Solving puzzles through programming

Vamshi Jandhyala

August 2024

Puzzles such as Sudoku, Kakuro etc. have captivated the minds of enthusiasts worldwide. Despite their seeming simplicity, these puzzles embed intricate logical structures that challenge human solvers and computational methods alike. This book introduces approaches for solving logical puzzles by formulating them as integer and constraint programming problems. By harnessing the expressiveness of mathematical programming, we capture the essence of these puzzles, translating their rules into compact and efficient models. Our Python-based implementation using Z3 and Or-tools offers an intuitive platform for both research and educational purposes. Experimental results showcase the efficacy of our approach, emphasizing its potential as a powerful tool for puzzle enthusiasts and researchers in the domain of combinatorial optimization.

Contents

1 Hashiwokakero

Hashiwokakero, or simply Hashi, is a Japanese single-player puzzle played on a rectangular grid with no standard size. Some cells of the grid contain a circle, called island, with a number inside it ranging from one to eight, and the number of islands is denoted by n. The remaining positions of the grid are empty. The player must connect all the islands by drawing bridges between them. For this reason, the game is often referred to as building bridges. The solution to the puzzle must respect the following rules:

(i) The bridges must begin and end at distinct islands;
(ii) They must not cross any other bridges or islands;
(iii) They may only run horizontally or vertically;
(iv) At most two bridges may connect any pair of islands;
(v) The number of bridges connected to each island must be equal to the number inscribed in the circle;
(vi) Each island must be reachable from any other island.

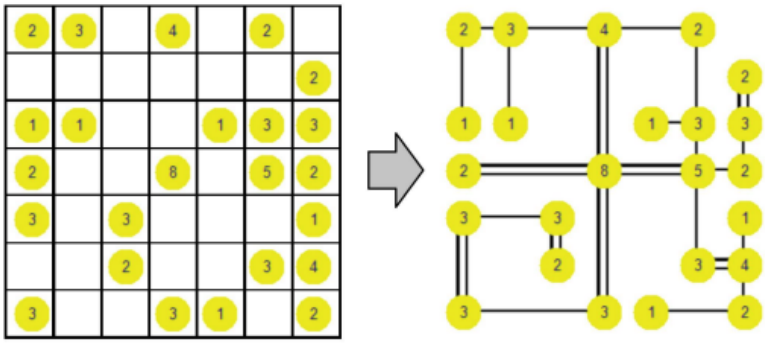


Figure 1: A Hashi puzzle (left) and a feasible solution (right)

1.1 Mathematical Model

The Hashi puzzle can be defined on an undirected graph $G = (V, E)$, where V is the set of vertices representing islands. Let d_i be the number of bridges to be constructed from island i, and $|V| = n$. Let $\delta(i)$ be the set of vertices adjacent to vertex i either horizontally or vertically. Let E be the set of all edges connecting two adjacent vertices of V. By convention, if $(i, j) \in E$, then $i < j$. Let Δ be the set of intersecting edge pairs $\{(i, j), (k, l)\} \in E$. We model Hashi as an integer linear program based on which admits a solution if and only if the Hashi puzzle is feasible. For $(i, j) \in E$, our model uses binary variables y_{ij} indicating whether two adjacent vertices i and j are connected by at least one bridge in the solution, and integer variables x_{ij} indicating the number of bridges between i and j. The formulation is then:

$$\sum_{i<k, i\in\delta(k)} x_{ik} + \sum_{j>k, i\in\delta(k)} x_{jk} = d_k, k \in V. \tag{1.1}$$

$$y_{ij} \leq x_{ij} \leq 2y_{ij}, (i, j) \in E. \tag{1.2}$$

$$y_{ij} + y_{kl} \leq 1, \{(i,j),(k,l)\} \in \Delta. \tag{1.3}$$

$$\sum_{\substack{i \in S, j \in V \setminus S \\ \vee j \in S, i \in V \setminus S}} y_{ij} \geq 1, S \subset V, 1 \leq |S| \leq n-1. \tag{1.4}$$

$$x_{ij} \in \{0,1,2\}, (i,j) \in E. \tag{1.5}$$

$$x_{ij} \in \{0,1\}, (i,j) \in E. \tag{1.6}$$

Constraints (1) force the presence of d_k bridges for each vertex k. According to constraints (2), at most two bridges can exist between any two connected vertices. These constraints also ensure consistency between the x_{ij} and y_{ij} variables. Constraints (3) prohibit intersecting bridges, and constraints (4) are strong connectivity constraints, enforcing the solution to be connected, as in the traveling salesman problem (Dantzig et al. 1954). Constraints (5) and (6) define the domains of the variables. This formulation can be strengthened by adding a valid inequality which exploits the fact that the graph induced by the positive y_{ij} variables must contain a spanning tree. It is called "weak connectivity constraint" and is found to be helpful in an algorithm in which the strong connectivity constraints (4) are relaxed.

$$\sum_{(i,j) \in E} y_{ij} \geq n-1. \tag{1.7}$$

1.2 Solved puzzles

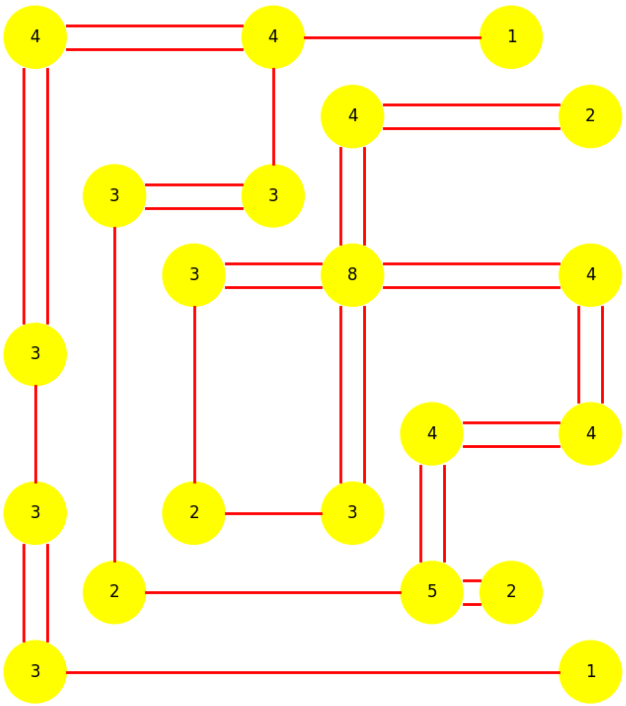


Figure 2: Solution to the Honatata puzzle

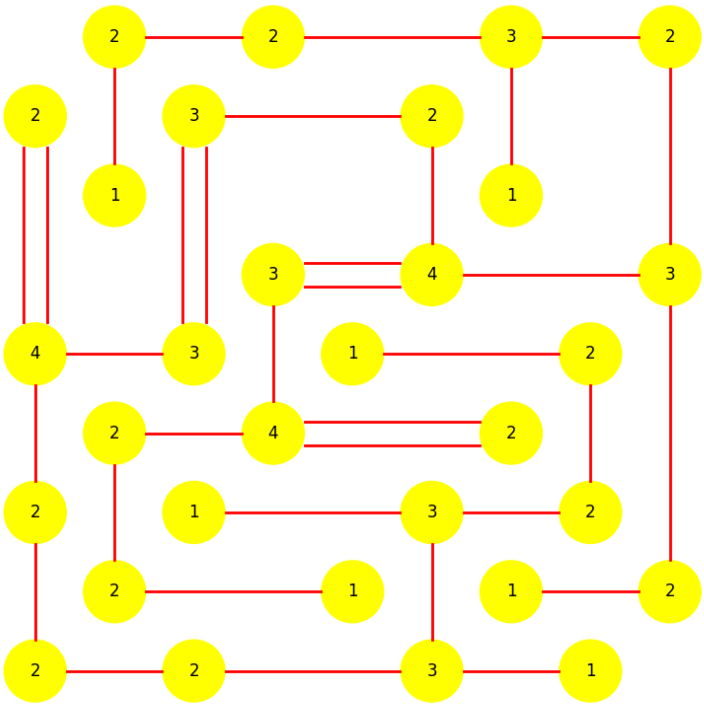


Figure 3: Solution to the Metasequoia1 puzzle

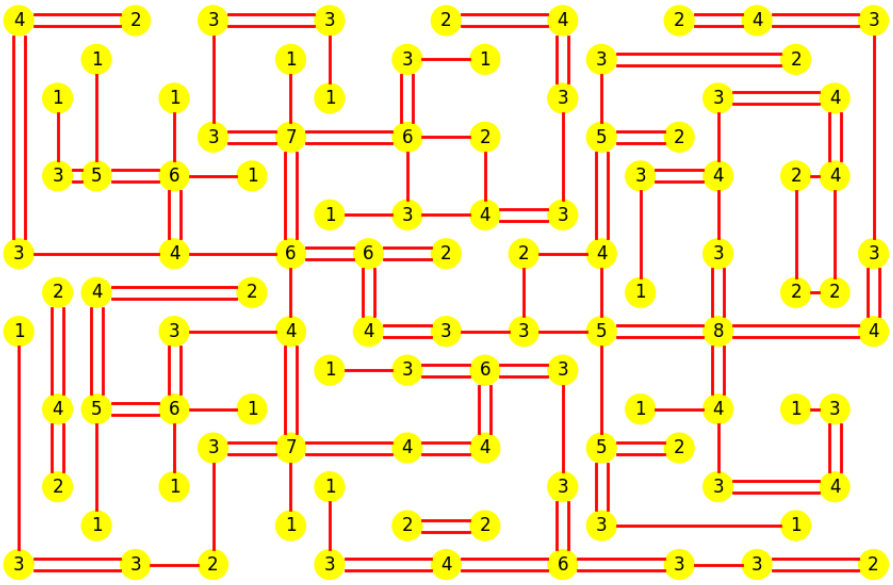


Figure 4: Solution to the Metasequoia2 puzzle

1.3 Python code

```
    import matplotlib.pyplot as plt
    import matplotlib.patches as patches
    import numpy as np
    from ortools.sat.python import cp_model
    import networkx as nx

    class Hashiwokakero:

        def __init__(self, puzzle):
            self.puzzle =  puzzle
            self.r, self.c = len(puzzle), len(puzzle[0])

            def is_path_clear(i1, j1, i2, j2):
                if i1 == i2:  # same row
                    for j in range(min(j1, j2) + 1, max(j1, j2)):
                        if self.puzzle[i1][j] != 0:
                            return False
                elif j1 == j2:  # same column
                    for i in range(min(i1, i2) + 1, max(i1, i2)):
                        if self.puzzle[i][j1] != 0:
                            return False
                return True

            horizontal_bridges, vertical_bridges = [], []
            self.G = nx.Graph()
            self.d = {}
            for i in range(self.r):
                for j in range(self.c):
                    if puzzle[i][j] != 0:
                        self.d[i*self.c + j] = puzzle[i][j]
                        for k in range(j + 1, self.c):  # search right
                            if self.puzzle[i][k] != 0 and
                                is_path_clear(i, j, i, k):
                                horizontal_bridges.append((i, j, i,
    k))
                                self.G.add_edge(i*self.c + j, i*self.c
    + k)
                        for k in range(j - 1, -1, -1):  # search left
                            if self.puzzle[i][k] != 0 and
                                is_path_clear(i, j, i, k):
                                horizontal_bridges.append((i, k, i,
    j))
                                self.G.add_edge(i*self.c + k, i*self.c
    + j)
                        for l in range(i + 1, self.r):  # search down
                            if self.puzzle[l][j] != 0 and
                                is_path_clear(i, j, l, j):
                                vertical_bridges.append((i, j, l, j))
                                self.G.add_edge(i*self.c + j, l*self.c
    + j)
                        for l in range(i - 1, -1, -1):  # search up
                            if self.puzzle[l][j] != 0 and
                                is_path_clear(i, j, l, j):
                                vertical_bridges.append((l, j, i, j))
                                self.G.add_edge(l*self.c + j, i*self.c
    + j)

            self.Delta = set()
            for vb in vertical_bridges:
                for hb in horizontal_bridges:
```

```
                if hb[1] < vb[1] < hb[3] and vb[0] < hb[0] <
vb[2]:
                    self.Delta.add(((hb[0]*self.c + hb[1],
                                     hb[2]*self.c + hb[3]),
                                    (vb[0]*self.c + vb[1],
                                     vb[2]*self.c + vb[3])))
    def plot(self, circle_radius=0.08, font_size=12,
line_color='red',
             circle_color='yellow', line_width=2):
        fig, ax = plt.subplots(figsize=(12, 12))

        # Grid spacing
        grid_spacing = 0.2  # Adjusted for larger grids
        # Plot islands (nodes)
        for i in range(self.r):
            for j in range(self.c):
                if self.puzzle[i][j] != 0:
                    circle = patches.Circle((j * grid_spacing, i *
grid_spacing),
                                            circle_radius,
fc=circle_color)
                    ax.add_patch(circle)
                    plt.text(j * grid_spacing, i * grid_spacing,
str(self.puzzle[i][j]),
                             ha='center', va='center',
fontsize=font_size)
        # Plot bridges
        for bridge in self.bridges:
            start, end, num_bridges = bridge
            if num_bridges == 0:
                continue
            # Adjusting for the gap between double bridges
            delta = 0.03 if num_bridges == 2 else 0  # Adjusted
gap
            # Horizontal bridges
            if start[0] == end[0]:
                y = start[0] * grid_spacing
                x1 = start[1] * grid_spacing + circle_radius
                x2 = end[1] * grid_spacing - circle_radius
                if num_bridges == 1:
                    plt.plot([x1, x2], [y, y], color=line_color,
lw=line_width)
                else:  # 2 bridges
                    plt.plot([x1, x2], [y - delta, y - delta],
color=line_color, lw=line_width)
                    plt.plot([x1, x2], [y + delta, y + delta],
color=line_color, lw=line_width)
            # Vertical bridges
            elif start[1] == end[1]:
                x = start[1] * grid_spacing
                y1 = start[0] * grid_spacing + circle_radius
                y2 = end[0] * grid_spacing - circle_radius
                if num_bridges == 1:
                    plt.plot([x, x], [y1, y2], color=line_color,
lw=line_width)
                else:  # 2 bridges
                    plt.plot([x - delta, x - delta], [y1, y2],
color=line_color, lw=line_width)
```

```
                    plt.plot([x + delta, x + delta], [y1, y2],
color=line_color, lw=line_width)
        ax.set_aspect('equal')
        ax.set_xlim(-0.5, self.c * grid_spacing)  # Adjusted for
reduced spacing
        ax.set_ylim(-0.5, self.r * grid_spacing)  # Adjusted for
reduced spacing
        plt.gca().invert_yaxis()  # To match matrix layout
        plt.axis('off')
        plt.show()

    def solve(self):
        self.model = cp_model.CpModel()
        self.x, self.y = {}, {}

        for (u,v) in self.G.edges:
            i, j = min(u,v), max(u,v)
            self.x[(i,j)] = self.model.NewIntVar(0, 2, 'x_%d_%d' %
(i,j))
            self.y[(i,j)] = self.model.NewIntVar(0, 1, 'y_%d_%d' %
(i,j))

        #Constraint 1
        for k in self.G.nodes:
            s = 0
            for i in self.G.neighbors(k):
                if i < k:
                    s += self.x[(i,k)]
            for j in self.G.neighbors(k):
                if j > k:
                    s += self.x[(k,j)]
            self.model.Add(s==self.d[k])

        #Constraint 2
        for (u,v) in self.G.edges:
            i, j = min(u,v), max(u,v)
            self.model.Add(self.y[(i,j)] <= self.x[(i,j)])
            self.model.Add( self.x[(i,j)] <= 2*self.y[(i,j)])

        #Constraint 3
        for ((i,j), (k,l),)  in self.Delta:
            self.model.Add(self.y[(i,j)] + self.y[(k,l)] <= 1)

        #Constraint 7
        s = 0
        for (u,v) in self.G.edges:
            i, j = min(u,v), max(u,v)
            s += self.y[(i,j)]
        self.model.Add(s >= self.G.number_of_nodes()-1)

        self.solver = cp_model.CpSolver()
        status = self.solver.Solve(self.model)
        self.bridges = None
        if status == cp_model.OPTIMAL or status ==
cp_model.FEASIBLE:
            for (i,j), v in self.x.items():
                val = int(self.solver.Value(self.x[(i,j)]))
                self.bridges.append((divmod(i,self.c),
                                     divmod(j, self.c), val))

    def plot_solution(self):
        self.solve()
        if self.bridges:
            self.plot()
```

```
            else:
                print("Could not find solution!")
    honatata = [
        [4, 0, 0, 4, 0, 0, 1, 0],
        [0, 0, 0, 0, 4, 0, 0, 2],
        [0, 3, 0, 3, 0, 0, 0, 0],
        [0, 0, 3, 0, 8, 0, 0, 4],
        [3, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 4, 0, 4],
        [3, 0, 2, 0, 3, 0, 0, 0],
        [0, 2, 0, 0, 0, 5, 2, 0],
        [3, 0, 0, 0, 0, 0, 0, 1]
    ]
    metasequoia  = [
        [0, 2, 0, 2, 0, 0, 3, 0, 2],
        [2, 0, 3, 0, 0, 2, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 3, 0, 4, 0, 0, 3],
        [4, 0, 3, 0, 1, 0, 0, 2, 0],
        [0, 2, 0, 4, 0, 0, 2, 0, 0],
        [2, 0, 1, 0, 0, 3, 0, 2, 0],
        [0, 2, 0, 0, 1, 0, 1, 0, 2],
        [2, 0, 2, 0, 0, 3, 0, 1, 0]
    ]
    metasequoia2  = [
        [4, 0, 0, 2, 0, 3, 0, 0, 3, 0, 0, 2, 0, 0, 4, 0, 0, 2, 0, 4,
    0, 0, 3],
        [0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 3, 0, 1, 0, 0, 3, 0, 0, 0, 0,
    2, 0, 0],
        [0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 3, 0, 0, 0, 3, 0,
    0, 4, 0],
        [0, 0, 0, 0, 0, 3, 0, 7, 0, 0, 6, 0, 2, 0, 0, 5, 0, 2, 0, 0,
    0, 0, 0],
        [0, 3, 5, 0, 6, 0, 1, 0, 0, 0, 0, 0, 0, 0 ,0, 0, 3, 0, 4, 0,
    2, 4, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 3, 0, 4, 0, 3, 0, 0, 0, 0, 0,
    0, 0, 0],
        [3, 0, 0, 0, 4, 0, 0, 6, 0, 6, 0, 2, 0, 2, 0, 4, 0, 0, 3, 0,
    0, 0, 3],
        [0, 2, 4, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
    2, 2, 0],
        [1, 0, 0, 0, 3, 0, 0, 4, 0, 4, 0, 3, 0, 3, 0, 5, 0, 0, 8, 0,
    0, 0, 4],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 3, 0, 6, 0, 3, 0, 0, 0, 0, 0,
    0, 0, 0],
        [0, 4, 5, 0, 6, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 4, 0,
    1, 3, 0],
        [0, 0, 0, 0, 0, 3, 0, 7, 0, 0, 4, 0, 4, 0, 0, 5, 0, 2, 0, 0,
    0, 0, 0],
        [0, 2, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 3, 0, 0, 0, 3, 0,
    0, 4, 0],
        [0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 2, 0, 2, 0, 0, 3, 0, 0, 0, 0,
    1, 0, 0],
        [3, 0, 0, 3, 0, 2, 0, 0, 3, 0, 0, 4, 0, 0, 6, 0, 0, 3, 0, 3,
    0, 0, 2]
    ]
    Hashiwokakero(honatata).plot_solution()
```

```
    Hashiwokakero(metasequoia).plot_solution()
    Hashiwokakero(metasequoia2).plot_solution()
```

2 Walls

Walls is a perfect piece of puzzle minimalism. No shading, no symbols. Just horizontal and vertical lines. The rules are incredibly simple but solving a puzzle can be spectacularly difficult. The challenge in a Walls puzzle is to fill each empty cell with either a vertical or a horizontal line, so that the number in each black cell equals the combined length of the lines ending at that cell. Lines cannot go throug the black cells. The figure below shows a Walls puzzle along with the solution.

