there is no distinction between Escargot and the beginner's grid of Figure 2.1.

✠

Sources. Howard Garns designed *Number Place* for Dell Pencil Puzzles & Word Games in 1979. Nikoli began publishing the puzzle in 1984 under the name *Sudoku*;

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

Figure 2.3: *AI Escargot. Twenty-three givens, arranged in a rotationally symmetric pattern; long forcing chains but still one completion.*

the convention of rotationally symmetric clues and a minimum-clue ambition is their contribution. The uniqueness minimum of seventeen was settled by McGuire, Tugemann and Civario in *Experimental Mathematics* 23 (2014), 190–217, after a brute-force enumeration of candidate seventeen-clue grids. *AI Escargot* was published by Arto Inkala in 2006; the puzzle appears in Inkala's book *AI Sudoku* and on his website. CP-SAT's all-different propagator follows Régin's filtering algorithm (AAAI 1994).

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

Figure 2.4: *The unique completion of AI Escargot, solved in roughly 29 ms.*

Chapter 3

Skyscrapers

Skyscrapers turns a Latin square into an aerial photograph. Each cell of an $n \times n$ grid holds the height of a building, a digit between 1 and n; each row and each column is a permutation of those heights. The puzzle's twist, new to this book, is visibility. Numbers written outside the grid along each edge record how many of the buildings in the corresponding row or column would be visible to an observer standing at that edge, with a taller building hiding every shorter one behind it.

The puzzle is sometimes attributed to Nikoli and sometimes to the broader tradition of Japanese visibility puzzles; an early form circulated at the World Puzzle Championship in the 1990s. The cleanest way to treat it is as a constraint-satisfaction problem built on three layers: a Latin square at the base, and then, for each edge clue, a counting constraint expressed through a running maximum over the corresponding row or column.

Rules and a small instance

A Skyscrapers puzzle is an $n \times n$ grid. Each cell holds a digit from 1 to n. Along any of the four sides of the grid, a subset of row- or column-positions carry a *visibility clue*: a positive integer.

1. Each row and each column contains every digit from 1 to n exactly once.

2. For any visibility clue k placed at the edge of a row (or column), exactly k cells of that row (column), read from the clue edge inward, are *visible*: not blocked by a strictly taller cell closer to the edge.

Figure 3.1 gives a 6×6 Skyscrapers puzzle with ten edge clues. The two west-edge clues sit on rows 3 and 6; the two east-edge clues on rows 2 and 5; the north and south edges carry three and three clues respectively.

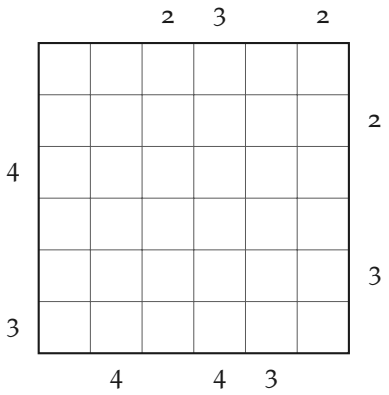

Figure 3.1: *A 6×6 Skyscrapers puzzle. Numbers sit outside the grid on whichever edge they belong to; a 4 on row 3's west edge demands four visible buildings looking east along that row.*

The programming model

Let $x_{r,c}$ denote the height in cell (r, c). The Latin-square layer is the usual pair of all-different families:

$$\text{alldifferent}(\{x_{r,c} : c = 1, \dots, n\}) \qquad \text{for each row } r,$$

$$\text{alldifferent}(\{x_{r,c} : r = 1, \dots, n\}) \qquad \text{for each column } c.$$

The visibility layer needs one new idea. For a given line of cells $v_1, v_2, \dots, v_n$ read from a clue edge inward, define *running max* $M_0 = 0$ and $M_i = \max(M_{i-1}, v_i)$. Cell i is visible from the edge exactly when $v_i > M_{i-1}$. The visibility count is

$$\text{vis}(v) \;=\; \sum_{i=1}^{n} \mathbf{1}[v_i > M_{i-1}],$$

and the clue k on that edge imposes $\text{vis}(v) = k$.

In CP-SAT, M_i enters the model as an auxiliary integer variable tied to the previous M and the current cell by a max constraint; $\mathbf{1}[v_i > M_{i-1}]$ is a boolean reified against the strict inequality. Each clue therefore adds n auxiliary integer variables, n booleans, and a single linear sum equation.

A solver in thirty-five lines

```
from ortools.sat.python import cp_model

def solve_skyscrapers(n, clues):
    """Solve an n x n Skyscrapers puzzle.

    clues : iterable of (side, idx, count) with
            side in {'N','E','S','W'}, idx 1-based.
    """
    m = cp_model.CpModel()
    x = [[m.NewIntVar(1, n, f"x{r}{c}")
          for c in range(n)] for r in range(n)]
    for r in range(n):
        m.AddAllDifferent(x[r])
    for c in range(n):
        m.AddAllDifferent([x[r][c] for r in range(n)])

    def add_vis(line, count):
        running = m.NewIntVar(0, n, "")
        m.Add(running == 0)
        vis = []
        for v in line:
            is_vis = m.NewBoolVar("")
            m.Add(v > running).OnlyEnforceIf(is_vis)
            m.Add(v <= running).OnlyEnforceIf(is_vis.Not())
            vis.append(is_vis)
            nm = m.NewIntVar(0, n, "")
            m.AddMaxEquality(nm, [running, v])
```

```
            running = nm
        m.Add(sum(vis) == count)

    for side, idx, cnt in clues:
        row = x[idx-1]
        col = [x[r][idx-1] for r in range(n)]
        if   side == 'W': add_vis(row, cnt)
        elif side == 'E': add_vis(row[::-1], cnt)
        elif side == 'N': add_vis(col, cnt)
        elif side == 'S': add_vis(col[::-1], cnt)

    s = cp_model.CpSolver()
    ok = s.Solve(m)
    if ok not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        raise RuntimeError("no solution")
    return [[s.Value(x[r][c])
             for c in range(n)] for r in range(n)]
```

The 6×6 instance of Figure 3.1 solves in about 11 milliseconds and admits a single completion; the solver returns Figure 3.2.

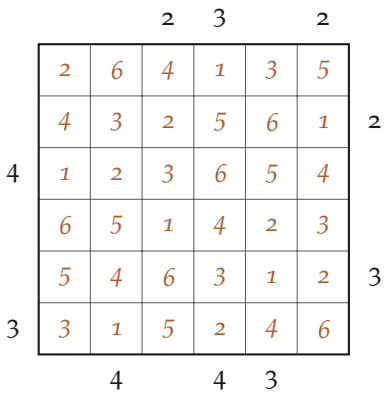

Figure 3.2: *The unique completion. Rows and columns are Latin; from the west edge of row* 3 *one sees* $1, 2, 3, 6$*, exactly four buildings, matching the clue.*

A hard instance

Figure 3.3 is Mikhail Khotiner's Puzzle 24 from the janko.at Wolkenkratzer archive, graded there as the single hardest puzzle in the collection. The grid is 8×8, with four edge clues missing, one interior given digit at $(3, 3) = 4$, and visibility counts that force long chains of deduction. CP-SAT returns the unique completion (Figure 3.4) in about 40 milliseconds.

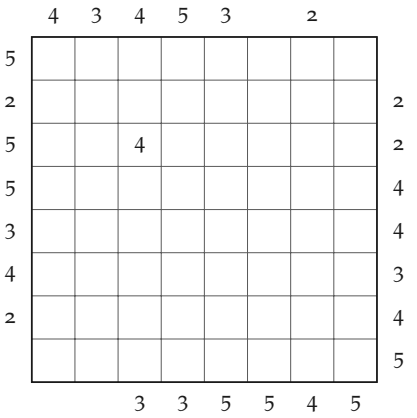

Figure 3.3: *Skyscrapers* 24 *from janko.at* (*Mikhail Khotiner,* Krossvordy and Golovolomki). *Grid size* 8×8*; one interior given,* $(3, 3) = 4$.

What the running-max trick buys

The visibility count is non-convex in the raw heights; a naive formulation would enumerate the $n!$ permutations of each row and column, filtering by the visibility count. For $n = 6$ that is 720 permutations per line, tractable but ugly. The running-max reification swaps the enumeration for a chain of

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

2	2	4	3	4	5	4	5

Figure 3.4: *The unique completion. Recovered in roughly* 40 *ms and agreeing with the published solution.*

n auxiliary variables per clue; the solver's propagation then stays local. Crucially, the reification is tight: if $v_i > M_{i-1}$ is forced true by domain reasoning, the booleans update without any search; if the strict inequality is forced false, likewise. Most real Skyscrapers puzzles have their visibility counts determined by the outer clues well before the Latin square is fully nailed down, and CP-SAT exploits this directly.

Two corner cases tighten the model without changing its surface. A clue of 1 on a given edge forces the n in the cell adjacent to that edge: nothing can be hidden behind a peak of height n, so exactly the one building is visible. A clue of n forces the line to be the strictly-increasing permutation $1, 2, \ldots, n$: every building must be taller than all its predecessors. Either fact, fed into the model as a direct equality, cuts search further. Neither is needed for correctness; both are present in the propagator by implication.

✠

Sources. Skyscrapers puzzles circulated at the World Puzzle Championship in the 1990s and were popularised by Japanese publishers and Web-based puzzle sites in the following decade. The running-max reification used in the model is

standard CP practice; see Simonis, *Sudoku as a constraint problem* (CP-AI-OR 2005) for a closely related treatment of the Latin square layer, and Mladenović et al., *Solving Skyscrapers puzzles with constraint programming*, Computers & Operations Research 45 (2014), for the full machinery. The CP-SAT solver's `AddMaxEquality` and reified-boolean mechanisms are documented at `developers.google.com/optimization`.

Chapter 4

Kakurasu

✠

Kakurasu is a shading puzzle dressed as a Kakuro. The grid is a rectangle of cells, each of which is to be declared either shaded or unshaded. Down the right-hand edge sits a column of integers, one per row; along the bottom sits a row of integers, one per column. The right-edge integer is the sum of the *column indices* of the shaded cells in its row; the bottom-edge integer is the sum of the *row indices* of the shaded cells in its column. Counting along each direction begins at 1 from the edge nearest the clue.

The puzzle looks, at first glance, like a Kakuro cousin. The arithmetic is the same shape: a sum over selected cells. What changes is the alphabet. In Kakuro each white cell takes a digit from 1 to 9; in Kakurasu each cell takes a value from $\{0, 1\}$, and the weights in the sum come not from the cells themselves but from their position along the run. That small shift, from digits-as-values to position-as-weight, moves the puzzle out of the constraint-programming register into something cleaner still: a pure binary integer program with only linear constraints and no all-different in sight.

This chapter models the puzzle in 0/1 variables, solves a 9×9 instance in roughly 20 milliseconds, and notes the single observation that makes the formulation water-tight: every row and every column's clue uniquely decodes the binary pattern of that row or column, in the same way that the positional notation for an integer uniquely decodes its digits.

✠

Rules and a small instance

A Kakurasu puzzle is an $n \times n$ grid of cells. Each cell (r, c), with $r, c \in \{1, \dots, n\}$, is either *shaded* or *unshaded*. Write $x_{r,c} \in \{0, 1\}$ for its state, with 1 for shaded. Two families of clues govern the shading.

1. For each row r, the sum $\sum_{c=1}^{n} c \cdot x_{r,c}$ equals a given integer R_r. That is: shaded cells in row r, weighted by their column indices, total R_r.
2. For each column c, the sum $\sum_{r=1}^{n} r \cdot x_{r,c}$ equals a given integer C_c. Shaded cells in column c, weighted by their row indices, total C_c.

Figure 4.1 gives a 9×9 Kakurasu with every row- and column-clue populated; each is a demand on the corresponding run. No pre-shaded cells: the solver fills the entire 81-bit array from the clues alone.

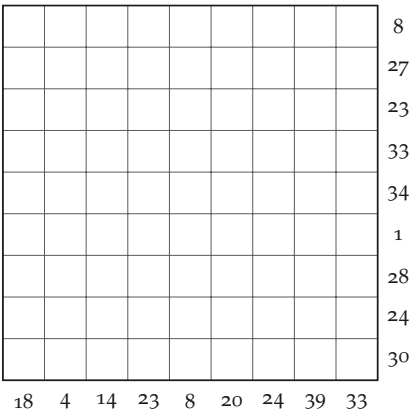

Figure 4.1: *A 9×9 Kakurasu. Right-edge integers are row-weighted sums over shaded columns; bottom-edge integers are column-weighted sums over shaded rows.*

✠

The programming model

Let $x_{r,c} \in \{0,1\}$ for $1 \le r, c \le n$. The puzzle is the system of linear equalities

$$\sum_{c=1}^{n} c \cdot x_{r,c} = R_r \qquad \text{for each row } r,$$

$$\sum_{r=1}^{n} r \cdot x_{r,c} = C_c \qquad \text{for each column } c.$$

This is a 0/1 integer linear program with n^2 variables and $2n$ equality constraints. No all-different, no reified booleans, no auxiliary variables: everything the puzzle says is a linear combination of indicator variables equal to a constant.

Why the clues decode uniquely

A single row, in isolation, looks like this: given $R \in \{0, 1, \dots, n(n+1)/2\}$, find $x_1, \dots, x_n \in \{0,1\}$ with $\sum_c c \cdot x_c = R$. The answer is not always unique in isolation: $R = 3$ is both $\{3\}$ and $\{1,2\}$. What makes the full puzzle uniquely solvable, when it is, is the interlocking of the row and column clues: the binary pattern of row r must simultaneously satisfy its own row equation and contribute consistently to all nine column equations. For the specific 9×9 in Figure 4.1, enumeration confirms one admissible matrix. The solver shows it in Figure 4.2.

A solver in a dozen lines

```
from ortools.sat.python import cp_model

def solve_kakurasu(n, row_clues, col_clues):
    """Return an n x n 0/1 array satisfying the clues."""
    m = cp_model.CpModel()
    x = [[m.NewBoolVar(f"x{r}{c}")
          for c in range(n)] for r in range(n)]
    for r in range(n):
        m.Add(sum((c+1) * x[r][c]
                  for c in range(n)) == row_clues[r])
```

```
    for c in range(n):
        m.Add(sum((r+1) * x[r][c]
                  for r in range(n)) == col_clues[c])
    s = cp_model.CpSolver()
    ok = s.Solve(m)
    if ok not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        raise RuntimeError("no solution")
    return [[s.Value(x[r][c])
             for c in range(n)] for r in range(n)]
```

For the 9×9 instance the solver returns in about 18 milliseconds. Enumeration of all feasible assignments returns one, so the puzzle is well-posed.

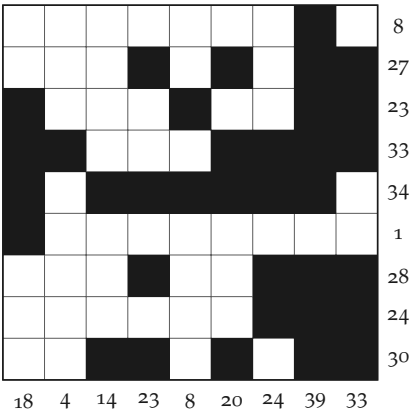

Figure 4.2: *The unique shading. Shaded cells are drawn in solid inkdeep; every row- and column-sum, under positional weighting, matches the outer clue.*

A larger instance

Figure 4.3 is Otto Janko's Puzzle 390 from the janko.at Kakurasu archive, at 11×11 and rated at the top tier of the site's difficulty scale. The row and column clues are large enough to force substantial interaction between constraints; CP-SAT returns the unique shading in Figure 4.4 in about

2 milliseconds, a reminder that 121 binary variables under 22 linear constraints is small work for a modern ILP backend, however challenging the puzzle is to solve by hand.

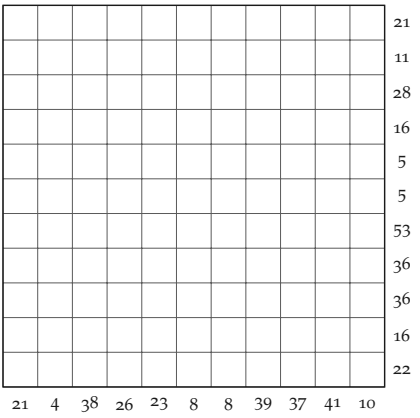

Figure 4.3: *Kakurasu* 390 *from janko.at, an* 11×11 *instance by Otto Janko.*

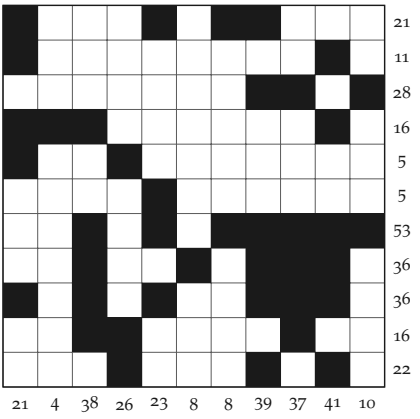

Figure 4.4: *The unique shading, recovered in about* 2 *ms and matching the published solution on janko.at.*

✠

The arithmetic geometry of the grid

A $n \times n$ Kakurasu sits naturally in the integer lattice $\mathbb{Z}^{n^2}$, each cell a coordinate. The shaded patterns form the integer points of the hypercube $\{0,1\}^{n^2}$; the clue set carves out an intersection of this cube with an affine subspace of codimension $2n$. A well-posed puzzle is one whose cube-affine intersection consists of a single point. The number of clue tuples that yield unique puzzles grows rapidly with n; for $n = 9$ the feasible volume under 18 clue values is already small enough that a well-tuned generator produces a near-unique puzzle the first time.

One further curiosity. The maximum row clue is $1 + 2 + \cdots + n = n(n+1)/2$, attained when the entire row is shaded. The minimum is 0, attained when none is. The same bounds apply to column clues. The middle clue values, around $n(n+1)/4$, support the greatest number of feasible row patterns and are the ones that, under random generation, most reliably lead to unique puzzles when paired in sufficient count.

✠

Sources. Kakurasu appears in the Nikoli family of sum-based puzzles as a counterpart to Kakuro; the positional weighting is a unique identifier. The 9×9 instance in Figure 4.1 is sourced from the *recreational_maths* collection and was originally solved with Z3; the CP-SAT port here is in the same variable space with equivalent constraints. For the broader family of positional-sum puzzles, see the survey material at the Nikoli English-language site.

Chapter 5

Takuzu

✠

Takuzu is the puzzle of balanced coin flips. The grid is a square of even side n, each cell shaded or unshaded (heads or tails, black or white, 0 or 1). A small number of cells carry given values; the solver fills the rest subject to two constraints. Every row and every column contains equal counts of the two states, and no row or column contains three consecutive cells of the same state.

The puzzle is known under several names: *Binairo* in European sources, *Tohu-Wa-Vohu* in the janko.at archive (after the Hebrew for 'formless and void'), *Takuzu* in Japan, *Binary Puzzle* in English-language newspapers. Nikoli publishes it under one of several Japanese names in the sum/shading family. The earliest version with the three-in-a-row prohibition seems to be due to Adolfo Zanellati circa 2009; earlier binary Latin-square puzzles without the consecutive constraint existed in recreational literature for decades.

Mathematically, Takuzu is a pure 0/1 integer program, like Kakurasu of the previous chapter, but with a richer constraint shape. The equal-counts constraint is one linear equality per row and per column. The no-three-consecutive constraint bans just two local patterns, 000 and 111; it translates cleanly into pairs of inequalities on sliding windows of three variables.

✠

Rules and a small instance

A Takuzu puzzle is an $n \times n$ grid with n even. Each cell (r, c) holds a value $x_{r,c} \in \{0, 1\}$. Two constraints govern the grid.

1. Every row and every column contains exactly $n/2$ zeros and $n/2$ ones.
2. No three consecutive cells in any row or column share the same value.

A well-posed Takuzu puzzle has a subset of cells pre-filled with 0 or 1; these are the *givens*. The solver's task is to extend the partial assignment to the unique complete grid that respects both rules.

Figure 5.1 shows a 6 × 6 Takuzu with eight given cells, four white and four black. The unique completion appears in Figure 5.2; givens in the solution are outlined in copper so as to distinguish them from cells the solver filled.

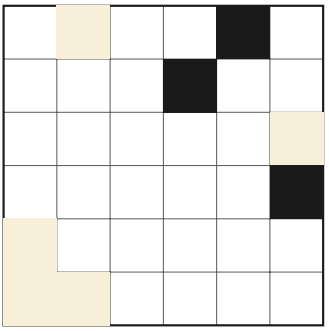

Figure 5.1: *A* 6 × 6 *Takuzu. Black cells are the given* 1*s; cream cells are the given* 0*s; blank cells are to be filled.*

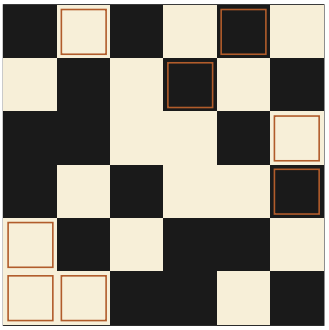

Figure 5.2: *The unique completion. Copper outlines mark the original givens; the rest is filled by the solver.*

✠

The programming model

Let $x_{r,c} \in \{0, 1\}$ for $0 \le r, c < n$. The equal-counts constraint is

$$\sum_{c=0}^{n-1} x_{r,c} = \frac{n}{2} \qquad \text{for each row } r,$$

$$\sum_{r=0}^{n-1} x_{r,c} = \frac{n}{2} \qquad \text{for each column } c.$$

The no-three-consecutive constraint is a pair of inequalities on every sliding window of three consecutive cells. For a window to avoid $(0, 0, 0)$ the sum must be at least 1; to avoid $(1, 1, 1)$ the sum must be at most 2:

$$1 \le x_{r,c} + x_{r,c+1} + x_{r,c+2} \le 2,$$
$$1 \le x_{r,c} + x_{r+1,c} + x_{r+2,c} \le 2.$$

On an $n \times n$ grid this gives $2n(n - 2)$ sliding-window inequalities plus $2n$ equal-counts equalities, all linear, all on binary variables. Unlike Kakurasu the constraints are local: each involves only two or three variables. The number of feasible assignments of a raw Takuzu grid (no givens) grows rapidly but slower than 2^{n^2}; a typical published puzzle

constrains it down to a single solution with only a modest number of givens.

A solver in a dozen lines

```
from ortools.sat.python import cp_model

def solve_takuzu(n, givens):
    """givens: (r, c, v) triples; v in {0, 1}."""
    m = cp_model.CpModel()
    x = [[m.NewBoolVar(f"x{r}{c}")
          for c in range(n)] for r in range(n)]
    for r, c, v in givens:
        m.Add(x[r][c] == v)
    for r in range(n):
        m.Add(sum(x[r]) == n // 2)
    for c in range(n):
        m.Add(sum(x[r][c] for r in range(n)) == n // 2)
    for r in range(n):
        for c in range(n - 2):
            s = x[r][c] + x[r][c+1] + x[r][c+2]
            m.Add(s >= 1); m.Add(s <= 2)
    for c in range(n):
        for r in range(n - 2):
            s = x[r][c] + x[r+1][c] + x[r+2][c]
            m.Add(s >= 1); m.Add(s <= 2)
    s = cp_model.CpSolver()
    ok = s.Solve(m)
    if ok not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        raise RuntimeError("no solution")
    return [[s.Value(x[r][c])
             for c in range(n)] for r in range(n)]
```

The toy instance solves in about 14 milliseconds.

A hard instance

Figure 5.3 is puzzle 140 from the janko.at Tohu-Wa-Vohu archive, Otto Janko's own, rated at the top tier of the site's difficulty scale. The grid is 14 × 14; seventy cells are given, leaving one hundred and twenty-six to deduce. The same solver code returns the unique completion in about

3 milliseconds, faster than the toy despite the larger grid: with a denser set of givens the local inequalities propagate earlier and the solver's search tree is narrower.

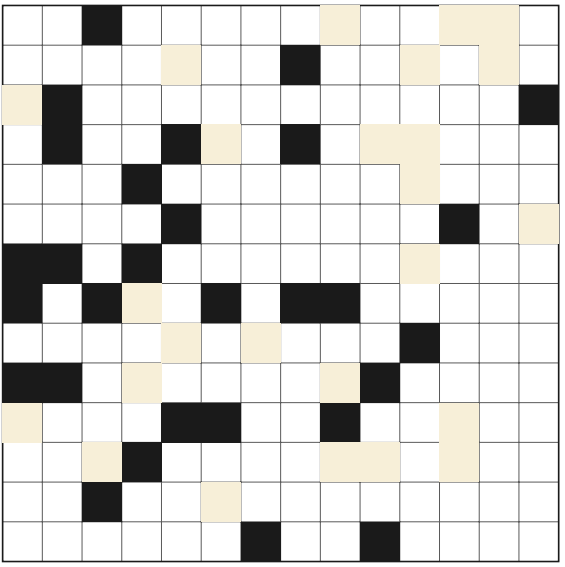

Figure 5.3: *Tohu-Wa-Vohu* 140 *from janko.at, Otto Janko* 14 × 14. *Seventy givens; three-in-a-row forbidden in either direction.*

The variant with distinct rows and columns

A stricter version of the puzzle, often marketed as *Binairo*, adds a third rule: no two rows are identical and no two columns are identical. This turns the puzzle into a kind of binary Latin-square relative: the rows and columns are $n/n/2$ balanced binary strings avoiding triple repeats, and the grid must be a set of such strings.

