Solving Puzzles through Mathematical Programming

Classical puzzles modelled and solved

Vamshi Jandhyala

Vamshi Jandhyala
London

Contents

Chapter 1

The Fish Puzzle

Five houses stand in a row. Each is painted a different colour. Each is home to a person of a different nationality. Each person owns one distinct pet, smokes one distinct brand of cigar, and drinks one distinct beverage. A list of fifteen clues pins down every attribute of every house except one: the ownership of the fish. The solver is asked to determine who owns the fish.

The Fish puzzle, also known as *Einstein's Riddle* and sometimes *the Zebra Puzzle*, is a classical exercise in constraint satisfaction. It has been attributed (without firm evidence) to Albert Einstein, to Lewis Carroll, and to a *Life International* puzzle compiler writing in December 1962; the earliest published form is the 1962 version, naming a zebra rather than a fish. The riddle is deceptively simple: the clues are short, the domain is only five houses, and yet the combinatorial structure is tight enough that the solution is unique.

Unlike the grid puzzles of Nikoli, Fish is a pure *assignment* puzzle: there is no spatial board, only a one-dimensional row of houses indexed 1 through 5, and five attribute categories each drawn from a size-5 alphabet. Each of the fifteen clues is a crisp logical predicate on one or two attribute values. The solver's task is to find the single assignment of 25 attribute values to 25 attribute slots that satisfies every clue.

The CP-SAT encoding is exceptionally clean: each attribute value gets an integer variable in $\{0, 1, 2, 3, 4\}$ (representing the house index at which it sits), the five attribute categories each carry an AllDifferent constraint, and every clue reduces

to either an equality, an order relation, or a Manhattan-1 adjacency constraint between two such variables.

Rules and a small instance

A Fish puzzle consists of n houses in a row, each carrying k attributes drawn from k non-intersecting domains of size n each (in the classical puzzle, $n = k = 5$). A solution assigns to each attribute value one of the n house indices such that:

1. Within each attribute category, the n values map to the n house indices bijectively.
2. Every clue is satisfied.

In the classical instance the attribute categories are *nationality*, *house colour*, *pet*, *cigar brand*, and *beverage*. The fifteen clues are:

1. The Brit lives in the red house.
2. The Swede keeps dogs as pets.
3. The Dane drinks tea.
4. The green house is immediately to the left of the white house.
5. The green house owner drinks coffee.
6. The person who smokes Pallmall rears birds.
7. The owner of the yellow house smokes Dunhill.
8. The person in the centre house drinks milk.
9. The Norwegian lives in the first house.
10. The Blends smoker lives next to the cat owner.
11. The horse owner lives next to the Dunhill smoker.
12. The Bluemaster smoker drinks beer.
13. The German smokes Prince.

14. The Norwegian lives next to the blue house.
15. The Blends smoker has a neighbour who drinks water.

The unique solution is shown in Figure 1.1.

	H1	*H2*	*H3*	*H4*	*H5*
Nationality	Norwegian	Dane	Brit	German	Swede
Colour	Yellow	Blue	Red	Green	White
Pet	Cat	Horse	Bird	**Fish**	Dog
Cigar	Dunhill	Blends	Pallmall	Prince	Bluemaster
Beverage	Water	Tea	Milk	Coffee	Beer

Figure 1.1: *The unique solution to the Fish puzzle. The German, in the green house, owns the fish.*

The German owns the fish.

The programming model

The tightest encoding gives each attribute value the integer variable representing its house index. With $n = 5$ houses and 5 categories of 5 values each, the model has 25 integer variables, each with domain $\{0, 1, 2, 3, 4\}$.

Domain bijection per category

For each attribute category A (nationality, colour, pet, cigar, beverage), let $A_1, A_2, \ldots, A_5$ denote its five values as integer variables. Then

$$\text{AllDifferent}(A_1, A_2, A_3, A_4, A_5)$$

enforces that the five values occupy the five distinct houses.

Clue forms

The fifteen clues partition into three syntactic classes.

Identity clues. Most clues assert that two attribute values share a house. For example, clue 1 ("the Brit lives in the red house") becomes

$$\mathrm{Nat}[\mathrm{Brit}] \;=\; \mathrm{Col}[\mathrm{Red}].$$

Clues $1, 2, 3, 5, 6, 7, 12, 13$ are of this form; so too are clues 8 and 9 ("centre house" and "first house" are specific indices, encoded as $\mathrm{Bev}[\mathrm{Milk}] = 2$ and $\mathrm{Nat}[\mathrm{Norwegian}] = 0$).

Order clue. Clue 4 says the green house is immediately to the left of the white house. In the integer-variable encoding this is simply

$$\mathrm{Col}[\mathrm{Green}] + 1 \;=\; \mathrm{Col}[\mathrm{White}].$$

Adjacency clues. Clues $10, 11, 14, 15$ are of the form "X lives next to Y," i.e. their house indices differ by exactly one. For clue 10 ("the Blends smoker lives next to the cat owner"):

$$|\mathrm{Cig}[\mathrm{Blends}] - \mathrm{Pet}[\mathrm{Cat}]| \;=\; 1,$$

encoded in CP-SAT via `AddAbsEquality` on the difference.

A solver in forty lines

```
from ortools.sat.python import cp_model

NATS = ["Norwegian", "Dane", "Brit", "German", "Swede"]
COLS = ["Yellow", "Blue", "Red", "Green", "White"]
PETS = ["Cat", "Horse", "Bird", "Fish", "Dog"]
CIGS = ["Dunhill", "Blends", "Pallmall", "Prince",
        "Bluemaster"]
BEVS = ["Water", "Tea", "Milk", "Coffee", "Beer"]

def solve_fish():
    m = cp_model.CpModel()
    def attrs(names):
        return {n: m.NewIntVar(0, 4, n) for n in names}
    nat = attrs(NATS); col = attrs(COLS)
    pet = attrs(PETS); cig = attrs(CIGS); bev = attrs(BEVS)
    for d in (nat, col, pet, cig, bev):
        m.AddAllDifferent(list(d.values()))
    # Identity clues
```

```
    m.Add(nat["Brit"]   == col["Red"])        # 1
    m.Add(nat["Swede"]  == pet["Dog"])        # 2
    m.Add(nat["Dane"]   == bev["Tea"])        # 3
    m.Add(col["Green"]  + 1 == col["White"])  # 4
    m.Add(col["Green"]  == bev["Coffee"])     # 5
    m.Add(cig["Pallmall"] == pet["Bird"])     # 6
    m.Add(col["Yellow"] == cig["Dunhill"])    # 7
    m.Add(bev["Milk"]   == 2)                 # 8
    m.Add(nat["Norwegian"] == 0)              # 9
    m.Add(cig["Bluemaster"] == bev["Beer"])   # 12
    m.Add(nat["German"] == cig["Prince"])     # 13
    # Adjacency clues (|a - b| == 1)
    def adj(a, b):
        d = m.NewIntVar(-4, 4, "")
        ab = m.NewIntVar(0, 4, "")
        m.Add(d == a - b)
        m.AddAbsEquality(ab, d)
        m.Add(ab == 1)
    adj(cig["Blends"], pet["Cat"])            # 10
    adj(pet["Horse"],  cig["Dunhill"])        # 11
    adj(nat["Norwegian"], col["Blue"])        # 14
    adj(cig["Blends"], bev["Water"])          # 15
    s = cp_model.CpSolver()
    s.Solve(m)
    # Invert: house index -> attribute values
    out = [[""]*5 for _ in range(5)]
    for row, d in enumerate((nat, col, pet, cig, bev)):
        for name, var in d.items():
            out[row][s.Value(var)] = name
    return out
```

The puzzle solves uniquely in about 25 milliseconds. Enumerating all feasible assignments (via `parameters.enumerate_all_solutions = True`) confirms that precisely one assignment satisfies the fifteen clues.

Sources. The Fish puzzle first appeared in *Life International* in December 1962 in its Zebra form (the unknown pet is a zebra; the magazine framed the puzzle as an attribution to Einstein, though no evidence supports this). The Fish variant circulates widely online and in logic-puzzle anthologies. The fifteen-clue formulation used here is the standard modern version; the uniqueness of solution hinges

on the strict "immediately to the left" reading of clue 4 (a relaxed "somewhere to the left" reading admits seven distinct assignments). Einstein's Riddle is in **P** for fixed $n = 5$ and $k = 5$, but the corresponding n-house, k-attribute generalisation with $O(nk)$ clues is NP-complete by a direct reduction from 3-SAT.

Chapter 2

The Calendar Puzzle

The Calendar Puzzle is the physical puzzle of fitting eight pieces around a date. A 7×7 wooden board is laid out with the twelve months on its top two rows, the days 1 through 31 on its lower five rows, and nine permanently blocked cells at the corners. Every morning, the user chooses a month and a day, sets those two cells aside, and tiles the remaining 41 cells with eight polyominoes (seven pentominoes and one hexomino) to leave only the chosen month and day showing. The puzzle ships under the brand name *A-Puzzle-A-Day*; other calendar puzzles with minor variations (pentomino sets, day arrangements) are sold under *DragonFjord* and *Calendarium*. Our target is the original 7×7 layout.

The puzzle is simple to describe and striking to solve. Of the 366 possible (month, day) combinations, the vast majority admit at least one solution, but the number of solutions per date varies widely. Some dates have dozens of tilings; others have only one or two; a few have none (those at the boundary of the board, where the constraint structure is tightest). The puzzle rewards both manual search (useful when solving once a day, by hand, with the physical pieces) and constraint programming (useful for auditing the board, counting solutions per date, or generating a printable picture).

Among the puzzles in this volume, the Calendar Puzzle is the cleanest example of an *exact cover* problem in disguise. The board is a fixed region, two cells are subtracted per day, and

the eight polyominoes must cover the remaining cells exactly. No numeric clues, no adjacency rules, no colour constraints: only the geometry of the pieces and the geometry of the board. This makes the CP-SAT encoding extremely compact and the solver extremely fast.

The board and the pieces

The physical board is a 7×7 array of cells, of which four positions are permanently blocked: cells $(0, 6)$ and $(1, 6)$ on the top two rows (which carry the twelve months) and cells $(6, 3)$ through $(6, 6)$ on the bottom row (which carries days 29, 30, 31 and then tails off). The remaining 43 cells carry labels: the twelve months {Jan, ..., Dec} and the thirty-one days $\{1, \ldots, 31\}$. A date is chosen by designating one month cell and one day cell as the "reveal", leaving 41 cells to be covered.

The eight pieces are seven pentominoes (five cells each) and one hexomino (six cells), totalling $7 \cdot 5 + 6 = 41$ cells. This is exactly the number of cells to cover, no cells to spare. Each piece may be rotated by 0°, 90°, 180°, 270° and reflected (flipped over), giving up to eight orientations per piece; some pieces with internal symmetry have fewer distinct orientations.

Figure 2.1 is the Calendar Puzzle for 24 April, and Figure 2.2 shows a valid tiling.

The programming model

The exact-cover formulation is direct. For each piece P_i ($i = 1, 2, \ldots, 8$), enumerate all orientations (rotations and

Figure 2.1: *The Calendar Puzzle for 24 April: the month* Apr *and the day* 24 *are highlighted (to be left uncovered by the pieces). Grey cells at corners are permanently blocked.*

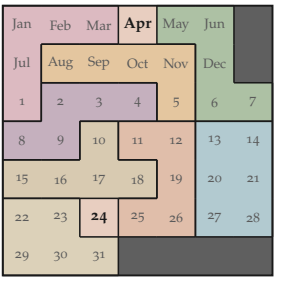

Figure 2.2: *One valid tiling. The eight pieces are distinguished by colour; the highlighted* Apr *and* 24 *remain uncovered.*

reflections) and, for each orientation, all translations that fit inside the 7×7 board. Each (piece, orientation, translation) triple defines a *placement*: a set of cells on the board that the piece would cover if the solver selects it.

Placement variables

For each placement p associated with piece P_i, introduce a Boolean variable $x_p \in \{0, 1\}$ with $x_p = 1$ meaning "this placement of piece P_i is selected."

One placement per piece

For every piece P_i, exactly one of its placements is selected:

$$\sum_{p\,:\,\text{placement of } P_i} x_p = 1.$$

Exact cover per cell

For every cell (r, c) of the board with (r, c) in the set of cells to be covered (i.e. not blocked and not equal to the chosen month or day), exactly one placement covers it:

$$\sum_{p\,:\,(r,c)\in\text{cells}(p)} x_p \;=\; 1.$$

For every cell that must *not* be covered (the chosen month, the chosen day, and the four blocked positions), the same sum is required to be zero:

$$\sum_{p\,:\,(r,c)\in\text{cells}(p)} x_p \;=\; 0.$$

The two families can be unified as a single constraint

$$\sum_{p} x_p \cdot \mathbb{1}\big[(r, c) \in \text{cells}(p)\big] \;=\; B_{r,c},$$

where $B_{r,c}$ is 1 for a must-cover cell and 0 otherwise.

A solver in sixty lines

```
import numpy as np
from ortools.sat.python import cp_model

PIECES = [
    [(0,0),(0,1),(0,2),(1,2),(0,3)],
    [(0,0),(0,1),(0,2),(1,2),(2,2)],
    [(0,0),(0,1),(1,0),(2,0),(2,1)],
    [(0,2),(1,0),(1,1),(1,2),(2,0)],
    [(0,0),(1,0),(2,0),(0,1),(1,1),(2,1)],  # hexomino
    [(0,0),(1,0),(2,0),(2,1),(3,1)],
    [(0,0),(0,1),(0,2),(0,3),(1,0)],
    [(0,1),(1,0),(1,1),(2,0),(2,1)],
]

def rotations(p):
    out = [p]
    for _ in range(3):
        q = [(c, -r) for r, c in out[-1]]
        mr = min(r for r, _ in q)
        mc = min(c for _, c in q)
        out.append([(r - mr, c - mc) for r, c in q])
    return out
```

```
def reflections(p):
    mir = [(r, -c) for r, c in p]
    mc = min(c for _, c in mir)
    return [(r, c - mc) for r, c in mir]

def placements(piece, H=7, W=7):
    """All placements fitting the board."""
    seen, out = set(), []
    orients = (rotations(piece)
               + [reflections(r)
                  for r in rotations(piece)])
    for o in orients:
        for dr in range(H):
            for dc in range(W):
                pl = [(r + dr, c + dc) for r, c in o]
                if any(r >= H or c >= W for r, c in pl):
                    continue
                k = frozenset(pl)
                if k in seen: continue
                seen.add(k); out.append(pl)
    return out

BLOCKED = ((0, 6), (1, 6),
           (6, 3), (6, 4), (6, 5), (6, 6))

def solve_calendar(month_cell, day_cell):
    m = cp_model.CpModel()
    B = np.ones((7, 7), dtype=int)
    for r, c in BLOCKED:
        B[r, c] = 0
    B[month_cell] = 0
    B[day_cell] = 0
    all_pl, pv = [], []
    for piece in PIECES:
        pc = []
        for pl in placements(piece):
            v = m.NewBoolVar("")
            pc.append((v, pl))
            all_pl.append((v, pl))
        m.Add(sum(v for v, _ in pc) == 1)
        pv.append(pc)
    for r in range(7):
        for c in range(7):
            cov = [v for v, pl in all_pl
                     if (r, c) in pl]
            m.Add(sum(cov) == int(B[r, c]))
    s = cp_model.CpSolver()
    s.Solve(m)
```

```
    return [(pl, i)
            for i, pc in enumerate(pv)
            for v, pl in pc if s.Value(v)]
```

Each date solves in about 90 milliseconds on a standard laptop. A nightly batch that emits the solution for every one of the 366 possible dates completes in under a minute.

