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.*

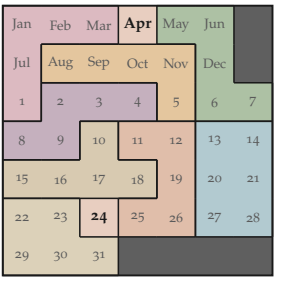

Figure 2.2: *One valid tiling. The eight pieces are distinguished by colour; the highlighted* Apr *and* 24 *remain uncovered.*

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

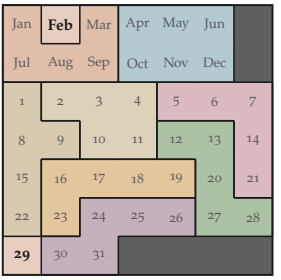

Figure 2.4: *One valid tiling for 29 February, recovered in about 95 milliseconds.*

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.