The distinctness constraint is cheap to add. For each pair of rows (r_1, r_2) with $r_1 < r_2$, require at least one column at which

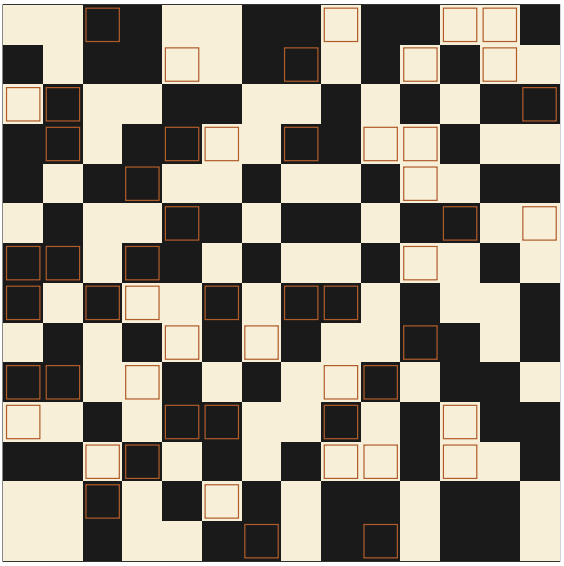

Figure 5.4: *The unique completion, verified against the janko.at solution. Copper-outlined cells were given; the rest is recovered by the solver in about 3 ms.*

they differ:

$$\sum_{c=0}^{n-1} \mathbf{1}[x_{r_1,c} \neq x_{r_2,c}] \;\geq\; 1,$$

reified cell-by-cell into booleans. The same for each pair of columns. In the CP-SAT model this adds $\binom{n}{2}$ row-pair constraints and $\binom{n}{2}$ column-pair constraints, each with n auxiliary booleans: $O(n^3)$ booleans in total. For $n = 14$ that is manageable; for n larger than about 20 the model starts to hurt and a dedicated all-different constraint over row-integers is preferable.

Curiously, the janko.at puzzle 140 does *not* satisfy the distinctness rule: one column appears twice in the solution. Whether the author intended the strict Binairo rules or the three-in-a-row rules alone is a curatorial question. The solver code above uses the latter; switching to the Binairo rules would have rendered the published puzzle infeasible.

Sources. The three-in-a-row prohibition and the equal-counts rule are due to Adolfo Zanellati (Italy, c. 2009), who marketed the puzzle as *Binairo*. The Hebrew-language naming *Tohu-Wa-Vohu* comes from the janko.at archive; puzzle 140 in that archive is by Otto Janko. Nikoli publishes the three-in-a-row variant under various Japanese names. The CP-SAT formulation here follows the natural 0/1 ILP approach; for a decision-theoretic study of the puzzle's complexity, see Utiyama and Uehara, *Computational complexity of the Binairo puzzle*, Proceedings of IPSJ Workshop on Game Informatics 27 (2012).

Chapter 6

KenKen

KenKen is Sudoku with arithmetic. The grid is again an $n \times n$ Latin square: each row and each column a permutation of $1, \ldots, n$. Layered on top, the grid is partitioned into *cages*, each carrying an arithmetic clue: the cells in a cage must combine, by addition, multiplication, subtraction, or division, to a target value declared in the cage's top-left corner.

The puzzle was invented by the Japanese mathematics teacher Tetsuya Miyamoto in 2004, originally as a teaching device for elementary-school students learning arithmetic; he called it *kashikoku naru puzzle* ('a puzzle that makes you smarter'). The Tokyo publisher Gakken adopted it and renamed it KenKen, short for *kenkou ken'i*, 'wisdom and cleverness'. Among many trademarked variants in English-speaking markets it goes by *Calcudoku*, *Mathdoku*, and the New York Times' *KenKen*.

The constraint set extends Sudoku in the cleanest possible way: the two all-different families remain, and every cage contributes one further equation. The arithmetic is local; the all-differents are global. CP-SAT's `AddMultiplicationEquality` and `AddAbsEquality` primitives make the cage clues a few lines each. The result is a model that, like Sudoku, scales smoothly from a 4×4 teaching example to the 9×9 puzzles that test computer solvers.

Rules and a small instance

A KenKen puzzle is an $n \times n$ grid. Each cell holds a digit from 1 to n. The grid is partitioned into *cages*: each cage is a connected polyomino of one or more cells, and each carries an arithmetic clue.

1. Each row contains each digit from 1 to n exactly once.
2. Each column contains each digit from 1 to n exactly once.
3. For each cage with target t and operator $\circ \in \{+, -, \times, \div\}$, the cell values combine under $\circ$ to t. Singleton cages carry only a target t and force the cell to hold t.

The arithmetic conventions need a moment. For addition and multiplication the cage's cells combine in the natural order-free way: their sum or product equals the target. For subtraction and division, which are not commutative, the convention is to take the absolute difference (subtraction) or the larger divided by the smaller (division), so the order of cells in the cage does not matter. By tradition, subtraction and division cages contain exactly two cells.

Figure 6.1 shows a 4×4 KenKen with nine cages, including two singleton 'givens' that are KenKen's analogue of Sudoku starter clues.

The programming model

Let $x_{r,c} \in \{1, \ldots, n\}$ for $0 \le r, c < n$. The Latin square base reuses the Sudoku constraints:

$$\begin{aligned} &\text{alldifferent}(\{x_{r,c} : c\}) && \text{for each row } r, \\ &\text{alldifferent}(\{x_{r,c} : r\}) && \text{for each column } c. \end{aligned}$$

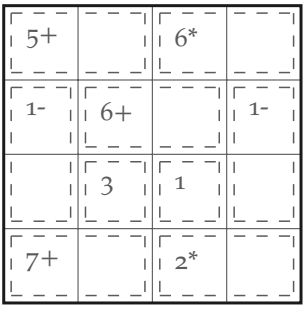

Figure 6.1: *A* 4×4 *KenKen with nine cages. Cage borders are dashed; the operator and target sit in the top-left cell of each cage.*

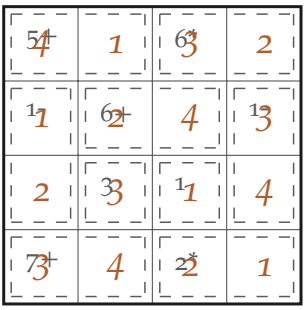

Figure 6.2: *The unique completion. Each row and column contains* $\{1, 2, 3, 4\}$*; each cage's contents combine under its operator to the target.*

Each cage with target t, operator $\circ$, and cell set C contributes one further constraint:

$$\begin{aligned}
\text{Sum:} &\quad \sum_{(r,c)\in C} x_{r,c} = t. \\
\text{Product:} &\quad \prod_{(r,c)\in C} x_{r,c} = t. \\
\text{Difference } (|C| = 2): &\quad |x_{r_1,c_1} - x_{r_2,c_2}| = t. \\
\text{Quotient } (|C| = 2): &\quad x_{r_1,c_1} = t \cdot x_{r_2,c_2} \text{ or } x_{r_2,c_2} = t \cdot x_{r_1,c_1}. \\
\text{Given:} &\quad x_{r,c} = t.
\end{aligned}$$

The product and difference constraints need a touch of CP-SAT plumbing. `AddMultiplicationEquality` ties an

auxiliary variable to the product of a list of variables in one call; `AddAbsEquality` ties an auxiliary to the absolute value of a single variable, used for the difference cage. The quotient cage uses two reified linear constraints under a Boolean indicator choosing which of the two cells is the dividend.

A solver in thirty lines

```
from ortools.sat.python import cp_model

def solve_kenken(n, cages):
    """cages: list of (op, target, [(r, c), ...]).
       op in {'+', '*', '-', '/', '='}."""
    m = cp_model.CpModel()
    x = [[m.NewIntVar(1, n, f"x{r}{c}")
          for c in range(n)] for r in range(n)]
    for r in range(n):
        m.AddAllDifferent(x[r])
    for c in range(n):
        m.AddAllDifferent([x[r][c] for r in range(n)])
    for op, t, cells in cages:
        vs = [x[r][c] for r, c in cells]
        if op == '+':
            m.Add(sum(vs) == t)
        elif op == '*':
            p = m.NewIntVar(1, n**len(vs), "")
            m.AddMultiplicationEquality(p, vs)
            m.Add(p == t)
        elif op == '-':
            d = m.NewIntVar(-n, n, "")
            ad = m.NewIntVar(0, n, "")
            m.Add(d == vs[0] - vs[1])
            m.AddAbsEquality(ad, d)
            m.Add(ad == t)
        elif op == '/':
            a, b = vs
            pick = m.NewBoolVar("")
            m.Add(a == t * b).OnlyEnforceIf(pick)
            m.Add(b == t * a).OnlyEnforceIf(pick.Not())
        elif op == '=':
            m.Add(vs[0] == t)
    s = cp_model.CpSolver()
    ok = s.Solve(m)
    if ok not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        raise RuntimeError("no solution")
    return [[s.Value(x[r][c])
             for c in range(n)] for r in range(n)]
```

The toy of Figure 6.1 solves in about 23 milliseconds and admits a single completion.

✠

A hard instance

Figure 6.3 is the 9 × 9 instance from the recreational maths archive, billed as the hardest KenKen recorded there. Thirty-three cages of mixed operator and length sit on top of the 9×9 Latin square base; the largest sums extend across five cells, the largest product is 378 over four cells. CP-SAT returns the unique solution (Figure 6.4) in about 29 milliseconds.

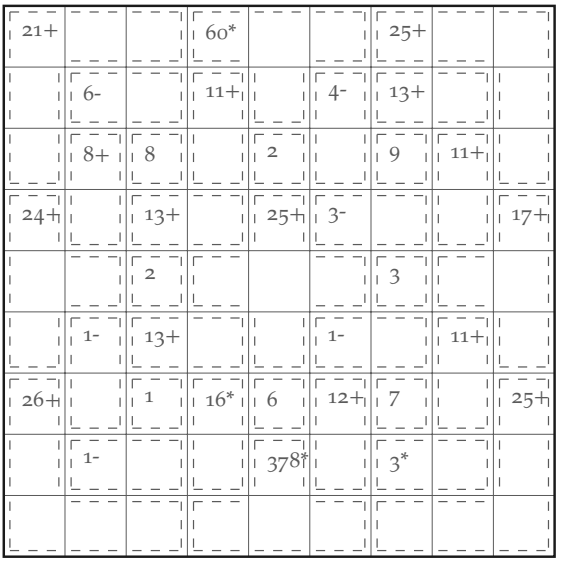

Figure 6.3: *A hard* 9×9 *KenKen with thirty-three cages. The largest cage carries the sum-clue* 25 *over five cells; the most demanding is the product-clue* 378 *over four.*

The propagation that makes the hard puzzle tractable is again the all-different constraint working in tandem with cage-local arithmetic. A cage like 378 = product over four cells, on a 9×9 grid, has the factorisations $\{2, 3, 7, 9\}$, $\{3, 6, 7, ?\}$

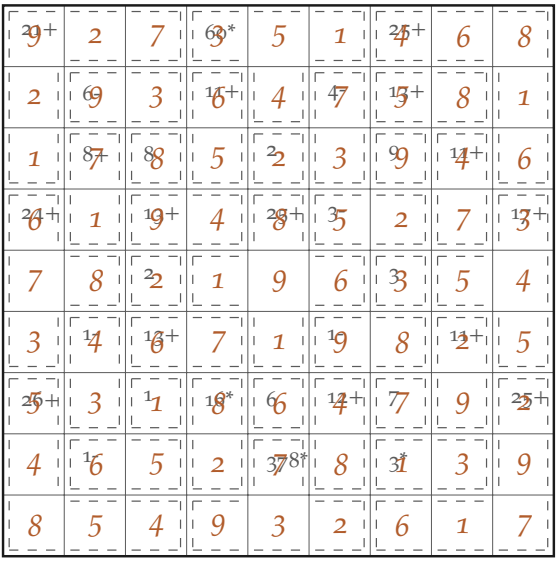

Figure 6.4: *The unique completion, recovered in about* 29 *ms.*

and a small handful of others over digits 1 to 9; CP-SAT enumerates these implicitly and prunes early.

✠

Sources. KenKen was invented by Tetsuya Miyamoto in 2004 and first published by Gakken in Japan as *kashikoku naru puzzle*. The puzzle reached an English audience through Will Shortz, the New York Times' crossword editor, in 2008. Robert Fuhrer's KenKen Puzzle LLC holds the trademark; *Calcudoku* and *Mathdoku* are widely-used free names for the same puzzle. The hard 9 × 9 instance is sourced from the recreational mathematics archive accompanying this book; its CP-SAT formulation matches the original Z3 model constraint by constraint.

Chapter 7

Flow Free

Flow Free is the puzzle of disjoint paths. The grid holds a set of coloured dots paired up by colour: two yellows, two blues, two reds. The task is to draw, for each colour, an unbroken path through orthogonally adjacent grid cells joining its two dots, with the strict additional rule that every cell of the grid must lie on exactly one path. No path crosses itself, no two paths share a cell, no cell is left blank.

The puzzle is sold under the trademark Flow Free by Big Duck Games (2012), but its lineage is older. Nikoli published essentially the same puzzle for years under the name *Arukone* (a contraction of *aru* "some" and *kone* "connection"), itself a relative of the long-standing *Number Link* puzzles by Walter Mott in 1897. The polynomial-time complexity of the two-colour version is classical; with three or more colours the puzzle is NP-complete (Adcock et al., 2015), making it the first in this book to be NP-complete by basic combinatorics rather than by sheer combinatorial scale.

The model in this chapter is, despite the complexity of the puzzle, strikingly compact: a colour-indicator variable per cell-colour pair, plus a degree constraint per cell-colour saying that endpoints have one same-colour neighbour and interior cells have two. The trick that makes it work is the strict "every cell must be filled" clause. Without it, paths could short-circuit through empty cells and the model would need a connectivity layer; with it, the degree constraints alone almost suffice, and a well-formed puzzle resolves uniquely.

✠

Rules and a small instance

A Flow Free puzzle is an $n \times n$ grid with K colours. Each colour $k \in \{1, \dots, K\}$ is assigned exactly two cells, its *endpoints*; remaining cells are unmarked.

1. Each unmarked cell receives exactly one colour from $\{1, \dots, K\}$.
2. For each colour k, the cells coloured k form a single path between the two endpoints of k. Equivalently: each endpoint of k has exactly one orthogonal neighbour of colour k, and every other cell of colour k has exactly two.
3. Every cell is covered by some colour (the *fill* property).

Figure 7.1 is a 5×5 Flow Free with four colours. The endpoints are drawn as small filled disks labelled with the colour number; the puzzle is to thread a path of each colour from dot to dot, covering every cell exactly once.

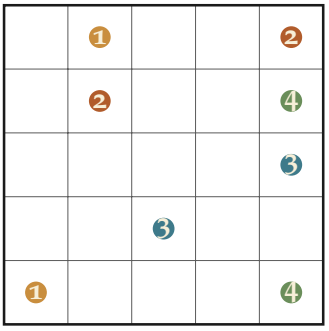

Figure 7.1: *A* 5×5 *Flow Free with four colours. Endpoint dots show the cells to connect; every empty cell must end up on exactly one colour's path.*

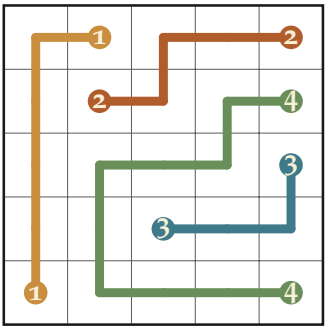

Figure 7.2: *The unique solution. Each path connects its two endpoints; together they cover all twenty-five cells.*

✠

The programming model

For each cell (r, c) and each colour k, introduce a Boolean variable $x_{r,c,k} \in \{0, 1\}$ that is 1 iff cell (r, c) is coloured k. The model has three families of constraints.

1. *Single colour per cell.* For each cell, $\sum_k x_{r,c,k} = 1$.
2. *Endpoints fixed.* For each pair of endpoints $\{e_1, e_2\}$ of colour k, set $x_{e_1,k} = 1$ and $x_{e_2,k} = 1$.
3. *Degree.* For each cell (r, c) and each colour k, if $x_{r,c,k} = 1$ then the count of k-coloured orthogonal neighbours equals 1 (when (r, c) is an endpoint of colour k) or 2 (otherwise).

The degree constraint is reified: $\sum_{(r',c')\sim(r,c)} x_{r',c',k} = t_{r,c,k}$ holds only when $x_{r,c,k} = 1$, where $t_{r,c,k} \in \{1, 2\}$ depending on endpoint status. CP-SAT expresses this with `OnlyEnforceIf`; no further machinery is needed.

A note on connectivity

The degree-and-fill conditions are necessary for a valid solution but not strictly sufficient: a colour could in principle form a path between its endpoints *plus* a separate disjoint cycle, and every cell would still have the right degree. In practice, a well-formed Flow Free puzzle has a unique grid-filling solution and the unique solution has no separate cycles; the degree model returns it. For adversarial inputs one would add a connectivity constraint (a multi-commodity flow on the cell graph, or a position-along-the-path variable for each cell). For every puzzle in this chapter the degree model alone suffices, and we verify the result by checking each colour's induced subgraph is connected.

A solver in twenty lines

```
from ortools.sat.python import cp_model

def nbrs(r, c, n):
    for dr, dc in [(-1,0),(1,0),(0,-1),(0,1)]:
        if 0 <= r+dr < n and 0 <= c+dc < n:
            yield (r+dr, c+dc)

def solve_flow(n, eps):
    """eps: dict {colour: [(r1, c1), (r2, c2)]}."""
    K = sorted(eps)
    m = cp_model.CpModel()
    x = {(r, c, k): m.NewBoolVar("")
         for r in range(n) for c in range(n) for k in K}
    for r in range(n):
        for c in range(n):
            m.Add(sum(x[(r, c, k)] for k in K) == 1)
            for k in K:
                if (r, c) in eps[k]:
                    m.Add(x[(r, c, k)] == 1)
    for r in range(n):
        for c in range(n):
            for k in K:
                same = [x[(nr, nc, k)]
                        for nr, nc in nbrs(r, c, n)]
                t = 1 if (r, c) in eps[k] else 2
                m.Add(sum(same) == t).OnlyEnforceIf(
                    x[(r, c, k)])
    s = cp_model.CpSolver()
    s.Solve(m)
```

```
    return [[next(k for k in K
                  if s.Value(x[(r, c, k)]))
             for c in range(n)] for r in range(n)]
```

The toy solves in about 5 milliseconds.

A larger instance: Arukone 300

Figure 7.3 is puzzle 300 from the janko.at Arukone archive, a 15×15 instance by ARA3 with ten colour pairs. Two of the ten pairs sit close together (the green pair occupies cells $(2, 4)$ and $(2, 10)$, on the same row); the remaining eight thread across the grid. The same solver code, called on this instance, returns the unique completion in about 90 milliseconds.

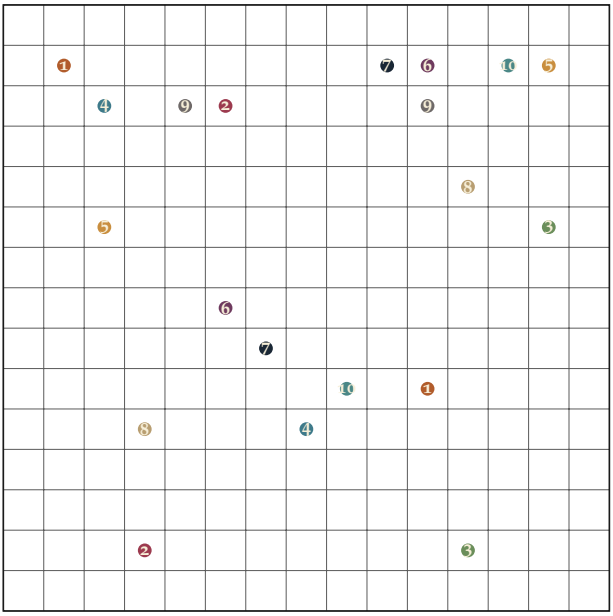

Figure 7.3: *Arukone* 300 *from janko.at* (*ARA3*), 15×15 *with ten colour pairs.*

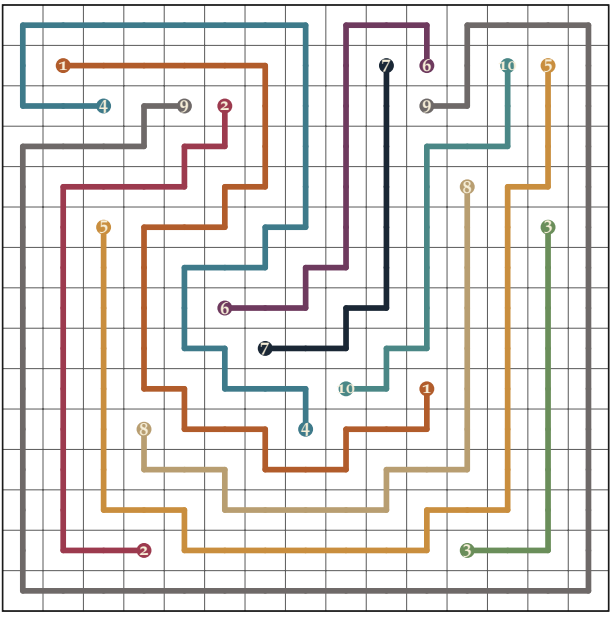

Figure 7.4: *The unique solution. Path* 9 *snakes around the border, while* 4 *and* 1 *make long internal detours.*

Sources. The Numberlink puzzle was first published by Walter Mott in *Tit-Bits* magazine in 1897; Nikoli rebranded a version of it as *Arukone* in the early 1980s. The Flow Free mobile game by Big Duck Games (2012) revived the puzzle in its strict-fill form. Adcock, Demaine, Demaine, O'Brien, Reidl and Sullivan, *Zig-zag Numberlink is NP-complete*, Journal of Information Processing 23 (2015), 239–245, established NP-completeness for the version with three or more colour pairs and the all-cells-covered constraint. The hard puzzle in Figure 7.3 is Arukone 300 from `janko.at`, attributed to ARA3 (`numberlink.ara3.net`).

Chapter 8

Squares Sudoku

Squares Sudoku is Killer Sudoku, restricted to perfect squares. The base puzzle is an ordinary 9×9 Sudoku grid: each row, column, and 3×3 box holds the digits 1 through 9 exactly once. Onto this base sits a partition of the grid into *cages*, each a connected polyomino of one or more cells, with one further demand: the digits in each cage sum to a perfect square. The squares allowed by convention are the small ones a 1-to-9 digit sum can reach, namely 4, 9, 16, and 25.

The puzzle sits in the family of Killer-style Sudoku variants that emerged in the wake of the early-2000s Sudoku boom, designed to add an arithmetic dimension without losing the purity of the underlying Latin-square logic. Killer Sudoku itself attaches an explicit sum value to each cage; Squares Sudoku replaces those values with the simple membership test "the cage's sum is one of $\{4, 9, 16, 25\}$". The model gains an *or* where Killer Sudoku has an $=$, and the puzzle becomes strictly harder for human solvers because the cage's target is hidden behind an enumeration.

For a CP solver the change is small. The Killer-cage equation $\sum_{(r,c)\in C} x_{r,c} = t$ becomes $\sum_{(r,c)\in C} x_{r,c} \in \{4, 9, 16, 25\}$, encoded as a one-hot Boolean over the four allowed targets with a reified equality on each. The solver searches over the joint space of digit assignment and target-square selection.

Rules and a small instance

A Squares Sudoku puzzle is a 9×9 grid of cells. The grid carries a partition into cages, each a connected polyomino of one or more cells. The puzzle's constraints are the three Sudoku constraints from Chapter 2 together with a fourth.

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 3×3 box contains each digit from 1 to 9 exactly once.
4. For every cage, the sum of its cells' digits lies in $\{4, 9, 16, 25\}$.

Unlike Killer Sudoku, the cage-sum target is not pre-declared; the solver must choose, for each cage, which perfect square the cage will hit. The fourth rule is therefore a kind of "meta-clue": it constrains the structure of the digit assignment without committing to a specific target per cage.

Figure 8.1 shows the 9×9 instance from the recreational mathematics archive: 24 cages of varying shape, no givens, and the perfect-square sum rule applied to each cage.

The programming model

Let $x_{r,c} \in \{1, \dots, 9\}$ for $0 \le r, c < 9$. The Sudoku layer is the three families of all-different constraints from Chapter 2. The cage layer adds, for each cage with cell set C, the constraint

$$\sum_{(r,c) \in C} x_{r,c} \;\in\; \{4, 9, 16, 25\}.$$

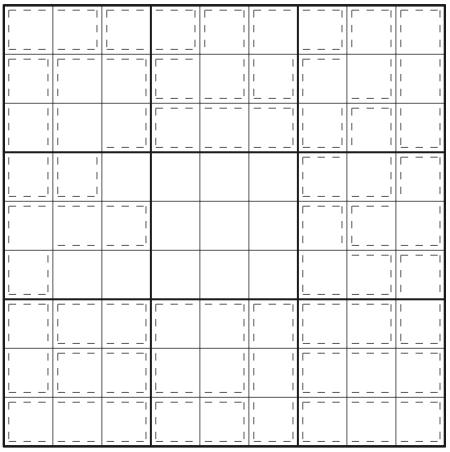

Figure 8.1: *A Squares Sudoku. Cages drawn with thin dashed borders; each cage's contents must sum to a perfect square in* $\{4, 9, 16, 25\}$.

In CP-SAT this is modelled with a one-hot Boolean for the four allowed targets:

$$\sum_{t \in \{4,9,16,25\}} b_{C,t} \;=\; 1, \qquad \sum_{(r,c) \in C} x_{r,c} \;=\; t \;\text{ when } b_{C,t} = 1,$$

the second equation reified against the indicator $b_{C,t}$ through `OnlyEnforceIf`. With 24 cages and four allowed targets, the additional Boolean count is 96, modest by CP-SAT standards.