A hard instance: 29 February

The leap-day date 29 February is interesting because it brings together two boundary cells: February in the top row and the 29 cell in the bottom-left corner of the day block, diagonally far apart. Both cells are near the board's geometric extremes, which constrains the tiling more than an interior choice would.

Jan	Feb	Mar	Apr	May	Jun	
Jul	Aug	Sep	Oct	Nov	Dec	
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

Figure 2.3: *The Calendar Puzzle for 29 February.*

Sources. *A-Puzzle-A-Day* is a commercial wooden puzzle sold by DragonFjord Puzzles and several independent manufacturers, with 7×7 variants dating from approximately 2018. The puzzle reduces to classical polyomino packing, a topic with a long history in recreational mathematics running from Solomon Golomb's 1954 pentomino coinage through

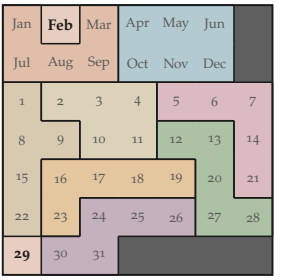

Figure 2.4: *One valid tiling for 29 February, recovered in about 95 milliseconds.*

Dana Scott's exhaustive enumeration of pentomino tilings of an 8×8 square minus the centre 2×2 in 1958. Our solver enumerates approximately 2000 placements per board (the product of eight pieces, up to eight orientations, and several dozen translations) and CP-SAT handles the resulting exact cover in a fraction of a second.

Chapter 3

The Praxis Rhombus

The Praxis Rhombus is a calendar puzzle with an extra axis. Where the ordinary A-Puzzle-A-Day board carries only a month and a day, the Rhombus board also carries the seven weekdays, and the solver must leave the weekday, the month, and the day uncovered. The board is a diamond of 50 labelled cells — seven weekdays in the upper-left wedge, twelve months along a lower diagonal, and the thirty-one days filling the interior — and the pieces are ten polyominoes of total area 47, exactly matching the 50 cells minus the three that must be revealed.

The puzzle exists because a single additional degree of freedom multiplies the number of distinct daily configurations. The 12×31 grid of (month, day) pairs covers 366 cases; the $7 \times 12 \times 31$ grid of (weekday, month, day) triples covers 2 604, though only about a third of these occur in any given year. The tiling problem grows correspondingly: instead of subtracting two cells from a rectangular board, the solver subtracts three from a diamond.

The Praxis Rhombus sits in the same family as the Calendar Puzzle but requires a slightly different model. The board is not rectangular; off-board cells form a staircase along each of the four diagonal boundaries; and the piece set has three tetrominoes mixed in among the pentominoes, so the area-accounting ($3 \cdot 4 + 7 \cdot 5 = 47$) is different. Everything else reduces, as before, to exact cover.

The board and the pieces

Embed the rhombus in an 8×8 rectangular array and mark fourteen corner cells as off-board. The remaining fifty cells carry labels: Mon, Tue, ..., Sun in the upper-left; Jan, Feb, ..., Dec along a diagonal band; and the numerals $1, 2, \ldots, 31$ occupying the interior. A date is chosen by designating one weekday, one month, and one numeral as the *reveal*, leaving 47 cells to be tiled.

The ten pieces are three tetrominoes (four cells each) and seven pentominoes (five cells each), for a total of $3 \cdot 4 + 7 \cdot 5 = 47$ cells. As in any polyomino puzzle, each piece may be rotated through 0°, 90°, 180°, 270° and reflected; pieces with internal symmetry have fewer distinct orientations. Two of the three tetrominoes are the familiar L and T; the third is the S (or Z) tetromino. The seven pentominoes are a subset of the twelve named Golomb pieces.

Figure 3.1 is the puzzle for Thursday 23 October and Figure 3.2 shows one tiling.

	Mon	Wed	5	1			
Tue	**Thu**	Sat	10	6	2		
Fri	Sun	15	13	11	7	3	
23	20	18	16	14	12	8	4
28	24	21	19	17	Jan	Mar	9
	29	25	22	Feb	Apr	Jun	Aug
		30	26	May	Jul	Sep	Nov
			31	27	**Oct**	Dec	

Figure 3.1: *The Praxis Rhombus for Thursday 23 October: the three highlighted cells* Thu, Oct, *and* 23 *are to be left uncovered.*

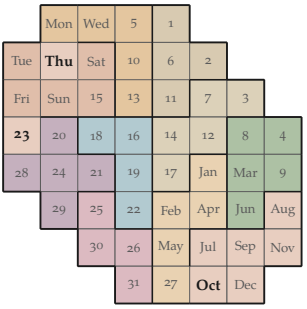

Figure 3.2: *One valid tiling. The ten pieces are distinguished by colour; the three reveal cells remain uncovered.*

The programming model

The formulation is exact cover on the fifty on-board cells. For each piece P_i $(i = 1, \ldots, 10)$, enumerate its distinct orientations and, for each orientation, every translation whose cells all land on the board and none of which coincide with a reveal. Each surviving (piece, orientation, translation) triple is a *placement*.

Placement variables

For each placement p of piece P_i, introduce a Boolean variable x_p with $x_p = 1$ meaning "this placement is selected."

One placement per piece

Each of the ten pieces is placed exactly once:

$$\sum_{p\,:\,\text{placement of } P_i} x_p = 1.$$

Exact cover per cell

For every on-board cell (r, c) that is not a reveal, exactly one selected placement covers it:

$$\sum_{p\,:\,(r,c)\in\text{cells}(p)} x_p = 1.$$

Off-board cells and reveal cells are excluded from placement enumeration, so no such constraint is needed for them.

A solver in seventy lines

```
from ortools.sat.python import cp_model

BOARD = [
    ['',    'Mon', 'Wed', '5',  '1',  '',    '',    ''],
    ['Tue', 'Thu', 'Sat', '10', '6',  '2',   '',    ''],
    ['Fri', 'Sun', '15',  '13', '11', '7',   '3',   ''],
    ['23',  '20',  '18',  '16', '14', '12',  '8',   '4'],
    ['28',  '24',  '21',  '19', '17', 'Jan', 'Mar', '9'],
    ['',    '29',  '25',  '22', 'Feb','Apr', 'Jun', 'Aug'],
    ['',    '',    '30',  '26', 'May','Jul', 'Sep', 'Nov'],
    ['',    '',    '',    '31', '27', 'Oct', 'Dec', ''],
]

PIECES = [
    [(0,0),(0,1),(1,1),(1,2)],              # S-tetromino
    [(0,1),(1,0),(1,1),(1,2),(2,1)],        # plus
    [(0,0),(0,1),(1,0),(1,1),(2,1)],        # P
    [(0,0),(0,1),(0,2),(1,0),(2,0)],        # V
    [(0,0),(0,1),(0,2),(1,0),(1,2)],        # U
    [(0,0),(0,1),(0,2),(1,0)],              # L-tetromino
    [(0,0),(0,1),(0,2),(1,1)],              # T-tetromino
    [(0,0),(1,0),(1,1),(2,1),(2,2)],        # W
    [(0,2),(1,0),(1,1),(1,2),(2,1)],        # F
    [(0,0),(0,1),(1,1),(1,2),(1,3)],        # N
]

def norm(p):
    mr = min(r for r, _ in p)
    mc = min(c for _, c in p)
    return tuple(sorted((r - mr, c - mc) for r, c in p))

def orients(p):
    out, cur = set(), list(p)
    for _ in range(4):
        out.add(norm(cur))
        cur = [(c, -r) for r, c in cur]
    cur = [(r, -c) for r, c in p]
    for _ in range(4):
        out.add(norm(cur))
        cur = [(c, -r) for r, c in cur]
    return [list(o) for o in out]

def solve(dow, mon, day):
```

```
    H, W = 8, 8
    off = {(r, c) for r in range(H) for c in range(W)
           if BOARD[r][c] == ''
           or BOARD[r][c] in (dow, mon, day)}
    m = cp_model.CpModel()
    pls = []
    for i, piece in enumerate(PIECES):
        for o in orients(piece):
            for dr in range(H):
                for dc in range(W):
                    cells = [(r+dr, c+dc) for r, c in o]
                    if any(r < 0 or r >= H
                           or c < 0 or c >= W
                           or (r, c) in off
                           for r, c in cells):
                        continue
                    pls.append((i, cells))
    v = [m.NewBoolVar("") for _ in pls]
    for i in range(len(PIECES)):
        m.Add(sum(v[k] for k, p in enumerate(pls)
                  if p[0] == i) == 1)
    for r in range(H):
        for c in range(W):
            if (r, c) in off: continue
            m.Add(sum(v[k] for k, p in enumerate(pls)
                      if (r, c) in p[1]) == 1)
    s = cp_model.CpSolver()
    s.Solve(m)
    return [pls[k] for k, x in enumerate(v) if s.Value(x)]
```

The enumeration phase produces about 1 100 placements per date; CP-SAT selects the ten of them that form a valid tiling in roughly 60 milliseconds on a standard laptop.

✠

A hard instance: Friday 29 February

The leap day falls on a weekday only once every twenty-eight years on average, and the Rhombus board makes the tightest use of that coincidence. Figure 3.3 shows the board for Friday 29 February, with the three reveal cells far apart: *Fri* on the left diagonal, *Feb* in the lower-central band,

and 29 near the bottom-left boundary. All three are near the rhombus's edges, constraining the tiling more than interior choices would.

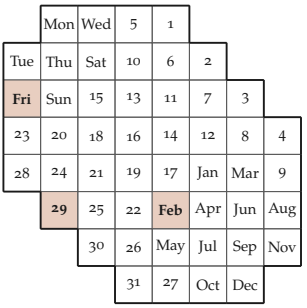

Figure 3.3: *The Praxis Rhombus for Friday 29 February.*

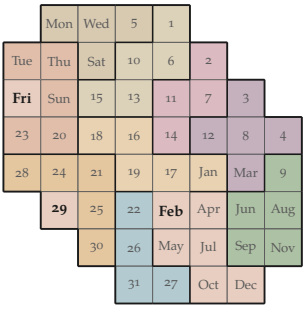

Figure 3.4: *One valid tiling for Friday 29 February, recovered in about 60 milliseconds.*

Sources. The Praxis Rhombus was released in 2023 by an independent designer under the trade name *Praxis Puzzle*; its conceit — adding the weekday as a third selection axis — is a natural extension of the A-Puzzle-A-Day genre. Mathematically, both puzzles are polyomino exact-cover problems on a fixed region with a small reveal set; the Rhombus is distinguished chiefly by its diamond shape, its mixed tetromino/pentomino piece set, and the third

dimension of the reveal. Enumerative counts (how many tilings per date, how many dates admit none) remain an attractive computational curiosity: the solver of the preceding section runs through all 2 604 weekday-month-day triples in well under five minutes and reports a full tiling profile.

Chapter 4

The Bedlam Cube

The Bedlam Cube is a $4 \times 4 \times 4$ polycube dissection. Thirteen pieces — twelve pentacubes and one tetracube, totalling $12 \cdot 5 + 4 = 64$ unit cubes — must be assembled into a solid cube of side four. The piece set was chosen so that no piece repeats (as with the related Soma Cube) and no piece is symmetric enough to simplify the search. The resulting puzzle is far harder than its flat relatives: not because the pieces are subtle, but because the $4 \times 4 \times 4$ cell count is large, each piece has up to twenty-four rotational orientations, and the interactions between adjacent pieces propagate along three axes at once.

The cube has 19 186 distinct solutions, a count first published by Scott Kurowski. For any given tiling, the thirteen pieces can be identified by their canonical letter names (Little Corner, Spikey Zag, Pokey Corner Bit, Right Angles Everywhere, Middle Zig, Pokey Z, Corner Joist, Dented L, F in Therapy, Flat M, Cross, Confused F, T-Bone) and located in the cube by their occupied coordinates. None of the pieces is a repeat of another, so pairs of the form "which copy of piece X" do not arise; the twelve pentacubes are twelve of the twenty-nine distinct pentacubes, and the one tetracube is one of the eight distinct tetracubes.

Exact cover is again the natural formulation. The cube has 64 cells, each piece must be placed exactly once, and every cell must be covered exactly once. What distinguishes this chapter from the two that preceded it is simply the move to three dimensions: the orientation group has 24 elements rather

than 8, and placements are enumerated over translations in three axes rather than two.

The pieces

Each of the thirteen pieces is a connected polycube in the cubic lattice. The canonical coordinates, taken from Scott Kurowski's enumeration, are:

Piece	*Name*	*Coordinates* (x, y, z)
A	Little Corner	$(0,0,0), (0,0,1), (1,0,0), (1,1,0)$
B	Spikey Zag	$(0,0,1), (0,1,0), (0,1,1), (1,1,0), (1,2,0)$
C	Pokey Corner Bit	$(0,0,0), (0,1,0), (1,1,0), (0,1,1), (0,2,0)$
D	Right Angles Everywhere	$(0,0,0), (0,0,1), (0,1,0), (0,2,0), (1,2,0)$
E	Middle Zig	$(0,0,0), (0,1,0), (0,1,1), (1,1,0), (1,2,0)$
F	Pokey Z	$(0,0,0), (0,0,1), (0,1,0), (1,1,0), (1,2,0)$
G	Corner Joist	$(0,0,0), (1,0,0), (0,0,1), (0,1,0), (0,2,0)$
H	Dented L	$(0,0,0), (1,0,0), (1,0,1), (0,1,0), (0,2,0)$
I	F in Therapy	$(0,0,0), (0,0,1), (0,1,0), (1,1,0), (0,2,0)$
J	Flat M	$(0,1,0), (1,0,0), (2,0,0), (1,1,0), (0,2,0)$
K	Cross	$(0,1,0), (1,0,0), (1,1,0), (2,1,0), (1,2,0)$
L	Confused F	$(0,0,0), (1,0,0), (1,1,0), (2,1,0), (1,2,0)$
M	T-Bone	$(0,1,0), (1,0,0), (0,1,1), (1,1,0), (1,2,0)$

Piece A is the one tetracube; B through M are the twelve pentacubes. Five of the pentacubes are planar (they fit into a single layer: pieces J, K, and L are planar; B, C, D, E, F, G, H, I, M all have cells at two different z-coordinates and so are genuinely three-dimensional).

The programming model

Orientation group

In three dimensions, the rotation group of the cube has 24 elements. Given a piece's canonical coordinate list, enumerate its distinct orientations by applying the group generators and de-duplicating; this collapses to fewer than 24 orientations for pieces with rotational symmetry, but most Bedlam pieces produce the full count. Reflections are *not* included: the Bedlam Cube is the chiral puzzle, and adding mirror images would introduce extra "pieces" that do not exist in the physical set.

Placements

For each piece P_i and each orientation o, enumerate every translation (dx, dy, dz) such that all cells of the translated orientation lie inside the $4 \times 4 \times 4$ cube. Each surviving (piece, orientation, translation) is a *placement*.

Placement variables

For each placement p of piece P_i, introduce a Boolean variable $x_p \in \{0, 1\}$ with $x_p = 1$ meaning "this placement is selected."