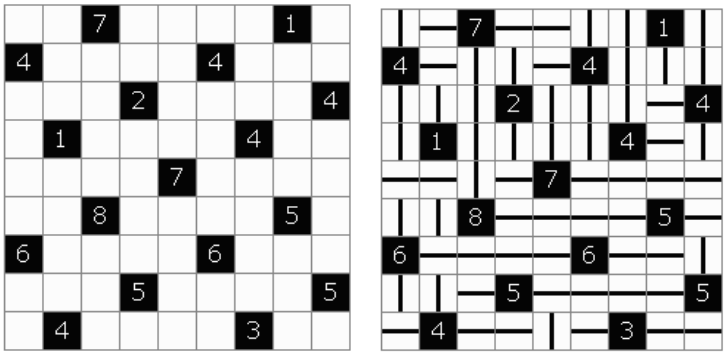


Figure 5: A Walls puzzle and its solution

2.1 Mathematical Model

We introduce binary variables v_{ij} and h_{ij} to represent a vertical or a horizontal line in cell (i, j). We have l_{ij} indicating the combined length of the lines (horizontal and vertical) ending at cell (i, j). We denote the set of filled cells on the board by F and the set of empty cells by B . We now have the following constraints:

$$h_{ij} + v_{ij} = 1, \forall (i, j) \in B \tag{2.1}$$

$$\vee_{p \in P_{ij}^{l}} \left(\sum_{(k,l) \in p_V} v_{kl} + \sum_{(k,l) \in p_H} h_{kl} = l_{ij} \right), \forall (i, j) \in F \tag{2.2}$$

$$\wedge_{p \in P_{ij}^{l+1}} \left(\sum_{(k,l) \in p_V} v_{kl} + \sum_{(k,l) \in p_H} h_{kl} \neq l_{ij} + 1 \right), \forall (i, j) \in F \tag{2.3}$$

The first constraint ensures that every cell only contains either a horizontal line or a vertical line. The second constraint and third constraints are interesting and require a more detailed explanation. We define P_{ij}^{l} as the set of all paths **accessible** from the cell (i, j) with each path containing l cells. A path p belonging to P_{ij}^{l} can contain cells which are either horizontally or vertically accessible from (i, j). The set of cells in path p which are horizontally accessible from (i, j) is denoted by p_H and the set of cells which are vertically accessible is denoted by p_V.The second constraint ensures that of all paths containing l cells accessible from (i, j) which have a combined line length of l, only one path is selected. In the figure below, for the cell $(1, 3), l = 3$. The set

P_{13}^3 contains 5 paths $[(0,3),(1,1),(1,2)]$, $[(2,3),(1,1),(1,2)]$, $[(0,3),(2,3),(3,3)]$, $[(0,3),(1,2),(2,3)]$, $[(1,2),(2,3),(3,3)]$.

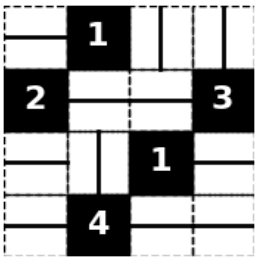


Figure 6: A Walls puzzle and its solution

For the path $[(0,3),(1,1),(1,2)]$, $p_V = \{(0,3)\}$ and $p_H = \{(1,1),(1,2)\}$.

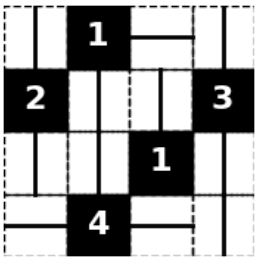


Figure 7: A Walls puzzle and its solution

From the above figure, it is easy to see that the second constraint is necessary but not sufficient to ensure that **maximimum** combined line length of lines ending in a given filled black cell (i,j) is l_{ij}. That is where the third constraint comes in. It ensures that none of the paths in P_{ij}^{l+1} have a combined line length $l+1$ which guarantees that **maximimum** combined line length of lines ending in (i,j) is indeed l_{ij}.

2.2 Notes on implementation

The crux of the model is generating the sets P_{ij}^l. We first identify all the cells accessible from a cell (i,j) in each of the four directions in increasing order of distance upto a **maximum** of l cells in each direction. Let R_{ij} be set of cells to the right of (i,j), L_{ij} be the set of cells to the left,U_{ij} be the set of cells in the upward direction and D_{ij} be the set of cells in the downward direction. We then generate the set of contiguous sub-paths for each of the above sets.We need to generate sets of sub-paths for the following reason. E.g. even if there are four cells in R_{ij}, we might use a sub-path containing two cells from R_{ij} when we are generating a path of length l. Let us denote the sets of sub-paths by R'_{ij}, L'_{ij}, D'_{ij} and U'_{ij}. The number of partitions of l gives the maximum number of ways in which a path of length l can be broken down into sub-paths in the horizontal and vertical directions. For each partition of l, we can generate one or more paths by choosing a combination of sub-paths from R'_{ij}, L'_{ij}, D'_{ij} and U'_{ij} accordingly. Here is an illustration of generating the paths in P_{13}^3 using the above procedure. We have

$$
\begin{aligned}
R_{13} &= \{\} \\
L_{13} &= \{(1,2),(1,1)\} \\
U_{13} &= \{(0,3)\} \\
D_{13} &= \{(2,3),(3,3)\}
\end{aligned}
\tag{2.1}
$$

$$
\begin{aligned}
R'_{13} &= \{\} \\
L'_{13} &= \{[(1,2)],[(1,2),(1,1)]\} \\
U_{13} &= \{[(0,3)]\} \\
D_{13} &= \{[(2,3)],[(2,3),(3,3)]\}
\end{aligned}
\tag{2.2}
$$

The partitions of 3 are $\{1,1,1\}$, $\{1,2\}$ and $\{3\}$. To use the first partition, we need 3 sub-paths of length 1.We can choose sub-path of length 1 from each of L'_{ij}, D'_{ij} and U'_{ij} which gives us the path $[(0,3),(1,2),(2,3)]$. To use the second partition, we need 1 sub-path of length 1 and 2 sub-paths of length 2. E.g. selecting $[(0,3)]$ from U'_{ij} and $[(2,3),(3,3)]$ from D'_{ij} gives us the path $[(0,3),(2,3),(3,3)]$.The other three paths can be generated in a similar fashion. None of the subsets R'_{ij}, L'_{ij}, D'_{ij} and U'_{ij} have 3 elements so the third partition cannot be used.

2.3 Solved puzzles

Here are a couple of hard puzzles from **Alex Bellos'** Puzzle Ninja that were solved using the Python implementation given in the Appendix in under a couple of seconds.

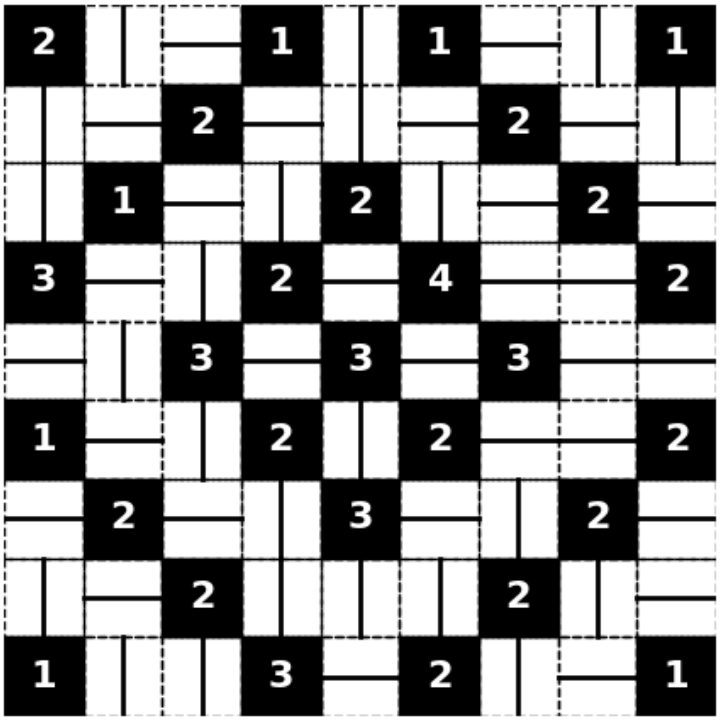


Figure 8: Puzzle 8 from Alex Bellos' Puzzle Ninja Book

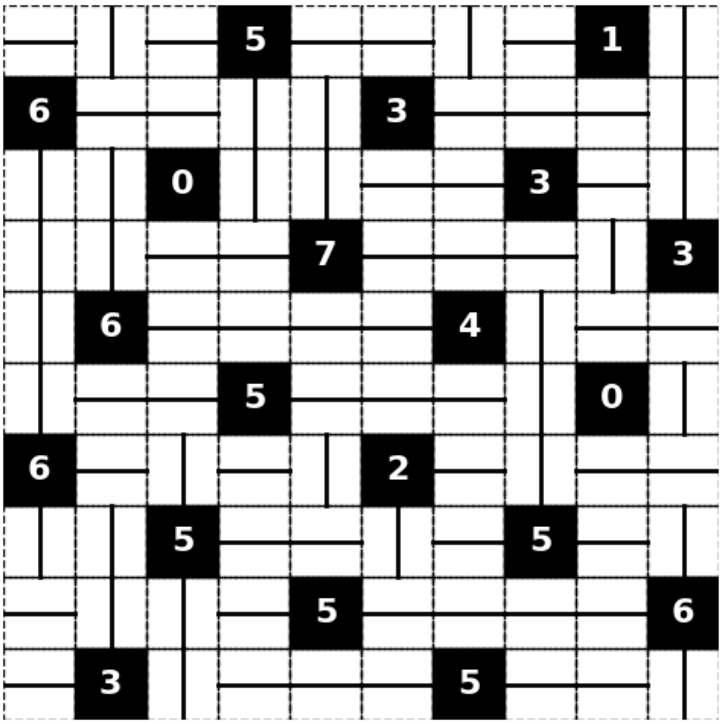


Figure 9: Puzzle 9 from Alex Bellos' Puzzle Ninja Book

2.4 Python code

```
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict
from z3 import *
from itertools import product, combinations
from collections import Counter

def partitions(n, I=1):
    yield (n,)
    for i in range(I, n//2 + 1):
        for p in partitions(n-i, i):
            yield (i,) + p

def csl(input_list):
    sublists = []
    for i in range(1, len(input_list)+1):
        sublists.append(input_list[:i])
    return sublists

def enforce_exactly_one_true(constraints):
    exactly_one_true = Or([Xor(c, And(constraints[:i] +
constraints[i+1:]))
                           for i, c in enumerate(constraints)])
    return exactly_one_true

def extract_tuples(nested_collection):
    tuples_at_lowest_level = []

    def _extract(collection):
        for item in collection:
            if isinstance(item, tuple):
                tuples_at_lowest_level.append(item)
            elif isinstance(item, (list, tuple)):
                _extract(item)
```

```
        _extract(nested_collection)
        return tuples_at_lowest_level
    class Walls:
        def __init__(self, puzzle):
            self.puzzle = puzzle
            self.r, self.c = self.puzzle.shape
            self.d = {}
            for i in range(self.r):
                for j in range(self.c):
                    if self.puzzle[i][j] != -1:
                        self.d[(i,j)] = self.puzzle[i][j]

        def accessible_cells(self, i, j, path_type="min"):
            HR, HL, VD, VU = defaultdict(list), defaultdict(list),
    defaultdict(list), defaultdict(list)
            if path_type=="min":
                d = self.puzzle[i][j]
            else:
                d = self.puzzle[i][j] + 1
            for i in range(self.r):
                for j in range(self.c):
                    for k in range(j+1, min(self.c, j+d+1)):  # search
    right
                        if self.puzzle[i][k] == -1:
                            HR[(i,j)].append((i,k))
                        else:
                            break
                    for k in range(j-1, max(-1, j-d-1), -1):  # search
    left
                        if self.puzzle[i][k] == -1:
                            HL[(i,j)].append((i,k))
                        else:
                            break
                    for l in range(i+1, min(i+d+1, self.r)):  # search
    down
                        if self.puzzle[l][j] == -1:
                            VD[(i,j)].append((l,j))
                        else:
                            break
                    for l in range(i-1, max(i-d-1,-1), -1):  # search
    up
                        if self.puzzle[l][j] == -1:
                            VU[(i,j)].append((l,j))
                        else:
                            break
            return HR, HL, VD, VU

        def contiguous_valid_paths(self, i, j, path_type="min" ):
            paths = []
            HR, HL, VD, VU = self.accessible_cells(i,j, path_type)
            if path_type=="min":
                d = self.puzzle[i][j]
            else:
                d = self.puzzle[i][j] + 1
            sets = defaultdict(list)
            eHR = [("h",c) for c in HR[(i,j)]]
            eHL = [("h",c) for c in HL[(i,j)]]
            eVD = [("v",c) for c in VD[(i,j)]]
            eVU = [("v",c) for c in VU[(i,j)]]
            for l in [csl(e) for e in [eHR,eHL,eVU,eVD]]:
```

```
                if l:
                    for c in l:
                        sets[len(c)].append(c)

            for ctr in [Counter(p) for p in partitions(d)]:
                l = []
                for k,v in ctr.items():
                    if len(sets[k]) >= v:
                        l.append([list(c) for c in
combinations(sets[k],v)])
                if l:
                    for p in product(*l):
                        ext_tuples = extract_tuples(p)
                        if len(set(ext_tuples))==d:
                            paths.append(ext_tuples)

            return paths

        def solve(self):
            self.solver = Solver()
            self.h, self.v = {}, {}
            for i in range(self.r):
                for j in range(self.c):
                    if self.puzzle[i][j] == -1:
                        self.h[(i,j)] = Int('h_%d_%d'% (i,j))
                        self.v[(i,j)] = Int('v_%d_%d'% (i,j))
                        self.solver.add(And(self.h[(i,j)] >=0,
self.h[(i,j)] <=1))
                        self.solver.add(And(self.v[(i,j)] >=0,
self.v[(i,j)] <=1))
            #Constraints
            for i in range(self.r):
                for j in range(self.c):
                    if self.puzzle[i][j] == -1:
                        self.solver.add(self.h[(i,j)] + self.v[(i,j)]
== 1)
                    if self.puzzle[i][j] != -1:
                        or_constraints = []
                        for path in self.contiguous_valid_paths(i,j):
                            s = 0
                            for t, c in path:
                                if t=="h":
                                    s += self.h[c]
                                else:
                                    s += self.v[c]
                            or_constraints.append(s == self.puzzle[i]
[j])
                        if or_constraints:
                            self.solver.add(Or(or_constraints))
                        not_constraints = []
                        for path in self.contiguous_valid_paths(i,j,
"ext"):
                            s = 0
                            for t, c in path:
                                if t=="h":
                                    s += self.h[c]
                                else:
                                    s += self.v[c]
                            not_constraints.append(s != self.puzzle[i]
[j]+1)
```

```
                    if not_constraints:
                        self.solver.add(And(not_constraints))

        self.sol = None
        if self.solver.check() == sat:
            m = self.solver.model()
            self.sol = np.zeros((self.r, self.c), dtype=np.int8)
            for i in range(self.r):
                for j in range(self.c):
                    if self.puzzle[i][j] == -1:
                        if m[self.h[(i,j)]] == 1:
                            self.sol[i][j] = -1
                        else:
                            self.sol[i][j] = -2
                    else:
                        self.sol[i][j] = self.puzzle[i][j]

    def plot(self, cell_size=0.5, font_size=16):
        fig, ax = plt.subplots(figsize=(cell_size*self.c,
cell_size*self.r))
        for y in range(self.r):
            for x in range(self.c):
                # Draw border for each cell
                ax.add_patch(plt.Rectangle((x, self.r-y-1), 1, 1,
fill=False, edgecolor='black', lw=1, linestyle='--'))  # <--
dotted line
                # Draw horizontal or vertical lines or number
                if self.sol[y, x] == -2:
                    ax.plot([x+0.5, x+0.5], [self.r-y, self.r-
y-1], color='black', lw=2)
                elif self.sol[y, x] == -1:
                    ax.plot([x, x+1], [self.r-y-0.5, self.r-
y-0.5], color='black', lw=2)
                else:
                    ax.add_patch(plt.Rectangle((x, self.r-y-1), 1,
1, color='black'))
                    ax.text(x+0.5, self.r-y-0.5, str(self.sol[y,
x]), color='white',
                            ha='center', va='center',
fontweight='bold', fontsize=font_size)

        ax.set_xlim(0, self.c)
        ax.set_ylim(0, self.r)
        ax.set_aspect('equal')
        ax.axis('off')
        plt.tight_layout()
        plt.show()

    def plot_solution(self):
        self.solve()
        if self.sol is not None:
            self.plot()
        else:
            print("Could not find solution!")

puzzle = np.array([
    [-1,1,-1,-1],
    [2,-1,-1,3],
    [-1,-1,1,-1],
    [-1,4,-1,-1]
])

Walls(puzzle).plot_solution()
```

3 L-Panel

The challenge is to tile the board with L-shaped tiles excluding the filled squares as shown in the figure below.