A solver in twenty-five lines

```
from ortools.sat.python import cp_model

SQUARES = [4, 9, 16, 25]

def solve_squares_sudoku(cages):
    n = 9
    m = cp_model.CpModel()
    x = [[m.NewIntVar(1, 9, f"x{r}{c}")
          for c in range(n)] for r in range(n)]
    for r in range(n): m.AddAllDifferent(x[r])
    for c in range(n):
        m.AddAllDifferent([x[r][c] for r in range(n)])
    for br in range(3):
        for bc in range(3):
            block = [x[br*3+i][bc*3+j]
```

```
                    for i in range(3) for j in range(3)]
            m.AddAllDifferent(block)
    for cage in cages:
        s = sum(x[r][c] for (r, c) in cage)
        bs = [m.NewBoolVar("") for _ in SQUARES]
        m.Add(sum(bs) == 1)
        for b, t in zip(bs, SQUARES):
            m.Add(s == t).OnlyEnforceIf(b)
    sv = cp_model.CpSolver()
    sv.Solve(m)
    return [[sv.Value(x[r][c])
             for c in range(n)] for r in range(n)]
```

The puzzle in Figure 8.1 solves uniquely in about 70 milliseconds, returning Figure 8.2.

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

Figure 8.2: *The unique completion. Each cage's digit sum is one of* 4, 9, 16, 25.

A spot-check confirms the rule. The two-cell cage $\{(0,0),(0,1)\}$ holds $\{6,3\}$, summing to 9. The five-cell cage $\{(0,7),(1,7),(1,6),(2,6)\}$ holds $\{7,9,5,4\}$, summing to 25. The two-cell cages $\{(6,6),(6,7)\}$, $\{(7,1),(7,2)\}$, $\{(8,3),(8,4)\}$ each hold pairs summing to 4, forcing the rare $\{1,3\}$ partition (since $\{2,2\}$ is forbidden by row distinctness).

✠

The arithmetic of the constraint

Each cage's sum constraint, before the Sudoku layer is imposed, admits a small number of cell-value assignments. A two-cell cage summing to 4 has the single multiset $\{1,3\}$; one summing to 9 has $\{1,8\}, \{2,7\}, \{3,6\}, \{4,5\}$; one summing to 16 has $\{7,9\}$; one summing to 25 is impossible (maximum two-digit sum is $9+8=17$). A three-cell cage summing to 4 requires $\{1,1,2\}$, which row distinctness forbids; summing to 9 has eight multisets; summing to 25 has three. The number of feasible cage-internal assignments shrinks fast with cage size and stays small enough that CP-SAT's constraint propagation prunes early; the solver's actual search work is in the global Sudoku layer.

The choice of allowed squares matters. With $\{1,4,9,16,25\}$ allowed, single-cell cages of value 1 would be permitted but not constraint-bearing. With $\{4,9,16,25,36,49\}$, larger cages of five or more cells (whose sums can exceed 25) become viable. The conventional set $\{4,9,16,25\}$ keeps the puzzle restricted to interesting cage shapes.

✠

Sources. Killer Sudoku, the parent of the Squares Sudoku variant, was popularised in Japan by Tetsuya Nishio and in the West through The Times of London in 2005; the Japanese name is *sankaku* or *Sumdoku*. The specific perfect-squares restriction used here is documented in the recreational mathematics archive accompanying this book; the unique-solution check was performed by enumeration with CP-SAT, and the puzzle does have a single completion under the constraint set $\{4,9,16,25\}$. The 9×9 instance is original to that archive.

Chapter 9

Slitherlink

Slitherlink draws a single closed loop on a lattice of dots. The grid is a rectangle of cells; some cells carry a small clue digit between 0 and 3. The solver's task is to draw a non-self-crossing closed curve along the lattice edges between dots, such that for every clue cell the number of edges on the curve incident to that cell exactly equals the clue. A clue of 0 forbids any of the cell's four edges from being on the loop; a 3 forces three of them on; a 1 or 2 leaves exactly one or two free.

The puzzle was published in 1989 in Nikoli's puzzle magazine under the name *Suri-rinku* (Japanese for "slitherlink"), designed by an editor going by the handle Renin. It travelled to the West in the late 1990s and is now solved by aficionados in several languages, often under the names *Loop the Loop* or *Fences*. Of all Nikoli puzzles, Slitherlink is perhaps the purest of the loop-puzzle family: the entire content of a solution is a single closed curve.

The new modelling ingredient is the curve itself. Up to this chapter the puzzles in this book have placed labels in cells. Here the unknowns live on the edges between dots, and the non-trivial constraint is that the chosen edges form a single closed loop, not two or more. CP-SAT supports this directly through its `AddCircuit` primitive, which lets a CP model declare that a set of directed arcs must form a Hamiltonian circuit; we use it via the standard trick that lets vertices "skip" the loop with a self-loop arc.

✠

Rules and a small instance

A Slitherlink puzzle is an $R \times C$ rectangle of cells. Vertices sit at lattice points (r, c) with $0 \le r \le R$ and $0 \le c \le C$; edges connect each pair of horizontally or vertically adjacent vertices. A subset of cells carries a *clue*: an integer from 0 to 3.

1. Each edge is either *on* the loop or *off*.
2. For each clue cell, the number of its four bounding edges that are on the loop equals the clue.
3. The set of on-edges forms a single closed loop: a non-self-crossing closed curve on the edge graph.

Equivalently, every vertex in the lattice has degree 0 or 2 in the on-edge graph (degree 0 if not on the loop, 2 if on), and the on-edges form a single connected component.

Figure 9.1 is Oleg Andrushko's puzzle 30 from the janko.at archive, a 6×6 instance with sixteen clues.

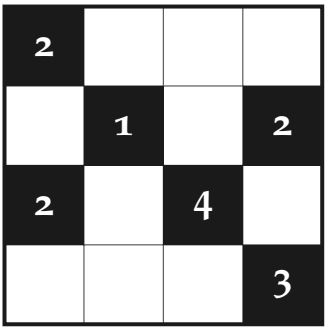

Figure 9.1: *A 6 × 6 Slitherlink (Oleg Andrushko, via janko.at). Lattice dots mark the vertices; clues sit in cell interiors. The task is to thread a single closed loop along the dot lattice satisfying every clue.*

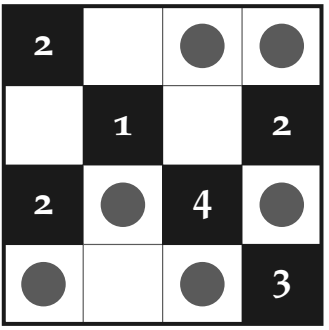

Figure 9.2: *The unique closed loop. The two* 3*s in diagonally adjacent cells force the loop to turn around their shared corner; the* 0 *at* $(2,3)$ *keeps the loop entirely off that cell.*

The programming model

Let $h_{r,c}$ denote the boolean for the horizontal edge from vertex (r,c) to vertex $(r,c+1)$, and $v_{r,c}$ for the vertical edge from (r,c) to $(r+1,c)$. A cell at (r,c) has bounding edges $h_{r,c}$ (top), $h_{r+1,c}$ (bottom), $v_{r,c}$ (left), and $v_{r,c+1}$ (right).

The clue constraint is direct:

$$h_{r,c} + h_{r+1,c} + v_{r,c} + v_{r,c+1} \;=\; \text{clue}(r,c) \quad \text{for each clue cell.}$$

The vertex degree constraint, for each lattice vertex (r,c), is that the sum of its incident edges is 0 or 2. CP-SAT expresses this with a Boolean indicator: a variable $in_{r,c}$ that is 1 iff the vertex is on the loop, with $\sum_{e \text{ incident}} e = 2in_{r,c}$.

The single-loop constraint

The clue and degree constraints alone admit configurations with multiple disjoint loops. To rule those out, the model uses CP-SAT's `AddCircuit` primitive. The trick: turn each undirected edge into two directed arcs (one per direction); add a self-loop arc at every vertex that is enabled iff the vertex is off the loop; then `AddCircuit` on the union of arcs forces the active set to be a single Hamiltonian circuit on the vertex set,

where "Hamiltonian" visits every vertex but uses the self-loop for vertices not on the actual loop.

Concretely, for each undirected edge $\{u, v\}$ we introduce two arc booleans $a_{u\to v}$ and $a_{v\to u}$ with the constraint $a_{u\to v} + a_{v\to u} = e_{uv}$, so exactly one is true when the edge is on the loop, and both are false otherwise. For each vertex w we introduce a self-loop arc s_w with the constraint $s_w = 1$ iff vertex w has degree 0 in the loop. `AddCircuit` on the resulting arc list does the rest.

A solver in forty lines

```
from ortools.sat.python import cp_model

def solve_slitherlink(R, C, clues):
    """clues: {(r, c): k} for clued cells, k in 0..3."""
    m = cp_model.CpModel()
    h = {(r, c): m.NewBoolVar("")
         for r in range(R+1) for c in range(C)}
    v = {(r, c): m.NewBoolVar("")
         for r in range(R) for c in range(C+1)}
    for (r, c), k in clues.items():
        m.Add(h[(r, c)] + h[(r+1, c)]
              + v[(r, c)] + v[(r, c+1)] == k)

    arcs = []
    vid = lambda r, c: r * (C+1) + c
    for (r, c), e in h.items():
        f = m.NewBoolVar(""); b = m.NewBoolVar("")
        m.Add(f + b == e)
        arcs.append((vid(r, c), vid(r, c+1), f))
        arcs.append((vid(r, c+1), vid(r, c), b))
    for (r, c), e in v.items():
        f = m.NewBoolVar(""); b = m.NewBoolVar("")
        m.Add(f + b == e)
        arcs.append((vid(r, c), vid(r+1, c), f))
        arcs.append((vid(r+1, c), vid(r, c), b))
    for r in range(R+1):
        for c in range(C+1):
            edges = []
            if c > 0: edges.append(h[(r, c-1)])
            if c < C: edges.append(h[(r, c)])
            if r > 0: edges.append(v[(r-1, c)])
            if r < R: edges.append(v[(r, c)])
            d = sum(edges)
            sl = m.NewBoolVar("")
```

```
            m.Add(d == 0).OnlyEnforceIf(sl)
            m.Add(d == 2).OnlyEnforceIf(sl.Not())
            arcs.append((vid(r, c), vid(r, c), sl))
    m.AddCircuit(arcs)

    s = cp_model.CpSolver()
    s.Solve(m)
    return ({(r, c): s.Value(h[(r, c)]) for (r, c) in h},
            {(r, c): s.Value(v[(r, c)]) for (r, c) in v})
```

The toy of Figure 9.1 solves in about 5 milliseconds.

A hard instance

Figure 9.3 is puzzle 332 from the janko.at Slitherlink archive, a 12 × 16 instance by Hirofumi Fujiwara graded at the site's top difficulty tier. The clue density is about half the cells; the loop must thread 122 edges through all 192 cells without intersecting itself. The same solver code returns the unique loop in about 22 milliseconds.

Sources. Slitherlink was first published by Nikoli in 1989 in the magazine *Puzzle Communication Nikoli* under the Japanese name *Suri-rinku*, attributed to the editor Renin. The puzzle is also known in English as *Loop the Loop* or *Fences*. The toy is puzzle 30 in the janko.at Slitherlink archive (Oleg Andrushko); the hard instance is puzzle 332 (Hirofumi Fujiwara). The `AddCircuit`-with-self-loops technique used in the model is documented in the OR-Tools reference and is the standard CP modelling trick for connectivity-with-skip on undirected graphs.

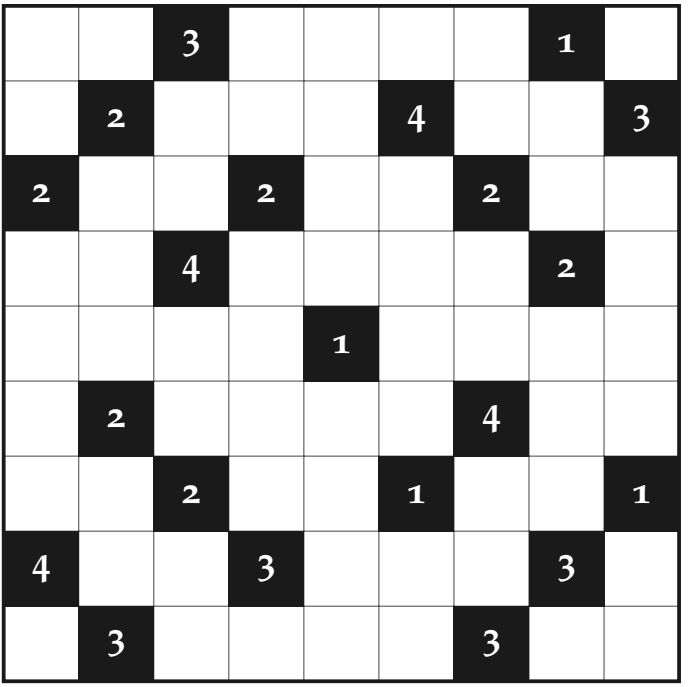

Figure 9.3: *Slitherlink* 332 *from janko.at* (*Hirofumi Fujiwara*), 12×16, *top-tier difficulty.*

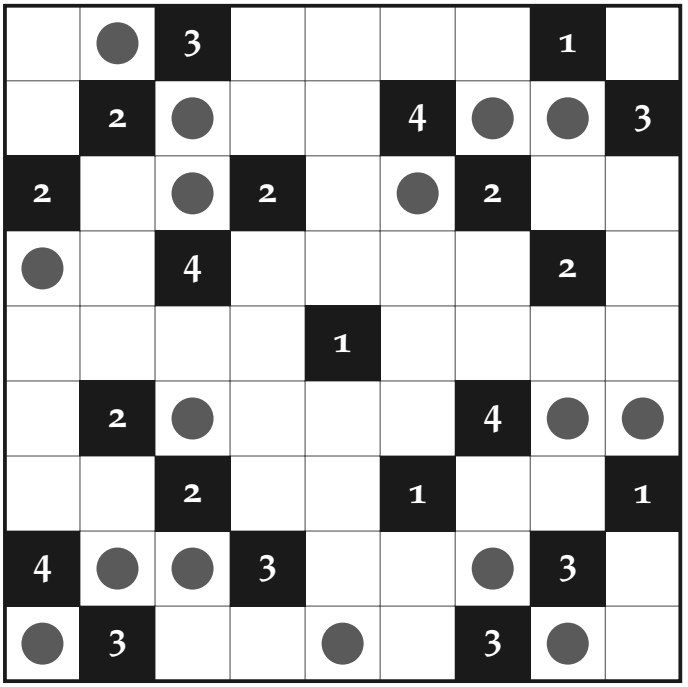

Figure 9.4: *The unique closed loop, recovered in about* 22 *ms. The loop satisfies every clue cell and visits no vertex more than once.*

Chapter 10

Hashiwokakero

Hashiwokakero is the puzzle of bridges between islands. An archipelago of circular islands sits on a grid; each island carries a small integer in its centre. The solver's task is to draw a network of straight horizontal and vertical bridges between islands such that each island has exactly the declared number of incident bridges, no two bridges cross, between any two islands lies at most a double bridge, and the resulting network is connected: any island can be reached from any other by following bridges.

The puzzle was first published in 1990 in Nikoli's magazine *Puzzle Communication Nikoli*, designed by editor Yamamoto. The Japanese name, *hashi-wo-kakero*, means "build bridges"; in English-language sources the puzzle goes by *Hashi* or *Bridges*. It joins the family of graph-style Nikoli puzzles whose solutions are determined by local degree counts and a global connectivity condition; in this respect it is a close cousin of Slitherlink, with islands replacing lattice corners and bridges replacing loop edges.

The model has three constraint families: a local degree equation per island, a no-crossing constraint per intersecting pair of candidate bridges, and a single-component connectivity constraint enforced through a flow on the candidate-bridge graph.

✠

Rules and a small instance

A Hashiwokakero puzzle is an $R \times C$ grid with a set of *islands*, each at a cell (r, c) and carrying a positive integer *degree* $d_{r,c} \in \{1, \dots, 8\}$. A *bridge* is a pair of horizontally or vertically aligned islands with no other island between them, plus a number of parallel lines connecting them: 0, 1, or 2. The puzzle's constraints:

1. Each island's incident bridge total equals its degree.
2. No two bridges cross: a horizontal bridge and a vertical bridge whose lines intersect cannot both have positive count.
3. The graph of islands joined by positive-count bridges is connected.

Figure 10.1 is a small constructed instance with nine islands on a 7×7 grid. The single hub at the centre has degree 4; the four mid-edge islands have degree 3; the four corner islands have degree 2.

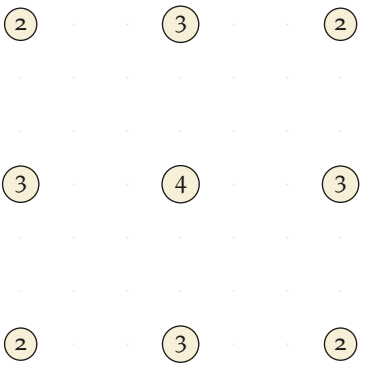

Figure 10.1: *A small Hashiwokakero with nine islands. Each circle's number is the count of bridges to draw incident to that island.*

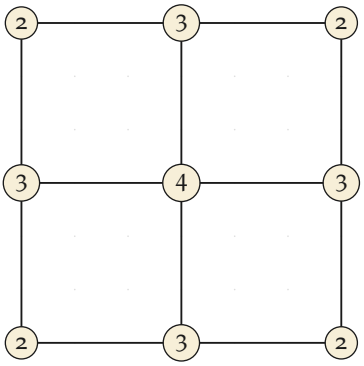

Figure 10.2: *The unique bridge network. Twelve single bridges form a* 3×3 *lattice of islands; every island's degree matches its label.*

The programming model

For each pair of islands (p, q) that lie in the same row or column with no other island between them, introduce an integer variable $b_{p,q} \in \{0, 1, 2\}$ counting the bridges on that pair. Call this set of pairs E.

The degree constraint is one equation per island:

$$\sum_{q:(p,q)\in E} b_{p,q} \;+\; \sum_{q:(q,p)\in E} b_{q,p} \;=\; d_p \quad \text{for every island } p.$$

The no-crossing constraint is one Boolean implication per intersecting pair (h, v) where h is a horizontal candidate bridge and v a vertical one whose line segments cross:

$$b_h \geq 1 \;\Rightarrow\; b_v = 0, \quad b_v \geq 1 \;\Rightarrow\; b_h = 0,$$

encoded by reified booleans on the strict positivity of b_h and b_v together with $\mathbf{1}[b_h \geq 1] + \mathbf{1}[b_v \geq 1] \leq 1$.

Connectivity by flow

The degree and no-crossing constraints alone permit solutions in which the islands break into two or more disjoint bridge components, each internally consistent. To force a single component we enforce a flow network on the candidate-bridge graph. Pick one island p_0 as the *root*. For every other island p, the model must route one unit of flow from p to p_0 along positive-count bridges. We aggregate these into a single signed flow f_e per candidate bridge e: f_e may range in $[-N, N]$ (where N is the island count) and must vanish whenever $b_e = 0$. Conservation at each non-root island demands net incoming flow $= 1$; at the root, net incoming flow $= N - 1$.

Flow exists between every pair of islands iff the bridge graph is connected, so a satisfying flow guarantees a single component.

A solver in fifty lines

```
from ortools.sat.python import cp_model

def solve_hashi(R, C, islands):
    """{(r,c): degree} -> {(p,q): bridges}."""
    iso = list(islands)
    iso_set = set(iso)
    pairs = set()
    for (r, c) in iso:
        for cc in range(c+1, C):
            if (r, cc) in iso_set:
                pairs.add(((r, c), (r, cc))); break
        for rr in range(r+1, R):
            if (rr, c) in iso_set:
                pairs.add(((r, c), (rr, c))); break
    pairs = sorted(pairs)
    m = cp_model.CpModel()
    b = {p: m.NewIntVar(0, 2, "") for p in pairs}
    for (r, c), d in islands.items():
        m.Add(sum(b[p] for p in pairs if (r, c) in p) == d)
    H = [p for p in pairs if p[0][0] == p[1][0]]
    V = [p for p in pairs if p[0][1] == p[1][1]]
    for h in H:
        (r, c1), (_, c2) = h
        for v in V:
            (r1, c), (r2, _) = v
```

```
            if r1 < r < r2 and c1 < c < c2:
                bh = m.NewBoolVar(""); bv = m.NewBoolVar("")
                m.Add(b[h] >= 1).OnlyEnforceIf(bh)
                m.Add(b[h] == 0).OnlyEnforceIf(bh.Not())
                m.Add(b[v] >= 1).OnlyEnforceIf(bv)
                m.Add(b[v] == 0).OnlyEnforceIf(bv.Not())
                m.Add(bh + bv <= 1)
    N = len(iso); root = iso[0]
    f = {p: m.NewIntVar(-N, N, "") for p in pairs}
    for p in pairs:
        bp = m.NewBoolVar("")
        m.Add(b[p] >= 1).OnlyEnforceIf(bp)
        m.Add(b[p] == 0).OnlyEnforceIf(bp.Not())
        m.Add(f[p] == 0).OnlyEnforceIf(bp.Not())
    for u in iso:
        s = []
        for p in pairs:
            if p[0] == u: s.append(-f[p])
            elif p[1] == u: s.append(f[p])
        m.Add(sum(s) == (N - 1 if u == root else -1))
    sv = cp_model.CpSolver()
    sv.Solve(m)
    return {p: sv.Value(b[p])
            for p in pairs if sv.Value(b[p]) > 0}
```

The toy of Figure 10.1 solves in about 4 milliseconds.

A hard instance

Figure 10.3 is puzzle 539 from the janko.at Hashiwokakero archive, a 14×14 instance by Otto Janko graded at difficulty 7. It carries sixty-four islands; the unique bridge network has sixty-five bridges, with several double bridges and a long-running connectivity that threads across most of the grid. The same solver code returns the solution in about 4 milliseconds.

Sources. Hashiwokakero was first published by Nikoli in 1990. The puzzle is also known as *Hashi*, *Bridges*, or *Chopsticks*. The

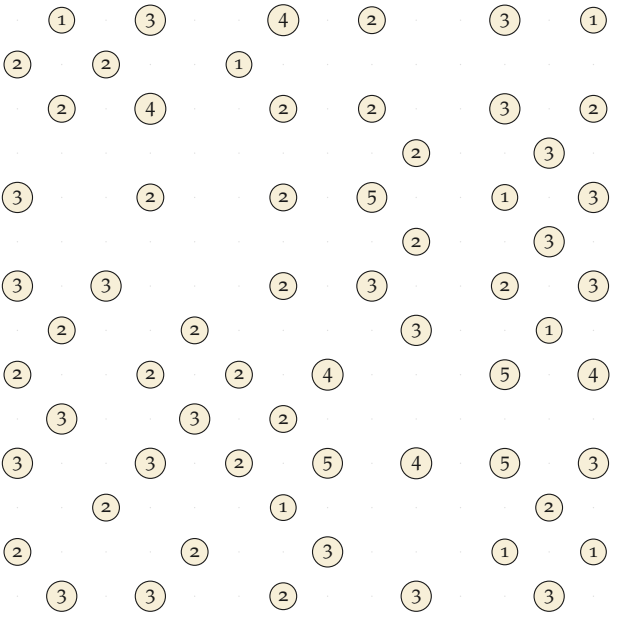

Figure 10.3: *Hashiwokakero* 539 *from janko.at*, 14 × 14, *sixty-four islands.*

hard instance is puzzle 539 from the janko.at Hashiwokakero archive, by Otto Janko. The flow-based connectivity formulation is standard for network-design integer programs; for a thorough treatment of single-commodity flow as a connectivity constraint see Magnanti and Wolsey, *Optimal trees*, in Handbooks in Operations Research and Management Science, vol. 7 (1995).

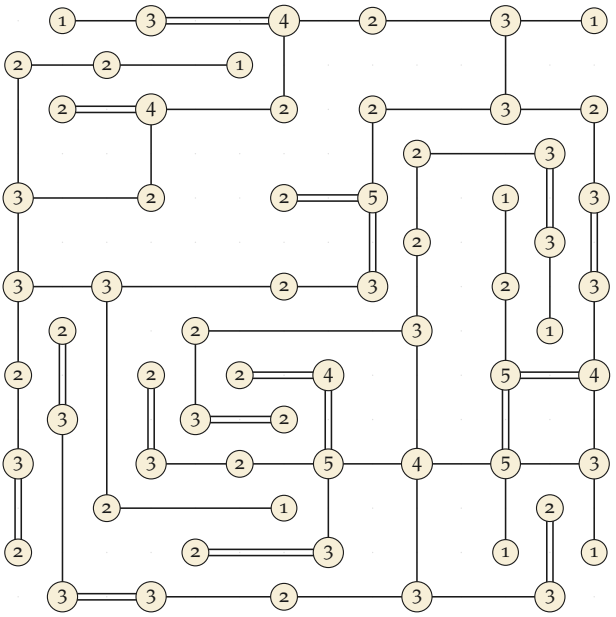

Figure 10.4: *The unique bridge network. Sixty-five bridges, several of them doubled; the entire network is connected.*

Chapter 11

Akari

Akari is a puzzle of darkness and lamplight. The grid is a rectangle of cells, some white, some black. Black cells either carry a clue digit between 0 and 4 or are unmarked. The solver places *lamps* on a subset of white cells. A lamp illuminates every white cell in its row and column out to the next black cell. Three constraints govern the placement.

1. Every white cell is illuminated by at least one lamp.
2. No lamp illuminates another lamp: no two lamps share a row or a column without a black cell between them.
3. Every numbered black cell has exactly that many lamps among its four orthogonal neighbours.