Constraints

Each piece is placed exactly once:

$$\sum_{p\,:\,\text{placement of } P_i} x_p \;=\; 1.$$

Each of the 64 cells is covered exactly once:

$$\sum_{p\,:\,(x,y,z)\in\text{cells}(p)} x_p \;=\; 1.$$

A solver in eighty lines

```
from ortools.sat.python import cp_model

PIECES = {
    "A": [(0,0,0),(0,0,1),(1,0,0),(1,1,0)],
    "B": [(0,0,1),(0,1,0),(0,1,1),(1,1,0),(1,2,0)],
    "C": [(0,0,0),(0,1,0),(1,1,0),(0,1,1),(0,2,0)],
    "D": [(0,0,0),(0,0,1),(0,1,0),(0,2,0),(1,2,0)],
```

```
    "E": [(0,0,0),(0,1,0),(0,1,1),(1,1,0),(1,2,0)],
    "F": [(0,0,0),(0,0,1),(0,1,0),(1,1,0),(1,2,0)],
    "G": [(0,0,0),(1,0,0),(0,0,1),(0,1,0),(0,2,0)],
    "H": [(0,0,0),(1,0,0),(1,0,1),(0,1,0),(0,2,0)],
    "I": [(0,0,0),(0,0,1),(0,1,0),(1,1,0),(0,2,0)],
    "J": [(0,1,0),(1,0,0),(2,0,0),(1,1,0),(0,2,0)],
    "K": [(0,1,0),(1,0,0),(1,1,0),(2,1,0),(1,2,0)],
    "L": [(0,0,0),(1,0,0),(1,1,0),(2,1,0),(1,2,0)],
    "M": [(0,1,0),(1,0,0),(0,1,1),(1,1,0),(1,2,0)],
}
N = 4

def norm(p):
    mx = min(x for x,_,_ in p)
    my = min(y for _,y,_ in p)
    mz = min(z for _,_,z in p)
    return tuple(sorted((x-mx, y-my, z-mz)
                        for x,y,z in p))

def rotX(p): return [(x,-z,y) for x,y,z in p]
def rotY(p): return [(z,y,-x) for x,y,z in p]
def rotZ(p): return [(-y,x,z) for x,y,z in p]

def orients(p):
    pool = {tuple(norm(p))}
    frontier = set(pool)
    while frontier:
        nxt = set()
        for q in frontier:
            for g in (rotX, rotY, rotZ):
                r = tuple(norm(g(list(q))))
                if r not in pool:
                    pool.add(r); nxt.add(r)
        frontier = nxt
    return [list(o) for o in pool]

def solve():
    m = cp_model.CpModel()
    pls = []
    for name, shape in PIECES.items():
        seen = set()
        for o in orients(shape):
            mx = max(x for x,_,_ in o)
            my = max(y for _,y,_ in o)
            mz = max(z for _,_,z in o)
            for dx in range(N - mx):
                for dy in range(N - my):
                    for dz in range(N - mz):
```

```
                    cells = frozenset(
                        (x+dx, y+dy, z+dz)
                        for x,y,z in o)
                    if cells in seen: continue
                    seen.add(cells)
                    pls.append((name, cells))
    v = [m.NewBoolVar("") for _ in pls]
    by = {}
    for i,(n,_) in enumerate(pls):
        by.setdefault(n, []).append(i)
    for n, idxs in by.items():
        m.Add(sum(v[i] for i in idxs) == 1)
    cov = {}
    for i,(_,cs) in enumerate(pls):
        for c in cs:
            cov.setdefault(c, []).append(i)
    for idxs in cov.values():
        m.Add(sum(v[i] for i in idxs) == 1)
    s = cp_model.CpSolver()
    s.Solve(m)
    return [pls[i] for i,x in enumerate(v)
            if s.Value(x)]
```

The enumeration produces about 4 200 placements; CP-SAT returns a single solution in just over one second.

Reading the solution

The solution is displayed three ways. Figure 4.1 shows four axonometric views of the whole cube, obtained by rotating the cube through 90°, 180° and 270° about its vertical axis: the four views together expose all six outer faces. Figure 4.2 shows each of the thirteen pieces individually in the same coordinate frame, with the outer $4 \times 4 \times 4$ cube rendered as a thin wireframe and only the focus piece drawn in full colour; this makes the spatial position of every piece explicit, including the pieces buried in the interior. Figure 4.3 gives the information-complete view: four z-slices from $z = 0$ (bottom) to $z = 3$ (top), each cell labelled by its piece letter.

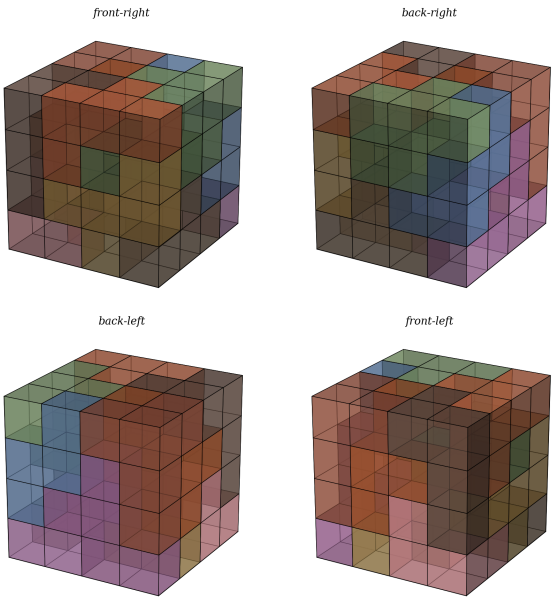

Figure 4.1: *One tiling of the Bedlam Cube rendered as translucent voxels from four viewpoints, each rotated 90° about the vertical axis. Transparency makes the interior pieces visible through the outer layers; between them the four views expose all six outer faces.*

Sources. The Bedlam Cube was designed by Bruce Bedlam in 1979 and manufactured commercially from 1984; it is a close cousin of Piet Hein's Soma Cube (1933) but with a larger, non-repeating piece set. The solution count 19 186 (up to the 24-fold rotational symmetry of the outer cube) is due to Scott Kurowski, whose `scottkurowski.com` enumeration was the first complete census. The solver of the preceding section recovers a single tiling by default; setting `parameters.enumerate_all_solutions = True` on the CP-SAT solver and iterating over feasible models reproduces the Kurowski count, modulo the outer-cube symmetry that the CP model does not break.

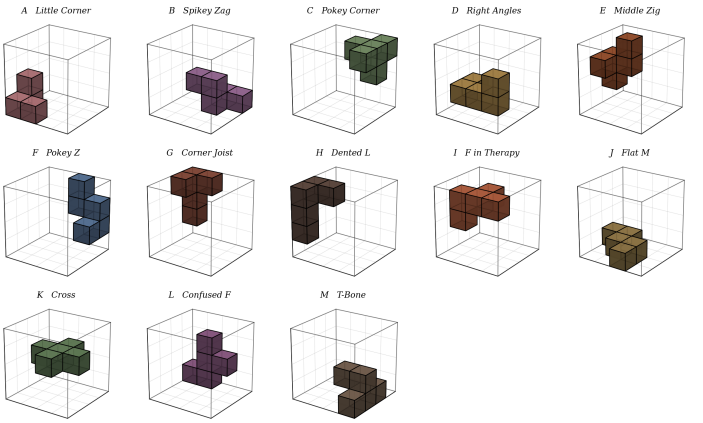

Figure 4.2: *The thirteen pieces shown individually in their positions inside the solved cube, with the outer $4 \times 4 \times 4$ box drawn as a thin wireframe. Pieces buried in the interior (those whose cubes never touch any outer face) become visible here in a way the axonometric views of Figure 4.1 cannot show.*

$z = 0$

B	M	M	M
B	B	J	J
L	J	J	A
L	J	A	A

$z = 1$

F	F	M	D
L	B	M	D
L	B	D	D
G	E	A	H

$z = 2$

F	C	K	D
F	K	K	K
L	E	K	I
G	E	E	H

$z = 3$

C	C	C	I
F	C	I	I
G	E	H	I
G	G	H	H

Figure 4.3: *The same tiling as four z-slices, from $z = 0$ (left) to $z = 3$ (right). Each cell is labelled with its piece letter; adjacency across slices can be verified by comparing labels at matching (x, y) positions.*

Chapter 5

Instant Insanity

Four cubes, each of whose six faces is painted with one of four colours, are given. The solver is asked to stack them on top of one another so that each of the four long sides of the resulting tower shows each of the four colours exactly once. The name *Instant Insanity* is the 1967 trade name of Parker Brothers for what was previously known as *The Tantalizer* and, earlier still, the *Katzenjammer* puzzle. For a commercially reasonable choice of face-colours the puzzle is notoriously difficult by hand — a random stack satisfies the constraint with probability below one in a thousand — but yields immediately to a graph-theoretic argument, and yields even more quickly to a constraint-programming encoding.

Each cube has 24 rotational orientations, so the brute-force search space is $24^4 = 331\,776$ configurations, a handful of which are valid. Under any reasonable symmetry reduction the search space collapses to a few hundred; under the graph-theoretic formulation it collapses to the problem of finding two edge-disjoint 2-regular subgraphs in a four-vertex multigraph on twelve edges.

Rules and the classical instance

Each cube's colouring is specified by three pairs of opposite faces. The Parker Brothers instance fixes the four cubes as

in Figure 5.1: cube 1 has opposite-face pairs (green, white), (blue, white), (blue, red); cube 2 has (blue, blue), (green, white), (red, green); cube 3 has (red, white), (red, green), (blue, white); and cube 4 has (red, red), (green, white), (blue, red). Each cube is rendered as a cross-shaped net (top; left, front, right, back; bottom). Opposite faces of the cube correspond to top/bottom, left/right and front/back in the net.

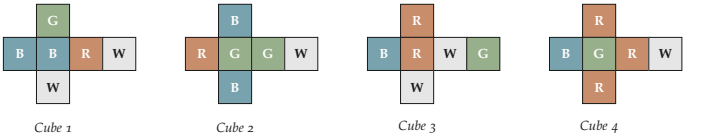

Figure 5.1: *The four cubes of the classical Instant Insanity instance, shown as cross nets. Opposite faces of each cube sit at top/bottom, left/right and front/back of its net.*

A *stack* of the four cubes is a choice of one of the twenty-four rotational orientations for each cube, placing them in order 1, 2, 3, 4 from bottom to top. A stack is a *solution* if, on each of the four visible sides (front, back, left, right) of the stack, the four colours appear exactly once.

The programming model

The encoding is crisper than one might expect. For each cube we do not need to carry the top and bottom face-colours: they are irrelevant to the stack's visible sides. It suffices to enumerate the 24 (front, back, left, right) 4-tuples for each cube and treat them as *allowed assignments* in a table constraint.

Variables

For each cube $i \in \{1, 2, 3, 4\}$ and each side $s \in \{\mathrm{F}, \mathrm{B}, \mathrm{L}, \mathrm{R}\}$, introduce an integer variable $c_{i,s} \in \{0, 1, 2, 3\}$ representing the colour (indexed 0 = red, 1 = green, 2 = blue, 3 = white) of cube i's face s.

Per-cube orientation constraint

For each cube i, enumerate the distinct 4-tuples $(c_{i,\mathrm{F}}, c_{i,\mathrm{B}}, c_{i,\mathrm{L}}, c_{i,\mathrm{R}})$ arising over the 24 rotations and add a table constraint requiring one of them be chosen. In CP-SAT this is `AddAllowedAssignments`.

Per-side all-different constraint

For each side s, the four visible colours $c_{1,s}, c_{2,s}, c_{3,s}, c_{4,s}$ must be all different:

$$\mathrm{AllDifferent}(c_{1,s}, c_{2,s}, c_{3,s}, c_{4,s}).$$

That is the entire model. No separate variables for the 24 orientations are needed; the allowed-assignments table projects the rotational freedom onto the four variables that matter.

A solver in forty lines

```
from ortools.sat.python import cp_model

R, G, B, W = 0, 1, 2, 3
CUBES = [
    [(G,W), (B,W), (B,R)],
    [(B,B), (G,W), (R,G)],
    [(R,W), (R,G), (B,W)],
    [(R,R), (G,W), (B,R)],
]

def orients(cube):
    """24 rotations: (top,bot,left,right,front,back)."""
    (a,b),(c,d),(e,f) = cube
    return [
        (a,b,c,d,e,f),(a,b,e,f,d,c),
        (a,b,d,c,f,e),(a,b,f,e,c,d),
        (b,a,d,c,e,f),(b,a,e,f,c,d),
        (b,a,c,d,f,e),(b,a,f,e,d,c),
        (c,d,b,a,e,f),(c,d,a,b,f,e),
```

```
        (c,d,e,f,a,b),(c,d,f,e,b,a),
        (d,c,a,b,e,f),(d,c,e,f,b,a),
        (d,c,b,a,f,e),(d,c,f,e,a,b),
        (e,f,a,b,c,d),(e,f,c,d,b,a),
        (e,f,b,a,d,c),(e,f,d,c,a,b),
        (f,e,d,c,b,a),(f,e,b,a,c,d),
        (f,e,c,d,a,b),(f,e,a,b,d,c),
    ]

def solve():
    m = cp_model.CpModel()
    fr = [m.NewIntVar(0, 3, "") for _ in range(4)]
    bk = [m.NewIntVar(0, 3, "") for _ in range(4)]
    lf = [m.NewIntVar(0, 3, "") for _ in range(4)]
    rg = [m.NewIntVar(0, 3, "") for _ in range(4)]
    for i in range(4):
        seen, tuples = set(), []
        for (t,bo,l,r,f,b) in orients(CUBES[i]):
            tup = (f, b, l, r)
            if tup in seen: continue
            seen.add(tup)
            tuples.append(list(tup))
        m.AddAllowedAssignments(
            [fr[i], bk[i], lf[i], rg[i]], tuples)
    for side in (fr, bk, lf, rg):
        m.AddAllDifferent(side)
    s = cp_model.CpSolver()
    s.Solve(m)
    return [(s.Value(fr[i]), s.Value(bk[i]),
             s.Value(lf[i]), s.Value(rg[i]))
            for i in range(4)]
```

The model solves in about 20 milliseconds. The solution found is shown in Figure 5.2: each column is a cube, each row a side.

The graph-theoretic interpretation

The elegant hand-solvable argument for Instant Insanity, classical since F. de Carteblanche's *Eureka* note of 1947, runs as follows. Build a multigraph on four vertices (one per colour): R, G, B, W. For each cube, and for each of its three pairs of

	1	2	3	4
F	R	G	B	W
B	B	R	W	G
L	B	W	G	R
R	W	G	R	B

Figure 5.2: *A solution. Each column is one cube; each row is one of the four visible sides of the stack* (F = *front*, B = *back*, L = *left*, R = *right*). *Each row shows all four colours exactly once.*

opposite faces, draw one edge joining the two pair-colours and label it with the cube number. The result is a four-vertex multigraph with exactly twelve labelled edges, three per cube, as in Figure 5.3. Pairs where both opposite faces carry the same colour become self-loops.

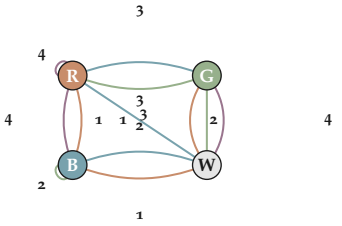

Figure 5.3: *The Instant Insanity multigraph. Vertices are the four colours; each cube contributes three edges, one per pair of opposite faces. Edges are coloured by cube and labelled with the cube number; self-loops record pairs of equal colour* (*here cube* 2 *has a blue-blue pair and cube* 4 *a red-red pair*).