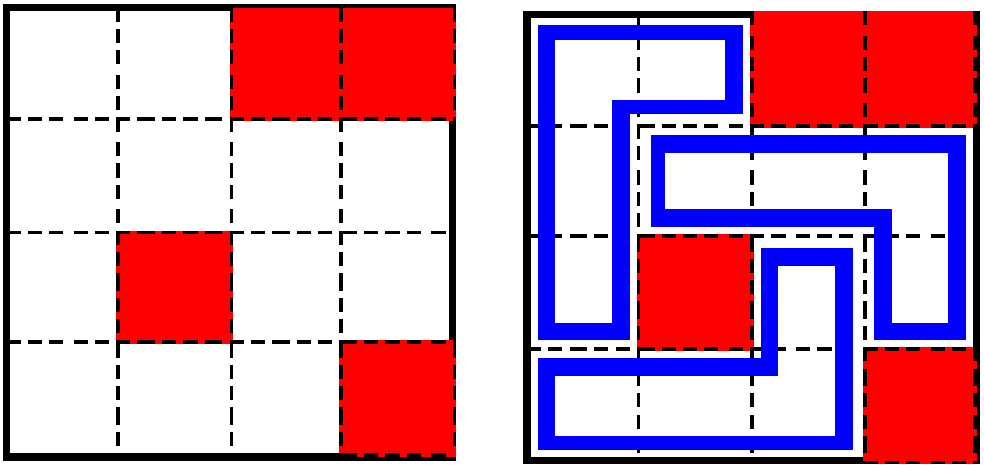


Figure 10: An L-Panel puzzle example

3.1 Solved Puzzles

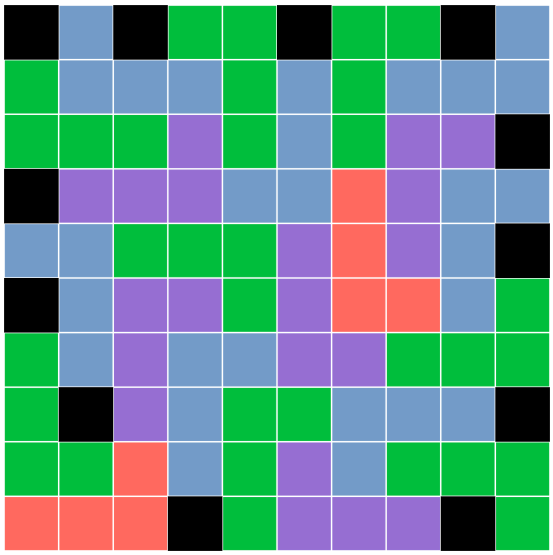


Figure 11: Puzzle 7 from Alex Bellos' Puzzle Ninja Book

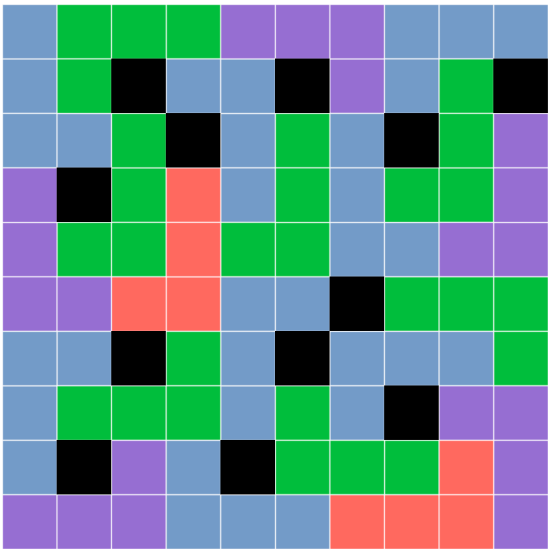


Figure 12: Puzzle 8 from Alex Bellos' Puzzle Ninja Book

3.2 Python code

```
    import matplotlib.pyplot as plt
    import matplotlib.patches as patches
    from collections import defaultdict
    from ortools.sat.python import cp_model

    class LPanel:
        def __init__(self, matrix):
            self.matrix = matrix
            self.r, self.c = len(matrix), len(matrix[0])

        def find_polyominoes_for_cells(self):
            configurations = [
                [(0, 0), (1, 0), (2, 0), (2, 1)],
                [(0, 0), (0, 1), (0, 2), (1, 2)],
                [(0, 1), (1, 1), (2, 1), (2, 0)],
                [(1, 0), (1, 1), (1, 2), (0, 0)],
                [(1, 0), (1, 1), (1, 2), (0, 2)],
                [(0, 0), (0, 1), (1, 0), (2, 0)],
                [(0, 0), (0, 1), (1, 1), (2, 1)],
                [(0, 0), (1, 0), (0, 1), (0, 2)],
            ]

            polyominoes_for_cells = defaultdict(set)
            possible_polyominoes = []
            for i in range(self.r):
                for j in range(self.c):
                    for config in configurations:
                        polyomino = [(x + i, y + j) for (x, y) in
    config]
                        if all(0 <= x < self.c and 0 <= y < self.r and
    self.matrix[x][y] != -1 for (x, y) in polyomino):

    possible_polyominoes.append(frozenset(polyomino))

            for polyomino in possible_polyominoes:
                for (i,j) in polyomino:
                    polyominoes_for_cells[(i, j)].add(polyomino)

            return polyominoes_for_cells

        def visualize_polyominoes(self, polyominoes):
```

```
        def get_adjacent_cells(x, y):
            return [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]

        def are_polyominoes_adjacent(p1, p2):
            for (x, y) in p1:
                for (ax, ay) in get_adjacent_cells(x, y):
                    if (ax, ay) in p2:
                        return True
            return False

        def get_adjacency_list(polyominoes):
            adj_list = {}
            for p in polyominoes:
                adj_list[p] = []
                for q in polyominoes:
                    if p != q and are_polyominoes_adjacent(p, q):
                        adj_list[p].append(q)
            return adj_list

        def color_polyominoes(polyominoes):
            adj_list = get_adjacency_list(polyominoes)
            colors = ['#779ECB', '#03C03C', '#966FD6', '#FF6961']
            color_assignment = {}

            for p in polyominoes:
                used_colors = {color_assignment[neighbor] for
neighbor in adj_list[p] if neighbor in color_assignment}
                available_colors = [color for color in colors if
color not in used_colors]
                color_assignment[p] = available_colors[0]

            return color_assignment

        fig, ax = plt.subplots(figsize=(10, 10))
        color_assignments = color_polyominoes(polyominoes)

        for p in polyominoes:
            color = color_assignments[p]
            for (x, y) in p:
                rect = patches.Rectangle((y, self.r - 1 - x), 1,
1, facecolor=color, edgecolor='white', linewidth=0.5)
                ax.add_patch(rect)

        for i in range(self.r):
            for j in range(self.c):
                if self.matrix[i][j] == -1:
                    ax.add_patch(patches.Rectangle((j, self.r - 1
- i), 1, 1, facecolor='black'))

        ax.set_xlim(0, self.c)
        ax.set_ylim(0, self.r)
        ax.set_aspect('equal', 'box')
        plt.axis('off')
        plt.show()

    def solve(self):
        model = cp_model.CpModel()
        vars = {}
        from random import random
        for polyominoes in
self.find_polyominoes_for_cells().values():
            for poly in polyominoes:
                vars[poly] = model.NewIntVar(0, 1, str(random()))

        for polyominoes in
self.find_polyominoes_for_cells().values():
            model.Add(sum(vars[poly] for poly in polyominoes)==1)
```

```
            solver = cp_model.CpSolver()
            status = solver.Solve(model)
            result = []
            if status == cp_model.OPTIMAL or status ==
    cp_model.FEASIBLE:
                for poly in vars.keys():
                    if solver.Value(vars[poly]) == 1:
                        result.append(poly)
            return result

        def visualize_result(self):
            result = self.solve()
            if result:
                self.visualize_polyominoes(result)
            else:
                print("No Solution")

    matrix = [
        [0, -1, 0, 0, 0, -1],
        [0, 0, 0, -1, 0, 0],
        [-1, 0, -1, 0, 0, 0],
        [0, 0, 0, 0, -1, 0],
        [0, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, -1]
    ]

    matrix7 = [
        [-1, 0, -1, 0, 0, -1, 0, 0, -1, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, -1],
        [-1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, -1],
        [-1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, -1, 0, 0, 0, 0, 0, 0, 0, -1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, -1, 0, 0, 0, 0, -1, 0],
    ]

    matrix8 = [
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, -1, 0, 0, -1, 0, 0, 0, -1],
        [0, 0, 0, -1, 0, 0, 0, -1, 0, 0],
        [0, -1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, -1, 0, 0, 0],
        [0, 0, -1, 0, 0, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, -1, 0, 0],
        [0, -1, 0, 0, -1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    ]

    dp = [
        [0, 0, 0, 0, 0, 0, -1, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, -1, 0, -1],
        [0, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0]
    ]

    LPanel(dp).visualize_result()
```

4 Marupeke

The challenge here is to fill each empty cell with either an O or an X, so that no more than two consecutive cells, either horizontally, vertically or diagonally, contain the same symbol.

4.1 Solved Puzzles

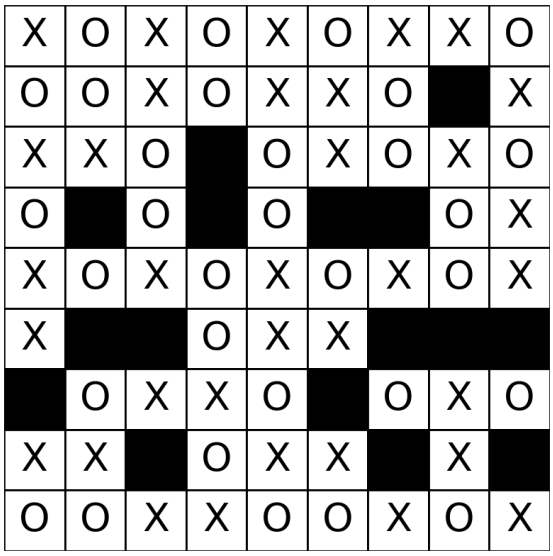


Figure 13: Puzzle 4 from Alex Bellos' Puzzle Ninja Book

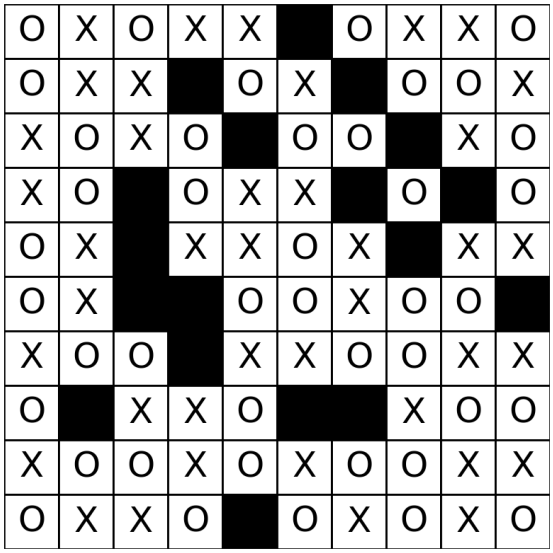


Figure 14: Puzzle 5 from Alex Bellos' Puzzle Ninja Book

4.2 Python code

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from z3 import *

class Marupeke:
    def __init__(self, matrix):
        self.matrix = matrix
```

```
            self.r, self.c = len(matrix), len(matrix[0])
        def get_all_consecutive_cells(self):
            def get_consecutive_cells(i, j):
                directions = [
                    [(0, 1), (0, 2)],  # Horizontal to the right
                    [(1, 0), (2, 0)],  # Vertical down
                    [(1, 1), (2, 2)],  # Diagonal right-down
                    [(1, -1), (2, -2)]  # Diagonal left-down
                ]

                valid_sequences = []
                for direction in directions:
                    sequence = [(i, j)]
                    valid = True

                    for dx, dy in direction:
                        x, y = i + dx, j + dy
                        if 0 <= x < self.r and 0 <= y < self.c:
                            if self.matrix[x][y] == -1:
                                valid = False
                                break
                            sequence.append((x, y))
                        else:
                            valid = False
                            break

                    if valid:
                        valid_sequences.append(sequence)

                return valid_sequences

            result = {}
            for i in range(self.r):
                for j in range(self.c):
                    if self.matrix[i][j] != -1:
                        sequences = get_consecutive_cells(i, j)
                        if sequences:
                            result[(i, j)] = sequences
            return result
        def solve(self):
            solver = Solver()
            v = {}
            for i in range(self.r):
                for j in range(self.c):
                    if self.matrix[i][j] != -1:
                        v[(i,j)] = Int("v_%d_%d"%(i,j))
                        solver.add(And(v[(i,j)] >=0, v[(i,j)] <=1))

            and_constraints = []
            for ind, seqs in self.get_all_consecutive_cells().items():
                not_constraints = []
                for cells in seqs:
                    not_constraints.append(sum(v[cell] for cell in
    cells)!=0)
                    not_constraints.append(sum(v[cell] for cell in
    cells)!=3)
                and_constraints.append(And(not_constraints))
            solver.add(And(and_constraints))

            for i in range(self.r):
                for j in range(self.c):
```

```
                if self.matrix[i][j] != -1 and self.matrix[i][j] !
= -2:
                    solver.add(And(v[(i,j)]==self.matrix[i][j]))
        print(solver)
        result = {}
        if solver.check() == sat:
            model = solver.model()
            for cell in v.keys():
                result[cell] = model[v[cell]]
        return result

    def visualize_matrix(self, cell_values):
        fig, ax = plt.subplots(figsize=(10, 10))
        fontsize = (min(fig.get_size_inches()) * 72 // max(self.r,
self.c)) * 0.5

        for i in range(self.r):
            for j in range(self.c):
                cell_color = 'white'
                # Black cell for value -1 in the matrix
                if self.matrix[i][j] == -1:
                    cell_color = 'black'
                ax.add_patch(patches.Rectangle((j, self.r - 1 -
i), 1, 1, facecolor=cell_color, edgecolor='black', linewidth=2))

                # Visualize values from the dictionary
                cell_value = cell_values.get((i, j), None)
                if cell_value == 0:
                    ax.text(j + 0.5, self.r - i - 0.5, 'O',
ha='center', va='center', fontsize=fontsize)
                elif cell_value == 1:
                    ax.text(j + 0.5, self.r - i - 0.5, 'X',
ha='center', va='center', fontsize=fontsize)
        ax.set_xlim(0, self.c)
        ax.set_ylim(0, self.r)
        ax.set_aspect('equal', 'box')
        plt.axis('off')
        plt.show()
    def visualize_result(self):
        result =  self.solve()
        if result:
            self.visualize_matrix(result)
        else:
            print("No Solution.")
matrix = [
    [-2, 1, -2, -2, -2, -2],
    [-2, -2, -2, -1, -2, 1],
    [0, -2, -2, -2, -2, 0],
    [-2, -2, -1, -2, -2, -2],
    [-1, -2, 0, -2, -2, 0],
    [1, -2, -2, 1, -2, -2],
]
matrix5 = [
    [-2, -2, -2, -2, 1, -1, -2, -2, -2, -2],
    [-2, -2, 1, -1, 0, -2, -1, 0, -2, -2],
    [-2, -2, -2, -2, -1, -2, -2, -1, -2, 0],
    [-2, -2, -1, -2, -2, -2, -1, -2, -1, 0],
    [0, 1, -1, -2, -2, -2, -2, -1, -2, -2],
```

```
        [0, -2, -1, -1, 0, -2, -2, -2, -2, -1],
        [-2, -2, 0, -1, -2, -2, 0, -2, -2, 1],
        [-2, -1, -2, -2, -2, -1, -1, -2, -2, -2],
        [-2, 0, -2, -2, -2, -2, -2, -2, -2, -2],
        [-2, 1, -2, -2, -1, 0, -2, -2, 1, -2],
    ]
    Marupeke(matrix5).visualize_result()
```

5 BlockNumber

In this puzzle, a grid is divided into blocks. Fill each block with the number(s) starting from 1 and counting upwards. So a single cell block contains just a 1. A two cell block contains 1 and 2. A three cell block contains 1, 2 and 3; and so on.

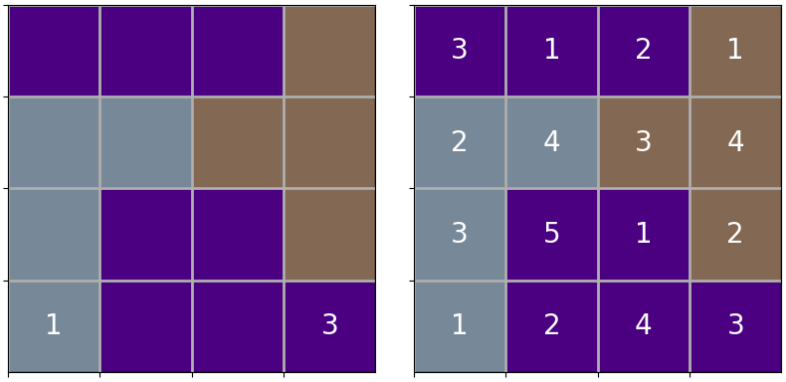


Figure 15: A BlockNumber puzzle and its solution

5.1 Solved Puzzles

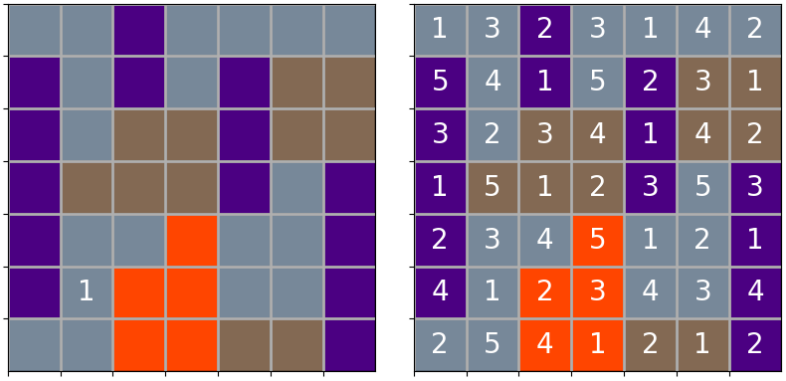


Figure 16: A BlockNumber puzzle and its solution

5.2 Python code

```
from z3 import *
import matplotlib.pyplot as plt
import networkx as nx
from itertools import combinations

class BlockNumber:
    def __init__(self, blocks, rows, cols):
        self.blocks = blocks
        self.r, self.c = rows, cols

    def solve(self):
        solver = Solver()
        v = {}
        for block in self.blocks:
            for cell, val in block.items():
                v[cell] = Int(str(cell))
                solver.add(And(v[cell] >=1, v[cell]
<=len(block.keys())))
                if val:
```

```
                    solver.add(v[cell] == val)

        for block in self.blocks:
            solver.add(Distinct([v[cell] for cell in
block.keys()]))
        def adjacent_cells(i, j):
            directions = [(-1, 0), (-1, 1), (0, 1), (1, 1), (1,
0), (1, -1), (0, -1), (-1, -1)]
            neighbors = []
            for dx, dy in directions:
                x, y = i + dx, j + dy
                if 0 <= x < self.r and 0 <= y < self.c:
                    neighbors.append((x, y))
            return neighbors
        for i in range(self.r):
            for j in range(self.c):
                for nc in adjacent_cells(i,j):
                    solver.add(v[(i,j)] != v[nc])

        result_blocks = []
        if solver.check() == sat:
            model = solver.model()
            for block in self.blocks:
                result_block = {}
                for cell in block.keys():
                    result_block[cell] = model[v[cell]]
                result_blocks.append(result_block)
        return result_blocks

    def visualize_blocks(self, block_list):
        def are_adjacent(block1, block2):
            for (i1, j1) in block1.keys():
                for (i2, j2) in block2.keys():
                    if abs(i1-i2) <= 1 and abs(j1-j2) <= 1:
                        return True
            return False
        def color_blocks(block_list):
            G = nx.Graph()
            for i, block in enumerate(block_list):
                G.add_node(i)
            for i, j in combinations(range(len(block_list)), 2):
                if are_adjacent(block_list[i], block_list[j]):
                    G.add_edge(i, j)
            coloring = nx.coloring.greedy_color(G,
strategy='largest_first')
            return coloring
        colors = ['#836953', '#778899', '#4B0082', '#FF4500']
        coloring = color_blocks(block_list)
        fig, ax = plt.subplots()
        for idx, block in enumerate(block_list):
            for (i, j), val in block.items():
                color = colors[coloring[idx] % len(colors)]
                rect = plt.Rectangle((j, i), 1, 1,
facecolor=color, edgecolor='black')
                ax.add_patch(rect)
                ax.text(j+0.5, i+0.5, str(val), ha='center',
va='center', color='white', fontsize=20)
        ax.set_xlim(0, self.c)
```

```
            ax.set_ylim(0, self.c)
            ax.set_xticks(range(self.c))
            ax.set_yticks(range(self.r))
            ax.grid(which='both', linewidth=2)
            ax.set_xticklabels([])
            ax.set_yticklabels([])
            ax.set_aspect('equal')
            plt.gca().invert_yaxis()
            plt.show()

        def visualize_solution(self):
            result = self.solve()
            if result:
                self.visualize_blocks(self.blocks)
                self.visualize_blocks(result)
            else:
                print("No solution found!")

    puzzle1 = [
        {(0,0):"",(0,1):"",(0,2):""},
        {(0,3):"",(1,2):"",(2,3):"",(1,3):""},
        {(1,0):"",(1,1):"",(2,0):"",(3,0):1},
        {(2,1):"",(2,2):"",(3,1):"",(3,2):"",(3,3):3},
    ]

    puzzle3= [
        {(0,0):"",(0,1):"",(1,1):"",(2,1):""},
        {(0,2):"",(1,2):""},
        {(0,3):"",(0,4):"",(0,5):"",(0,6):"",(1,3):""},
        {(1,0):"",(2,0):"",(3,0):"",(4,0):"",(5,0):""},
        {(2,2):"",(2,3):"",(3,1):"",(3,2):"",(3,3):""},
        {(1,4):"",(2,4):"",(3,4):""},
        {(1,5):"",(1,6):"",(2,5):"",(2,6):""},
        {(4,1):"",(4,2):"",(5,1):1,(6,0):"",(6,1):""},
        {(4,3):"",(5,2):"",(5,3):"",(6,2):"",(6,3):""},
        {(3,5):"",(4,4):"",(4,5):"",(5,4):"",(5,5):""},
        {(3,6):"",(4,6):"",(5,6):"",(6,6):""},
        {(6,4):"",(6,5):""},
    ]
    BlockNumber(puzzle3,7,7).visualize_solution()
```

6 Searchlights

The challenge is to place circles in some cells of the grid following the rules below.A number in a black cell indicates how many lights you would see from that cell looking horizontally and vertically(but not diagonally). You can see through lights but not through black cells. A cell can have atmost one light.

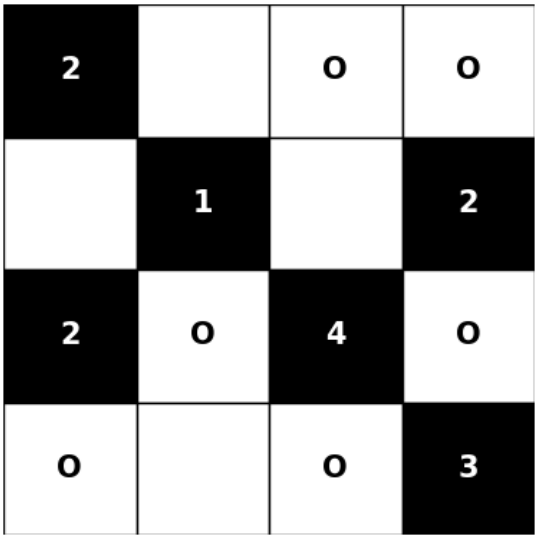


Figure 17: An examples Searchlights puzzle

6.1 Solved Puzzles

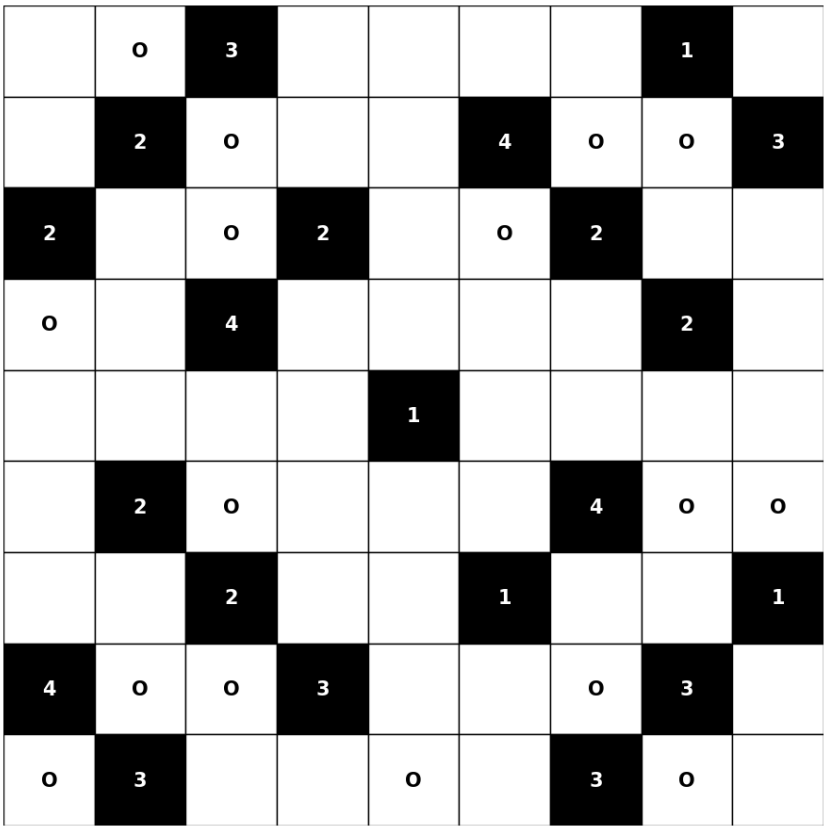


Figure 18: Puzzle 4 from Alex Bellos' Puzzle Ninja Book

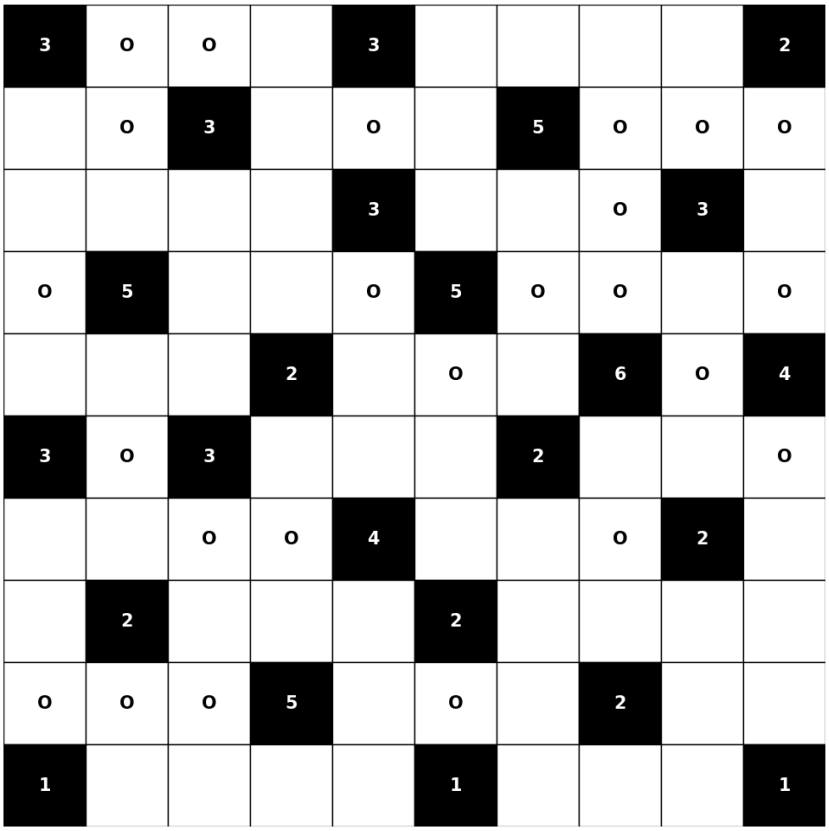


Figure 19: Puzzle 5 from Alex Bellos' Puzzle Ninja Book

6.2 Python code