The puzzle was published by Nikoli in 2001 under the name *Akari* (Japanese for "light"), and is sometimes marketed in English as *Light Up* or *Bijutsukan*. It joins the shading-puzzle family alongside Nurikabe and Heyawake, but with a constraint of physical-light propagation rather than combinatorial connectivity.

The CP-SAT model is direct: a Boolean variable per white cell indicating "lamp here", three families of constraints matching the three rules. The rules together are tight enough that the solver returns in milliseconds for puzzles up to 20×20.

Rules and a small instance

An Akari puzzle is an $R \times C$ grid. Each cell is one of three kinds: *white* (the candidate lamp positions), *black with no clue* (impassable), or *black with a clue* $k \in \{0, 1, 2, 3, 4\}$.

Figure 11.1 is puzzle 101 from the janko.at Akari archive, a 10×10 instance with eleven black cells of which seven carry clues.

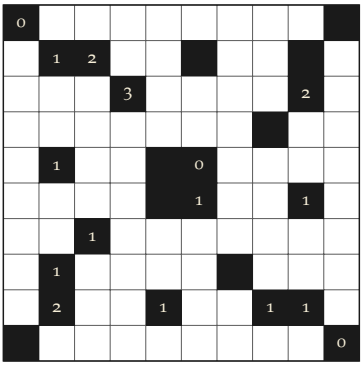

Figure 11.1: *Akari* 101 *from janko.at* (*Warai Kamosika*), 10×10. *Black cells are blocking; numbers on black cells declare the count of lamp neighbours.*

The programming model

For each white cell (r, c) define a Boolean $L_{r,c} \in \{0, 1\}$, with $L_{r,c} = 1$ if a lamp is placed in that cell. Black cells carry no variable.

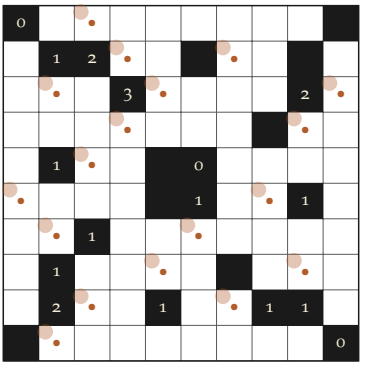

Figure 11.2: *The unique lamp placement. Copper dots mark lamps; faint copper circles mark the cells those lamps occupy before lighting their rows and columns.*

Illumination

For a white cell (r, c), write $\mathcal{S}(r, c)$ for the set of white cells in its row and column reaching outward until the nearest black cell in each direction (and (r, c) itself). The illumination constraint says some lamp must lie in this segment:

$$\sum_{(r',c') \in \mathcal{S}(r,c)} L_{r',c'} \;\geq\; 1 \quad \text{for every white cell.}$$

No two lamps see each other

For each maximal run of white cells in a row (a sequence of consecutive white cells bounded by black cells or the grid edge), the run holds at most one lamp:

$$\sum_{(r,c) \in \text{row run}} L_{r,c} \;\leq\; 1, \qquad \sum_{(r,c) \in \text{column run}} L_{r,c} \;\leq\; 1.$$

A row of length w with no black cell contributes a single inequality $\sum_c L_{r,c} \;\leq\; 1$; black cells split the row into independent runs each with its own inequality.

Numbered black cells

For each black cell carrying a clue k, the count of lamp neighbours in its four orthogonal cells equals k:

$$\sum_{(r',c')\ \text{orth. nbr}} L_{r',c'} \;=\; k.$$

A clue of 0 forbids any of the four neighbours from being a lamp; a clue of 4 forces all four to be lamps. Intermediate clues constrain the count of lamp positions among the at-most four candidates.

A solver in thirty-five lines

```
from ortools.sat.python import cp_model

def solve_akari(R, C, grid):
    """grid[r][c]: None (white), 'x' (black), or int."""
    m = cp_model.CpModel()
    is_w = lambda r, c: (0 <= r < R and 0 <= c < C
                         and grid[r][c] is None)
    is_b = lambda r, c: (0 <= r < R and 0 <= c < C
                         and grid[r][c] is not None)
    L = {(r, c): m.NewBoolVar("")
         for r in range(R) for c in range(C) if is_w(r, c)}

    def segment(r, c):
        seg = [(r, c)]
        for cc in range(c-1, -1, -1):
            if is_b(r, cc): break
            seg.append((r, cc))
        for cc in range(c+1, C):
            if is_b(r, cc): break
            seg.append((r, cc))
        for rr in range(r-1, -1, -1):
            if is_b(rr, c): break
            seg.append((rr, c))
        for rr in range(r+1, R):
            if is_b(rr, c): break
            seg.append((rr, c))
        return seg

    for (r, c) in L:
        m.Add(sum(L[s] for s in segment(r, c)) >= 1)

    def runs(line, is_blocker):
        out, buf = [], []
```

```
        for i, here in enumerate(line):
            if is_blocker(i):
                if buf: out.append(buf); buf = []
            else:
                buf.append(here)
        if buf: out.append(buf)
        return out

    for r in range(R):
        for run in runs(range(C), lambda c: is_b(r, c)):
            m.Add(sum(L[(r, c)] for c in run) <= 1)
    for c in range(C):
        for run in runs(range(R), lambda r: is_b(r, c)):
            m.Add(sum(L[(r, c)] for r in run) <= 1)

    NBR = [(-1,0),(1,0),(0,-1),(0,1)]
    for r in range(R):
        for c in range(C):
            if isinstance(grid[r][c], int):
                nb = [L[(r+dr, c+dc)] for dr, dc in NBR
                      if is_w(r+dr, c+dc)]
                m.Add(sum(nb) == grid[r][c])

    s = cp_model.CpSolver()
    s.Solve(m)
    return {pos for pos in L if s.Value(L[pos])}
```

The toy of Figure 11.1 solves uniquely in about 23 milliseconds.

✠

A hard instance

Figure 11.3 is puzzle 876 from the janko.at Akari archive, a 16 × 16 instance by Sakuhina at the site's top difficulty tier. The clue density is sparse, and the illumination constraint propagates only weakly through cells far from clues; the same solver code returns the unique solution (thirty-three lamps) in about 6 milliseconds.

✠

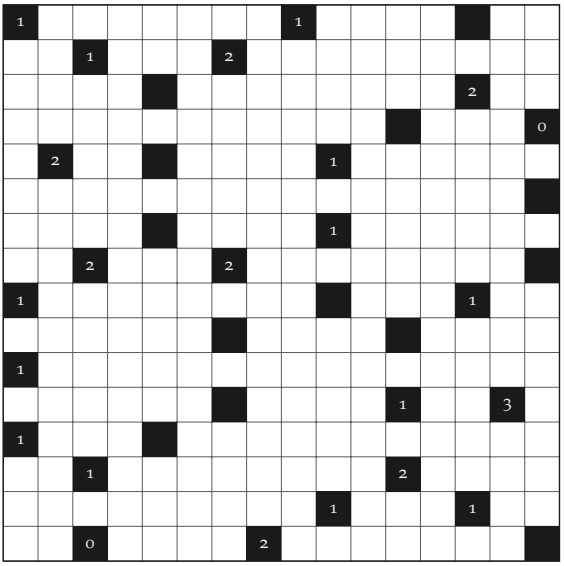

Figure 11.3: *Akari* 876 *from janko.at* (*Sakuhina*), 16 × 16, *top difficulty tier.*

Sources. Akari was published by Nikoli in 2001, and is also known as *Light Up* or *Bijutsukan*. The toy is puzzle 101 from the janko.at Akari archive (Warai Kamosika), and the hard instance is puzzle 876 (Sakuhina). McPhail and Boucher established the NP-completeness of Akari in 2007 in a Concordia University technical report; the proof reduces from 3-SAT via gadgets that encode each clause as a localised lighting puzzle.

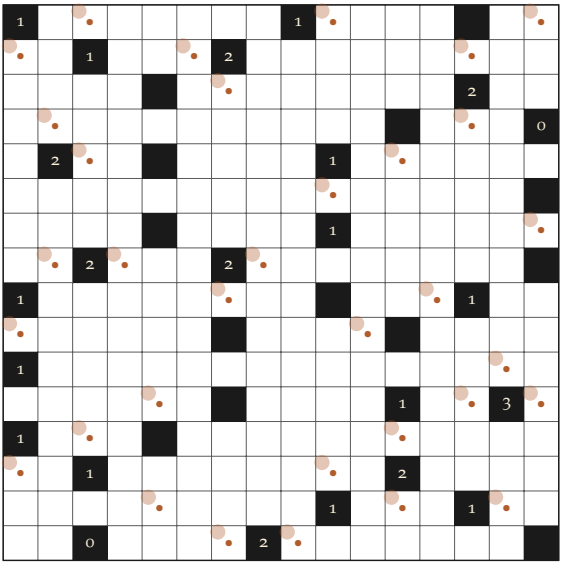

Figure 11.4: *The unique placement, recovered in about* 6 *ms.*

Chapter 12

Shikaku

✠

Shikaku is the puzzle of partitioning a rectangle into rectangles. The grid is rectangular; certain cells carry positive integers. The solver must cut the grid into a set of non-overlapping axis-aligned rectangles such that every cell is contained in exactly one rectangle, every rectangle contains exactly one numbered cell, and that number equals the rectangle's area in cells.

The puzzle was published by Nikoli in 1995, going by the names *Shikaku ni kire* ('divide into rectangles') in Japanese and *Sikaku* in transliteration. It has the cleanest combinatorial structure of any puzzle in this book: the solution is a partition, the constraint is a set-cover, and the entire problem fits naturally into an exact-cover framework. Knuth's Algorithm X (Dancing Links) was famously applied to Shikaku and its cousin Pentominoes; CP-SAT solves the same problem with the same speed and a tenth as much code.

✠

Rules and a small instance

A Shikaku puzzle is an $R \times C$ grid. A subset of cells carries a positive integer *clue* k. The solver's task is to partition the entire grid into a set of axis-aligned rectangles satisfying three conditions.

1. Every rectangle is axis-aligned and consists of one or more whole cells.
2. Every cell of the grid is contained in exactly one rectangle.
3. Every rectangle contains exactly one clue cell, and that clue's value equals the rectangle's area.

The clues' values must therefore sum to the total cell count $R \cdot C$ (a useful sanity check on any candidate puzzle). The puzzle is well-formed if there is exactly one such partition.

Figure 12.1 is puzzle 1 from the janko.at Sikaku archive, a 10×10 instance by Hirofumi Fujiwara with twenty-six clues.

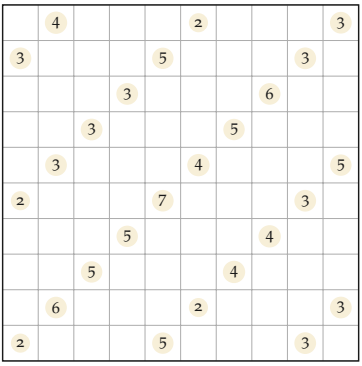

Figure 12.1: *Shikaku* 1 *from janko.at* (*Hirofumi Fujiwara*), 10×10. *Clue k in a cell demands that the rectangle containing it has area k.*

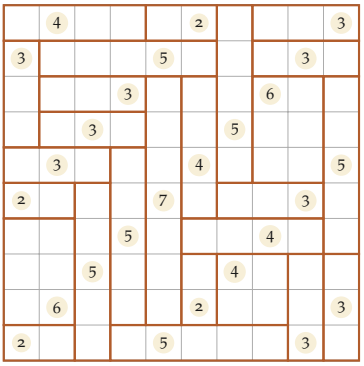

Figure 12.2: *The unique partition. Each copper rectangle encloses one clue and as many cells as the clue declares.*

✠

The programming model

The natural encoding is enumeration. For each clue cell (r, c) with area k, enumerate every axis-aligned rectangle of area k that contains (r, c) and contains no other clue; collect these as the candidate rectangles for that clue.

For an $R \times C$ grid, a clue of area k admits at most $\sigma_0(k) \cdot k$ rectangles centred on (r, c) (one per divisor pair (h, w) with $h \cdot w = k$, then a translation window of dimensions $h \times w$). For typical $k \le 25$ this is a handful per clue.

Introduce a Boolean $x_{c,R}$ for each clue c and each of its candidate rectangles R. The model:

1. For each clue c, $\sum_{R \in \mathrm{cand}(c)} x_{c,R} = 1$ (exactly one rectangle is chosen).
2. For each grid cell (r, c), $\sum_{(c,R):(r,c)\in R} x_{c,R} = 1$ (each cell is covered by exactly one chosen rectangle).

The first family ensures each clue is matched; the second ensures the chosen rectangles partition the grid. CP-SAT solves the system as a set-partitioning ILP.

A solver in thirty lines

```
from ortools.sat.python import cp_model

def candidate_rects(R, C, clue, area, all_clues):
    """Area-k rects in R x C with 'clue' alone inside."""
    out = []
    cr, cc = clue
    for h in range(1, R+1):
        if area % h: continue
        w = area // h
        if w > C: continue
        for r1 in range(max(0, cr-h+1),
                        min(cr, R-h)+1):
            for c1 in range(max(0, cc-w+1),
                            min(cc, C-w)+1):
                r2, c2 = r1+h-1, c1+w-1
                if all((p, q) == clue or not
                       (r1 <= p <= r2 and c1 <= q <= c2)
                       for (p, q) in all_clues):
                    out.append((r1, c1, r2, c2))
    return out

def solve_shikaku(R, C, clues):
    """clues: {(r, c): area}."""
    cands = {c: candidate_rects(R, C, c, k, clues)
             for c, k in clues.items()}
    m = cp_model.CpModel()
    x = {(c, rect): m.NewBoolVar("")
         for c in cands for rect in cands[c]}
    for c, rs in cands.items():
        m.Add(sum(x[(c, r)] for r in rs) == 1)
    for r in range(R):
        for c in range(C):
            cov = [x[k] for k in x
                   if k[1][0] <= r <= k[1][2]
                   and k[1][1] <= c <= k[1][3]]
            m.Add(sum(cov) == 1)
    s = cp_model.CpSolver()
    s.Solve(m)
    return {c: rect for (c, rect) in x
            if s.Value(x[(c, rect)])}
```

The toy of Figure 12.1 solves uniquely in about 25 milliseconds.

A hard instance

Figure 12.3 is puzzle 50 from the janko.at Sikaku archive, a 14 × 18 instance by Hirofumi Fujiwara graded at the site's top difficulty tier. With 68 clues over 252 cells, the candidate-rectangle space is dense enough that hand-solving takes considerable time; CP-SAT returns the unique partition in about 3 milliseconds.

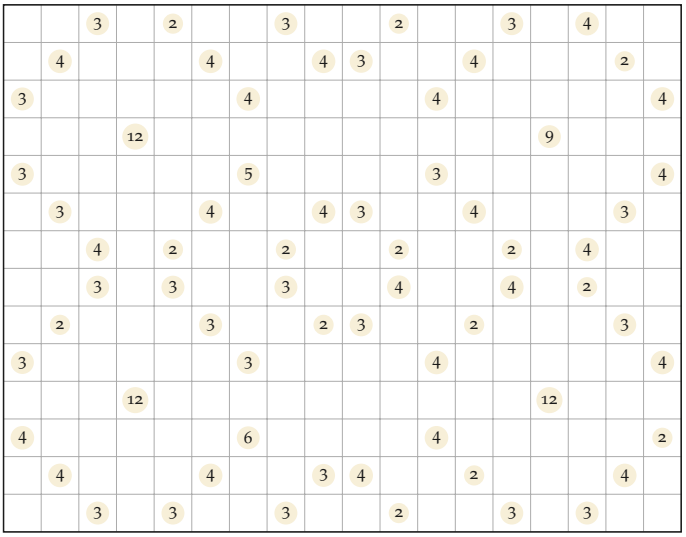

Figure 12.3: *Shikaku* 50 *from janko.at* (*Hirofumi Fujiwara*), 14×18, *top tier.*

Sources. Shikaku was published by Nikoli in 1995 in the magazine *Puzzle Communication Nikoli*; the Japanese name *Shikaku ni kire* translates as "divide into rectangles". The exact-cover formulation used here goes back to Knuth's *Dancing Links* (2000), which Knuth applied to the closely related pentomino-tiling problem. Toy and hard instances are

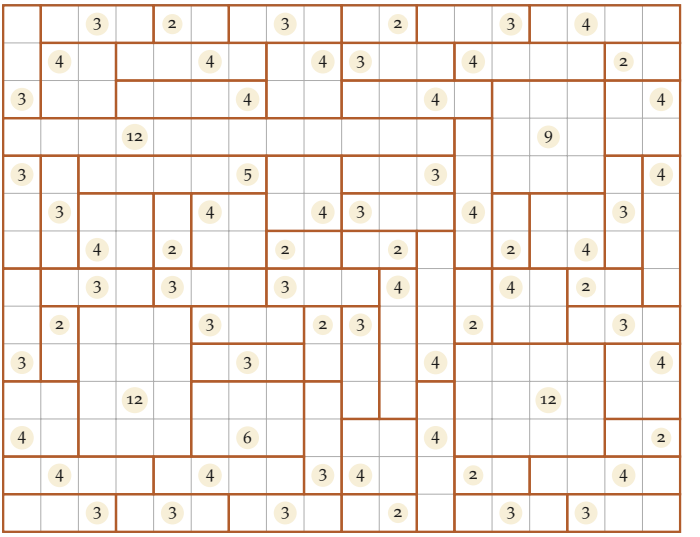

Figure 12.4: *The unique partition. Notice the four large* 12*-cell rectangles forming a near-symmetric backbone.*

puzzles 1 and 50 from the janko.at Sikaku archive (Hirofumi Fujiwara).

Chapter 13

Nurikabe

Nurikabe is the puzzle of islands in a sea. The grid is a rectangle of cells; some cells carry positive integers. The solver must shade some cells black and leave the rest white, subject to four rules. Every numbered cell is white and serves as the seed of a connected white *island* of exactly the declared size. Islands of different seeds do not touch each other along an edge. The black cells together form a single connected *sea*. Nowhere does the sea contain a 2×2 all-black square.

The puzzle was published by Nikoli in 1991, with the name *Nurikabe* drawn from a yokai of Japanese folklore: a white-walled spirit that obstructs travellers' paths. It joins the family of shading puzzles of which Heyawake, LITS, and Hitori are also members; among them it is the one whose constraints are most balanced between local (no 2×2, adjacency) and global (per-region connectivity, region size).

The chapter introduces the cleanest CP-SAT pattern for *multi-region connectivity*: one BFS-distance variable per cell, with the distance equal to zero at the region's root (a clue cell for islands, a designated cell for the sea), and a reified "has a same-region neighbour with $d - 1$" constraint elsewhere. The same pattern will return in subsequent shading puzzles.

✠

Rules and a small instance

A Nurikabe puzzle is an $R \times C$ grid. A subset of cells carry a positive integer *clue* indicating the size of the white island whose seed it is. The constraints:

1. Every cell is either black or white.
2. Each numbered cell is white and belongs to a connected white region of exactly the declared size.
3. Two distinct islands share no edge.
4. The set of black cells is connected.
5. No 2×2 block of cells is entirely black.

Figure 13.1 is puzzle 50 from the janko.at Nurikabe archive, an 8×8 instance by Ooya Tate Tadashi. The clues run from 1 (a singleton island) to 6.

						5	
				4			
3		2					
	6		2			3	
							1

Figure 13.1: *An 8×8 Nurikabe (Ooya Tate Tadashi, via janko.at). Each integer marks the seed and size of a white island; unshaded cells are to be filled in by the solver.*

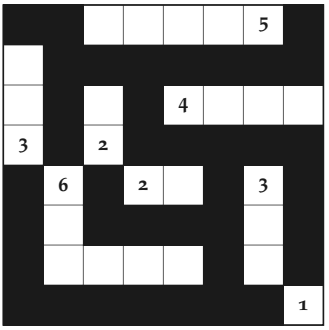

Figure 13.2: *The unique shading. Eight islands of total size* 26 *and one connected sea of* 38 *black cells; no* 2×2 *is entirely black.*

The programming model

Let $r_{r,c}$ denote the *region label* of cell (r, c), an integer between 0 and N where N is the number of clues. The label 0 marks the sea; labels $1, \dots, N$ mark the islands in some fixed enumeration of the clue cells. Let $s_{r,c} \in \{0, 1\}$ be the indicator $r_{r,c} = 0$ ("(r, c) is sea").

The *local* constraints are direct.

1. Each clue cell (c_i) has $r_{c_i} = i$ and $s_{c_i} = 0$.
2. For each island i, $|\{(r, c) : r_{r,c} = i\}| = k_i$ (the clue's value).
3. For each pair of orthogonally adjacent cells (p, q) with $r_p \neq r_q$, at least one of s_p, s_q is 1. Equivalently: two whites that are adjacent must share a label.
4. The total number of sea cells is $R \cdot C - \sum_i k_i$.
5. No 2×2 block is entirely sea: $s_{r,c} + s_{r+1,c} + s_{r,c+1} + s_{r+1,c+1} \leq 3$.

Connectivity by BFS-distance

Local constraints alone do not enforce that each region is connected. To do so, we attach a distance variable $d_{r,c} \in \{0, 1, \dots, R \cdot C - 1\}$ to every cell, intended to hold the BFS distance from the cell to the root of its region.

- For each clue c_i, $d_{c_i} = 0$ (the clue is the root of island i).
- Exactly one sea cell has $d = 0$ (the *sea root*); introduce a Boolean $\rho_{r,c}$ for each cell with $\rho_{r,c} = 1 \Rightarrow d_{r,c} = 0 \wedge s_{r,c} = 1$, and $\sum_{r,c} \rho_{r,c} = 1$.
- For every other cell, $d_{r,c} \geq 1$ and there exists an orthogonal neighbour (r', c') with $r_{r',c'} = r_{r,c}$ and $d_{r',c'} + 1 = d_{r,c}$.

The third bullet's existential is a reified disjunction over the four neighbours; CP-SAT handles it through `AddBoolOr` guarded by `OnlyEnforceIf`. The result is that the cells of any region form a tree rooted at the declared root, hence are all reachable from the root, hence connected.

A solver in seventy lines

```
from ortools.sat.python import cp_model

def solve_nurikabe(R, C, clues):
    """{(r,c): size} -> 2D grid of {0,1}."""
    cs = sorted(clues)
    N = len(cs)
    sea_total = R * C - sum(clues.values())
    m = cp_model.CpModel()
    reg = {(r, c): m.NewIntVar(0, N, "")
           for r in range(R) for c in range(C)}
    sea = {(r, c): m.NewBoolVar("")
           for r in range(R) for c in range(C)}
    for k in reg:
        m.Add(reg[k] == 0).OnlyEnforceIf(sea[k])
        m.Add(reg[k] >= 1).OnlyEnforceIf(sea[k].Not())
    for i, c in enumerate(cs, 1):
        m.Add(reg[c] == i)
    for i, c in enumerate(cs, 1):
        bs = []
        for k in reg:
            b = m.NewBoolVar("")
```

```
            m.Add(reg[k] == i).OnlyEnforceIf(b)
            m.Add(reg[k] != i).OnlyEnforceIf(b.Not())
            bs.append(b)
        m.Add(sum(bs) == clues[c])
    m.Add(sum(sea.values()) == sea_total)
    for r in range(R-1):
        for c in range(C-1):
            m.Add(sea[(r,c)] + sea[(r+1,c)]
                  + sea[(r,c+1)] + sea[(r+1,c+1)] <= 3)
    def same_region(p, q):
        b = m.NewBoolVar("")
        m.Add(reg[p] == reg[q]).OnlyEnforceIf(b)
        m.Add(reg[p] != reg[q]).OnlyEnforceIf(b.Not())
        return b
    for r in range(R):
        for c in range(C):
            for dr, dc in [(0,1),(1,0)]:
                nr, nc = r+dr, c+dc
                if not (0 <= nr < R and 0 <= nc < C):
                    continue
                m.AddBoolOr([same_region((r,c), (nr,nc)),
                             sea[(r,c)], sea[(nr,nc)]])
    d = {(r, c): m.NewIntVar(0, R*C-1, "")
         for r in range(R) for c in range(C)}
    for c in cs: m.Add(d[c] == 0)
    rho = {(r, c): m.NewBoolVar("")
           for r in range(R) for c in range(C)}
    for k in rho:
        m.Add(d[k] == 0).OnlyEnforceIf(rho[k])
        m.Add(sea[k] == 1).OnlyEnforceIf(rho[k])
    if sea_total > 0:
        m.Add(sum(rho.values()) == 1)
    def less_dist(p, q):
        b = m.NewBoolVar("")
        m.Add(d[q] + 1 == d[p]).OnlyEnforceIf(b)
        m.Add(d[q] + 1 != d[p]).OnlyEnforceIf(b.Not())
        return b
    cs_set = set(cs)
    for r in range(R):
        for c in range(C):
            if (r, c) in cs_set: continue
            sr = rho[(r, c)]
            m.Add(d[(r,c)] >= 1).OnlyEnforceIf(sr.Not())
            preds = []
            for dr, dc in [(-1,0),(1,0),(0,-1),(0,1)]:
                nr, nc = r+dr, c+dc
                if not (0 <= nr < R and 0 <= nc < C):
                    continue
```

```
                same = same_region((r,c), (nr,nc))
                less = less_dist((r,c), (nr,nc))
                pred = m.NewBoolVar("")
                pair = [same, less]
                neg = [same.Not(), less.Not()]
                m.AddBoolAnd(pair).OnlyEnforceIf(pred)
                m.AddBoolOr(neg).OnlyEnforceIf(pred.Not())
                preds.append(pred)
            m.AddBoolOr(preds).OnlyEnforceIf(sr.Not())
    s = cp_model.CpSolver()
    s.Solve(m)
    return [[s.Value(sea[(r, c)])
             for c in range(C)] for r in range(R)]
```

The toy of Figure 13.1 solves uniquely in about 75 milliseconds.