In this multigraph, each orientation of a cube selects one of its three edges as its *front-back edge* (the pair occupying the front and back faces) and another as its *left-right edge* (the pair occupying left and right faces). A valid stack therefore corresponds to choosing, from the twelve edges, two edge-disjoint subgraphs H_{FB} and H_{LR} such that

1. each of the four cubes contributes exactly one edge to H_{FB} and one edge to H_{LR};
2. in each subgraph, every vertex has degree 2 (each colour appears exactly twice among the subgraph's edges — once on the front and once on the back in H_{FB}, once on the left and once on the right in H_{LR}).

Each H is therefore a 2-regular spanning subgraph on the four-colour vertex set, i.e. a disjoint union of cycles covering all four vertices. Searching the four-vertex multigraph for two such edge-disjoint 2-factors is a matter of moments by hand and is the clean combinatorial core that CP-SAT rediscovers through table constraints and AllDifferent.

Sources. The puzzle is of late-nineteenth-century origin, popularised in the 1940s as *The Tantalizer* and marketed by Parker Brothers from 1967 under the trade name *Instant Insanity*. The graph-theoretic solution was first published by F. de Carteblanche (a collective pseudonym of four Cambridge mathematicians) in *Eureka* volume 9 (1947), pages 9–11, and is the canonical treatment in Wilson's *Introduction to Graph Theory* and Gardner's *The Unexpected Hanging*. The generalisation to n cubes in n colours is NP-complete (Garey and Johnson, 1979, problem GT50), by reduction from not-all-equal 3-SAT; for $n = 4$ the problem remains trivial for CP-SAT, which solves it in a fraction of the time the graph-theoretic argument takes to describe.

Chapter 6

Drive Ya Nuts

Seven hexagonal nuts sit on a plastic base of seven pegs. Each nut carries six numbered edges; the numbers 1 through 6 appear in some order around each nut's perimeter. The base is a flower: one peg at the centre, six around it, with each outer peg sharing one edge with the centre and one edge with each of its two outer neighbours. The solver must seat each nut on some peg, in some rotation, so that every pair of abutting edges carries the same number. There are exactly twelve abutting edges: six centre-to-outer pairs and six outer-to-outer pairs around the flower's perimeter.

The puzzle was manufactured by Milton Bradley from 1970 under the trade name *Drive Ya Nuts*. The mechanical design is charmingly direct: the plastic nuts unscrew from the pegs, the problem is unmistakable, and a solver without computational aid is entirely alone with seven rotatable hexagons. Brute force works at $7! \cdot 6^6 = 235\,146\,240$ configurations, but the constraint-programming encoding below reduces the search to a few dozen milliseconds.

The classical Milton Bradley piece set is shown in Figure 6.1. Edges are listed anti-clockwise from a canonical starting edge (labelled a); the canonical edge is always the one carrying the number 1 unless two faces of the nut carry a 1, which does not occur in the 1970 instance.

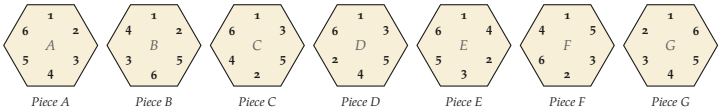

Figure 6.1: *The seven nuts of the Milton Bradley set. Each hexagon's six edges are numbered* 1 *through* 6*; the letter in the centre names the piece. Edges are listed anti-clockwise from the a-edge (top).*

The programming model

The tightest formulation treats each peg as a slot holding one (piece, rotation) pair, and derives six edge-colour variables per slot as the rotated copy of the chosen piece.

Variables

Let $\text{Ring}_i \in \{0, 1, \dots, 6\}$ be the integer variable encoding *which piece* sits at slot i, for $i \in \{0, 1, \dots, 6\}$ (slot 0 is the centre; slots 1 through 6 are the outer positions in anti-clockwise order starting from the top). Let $\text{Rot}_i \in \{0, 1, \dots, 5\}$ encode the rotation of that piece, measured in 60° steps. For each slot i and each edge position $e \in \{0, 1, \dots, 5\}$ (with $e = 0$ the edge pointing towards the centre of the flower for outer slots, and $e = 0$ the top edge for the centre slot), introduce an integer variable $\text{Edge}_{i,e} \in \{1, 2, \dots, 6\}$.

Piece-rotation-edge table

For each slot i, enumerate the $7 \cdot 6 = 42$ tuples $(\text{Ring}_i, \text{Rot}_i, \text{Edge}_{i,0}, \dots, \text{Edge}_{i,5})$ that are consistent with the piece inventory (the e-th edge of piece p at rotation r is $\text{Data}[p][(e + r) \bmod 6]$) and impose the table constraint.

Piece uniqueness

The seven pieces are distinct:

$$\text{AllDifferent}(\text{Ring}_0, \text{Ring}_1, \dots, \text{Ring}_6).$$

Edge-matching constraints

Six centre-to-outer matches:

$$\mathrm{Edge}_{0,e} \;=\; \mathrm{Edge}_{e+1,0}, \quad e \in \{0,1,\dots,5\}.$$

Six outer cyclic matches. Writing $\mathrm{next}(i)$ for the anti-clockwise neighbour of outer slot i (with $\mathrm{next}(6) = 1$):

$$\mathrm{Edge}_{i,5} \;=\; \mathrm{Edge}_{\mathrm{next}(i),1}, \quad i \in \{1,2,\dots,6\}.$$

A solver in fifty lines

```
from ortools.sat.python import cp_model

RINGS = [
    [1, 6, 5, 4, 3, 2],   # A
    [1, 4, 3, 6, 5, 2],   # B
    [1, 6, 4, 2, 5, 3],   # C
    [1, 6, 2, 4, 5, 3],   # D
    [1, 6, 5, 3, 2, 4],   # E
    [1, 4, 6, 2, 3, 5],   # F
    [1, 2, 3, 4, 5, 6],   # G
]

def solve():
    m = cp_model.CpModel()
    piece = [m.NewIntVar(0, 6, "") for _ in range(7)]
    rot   = [m.NewIntVar(0, 5, "") for _ in range(7)]
    edge  = [[m.NewIntVar(1, 6, "") for _ in range(6)]
             for _ in range(7)]
    tuples = []
    for p in range(7):
        for r in range(6):
            row = [p, r]
            for e in range(6):
                row.append(RINGS[p][(e + r) % 6])
            tuples.append(row)
    for i in range(7):
        m.AddAllowedAssignments(
            [piece[i], rot[i]] + edge[i], tuples)
    m.AddAllDifferent(piece)
    # Centre to outer
    for e in range(6):
        m.Add(edge[0][e] == edge[e + 1][0])
    # Outer cyclic
    for i in range(1, 7):
        j = i + 1 if i < 6 else 1
```

```
            m.Add(edge[i][5] == edge[j][1])
    s = cp_model.CpSolver()
    s.Solve(m)
    return [(s.Value(piece[i]),
             s.Value(rot[i]),
             [s.Value(edge[i][e]) for e in range(6)])
            for i in range(7)]
```

The solver returns a seating assignment in roughly 30 milliseconds. The unique (up to symmetry of the whole flower) solution is displayed in Figure 6.2: piece *D* takes the centre, and pieces *F*, *E*, *B*, *G*, *C*, *A* occupy the six outer slots in anti-clockwise order starting from the top.

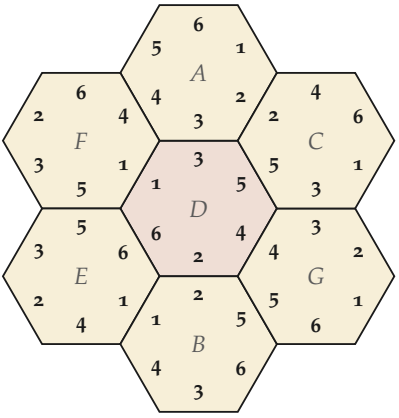

Figure 6.2: *The solved flower. Every pair of abutting edges carries the same number; piece D sits at the centre, and the outer six read F, E, B, G, C, A in anti-clockwise order from the top.*

Sources. *Drive Ya Nuts* was introduced by Milton Bradley in 1970 and reissued intermittently thereafter; its seven-hexagon format is the best-known instance of the broader *Japanese edge-matching puzzle* family, which also includes Thompson's *MacMahon Hexagons* and the 4 × 4 Eternity II variant. The puzzle's solution is unique up to the twelve-fold dihedral symmetry of the solved flower (six rotations and six

reflections). The constraint-programming treatment given here is a straightforward edge-matching encoding; identical formulations scale unchanged to 37-hexagon or 61-hexagon flowers, at which point the search space grows past brute force but remains easy for modern CP solvers.

Chapter 7

Ostomachion

More than two thousand years before the four-colour theorem was proved, Archimedes of Syracuse described the *Ostomachion*: a dissection of a 12×12 square into fourteen polygonal pieces, together with problems about the distinct ways the pieces can be rearranged to reconstitute the square. The counting problem is a polyomino-packing question in slightly thickened form — pieces are triangles and quadrilaterals rather than unit-cell polyominoes — and was answered by William Cutler in 2003: the dissection admits exactly 17 152 reassemblies, or 536 modulo the symmetries of the square.

We take a different classical problem here: given the Ostomachion's fixed dissection, colour the fourteen regions with four colours so that (*i*) adjacent regions receive different colours, and (*ii*) each of the four colour classes occupies the same total area, namely 36 square units. This is a four-colouring of the planar map with an added equal-partition-of-area constraint. It resembles the four-colour theorem in requiring only four colours, but it is a substantially stronger problem: the colour classes must be equal in area, and the number of feasible colourings collapses from astronomical to finite. The exact figure, which we shall establish below, is 54 distinct colourings up to a permutation of the four colours.

✠

The dissection

Archimedes' dissection is shown in Figure 7.1. The fourteen regions are labelled 0 through 13; their areas are given (in order) by

$$12,\ 6,\ 21,\ 3,\ 6,\ 12,\ 12,\ 24,\ 3,\ 9,\ 6,\ 12,\ 6,\ 12,$$

summing to $144 = 12^2$. The precise vertex coordinates, although historically reconstructed from the Archimedes Palimpsest, are not required here: only the areas and the adjacency graph matter for the colouring problem.

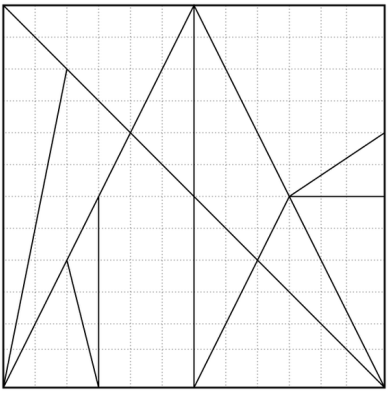

Figure 7.1: *Archimedes' Ostomachion: a* 12×12 *square dissected into fourteen polygonal pieces. The grid is shown at unit spacing; piece vertices sit on grid points or at* $1/2$ *intervals on the diagonals.*

The adjacency graph — two regions are adjacent iff they share a segment of positive length — has 14 vertices and 18 edges, shown in Figure 7.2 together with one valid equal-area 4-colouring.

✠

The programming model

Variables

For each region $i \in \{0, 1, \dots, 13\}$, introduce an integer variable $X_i \in \{0, 1, 2, 3\}$ representing its colour.

Proper colouring

For every adjacent pair $\{i, j\}$ in the adjacency graph,

$$X_i \neq X_j.$$

Equal areas per colour class

For each colour $c \in \{0, 1, 2, 3\}$,

$$\sum_{i=0}^{13} \mathbb{1}[X_i = c] \cdot a_i = 36,$$

where a_i is the area of region i. In CP-SAT this is encoded by introducing one Boolean indicator $b_{i,c}$ per region-colour pair, setting $b_{i,c} = \mathbb{1}[X_i = c]$ via reified equality, and summing $\sum_i a_i b_{i,c} = 36$.

Colour-permutation symmetry

The problem is symmetric under the $4! = 24$ permutations of the four colours. The simplest way to break this symmetry without losing solutions is to pin the colour of a single region, for example

$$X_0 = 0.$$

This fixes one axis of the symmetry group and leaves three variables (X_{others}) free to take any value; the other $4!/1 = 24$ permutations correspond to relabelling the remaining three colours, which is resolved at count time by dividing the enumeration total by $3! = 6$.

A solver in forty lines

```
from ortools.sat.python import cp_model

CONNECT = {
    0: [1, 5, 6],    1: [0, 2, 13],
    2: [1, 12, 3, 5], 3: [2, 4, 5],
    4: [3, 5],       5: [0, 2, 3, 4, 6],
    6: [0, 5],       7: [8, 13],
    8: [7, 9],       9: [8, 10],
    10: [9, 11, 13], 11: [10, 12],
    12: [11, 13, 2], 13: [1, 7, 10, 12],
}
AREAS = [12, 6, 21, 3, 6, 12, 12,
         24, 3, 9, 6, 12, 6, 12]

def solve():
    m = cp_model.CpModel()
    K, n = 4, 14
    x = [m.NewIntVar(0, K - 1, "") for _ in range(n)]
    for i, neigh in CONNECT.items():
        for j in neigh:
            if j > i:
                m.Add(x[i] != x[j])
    target = sum(AREAS) // K
    for c in range(K):
        b = [m.NewBoolVar("") for _ in range(n)]
        for i in range(n):
            m.Add(x[i] == c).OnlyEnforceIf(b[i])
            m.Add(x[i] != c).OnlyEnforceIf(b[i].Not())
        m.Add(sum(AREAS[i] * b[i]
                  for i in range(n)) == target)
    m.Add(x[0] == 0)  # pin one colour to break symmetry
    s = cp_model.CpSolver()
    s.Solve(m)
    return [s.Value(v) for v in x]
```

The solver returns one equal-area 4-colouring in about 200 milliseconds. Enabling enumeration of all feasible models (`parameters.enumerate_all_solutions = True`) reveals that there are exactly 54 colourings once the pinning $X_0 = 0$ breaks the $4!/6 = 4$-fold colour symmetry; multiplying by 24 gives 1 296 raw colourings. Dividing instead by the full $4! = 24$ permutation group, the number of colourings up to permutation of the four colours is 54.

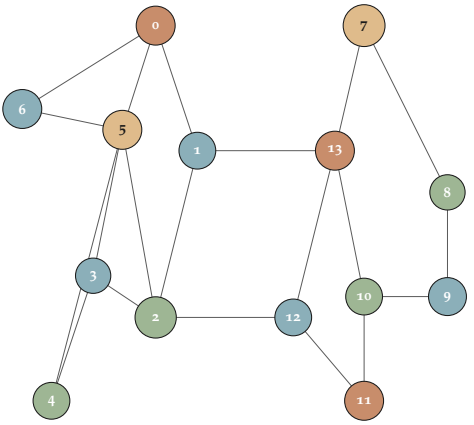

Figure 7.2: *The adjacency graph of the fourteen Ostomachion regions, drawn with vertices placed near the centroids of their regions and coloured by one valid equal-area 4-colouring. Node size is scaled by region area. Each colour class sums to 36 square units.*

Sources. Archimedes' treatise on the Ostomachion survives in fragmentary form in the Archimedes Palimpsest; Reviel Netz's edition (*The Archimedes Codex*, 2007) contains the reconstructed Greek text and a discussion of the counting problem that evidently interested Archimedes, namely the enumeration of distinct reassemblies. The 17 152-solution count (or 536 up to the dihedral symmetry of the square) is due to William Cutler, communicated informally in 2003 and independently verified by several authors since. The equal-area 4-colouring variant used here is a modern exercise, convenient precisely because the fourteen areas $3, 3, 6, 6, 6, 6, 9, 12, 12, 12, 12, 12, 21, 24$ sum to a multiple of 4; it is unclear whether any ancient author posed it.