```
import numpy as np
import matplotlib.pyplot as plt
from z3 import *

class Searchlights:
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.r, self.c = self.puzzle.shape
        self.d = {}
        for i in range(self.r):
            for j in range(self.c):
                if self.puzzle[i][j] != 0:
                    self.d[(i,j)] = self.puzzle[i][j]

    def accessible_cells(self, i, j):
        cells = []
        for k in range(j+1, self.c):  # search right
            if self.puzzle[i][k] == 0:
                cells.append((i,k))
            else:
                break
        for k in range(j-1, -1, -1):  # search left
            if self.puzzle[i][k] == 0:
                cells.append((i,k))
            else:
                break
        for l in range(i+1, self.r):  # search down
            if self.puzzle[l][j] == 0:
                cells.append((l,j))
```

```
                else:
                    break
            for l in range(i-1, -1, -1):  # search up
                if self.puzzle[l][j] == 0:
                    cells.append((l,j))
                else:
                    break
            return cells

        def solve(self):
            solver = Solver()
            v = {}
            for i in range(self.r):
                for j in range(self.c):
                    if self.puzzle[i][j] == 0:
                        v[(i,j)] = Int('v_%d_%d'% (i,j))
                        solver.add(And(v[(i,j)] >=0, v[(i,j)] <=1))

            #Constraints
            for i in range(self.r):
                for j in range(self.c):
                    if self.puzzle[i][j] != 0:
                        print(i,j,self.accessible_cells(i,j))
                        solver.add(And(sum([v[c] for c in
    self.accessible_cells(i,j)])==self.d[(i,j)]))

            result = {}
            if solver.check() == sat:
                model = solver.model()
                for i in range(self.r):
                    for j in range(self.c):
                        if self.puzzle[i][j] != 0:
                            result[(i,j)] = self.d[(i,j)]
                        else:
                            if model[v[(i,j)]] == 1:
                                result[(i,j)] = "O"
                            else:
                                result[(i,j)] = ""
            return result

        def visualize_grid(self, data):
            fig, ax = plt.subplots(figsize=(self.c, self.r))
            for i in range(self.r):
                for j in range(self.c):
                    facecolor = "white"  # default color
                    textcolor = "black"
                    text = ""

                    if data[(i, j)] != "O" and data[(i, j)] != "" :
                        facecolor = "black"
                        textcolor = "white"
                        text = str(data[(i, j)])
                    elif data[(i, j)] == "O":
                        text = "O"

                    ax.add_patch(plt.Rectangle((j, i), 1, 1,
    facecolor=facecolor, edgecolor='black'))
                    if text:
                        ax.text(j + 0.5, i + 0.5, text, ha='center',
    va='center', color=textcolor, fontsize=15, fontweight='bold')

            ax.set_xlim(0, self.c)
            ax.set_ylim(0, self.r)
```

```
            ax.set_aspect('equal')
            ax.axis('off')
            plt.gca().invert_yaxis()
            plt.tight_layout()
            plt.show()

        def visualize_solution(self):
            result = self.solve()
            if result:
                self.visualize_grid(result)

    puzzle = np.array([
        [2,0,0,0],
        [0,1,0,2],
        [2,0,4,0],
        [0,0,0,3]
    ])

    puzzle4 = np.array([
        [0,0,3,0,0,0,0,1,0],
        [0,2,0,0,0,4,0,0,3],
        [2,0,0,2,0,0,2,0,0],
        [0,0,4,0,0,0,0,2,0],
        [0,0,0,0,1,0,0,0,0],
        [0,2,0,0,0,0,4,0,0],
        [0,0,2,0,0,1,0,0,1],
        [4,0,0,3,0,0,0,3,0],
        [0,3,0,0,0,0,3,0,0],
    ])

    Searchlights(puzzle4).visualize_solution()
```

7 Calendar Puzzle

A-Puzzle-A-Day is a very fun and addictive puzzle that gives you a new challenge every single day of the year. All you need to do is fit these eight pieces into the calendar frame to leave one month and one day showing. Can you find your birthday? Your favorite holidays? Your anniversary? Today's date? Every day you have a new puzzle!

7.1 Python code

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

    def polyomino_rotations(polyomino):
        def rotate_polyomino(polyomino):
            rot_poly = [(square[1], -square[0]) for square in
    polyomino]
            min_x = min(rot_poly, key=lambda square: square[0])[0]
            min_y = min(rot_poly, key=lambda square: square[1])[1]
            return [(square[0] - min_x, square[1] - min_y) for square
    in rot_poly]
        rotations = [polyomino]
        for i in range(3):
            rotations.append(rotate_polyomino(rotations[-1]))
        return rotations

    def polyomino_reflections(polyomino):
        def reflect_polyomino(polyomino, axis="x"):
            ref_poly = []
            for x, y in polyomino:
                if axis == "x":
                    ref_poly.append((x, -y))
                else:
                    ref_poly.append((-x, y))
            min_x = min(ref_poly, key=lambda square: square[0])[0]
            min_y = min(ref_poly, key=lambda square: square[1])[1]
            return [(square[0] - min_x, square[1] - min_y) for square
    in ref_poly]
        return [reflect_polyomino(polyomino, "x"),
    reflect_polyomino(polyomino, "y")]

    def polyomino_translations(polyomino, h=6, w=6):
        translations = []
        for x in range(0, h+1):
            for y in range(0, w+1):
                trans_poly = [(square[0] + x, square[1] + y) for
    square in polyomino]
                max_x = max(trans_poly, key=lambda square: square[0])
    [0]
                max_y = max(trans_poly, key=lambda square: square[1])
    [1]
                if max_x <= h and max_y <= w:
                    translations.append(trans_poly)
        return translations

    def polyomino_matrix(polyomino, h=6, w=6):
```

```
    poly_mat = np.zeros((h + 1, w + 1), dtype=int)
    for coords in polyomino:
        poly_mat[coords[0], coords[1]] = 1
    return poly_mat

def date_matrix(month, day):
    month_coords = {
        "Jan": (0, 0),
        "Feb": (0, 1),
        "Mar": (0, 2),
        "Apr": (0, 3),
        "May": (0, 4),
        "Jun": (0, 5),
        "Jul": (1, 0),
        "Aug": (1, 1),
        "Sep": (1, 2),
        "Oct": (1, 3),
        "Nov": (1, 4),
        "Dec": (1, 5),
    }
    day_coords = {}
    for i in range(1, 32):
        if i % 7:
            day_coords[i] = (2 + i // 7, i % 7 - 1)
        else:
            day_coords[i] = (1 + i // 7, 6)
    month_c, day_c = month_coords[month], day_coords[day]
    board_mat = np.ones((7, 7), dtype=int)
    board_mat[0, 6] = 0
    board_mat[1, 6] = 0
    board_mat[6, 3:] = 0
    board_mat[day_c[0], day_c[1]] = 0
    board_mat[month_c[0], month_c[1]] = 0
    return board_mat

polyominoes = [
    [(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)],
    [(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 solve_puzzle(puzzle_matrix):
    model = cp_model.CpModel()
    all_variables, var_matrix_map = [], {}
    i = 0
    for polyomino in polyominoes:
        row_variables, polyomino_matrices = [], []
        orientations = polyomino_rotations(polyomino) +
polyomino_reflections(polyomino)
        for orientation in orientations:
            translations = polyomino_translations(orientation)
            for elem in translations:
                i += 1
                mat = polyomino_matrix(elem)
                polyomino_matrices.append(mat)
                var = model.NewBoolVar(f"x_{i}")
```

```
                row_variables.append(var)
                var_matrix_map[var] = mat
            all_variables.append(row_variables)

        for row in all_variables:
            s = 0
            for var in row:
                s += var
            model.Add(s == 1)

        for i in range(puzzle_matrix.shape[0]):
            for j in range(puzzle_matrix.shape[1]):
                s = 0
                for var, mat in var_matrix_map.items():
                    s += var * mat[i, j]
                model.Add(s == puzzle_matrix[i, j])

        solver = cp_model.CpSolver()
        status = solver.Solve(model)
        if status == cp_model.OPTIMAL or status == cp_model.FEASIBLE:
            return (solver, all_variables, var_matrix_map)
        else:
            print(solver.StatusName(status))
            return (None, None, None)

    def create_svg(color_polyomino_map, filename):
        square_size = 50
        padding = 10
        image_size = (7 * square_size + 6 * padding, 7 * square_size +
    6 * padding)
        dwg = svgwrite.Drawing(filename, size=image_size)
        group = dwg.add(dwg.g())
        for color, mat in color_polyomino_map.items():
            for i in range(mat.shape[0]):
                for j in range(mat.shape[1]):
                    if mat[i, j] == 1:
                        group.add(
                            dwg.rect(
                                (j * (square_size + padding), i *
    (square_size + padding)),
                                (square_size, square_size),
                                fill=svgwrite.rgb(*color),
                            )
                        )
        dwg.save()

    colors = [
        (141, 198, 63),  # Light green
        (179, 153, 255),  # Lavender
        (246, 178, 107),  # Light peach
        (128, 223, 255),  # Light blue
        (243, 234, 90),  # Light yellow
        (251, 117, 89),  # Coral
        (150, 199, 237),  # Sky blue
        (227, 147, 186),  # Light pink
    ]

    def render_puzzle_solution(month, date, filename):
        puzzle_matrix = date_matrix(month, date)
        color_polyomino_map = {}
        solver, all_variables, var_matrix_map =
    solve_puzzle(puzzle_matrix)
        if solver:
```

```
            for col, row in zip(colors, all_variables):
                for var in row:
                    if solver.Value(var) == 1:
                        color_polyomino_map[col] = var_matrix_map[var]
            create_svg(color_polyomino_map, filename)

    def render_puzzle_solutions_year():
        date_list = []
        for month in range(1, 4):
            for day in range(1, 32):
                try:
                    date = datetime.datetime(year=2023, month=month,
    day=day)
                    month_string = date.strftime("%b")
                    date_list.append((month_string, day))
                except ValueError:
                    pass
        try:
            render_puzzle_solution(month, date, month + str(date) +
    ".svg")
        except:
            print("No solution found for", month, date)

    def render_puzzle_solutions_date(month, date):
        render_puzzle_solution(month, date, month + str(date) +
    ".svg")

    render_puzzle_solutions_date("Feb", 29)
```

8 Instant Insanity

Instant Insanity is a puzzle consisting of four cubes. Each of the six faces of each cube is coloured with one of four colours: Blue, Green, Red, or White. The goal is to stack the four cubes on top of each other such that each colour appears exactly once on each of the four sides of the resulting tower. Here is a sample configuration of the cubes in the puzzle:

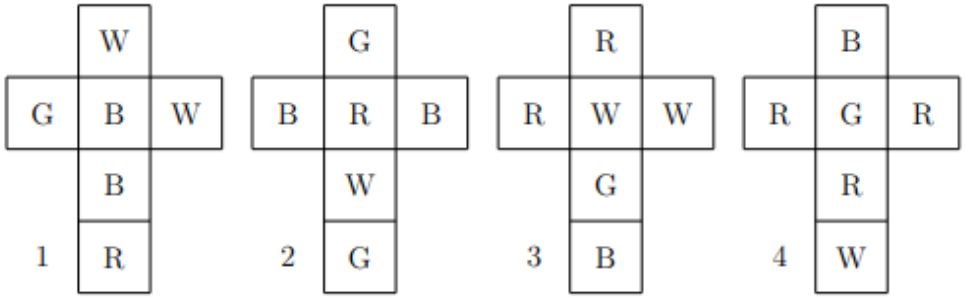


Figure 20: Example configuration of the cubes

8.1 Solution

The brute force approach is straightforward and can be extended to 5 cubes as well.

We generate all possible orientations of each cube. There are 24 orientations as the front face could be selected in 6 ways and for each selection of the front face, you have 4 ways of choosing the top face.

Generate all possible stacks of four cubes where each cube is in one of the 24 orientations.

Check if the colours on the faces making up the front, back, left and right sides of the stack are all different.

The code below gives us the following solution for the puzzle above:

Cube	Front	Back	Left	Right
1	B	W	B	R
2	W	G	R	G
3	G	R	W	B
4	R	B	G	W

The solution to the puzzle given in 1.1

8.1.1 Python Code

```
from enum import Enum
import numpy as np
```

```
from itertools import product

class Col(Enum):
    R = 1
    G = 2
    B = 3
    W = 5

cubes = [[(Col.G, Col.W),(Col.B, Col.W),(Col.B, Col.R)],
         [(Col.B, Col.B),(Col.G, Col.W),(Col.R, Col.G)],
         [(Col.R, Col.W),(Col.R, Col.G),(Col.B, Col.W)],
         [(Col.R, Col.R),(Col.G, Col.W),(Col.B, Col.R)]]

def cube_syms(cube):
    (a,b),(c,d),(e,f) = cube
    return np.array([
    [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 solution(cubes):
    n = len(cubes)
    for cubes in product(*[cube_syms(c) for c in cubes]):
        stack = np.vstack(list(cubes))
        f, b, l, r  = [stack[:,i] for i in range(n)]
        if len(set(f))== len(set(b))==len(set(l))==len(set(r))==n:
            return (f,b,l,r)

def pretty_print(sol):
    print('Cube', 'F','B', 'L','R')
    for i, (f,l,b,r) in enumerate(zip(*sol)):
        print(f'{i+1}   ', f.name, l.name, b.name, r.name)

pretty_print(solution(cubes))
```

9 Drive Ya Nuts

Remove the hexagonal pieces from the pegs and then try to put them back so that the numbers on all edges match.

Figure 21: Instant Insanity

9.1 Solution

The numbering scheme of each nut starts with 1 and proceeds in the **anti-clockwise** direction as given below:

(i) 1 6 5 4 3 2

(ii) 1 4 3 6 5 2

(iii) 1 6 4 2 5 3

(iv) 1 6 2 4 5 3

(v) 1 6 5 3 2 4

(vi) 1 4 6 2 3 5

(vii) 1 2 3 4 5 6

The algorithm proceeds as follows:

Choose one of the seven rings as the central ring.

Choose one permutation of the other 6 rings from 6! permutations.

For each of the six rings in the above permutation, generate all possible rotations i.e. 6 rotations.

Check if numbers match along the aligned edges of the rings. If the 7 rings and their edges are labelled as shown in the diagram below, then the following constraints must be satisfied:

$$\begin{aligned} r_2a = r_1a \vee r_2f = r_3b \vee r_2b = r_7f \vee \\ r_3a = r_1b \vee r_3f = r_4b \vee r_1c = r_4a \vee \\ r_4f = r_5b \vee r_1d = r_5a \vee r_5f = r_6b \vee \\ r_6a = r_1e \vee r_6f = r_7b \vee r_7a = r_1f \end{aligned} \tag{9.1}$$

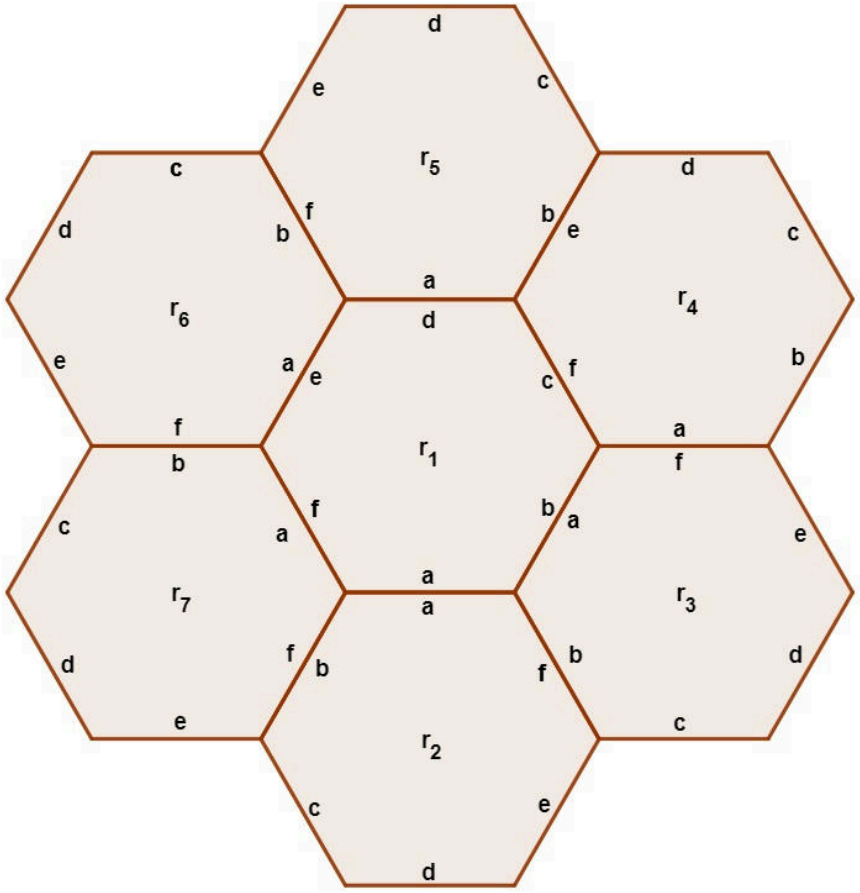


Figure 22: Configuration State

The code below gives the following solution:

Ring	a	b	c	d	e	f
r_1	1	6	2	4	5	3
r_2	1	4	6	2	3	5

r_3	6	5	3	2	4	1
r_4	2	1	4	3	6	5
r_5	4	5	6	1	2	3
r_6	5	3	1	6	4	2
r_7	3	2	1	6	5	4

The solution to the puzzle

9.2 Python Code

```
from itertools import permutations, product
from numpy import array, roll

rings = {'A':[1, 6, 5, 4, 3, 2],
         'B':[1, 4, 3, 6, 5, 2],
         'C':[1, 6, 4, 2, 5, 3],
         'D':[1, 6, 2, 4, 5, 3],
         'E':[1, 6, 5, 3, 2, 4],
         'F':[1, 4, 6, 2, 3, 5],
         'G':[1, 2, 3, 4, 5, 6]}

def rotations(ring):
    rot, rots = array(ring), []
    for i in range(6):
        rots.append(roll(rot, i))
    return rots

def solution(rings):
    for r1k, r1v in rings.items():
        r1a, r1b, r1c, r1d, r1e, r1f = r1v
        for perm in permutations(list(set(rings.keys())-set([r1k]))):
            for (r2, r3, r4, r5, r6, r7) in
product(*[rotations(rings[r]) for r in perm]):
                r2a, r2b, _, _, _, r2f = r2
                r3a, r3b, _, _, _, r3f = r3
                r4a, r4b, _, _, _, r4f = r4
                r5a, r5b, _, _, _, r5f = r5
                r6a, r6b, _, _, _, r6f = r6
                r7a, r7b, _, _, _, r7f = r7
                if r2a == r1a and r2f == r3b and r2b == r7f and \
                r3a == r1b and r3f == r4b and r1c == r4a and \
                r4f == r5b and r1d == r5a and r5f == r6b and \
                r6a == r1e and r6f == r7b and r7a == r1f:
                    return (r1,r2,r3,r4,r5,r6,r7)

print(solution(rings))
```

10 Squares Sudoku

In addition to the normal Sudoku rules, there is one additional rule for a Squares Sudoku puzzle - sum of the numbers in each **cage** should be a perfect square. Here is a hard Squares Sudoku puzzle:

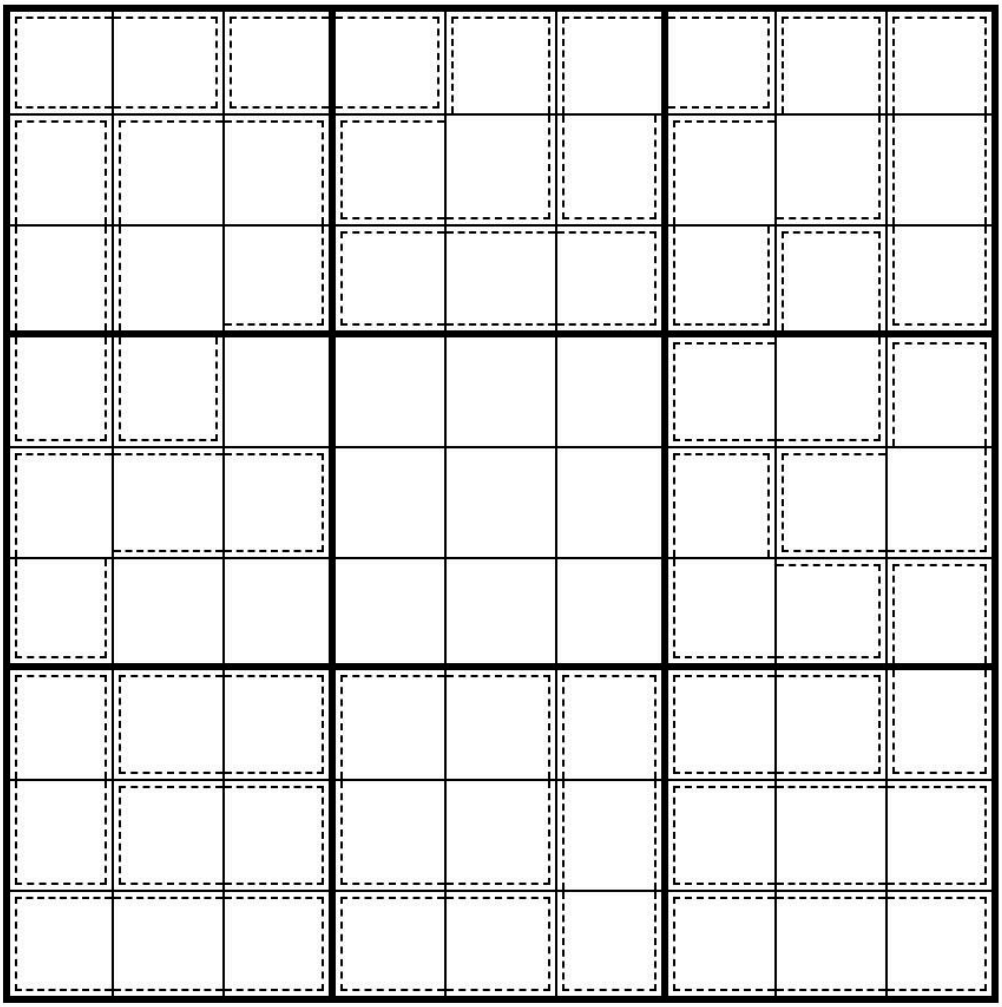


Figure 23:

10.1 Python code

```
from z3 import Solver, And, Int, Distinct, sat, If, Or

puzzle = [
    [(0,0),(0,1)],
    [(0,2),(0,3)],
    [(0,4),(1,3),(1,4)],
    [(0,5),(1,5),(0,6)],
    [(0,7),(1,7),(1,6),(2,6)],
    [(0,8),(1,8),(2,8)],
    [(1,0),(2,0),(3,0)],
    [(1,1),(2,1),(3,1),(1,2),(2,2)],
    [(2,3),(2,4),(2,5)],
    [(3,6),(3,7),(2,7)],
    [(4,0),(4,1),(4,2),(5,0)],
```

```
    [(4,6),(5,6),(5,7)],
    [(4,7),(4,8),(3,8)],
    [(5,8),(6,8)],
    [(6,0),(7,0)],
    [(6,1),(6,2)],
    [(6,3),(6,4),(7,3),(7,4)],
    [(6,5),(7,5),(8,5)],
    [(6,6),(6,7)],
    [(7,1),(7,2)],
    [(7,6),(7,7),(7,8)],
    [(8,0),(8,1),(8,2)],
    [(8,3),(8,4)],
    [(8,6),(8,7),(8,8)],
]

def print_grid(mod, x, rows, cols):
    for i in range(rows):
        print("  ".join([str(mod.eval(x[i][j])) for j in range(cols)]))

def solveSqudoku(puzzle, n):
    X = [[Int("x_%s_%s" % (i+1, j+1)) for j in range(n)] for i in
range(n)]

    # each cell contains a value in {1, ..., n}
    cells_c = [And(1 <= X[i][j], X[i][j] <= n) for i in range(n)
               for j in range(n)]

    # each row contains a digit at most once
    rows_c = [Distinct(X[i]) for i in range(n)]

    # each column contains a digit at most once
    cols_c = [Distinct([X[i][j] for i in range(n)]) for j in range(n)]

    # each 3x3 square contains a digit at most once
    sq_c = [ Distinct([ X[3*i0 + i][3*j0 + j]
                            for i in range(3) for j in range(3) ])
                for i0 in range(3) for j0 in range(3) ]

    # sum of numbers in each cage is a square
    puzz_c =[]
    for cage in puzzle:
        cs = sum([X[i][j] for i,j in cage])
        puzz_c.append(Or([(cs==k) for k in [4, 9, 16, 25]]))

    squdoku_c = cells_c + rows_c + cols_c + [And(puzz_c)] + sq_c

    s = Solver()
    s.add(squdoku_c)
    if s.check() == sat:
        m = s.model()
        print("Here is the solution")
        print_grid(m, X, n, n)
    else:
```

```
        print("Failed to solve the puzzle")

solveSqudoku(puzzle, 9)
```

Here is the solution:

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

11 Calcudoku

Calcudoku is just like Sudoku - you must enter numbers into a grid in such a way so that no number is repeated in any row or column. But Calcudoku puzzles have an added mathematical component! Each grid is split up into smaller sections of 2 or more squares, and each of those sections has an arithmetic equation attached to it (either addition, subtraction, multiplication or division). You must complete the grid so that the numbers is each section equal the mathematical formula assigned to it.Here is a hard Calcudoku puzzle

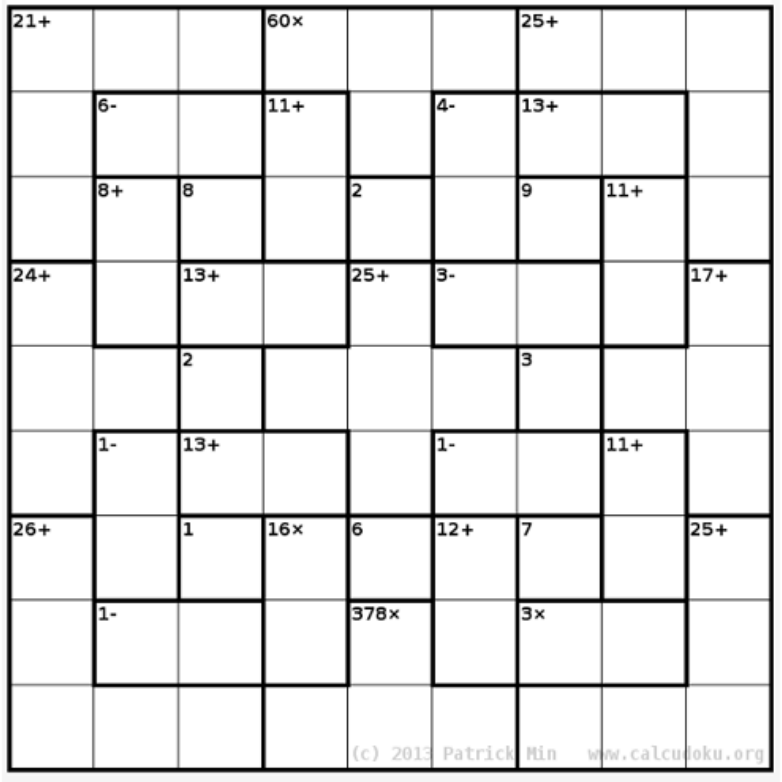


Figure 24:

11.1 Python code

```
from z3 import Solver, And, Int, Distinct, sat, If
from functools import reduce
import operator

def Max(x):
    return reduce(lambda a, b: If(a > b, a, b), x)

def cd_div(v, *x):
    m = Max(x)
    m /= reduce(operator.mul, (If(i != m, i, 1) for i in x))
    return m == v

def cd_sub(v, *x):
    m = Max(x)
    m -= sum(If(i != m, i, 0) for i in x)
    return m == v
```

```
def print_grid(mod, x, rows, cols):
    for i in range(rows):
        print("  ".join([str(mod.eval(x[i][j])) for j in range(cols)]))

# This is the encoding of the puzzle
def hardest_calcudoku(X):
    return [
        X[0][0] + X[0][1] + X[0][2] + X[1][0] + X[2][0] == 21,
        X[0][3] * X[0][4] * X[0][5] * X[1][4] == 60,
        X[0][6] + X[0][7] + X[0][8] + X[1][8] + X[2][8] == 25,
        cd_sub(6, X[1][1], X[1][2]),
        X[1][3] + X[2][3] == 11,
        cd_sub(4, X[1][5], X[2][5]),
        X[1][6] + X[1][7] == 13,
        X[2][1] + X[3][1] == 8,
        X[2][2] == 8,
        X[2][4] == 2,
        X[2][6] == 9,
        X[2][7] + X[3][7] == 11,
        X[3][0] + X[4][0] + X[5][0] + X[4][1] == 24,
        X[3][2] + X[3][3] == 13,
        X[3][4] + X[4][4] + X[5][4] + X[4][5] + X[4][3]== 25,
        cd_sub(3, X[3][5], X[3][6]),
        X[3][8] + X[4][8] + X[5][8] + X[4][7] == 17,
        X[4][2] == 2,
        X[4][6] == 3,
        cd_sub(1, X[5][1], X[6][1]),
        X[5][2] + X[5][3] == 13,
        cd_sub(1, X[5][5], X[5][6]),
        X[5][7] + X[6][7] == 11,
        X[6][0] + X[7][0] + X[8][0] + X[8][1] + X[8][2] == 26,
        X[6][2] == 1,
        X[6][3] * X[7][3] == 16,
        X[6][4] == 6,
        X[6][5] + X[7][5] == 12,
        X[6][6] == 7,
        X[6][8] + X[7][8] + X[8][8] + X[8][7] + X[8][6] == 25,
        cd_sub(1, X[7][1], X[7][2]),
        X[7][4] * X[8][4] * X[8][3] * X[8][5] == 378,
        X[7][6] * X[7][7] == 3,
    ]

def solveCalcudoku(puzzle, n):
    X = [[Int("x_%s_%s" % (i+1, j+1)) for j in range(n)] for i in
range(n)]

    # each cell contains a value in {1, ..., 9}
    cells_c = [And(1 <= X[i][j], X[i][j] <= n) for i in range(n)
               for j in range(n)]

    # each row contains a digit at most once
    rows_c = [Distinct(X[i]) for i in range(n)]
```

```
    # each column contains a digit at most once
    cols_c = [Distinct([X[i][j] for i in range(n)]) for j in range(n)]

    calcudoku_c = cells_c + rows_c + cols_c + list(map(And, puzzle(X)))

    s = Solver()
    s.add(calcudoku_c)
    if s.check() == sat:
        m = s.model()
        print("Here is the solution")
        print_grid(m, X, n, n)
    else:
        print("Failed to solve the puzzle")

solveCalcudoku(hardest_calcudoku, 9)
```

11.2 Solution

Here is the solution

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

12 Unusual Crossword

Here is an unusual crossword based on the first 30 digits of π.

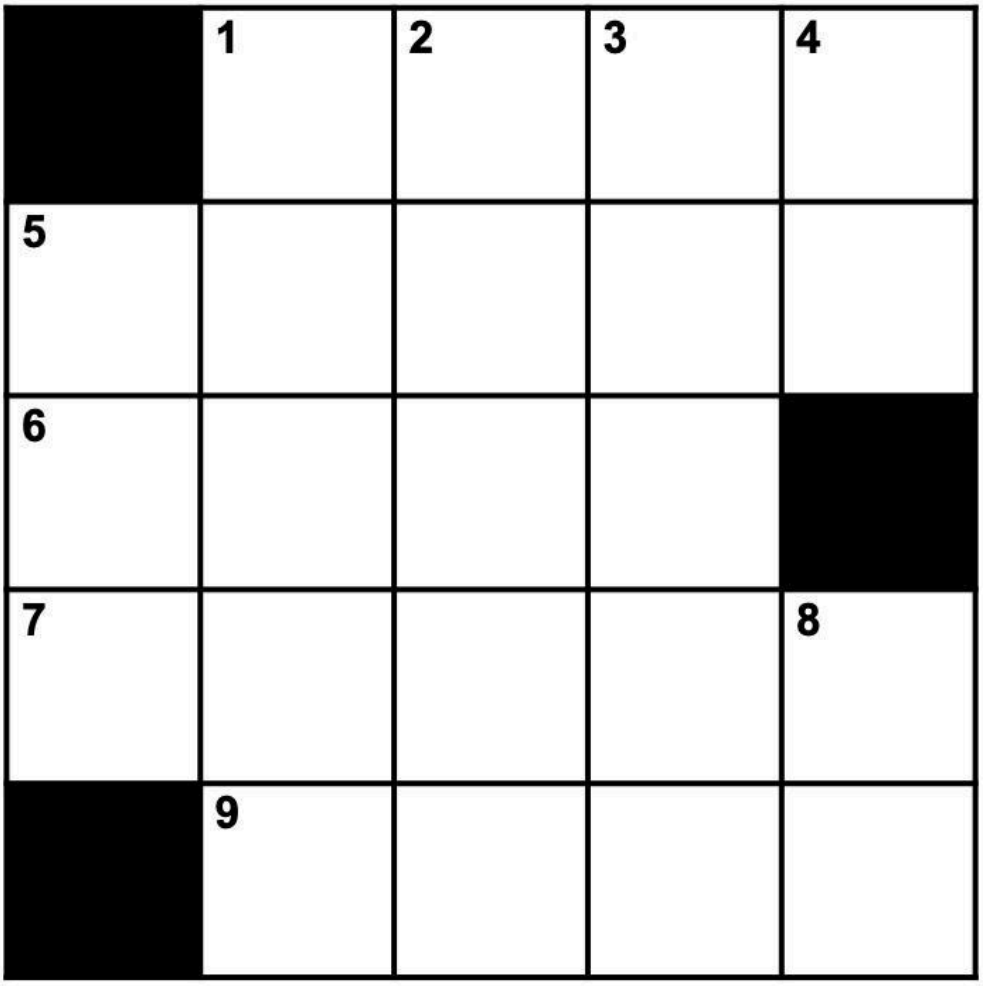


Figure 25: The Pi Crossword

12.1 Across

1.A contiguous subsequence of digits from

(iii) 14159265358979323846264338328

distinct from all the other answers (ignore the decimal point).

5.Same clue as above.

6.Same clue as above.

7.Same clue as above.

9.Same clue as above.

12.2 Down

1.Same clue as above.

2.Same clue as above.

3.Same clue as above.

4.Same clue as above.

5.Same clue as above.

8.Same clue as above.

Source. Composed by Johan de Ruiter for Pi Day, March 14, 2021;

12.3 Python code

Here is the code for solving the puzzle:

```
from itertools import product
pi30 = "31415926535897932384626433832 8"
ss = {}
for l in range(2, 6):
    ss[l] = [pi30[i:i+l] for i in range(0,30-l+1)]

for a1,a5,a6,a7,a9 in product(*[ss[4],ss[5],ss[4],ss[5],ss[4]]):
    d1 = "".join([a1[0],a5[1],a6[1],a7[1],a9[0]])
    d2 = "".join([a1[1],a5[2],a6[2],a7[2],a9[1]])
    d3 = "".join([a1[2],a5[3],a6[3],a7[3],a9[2]])
    d5 = "".join([a5[0],a6[0],a7[0]])
    d8 = "".join([a1[3],a5[4]])
    d9 = "".join([a7[4],a9[3]])
    if a1 != a6 and a6!= a9 and a1 != a9 and a5 != a7 and \
        d1 != d2 and d1 != d3 and d2 != d3 and \
        d1 != a5 and d1 != a7 and \
        d2 != a5 and d2 != a7 and \
        d3 != a5 and d3 != a7 and \
        d8 != d9 and \
        d5 in ss[3] and \
        d1 in ss[5] and \
        d2 in ss[5] and \
        d3 in ss[5] and \
        d8 in ss[2] and \
        d9 in ss[2]:
        print("x" + a1)
        print(a5)
        print(a6 +"x")
        print(a7)
        print("x"+a9)
```

Here is the solution:

```
x 2 6 4 3
2 6 5 3 5
6 4 3 3 x
5 3 5 8 9
x 3 8 3 2
```

13 The Riddle of the Pilgrims

The Canterbury Puzzles is a delightful collection of posers based on the exploits of the same group of pilgrims introduced by Geoffrey Chaucer in The Canterbury Tales. The anthology was compiled by the English puzzlist Henry Ernest Dudeney and first published in 1907. All the puzzles are mathematical in nature and many of them may be used to illustrate O.R. techniques. The following riddle, taken from the chapter entitled 'The Merry Monks of Riddlewell' is a classical I.P. allocation problem.

> One day, when the monks were seated at their repast, the Abbot announced that a messenger had that morning brought news that a number of pilgrims were on the road and would require their hospitality. "You will put them," he said, "in the square dormitory that has two floors with eight rooms on each floor. There must be eleven persons sleeping on each side of the building, and twice as many on the upper floor as the lower floor. Of course every room must be occupied, and you know my rule that not more than three persons may occupy the same room." I give a plan of the two floors, from which it will be seen that the sixteen rooms are approached by a well staircase in the centre. After the monks had solved this little problem of accommodation, the pilgrims arrived, when it was found that they were three more in number than was at first stated. This necessitated a reconsideration of the question, but the wily monks succeeded in getting over the new difficulty without breaking the Abbot's rules. The curious point of this puzzle is to discover the total number of pilgrims.

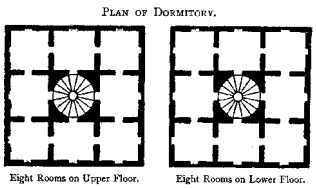


Figure 26:

13.1 Model

The monks were required to perform two allocations of pilgrims each fulfilling the Abbot's requirements and with a difference of three pilgrims in total between each allocation. On their behalf, we therefore define variables as follows

$$X_{ijkm} \in \mathbb{Z}+, \text{for } \; i = 1...n, j = 1...f, k = 1...r, m = 1...o \quad (13.1)$$

where $n = 2$ (allocations), $f = 2$ (floors), $r = 3$ (rows) and $c = 3$ (columns).