A hard instance

Figure 13.3 is puzzle 38 from the janko.at Nurikabe archive, the hardest 10×10 instance there, by Koyoppz. With fourteen clues totalling 48 white cells, the sea has 52 cells and threads sinuously around several neighbouring islands. CP-SAT returns the unique shading in about 180 milliseconds.

Sources. Nurikabe was published by Nikoli in 1991. The name comes from a Japanese folkloric *yokai*, a white-walled spirit that blocks travellers. The toy is puzzle 50 in the janko.at Nurikabe archive (Ooya Tate Tadashi); the hard instance is puzzle 38 (Koyoppz). McPhail established the NP-completeness of Nurikabe in 2003 via a reduction from 3-SAT (Carleton University technical report); the BFS-distance connectivity formulation used here is standard in CP-SAT modelling and goes back to the network-design literature.

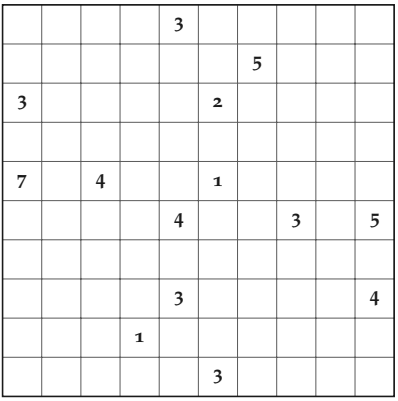

Figure 13.3: *Nurikabe* 38 *from janko.at* (*Koyoppz*), 10×10, *hardest at this size.*

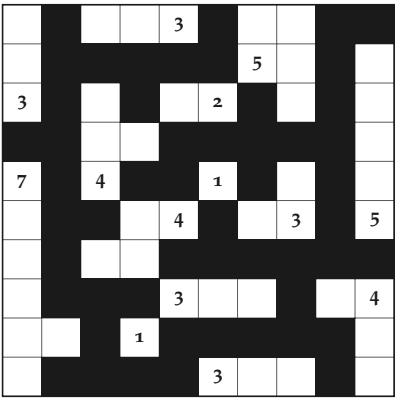

Figure 13.4: *The unique shading. The seven-cell island in column* 0 *snakes south, with the sea winding around it.*

Chapter 14

Masyu

Masyu draws a single closed loop through pearls. The grid is a rectangle of cells; some cells contain a pearl, either *white* (an open ring) or *black* (a filled disk). The solver must draw a single non-self-crossing closed loop on the grid that passes through the centre of every pearl, plus possibly some empty cells, subject to two pearl-specific rules.

- At a white pearl, the loop passes *straight through* (horizontally or vertically), but in at least one of the two cells immediately before or after, the loop turns.
- At a black pearl, the loop *turns* (a right-angle bend), and the loop passes straight through both of the cells immediately before and after the pearl.

The puzzle was published by Nikoli in 1999 in *Puzzle Communication Nikoli*; *masyu* translates to "evil influence" in Japanese, with the pearls (*shinju*) as the visual hook. The puzzle is also marketed in English as *White Pearls and Black Pearls* and occasionally as *Pearl Loop*. Among Nikoli loop puzzles it is arguably the most photogenic; the rules are local enough to admit elegant hand-solving and the resulting loops are visually striking.

The model reuses the `AddCircuit`-with-self-loops machinery from Slitherlink (Chapter 9), but here the loop runs on *cell centres* rather than lattice corners, so the nodes of the circuit graph are the cells themselves and the edges connect orthogonally adjacent cells. The pearl rules become

local constraints on each pearl cell about the combinations of incident edges and the turn behaviour of neighbouring cells.

Rules and a small instance

A Masyu puzzle is an $R \times C$ grid. Some cells carry a pearl. The solver must place a single closed loop on the grid such that:

1. Every cell is either on the loop (with two of its four edges to neighbouring cells in the loop) or off the loop (with zero loop-edges).
2. Every pearl cell is on the loop.
3. At a white pearl, the two loop edges are collinear (both horizontal or both vertical), and at least one of the two cells in that direction is itself a turn (its two edges are perpendicular).
4. At a black pearl, the two loop edges are perpendicular (a turn), and the cell on each of the two active sides is straight through.
5. The loop is a single non-self-crossing closed curve.

Figure 14.1 is puzzle 1 from the janko.at Masyu archive, a 6× 6 instance by Warai Kamosika with two black and two white pearls.

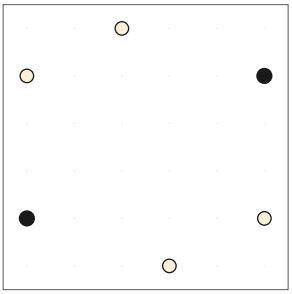

Figure 14.1: *Masyu 1 from janko.at (Warai Kamosika), 6×6. Open rings are white pearls; filled disks are black pearls.*

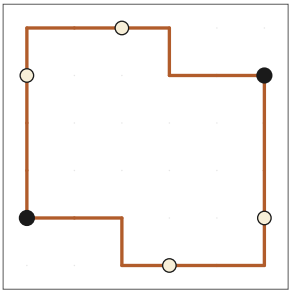

Figure 14.2: *The unique loop. Twenty edges; the two black pearls force right-angle turns with straight neighbours; the two white pearls' loop passes straight through with at least one turn alongside.*

The programming model

The variables are edge booleans on the cell-graph, plus, for each cell, indicator variables describing the local pattern of incident loop-edges.

- $h_{r,c}$ is the edge between cells (r, c) and $(r, c+1)$, for $0 \le r < R$ and $0 \le c < C-1$.
- $v_{r,c}$ is the edge between cells (r, c) and $(r+1, c)$, for $0 \le r < R-1$ and $0 \le c < C$.

For each cell define three indicator booleans:

- $\ell_{r,c}$, true iff the cell is *on the loop* (sum of incident edges $= 2$, otherwise $= 0$).
- $\sigma^h_{r,c}$, true iff the cell is straight horizontal: both E and W edges are in the loop, both N and S are not.
- $\sigma^v_{r,c}$, true iff straight vertical.
- $\tau_{r,c}$, true iff the cell turns (perpendicular edges).

If $\ell_{r,c}$ is true, exactly one of $\{\sigma^h_{r,c}, \sigma^v_{r,c}, \tau_{r,c}\}$ is true.

Pearl rules

For a white pearl at (r, c):

$$\sigma^h_{r,c} \vee \sigma^v_{r,c}.$$

If $\sigma^h_{r,c}$ is true, at least one of $\tau_{r,c-1}, \tau_{r,c+1}$ is true. If $\sigma^v_{r,c}$ is true, at least one of $\tau_{r-1,c}, \tau_{r+1,c}$ is true.

For a black pearl at (r, c): $\tau_{r,c}$. For each active incident edge, the cell on the other side of that edge is straight in the same direction. For example, if the E edge of (r, c) is in the loop, then $(r, c + 1)$ is straight horizontal.

Single-loop constraint

Same as Slitherlink: `AddCircuit` on a directed-arc graph where each undirected edge contributes two arcs (one per direction) summing to the edge variable, and each vertex (cell) has a self-loop arc enabled iff $\ell_{r,c}$ is false.

A solver in fifty lines

```
from ortools.sat.python import cp_model

def solve_masyu(R, C, pearls):
    """pearls: {(r, c): 'w' | 'b'}.  Returns edge dicts."""
    m = cp_model.CpModel()
    h = {(r, c): m.NewBoolVar("")
         for r in range(R) for c in range(C-1)}
    v = {(r, c): m.NewBoolVar("")
         for r in range(R-1) for c in range(C)}
```

```
    def edges(r, c):
        e = {}
        if c > 0:    e['W'] = h[(r, c-1)]
        if c < C-1:  e['E'] = h[(r, c)]
        if r > 0:    e['N'] = v[(r-1, c)]
        if r < R-1:  e['S'] = v[(r, c)]
        return e

    on_loop, turn, sh, sv = {}, {}, {}, {}
    for r in range(R):
        for c in range(C):
            e = edges(r, c)
            elist = list(e.values())
            ol = m.NewBoolVar("")
            m.Add(sum(elist) == 2).OnlyEnforceIf(ol)
            m.Add(sum(elist) == 0).OnlyEnforceIf(ol.Not())
            ns = sum(e[k] for k in ('N','S') if k in e)
            ew = sum(e[k] for k in ('E','W') if k in e)
            hh = m.NewBoolVar("")
            m.Add(ew == 2).OnlyEnforceIf(hh)
            m.Add(ew <= 1).OnlyEnforceIf(hh.Not())
            vv = m.NewBoolVar("")
            m.Add(ns == 2).OnlyEnforceIf(vv)
            m.Add(ns <= 1).OnlyEnforceIf(vv.Not())
            tt = m.NewBoolVar("")
            m.AddBoolAnd([ol, hh.Not(), vv.Not()]
                         ).OnlyEnforceIf(tt)
            m.AddBoolOr([ol.Not(), hh, vv]
                        ).OnlyEnforceIf(tt.Not())
            on_loop[(r,c)] = ol
            turn[(r,c)] = tt; sh[(r,c)] = hh; sv[(r,c)] = vv

    def need_turn(flag, neighbours):
        ts = [turn[n] for n in neighbours if n in turn]
        if len(ts) == 2:
            m.AddBoolOr(ts).OnlyEnforceIf(flag)
        elif ts:
            m.Add(ts[0] == 1).OnlyEnforceIf(flag)

    def black_side(edge, side_cell, side_dir):
        target = sh if side_dir == 'h' else sv
        m.Add(target[side_cell] == 1).OnlyEnforceIf(edge)

    for p, k in pearls.items():
        m.Add(on_loop[p] == 1)
        r, c = p
        if k == 'w':
            m.AddBoolOr([sh[p], sv[p]])
```

```
            need_turn(sh[p], [(r, c-1), (r, c+1)])
            need_turn(sv[p], [(r-1, c), (r+1, c)])
        else:
            m.Add(turn[p] == 1)
            e = edges(r, c)
            if 'N' in e: black_side(e['N'], (r-1, c), 'v')
            if 'S' in e: black_side(e['S'], (r+1, c), 'v')
            if 'E' in e: black_side(e['E'], (r, c+1), 'h')
            if 'W' in e: black_side(e['W'], (r, c-1), 'h')

    arcs = []
    vid = lambda r, c: r * C + c
    for (r, c), e in h.items():
        f = m.NewBoolVar(""); b = m.NewBoolVar("")
        m.Add(f + b == e)
        arcs.append((vid(r, c), vid(r, c+1), f))
        arcs.append((vid(r, c+1), vid(r, c), b))
    for (r, c), e in v.items():
        f = m.NewBoolVar(""); b = m.NewBoolVar("")
        m.Add(f + b == e)
        arcs.append((vid(r, c), vid(r+1, c), f))
        arcs.append((vid(r+1, c), vid(r, c), b))
    for p in on_loop:
        arcs.append((vid(*p), vid(*p), on_loop[p].Not()))
    m.AddCircuit(arcs)

    s = cp_model.CpSolver()
    s.Solve(m)
    H = {k: s.Value(v) for k, v in h.items()}
    V = {k: s.Value(v) for k, v in v.items()}
    return H, V
```

The toy of Figure 14.1 solves uniquely in about 6 milliseconds.

A hard instance

Figure 14.3 is puzzle 80 from the janko.at Masyu archive, a 16 × 16 instance by Grant Fikes (the prolific puzzle blogger at `mathgrant`). It carries 54 pearls, several of them in tightly clustered runs that force long sequences of black-pearl-and-straight cells. The same solver code returns the unique loop in about 65 milliseconds.

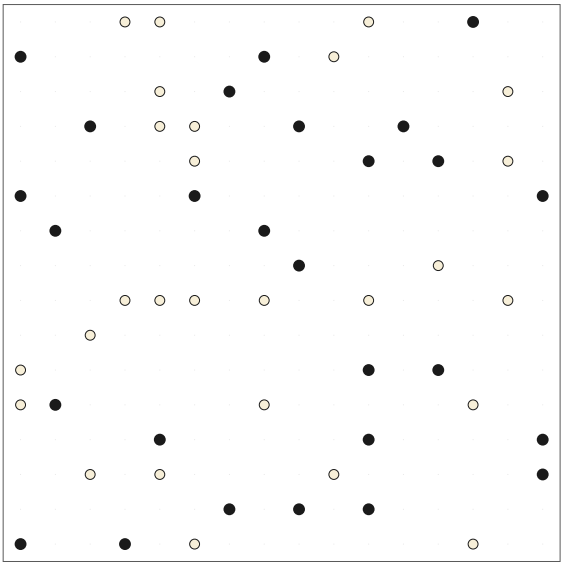

Figure 14.3: *Masyu* 80 *from janko.at* (*Grant Fikes*), 16 × 16 *with* 54 *pearls.*

Sources. Masyu was published by Nikoli in 1999. The toy is puzzle 1 from the janko.at Masyu archive (Warai Kamosika); the hard instance is puzzle 80 (Grant Fikes). Friedman established NP-completeness of Masyu in 2002 in a self-published note (`erich-friedman.github.io`). The CP

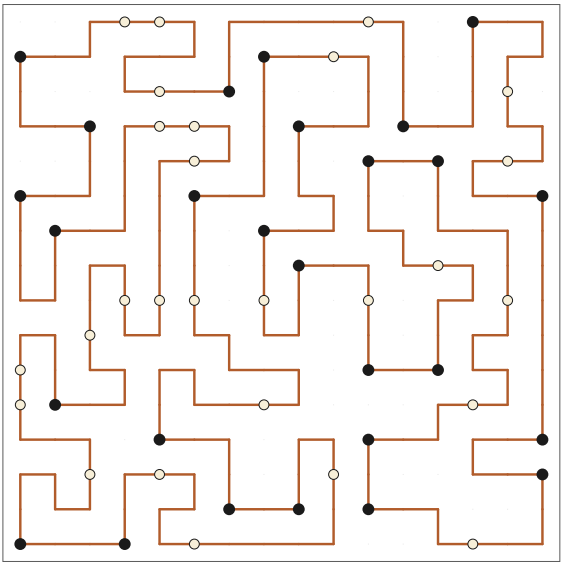

Figure 14.4: *The unique closed loop. Total* 222 *edges; the loop visits every pearl, turning at every black, sailing straight through every white.*

modelling pattern of `AddCircuit`-with-self-loops is the same as in Chapter 9.

Chapter 15

Hitori

Hitori is the puzzle of hiding duplicates. The grid is a square of digits; the solver must shade some cells black and leave the rest unshaded, subject to three rules. Every row and every column must contain each digit at most once among its *unshaded* cells. No two shaded cells may touch along an edge. And every unshaded cell must reach every other unshaded cell through a chain of orthogonally adjacent unshaded cells, i.e. the unshaded region must be connected.

The puzzle was published by Nikoli in 1990; the Japanese name, *Hitori ni shite kure*, literally "leave me alone," is a pun on the solving procedure of eliminating duplicates by shading. The puzzle has the unusual property, among Nikoli shadings, of operating on a *complement* region: the unshaded cells carry the content and must be connected, while the shaded cells are merely the deletion pattern.

Each of the three rules contributes a distinct CP-SAT constraint register. The row and column uniqueness is a "pairwise conflict" family: for each pair of cells sharing a row or column and carrying the same digit, at least one must be shaded. The adjacency rule is a pairwise ≤ 1 constraint on neighbouring cells. The connectivity rule is, as in Nurikabe, a distance-from-root flow but applied to the *complement* of the shading.

✠

Rules and a small instance

A Hitori puzzle is an $R \times C$ grid (usually with $R = C$, and typically $R \in \{5, \dots, 17\}$). Each cell carries a positive integer. The solver chooses a subset of cells to shade subject to three rules.

1. In every row, each digit appears at most once among the unshaded cells; likewise in every column.
2. No two shaded cells are orthogonally adjacent.
3. The unshaded cells form a single orthogonally connected region.

Figure 15.1 is puzzle 4 from the janko.at Hitori archive, an 8×8 instance by Otto Janko.

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

Figure 15.1: *Hitori 4 from janko.at (Otto Janko), 8 × 8. Each row and column already carries some repeated digits; the solver's task is to shade the redundant copies.*

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

Figure 15.2: *The unique shading. Black cells are orthogonally non-adjacent; unshaded cells are connected; every row and column has no unshaded digit repeated.*

The programming model

For each cell (r, c) define a Boolean $s_{r,c}$, with $s_{r,c} = 1$ meaning shaded.

Pairwise uniqueness on unshaded cells

For each row r and each pair of distinct columns $c_1 < c_2$ with grid digits equal, at least one of the two cells must be shaded:

$$s_{r,c_1} + s_{r,c_2} \geq 1 \quad \text{whenever } g_{r,c_1} = g_{r,c_2}.$$

The analogous constraint holds for every column and every pair of rows with equal column digits.

No adjacent black cells

For each pair of orthogonally adjacent cells (p, q),

$$s_p + s_q \leq 1.$$

Connectivity of the unshaded region

Attach a distance variable $d_{r,c} \in \{0, 1, \dots, RC - 1\}$ to every cell, and a Boolean $\rho_{r,c}$ marking a designated *unshaded root*. The root's d is 0 and ρ implies the cell is unshaded. Every other unshaded cell must have $d \geq 1$ and an orthogonally adjacent unshaded neighbour with $d - 1$.

Exactly one ρ is set: $\sum_{r,c} \rho_{r,c} = 1$.

A solver in fifty lines

```
from ortools.sat.python import cp_model

def solve_hitori(R, C, grid):
    """grid[r][c] = digit. Returns shaded {0, 1} grid."""
    m = cp_model.CpModel()
    s = {(r, c): m.NewBoolVar("")
         for r in range(R) for c in range(C)}
    for r in range(R):
        for c1 in range(C):
            for c2 in range(c1+1, C):
                if grid[r][c1] == grid[r][c2]:
                    m.Add(s[(r, c1)] + s[(r, c2)] >= 1)
    for c in range(C):
        for r1 in range(R):
            for r2 in range(r1+1, R):
                if grid[r1][c] == grid[r2][c]:
                    m.Add(s[(r1, c)] + s[(r2, c)] >= 1)
    for r in range(R):
        for c in range(C):
            for dr, dc in [(0,1),(1,0)]:
                nr, nc = r+dr, c+dc
                if 0 <= nr < R and 0 <= nc < C:
                    m.Add(s[(r,c)] + s[(nr,nc)] <= 1)

    d = {(r, c): m.NewIntVar(0, R*C-1, "")
         for r in range(R) for c in range(C)}
    rho = {(r, c): m.NewBoolVar("")
           for r in range(R) for c in range(C)}
    un = {k: m.NewBoolVar("") for k in s}
    for k in s:
        m.Add(s[k] == 0).OnlyEnforceIf(un[k])
        m.Add(s[k] == 1).OnlyEnforceIf(un[k].Not())
        m.Add(d[k] == 0).OnlyEnforceIf(rho[k])
        m.Add(un[k] == 1).OnlyEnforceIf(rho[k])
    m.Add(sum(rho.values()) == 1)
```

```
    def less(p, q):
        b = m.NewBoolVar("")
        m.Add(d[q] + 1 == d[p]).OnlyEnforceIf(b)
        m.Add(d[q] + 1 != d[p]).OnlyEnforceIf(b.Not())
        return b

    for r in range(R):
        for c in range(C):
            cell = (r, c); sr = rho[cell]
            m.Add(d[cell] >= 1).OnlyEnforceIf([un[cell],
                                               sr.Not()])
            preds = []
            for dr, dc in [(-1,0),(1,0),(0,-1),(0,1)]:
                nr, nc = r+dr, c+dc
                if not (0 <= nr < R and 0 <= nc < C):
                    continue
                nb = (nr, nc)
                lb = less(cell, nb)
                p = m.NewBoolVar("")
                m.AddBoolAnd([un[nb], lb]).OnlyEnforceIf(p)
                m.AddBoolOr([un[nb].Not(), lb.Not()]
                            ).OnlyEnforceIf(p.Not())
                preds.append(p)
            if preds:
                m.AddBoolOr(preds).OnlyEnforceIf(
                    [un[cell], sr.Not()])

    sol = cp_model.CpSolver()
    sol.Solve(m)
    return [[sol.Value(s[(r, c)])
             for c in range(C)] for r in range(R)]
```

The toy of Figure 15.1 solves uniquely in about 70 milliseconds.

A hard instance

Figure 15.3 is puzzle 78 from the janko.at Hitori archive, a 17 × 17 instance by Koyoppz, the only puzzle in the archive at the top-tier difficulty "schwer". The grid carries 289 cells filled with digits 1 to 15, many of them with multiple repetitions; the solver's task is to find the unique shading pattern that satisfies all three rules. CP-SAT returns the result in about 1.4 seconds.

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

Figure 15.3: *Hitori 78 from janko.at (Koyoppz), 17 × 17, the archive's single "schwer" instance.*

Sources. Hitori was first published by Nikoli in 1990. The toy is puzzle 4 from the janko.at Hitori archive (Otto Janko);

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

Figure 15.4: *The unique shading. Shaded cells are non-adjacent, the unshaded region is connected, and every row and column has unique unshaded digits.*

the hard instance is puzzle 78 (Koyoppz). Hearn and Demaine (2005) proved the general Hitori decision problem NP-complete via a reduction from planar 3-SAT in *Games, Puzzles and Computation* (AK Peters).

Chapter 16

Fillomino

Fillomino is the puzzle of self-describing polyominoes. The grid is rectangular; some cells carry positive integers. The solver must fill every empty cell with a positive integer such that, when cells are partitioned into the maximal connected regions of cells sharing a common value, every region of value k contains exactly k cells. Two regions of the same value are forbidden from sharing an edge; equivalently, two adjacent cells with equal values must lie in the same region.

The puzzle was published by Nikoli in 1994, in the magazine *Puzzle Communication Nikoli*; the name *fillomino* is a contraction of "fill in polyominoes". It generalises Shikaku of Chapter 12 from the rectangle case to arbitrary polyomino shapes, and at the same time relaxes the requirement that every region contain a clue: *anonymous polyominoes*, with no clue cell anywhere inside, are permitted whenever the constraints force them.

The presence of anonymous polyominoes makes polyomino enumeration intractable: anonymous regions can be of any size up to the grid area, and the number of polyomino shapes grows fast with size (about 4655 free decominoes, nine hundred thousand free 14-ominoes, and so on). The cleanest CP-SAT model abandons enumeration entirely and instead runs a parent-pointer forest over the grid: each cell carries a parent direction or a "root" marker, and a recursive subtree-size variable aggregates the count from

leaves up to roots, with each root's v constrained to equal its subtree size.

Rules and a small instance

A Fillomino puzzle is an $R \times C$ grid. A subset of cells carry positive integer clues. The solver assigns a positive integer $v_{r,c}$ to every cell, satisfying:

1. For each clue cell (r, c) with clue k, $v_{r,c} = k$.
2. For each cell (r, c), the maximal connected (orthogonally adjacent) region of cells with value $v_{r,c}$ that contains (r, c) has size exactly $v_{r,c}$.

The two-rule formulation is concise; equivalent restatements spell out the no-touching rule, 'any two adjacent cells with the same value are in the same region', which prevents two distinct same-value polyominoes from sharing an edge.

Figure 16.1 is puzzle 5 from the janko.at Fillomino archive, an 8×8 instance.

		2			1		
5							5
2			3	2	1		
		3	4				
				5	1		
		3	2	1			5
1							4
		3			1		

Figure 16.1: *An 8×8 Fillomino. Clue cells carry bold integers; empty cells must be filled.*

5	2	2	4	4	1	5	5
5	5	4	4	2	5	5	5
2	5	3	3	2	1	2	2
2	5	3	4	5	5	5	5
1	4	4	4	5	1	2	2
3	3	3	2	1	5	5	5
1	4	4	2	5	5	4	4
4	4	3	3	3	1	4	4

Figure 16.2: *The unique completion. Bold digits are the givens; copper digits are filled by the solver. Heavy borders separate distinct polyominoes; cells in the same polyomino share a value.*

The programming model

Let $v_{r,c} \in \{1, \dots, V_{\max}\}$ denote the value at cell (r, c). The clue cells fix some of these. The connectivity and size constraints are imposed via a parent-pointer forest.

Parent-pointer forest

For each cell (r, c) define an integer $\pi_{r,c} \in \{0, 1, 2, 3, 4\}$ encoding the parent direction: 0 for "root," 1–4 for north, south, east, west respectively.

- $\pi_{r,c} = 0 \iff$ cell is the root of its polyomino.
- If $\pi_{r,c} = d \in \{1, 2, 3, 4\}$, the neighbour in direction d exists and shares the value $v_{r,c}$.

To prevent cycles in the parent graph, attach a distance-from- root variable $\delta_{r,c} \in \{0, \dots, V_{\max} - 1\}$ with $\delta_{\text{root}} = 0$ and $\delta_{\text{cell}} = \delta_{\text{parent}} + 1$. Bounded distances guarantee acyclicity.

Component identity for adjacent same-value cells

To force 'two same-value adjacent cells must be in the same polyomino', attach a $\rho_{r,c} \in \{0, \dots, R \cdot C - 1\}$ acting as a component-identifier. Roots set ρ to their own cell index; non-roots inherit ρ from their parent. For adjacent cells with equal v, we then demand $\rho_p = \rho_q$.

Subtree size = value

For each cell, $\sigma_{r,c}$ is its subtree size:

$$\sigma_{r,c} \;=\; 1 + \sum_{\text{neighbour } q \text{ pointing to } (r,c)} \sigma_q .$$

For each root cell, $\sigma_{r,c} = v_{r,c}$.