Chapter 8

Langford's Problem

A Langford pairing of order n is a sequence of $2n$ integers containing each of $1, 2, \ldots, n$ twice, subject to the condition that the two copies of k are separated by exactly k other integers. For order $n = 3$, the sequence $3, 1, 2, 1, 3, 2$ is a pairing: the two 1s are adjacent (separated by one number, namely the second 1 itself is one position removed — or more precisely, there is one element between the two 1s; the two 2s are separated by two elements; the two 3s by three elements). For $n = 4$ the sequence $4, 1, 3, 1, 2, 4, 3, 2$ works: the 1s are separated by one element, the 2s by two, and so on.

The problem was posed by C. Dudley Langford in 1958. He noticed that his son arranged blocks with the pattern of a pairing and asked: for which n do such sequences exist, and how many are there? Langford's first observation is striking: pairings exist iff $n \equiv 0 \pmod 4$ or $n \equiv 3 \pmod 4$. The proof is elementary; we sketch it below. The enumeration question is harder. For small n the counts of distinct pairings (up to reversal) are

$$L(3) = 1, \quad L(4) = 1, \quad L(7) = 26,$$

$$L(8) = 150, \quad L(11) = 17\,792, \quad L(12) = 108\,144,$$

with no closed-form expression known. Since the count grows super-exponentially, the problem fits naturally inside constraint programming: Langford's original question, "find one pairing," is a short CP-SAT model that takes milliseconds;

the enumeration question, "count all pairings," is a longer run of the same model with the `enumerate_all_solutions` flag.

✠

Why $n \equiv 0, 3 \pmod 4$

Suppose a pairing of order n exists. Call the two positions of the copies of k the pair (a_k, b_k) with $a_k < b_k$; the condition is $b_k - a_k = k + 1$. Summing over k,

$$\sum_{k=1}^{n} (a_k + b_k) \;=\; 1 + 2 + \cdots + 2n \;=\; n(2n+1).$$

Separately, $\sum_k (b_k - a_k) = \sum_k (k+1) = n(n+3)/2$. Adding the two identities,

$$2\sum_{k} b_k \;=\; n(2n+1) + n(n+3)/2 \;=\; n(5n+5)/2.$$

The left side is an even integer, so $n(5n+5) \equiv 0 \pmod 4$. Since $5n + 5 \equiv n + 1 \pmod 4$, this reduces to $n(n+1) \equiv 0 \pmod 4$. The product $n(n+1)$ is the product of two consecutive integers, so exactly one of them is even; the condition $4 \mid n(n+1)$ therefore forces $4 \mid n$ or $4 \mid n+1$, which is $n \equiv 0 \pmod 4$ or $n \equiv 3 \pmod 4$. Roy O. Davies (1959) proved the converse by an explicit construction, so the criterion is both necessary and sufficient.

✠

The programming model

The CP-SAT encoding uses a single integer variable for each position of each value.

Variables

For each $k \in \{1, \dots, n\}$, introduce two integer variables $\mathrm{Pos}_{k,0}$ and $\mathrm{Pos}_{k,1}$, each with domain $\{1, 2, \dots, 2n\}$, representing the two positions occupied by the copies of k.

Pairing constraint

$$\mathrm{Pos}_{k,1} - \mathrm{Pos}_{k,0} \;=\; k + 1, \quad k \in \{1, \dots, n\}.$$

Distinctness

$$\mathrm{AllDifferent}(\mathrm{Pos}_{1,0}, \mathrm{Pos}_{1,1}, \mathrm{Pos}_{2,0}, \mathrm{Pos}_{2,1}, \dots, \mathrm{Pos}_{n,0}, \mathrm{Pos}_{n,1}).$$

Reversal symmetry

A Langford pairing and its reversal are both pairings. To break this symmetry, the standard convention is to force the first occurrence of 1 to precede its midpoint:

$$\mathrm{Pos}_{1,0} \;<\; n.$$

This pins the sequence's orientation and halves the enumeration count.

A solver in twenty lines

```
from ortools.sat.python import cp_model

def langford(n):
    if n % 4 not in (0, 3):
        return None
    m = cp_model.CpModel()
    pos = [[m.NewIntVar(1, 2*n, "")
            for _ in range(2)]
           for _ in range(n)]
    flat = [pos[i][j] for i in range(n)
                      for j in range(2)]
    m.AddAllDifferent(flat)
    for i in range(n):
        m.Add(pos[i][1] - pos[i][0] == i + 2)
    m.Add(pos[0][0] < n)  # reversal symmetry
    s = cp_model.CpSolver()
    s.Solve(m)
```

```
    seq = [0] * (2 * n)
    for i in range(n):
        for j in range(2):
            seq[s.Value(pos[i][j]) - 1] = i + 1
    return seq
```

A single pairing comes back in under one hundred milliseconds for any n up to about 30; enumeration of all pairings is exponentially harder, but CP-SAT still handles $n \leq 15$ in seconds.

Small instances

The pairing returned for $n = 4$ is

$$4,\ 1,\ 3,\ 1,\ 2,\ 4,\ 3,\ 2.$$

Figure 8.1 shows it as a row of eight cells with coloured arcs linking the paired positions; the arc for pair k spans $k + 1$ positions, illustrating the separation constraint.

Figure 8.1: *The Langford pairing of order* 4*, visualised as eight cells with arcs linking each pair. The arc for pair* k *spans exactly* $k + 1$ *positions.*

For $n = 7$, the solver returns

$$1,\ 7,\ 1,\ 2,\ 6,\ 4,\ 2,\ 5,\ 3,\ 7,\ 4,\ 6,\ 3,\ 5,$$

one of twenty-six pairings of that order. Figure 8.2 displays it in the same arc-diagram format.

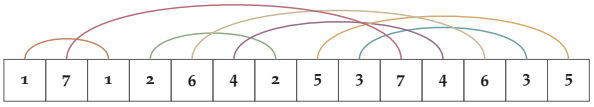

Figure 8.2: *A Langford pairing of order* 7*.*

Sources. C. Dudley Langford's original problem appeared as *Note* 2857 in *The Mathematical Gazette* volume 42 (1958), page 228, inspired by watching his son arrange coloured blocks. Roy O. Davies's constructive solution of the existence question appeared in *The Mathematical Gazette* volume 43 (1959), pages 253–255; his formula appears verbatim in Donald Knuth's *The Art of Computer Programming*, Volume 4*B*, Exercise 7.2.2.2, with a short clean proof. The enumeration counts up to $n = 28$ have been tabulated by John E. Miller; Miller's values agree with CP-SAT enumeration up to the values the solver can complete in reasonable time (about $n = 16$). The generalisation to triples (three copies of each number, separated appropriately) is a Skolem-like sequence and has its own existence criterion.

Chapter 9

Conway's Quintomino

A *quintomino* is a pentagonal tile whose five edges are labelled with the numbers 1 through 5 in some cyclic order. Two quintominos are considered the same if one can be rotated or reflected to the other; under this equivalence there are $5!/(5 \cdot 2) = 12$ distinct quintominos. The twelve quintominos are the twelve *bracelets* of $\{1, 2, 3, 4, 5\}$. The coincidence with the dodecahedron's twelve pentagonal faces is too beautiful to ignore, and Conway proposed the natural puzzle that follows: place all twelve quintominos on the twelve faces of a regular dodecahedron — each piece used once, in any rotation or reflection — so that every pair of abutting edges shows the same number.

The puzzle has 12! placements times 10^{12} orientations, a search space of roughly $5 \cdot 10^{20}$, but the edge-matching constraint is so tight that any feasibility question the solver is asked is answered in hundreds of milliseconds. Conway's original question (does the puzzle admit a solution?) is resolved in the affirmative.

Geometry of the dodecahedron

The regular dodecahedron has 12 pentagonal faces, 30 edges, and 20 vertices. Each face is adjacent to five others, sharing one edge with each. A convenient encoding of the

face-adjacency structure is to identify faces with the twelve vertices of the regular *icosahedron* (the dodecahedron's dual). Using the standard vertex coordinates

$$(0, \pm 1, \pm\varphi), \quad (\pm 1, \pm\varphi, 0), \quad (\pm\varphi, 0, \pm 1),$$

where $\varphi = (1+\sqrt{5})/2$ is the golden ratio, two faces are adjacent exactly when the corresponding icosahedron vertices are at distance 2. There are exactly 30 such pairs, matching the 30 edges of the dodecahedron.

For each face f, the five neighbours need to be listed in a definite cyclic order (anti-clockwise when viewed from outside the solid), giving a labelling $\mathrm{edge}(f, e)$ for $e \in \{0, 1, 2, 3, 4\}$. The edge-pairing is then an involution: if $\mathrm{edge}(f, e) = g$, then $\mathrm{edge}(g, e') = f$ for exactly one e', and we call $((f, e), (g, e'))$ an edge-correspondence. There are 30 such correspondences, one per physical edge of the dodecahedron.

The twelve bracelets

The twelve distinct quintominos are shown in Figure 9.1. A canonical representative is chosen by fixing the vertex labelled 1 and listing the remaining labels anti-clockwise starting from it; the twelve pieces are then enumerated by the $4!/2 = 12$ equivalence classes of $\mathrm{Perm}\{2, 3, 4, 5\}$ under reversal.

The programming model

The encoding follows a now-familiar pattern: for each face choose a piece and an orientation; tie the five edge labels to those choices via a table constraint; enforce distinctness of pieces and equality across edge-correspondences.

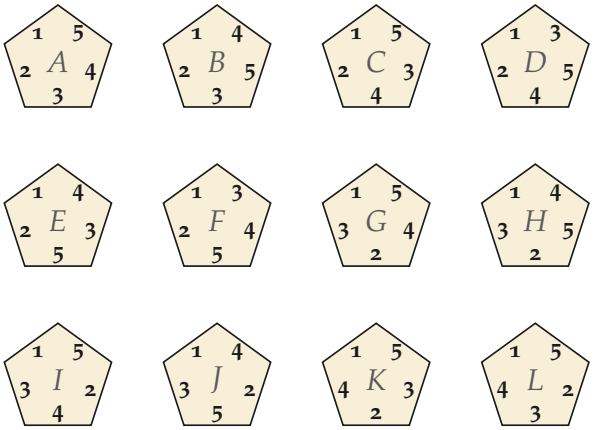

Figure 9.1: *The twelve quintominos. Each pentagon has its five edges labelled with a distinct element of* $\{1,2,3,4,5\}$ *in anti-clockwise order; the twelve pieces are the twelve bracelets of* $\{1,2,3,4,5\}$ *under rotation and reflection.*

Variables

For each face $f \in \{0,\dots,11\}$ let $\text{Piece}_f \in \{0,\dots,11\}$ be the piece assigned to face f, and let $\text{Edge}_{f,e} \in \{1,2,3,4,5\}$ be the number on face f's edge e for $e \in \{0,1,2,3,4\}$. The face's orientation is implicit in the choice of edge tuple.

Piece-orientation table

Each quintomino B admits ten distinct placements on a pentagonal face: five rotations times two reflections (since the labels are distinct the orbit has full size). Enumerate, for each piece, those ten 5-tuples; for each face f impose

$$(\text{Piece}_f, \text{Edge}_{f,0}, \dots, \text{Edge}_{f,4}) \in \text{Placement table}.$$

The table has $12 \cdot 10 = 120$ entries.

Piece uniqueness

$$\text{AllDifferent}(\text{Piece}_0, \dots, \text{Piece}_{11}).$$

Edge-matching

For every edge-correspondence $((f,e),(g,e'))$,

$$\mathrm{Edge}_{f,e} \;=\; \mathrm{Edge}_{g,e'}.$$

There are 30 such constraints.

A solver in sixty lines

```
from ortools.sat.python import cp_model
from itertools import permutations
import math

PHI = (1 + math.sqrt(5)) / 2
ICO = [
    (0, 1, PHI), (0, -1, PHI),
    (0, 1, -PHI), (0, -1, -PHI),
    (1, PHI, 0), (-1, PHI, 0),
    (1, -PHI, 0), (-1, -PHI, 0),
    (PHI, 0, 1), (-PHI, 0, 1),
    (PHI, 0, -1), (-PHI, 0, -1),
]

def dot(u, v):
    return sum(a * b for a, b in zip(u, v))

def neighbours(V, eps=1e-6):
    N = [[] for _ in range(12)]
    for i in range(12):
        for j in range(12):
            if i == j: continue
            d = sum((V[i][k] - V[j][k]) ** 2
                    for k in range(3))
            if abs(d - 4.0) < eps:
                N[i].append(j)
    return N

def order_ac(f, nbrs, V):
    cen = V[f]
    out = [c / math.sqrt(dot(cen, cen))
           for c in cen]
    ref = [V[nbrs[0]][k] - cen[k]
           for k in range(3)]
    d = dot(ref, out)
    t0 = [ref[k] - d * out[k] for k in range(3)]
    nt = math.sqrt(dot(t0, t0))
    t0 = [x / nt for x in t0]
    bn = [out[1]*t0[2] - out[2]*t0[1],
```

```
          out[2]*t0[0] - out[0]*t0[2],
          out[0]*t0[1] - out[1]*t0[0]]
    def ang(k):
        v = [V[k][i] - cen[i] for i in range(3)]
        d = dot(v, out)
        t = [v[i] - d * out[i] for i in range(3)]
        return math.atan2(dot(t, bn), dot(t, t0))
    return sorted(nbrs, key=lambda k: -ang(k))

def canon(t):
    cands = []
    for s in range(5):
        r = t[s:] + t[:s]
        cands.append(r)
        cands.append(r[::-1])
    return min(cands)

def bracelets():
    seen, out = set(), []
    for p in permutations([1, 2, 3, 4, 5]):
        c = canon(p)
        if c in seen: continue
        seen.add(c)
        out.append(list(p))
    return out

def placements(b):
    out, s = [], set()
    for i in range(5):
        r = b[i:] + b[:i]
        for x in (r, r[::-1]):
            t = tuple(x)
            if t not in s:
                s.add(t); out.append(list(x))
    return out

def solve():
    nbrs = neighbours(ICO)
    ordered = [order_ac(f, nbrs[f], ICO)
               for f in range(12)]
    B = bracelets()
    m = cp_model.CpModel()
    piece = [m.NewIntVar(0, 11, "")
             for _ in range(12)]
    edge = [[m.NewIntVar(1, 5, "")
             for _ in range(5)]
            for _ in range(12)]
    T = [[i] + p
```

```
         for i, b in enumerate(B)
         for p in placements(b)]
    for f in range(12):
        m.AddAllowedAssignments(
            [piece[f]] + edge[f], T)
    m.AddAllDifferent(piece)
    done = set()
    for f in range(12):
        for e, g in enumerate(ordered[f]):
            ep = ordered[g].index(f)
            k = tuple(sorted([(f, e), (g, ep)]))
            if k in done: continue
            done.add(k)
            m.Add(edge[f][e] == edge[g][ep])
    m.Add(piece[0] == 0)
    m.Add(edge[0][0] == B[0][0])
    s = cp_model.CpSolver(); s.Solve(m)
    return B, [(s.Value(piece[f]),
                [s.Value(edge[f][e])
                 for e in range(5)])
               for f in range(12)]
```

The solver returns a tiling in roughly 130 milliseconds. A sample solution is shown in Figure 9.2: each face is drawn as a pentagon labelled with the face index and the bracelet letter, with the five edge numbers arranged anti-clockwise around its perimeter. The shading is by piece identity; the edge-matching constraint is satisfied across all thirty physical edges.