Maximize/minimize the number of pilgrims in the final allocation. This is to demonstrate that the solution to the puzzle is unique.

$$\max \sum_{j=1}^{f} \sum_{k=1}^{r} \sum_{m=1}^{c} X_{2jkm} \quad (13.2)$$

(i) Three more pilgrims in final allocation than in initial allocation

$$\sum_{j=1}^{f}\sum_{k=1}^{r}\sum_{m=1}^{c} X_{1jkm} + 3 = \sum_{j=1}^{f}\sum_{k=1}^{r}\sum_{m=1}^{c} X_{2jkm} \tag{13.3}$$

(ii) Twice as many pilgrims on upper floor than lower floor in both allocations

$$2\sum_{k=1}^{r}\sum_{m=1}^{c} X_{i1km} = \sum_{k=1}^{r}\sum_{m=1}^{c} X_{i2km}, \text{ for } \; i = 1...n \tag{13.4}$$

(iii) Eleven pilgrims in first and third rows (i.e. front and back sides)

$$\sum_{j=1}^{f}\sum_{m=1}^{c} X_{ijkm} = 11, \text{ for } \; i = 1...n, k = 1...r, k \neq 2 \tag{13.5}$$

(iv) Eleven pilgrims in first and third columns (i.e. left and right sides)

$$\sum_{j=1}^{f}\sum_{k=1}^{r} X_{ijkm} = 11, \text{ for } \; i = 1...n, m = 1...c, m \neq 2 \tag{13.6}$$

(v) Each room is atleast oocupied by at least one and no more than three pilgrims

$$1 \leq X_{ijkm} \leq 3, \text{ for } \; i = 1...n, j = 1...f, k = 1...r, m = 1...c, k \neq 2 \vee m \neq 2 \tag{13.7}$$

(vi) No pilgrims allocated to the center cells (i.e. well stair case)

$$X_{ij22} = 0, \text{for } \; i = 1...n, j = 1...f \tag{13.8}$$

13.2 Python code

The Python code for solving the puzzle using Google OR-Tools library is given below:

```
from ortools.linear_solver import pywraplp

def riddle_of_pilgrims():
    n, f, r, c = 2, 2, 3, 3
    solver = pywraplp.Solver.CreateSolver('SCIP')
    x = {(i,j,k,m): solver.IntVar(0, 3, 'x[%i][%i][%i][%i]' %
(i,j,k,m))
              for m in range(c) for k in range(r)
              for j in range(f) for i in range(n)}

    solver.Add(sum([x[(0,j,k,m)] for j in range(f)
        for k in range(r) for m in range(c)]) + 3 ==
        sum([x[(1,j,k,m)] for j in range(f)
        for k in range(r) for m in range(c)]))

    for i in range(n):
```

```
        solver.Add(2*sum([x[(i,0,k,m)] for k in range(r) for m in
range(c)]) ==
            sum([x[(i,1,k,m)] for k in range(r) for m in range(c)]))

    for i in range(n):
        for k in set(range(r)) - {1}:
            solver.Add(sum([x[(i,j,k,m)] for j in range(f) for m in
range(c)]) == 11)

    for i in range(n):
        for m in set(range(c)) - {1}:
            solver.Add(sum([x[(i,j,k,m)] for j in range(f) for k in
range(r)]) == 11)

    for i in range(n):
        for j in range(f):
            for k in set(range(c)):
                for m in set(range(c)):
                    if (k,m) != (1,1):
                        solver.Add(x[(i,j,k,m)] >= 1)
                        solver.Add(x[(i,j,k,m)] <= 3)

    for i in range(n):
        for j in range(f):
            solver.Add(x[(i,j, 1, 1)] == 0)

    solver.Minimize(sum([x[(1,j,k,m)] for j in range(f)
                        for k in range(r) for m in range(c)]))

    status = solver.Solve()
    if status == pywraplp.Solver.OPTIMAL:
        return solver.Objective().Value()
    else:
        return -1

print(riddle_of_pilgrims())
```

Using the code above, we see that the total number of pilgrims is **30**.

14 The Langford Problem

In combinatorial mathematics, a **Langford pairing**, also called a **Langford sequence**, is a permutation of the sequence of $2n$ numbers $1, 1, 2, 2, \ldots, n, n$ in which the two 1s are one unit apart, the two 2s are two units apart, and more generally the two copies of each number k are k units apart. Langford pairings are named after C. Dudley Langford, who posed the problem of constructing them in 1958. **Langford's problem** is the task of finding Langford pairings $L(n)$ for a given value of n.

14.1 Beautiful analytical solution

For the positive cases ($n = 4k$ or $4k + 3$) an algorithm for calculating the sequence can be found [here](https://susam.in/blog/langford-pairing.html). This beautiful algorithm was discovered by Roy Davies in **1959**.

Here are the details, where R denotes the reversal of a sequence.

$$\begin{aligned} x &= Ceiling[n/4] \\ \{a, b, c, d\} &= \{2x - 1, 4x - 2, 4x - 1, 4x\} \\ p &= \text{odds in } [1, a - 1] \\ q &= \text{evens in } [2, a - 1] \\ r &= \text{odds in } [a + 2, b - 1] \\ s &= \text{evens in } [a + 1, b - 1] \end{aligned} \tag{14.1}$$

If 4 divides n, the sequence is
$\{R[s], R[p], b, p, c, s, d, R[r], R[q], b, a, q, c, r, a, d\}$.

If $n \equiv 3 (\text{mod } 4)$, it is $\{R[s], R[p], b, p, c, s, a, R[r], R[q], b, a, q, c, r\}$.

The Python code implementing the above algorithm is given below

```
from math import ceil

def R(l):
    return list(reversed(l))

def langford_davies(n):
    x = ceil(n/4)
    a, b, c, d = 2*x-1, 4*x-2, 4*x-1, 4*x
    p = [i for i in range(1, a) if i % 2==1]
    q = [i for i in range(2, a) if i % 2==0]
    r = [i for i in range(a+2, b) if i % 2==1]
    s = [i for i in range(a+1, b) if i % 2==0]
    if n%4 == 0:
        return R(s) + R(p) + [b] + p + [c] + s + [d] + R(r) + R(q) +
[b,a] + q + [c]  + r + [a, d]
    if n%4 == 3:
        return R(s) + R(p) + [b] + p + [c] + s + [a] + R(r) + R(q) +
[b,a] + q + [c]  + r
    return None
```

14.2 Model using Integer Programming

Another way to solve the Langford problem is to treat it as a set covering problem. To visualize this we make use of the following array for $L(3)$:

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

To solve the problem, we need to select one row for the 1's in the sequence, one row for the 2's and one row for the 3's, such that if we stack these rows on top of each other, no column contains more than one number.

In case of $L(n)$ it is easy to see that the number of columns in the matrix will be $2n$ and the number of rows will be $r = 2n - 2 + ... + n - 1$. Let $\{x_i \mid 1 \leq i \leq r\}$ be the set of decision variables, one for each row in the matrix such that $x_i \in \{0, 1\}$. We have the following constraints:

(i) We choose only 1 row among all rows containing the number k in

the matrix where $1 \leq k \leq n$. 2. For each column, we choose only 1 row among all rows containing non zero values.

14.3 Python code using integer programming

The Python code implementing the above model using the Google OR-Tools library is given below:

```
from ortools.linear_solver import pywraplp

def langford_ip(n):
    solver = pywraplp.Solver.CreateSolver('SCIP')

    n_rows, n_cols = sum(range(n-1, 2*n-1)), 2*n
    matrix = [[0 for j in range(n_cols)] for i in range(n_rows)]
    out = [0 for i in range(n_cols)]

    # setting up the covering matrix
    j = 0
    for i in range(n):
        for k in range(2*n-i-2):
            matrix[j][k] = i + 1
```

```
            matrix[j][k+i+2] = i + 1
            j += 1

    x = [solver.IntVar(0, 1, 'x[%i]' % j) for j in range(n_rows)]

    # row constraints
    j = 0
    for i in range(n):
        solver.Add(sum([x[k] for k in range(j, j + 2*n-i-2)])==1)
        j += 2*n-i-2

    # column constraints
    for i in range(n_cols):
        inds = []
        for j in range(n_rows):
            if matrix[j][i]:
                inds.append(j)
        solver.Add(sum([x[k] for k in inds])==1)

    solver.Minimize(sum([x[i] for i in  range(n_rows)]))

    status = solver.Solve()
    if status == pywraplp.Solver.OPTIMAL:
        for i in range(n_rows):
            if x[i].solution_value():
                for j in range(n_cols):
                    out[j] += matrix[i][j]
        return out
    else:
        return None
```

14.4 Model using Constraint Programming

Let (x_{is}, x_{ie}) be the tuple of decision variables indicating the starting and ending position of number i in the Langford sequence. The decision variables need to satisfy the following constraints:

$$\begin{aligned} 1 \leq x_{is}, x_{ie} \leq 2n, 1 \leq i \leq n \\ x_{ie} = x_{is} + (i+1), 1 \leq i \leq n \end{aligned} \tag{14.1}$$

All members of the set $\{x_{it} \mid 1 \leq i \leq n, t \in \{s, e\}\}$ are different.

14.5 Python code using constraint programming

Here is the Python code implementing the above model using the fantastic Google **OR-Tools** library:

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

def langford_seq_checker(seq):
    if not seq:
        return False
    pos_map = defaultdict(list)
```

```
    for i, n in enumerate(seq):
        pos_map[n].append(i)
    for n,p in pos_map.items():
        if len(p) != 2:
            return False
        if (p[1] - p[0]) != n + 1:
            return False
    return True

def langford_cp(n):
    model = cp_model.CpModel()
    x = [[model.NewIntVar(1, 2*n, 'x[%i][%i]' % (i,j)) for j in
range(2)] for i in range(n)]

    model.AddAllDifferent([x[i][j] for i in range(n) for j in
range(2)])
    for i in range(n):
        model.Add(x[i][1] - x[i][0] == i+2)

    solver = cp_model.CpSolver()
    status = solver.Solve(model)
    if status == cp_model.OPTIMAL or status == cp_model.FEASIBLE:
        out = [0]*(2*n+1)
        for i in range(n):
            for j in range(2):
                out[solver.Value(x[i][j])] = i+1
        return out[1:]
    else:
        return None
```

14.6 Solution

Here is the sequence $L(100)$ calculated using the above code:

```
[51, 79, 80, 82, 1, 30, 1, 64, 70, 87, 95, 50, 21, 66, 33, 4, 29, 97,
20, 15, 4, 69, 22,
52, 59, 28, 100, 81, 46, 26, 36, 57, 49, 8, 21, 15, 30, 91, 31, 20, 83,
86, 8, 34, 23, 22,
29, 68, 33, 40, 53, 14, 51, 72, 28, 84, 26, 74, 89, 63, 32, 55, 50, 75,
98, 76, 14, 36, 23,
 7, 31, 48, 64, 93, 96, 46, 52, 7, 34, 70, 66, 79, 49, 80, 59, 65, 82,
92, 71, 57, 40, 69,
 94, 32, 99, 78, 61, 87, 67, 90, 6, 43, 62, 88, 53, 19, 95, 6, 60, 81,
35, 77, 24, 85, 73,
97, 68, 55, 56, 11, 48, 54, 58, 63, 83, 19, 72, 100, 86, 91, 25, 11,
74, 13, 27, 47, 17,
24, 45, 75, 84, 10, 76, 38, 41, 43, 35, 13, 89, 9, 44, 65, 10, 42, 17,
37, 25, 39, 61, 9,
71, 16, 27, 98, 12, 62, 67, 93, 3, 60, 2, 96, 3, 2, 78, 56, 54, 12, 16,
18, 92, 58, 38, 47,
45, 5, 41, 94, 73, 77, 90, 5, 88, 37, 99, 44, 42, 39, 18, 85]
```

15 Skyscrapers

Fill the grid with numbers, so that every number appears only once in every row and column. The numbers used range from 1 upto the length of each row or column. Imagine the grid is the aerial view of a city block of skyscrapers of varying heights, one within each cell in the grid. Each skyscraper is to be represented as a number indicating its height.A number outside the grid describes how many skyscrapers can be seen along that row or up/down that column from the perspective of that number on the ground. You can only see a skyscraper if smaller skyscrapers are in front of it; you cannot see one if a taller skyscraper is in front of it, blocking the view. Here is a 6×6 Skyscraper puzzle

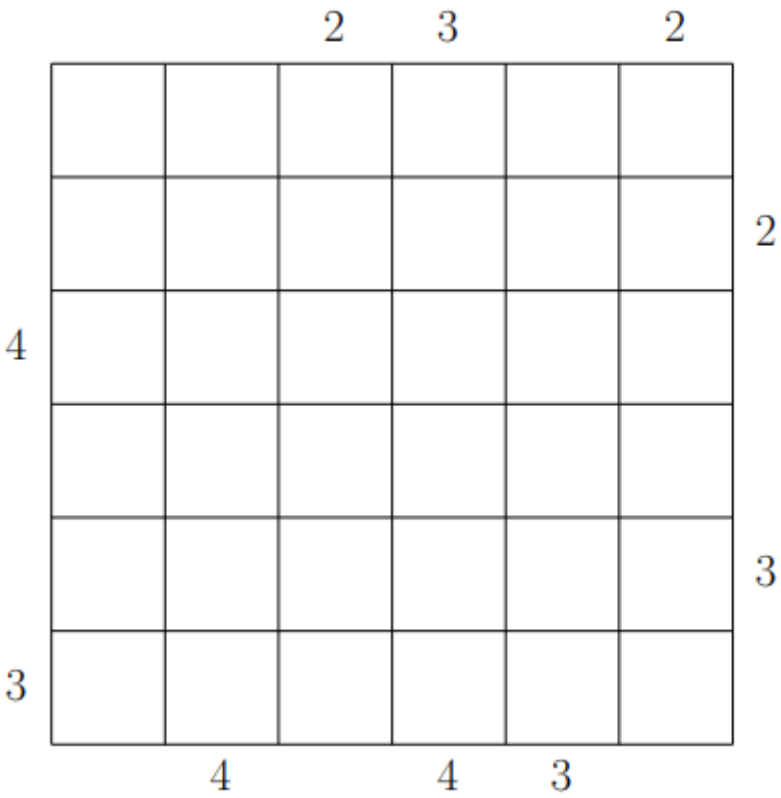


Figure 27:

15.1 Python code

```
from z3 import *
from itertools import permutations
from collections import defaultdict

class SkyscrapersSolver:
    def __init__(self, n, puzzle):
        self.n = n
        self.input = puzzle
        self.perms = defaultdict(set)
        self.X = [[Int("x_%s_%s" % (i+1, j+1)) for j in range(n)]
                    for i in range(n)]
```

```
        self.s = Solver()
        self.s.add([And(1 <= self.X[i][j], self.X[i][j]<= n) for i in
range(n) for j in range(n)])
        for i in range(n):
            self.s.add(And(Distinct(self.X[i])))
        for j in range(n):
            self.s.add(And(Distinct([self.X[i][j] for i in range(n)])))

    def classify_permutations(self):
        def num_of_visible_ss(arr, reverse=False):
            if reverse:
                arr = arr[-1::-1]
            v = 0
            for i in range(0, len(arr)):
                if arr[i] >= max(arr[0:i+1]):
                    v += 1
            return v

        for p in permutations(range(1, self.n+1)):
            self.perms[("R", num_of_visible_ss(p))].add(p)
            self.perms[("L", num_of_visible_ss(p,
reverse=True))].add(p)

    def set_constraints(self):
        gl_consts = []
        for dr, rn, ns  in self.input:
            if dr == "R" or dr == "L":
                poss_perms = []
                for p in self.perms[(dr,ns)]:
                    poss_perms.append(And([self.X[rn-1][i] == v for i,v
in enumerate(p)]))
                gl_consts.append(Or(poss_perms))
            else:
                mdr = "R" if dr == "D" else "L"
                poss_perms = []
                for p in self.perms[(mdr, ns)]:
                    poss_perms.append(And([self.X[i][rn-1] == v for i,v
in enumerate(p)]))
                gl_consts.append(Or(poss_perms))
        self.s.add(And(gl_consts))

    def output_solution(self):
        m = self.s.model()
        for i in range(self.n):
            print("  ".join(['{:>2}'.format(str(m.evaluate(self.X[i]
[j])))
                    for j in range(self.n)]))

    def solve(self):
        self.classify_permutations()
        self.set_constraints()
        if self.s.check() == sat:
            self.output_solution()
```

```
        else:
            print(self.s)
            print("Failed to solve.")

puzzle = ([("R",3,4),("R",6,3),("D",3,2),("D",4,3),("D",6,2),("L",2,2),
("L",5,3),
            ("U",2,4),("U",4,4),("U",5,3)])
sss = SkyscrapersSolver(6, puzzle)
sss.solve()
```

15.2 Solution

Here is the solution to the puzzle above

```
2   6   4   1   3   5

4   3   2   5   6   1

1   2   3   6   5   4

6   5   1   4   2   3

5   4   6   3   1   2

3   1   5   2   4   6
```

16 Numbrix

Just fill in the puzzle so the consecutive numbers follow a horizontal or vertical path (no diagonals).Here is a hard numbrix puzzle

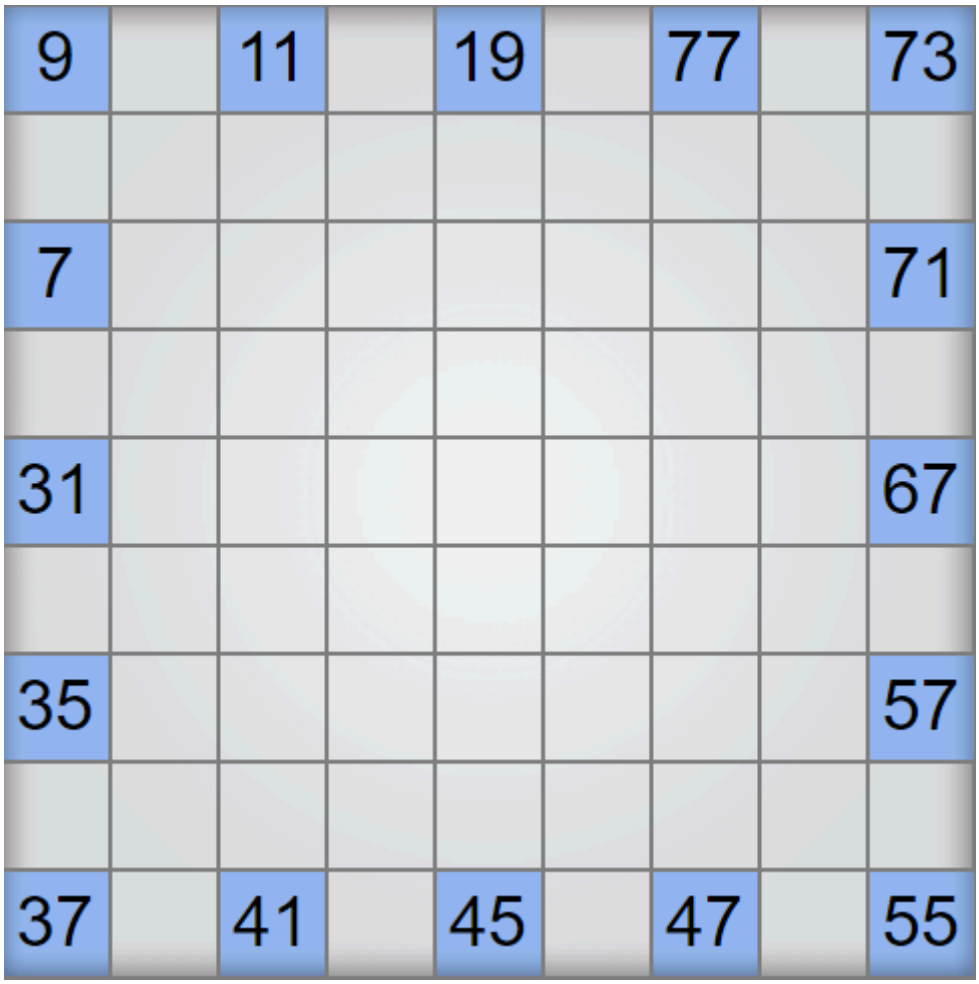


Figure 28:

16.1 Python code

```
from z3 import *
import re
from itertools import combinations

def Abs(x):
    return If(x >=0, x, -x)

class NumbricksSolver:
    def __init__(self, n):
        self.n = n
        self.X = [[Int("x_%s_%s" % (i+1, j+1)) for j in range(n)]
                  for i in range(n)]
        self.s = Solver()
        self.s.add([And(1 <= self.X[i][j], self.X[i][j]<= n*n) for i in
range(n) for j in range(n)])
        self.s.add([And(Distinct([self.X[i][j] for i in range(n) for j
```

```
in range(n)])])])
        for i in range(n):
            for j in range(n):
                ns = []
                if (i - 1) >= 0:
                    ns.append(self.X[i-1][j])
                if (i + 1) < n:
                    ns.append(self.X[i+1][j])
                if (j - 1) >= 0:
                    ns.append(self.X[i][j-1])
                if (j + 1) < n:
                    ns.append(self.X[i][j+1])
                c_n1_ne = Or(*[And(*[Abs(self.X[i][j]-nb)==1 for nb in
nbs])
                               for nbs in combinations(ns, 2)])
                c_1_or_e = Or(*[Abs(self.X[i][j]-nb)==1 for nb in ns])
                self.s.add(If(self.X[i][j] == 1, c_1_or_e, If(self.X[i]
[j] == n*n, c_1_or_e, c_n1_ne)))

    def load_puzzle(self, puzzle):
        for i, line in enumerate(re.split("\n", puzzle)):
            for j, v in enumerate(re.split(",", line)):
                if v != "_":
                    self.s.add(And(self.X[i][j] == int(v)))

    def output_solution(self):
        m = self.s.model()
        for i in range(self.n):
            print("  ".join(['{:>2}'.format(str(m.evaluate(self.X[i]
[j])))
                for j in range(self.n)]))
            print("\n")

    def solve(self, puzzle):
        self.load_puzzle(puzzle)
        if self.s.check() == sat:
            self.output_solution()
        else:
            print(self.s)
            print("Failed to solve.")