The σ recursion is a CP-SAT linear constraint per cell, with reified contributions per neighbour: each neighbour q contributes σ_q if π_q points back to the cell, else 0.

A solver in seventy lines

```
from ortools.sat.python import cp_model

DT   = {1:(-1,0), 2:(1,0), 3:(0,1), 4:(0,-1)}
DREV = {1:2, 2:1, 3:4, 4:3}

def solve_fillomino(R, C, clues, V):
    """{(r, c): value} -> R x C grid; V = max size."""
    m = cp_model.CpModel()
    cells = [(r, c) for r in range(R) for c in range(C)]
    v   = {k: m.NewIntVar(1, V, "") for k in cells}
    par = {k: m.NewIntVar(0, 4, "") for k in cells}
    rt  = {k: m.NewBoolVar("")      for k in cells}
    d   = {k: m.NewIntVar(0, V-1, "") for k in cells}
    rid = {k: m.NewIntVar(0, R*C-1, "") for k in cells}
    sub = {k: m.NewIntVar(1, V, "") for k in cells}

    for k, val in clues.items():
        m.Add(v[k] == val)
    for k in cells:
        m.Add(par[k] == 0).OnlyEnforceIf(rt[k])
        m.Add(par[k] != 0).OnlyEnforceIf(rt[k].Not())
        m.Add(d[k] == 0).OnlyEnforceIf(rt[k])
        m.Add(d[k] >= 1).OnlyEnforceIf(rt[k].Not())

    def eq(x, val):
        b = m.NewBoolVar("")
```

```
        m.Add(x == val).OnlyEnforceIf(b)
        m.Add(x != val).OnlyEnforceIf(b.Not())
        return b

    def in_grid(r, c):
        return 0 <= r < R and 0 <= c < C

    for r, c in cells:
        cell = (r, c)
        m.Add(rid[cell] == r*C + c).OnlyEnforceIf(rt[cell])
        for did, (dr, dc) in DT.items():
            nr, nc = r+dr, c+dc
            if not in_grid(nr, nc):
                m.Add(par[cell] != did)
                continue
            nb = (nr, nc)
            f = eq(par[cell], did)
            m.Add(v[cell] == v[nb]).OnlyEnforceIf(f)
            m.Add(d[nb] + 1 == d[cell]).OnlyEnforceIf(f)
            m.Add(rid[cell] == rid[nb]).OnlyEnforceIf(f)

    for r, c in cells:
        for dr, dc in [(0, 1), (1, 0)]:
            nr, nc = r+dr, c+dc
            if not in_grid(nr, nc): continue
            a, b2 = (r, c), (nr, nc)
            same = m.NewBoolVar("")
            m.Add(v[a] == v[b2]).OnlyEnforceIf(same)
            m.Add(v[a] != v[b2]).OnlyEnforceIf(same.Not())
            m.Add(rid[a] == rid[b2]).OnlyEnforceIf(same)

    for r, c in cells:
        contribs = []
        for did, (dr, dc) in DT.items():
            nr, nc = r+dr, c+dc
            if not in_grid(nr, nc): continue
            ch = eq(par[(nr, nc)], DREV[did])
            cont = m.NewIntVar(0, V, "")
            m.Add(cont == sub[(nr, nc)]).OnlyEnforceIf(ch)
            m.Add(cont == 0).OnlyEnforceIf(ch.Not())
            contribs.append(cont)
        m.Add(sub[(r, c)] == 1 + sum(contribs))
    for k in cells:
        m.Add(v[k] == sub[k]).OnlyEnforceIf(rt[k])
    s = cp_model.CpSolver()
    s.Solve(m)
    return [[s.Value(v[(r,c)])
             for c in range(C)] for r in range(R)]
```

The toy of Figure 16.1 solves uniquely in about 80 milliseconds.

A hard instance

Figure 16.3 is puzzle 91 from the janko.at Fillomino archive, a 10 × 10 instance by Grant Fikes. Its solution contains a fourteen-cell anonymous polyomino that no clue identifies; the solver must infer both its existence and its precise shape. CP-SAT returns the unique solution in about four seconds.

	5							2	
5			6	6	4	6			2
	4							3	
		4	4			5	3		
2									1
5									4
		4	4			2	4		
	3							4	
1			2	5	6	2			3
	3							4	

Figure 16.3: *Fillomino* 91 *from janko.at* (*Grant Fikes*), 10×10. *The clues hint at small polyominoes; the global structure conceals a fourteen-cell anonymous region.*

Sources. Fillomino was first published by Nikoli in 1994. The toy is puzzle 5 from the janko.at Fillomino archive (Hisao Fukushima, via Keio University); the hard instance is puzzle 91 (Grant Fikes). The polyomino-count sequence 1, 1, 2, 5, 12, 35, 108, … is OEIS A000105; explosive growth makes anonymous-polyomino enumeration infeasible above

5	5	6	4	4	4	6	2	2	1
5	6	6	6	6	4	6	6	6	2
5	4	6	5	5	6	6	3	3	2
5	4	4	4	5	5	5	3	14	14
2	2	5	5	14	14	14	14	14	1
5	5	5	4	14	6	2	4	4	4
3	4	4	4	14	6	2	4	1	3
3	3	14	14	14	6	6	6	4	3
1	14	14	2	5	6	2	2	4	3
3	3	3	2	5	5	5	5	4	4

Figure 16.4: *The unique completion. Heavy borders separate distinct polyominoes; the central* 14*-region snakes diagonally across the grid with no clue inside.*

size $\sim$ 8, motivating the parent-pointer forest model used here. NP-completeness of Fillomino was established by Yato and Seta in 2003 (*Information and Computation* 185, 159–179).

Chapter 17

Heyawake

Heyawake is the puzzle of dividing rooms. The grid is partitioned into rectangular *rooms* drawn with thick borders; some rooms carry a clue digit declaring the number of black cells the room must contain. The solver shades a subset of cells to satisfy four rules: the clue counts, no two black cells orthogonally adjacent, every white cell connected to every other white cell by a path of orthogonally adjacent white cells, and no straight horizontal or vertical line of consecutive white cells passes through more than two rooms.

The puzzle was published by Nikoli in 1992 in the magazine *Puzzle Communication Nikoli*; *heyawake* translates literally as "rooms divided", combining *heya* ("room") and *wake* ("divided"). Among Nikoli shading puzzles it sits between Nurikabe and LITS in flavour: it shares Nurikabe's connectivity register but operates on the *complement*, and like LITS it imposes a structural constraint linked to the room partition.

The white-line-spans-at-most-two-rooms rule is the puzzle's signature constraint. It propagates information along entire row and column scans, often forcing black cells far from any clue; hand-solvers learn to spot triples of consecutive rooms that must therefore contain at least one shaded cell.

✠

Rules and a small instance

A Heyawake puzzle is an $R \times C$ grid partitioned into *rooms*. A subset of rooms carry a positive integer clue. The solver shades a subset of cells subject to:

1. For every clued room, the count of shaded cells in the room equals the clue.
2. No two shaded cells share an edge.
3. All unshaded cells form one orthogonally connected region.
4. No row segment or column segment of consecutive unshaded cells passes through three or more rooms.

Figure 17.1 is puzzle 31 from the janko.at Heyawake archive, a 6×6 instance by Abe Hajime, with seventeen rooms and no clue cells; the solution is forced by the geometry of the partition alone.

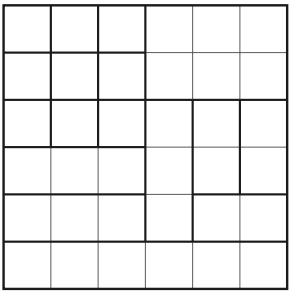

Figure 17.1: *A* 6×6 *Heyawake (Abe Hajime, via janko.at). No clue cells; the partition into seventeen rooms alone determines the unique shading.*

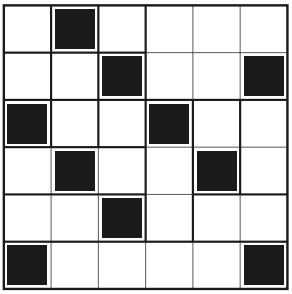

Figure 17.2: *The unique shading. Eleven black cells, no two orthogonally adjacent, and every row and column run of whites stays within at most two rooms.*

The programming model

For each cell (r, c) define a Boolean $b_{r,c}$ for shaded. The four rules become:

- *Room counts.* For each clued room with cells C_R and clue k_R: $\sum_{(r,c)\in C_R} b_{r,c} = k_R$.
- *Black non-adjacency.* For each pair of orthogonally adjacent cells (p, q): $b_p + b_q \le 1$.
- *White connectivity.* Same BFS-distance pattern as Nurikabe (Chapter 13), applied to the unshaded cells.
- *Three-room runs.* For each row r and starting column s, find the smallest e such that cells $r, c \in [s, e]$ touch at least three distinct rooms. Then $\sum_{c=s}^{e} b_{r,c} \ge 1$. Same for columns.

The three-room-run constraint deserves a moment. For a fixed row r, walk through the columns $0, 1, \ldots, C-1$. As you walk, maintain the set of distinct room IDs seen since position s; when this set first reaches size 3 at some position e, the window $[s, e]$ is a *minimal* run that touches three rooms. To keep that run from being entirely white, at least one of

$b_{r,s}, \dots, b_{r,e}$ must be 1. Repeating for every s generates the full set of run-three-room constraints in $O(R \cdot C)$ time. Symmetric processing on columns completes the picture.

A solver in fifty lines

```
from ortools.sat.python import cp_model

def solve_heyawake(R, C, room_grid, rooms, clues):
    """rooms: room_id -> cell list.
       clues: cell -> black count of its room."""
    m = cp_model.CpModel()
    cells = [(r, c) for r in range(R) for c in range(C)]
    b = {k: m.NewBoolVar("") for k in cells}
    cpr = {room_grid[r][c]: v for (r, c), v in clues.items()}
    for rid, cs in rooms.items():
        if rid in cpr:
            m.Add(sum(b[c] for c in cs) == cpr[rid])
    for r, c in cells:
        for dr, dc in [(0, 1), (1, 0)]:
            nr, nc = r+dr, c+dc
            if 0 <= nr < R and 0 <= nc < C:
                m.Add(b[(r, c)] + b[(nr, nc)] <= 1)
    for r in range(R):
        for s in range(C):
            seen = set()
            for e in range(s, C):
                seen.add(room_grid[r][e])
                if len(seen) >= 3:
                    m.Add(sum(b[(r, k)]
                              for k in range(s, e+1)) >= 1)
                    break
    for c in range(C):
        for s in range(R):
            seen = set()
            for e in range(s, R):
                seen.add(room_grid[e][c])
                if len(seen) >= 3:
                    m.Add(sum(b[(k, c)]
                              for k in range(s, e+1)) >= 1)
                    break
    # White connectivity (Nurikabe pattern)
    d   = {k: m.NewIntVar(0, R*C-1, "") for k in cells}
    rho = {k: m.NewBoolVar("") for k in cells}
    w   = {k: m.NewBoolVar("") for k in cells}
    for k in cells:
        m.Add(b[k] == 0).OnlyEnforceIf(w[k])
```

```
            m.Add(b[k] == 1).OnlyEnforceIf(w[k].Not())
            m.Add(d[k] == 0).OnlyEnforceIf(rho[k])
            m.Add(w[k] == 1).OnlyEnforceIf(rho[k])
        m.Add(sum(rho.values()) == 1)
        def less_dist(p, q):
            bb = m.NewBoolVar("")
            m.Add(d[q] + 1 == d[p]).OnlyEnforceIf(bb)
            m.Add(d[q] + 1 != d[p]).OnlyEnforceIf(bb.Not())
            return bb
        for r, c in cells:
            cell = (r, c); sr = rho[cell]
            m.Add(d[cell] >= 1).OnlyEnforceIf(
                [w[cell], sr.Not()])
            preds = []
            for dr, dc in [(-1,0),(1,0),(0,-1),(0,1)]:
                nr, nc = r+dr, c+dc
                if not (0 <= nr < R and 0 <= nc < C):
                    continue
                nb = (nr, nc)
                less = less_dist(cell, nb)
                p = m.NewBoolVar("")
                m.AddBoolAnd([w[nb], less]).OnlyEnforceIf(p)
                m.AddBoolOr([w[nb].Not(), less.Not()]
                            ).OnlyEnforceIf(p.Not())
                preds.append(p)
            if preds:
                m.AddBoolOr(preds).OnlyEnforceIf(
                    [w[cell], sr.Not()])
        s = cp_model.CpSolver()
        s.Solve(m)
        return [[s.Value(b[(r, c)])
                 for c in range(C)] for r in range(R)]
```

The toy of Figure 17.1 solves uniquely in about 20 milliseconds.

✠

A hard instance

Figure 17.3 is puzzle 297 from the janko.at Heyawake archive, a 10×18 instance by Palmer Mebane (of *mellowmelon*), graded at the site's level 7 difficulty. Sparse clues sit on a partition of 36 rooms; the three-room constraint does most

of the lifting. CP-SAT returns the unique shading in about 340 milliseconds.

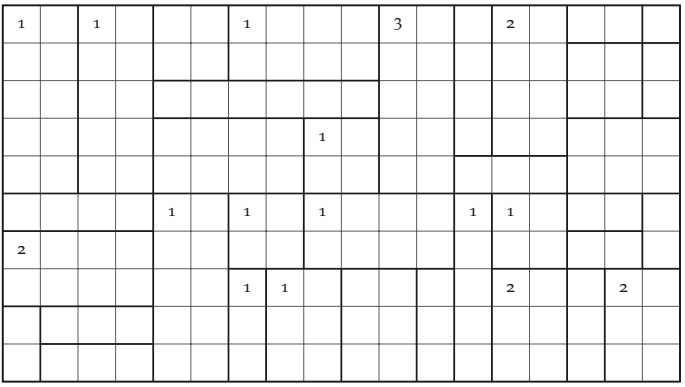

Figure 17.3: *Heyawake* 297 *from janko.at* (*Palmer Mebane*), 10×18 *with* 36 *rooms and* 13 *clues.*

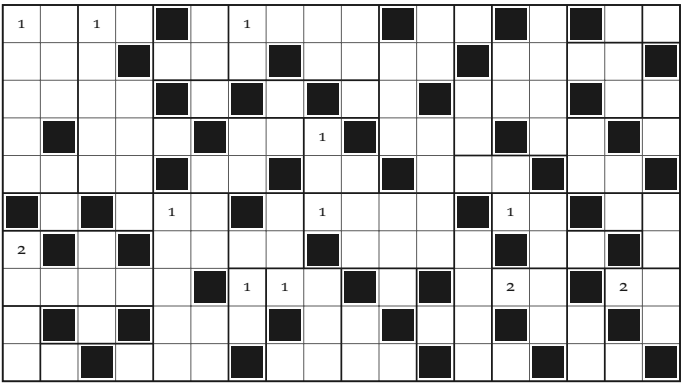

Figure 17.4: *The unique shading, recovered in about* 340 *ms. Black cells are non-adjacent, white cells are connected, and every row and column white-run stays inside at most two rooms.*

Sources. Heyawake was first published by Nikoli in 1992. The toy is puzzle 31 from the janko.at Heyawake archive (Abe

Hajime); the hard instance is puzzle 297 (Palmer Mebane). Holzer and Ruepp (2007) proved Heyawake NP-complete via a reduction from Hamiltonian Cycle in cubic planar graphs; the connectivity constraint and the three-room-run constraint together carry the encoding.

Chapter 18

Shakashaka

Shakashaka is the puzzle of triangular halves. The grid is a rectangle of black and white cells; some black cells carry a clue digit in $\{0, 1, 2, 3, 4\}$. The solver fills a subset of white cells with an isosceles right triangle occupying half the cell (four orientations are available) so that two conditions hold: each clued black cell has exactly as many triangle-filled neighbours as its clue declares, and every connected region of remaining white area forms an axis-aligned rectangle or a 45°-rotated rectangle (a diamond). No L-shaped voids, no staircases, no rings are permitted.

The puzzle was published by Nikoli in the magazine *Puzzle Communication Nikoli* in 2008, invented by Guten; the name is onomatopoeic, evoking the clattering sound of the triangles snapping into place. It is also known in translation as *Proof of Quilt*. Among Nikoli shading puzzles it is unusual: the state per cell is not binary but has five options (four triangle orientations plus empty), and the global constraint is geometric rather than connective.

The rectangle-or-diamond rule is the puzzle's signature. It is a global shape constraint on an a priori unknown partition of the white cells, and it has real teeth: Demaine, Okamoto, Uehara and Uno proved Shakashaka NP-complete by reducing planar 3-SAT to it, and showed that counting solutions is #P-complete. They also supplied the integer-programming formulation reproduced in this chapter, in which five constraint families enforce the rule geometrically.

✠

Rules and a small instance

A Shakashaka puzzle is a $R \times C$ grid of cells, each either *black* (optionally labelled with a clue in $\{0, 1, 2, 3, 4\}$) or *white*. A solution assigns to each white cell one of five states: *empty*, or filled by a triangle with its black right-angle at its NW, NE, SW or SE corner. A valid solution satisfies:

1. For every clued black cell with clue k, the number of triangle-filled cells among its four orthogonal neighbours whose triangle presents a full black edge to the clue is exactly k.
2. Every connected region of white non-triangle area is either an axis-aligned rectangle or a 45°-rotated rectangle (diamond). No other shape is permitted.

Figure 18.1 is a toy 5×5 Shakashaka in which three black cells carry clues; the solution is unique.

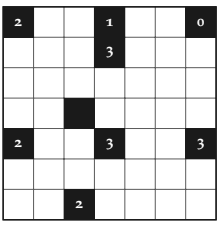

Figure 18.1: *A* 5×5 *Shakashaka puzzle with three clues.*

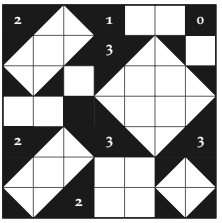

Figure 18.2: *The unique solution. Each clue counts its triangle-facing neighbours; the remaining white cells partition into an axis-aligned rectangle and two diamonds.*

The programming model

For every white cell (r, c) define five Boolean variables,

$$x_{r,c,\emptyset},\ x_{r,c,\mathrm{NW}},\ x_{r,c,\mathrm{NE}},\ x_{r,c,\mathrm{SW}},\ x_{r,c,\mathrm{SE}},$$

one per state. Black cells and cells outside the grid are not assigned variables; whenever a constraint references such a cell, every term is treated as the constant 0. This is the *Demaine boundary convention*, and it is load-bearing: it lets every constraint be written uniformly, without special cases for the grid border.

Under this convention every triangle has its black right-angle at the corner indicated by its name and its hypotenuse between the two adjacent corners of the cell. The missing corner (the 45° notch poking into the white half of the cell) is the diagonally opposite one.

Five constraint families

A. One state per cell. For every white cell (r, c),

$$x_{r,c,\emptyset} + x_{r,c,\mathrm{NW}} + x_{r,c,\mathrm{NE}} + x_{r,c,\mathrm{SW}} + x_{r,c,\mathrm{SE}} \;=\; 1.$$

This is the only equality in the model; every other constraint is an inequality.

B. Clues and wedge prohibition. A triangle in a neighbour of a black cell B either presents a full black side to B or leaves a 45° wedge poking into B. The latter is forbidden because it would place a non-rectangular segment on the boundary of the white region adjoining B. For the upper neighbour $(r-1, c)$ of B, the forbidden wedge orientations are NW and NE; the orientations contributing a black side to B are SW and SE. Summing over all four axial neighbours and setting the sum equal to the clue k yields the clue constraint; the wedge prohibition is a stronger inequality setting each wedge orientation to zero for every neighbour of every black cell.

C. Triangle chaining. The slanted edges of diamond-shaped white regions must close smoothly. Consider a cell (r, c) in the SW state, so its hypotenuse runs from the NW corner to the SE corner. The SE endpoint of this hypotenuse sits at grid vertex $(r+1, c+1)$. For a local rectangular white boundary to exist near that vertex, one of two things must happen: the slanted edge *continues* as another SW triangle at $(r+1, c+1)$, with the sandwiched axial neighbour $(r, c+1)$ empty; or the slanted edge *turns* 90°, met by an SE triangle at $(r, c+1)$ whose hypotenuse is perpendicular to ours. The two requirements are encoded as two inequalities,

$$x_{r,c,\mathrm{SW}} \;\le\; x_{r,c+1,\mathrm{SE}} + x_{r+1,c+1,\mathrm{SW}},$$

$$x_{r,c,\mathrm{SW}} + x_{r+1,c+1,\mathrm{SW}} \;\le\; x_{r,c+1,\emptyset} + 1.$$

Four triangle orientations times two hypotenuse endpoints give eight symmetric pairs in total, sixteen inequalities per cell.

D. Concave-corner exclusion. After *A*, *B*, *C* every angle on the boundary of a white region is either 90° (convex) or 270° (concave). Shakashaka forbids 270°: the interior angles must all be right angles pointing outward. A 270° angle arises only when three cells of a 2×2 block are all empty and the fourth cell sits at the concave vertex. Healing requires the fourth cell to be either empty (turning the L-shape into a 2×2 white square) or to hold the triangle whose black right-angle bites off the concave vertex. For the L-shape with corners at

(r, c), $(r+1, c)$, $(r, c+1)$ all empty,

$$x_{r,c,\emptyset} + x_{r+1,c,\emptyset} + x_{r,c+1,\emptyset} \;\le\; x_{r+1,c+1,\emptyset} + x_{r+1,c+1,\mathrm{NW}} + 2.$$

Three rotations cover the other three L-shape orientations.

E. Nested-diamond exclusion. *A–D* guarantee that every white region has a locally rectangular boundary with only 90° angles. One loophole remains: two nested same-orientation diamonds can coexist with nothing between them, and neither *A* nor *B* nor *C* nor *D* forbids it. The fix is a non-local inequality on pairs of cells sharing a diagonal: two SE triangles at (r, c) and $(r+k, c+k)$ either lie on the same diamond or have some NW strictly between them breaking the nesting:

$$x_{r,c,\mathrm{SE}} + x_{r+k,c+k,\mathrm{SE}} \;\le\; \sum_{0<k'<k} x_{r+k',c+k',\mathrm{NW}} + 1 \qquad (k \ge 2).$$

Three rotations cover the other three diagonal cases.

A solver in sixty lines

```
from ortools.sat.python import cp_model

def solve_shakashaka(R, C, black, clues):
    """black: set of (r,c) black cells.
       clues: {(r,c) -> int} for clued cells."""
    m = cp_model.CpModel()
    S = ["E", "NW", "NE", "SW", "SE"]
    x = {}
    for r, c in ((r, c) for r in range(R)
                        for c in range(C)):
        if (r, c) in black: continue
        for s in S:
            x[(r, c, s)] = m.NewBoolVar("")
    def v(r, c, s):
        if not (0 <= r < R and 0 <= c < C): return 0
        if (r, c) in black: return 0
        return x[(r, c, s)]
    # (A) one state per white cell
    for r, c, *_ in [(r, c) for r in range(R)
                            for c in range(C)
                            if (r, c) not in black]:
        m.Add(sum(x[(r, c, s)] for s in S) == 1)
    # (B) wedge prohibition and clue counts
    WEDGE = {(-1, 0): ("NW", "NE"),
             ( 1, 0): ("SW", "SE"),
```

```
            ( 0,-1): ("NW", "SW"),
            ( 0, 1): ("NE", "SE")}
    FACE  = {(-1, 0): ("SW", "SE"),
            ( 1, 0): ("NW", "NE"),
            ( 0,-1): ("NE", "SE"),
            ( 0, 1): ("NW", "SW")}
    for (r, c) in black:
        for (dr, dc), fb in WEDGE.items():
            nr, nc = r+dr, c+dc
            for s in fb:
                if v(nr, nc, s) != 0:
                    m.Add(x[(nr, nc, s)] == 0)
    for (r, c), k in clues.items():
        ts = [v(r+dr, c+dc, s)
              for (dr, dc), fa in FACE.items()
              for s in fa]
        m.Add(sum(ts) == k)
    # (C) chain-or-turn plus sandwich parity
    CHAINS = [
        ("SW", (0, 1), "SE", ( 1,  1)),
        ("SE", (0,-1), "SW", ( 1, -1)),
        ("NW", (0, 1), "NE", (-1,  1)),
        ("NE", (0,-1), "NW", (-1, -1)),
        ("SW", (1, 0), "NE", ( 1,  1)),
        ("NE",(-1, 0), "SW", (-1, -1)),
        ("SE", (1, 0), "NW", ( 1, -1)),
        ("NW",(-1, 0), "SE", (-1,  1)),
    ]
    for g, (ar, ac), ax, (fr, fc) in CHAINS:
        for r in range(R):
            for c in range(C):
                lhs = x.get((r, c, g))
                if lhs is None: continue
                ax_ = v(r+ar, c+ac, ax)
                fa_ = v(r+fr, c+fc, g)
                m.Add(lhs <= ax_ + fa_)
                if fa_ != 0:
                    em = v(r+ar, c+ac, "E")
                    m.Add(lhs + fa_ <= em + 1)
    # (D) concave-corner exclusion (4 rotations)
    for r in range(R-1):
        for c in range(C-1):
            E00 = v(r,   c,   "E"); E10 = v(r+1, c,   "E")
            E01 = v(r,   c+1, "E"); E11 = v(r+1, c+1, "E")
            m.Add(E00 + E10 + E01
                  <= E11 + v(r+1, c+1, "NW") + 2)
            m.Add(E00 + E10 + E11
                  <= E01 + v(r,   c+1, "SW") + 2)
```

```
            m.Add(E00 + E01 + E11
                  <= E10 + v(r+1, c,   "NE") + 2)
            m.Add(E10 + E01 + E11
                  <= E00 + v(r,   c,   "SE") + 2)
    # (E) nested-diamond exclusion (4 diagonals)
    NEST = [("SE","NW", 1, 1), ("NW","SE", 1, 1),
            ("SW","NE", 1,-1), ("NE","SW", 1,-1)]
    for gA, gB, dr, dc in NEST:
        for r in range(R):
            for c in range(C):
                if v(r, c, gA) == 0: continue
                k = 2
                while (0 <= r+k*dr < R
                       and 0 <= c+k*dc < C):
                    far = v(r+k*dr, c+k*dc, gA)
                    if far == 0:
                        k += 1; continue
                    sep = [v(r+kp*dr, c+kp*dc, gB)
                           for kp in range(1, k)]
                    m.Add(x[(r, c, gA)] + far
                          <= sum(t for t in sep
                                 if t != 0) + 1)
                    k += 1
    s = cp_model.CpSolver()
    s.Solve(m)
    out = {(r, c): next(st for st in S
                        if s.Value(x[(r, c, st)]))
           for (r, c, st) in x if st == "E"}
    return out
```

The toy puzzle of Figure 18.1 solves in about 10 milliseconds.