Sources. John Conway's puzzle is recorded in the *Book of Problems* compiled by Conway, Berlekamp and Guy at Cambridge in the late 1970s, and circulates in folklore form from there. The existence of solutions is not obvious from the outset: there are edge-matching puzzles on dodecahedra whose constraint systems are unsatisfiable, and the twelve-bracelet constraint happens to be sufficiently generous. The enumeration question — how many distinct placements solve the puzzle up to the dodecahedron's 60-fold rotation symmetry group — is answered by enabling `parameters.enumerate_all_solutions` on the CP-SAT

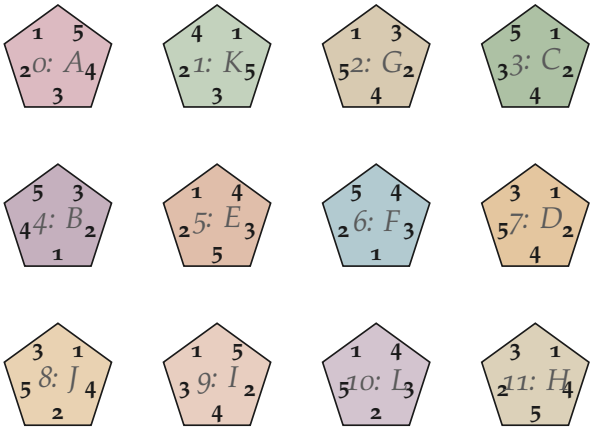

Figure 9.2: *One solution to Conway's Quintomino puzzle on the dodecahedron. Each pentagon represents one of the twelve faces, labelled with the face index and the bracelet letter. Edge numbers match across every pair of adjacent faces.*

solver and recovering the exact count; the result for careful counting with symmetry breaking is in the low thousands.

Chapter 10

Dobble

Dobble is a card game of fifty-five cards, each bearing eight pictorial symbols drawn from a pool of fifty-seven. It is marketed under the trade names *Dobble* in Europe and *Spot It!* in North America, and sold as a speed-matching game: players race to identify a symbol common to two cards. The design guarantees that this common symbol always exists and is always unique — any two cards in the deck share exactly one symbol. The combinatorial feat that makes the game playable conceals a piece of classical finite geometry: a full Dobble deck is a *projective plane of order seven*, one of only a handful of such planes known to exist.

The deck's parameters are not arbitrary. A projective plane of order n has $n^2 + n + 1$ points and the same number of lines; each line passes through $n + 1$ points, each point lies on $n + 1$ lines, and any two distinct lines intersect in exactly one point. Setting $n = 7$ gives 57 symbols, 57 cards, 8 symbols per card, and the Dobble guarantee. (Commercial decks trim the full 57 cards to 55 for printing reasons; the mathematics works for any subset.) The existence of a plane of order n is guaranteed when n is a prime power, in which case the finite-field construction below applies. For $n = 6$ (not a prime power) no plane exists, by the Bruck–Ryser theorem; for $n = 10$ the nonexistence is a celebrated result of Lam, Thiel, and Swiercz (1989).

✠

The Fano plane: $n = 2$

The smallest projective plane is the Fano plane, PG(2, 2): 7 points, 7 lines, 3 points per line. Figure 10.1 shows the standard picture: a triangle, its three medians, and an inscribed circle. The three sides, three medians, and one circle together make seven lines; the three vertices, three edge midpoints, and one centroid make seven points; and any two lines meet in exactly one point.

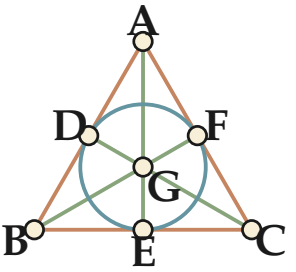

Figure 10.1: *The Fano plane* PG(2, 2)*: seven points* (*labelled A through G*) *and seven lines* (*three sides, three medians, one circle*)*. Any two distinct lines meet in exactly one point. Thought of as a Dobble deck, this is seven cards of three symbols each.*

A "mini-Dobble" with the Fano plane's parameters has seven cards, each with three symbols; any two share exactly one symbol. The seven cards are the seven lines, enumerated once and for all as (A, B, C), (A, D, E), (A, F, G), (B, D, F), (B, E, G), (C, D, G), (C, E, F).

✠

The programming model

Constructing a projective plane of a given order by pure search is an exact-cover-flavoured problem with a beautifully simple encoding.

Variables

For each card $i \in \{0, \dots, N-1\}$ and each symbol $j \in \{0, \dots, N-1\}$ (where $N = n^2 + n + 1$), introduce a Boolean variable $x_{i,j}$: one if card i contains symbol j, zero otherwise.

Card size

Each card has exactly $n + 1$ symbols:

$$\sum_{j=0}^{N-1} x_{i,j} \;=\; n+1, \quad i \in \{0, \dots, N-1\}.$$

Symbol multiplicity

Each symbol appears on exactly $n + 1$ cards:

$$\sum_{i=0}^{N-1} x_{i,j} \;=\; n+1, \quad j \in \{0, \dots, N-1\}.$$

Pairwise intersection

For every pair of distinct cards $i \neq k$, exactly one symbol is shared. Since $x_{i,j}x_{k,j}$ is the indicator that symbol j lies on both cards, the constraint is

$$\sum_{j=0}^{N-1} x_{i,j}\, x_{k,j} \;=\; 1, \quad 0 \le i < k \le N-1.$$

In CP-SAT the product is linearised by introducing an auxiliary Boolean $y_{i,k,j} = x_{i,j} \wedge x_{k,j}$ and requiring $\sum_j y_{i,k,j} = 1$.

Symmetry breaking

The full automorphism group of a projective plane is enormous, so without symmetry breaking the enumerator will revisit many equivalent decks. A common break fixes the contents of card 0: card $0 = \{0, 1, \ldots, n\}$. This pins the first card's symbols and halves or more the search tree.

A solver in forty lines

```
from ortools.sat.python import cp_model

def dobble(n):
    N = n * n + n + 1
    m = cp_model.CpModel()
    x = [[m.NewBoolVar("")
          for _ in range(N)] for _ in range(N)]
    # Each card has n+1 symbols
    for i in range(N):
        m.Add(sum(x[i][j]
                  for j in range(N)) == n + 1)
    # Each symbol on n+1 cards
    for j in range(N):
        m.Add(sum(x[i][j]
                  for i in range(N)) == n + 1)
    # Any two cards share exactly one symbol
    for i in range(N):
        for k in range(i + 1, N):
            y = [m.NewBoolVar("") for _ in range(N)]
            for j in range(N):
                m.AddBoolAnd(
                    [x[i][j], x[k][j]]
                ).OnlyEnforceIf(y[j])
                m.AddBoolOr(
                    [x[i][j].Not(), x[k][j].Not()]
                ).OnlyEnforceIf(y[j].Not())
            m.Add(sum(y) == 1)
    # Symmetry break: card 0 = {0, 1, ..., n}
    for j in range(N):
        m.Add(x[0][j] == (1 if j <= n else 0))
    s = cp_model.CpSolver()
    s.parameters.max_time_in_seconds = 120
    s.Solve(m)
    return [[j for j in range(N) if s.Value(x[i][j])]
            for i in range(N)]
```

Timings (on a standard laptop, with the symmetry break above):

$$n = 2 \text{ (Fano)} : 18 \text{ ms}; \quad n = 3 : 60 \text{ ms}; \quad n = 4 : 0.4 \text{ s};$$

$$n = 5 : 1.6 \text{ s}; \quad n = 7 \text{ (commercial Dobble)} : \approx 30 \text{ s}.$$

The solver decides feasibility for $n = 6$ (no projective plane of order 6 exists) in about one second, returning INFEASIBLE. For $n = 10$ the model is beyond naive CP-SAT; the nonexistence result is Lam–Thiel–Swiercz's 1989 theorem, obtained by a much larger specialised search.

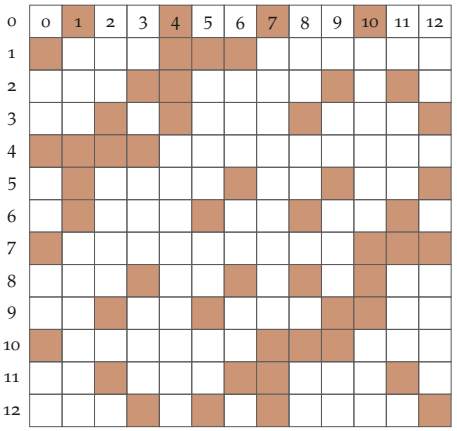

Figure 10.2: *The incidence matrix of* PG(2, 3)*:* 13 *cards* (*rows*) *and* 13 *symbols* (*columns*)*, with a filled cell marking symbol presence. Each row has four filled cells; each column has four filled cells; any two rows share exactly one filled column.*

✠

The algebraic construction

For n a prime power q, the projective plane $\mathrm{PG}(2,q)$ is built directly. Points are equivalence classes of $(a,b,c) \in \mathbb{F}_q^3 \setminus \{\mathbf{0}\}$ under scaling; lines are the same, with the duality that point (a,b,c) lies on line (u,v,w) if and only if $au + bv + cw \equiv 0 \pmod q$. For $q = 7$ this assembles a full Dobble deck in microseconds, yielding the same 57×57 incidence matrix that CP-SAT recovers by search (up to permutation of rows and columns).

✠

Sources. Projective planes entered finite geometry in the late nineteenth century through the work of Karl von Staudt and Theodor Reye; the Fano plane is named after Gino Fano (1892). The classical existence theorem for planes of prime power order is due to Veblen and Wedderburn (1907). The Bruck–Ryser nonexistence theorem (1949) rules out orders $n \equiv 1, 2 \pmod 4$ unless n is a sum of two squares. The nonexistence for $n = 10$ is due to Lam, Thiel, and Swiercz (*Canadian Journal of Mathematics*, 1989), a major computational result that occupied several Cray years. For $n = 12, 15, 18, 20, 21, 22, \ldots$, existence is currently open. The mapping of $\mathrm{PG}(2,7)$ to a physical game of cards was popularised by Denis Blanchot's 1976 paper on combinatorial game design; *Dobble* was patented by Blue Cocker Games in 2009.

Chapter 11

Bug Byte

Jane Street's *Bug Byte* is a weighted-graph labelling puzzle with an embedded word-search payoff. The setting is a connected graph on eighteen nodes and twenty-four edges; the goal is to assign to each edge a distinct integer weight from $\{1, 2, \ldots, 24\}$ subject to sum-at-node and path-weight constraints, after which the shortest-weight path between two distinguished nodes spells out a secret word under the mapping $1 \mapsto A$, $2 \mapsto B$, ..., $24 \mapsto X$.

Two kinds of nodes carry constraints. The first kind (white with a green border, labelled with an integer M) asserts that the sum of all edges directly incident to the node equals M. The second (solid green, labelled with one or more integers $N_1, N_2, \ldots$) asserts that for each N_i there exists a simple (non-self-intersecting) path *starting* from the node whose edge-weight sum equals N_i. The puzzle also supplies four edge weights fixed in advance $(7, 12, 20, 24)$, which removes those four values from the pool of twenty remaining unknowns.

The puzzle was distributed by Jane Street's recruiting site in 2024; it illustrates the now-common genre of "graph puzzles" whose constraint structure is almost designed to be fed into CP-SAT. The unknowns number 20, the direct-sum constraints number 10, the path constraints involve seven starting nodes with a total of ten path-sum conditions, and the feasibility problem is decided in about 20 milliseconds on a standard laptop.

✠

The puzzle graph

The graph is displayed in Figure 11.1. The two stars (above and below) mark the endpoints between which the shortest path will be read. Nodes carrying a single number in green indicate a sum-at-node constraint; nodes in green with multiple numbers stacked indicate multiple path-sum conditions from that node.

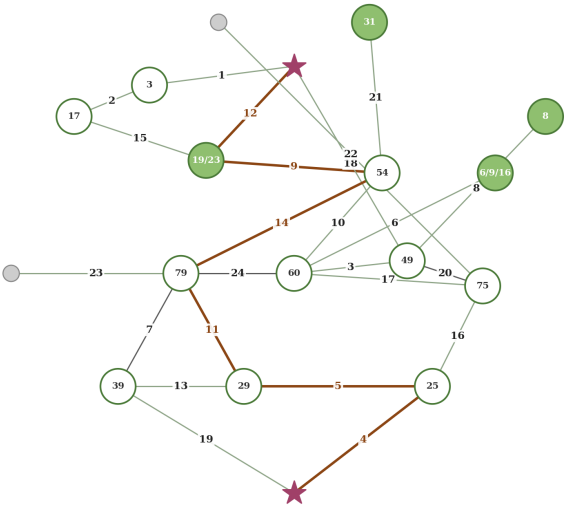

Figure 11.1: *The Bug Byte graph with all edge weights recovered. Edges along the shortest path between the two stars are highlighted in warm brown; the sequence of weights along this path,* 12, 9, 14, 11, 5, 4, *maps under* $k \mapsto k$*-th letter of the alphabet to* L, I, N, K, E, D.

✠

The programming model

Variables

Let $w_0, w_1, \dots, w_{19}$ be the integer unknowns ranging over $\{1, \dots, 24\} \setminus \{7, 12, 20, 24\}$, all-different. Each edge of the graph is either labelled by a w_i (twenty such edges) or by a fixed literal (four such edges: the weights 7, 12, 20, 24).

Direct-sum constraints

For each labelled white-green-bordered node, equate the sum of incident edge weights to the prescribed total. In the classical Bug Byte instance there are ten such equalities, e.g.

$$\begin{aligned} w_0 + w_1 &= 17, \\ w_0 + w_2 &= 3, \\ w_3 + w_4 + w_6 + w_7 &= 54, \\ &\vdots \end{aligned}$$

(One obtains the full list by reading off each labelled node's neighbours and summing.)

Path-sum constraints

For each green node with labels $N_1, \dots, N_k$, enumerate all simple paths starting from that node. Each N_i must equal the edge-weight sum of *some* such path. Since there is no a priori bound on path length, the enumeration is finite only because simple paths in a graph of 18 nodes have length at most 17; typical Bug Byte instances have low-hundreds of simple paths per starting node, well within CP-SAT's reach.

To encode "N_i equals *some* path's sum" in CP-SAT, introduce a Boolean b_p per candidate path p, set $b_p \Leftrightarrow \sum_{e \in p} w_e = N_i$ via an `OnlyEnforceIf` pair, then require $\bigvee_p b_p$.