puzzle = '''9,_,11,_,19,_,77,_,73
_,_,_,_,_,_,_,_,_
7,_,_,_,_,_,_,_,71
_,_,_,_,_,_,_,_,_
31,_,_,_,_,_,_,_,67
_,_,_,_,_,_,_,_,_
35,_,_,_,_,_,_,_,57
_,_,_,_,_,_,_,_,_
37,_,41,_,45,_,47,_,55'''

ns = NumbricksSolver(9)
ns.solve(puzzle)
```

16.2 Solution

Here is the solution to the above puzzle

```
9  10  11  18  19  78  77  74  73

 8  13  12  17  20  79  76  75  72

 7  14  15  16  21  80  81  70  71

 6   5   4   3  22  63  64  69  68

31  30  29   2  23  62  65  66  67

32  33  28   1  24  61  60  59  58

35  34  27  26  25  50  51  52  57

36  39  40  43  44  49  48  53  56

37  38  41  42  45  46  47  54  55
```

17 Kakuro

Kakuro is like a crossword puzzle with numbers. Each "word" must add up to the number provided in the clue above it or to the left. Words can only use the numbers 1 through 9, and a given number can only be used once in a word. Here is a hard Kakuro puzzle

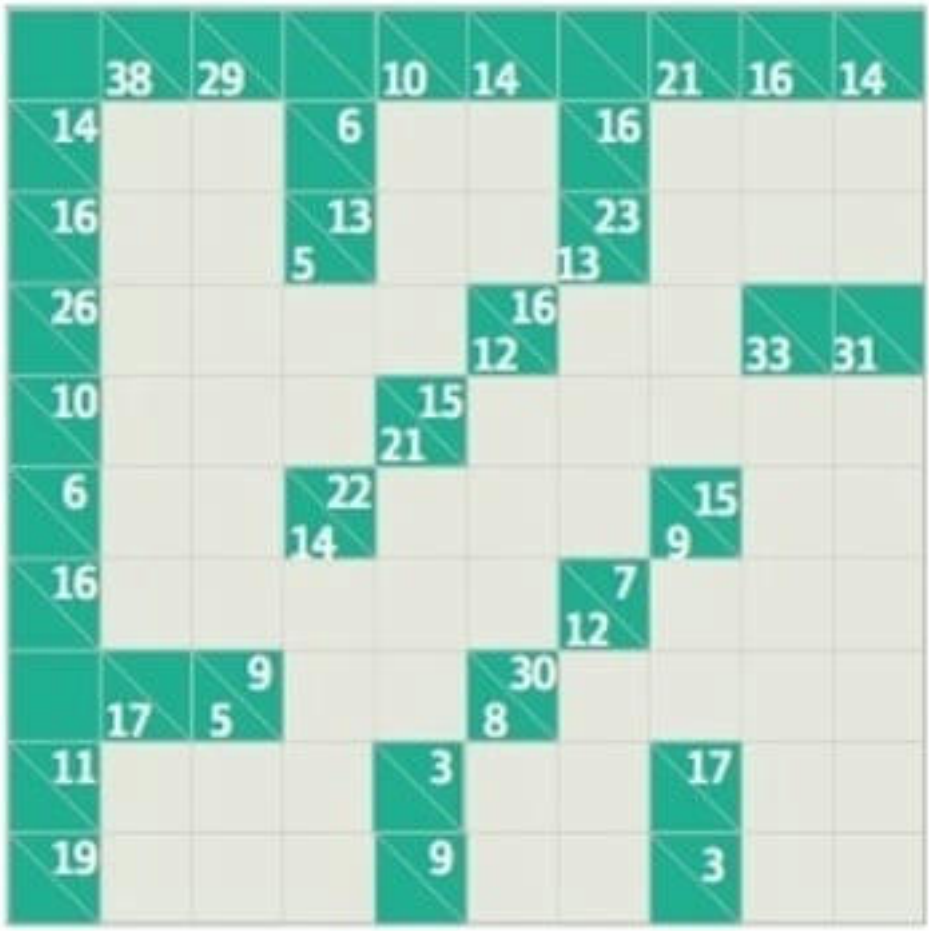


Figure 29:

17.1 Python code

Here is how the above puzzle is encoded:

```
x,38|,29|,x,10|,14|,x,21|,16|,14|
|14,_,_,|6,_,_,|16,_,_,_
|16,_,_,5|13,_,_,13|23,_,_,_
|26,_,_,_,_,12|16,_,_,33|,31|
|10,_,_,_,21|15,_,_,_,_,_
|6,_,_,14|22,_,_,_,9|15,_,_
|16,_,_,_,_,_,12|7,_,_,_
x,17|,5|9,_,_,8|30,_,_,_,_
|11,_,_,_,|3,_,_,|17,_,_
|19,_,_,_,|9,_,_,|3,_,_
```

Here is the python code using Z3

```
import sys
from z3 import *

class KakuroSolver:
    def __init__(self, fp):
        self.inp, self.vars = self.load_puzzle(fp)
        self.rows = len(self.inp)
        self.cols = len(self.inp[0])
        self.X_map = {(i, j): Int("x_%s_%s" % (i+1, j+1)) for (i, j) in
self.vars}
        self.s = Solver()
        self.s.add([And(1 <= v, v <= 9) for v in self.X_map.values()])

    def load_puzzle(self, fp):
        X = []
        with(open(fp, "r")) as f:
            for line in f.readlines():
                X.append(line.strip("\n").split(","))
        var_pos = [(i, j) for (i, l) in enumerate(X) for (j, t) in
enumerate(l) if t == "_"]
        return X, var_pos

    def set_constraints(self):
        for i in range(self.rows):
            for j in range(self.cols):
                if "|" in self.inp[i][j]:
                    c, r = self.inp[i][j].split("|")
                    if r:
                        bvars, r_p = [], j+1
                        while (r_p < self.cols  and self.inp[i][r_p] ==
"_"):
                            bvars.append(self.X_map[(i, r_p)])
                            r_p += 1
                        self.s.add(And(Distinct(bvars)))
                        self.s.add(And(sum(bvars) == int(r)))
                    if c:
                        bvars, c_p = [], i+1
                        while (c_p <self.rows and self.inp[c_p][j] ==
"_"):
                            bvars.append(self.X_map[(c_p, j)])
                            c_p += 1
                        self.s.add(And(Distinct(bvars)))
                        self.s.add(And(sum(bvars) == int(c)))

    def print_grid(self):
        print("Here is the solution")
        m = self.s.model()
        for i in range(self.rows):
            print("  ".join(['{:>3}'.format(str(m.eval(self.X_map[(i,
j)]))) if self.inp[i][j] == "_" else '{:>3}'.format(" ")
                            for j in range(self.cols)]))

    def solve(self):
```

```
        self.set_constraints()
        if self.s.check() == sat:
            self.print_grid()
        else:
            print("Failed to solve the puzzle")

if __name__ == "__main__":
    ks = KakuroSolver(sys.argv[1])
    ks.solve()
```

18 Kakurasu

Kakurasu is played on a rectangular grid with no standard size. The goal is to make some of the cells black in such a way that:

(i) The black cells on each row sum up to the number on the right.

(ii) The black cells on each column sum up to the number on the bottom.

(iii) If a black cell is first on its row/column its value is 1. If it is second its value is 2 etc.

Here is a 9×9 hard Kakurasu puzzle

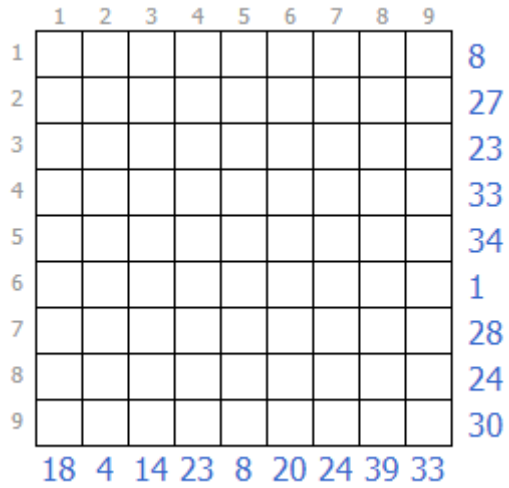


Figure 30:

18.1 Python code

```
from z3 import *
import seaborn as sns
sns.set()

class KakurasuSolver:
    def __init__(self, n, puzzle, outputfilename):
        self.n = n
        self.input = puzzle
        self.output = outputfilename
        self.X = [[Int("x_%s_%s" % (i+1, j+1)) for j in range(n)]
                  for i in range(n)]
        self.s = Solver()
        self.s.add([And(0 <= self.X[i][j], self.X[i][j]<= 1) for i in
range(n) for j in range(n)])

    def set_constraints(self):
        for d, n, s in self.input:
            if d == "C":
                self.s.add(And(sum([(i+1)*self.X[i][n-1] for i in
range(0,self.n)]) == s))
            if d == "R":
                self.s.add(And(sum([(i+1)*self.X[n-1][i] for i in
```

```
range(0,self.n)]) == s))

    def output_solution(self):
        m = self.s.model()
        data = [[int(str(m.evaluate(self.X[i][j])))*-1 for j in
range(self.n)] for i in range(self.n)]
        snsplot = sns.heatmap(data, square=True, linewidths=1.0,
xticklabels=False, yticklabels=False, cbar=False)
        snsplot.get_figure().savefig(self.outputfilename + ".png")

    def solve(self):
        self.set_constraints()
        if self.s.check() == sat:
            self.output_solution()
        else:
            print(self.s)
            print("Failed to solve.")

puzzle = [
    ("C",1,18),
    ("C",2,4),
    ("C",3,14),
    ("C",4,23),
    ("C",5,8),
    ("C",6,20),
    ("C",7,24),
    ("C",8,39),
    ("C",9,33),
    ("R",1,8),
    ("R",2,27),
    ("R",3,23),
    ("R",4,33),
    ("R",5,34),
    ("R",6,1),
    ("R",7,28),
    ("R",8,24),
    ("R",9,30),
]

outputfilename = "kakurasu_sol"
ks = KakurasuSolver(9, puzzle, outputfilename)
ks.solve()
```

18.2 Solution

The solution for above puzzle is given below

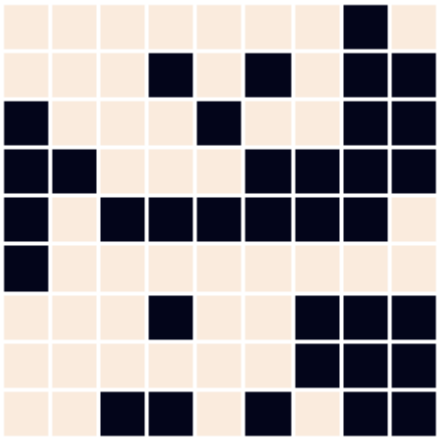


Figure 31:

19 3-In-A-Row puzzle

Can you fill the grid with Blue and White squares without creating a 3-In-A-Row of the same colour? Each row and column has an equal number of Blue and White squares. Blue is represented by "X" and white is represented by "O" in the picture below

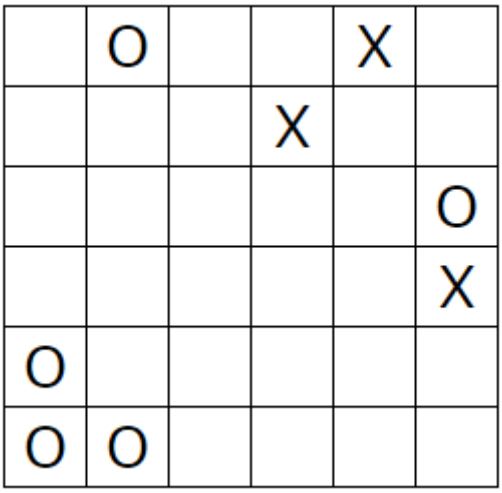


Figure 32:

19.1 Python code

```
from z3 import *

class ThreeInARowSolver:
    def __init__(self, n, puzzle):
        self.n = n
        self.input = puzzle
        self.X = [[Int("x_%s_%s" % (i, j)) for j in range(n)]
                      for i in range(n)]
        self.s = Solver()
        self.s.add([And(0 <= self.X[i][j], self.X[i][j]<= 1) for i in
range(n) for j in range(n)])

    def set_constraints(self):
        for i,j,v in self.input:
            self.s.add(And(self.X[i][j] == v))
        for i in range(self.n):
            self.s.add(And(sum(self.X[i]) == self.n/2))
        for j in range(self.n):
            self.s.add(And(sum([self.X[i][j] for i in range(self.n)])
== self.n/2))
        for i in range(0, self.n):
            for j in range(0, self.n-2):
                self.s.add(And(self.X[i][j] + self.X[i][j+1] +
self.X[i][j+2] != 0))
                self.s.add(And(self.X[i][j] + self.X[i][j+1] +
self.X[i][j+2] != 3))
```

```
        for j in range(0, self.n):
            for i in range(0, self.n-2):
                self.s.add(And(self.X[i][j] + self.X[i+1][j] +
self.X[i+2][j] != 0))
                self.s.add(And(self.X[i][j] + self.X[i+1][j] +
self.X[i+2][j] != 3))

    def output_solution(self):
        m = self.s.model()
        for i in range(self.n):
            print(" ".join(["X" if m.evaluate(self.X[i][j])==1 else "O"
for j in range(self.n)]))

    def solve(self):
        self.set_constraints()
        if self.s.check() == sat:
            self.output_solution()
        else:
            print(self.s)
            print("Failed to solve.")

""" The puzzle is encoded using a 1 for an X and an O for an O. The
positions of the X's and O's are captured as a list of 3-tuples (row,
column, symbol)."""

puzzle = [
    (0,4,1),
    (1,3,1),
    (3,5,1),
    (0,1,0),
    (2,5,0),
    (4,0,0),
    (5,0,0),
    (5,1,0)
]

ts = ThreeInARowSolver(6, puzzle)
ts.solve()
```

19.2 Solution

```
X O X O X O
O X O X O X
X X O O X O
X O X O O X
O X O X X O
O O X X O X
```

20 Fish

20.1 The Situation

(i) There are 5 houses in five different colours.

(ii) In each house lives a person with a different nationality.

(iii) These five owners drink a certain type of beverage, smoke a certain brand of cigar and keep a certain pet.

(iv) No owners have the same pet, smoke the same brand of cigar or drink the same beverage.

The question is who owns the fish?

20.2 Hints

(i) The Brit lives in the red house.

(ii) The swede keeps dogs as pets.

(iii) The dane drinks tea.

(iv) The green house is on the left of the white house.

(v) The green house owner drinks coffee.

(vi) The person who smokes pallmall rears birds.

(vii) The owner of the yellow house smokes dunhill.

(viii) The man living in the centre house drinks milk.

(ix) The Norwegian lives in the first house.

(x) The man who smokes blends lives next to the one who keeps cats.

(xi) The man who keeps horses lives next to the man who smokes dunhill.

(xii) The owner who smokes bluemaster drinks beer.

(xiii) The German smokes prince.

(xiv) The Norwegian lives next to the blue house.

(xv) The man who smokes blends has a neighbour who drinks water.

20.3 Solution

The German owns the **fish**. Here is a possible assignment satisfying all the constraints:

House	Nationality	Colour	Pets	Cigars	Beverages
1	Norweigan	Green	Bird	Pallmall	Coffee
2	German	Blue	Fish	Prince	Water
3	Brit	Red	Horse	Blends	Milk
4	Dane	Yellow	Cat	Dunhill	Tea

5	Swede	White	Dog	Bluemaster	Beer

20.4 Python code

```
from z3 import *

house, nat, col, pet, cig, bev = [0,1,2,3,4,5]
houses = {0:"House1", 1:"House2", 2:"House3", 3:"House4", 4:"House5"}

red, blue, green, yellow, white = [0,1,2,3,4]
colours = {red:"Red", blue: "Blue", green:"Green", yellow:"Yellow",
white:"White"}

brit, swede, dane, german, norwegian = [0,1,2,3,4]
nationality = {brit:"Brit", swede:"Swede", dane:"Dane",
german:"German", norwegian:"Norw"}

fish, cat, bird, dog, horse = [0,1,2,3,4]
pets = {fish:"Fish", cat:"Cat", bird:"Bird", dog:"Dog", horse:"Horse"}

pallmall, dunhill, bluemaster, blends, prince = [0,1,2,3,4]
cigars = {pallmall:"Pallmall", dunhill:"Dunhill",
bluemaster:"Bluemaster", blends:"Blends", prince:"Prince"}

milk, tea, coffee, beer, water = [0,1,2,3,4]
bevs = {milk:"Milk", tea:"Tea", coffee:"Coffee", beer:"Beer",
water:"Water"}

columns = {house:houses, nat:nationality, col:colours, pet:pets,
cig:cigars, bev:bevs}

def Abs(x):
    return If(x >=0, x, -x)

class AssignmentPuzzleSolver:
    def __init__(self):
        self.X = [[Int("x_%s_%s" % (i, j)) for j in range(6)]
                      for i in range(5)]
        self.s = Solver()
        self.s.add([And(0 <= self.X[i][j], self.X[i][j]<= 4)
                    for i in range(5) for j in range(6)])

    def set_constraints(self):
        cons = []

        # there is no repetition along each dimension
        cols_c = [Distinct([self.X[i][j] for i in range(5)]) for j in
range(6)]
        cons.append(And(cols_c))

        # The brit lives in the red house
        cons1 = Or([And(self.X[i][nat] == brit, self.X[i][col] == red)
```

```
                        for i in range(5)])
        cons.append(cons1)

        # The swede keeps dogs as pets
        cons2 = Or([And(self.X[i][nat] == swede, self.X[i][pet] == dog)
                            for i in range(5)])
        cons.append(cons2)

        # The dane drinks tea
        cons3 = Or([And(self.X[i][nat] == dane, self.X[i][bev] == tea)
                            for i in range(5)])
        cons.append(cons3)

        # The green house is on the left of the  white house
        cons4 = []
        for i in range(4):
            for j in range(i+1,5):
                cons4.append(And(self.X[i][col] == green, self.X[j]
[col] == white))
        cons4 = Or(cons4)
        cons.append(cons4)

        # The green house owner drinks coffee
        cons5 = Or([And(self.X[i][col] == green, self.X[i][bev] ==
coffee)
                            for i in range(5)])
        cons.append(cons5)

        # The person who smokes pallmall rears birds
        cons6 = Or([And(self.X[i][cig] == pallmall, self.X[i][pet] ==
bird)
                            for i in range(5)])
        cons.append(cons6)

        # The owner of the yellow house smokes dunhill
        cons7 = Or([And(self.X[i][col] == yellow, self.X[i][cig] ==
dunhill)
                            for i in range(5)])
        cons.append(cons7)

        # The man living in the centre house drinks milk
        cons8 = Or([And(self.X[i][house] == 2, self.X[i][bev] == milk)
                            for i in range(5)])
        cons.append(cons8)

        # The Norwegian lives in the first house
        cons9 = Or([And(self.X[i][nat] == norwegian, self.X[i][house]
== 0)
                        for i in range(5)])
        cons.append(cons9)

        # The man who smokes blends lives next to the one who keeps
cats
```

```
        cons10 = []
        for i in range(5):
            for j in range(5):
                if i != j:
                    cons10.append(And(self.X[i][cig] == blends,
self.X[j][pet] == cat,
                          Abs(self.X[i][house]-self.X[j][house]) == 1))
        cons10 = Or(cons10)
        cons.append(cons10)

        # The man who keeps horses lives next to the man who smokes
dunhill
        cons11 = []
        for i in range(5):
            for j in range(5):
                if i != j:
                    cons11.append(And(self.X[i][pet] == horse,
self.X[j][cig] == dunhill,
                                              Abs(self.X[i][house]-
self.X[j][house]) == 1))
        cons11 = Or(cons11)
        cons.append(cons11)

        # The owner who smokes bluemaster drinks beer
        cons12 = Or([And(self.X[i][cig] == bluemaster,self.X[i][bev] ==
beer)
                             for i in range(5)])
        cons.append(cons12)

        # The german smokes prince
        cons13 = Or([And(self.X[i][nat] == german, self.X[i][cig] ==
prince)
                             for i in range(5)])
        cons.append(cons13)

        # The Norwegian lives next to the blue house
        cons14 = Or([And(self.X[i][house] == 1, self.X[i][col] == blue)
                             for i in range(5)])
        cons.append(cons14)