A hard instance

Figure 18.3 is a 20 × 20 Shakashaka harvested from *puzzle-shakashaka.com*'s expert tier. CP-SAT returns the unique solution in about 1.2 seconds.

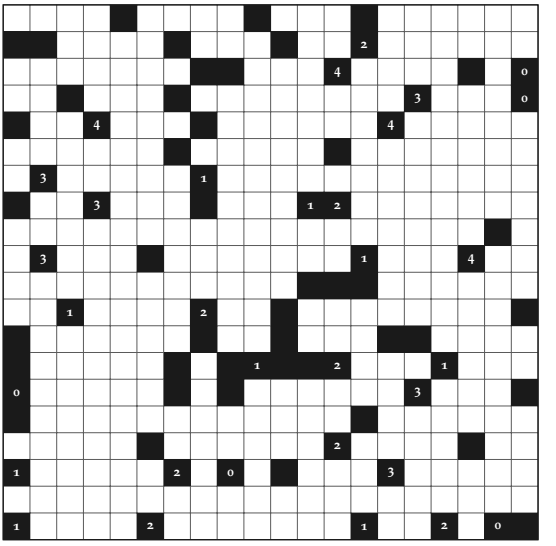

Figure 18.3: *A* 20×20 *Shakashaka from* puzzle-shakashaka.com.

Sources. Shakashaka was first published by Nikoli in 2008. The toy instance of Figure 18.1 is adapted from *Puzzle Communication Nikoli*; the hard instance of Figure 18.3 is puzzle 1127 from *puzzle-shakashaka.com*. Demaine, Okamoto, Uehara and Uno (*Computational Geometry*, 2013) proved Shakashaka NP-complete via a reduction from planar 3-SAT, and provided the integer-programming formulation reproduced here.

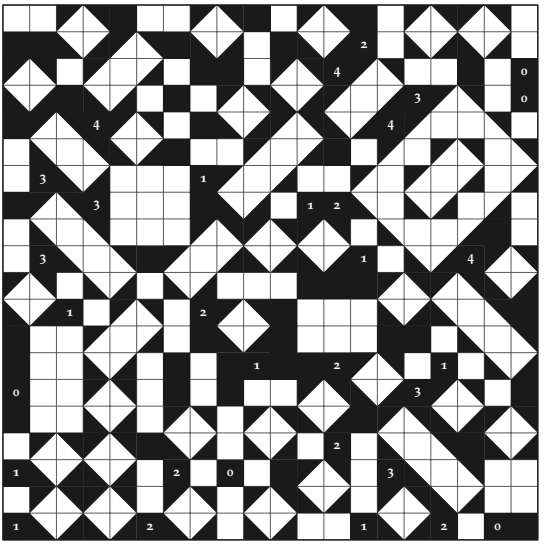

Figure 18.4: *The unique solution. Every clue counts its triangle-facing neighbours; every white region is a rectangle or diamond.*

Chapter 19

Marupeke

Marupeke is the puzzle of circles and crosses. The grid is a rectangle of white cells and optional black blockers. Each white cell must be filled with either a circle ○ or a cross ×, subject to a single sweeping constraint: no three cells of the same symbol may appear consecutively along any row, column, or either diagonal. Some cells are pre-filled as clues; black blockers interrupt lines, so a sequence of three is counted only when none of its members is black.

The puzzle's name combines *maru* (circle) and *peke* (cross), two ubiquitous marks in Japanese pencil puzzling: ○ for "correct" and × for "wrong." Marupeke is a close relative of Takuzu (Chapter 5), sharing the no-triple rule; the difference is that Marupeke has no balance constraint on row or column totals, but enforces the triple rule along both diagonals as well. This single change makes the flavour strikingly different: Marupeke solutions have no large-scale structure of equal-counts rows, only the local forbidden-pattern scaffolding threading every line of the grid.

The black blockers are load-bearing. Three cells in a line that straddle a blocker are *not* a triple, because the line segment is broken. Setters use blockers to seed a puzzle: a lattice of blockers carves the grid into many short runs, and the triple constraint then propagates tightly through each run.

Rules and a small instance

A Marupeke puzzle is an $R \times C$ grid. Each cell is either a *white cell* (possibly pre-filled with ○ or × as a clue) or a *black blocker*. The solver must fill every remaining white cell with ○ or × such that:

1. Every pre-filled clue is preserved.
2. No three consecutive cells along any row, column, NW-SE diagonal, or NE-SW diagonal carry the same symbol, where "consecutive" means three white cells in a straight line with no black blocker among them.

Figure 19.1 is a 6 × 6 Marupeke, puzzle 4 from Alex Bellos's *Puzzle Ninja* (2017), with three blockers and nine clues.

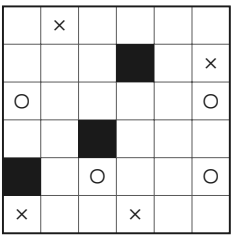

Figure 19.1: *A* 6 × 6 *Marupeke* (*Bellos,* Puzzle Ninja, *puzzle* 4). *Bold glyphs are clues; black squares are blockers.*

The programming model

Assign each white cell a Boolean $v_{r,c}$ with 0 meaning ○ and 1 meaning ×. Black blockers are not assigned variables; whenever a constraint references a black blocker, the triple through that cell is skipped entirely (unlike the Demaine

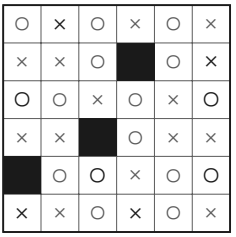

Figure 19.2: *The unique solution. Every row, column, and diagonal respects the no-three-same rule, counting only runs uninterrupted by a blocker.*

boundary convention used for Shakashaka, here skipping rather than zero-treating is the natural encoding).

The constraint families reduce to a single clean template.

Clue preservation

For each pre-filled cell (r, c) with clue $k \in \{0, 1\}$,

$$v_{r,c} = k.$$

No-triple-same

For each direction $\vec{d} \in \{(0,1), (1,0), (1,1), (1,-1)\}$ and each cell (r, c) such that (r, c), $(r + d_r, c + d_c)$, and $(r + 2d_r, c + 2d_c)$ all lie inside the grid and are all white (no blockers), let

$$S = v_{r,c} + v_{r+d_r,c+d_c} + v_{r+2d_r,c+2d_c}.$$

The triple is forbidden from being all ○ ($S = 0$) and from being all × ($S = 3$), so the constraint becomes

$$1 \leq S \leq 2.$$

This is tight: if $S = 1$ or $S = 2$, the triple contains at least one of each symbol, which is exactly what the rule demands.

A solver in thirty lines

```
from ortools.sat.python import cp_model

def solve_marupeke(board):
    """board: R x C with entries in {-1, 0, 1, None}
         -1 = black blocker
          0 = clue O (bigcirc)
          1 = clue X (times)
          None = empty white cell"""
    R, C = len(board), len(board[0])
    m = cp_model.CpModel()
    v = {}
    for r in range(R):
        for c in range(C):
            if board[r][c] != -1:
                v[(r, c)] = m.NewBoolVar("")
    # clue preservation
    for r in range(R):
        for c in range(C):
            if board[r][c] in (0, 1):
                m.Add(v[(r, c)] == board[r][c])
    # no-triple-same along four directions
    DIRS = [(0, 1), (1, 0), (1, 1), (1, -1)]
    for r in range(R):
        for c in range(C):
            if board[r][c] == -1: continue
            for dr, dc in DIRS:
                pts = [(r+i*dr, c+i*dc) for i in range(3)]
                inside = all(0 <= pr < R and 0 <= pc < C
                             for (pr, pc) in pts)
                if not inside: continue
                if any(board[pr][pc] == -1
                       for (pr, pc) in pts): continue
                S = sum(v[p] for p in pts)
                m.Add(1 <= S)
                m.Add(S <= 2)
    s = cp_model.CpSolver()
    s.Solve(m)
    return [[(None if board[r][c] == -1
              else int(s.Value(v[(r, c)])))
             for c in range(C)]
            for r in range(R)]
```

The toy of Figure 19.1 solves uniquely in about 14 milliseconds.

A hard instance

Figure 19.3 is the 10 × 10 Marupeke, puzzle 5 from the same volume. It has fifteen blockers arranged in an irregular lattice and twenty clues. CP-SAT returns the unique solution in about 2 milliseconds: the blocker lattice breaks the grid into so many short runs that each triple constraint is local and the solver's propagation engine dispatches them immediately.

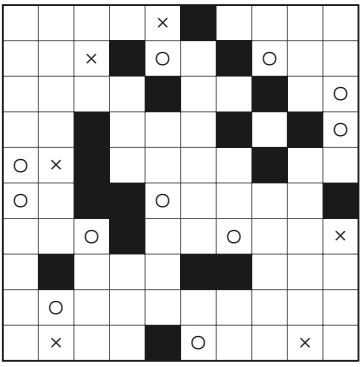

Figure 19.3: *A* 10×10 *Marupeke* (*Bellos,* Puzzle Ninja, *puzzle* 5).

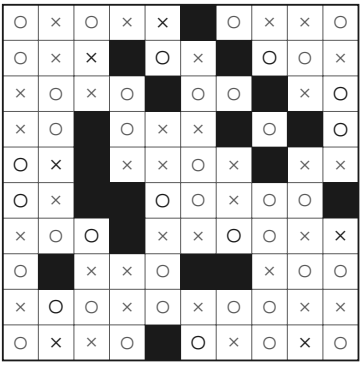

Figure 19.4: *The unique solution, recovered in about* 2 *milliseconds.*

Sources. Marupeke is a Japanese pencil puzzle brought to English-language audiences via Alex Bellos's *Puzzle Ninja: Pit Your Wits Against the Japanese Puzzle Masters* (Guardian Faber, 2017). The toy and hard instances of Figures 19.1 and 19.3 are puzzles 4 and 5 from that collection. Like Takuzu (Chapter 5), Marupeke is solvable in polynomial time at any fixed grid size once the blockers are seen as edges in a constraint graph, but for the unconstrained problem (arbitrary grids, variable blocker placement) the complexity question remains open in the literature.

Chapter 20

Walls

Walls is the puzzle of minimal strokes. The grid is a rectangle of white cells punctuated by black cells carrying clue integers. The solver fills every white cell with either a short horizontal segment or a short vertical segment, meeting edge-to-edge along adjacent cells to form continuous straight lines. Lines cannot pass through black cells; each line terminates at either a black cell, the grid boundary, or an adjacent perpendicular segment. The clue in a black cell equals the *combined length*, summed across all four directions, of the line segments terminating at that black cell.

Walls is a piece of puzzle minimalism: no shading, no symbols, no colour, just a lattice of horizontal and vertical strokes. The rules are sparse enough to fit on a single line, but solving a hard instance is a genuine struggle, because the combined-length clue is inherently non-local. The value in a black cell is determined by lines that may start anywhere along the ray from the black cell, and the ray can be arbitrarily long, constrained only by the other black cells and the grid boundary.

Among the solver-friendly logic puzzles, Walls sits on the challenging side of the constraint-modelling spectrum. The clue-to-combined-length rule does not reduce to a local window; it requires the model to count the length of a maximal run of same-type segments extending out from each black cell. Two families of constraints capture this cleanly: one forcing the run lengths to sum to the clue, and an

auxiliary family that counts each directional run via a ladder of prefix-all-same indicators.

Rules and a small instance

A Walls puzzle is an $R \times C$ grid of cells. Each cell is either *white* (empty, to be filled) or *black* (with a non-negative integer clue). A solution assigns to every white cell one of two tokens: a horizontal segment `-` or a vertical segment `|`. The solution satisfies:

1. Every black cell's clue k equals the sum, over the four axial directions, of the length of the *maximal run* of segments with matching orientation extending from the black cell in that direction (horizontal segments on the east-west axis, vertical segments on the north-south axis). A maximal run terminates at either a black cell, the grid boundary, or the first segment of the opposite orientation.

Figure 20.1 is a 4×4 Walls puzzle; five black cells carry clues summing to 11, which fixes the fill of the remaining eleven white cells.

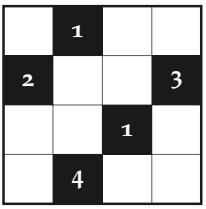

Figure 20.1: *A* 4×4 *Walls puzzle.*

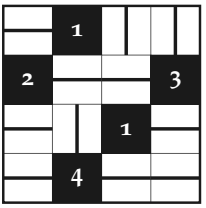

Figure 20.2: *The unique solution. Every clue counts its total incident line length across all four directions.*

The programming model

For every white cell (r, c) introduce two Boolean variables $h_{r,c}$ and $v_{r,c}$, with

$$h_{r,c} + v_{r,c} \;=\; 1,$$

so every white cell hosts exactly one segment orientation. Black cells carry no variables.

The interesting machinery is the *directional run length*. For a black cell (r, c) and a direction $\vec{d} \in \{(0,1),(0,-1),(1,0),(-1,0)\}$, let $c_1, c_2, \ldots, c_D$ be the consecutive white cells extending from (r, c) in direction $\vec{d}$ up to (but not including) the first black cell or the grid edge. The length of the run ending at (r, c) from direction $\vec{d}$ is

$$L \;=\; |\{i : c_1, c_2, \ldots, c_i \text{ all carry the matching orientation}\}|.$$

Here "matching orientation" means $h = 1$ for the east-west directions, $v = 1$ for north-south.

To encode L in CP-SAT, introduce for each $K \in \{1, 2, \ldots, D\}$ a Boolean indicator b_K with

$$b_K = 1 \iff s_1 + s_2 + \cdots + s_K = K,$$

where s_i is the matching variable at c_i. In other words, b_K asserts that the first K cells along the ray all have the matching orientation. Then

$$L \;=\; \sum_{K=1}^{D} b_K,$$

because b_K drops from 1 to 0 at exactly the index K where the run first breaks.

For each black cell with clue k, the clue constraint is

$$L^{\text{right}} + L^{\text{left}} + L^{\text{up}} + L^{\text{down}} \;=\; k.$$

A solver in forty-five lines

```
from ortools.sat.python import cp_model

def solve_walls(board):
    """board[r][c] = -1 for white cell,
       non-negative int for black clue."""
    R, C = len(board), len(board[0])
    m = cp_model.CpModel()
    h, v = {}, {}
    for r in range(R):
        for c in range(C):
            if board[r][c] == -1:
                h[(r, c)] = m.NewBoolVar("")
                v[(r, c)] = m.NewBoolVar("")
                m.Add(h[(r, c)] + v[(r, c)] == 1)
    def ray(r, c, dr, dc):
        cells = []
        nr, nc = r + dr, c + dc
        while (0 <= nr < R and 0 <= nc < C
               and board[nr][nc] == -1):
            cells.append((nr, nc))
            nr += dr; nc += dc
        return cells
    def run_len(cells, is_h):
        if not cells: return 0
        vs = [h[c] if is_h else v[c] for c in cells]
        bs = []
        for K in range(1, len(cells) + 1):
            b = m.NewBoolVar("")
            m.Add(sum(vs[:K]) >= K).OnlyEnforceIf(b)
            m.Add(sum(vs[:K]) <= K - 1
                 ).OnlyEnforceIf(b.Not())
            bs.append(b)
        return sum(bs)
    for r in range(R):
        for c in range(C):
            if board[r][c] == -1: continue
            k = board[r][c]
            m.Add(run_len(ray(r, c,  0,  1), True)
                  + run_len(ray(r, c,  0, -1), True)
```

```
                + run_len(ray(r, c,  1,  0), False)
                + run_len(ray(r, c, -1,  0), False)
                == k)
    s = cp_model.CpSolver()
    s.Solve(m)
    return [[('h' if s.Value(h[(r, c)]) else 'v')
             if board[r][c] == -1 else board[r][c]
             for c in range(C)]
            for r in range(R)]
```

The toy of Figure 20.1 solves uniquely in about 3 milliseconds.

A hard instance

Figure 20.3 is a 7 × 7 Walls with fourteen black clues, constructed in the style of Alex Bellos's puzzles in *Puzzle Ninja*. CP-SAT returns the unique solution in about 22 milliseconds.

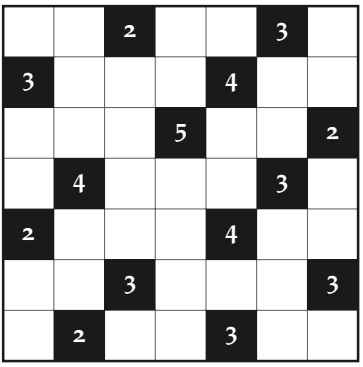

Figure 20.3: *A 7 × 7 Walls puzzle.*

Sources. Walls appears in Alex Bellos's *Puzzle Ninja: Pit Your Wits Against the Japanese Puzzle Masters* (Guardian Faber, 2017), where it is credited to Japanese pencil-puzzle

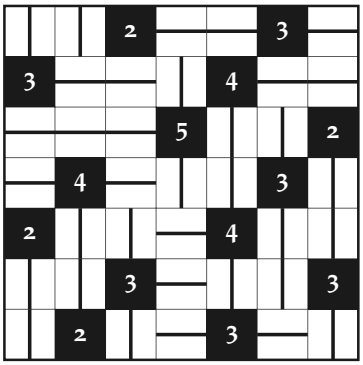

Figure 20.4: *The unique solution, recovered in about 22 milliseconds.*

tradition. The clue-as-combined-run-length rule generalises the directional-sum rule seen in Nurikabe and Heyawake variants, but the combined-length encoding via ladder indicators is specific to Walls and the family of "count what you can reach" puzzles. Complexity is open: the puzzle is clearly in NP; no polynomial-time algorithm is known.

Chapter 21

L-Panel

L-Panel is the puzzle of tiling with bent pieces. The grid is a rectangle of white cells interrupted by an irregular scattering of black cells. The solver covers every white cell with L-shaped tetrominoes, each occupying exactly four cells in one of eight orientations (four rotations of the L-tetromino and their four mirror images, equivalently the L- and J-tetrominoes of Tetris). No tile may overlap another, no tile may cross a black cell, and no white cell may be left uncovered.

L-Panel is a pure *exact-cover* puzzle. Unlike the shading and line-drawing puzzles elsewhere in this volume, there are no numeric clues to satisfy and no symmetry conditions to obey: the black cells alone determine the geometry, and the solver's task is to jigsaw the remaining white region into L-tiles. The puzzle rewards patience with pattern-recognition over calculation, and it is particularly satisfying to watch a constraint solver reduce it to a single integer-programming expression and return the tiling in a few milliseconds.

The L-tetromino (also called the "L-piece") is one of the seven one-sided tetrominoes studied in mathematical recreational theory. Every connected region with area divisible by four is L-tileable provided its shape is not too pathological, but deciding tileability of an arbitrary region is NP-hard in general (Moore and Robson, 2001); L-Panel instances are hand-designed so that the black-cell scaffolding leaves a uniquely tileable white region.

✠

Rules and a small instance

An L-Panel puzzle is an $R \times C$ grid of cells, each either *white* or *black*. The number of white cells must be divisible by four. A solution is a set of L-tetrominoes (any of the eight orientations) such that:

1. Every white cell belongs to exactly one tile.
2. No tile covers a black cell.
3. No two tiles overlap.

Figure 21.1 is a 6×6 L-Panel with eight black cells. The remaining 28 white cells partition uniquely into 7 L-tiles.

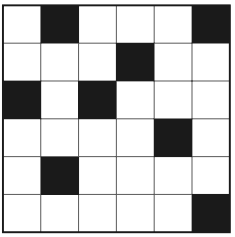

Figure 21.1: *A* 6×6 *L-Panel puzzle with* 8 *black cells and* 28 *white cells.*

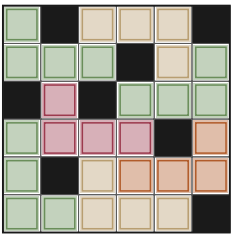

Figure 21.2: *The unique L-tiling. Adjacent tiles receive different colours to separate them visually.*

The programming model

The model is the textbook exact-cover formulation.

Candidate tiles

Enumerate the eight orientations of the L-tetromino as lists of $(\Delta r, \Delta c)$ offsets:

$$\mathcal{S} \;=\; \{\text{L, J, and all } 90^\circ \text{ rotations}\}.$$

For every cell (r, c) of the grid and every shape $S \in \mathcal{S}$, form the placement

$$P(r, c, S) \;=\; \{(r + \Delta r, c + \Delta c) \;:\; (\Delta r, \Delta c) \in S\}.$$

Retain $P(r, c, S)$ as a *candidate tile* if all four of its cells lie inside the grid and none is black. Let $\mathcal{T}$ denote the resulting set of candidate tiles (as frozensets, to deduplicate placements of the same tile encountered via different anchor cells).

Exact cover

Introduce a Boolean variable $t_T \in \{0, 1\}$ for each $T \in \mathcal{T}$, with $t_T = 1$ meaning "tile T is selected." For every white cell (r, c),

$$\sum_{T \in \mathcal{T} \,:\, (r,c) \in T} t_T \;=\; 1.$$

This is a classical exact-cover constraint: the $|\mathcal{T}|$-by-N incidence matrix $[(r, c) \in T]$ has row sum equal to 1 in every white-cell row. The constraint is tight: if a white cell is covered by zero tiles, it is uncovered (forbidden); if it is covered by two or more tiles, they overlap (forbidden).

No explicit overlap constraint is needed, because the exact-cover condition on every white cell implies that every chosen tile's four cells are each covered by exactly that tile and no other.

A solver in thirty lines

```
from ortools.sat.python import cp_model
from collections import defaultdict

L_SHAPES = [
    [(0,0),(1,0),(2,0),(2,1)],
    [(0,0),(0,1),(0,2),(1,2)],
    [(0,1),(1,1),(2,1),(2,0)],
    [(1,0),(1,1),(1,2),(0,0)],
    [(1,0),(1,1),(1,2),(0,2)],
    [(0,0),(0,1),(1,0),(2,0)],
    [(0,0),(0,1),(1,1),(2,1)],
    [(0,0),(1,0),(0,1),(0,2)],
]

def solve_lpanel(board):
    R, C = len(board), len(board[0])
    polys = set()
    for r in range(R):
        for c in range(C):
            for sh in L_SHAPES:
                p = frozenset(
                    (r+dr, c+dc) for dr, dc in sh)
                if all(0 <= pr < R and 0 <= pc < C
                       and board[pr][pc] != -1
                       for pr, pc in p):
                    polys.add(p)
    m = cp_model.CpModel()
    t = {p: m.NewBoolVar("") for p in polys}
    cell_to_polys = defaultdict(list)
    for p in polys:
        for cell in p:
            cell_to_polys[cell].append(p)
    for r in range(R):
        for c in range(C):
            if board[r][c] != -1:
                m.Add(sum(t[p]
                          for p in cell_to_polys[(r, c)])
                      == 1)
    s = cp_model.CpSolver()
    s.Solve(m)
    return [p for p in polys if s.Value(t[p])]
```

The toy of Figure 21.1 solves uniquely in about 14 milliseconds.

A hard instance

Figure 21.3 is a 10 × 10 L-Panel in the style of Alex Bellos's *Puzzle Ninja*: twelve black cells scatter the interior, leaving 88 white cells (exactly 22 L-tiles) for the solver. CP-SAT returns the unique tiling in about 3 milliseconds.

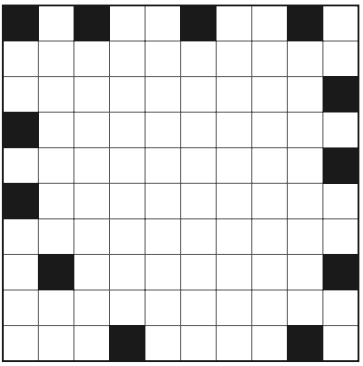

Figure 21.3: *A* 10 × 10 *L-Panel with* 12 *black cells and* 88 *white cells.*

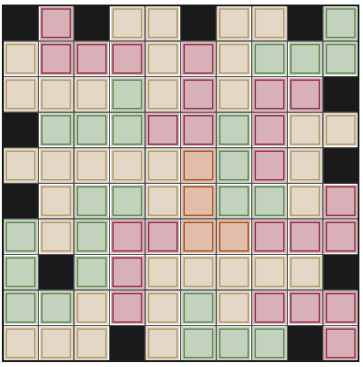

Figure 21.4: *The unique* 22*-tile L-panel, recovered in about* 3 *milliseconds.*

Sources. L-Panel appears in Alex Bellos's *Puzzle Ninja: Pit Your Wits Against the Japanese Puzzle Masters* (Guardian Faber, 2017), where it is credited to Japanese pencil-puzzle tradition. The hard instance of Figure 21.3 is puzzle 7 from that collection. The exact-cover formulation used here is due to Dancing Links (Knuth, 2000), though CP-SAT propagates the same logic via row-sum unit constraints; Knuth's DLX algorithm remains the canonical exact-cover tool for hand-tuned enumeration. Tiling a region by L-tetrominoes is NP-hard in general (Moore and Robson, 2001).