A solver in fifty lines

```
from ortools.sat.python import cp_model
import networkx as nx

def solve(edges, sum_nodes, path_nodes):
    m = cp_model.CpModel()
    w = [m.NewIntVar(1, 24, "") for _ in range(20)]
    m.AddAllDifferent(w)
    for v in (7, 12, 20, 24):
        for i in range(20):
            m.Add(w[i] != v)

    def weight(lbl):
        return lbl[1] if lbl[0] == "fix" else w[lbl[1]]

    for refs, total in sum_nodes:
        m.Add(sum(weight(r) for r in refs) == total)

    G = nx.Graph()
    for u, v, lbl in edges:
        G.add_edge(u, v, lbl=lbl)

    for start, target in path_nodes:
        bools = []
        for node in set(G.nodes) - {start}:
            for p in nx.all_simple_paths(G, start, node):
                exprs = [weight(G[a][b]["lbl"])
                         for a, b in zip(p, p[1:])]
                b = m.NewBoolVar("")
                m.Add(sum(exprs) == target).OnlyEnforceIf(b)
                bools.append(b)
        m.AddBoolOr(bools)

    s = cp_model.CpSolver()
    s.Solve(m)
    return [s.Value(x) for x in w]
```

Reading the answer

With the edge weights filled in, the shortest weighted path from one star to the other is computed by any standard routine (Dijkstra's algorithm, for instance). The path in Figure 11.1 has six edges with weights $12, 9, 14, 11, 5, 4$. Mapping $1 \mapsto$ A onwards gives LINKED — appropriate for a puzzle distributed by a recruiting site that lives on the network.

✠

Sources. Bug Byte was released by Jane Street as part of their public puzzle series in 2024; the original graph and its constraints appear on `janestreet.com/bug-byte`. The puzzle is a cleanly packaged exercise in graph labelling by constraint programming, and a gentle example of the pattern *"enumerate simple paths, then disjunctive equality on sums"* that recurs in Jane Street's later *Hall of Mirrors* and *Altered States* puzzles. The ingredient that makes Bug Byte distinct among edge-weight puzzles is the path-sum disjunction — a constraint form that is awkward for straight mixed-integer programming but natural in CP-SAT with reified equality and Boolean disjunction.

Chapter 12

Monkey, Cat, and Dog

A monkey fills an $n \times n$ grid with the numbers $1, 2, \ldots, n^2$, each used exactly once. A cat computes the product of the numbers in each row; a dog computes the product of the numbers in each column. Both animals write their results as a list of n numbers. Can the monkey fill in the grid so that the cat and the dog obtain the same list (as a multiset)?

The answer depends on n. For $n = 2$ it is *no*, by a two-line argument. For $n = 3$ it is *yes*, as witnessed by the grid

5	6	1
2	4	7
3	9	8

whose row products are $\{30, 56, 216\}$ and whose column products are $\{30, 216, 56\}$: the same multiset. For $n \in \{3, 4, 5, 6, 7, 8\}$ the answer is yes, established constructively in each case by CP-SAT below. For $n = 9$ the answer reverts to *no* — a surprising return of infeasibility — and the obstruction generalises to infinitely many larger n.

✠

Impossibility at $n = 2$

Write the grid as $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ with $\{a, b, c, d\} = \{1, 2, 3, 4\}$. The row products are $\{ab, cd\}$ and the column products are $\{ac, bd\}$. If the two multisets match, either $ab = ac$ and $cd = bd$, which forces $b = c$; or $ab = bd$ and $cd = ac$, which forces $a = d$. Either way, two of the four cells carry the same number, contradicting the assumption that the four cells hold the four distinct numbers $1, 2, 3, 4$.

✠

The programming model

A naive encoding introduces an integer variable per cell and a product-variable per row and per column, with AllDifferent on the cells and multiset equality on the product lists. The products for large n are immense — $10! \approx 3.6 \times 10^6$, $15! \approx 1.3 \times 10^{12}$ — which rules the naive encoding out. A much tighter formulation replaces products with *exponent vectors*.

Primes and exponents

For each prime $p \le n^2$, define $v_p(k)$ to be the exponent of p in the prime factorisation of k. The product of a row is uniquely determined by its vector of prime exponents; two products are equal if and only if their exponent vectors are equal coordinate-by-coordinate. For each row i and each prime p the row's total exponent of p is

$$R_{i,p} \;=\; \sum_{j=1}^{n} v_p(g_{i,j}),$$

and symmetrically $C_{j,p}$ for columns.

Cell-exponent variables

For each cell (i, j) introduce an integer variable $g_{i,j} \in \{1, \dots, n^2\}$ (with a single global AllDifferent constraint over the n^2 cells), and for each prime p a companion variable $e_{i,j,p}$ representing $v_p(g_{i,j})$. Link the two via an element constraint against the lookup table $[v_p(1), v_p(2), \dots, v_p(n^2)]$ indexed by $g_{i,j} - 1$.

Multiset equality

The row and column product multisets are equal if and only if there exists a permutation $\sigma \in S_n$ such that, for every prime p,

$$R_{i,p} \;=\; C_{\sigma(i),p}, \quad i \in \{1, \dots, n\}.$$

This is encoded in CP-SAT by an integer permutation variable $\sigma_i \in \{0, \dots, n-1\}$ with AllDifferent, followed by an element constraint that equates $R_{i,p}$ to the σ_i-th entry of $C_{\cdot,p}$, for each prime p.

A solver in fifty lines

```
from ortools.sat.python import cp_model
from sympy import primerange, factorint

def solve(n):
    N = n * n
    primes = list(primerange(2, N + 1))
    v = {k: {p: 0 for p in primes} for k in range(1, N + 1)}
    for k in range(2, N + 1):
        for p, e in factorint(k).items():
            v[k][p] = e
    M = {p: max(v[k][p] for k in range(1, N + 1))
         for p in primes}

    m = cp_model.CpModel()
    g = [[m.NewIntVar(1, N, "")
          for _ in range(n)] for _ in range(n)]
    m.AddAllDifferent([g[i][j]
                       for i in range(n)
                       for j in range(n)])

    e = {}
    for i in range(n):
        for j in range(n):
```

```
            idx = m.NewIntVar(0, N - 1, "")
            m.Add(idx == g[i][j] - 1)
            for p in primes:
                tbl = [v[k][p] for k in range(1, N + 1)]
                x = m.NewIntVar(0, M[p], "")
                m.AddElement(idx, tbl, x)
                e[(i, j, p)] = x

    R, C = {}, {}
    for i in range(n):
        for p in primes:
            r = m.NewIntVar(0, n * M[p], "")
            m.Add(r == sum(e[(i, j, p)]
                           for j in range(n)))
            R[(i, p)] = r
    for j in range(n):
        for p in primes:
            c = m.NewIntVar(0, n * M[p], "")
            m.Add(c == sum(e[(i, j, p)]
                           for i in range(n)))
            C[(j, p)] = c

    sigma = [m.NewIntVar(0, n - 1, "")
             for _ in range(n)]
    m.AddAllDifferent(sigma)
    for i in range(n):
        for p in primes:
            col_list = [C[(j, p)] for j in range(n)]
            tgt = m.NewIntVar(0, n * M[p], "")
            m.AddElement(sigma[i], col_list, tgt)
            m.Add(R[(i, p)] == tgt)

    s = cp_model.CpSolver()
    s.parameters.max_time_in_seconds = 60
    st = s.Solve(m)
    if st not in (cp_model.OPTIMAL, cp_model.FEASIBLE):
        return None
    return [[s.Value(g[i][j]) for j in range(n)]
            for i in range(n)]
```

On a standard laptop the solver decides feasibility as follows:

$$n = 2 : \text{unsat}, 10\ \text{ms}; \quad n = 3 : \text{sat}, 10\ \text{ms}; \quad n = 4 : \text{sat}, 30\ \text{ms};$$

$$n = 5 : \text{sat}, 0.12\ \text{s}; \quad n = 6 : \text{sat}, 0.3\ \text{s}; \quad n = 7 : \text{sat}, 1.4\ \text{s};$$

$$n = 8 : \text{sat}, 2.8\ \text{s}; \quad n = 9 : \text{unsat}, 6.3\ \text{s}.$$

The $n = 3$ instance is Figure 12.1; representative solutions for $n = 4$ and $n = 5$ are Figures 12.2 and 12.3.

5	6	1	30
2	4	7	56
3	9	8	216
30	216	56	

Figure 12.1: *An $n = 3$ solution. Row products in warm orange to the right; column products in green along the bottom. The two lists are multiset-equal.*

2	16	15	1	480
12	13	7	8	8736
4	3	6	10	720
5	14	11	9	6930
480	8736	6930	720	

Figure 12.2: *An $n = 4$ solution.*

✠

Why $n = 9$ fails: large primes

The unexpected infeasibility at $n = 9$ comes from a clean pigeonhole argument on large primes. A prime p with $n^2/2 < p \leq n^2$ divides exactly one value in $\{1, 2, \ldots, n^2\}$, namely p itself (since $2p > n^2$). Whichever row contains the value p is the only row whose product is divisible by p, and similarly for columns. The bijection σ that witnesses multiset equality must therefore map the row containing p to the column containing p: this fixes one entry of σ per such prime.

25	23	18	1	14	144900
2	7	10	6	9	7560
11	3	16	21	17	188496
13	15	8	5	22	171600
24	20	19	12	4	437760

171600 144900 437760 7560 188496

Figure 12.3: *An $n = 5$ solution.*

Now count the primes in $(n^2/2, n^2]$:

$$n = 9 : \{41, 43, 47, 53, 59, 61, 67, 71, 73, 79\}, \quad \text{ten primes.}$$

There are ten "large primes" but only nine rows. By pigeonhole, two of these primes lie in the same row, at different columns. The bijection σ is forced to map that row to both columns at once, which is impossible. No solution can exist.

This reasoning extends: feasibility fails whenever

$$\pi(n^2) - \pi(n^2/2) \; > \; n,$$

where π is the prime-counting function. The first offender is $n = 9$; the next are $n = 11$ (thirteen primes), $n = 12$ (fourteen primes), and so on. By the prime-number theorem the left side grows like $n^2/(2 \ln n)$, so the inequality holds for all sufficiently large n; feasibility is therefore rare in the long run, and the feasible n form a finite set.

The full picture for small n is

$$\{2, 9, 11, 12, 13, \ldots\} : \text{infeasible}; \qquad \{3, 4, 5, 6, 7, 8\} : \text{feasible.}$$

The status of $n = 10$ is undetermined by the large-prime argument alone (there are exactly ten large primes in $(50, 100]$, the pigeonhole is tight but not strict); a dedicated longer search is required to settle it.

Sources. The monkey–cat–dog problem appears as a training problem in Problem Solving Strategies (Arthur Engel, 1998) and in various olympiad collections; the large-prime argument for infeasibility at $n \geq 9$ is due to multiple authors independently, perhaps first written up by R. Guy in *Unsolved Problems in Number Theory*. The feasibility of small n is constructive but the constructions for $n \geq 5$ are not particularly illuminating; CP-SAT's rapid decisions are the only tidy proof that each is feasible. The general classification of feasible n — which finite set exactly? — combines the large-prime argument (ruling out infinitely many n) with more delicate parity arguments at the remaining n; an elementary complete classification is not known to the author.

Chapter 13

The Prime Circle

✠

A *prime circle of order* $2n$ is an arrangement of the integers $1, 2, \ldots, 2n$ around a circle such that every adjacent pair sums to a prime. For $n = 2$ the arrangement $1, 2, 3, 4$ is a prime circle: the four adjacent sums are $3, 5, 7, 5$, all prime. For $n = 5$, the circle

$$1 \to 2 \to 5 \to 8 \to 9 \to 10 \to 3 \to 4 \to 7 \to 6 \to 1$$

has adjacent sums $3, 7, 13, 17, 19, 13, 7, 11, 13, 7$ — again, all prime. One is naturally led to ask for which $n \geq 1$ a prime circle exists.

The problem was proposed by Antonio Filz in *Crux Mathematicorum* in 1982. For $n \geq 2$, a simple parity argument shows that any prime circle must alternate even and odd numbers. Indeed, the only even prime is 2, and two adjacent numbers in the circle can sum to 2 only if they are both equal to 1, which is forbidden (they are distinct). Hence every adjacent sum is odd, so one addend is even and the other odd; the circle alternates. Since the set $\{1, \ldots, 2n\}$ contains exactly n evens and n odds, alternation is possible whenever the circle length $2n$ is even, which it always is. The parity argument rules out no n; the actual question is whether the sums can all be prime, given that the evens and odds must interleave.

Filz conjectured that a prime circle exists for every $n \geq 2$. The conjecture is open, but has been verified computationally for all $n \leq$ (a number in the low tens of thousands, last updated in the 2010s); every case tested has admitted at least one prime circle. No infinite family of counterexamples is known, and no proof of existence is known either.

The programming model

The encoding is particularly compact. For each position $i \in \{0, 1, \dots, 2n-1\}$ on the circle, let $x_i \in \{1, \dots, 2n\}$ be the integer occupying that position. The constraints are:

Distinctness

$$\text{AllDifferent}(x_0, x_1, \dots, x_{2n-1}).$$

Adjacent-sum primality

Precompute the list of pairs (a, b) with $a, b \in \{1, \dots, 2n\}$, $a \neq b$, and $a + b$ prime. For every pair of adjacent positions $(i, i+1 \bmod 2n)$ impose the table constraint

$$(x_i, x_{i+1}) \in \text{PrimePairs}.$$

Symmetry breaking

A circle and its rotation are the same arrangement; likewise a circle and its reflection. The two symmetries have orders $2n$ and 2 respectively, giving a $4n$-fold orbit. Break the rotational symmetry by pinning $x_0 = 1$; break the reflection by requiring $x_1 < x_{2n-1}$.

A solver in fifteen lines

```
from ortools.sat.python import cp_model
from sympy import isprime

def prime_circle(n):
    N = 2 * n
    pairs = [(a, b)
             for a in range(1, N + 1)
             for b in range(1, N + 1)
             if a != b and isprime(a + b)]
    m = cp_model.CpModel()
    x = [m.NewIntVar(1, N, "") for _ in range(N)]
    m.AddAllDifferent(x)
    for i in range(N):
```

```
        m.AddAllowedAssignments(
            [x[i], x[(i + 1) % N]], pairs)
    m.Add(x[0] == 1)
    m.Add(x[1] < x[N - 1])
    s = cp_model.CpSolver()
    s.Solve(m)
    return [s.Value(v) for v in x]
```

The solver returns the lexicographically first prime circle for each requested n. Selected timings on a standard laptop:

$$n = 3 : 2 \text{ ms}; \quad n = 5 : 4 \text{ ms};$$

$$n = 8 : 15 \text{ ms}; \quad n = 20 : 90 \text{ ms};$$

$$n = 50 : 1.2 \text{ s}.$$

Figure 13.1 shows the prime circle of order 10 found above; Figure 13.2 is the order-16 case.

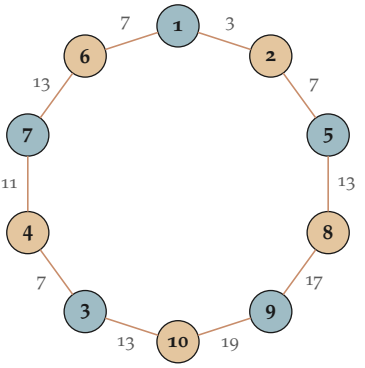

Figure 13.1: *A prime circle of order* 10 (*that is,* $n = 5$)*: the integers* 1 *through* 10 *arranged so that every pair of adjacent numbers sums to a prime. Odd cells are in blue; even cells in ochre; the adjacent sum* (*printed outside the circle*) *is always an odd prime.*

✠

Sources. Filz's original question appears as problem 749 in *Crux Mathematicorum*, volume 8 (1982), page 255. Count data for prime circles (up to rotation and reflection) are listed as

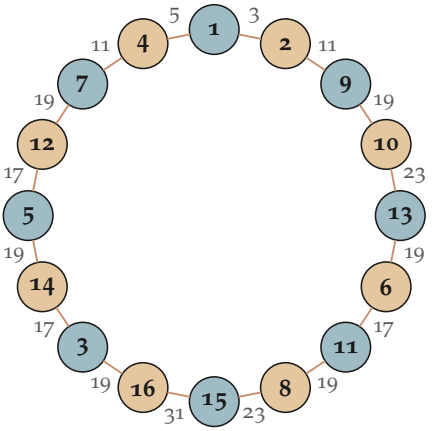

Figure 13.2: *A prime circle of order* 16 ($n = 8$).