Chapter 22

BlockNumber

BlockNumber is the puzzle of counting up each region. The grid is partitioned into irregular regions (the "blocks"), and every block carries an implicit local counter: a block of size n must be filled with the numbers $1, 2, \ldots, n$, each exactly once. Some cells inside blocks are pre-filled as clues. A single further rule knits the blocks together: no two cells that are adjacent in the *king-move* sense (orthogonally or diagonally) may hold the same number, even across a block boundary.

The king-move constraint is the puzzle's signature. Where Sudoku forbids repetition along rows, columns, and regions, BlockNumber forbids repetition in every 3×3 neighbourhood around every cell. This makes solving feel like threading a low-collision schedule: small blocks (size 1 or 2) are forced by their neighbours' values, and the force propagates outward through the king-move contact graph.

BlockNumber is one of the younger Japanese pencil puzzles collected in Alex Bellos's *Puzzle Ninja* anthology. The block partition is hand-designed so that the combination of "count 1 to block size" and the king-move constraint yields a unique solution. Variants of the puzzle run under different English names (*Fillomino*-with-bounds, *LITS*-counter, etc.), but the core rule set is stable.

Rules and a small instance

A BlockNumber puzzle is an $R \times C$ grid of cells, partitioned into disjoint connected blocks $B_1, B_2, \ldots, B_m$ covering every cell. Some cells carry pre-filled positive-integer clues. A solution assigns to every cell a positive integer such that:

1. For every block B_i of size n_i, the values assigned to its cells are exactly $\{1, 2, \ldots, n_i\}$, each occurring once.
2. For every pair of cells (r, c) and (r', c') with $\max(|r - r'|, |c - c'|) = 1$ (i.e. king-move adjacent), the values assigned are different.
3. Every pre-filled clue is preserved.

Figure 22.1 is a 4×4 BlockNumber with four blocks of sizes $3, 4, 4, 5$ and two clues.

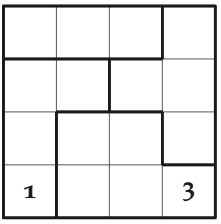

Figure 22.1: *A 4×4 BlockNumber with four blocks (sizes $3, 4, 4, 5$). Thick lines mark block boundaries; two cells carry clues.*

The programming model

Assign each cell (r, c) an integer variable $v_{r,c}$. If the cell belongs to a block of size n, its domain is $\{1, 2, \ldots, n\}$.

3	1	2	1
2	4	3	4
3	5	1	2
1	2	4	**3**

Figure 22.2: *The unique solution. Each block holds* 1 *through its size, and no two king-adjacent cells share a value.*

Block domains and uniqueness

For every block $B = \{(r_1, c_1), \dots, (r_n, c_n)\}$ of size n,

$$v_{r_i,c_i} \in \{1, 2, \dots, n\} \qquad \text{and} \qquad \text{AllDifferent}(v_{r_1,c_1}, \dots, v_{r_n,c_n}).$$

Together these assert that B's values are a permutation of $\{1, 2, \dots, n\}$.

King-move no-equal

For every cell (r, c) and each of its eight king-move neighbours $(r + \delta_r, c + \delta_c)$ with $\max(|\delta_r|, |\delta_c|) = 1$ lying inside the grid:

$$v_{r,c} \neq v_{r+\delta_r,c+\delta_c}.$$

Because the relation is symmetric, the constraint is posted only for ordered pairs $((r, c), (r', c'))$ with $(r, c) < (r', c')$ lexicographically.

Clue preservation

For every pre-filled cell (r, c) with clue k,

$$v_{r,c} = k.$$

A solver in thirty lines

```
from ortools.sat.python import cp_model

def solve_blocknumber(blocks, R, C):
    """blocks: list of dicts {(r,c): clue or ""};
       each dict is one block."""
    m = cp_model.CpModel()
    v = {}
```

```
    for block in blocks:
        n = len(block)
        for (r, c) in block:
            v[(r, c)] = m.NewIntVar(1, n, "")
    # Clue preservation
    for block in blocks:
        for (r, c), clue in block.items():
            if isinstance(clue, int):
                m.Add(v[(r, c)] == clue)
    # Block all-different
    for block in blocks:
        m.AddAllDifferent([v[c] for c in block])
    # King-move no-equal
    for (r, c) in v:
        for dr in (-1, 0, 1):
            for dc in (-1, 0, 1):
                if (dr, dc) == (0, 0): continue
                nb = (r + dr, c + dc)
                if nb in v and nb > (r, c):
                    m.Add(v[(r, c)] != v[nb])
    s = cp_model.CpSolver()
    s.Solve(m)
    return {c: s.Value(v[c]) for c in v}
```

The toy of Figure 22.1 solves uniquely in about 15 milliseconds.

A hard instance

Figure 22.3 is a 7 × 7 BlockNumber with twelve blocks of mixed sizes (ranging from 2 to 5), a single clue, and 49 cells to fill. The block partition is visible from the thick boundary lines. CP-SAT returns the unique solution in about 4 milliseconds.

Sources. BlockNumber is a contemporary Japanese pencil-puzzle kind collected in Alex Bellos's *Puzzle Ninja: Pit Your Wits Against the Japanese Puzzle Masters* (Guardian Faber,

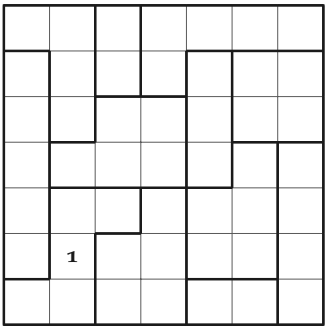

Figure 22.3: *A* 7 × 7 *BlockNumber with twelve blocks.*

1	3	2	3	1	4	2
5	4	1	5	2	3	1
3	2	3	4	1	4	2
1	5	1	2	3	5	3
2	3	4	5	1	2	1
4	1	2	3	4	3	4
2	5	4	1	2	1	2

Figure 22.4: *The unique fill, recovered in about* 4 *milliseconds.*

2017), where the style is sometimes attributed to the *Puzzle Communication Nikoli* tradition of irregular-region fillers. The king-move constraint distinguishes BlockNumber from its closer relative Fillomino (Chapter 16), which forbids only orthogonal equal neighbours and has no fixed block sizes. Computational complexity of BlockNumber is open; like most region filler puzzles, it is likely NP-complete under suitable complexity-theoretic reductions, but no published proof is known.

Chapter 23

Searchlights

Searchlights is the puzzle of seeing through a row. The grid is a rectangle of white cells and black cells; each black cell carries a non-negative integer clue. The solver places "lights" in a subset of white cells. Each clue counts the number of lights visible from its black cell looking in the four orthogonal directions (up, down, left, right, but not diagonally), where a light is visible through other lights but not through black cells: a black cell terminates the ray.

Searchlights belongs to the family of *line-of-sight* puzzles that also includes Akari (Chapter 11) and the lesser-known Karesansui. All three work with the same geometric primitive, an unobstructed row or column segment bounded by black cells, and give the solver a different kind of task on each segment. Akari asks about illumination (every white cell seen from at least one light); Searchlights asks about counting (each clue equals the number of lights in its four-way view cone). The line-of-sight model makes Searchlights naturally linear: every clue is a single linear sum of Boolean light variables over a fixed set of cells.

Among the puzzles in this volume, Searchlights has the simplest constraint structure, but the deductive work to solve by hand is non-trivial. Tight clues like 0 and the maximum "all four rays full" force lights into predictable positions, but intermediate clues leave parity and placement ambiguities that must be triangulated against several other rays. The solver handles this combinatorial triangulation in a millisecond; by hand it is an hour.

✠

Rules and a small instance

A Searchlights puzzle is an $R \times C$ grid of cells. Each cell is either *white* or *black with a non-negative integer clue*. A solution assigns to every white cell the status *light* or *empty* such that:

1. Every black clue k equals the number of lights in the *accessible cells* of the clue: all white cells lying in the same row or column as the clue, between the clue and the first obstruction (another black cell or the grid boundary) in each of the four orthogonal directions.
2. Every white cell either holds at most one light or is empty.

Figure 23.1 is a 4×4 Searchlights with six black clues. The lights are marked by filled circles in the solution.

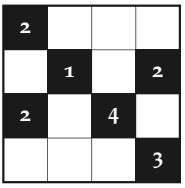

Figure 23.1: *A* 4×4 *Searchlights puzzle.*

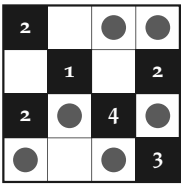

Figure 23.2: *The unique solution. Filled circles mark the lights.*

The programming model

For every white cell (r, c) introduce a Boolean $v_{r,c}$, with $v_{r,c} = 1$ meaning "light at (r, c)."

For every black cell at (r, c) with clue k, compute its *accessible set* $A(r, c)$: the union over the four axial directions of the white cells from the neighbour of (r, c) in that direction up to (but not including) the first black cell or the grid edge, whichever comes first.

The clue constraint is a single linear equation:

$$\sum_{(r',c') \in A(r,c)} v_{r',c'} \;=\; k.$$

That is the entire model. No at-most-one-per-cell constraint is needed because each v is already Boolean (0 or 1); the "at most one light per cell" phrasing in the rules is a natural consequence of the domain $\{0, 1\}$.

A solver in twenty lines

```
from ortools.sat.python import cp_model

def solve_searchlights(board):
    """board[r][c] = 0 for empty white cell,
       or a positive integer clue for a black cell."""
    R, C = len(board), len(board[0])
    m = cp_model.CpModel()
    v = {}
    for r in range(R):
        for c in range(C):
            if board[r][c] == 0:
                v[(r, c)] = m.NewBoolVar("")
    def accessible(r, c):
        out = []
        for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            nr, nc = r + dr, c + dc
            while (0 <= nr < R and 0 <= nc < C
                   and board[nr][nc] == 0):
                out.append((nr, nc))
                nr += dr; nc += dc
        return out
```

```
    for r in range(R):
        for c in range(C):
            if board[r][c] != 0:
                A = accessible(r, c)
                m.Add(sum(v[p] for p in A)
                      == board[r][c])
    s = cp_model.CpSolver()
    s.Solve(m)
    return [[(s.Value(v[(r, c)]) if board[r][c] == 0
              else -board[r][c])
             for c in range(C)]
            for r in range(R)]
```

The toy of Figure 23.1 solves uniquely in about 14 milliseconds.

A hard instance

Figure 23.3 is a 9 × 9 Searchlights from Alex Bellos's *Puzzle Ninja*, puzzle 4. Twenty-eight black clues dominate the grid with clue values ranging from 1 to 4; the remaining 53 white cells accommodate the unique light placement. CP-SAT returns the solution in about 3 milliseconds.

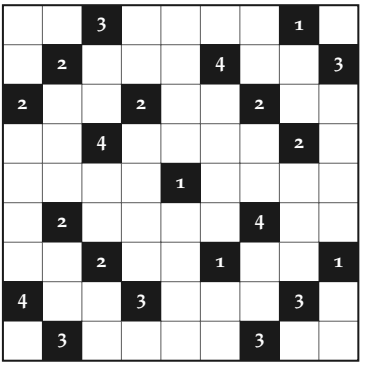

Figure 23.3: *A* 9×9 *Searchlights* (*Bellos*, Puzzle Ninja, *puzzle* 4).

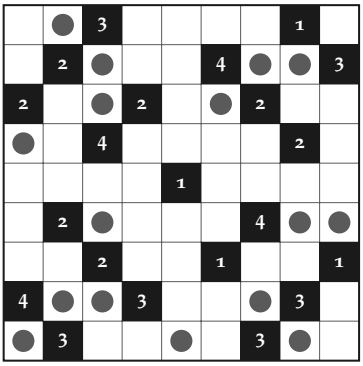

Figure 23.4: *The unique solution, recovered in about 3 milliseconds.*

Sources. Searchlights appears in Alex Bellos's *Puzzle Ninja: Pit Your Wits Against the Japanese Puzzle Masters* (Guardian Faber, 2017); the toy of Figure 23.1 is a small instance of the same family. The line-of-sight primitive is shared with Akari (Chapter 11, *Bijutsukan*) and with Leah Goldberg's academic puzzle *Art Gallery* that underlies the NP-completeness proofs for visibility-based pencil puzzles. Searchlights, unlike Akari, has no uniqueness-of-light constraint inside a line: multiple lights may share a ray, and only their count against a clue matters.

Chapter 24

Numbrix

Numbrix is the puzzle of Hamiltonian numbering. The grid is an $N \times N$ square in which every cell receives a distinct integer from 1 to N^2. The solver must arrange the numbers so that consecutive values are orthogonally adjacent: for every $k \in \{1, 2, \ldots, N^2 - 1\}$, the cells holding k and $k+1$ share an edge (never a corner). Some cells are pre-filled as clues, typically along the border, and the solver threads the single Hamiltonian path through the grid that respects every clue.

Numbrix was introduced by Marilyn vos Savant in her *Parade* column in 2008; the name is a portmanteau of "number" and "matrix." Structurally it is a rigid cousin of the much older Hidato (also known as Hidoku) and of the sliding-tile 15-puzzle. The defining feature is the orthogonal-only neighbour rule: unlike Hidato, which admits diagonal adjacency, Numbrix threads a strict rook's path, and the strictness makes every clue near the corners exceptionally informative.

Among the solver-friendly puzzles in this volume, Numbrix is the first that asks the engine to reason about a *permutation* of N^2 distinct values rather than a fill of small integers. The constraint model mirrors the path-finding structure literally: each value k gets a position variable, and consecutive-value positions are forced to be one unit apart in Manhattan distance.

✠

Rules and a small instance

A Numbrix puzzle is an $N \times N$ grid of cells, some of which carry pre-filled positive-integer clues between 1 and N^2. A solution assigns to every cell a distinct integer from 1 to N^2 such that:

1. Every clue is preserved.
2. Every value in $\{1, 2, \ldots, N^2\}$ appears in exactly one cell.
3. For every $k \in \{1, 2, \ldots, N^2 - 1\}$, the cells holding k and $k + 1$ are orthogonal neighbours (share a full edge).

Figure 24.1 is a 5×5 Numbrix with three clues at corners and at the centre, forcing a snake-shaped Hamiltonian path to thread the entire grid.

1				
		13		
				25

Figure 24.1: *A 5×5 Numbrix with three clues.*

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

Figure 24.2: *The unique solution. Consecutive values are orthogonally adjacent; together they thread a Hamiltonian path.*

✠

The programming model

The natural encoding inverts the usual "value per cell" framing. Instead of assigning to each cell a value, we assign to each value a *position*, and enforce adjacency on consecutive positions.

For each $k \in \{1, 2, \dots, N^2\}$ let $p_k \in \{0, 1, \dots, N^2 - 1\}$ be the linear index of the cell containing k, so that row $r_k = \lfloor p_k / N \rfloor$ and column $c_k = p_k \bmod N$.

Distinctness

All values occupy distinct cells:

$$\text{AllDifferent}(p_1, p_2, \dots, p_{N^2}).$$

Clue preservation

For every pre-filled cell (r, c) with clue k,

$$p_k \;=\; r \cdot N + c.$$

Consecutive adjacency

For every $k \in \{1, 2, \dots, N^2 - 1\}$,

$$|r_k - r_{k+1}| + |c_k - c_{k+1}| \;=\; 1.$$

The Manhattan-distance-equals-one constraint is CP-SAT-native via `AddAbsEquality` on each of $r_k - r_{k+1}$ and $c_k - c_{k+1}$, followed by a linear sum.

A solver in thirty-five lines

```
from ortools.sat.python import cp_model

def solve_numbrix(board):
    """board[r][c] = 0 for empty, positive int for clue.
       Fills board with a Hamiltonian numbering 1..N^2."""
    N = len(board); T = N * N
    m = cp_model.CpModel()
```

```
    pos = {k: m.NewIntVar(0, T - 1, "")
           for k in range(1, T + 1)}
    m.AddAllDifferent(list(pos.values()))
    for r in range(N):
        for c in range(N):
            if board[r][c] != 0:
                m.Add(pos[board[r][c]] == r * N + c)
    for k in range(1, T):
        r1 = m.NewIntVar(0, N - 1, "")
        c1 = m.NewIntVar(0, N - 1, "")
        r2 = m.NewIntVar(0, N - 1, "")
        c2 = m.NewIntVar(0, N - 1, "")
        m.AddDivisionEquality(r1, pos[k], N)
        m.AddModuloEquality (c1, pos[k], N)
        m.AddDivisionEquality(r2, pos[k+1], N)
        m.AddModuloEquality (c2, pos[k+1], N)
        adr = m.NewIntVar(0, N - 1, "")
        adc = m.NewIntVar(0, N - 1, "")
        m.AddAbsEquality(adr, r1 - r2)
        m.AddAbsEquality(adc, c1 - c2)
        m.Add(adr + adc == 1)
    s = cp_model.CpSolver()
    s.Solve(m)
    grid = [[0] * N for _ in range(N)]
    for k in range(1, T + 1):
        p = s.Value(pos[k])
        grid[p // N][p % N] = k
    return grid
```

The toy of Figure 24.1 solves uniquely in about 50 milliseconds.

9		**11**		**19**		**77**		**73**
7								**71**
31								**67**
35								**57**
37		**41**		**45**		**47**		**55**

Figure 24.3: *A* 9 × 9 *Numbrix with* 17 *clues.*

9	10	**11**	18	**19**	78	**77**	74	**73**
8	13	12	17	20	79	76	75	72
7	14	15	16	21	80	81	70	**71**
6	5	4	3	22	63	64	69	68
31	30	29	2	23	62	65	66	**67**
32	33	28	1	24	61	60	59	58
35	34	27	26	25	50	51	52	**57**
36	39	40	43	44	49	48	53	56
37	38	**41**	42	**45**	46	**47**	54	**55**

Figure 24.4: *The unique Hamiltonian numbering, recovered in about* 14 *milliseconds. Values* 1 *through* 81 *trace a single rook's path visiting every cell once.*

A hard instance

Figure 24.3 is a 9×9 Numbrix with clues along the perimeter and the cardinal axes, 17 clues in total. CP-SAT returns the unique solution in about 14 milliseconds.

Sources. Numbrix was created by Marilyn vos Savant for her *Parade* column in 2008 and has since been syndicated

in many English-language newspapers. The hard instance of Figure 24.3 is a classic 9×9 harvested from the *Parade* archive. The puzzle is closely related to Hidato (Gyora Benedek, 2008), which relaxes the orthogonal adjacency rule to king-move adjacency; both are NP-complete to decide (Katagiri et al., 2009), since they embed Hamiltonian-path problems. Although Numbrix is not of Japanese origin, its structural kinship to Masyu and its amenability to the same CP-SAT machinery earn it a place in this volume.

Chapter 25

3-in-a-Row

The 3-in-a-Row puzzle is the sibling of Takuzu. The grid is a square with an even side length N. The solver fills every cell with one of two symbols, conventionally a filled square and an open circle, subject to three rules: each row and each column contains exactly $N/2$ of each symbol, and no three consecutive cells in a row or column share the same symbol. Some cells are pre-filled as clues.

The rule set is almost identical to Takuzu (Chapter 5), and for most puzzles the two are interchangeable. The one distinction is that Takuzu adds a fourth rule, that all rows are mutually distinct and all columns are mutually distinct; 3-in-a-Row omits it. In practice the distinction rarely matters: a typical hand-designed instance is uniquely solvable under the three rules alone, because the balance and no-three constraints together are very sharp once a handful of clues are placed.

3-in-a-Row circulates widely in the English-language puzzle press as a beginner-friendly pencil puzzle, but its deeper lineage traces through Japanese *Tentai Show* and allied variants that appeared in *Puzzle Communication Nikoli* during the 2000s. In the constraint-programming setting, the puzzle is one of the purest instances of a linear-sum-plus-window-constraint encoding: every rule reduces to a small integer-programming statement, and the CP-SAT solver handles an entire grid in a few milliseconds.

Rules and a small instance

A 3-in-a-Row puzzle is an $N \times N$ grid of cells, with N even. Some cells carry pre-filled symbols drawn from $\{0, 1\}$ (conventionally 1 rendered as a filled square, 0 as an open circle). A solution assigns a symbol to every cell such that:

1. Every pre-filled clue is preserved.
2. Each row contains exactly $N/2$ zeros and $N/2$ ones.
3. Each column contains exactly $N/2$ zeros and $N/2$ ones.
4. No three consecutive cells in any row have the same symbol.
5. No three consecutive cells in any column have the same symbol.

Figure 25.1 is a 6×6 3-in-a-Row with eight clues; the solution is unique.

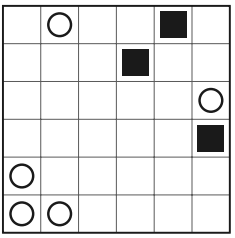

Figure 25.1: *A 6 × 6 3-in-a-Row with eight clues. Filled squares are 1-clues, circles are 0-clues.*

The programming model

For every cell (r, c) introduce a Boolean $x_{r,c} \in \{0, 1\}$.

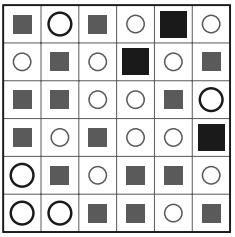

Figure 25.2: *The unique solution. Bold glyphs are clues; the lighter filled squares and smaller circles are solver-supplied.*

Row and column balance

For each row $r \in \{0, 1, \ldots, N-1\}$,

$$\sum_{c=0}^{N-1} x_{r,c} \;=\; \frac{N}{2}.$$

Symmetrically for each column c,

$$\sum_{r=0}^{N-1} x_{r,c} \;=\; \frac{N}{2}.$$

No three consecutive same

For each row r and each starting column $c \in \{0, 1, \ldots, N-3\}$, the three cells $(r, c), (r, c+1), (r, c+2)$ cannot all be 0 and cannot all be 1. Since each variable is Boolean, the sum is in $\{0, 1, 2, 3\}$, and the rule is exactly

$$1 \;\le\; x_{r,c} + x_{r,c+1} + x_{r,c+2} \;\le\; 2.$$

Symmetrically for every column and starting row.

Clue preservation

For every pre-filled cell (r, c) with clue k,

$$x_{r,c} \;=\; k.$$

A solver in twenty-five lines

```
from ortools.sat.python import cp_model

def solve_threeinarow(N, clues):
    """N x N grid (N even); clues = list of (r, c, v)."""
    m = cp_model.CpModel()
    x = [[m.NewBoolVar("") for _ in range(N)]
         for _ in range(N)]
    for r, c, v in clues:
        m.Add(x[r][c] == v)
    for r in range(N):
        m.Add(sum(x[r]) == N // 2)
    for c in range(N):
        m.Add(sum(x[r][c] for r in range(N))
              == N // 2)
    for r in range(N):
        for c in range(N - 2):
            s = x[r][c] + x[r][c+1] + x[r][c+2]
            m.Add(1 <= s); m.Add(s <= 2)
    for c in range(N):
        for r in range(N - 2):
            s = x[r][c] + x[r+1][c] + x[r+2][c]
            m.Add(1 <= s); m.Add(s <= 2)
    s = cp_model.CpSolver()
    s.Solve(m)
    return [[s.Value(x[r][c]) for c in range(N)]
            for r in range(N)]
```

The toy of Figure 25.1 solves uniquely in about 15 milliseconds.

A hard instance

Figure 25.3 is a 10×10 3-in-a-Row with twenty clues spread across the grid. CP-SAT returns the unique solution in about 5 milliseconds.

Sources. 3-in-a-Row appears under various names in the English-language puzzle press: *Binairo*, *Tango*,

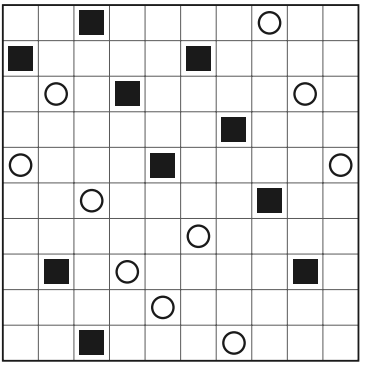

Figure 25.3: *A* 10 × 10 *3-in-a-Row with twenty clues.*

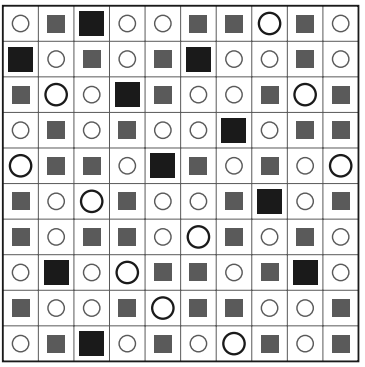

Figure 25.4: *The unique fill, recovered in about* 5 *milliseconds.*

Takuzu, and *Binary Puzzle* are all cognate. The closest Japanese-published form is Takuzu (Chapter 5), published in *Puzzle Communication Nikoli*, which adds the row-and-column uniqueness rule; the version here is the lightly-rule-set variant that omits uniqueness. Both versions are NP-complete to decide (Pedro, 2013, treated the Takuzu version directly; the same reduction works for 3-in-a-Row by dropping the uniqueness constraint). For this chapter, the toy of Figure 25.1 is the small benchmark distributed with the *Solving Logic Puzzles* Python tooling; the hard instance of Figure 25.3 was generated to test CP-SAT's propagation on a scale-up of the same rule system.