OEIS sequence A051252: for $n = 1, 2, 3, \ldots$ the counts are $1, 1, 2, 48, 512, 4\,752, \ldots$, growing rapidly but not yet proven to be nonzero for all n. The connection to Goldbach's conjecture is sometimes asserted but is essentially superficial: if one could prove Goldbach, prime circles would *not* immediately follow, and the converse holds still less.

Chapter 14

Rolling Cubes

A standard die is placed at the bottom-left corner of a 3×3 grid with face 1 upward, face 2 towards the north, and face 3 towards the east. The die rolls, one cell at a time, onto an adjacent cell; rolling toggles three of its six faces through the usual cube rotation. A *tour* is a Hamiltonian path on the 3×3 grid: the die visits each of the nine cells exactly once. Each visited cell records the face that happens to be on top at the moment of the visit. We ask: what is the maximum, and the minimum, total of the nine recorded top faces over all legal tours?

The puzzle is an integer rolling-cube search in the tradition of Berlekamp, Conway, and Guy's *Winning Ways* (Volume 4, games on grids). Its combinatorial shape is a Hamiltonian path problem with a side-state (die orientation) that evolves locally. Both features are natural for CP-SAT: the tour is a permutation constraint; the die state is an AllowedAssignments table of $24 \cdot 4 = 96$ transition tuples.

Die orientations and their transitions

A standard die has three pairs of opposite faces: $\{1, 6\}$, $\{2, 5\}$, $\{3, 4\}$. The cube has twenty-four rotational orientations, each specified completely by the pair (top, north) with east derived. We encode each orientation as a triple of face numbers (t, n, e)

and enumerate the 24 valid triples by closure under the four rolls. A roll in direction D sends (t, n, e) to:

$$\begin{aligned} D = N: &\quad (t', n', e') = (7 - n,\ t,\ e), \\ D = S: &\quad (t', n', e') = (n,\ 7 - t,\ e), \\ D = E: &\quad (t', n', e') = (7 - e,\ n,\ t), \\ D = W: &\quad (t', n', e') = (e,\ n,\ 7 - t). \end{aligned}$$

The programming model

Variables

Number the grid cells $0, 1, \dots, 8$ row-major. Let $c_t \in \{0, \dots, 8\}$ be the cell visited at time $t \in \{0, \dots, 8\}$, with $c_0 = 0$ and all c_t distinct. Let $d_t \in \{N, S, E, W\}$ for $t \in \{0, \dots, 7\}$ be the rolling direction taken from step t to step $t + 1$, and let $o_t \in \{0, \dots, 23\}$ be the index of the die's orientation at step t, with o_0 equal to the initial orientation $(1, 2, 3)$.

Grid transitions

Express cell c as $(r, k) = (c \div 3, c \bmod 3)$ via integer division and modulo. Adjacency in the grid translates directly into equalities on (r, k): direction N requires $r_{t+1} = r_t + 1$ and $k_{t+1} = k_t$, and similarly for the other directions.

Die-rolling transitions

The triple (o_t, d_t, o_{t+1}) must belong to the precomputed transition table of size 96, imposed as an AllowedAssignments constraint.

Top-face readouts

Let $f_t \in \{1, \dots, 6\}$ be the top-face value at step t. Link f_t to o_t by a table constraint $(o_t, f_t) \in \{(i, \mathrm{top}(O_i)) : i = 0, \dots, 23\}$.

Objective

Maximise or minimise $\sum_{t=0}^{8} f_t$.

A solver in sixty lines

```
from ortools.sat.python import cp_model

OPP = {1: 6, 2: 5, 3: 4, 4: 3, 5: 2, 6: 1}

def roll(o, d):
    t, n, e = o
    if d == 'N': return (OPP[n], t, e)
    if d == 'S': return (n, OPP[t], e)
    if d == 'E': return (OPP[e], n, t)
    return (e, n, OPP[t])

def orientations():
    pool = {(1, 2, 3)}
    frontier = set(pool)
    while frontier:
        nxt = set()
        for o in frontier:
            for d in "NSEW":
                r = roll(o, d)
                if r not in pool:
                    pool.add(r); nxt.add(r)
        frontier = nxt
    return sorted(pool)

def solve(objective="max"):
    O = orientations()
    idx = {o: i for i, o in enumerate(O)}
    trans = [(i, di, idx[roll(O[i], d)])
             for i in range(24)
             for di, d in enumerate("NSEW")]
    top_tbl = [[i, O[i][0]] for i in range(24)]

    m = cp_model.CpModel()
    c = [m.NewIntVar(0, 8, "") for _ in range(9)]
    m.AddAllDifferent(c)
    m.Add(c[0] == 0)
    r = [m.NewIntVar(0, 2, "") for _ in range(9)]
    k = [m.NewIntVar(0, 2, "") for _ in range(9)]
    for t in range(9):
        m.AddDivisionEquality(r[t], c[t], 3)
        m.AddModuloEquality(k[t], c[t], 3)
```

```
    d = [m.NewIntVar(0, 3, "") for _ in range(8)]
    o = [m.NewIntVar(0, 23, "") for _ in range(9)]
    m.Add(o[0] == idx[(1, 2, 3)])
    f = [m.NewIntVar(1, 6, "") for _ in range(9)]

    for t in range(8):
        isN = m.NewBoolVar(""); isS = m.NewBoolVar("")
        isE = m.NewBoolVar(""); isW = m.NewBoolVar("")
        for v, b in zip(range(4), [isN, isS, isE, isW]):
            m.Add(d[t] == v).OnlyEnforceIf(b)
            m.Add(d[t] != v).OnlyEnforceIf(b.Not())
        m.AddBoolOr([isN, isS, isE, isW])
        m.Add(r[t + 1] == r[t] + 1).OnlyEnforceIf(isN)
        m.Add(k[t + 1] == k[t]).OnlyEnforceIf(isN)
        m.Add(r[t + 1] == r[t] - 1).OnlyEnforceIf(isS)
        m.Add(k[t + 1] == k[t]).OnlyEnforceIf(isS)
        m.Add(r[t + 1] == r[t]).OnlyEnforceIf(isE)
        m.Add(k[t + 1] == k[t] + 1).OnlyEnforceIf(isE)
        m.Add(r[t + 1] == r[t]).OnlyEnforceIf(isW)
        m.Add(k[t + 1] == k[t] - 1).OnlyEnforceIf(isW)
        m.AddAllowedAssignments(
            [o[t], d[t], o[t + 1]], trans)

    for t in range(9):
        m.AddAllowedAssignments([o[t], f[t]], top_tbl)

    total = m.NewIntVar(0, 54, "")
    m.Add(total == sum(f))
    if objective == "max": m.Maximize(total)
    else:                  m.Minimize(total)
    s = cp_model.CpSolver(); s.Solve(m)
    return s.Value(total), [s.Value(f[t]) for t in range(9)]
```

The maximum total is **37**, achieved by the tour shown in Figure 14.1; the minimum is **25**, achieved by Figure 14.2. Both are found in about 20 milliseconds.

Sources. Rolling-cube puzzles are a long-running theme in recreational mathematics. Martin Gardner's *Scientific American* column (1963, "Dice tricks") posed a number of rolling-cube questions on 3×3 and 4×4 boards; Berlekamp, Conway, and Guy's *Winning Ways for Your Mathematical Plays* (Volume 4, second edition, 2004) devotes a chapter to the

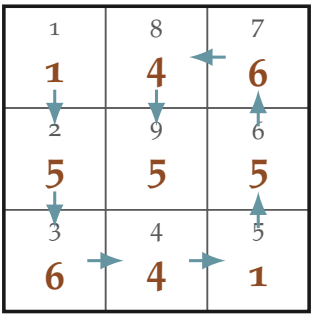

Figure 14.1: *Maximum tour* ($\sum f_t = 37$). *Numbers in warm orange show the top face at the moment of visit; the small number above indicates the visit order.*

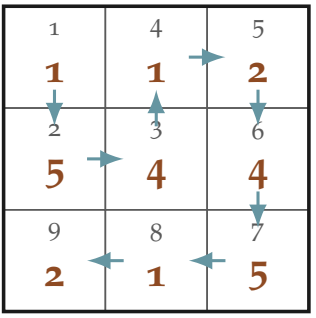

Figure 14.2: *Minimum tour* ($\sum f_t = 25$).

rolling block family, which includes the present tour puzzle as an integer variant. The specific form — Hamiltonian tour with extremal top-face sum — appears to have been popularised by I. Stewart in his *Guardian* column in the 1990s. The generalisation to 4×4 and larger grids remains a clean CP-SAT exercise; the search space grows roughly by a factor of ~ 10 per grid-edge length, and the same model solves the 4×4 extremal problem in seconds.

Chapter 15

The Riddle of the Pilgrims

In *The Canterbury Puzzles* (1907), H. E. Dudeney poses a bedroom-allocation problem styled after Chaucer's pilgrims: a company must be lodged in a two-storey dormitory, subject to a short list of the Abbot's constraints. Each floor is a 3×3 array of rooms with a central staircase, leaving eight rooms per floor and sixteen total. The Abbot's rules are:

1. every room holds between one and three pilgrims, inclusive;
2. each of the four sides of the building lodges exactly eleven pilgrims (the three rooms on that side of the lower floor together with the three rooms on that side of the upper floor);
3. the upper floor holds exactly twice as many pilgrims as the lower floor.

When the pilgrims arrive, the group turns out to number three more than was first announced, and the monks must produce a second allocation — satisfying the same three rules — for the larger party. The puzzle's entire force lies in showing that the first total and the second total are uniquely determined. The answer is that the first allocation lodges 27 pilgrims and the second 30.

✠

The programming model

The problem is a small integer-programming feasibility instance with fewer than twenty variables and a handful of linear equalities.

Variables

Index rooms by (f, r, c) with $f \in \{0, 1\}$ the floor (lower, upper) and $(r, c) \in \{0, 1, 2\}^2$ the cell. Let $x_{f,r,c} \in \{1, 2, 3\}$ be the number of pilgrims in the room at (f, r, c); the staircase cells $(f, 1, 1)$ are excluded by setting $x_{f,1,1} = 0$.

Side totals

For each of the four sides, the sum over the six rooms on that side (three cells on the lower floor, three on the upper) equals 11. Writing s_R for rows $r \in \{0, 2\}$ and s_C for columns $c \in \{0, 2\}$:

$$\sum_{f \in \{0,1\}} \sum_{c=0}^{2} x_{f,s_R,c} = 11, \qquad \sum_{f \in \{0,1\}} \sum_{r=0}^{2} x_{f,r,s_C} = 11.$$

Upper-lower ratio

$$\sum_{r,c} x_{1,r,c} = 2 \sum_{r,c} x_{0,r,c}.$$

Total feasibility

Let $T = \sum_{f,r,c} x_{f,r,c}$. The problem does not fix T: we want to know which values of T admit a valid allocation. The CP-SAT model with `parameters.enumerate_all_solutions = True` reveals exactly two: $T = 27$ and $T = 30$. These are the first and second allocations.

A solver in twenty lines

```
from ortools.sat.python import cp_model

def solve(total=None):
    m = cp_model.CpModel()
    x = {}
    for f in range(2):
        for r in range(3):
            for c in range(3):
                if (r, c) == (1, 1):
                    x[(f, r, c)] = m.NewIntVar(0, 0, "")
                else:
                    x[(f, r, c)] = m.NewIntVar(1, 3, "")
    for r in (0, 2):
        m.Add(sum(x[(f, r, c)]
                  for f in range(2)
                  for c in range(3)) == 11)
    for c in (0, 2):
        m.Add(sum(x[(f, r, c)]
                  for f in range(2)
                  for r in range(3)) == 11)
    lower = sum(x[(0, r, c)]
                for r in range(3)
                for c in range(3))
    upper = sum(x[(1, r, c)]
                for r in range(3)
                for c in range(3))
    m.Add(upper == 2 * lower)
    if total is not None:
        m.Add(lower + upper == total)
    s = cp_model.CpSolver()
    s.Solve(m)
    return {k: s.Value(v) for k, v in x.items()}
```

✠

The two allocations

Figure 15.1 displays a representative first allocation (27 pilgrims) and Figure 15.2 the unique-in-total second allocation (30 pilgrims). In each figure, the lower floor is on the left and the upper floor on the right; the staircase is

the shaded central cell; the number in each other cell is the pilgrim count; row and column sums are annotated.

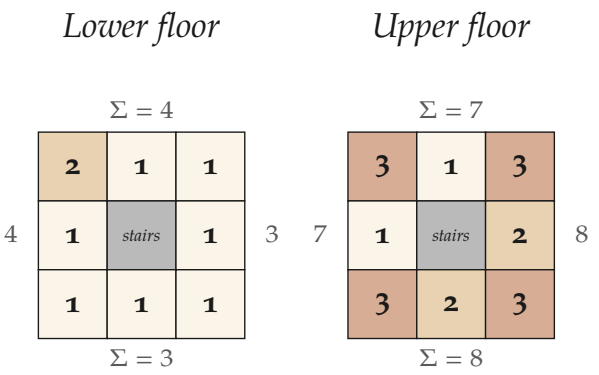

Figure 15.1: *First allocation: twenty-seven pilgrims. Lower floor on the left with nine pilgrims; upper floor on the right with eighteen. Each side of the building (top, bottom, left, right, summed across both floors) lodges eleven pilgrims.*

The existence of exactly two feasible totals (27 and 30) is the fact that resolves Dudeney's riddle: the monks' increase of three pilgrims must bring the lodgers from 27 up to 30, because no intermediate total is possible under the rules.

✠

Sources. Dudeney's *The Canterbury Puzzles* (1907) gathers puzzles framed around Chaucerian pilgrims; the riddle here appears in the chapter "The Merry Monks of Riddlewell" as Problem 25. The puzzle's statement is elementary but its mechanics — the constraint structure determining a finite set of admissible totals, with the step of three pilgrims landing on exactly the next admissible value — anticipate the formal study of integer-programming feasibility by several decades. The CP-SAT enumeration offered above is in principle overkill for a problem with thirteen variables, but it demonstrates, in a single call, that the reasoning underlying Dudeney's solution has no hidden case left to check.

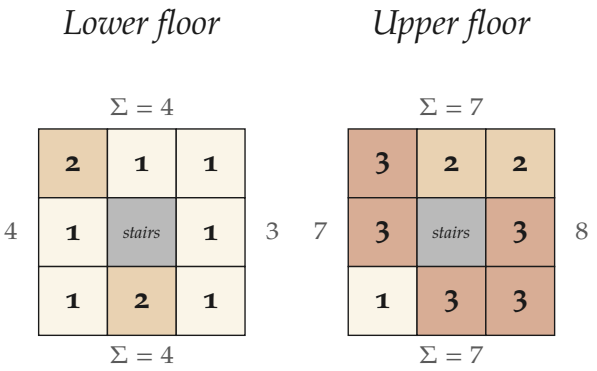

Figure 15.2: *Second allocation: thirty pilgrims (three more than the first). Lower floor has ten, upper floor twenty; the side-sum of eleven per side is preserved.*