        # The man who smokes blends has a neighbour who drinks water
        cons15 = []
        for i in range(5):
            for j in range(5):
                if i != j:
                    cons15.append(And(self.X[i][cig] == blends,
self.X[j][bev] == water,
                                  Abs(self.X[i][house]-self.X[j][house])
== 1))
        cons15 = Or(cons15)
        cons.append(cons15)

        self.s.add(And(cons))
```

```
    def output_solution(self):
        m = self.s.model()

print("\t".join(["House","Nationality","Colour","Pets","Cigars","Beverages"]))
        for i in range(5):
            print("\t".join([columns[j][m.evaluate(self.X[i]
[j]).as_long()] for j in range(6)]))

    def solve(self):
        self.set_constraints()
        if self.s.check() == sat:
            self.output_solution()
        else:
            print(self.s)
            print("Failed to solve.")

s = AssignmentPuzzleSolver()
s.solve()
```

21 Flowfree

Flowfree is a puzzle game that is available as an android/ios app and online. The game is played on a square grid with n pairs of same-colored squares. The objective is to join each of the same-colored pairs by means of an unbroken chain of squares of the same color. A typical starting position together with the solution is in shown in the figure below:

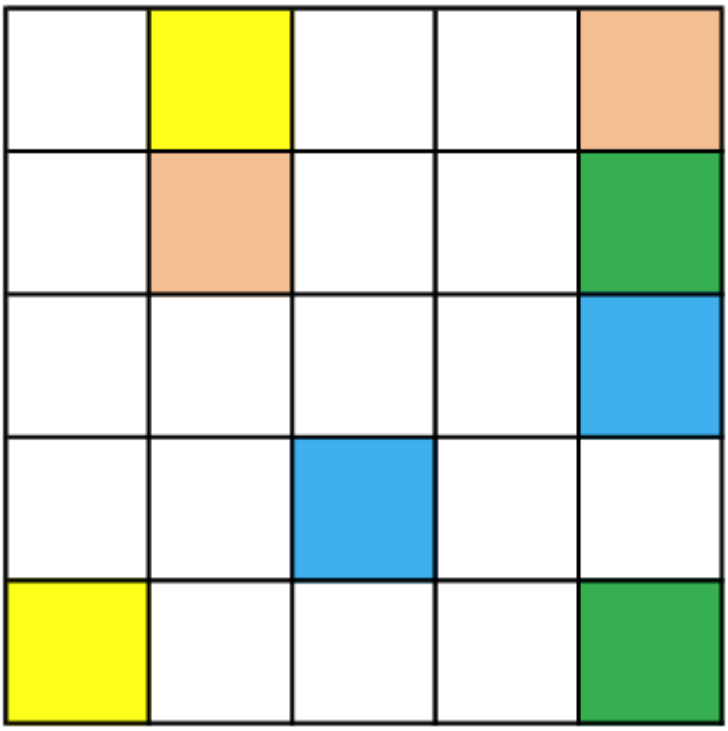


Figure 33:

21.1 Model

Define the sets $M = 1...m$ and $N = 1...n$, where m is the size of the grid and n is the number of pairs of same-colored cells. Also define variables $x_{ijk} = 1$ if cell (i, j) requires color k, otherwise 0, $\forall i \in M, j \in M, k \in N$ and parameters $S_{ijk} = 1$ if cell (i, j) has color k, otherwise 0. The conditions required by the puzzle are enforced as follows.

(i) Ensure the solution is consistent with the starting configuration.

$$x_{ijk} \geq S_{ijk}, \forall i \in M, j \in M, k \in N \tag{21.1}$$

(ii) Each cell contains a single color.

$$\sum_{k \in N} x_{ijk} = 1, \forall i \in M, j \in M \tag{21.2}$$

(iii) Ensure a continuous chain for each color. This is done by ensuring

that the initially colored squares have exactly one adjacent square of the same color and all other squares will have exactly two adjacent squares of the same color. Define

$$
\begin{aligned}
f(x_{ijk}) &= \sum_{p=i-1,p\neq i,p\in M}^{i+1} x_{pjk} + \sum_{q=j-1,q\neq j,q\in M}^{j+1} x_{iqk} + S_{ijk} \\
f(x_{ijk}) &\geq 2x_{ijk} - 5(1-x_{ijk}), \forall i \in M, j \in M, k \in N \\
f(x_{ijk}) &\leq 2x_{ijk} + 5(1-x_{ijk}), \forall i \in M, j \in M, k \in N
\end{aligned}
\tag{21.3}
$$

21.2 Python code

Here is the code for the above formulation and puzzle:

```
from ortools.sat.python import cp_model
from enum import IntEnum

class Color(IntEnum):
    YELLOW = 0,
    BROWN = 1,
    GREEN = 2,
    BLUE = 3

def puzzle():
    m, n = 5, 4
    s = {(i,j,k):0 for i in range(m) for j in range(m) for k in
range(n)}
    s[(0, 1, Color.YELLOW)], s[(4, 0, Color.YELLOW)] = 1, 1
    s[(0, 4, Color.BROWN)], s[(1, 1, Color.BROWN)] = 1, 1
    s[(3, 2, Color.BLUE)], s[(2, 4, Color.BLUE)] = 1, 1
    s[(1, 4, Color.GREEN)], s[(4, 4, Color.GREEN)] = 1, 1
    return m, n, s

def flowfree(puzzle):
    m, n, s = puzzle
    model = cp_model.CpModel()
    x = {(i,j,k): model.NewIntVar(0, 1, 'x(%i,%i,%i)' % (i,j,k))
            for i in range(m) for j in range(m) for k in range(n)}

    for i in range(m):
        for j in range(m):
            for k in range(n):
                model.Add(x[(i,j,k)] >= s[(i,j,k)])

    for i in range(m):
        for j in range(m):
            model.Add(sum(x[(i,j,k)] for k in range(n))==1)

    M = list(range(m))
    for i in range(m):
        for j in range(m):
            for k in range(n):
                f = s[(i,j,k)] + \
                    sum(x[(p,j,k)] for p in range(i-1, i+2) if p != i
and p in M) + \
                    sum(x[(i,q,k)] for q in range(j-1, j+2) if q != j
```

```
and q in M)
                model.Add(f >= 2*x[(i,j,k)]-5*(1-x[(i,j,k)]))
                model.Add(f <= 2*x[(i,j,k)]+5*(1-x[(i,j,k)]))

    solver = cp_model.CpSolver()
    status = solver.Solve(model)
    if status == cp_model.OPTIMAL or status == cp_model.FEASIBLE:
        for i in range(m):
            print(" ".join(str(Color(k)).ljust(15) for j in range(m)
for k in range(n)
                            if solver.Value(x[(i,j,k)]) != 0))
    else:
        print("Couldn't solve.")

flowfree(puzzle())
```

21.3 Solution

The output of the above program is as follows:

```
Color.YELLOW    Color.YELLOW    Color.BROWN     Color.BROWN
Color.BROWN
Color.YELLOW    Color.BROWN     Color.BROWN     Color.GREEN
Color.GREEN
Color.YELLOW    Color.GREEN     Color.GREEN     Color.GREEN
Color.BLUE
Color.YELLOW    Color.GREEN     Color.BLUE      Color.BLUE
Color.BLUE
Color.YELLOW    Color.GREEN     Color.GREEN     Color.GREEN
Color.GREEN
```

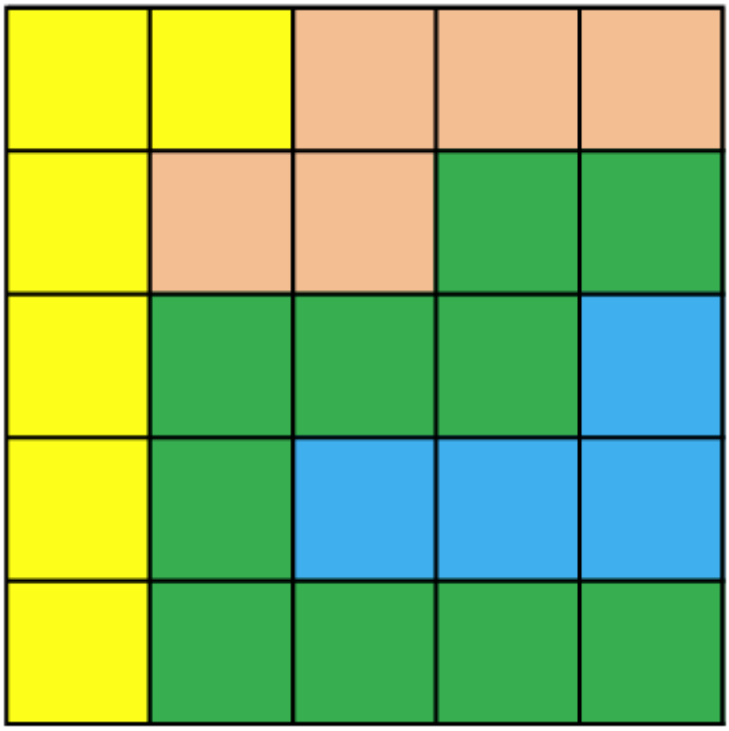


Figure 34:

22 Ostomachion

The famous four-color theorem states, essentially, that you can color in the regions of any map using at most four colors in such a way that no neighboring regions share a color. A computer-based proof of the theorem was offered in 1976.

Some 2,200 years earlier, the legendary Greek mathematician Archimedes described something called an Ostomachion. It's a group of pieces, similar to tangrams, that divides a 12-by-12 square into 14 regions. The object is to rearrange the pieces into interesting shapes, such as a Tyrannosaurus rex. It's often called the oldest known mathematical puzzle.

Your challenge today: Color in the regions of the Ostomachion square with four colors such that each color shades an equal area. (That is, each color needs to shade 36 square units.)

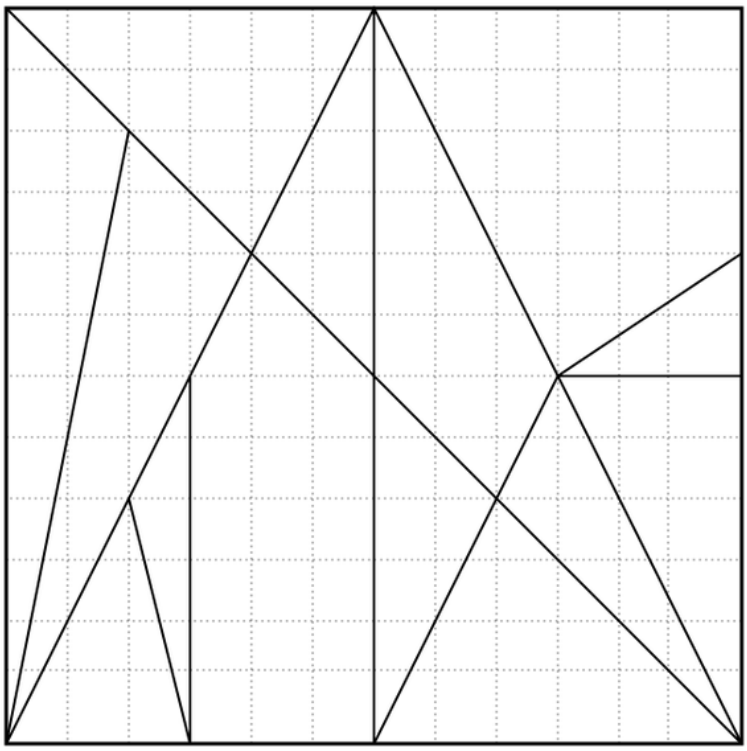


Figure 35:

22.1 Python code

Let the areas be labelled as follows:

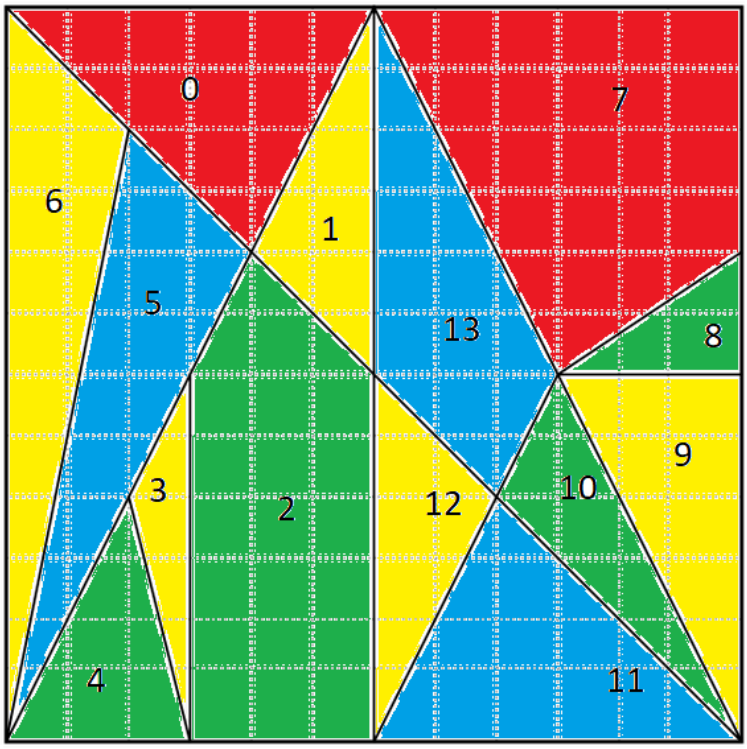


Figure 36:

```
from z3 import *
from math import factorial

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

col_map = {0:"Red", 1:"Blue", 2:"Green", 3:"Yellow", 4:"Orange",
5:"Pink"}

num_colours = 4
total_area = 144
def OstomachionSolver():
    X = [Int("x_%s" % i) for i in range(14)]
    s = Solver()
    s.add([And(0 <= X[i], X[i]<= num_colours-1) for i in range(14)])
    for c in range(num_colours):
        s.add(sum([If(X[i] == c, areas[i], 0) for i in range(14)])
                == (total_area//num_colours))
    for i in range(14):
        for j in connections[i]:
            s.add(X[i] != X[j])
```

```
    cnt_sol = 0
    while s.check() == sat:
        cnt_sol += 1
        m = s.model()
        s.add(Or([X[i] != m.eval(X[i]) for i in range(14)]))
    print("Unique solutions :", cnt_sol / factorial(num_colours))
    for i in range(14):
        print("Area %d - %s " % (i ,
col_map[int(str(m.evaluate(X[i])))]))

OstomachionSolver()
```

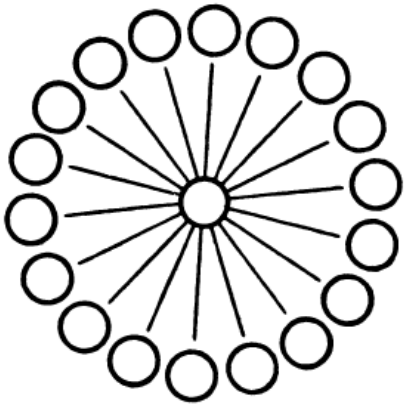
22.2 Solution

A colouring which satisfies the constraints is as follows:

```
Area 0 - Red

Area 1 - Yellow

Area 2 - Green

Area 3 - Yellow

Area 4 - Green

Area 5 - Blue

Area 6 - Yellow

Area 7 - Red

Area 8 - Green

Area 9 - Yellow

Area 10 - Green

Area 11 - Blue

Area 12 - Yellow

Area 13 - Blue
```

23 Numbers in circles

Write numbers from 1 through 19 in the circles so that the numbers in every 3 circles on a straight line total 30.



23.1 Computational Solution

We use a constraint programming to solve this puzzle. Using the code below, we see that the number in the central circle is 10 and the other pairs are $(18, 2), (17, 3), (16, 4)$,**(15,5)**, $(14, 6)$,$(13, 7), (12, 8), (19, 1), (11, 9)$.

```
from ortools.sat.python import cp_model

def solve():
    model = cp_model.CpModel()
    x = [model.NewIntVar(1, 19, 'x_%i' % i) for i in range(19)]

    model.AddAllDifferent(x)
    for i in range(1, 10):
        model.Add(x[0] + x[i] + x[i + 9] == 30)

    solver = cp_model.CpSolver()
    status = solver.Solve(model)
    if status == cp_model.FEASIBLE or status == cp_model.OPTIMAL:
        print(solver.Value(x[0]))
        for i in range(1,10):
            print(solver.Value(x[i]), solver.Value(x[i+9]))
    else:
        print('No solution found!')

solve()
```

24 Sweets in a box

Sixteen chocolates sit in a four-by-four box. What is the number of different ways you can choose six chocolates to remove such that an even number is left in each row and column?

24.1 Analytical solution

The number 6 is particularly nice: if you remove one chocolate from a given row, you have to remove at least 2 (to leave it even). Thus, you can only remove

chocolates from up to 3 rows, so there is an untouched row. By symmetry, there is an untouched column. In the remaining rows and columns, we have to remove 2 each – but this is the same as saying we have to leave exactly 1. So the answer is $4 \times 4 \times 3! = 96$.

24.2 Computational solution

We implement a simple brute-force solution where we remove any 6 of the 16 chocolates and check if the required condition holds true. Using the code below, we see that the number of different ways of removing 6 chocolates in 96.

24.2.1 Python code

```
from itertools import combinations
import numpy as np

r, c, s = 4, 4, 6
lines = []
for i in range(r):
    lines.append(list(range(i*c, (i+1)*c)))
for i in range(c):
    lines.append(list(range(i, r*c, c)))

cnt = 0
for mps in combinations(range(r*c), s):
    sweets = np.ones(r*c)
    sweets[list(mps)] = 0
    if all([sum(sweets[l]) % 2 == 0 for l in lines]):
        cnt += 1
print(cnt)
```

25 Bug Byte

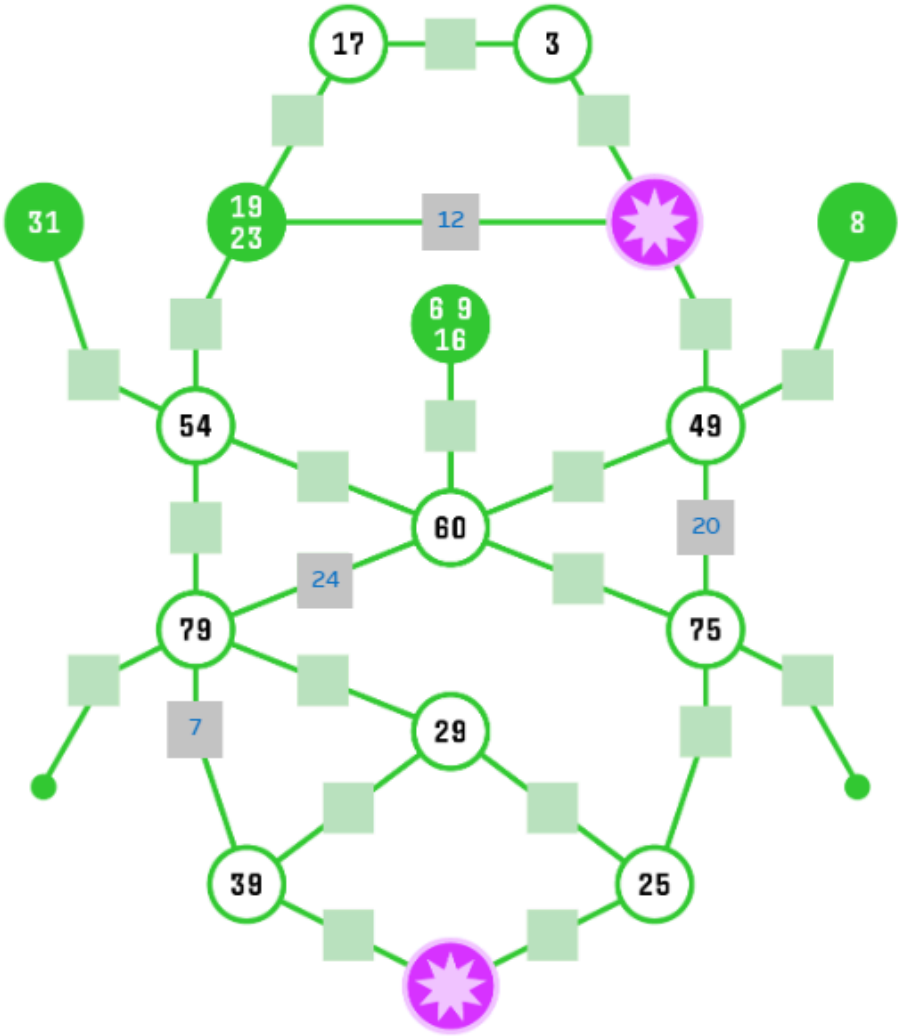
I recently came across this fantastic puzzle Bug Byte devised by the folks at **Jane Street** from **Numberphile.**

Fill in the edge weights in the graph below with the numbers 1 through 24, using each number exactly once. Labeled nodes provide some additional constraints:

M The sum of all edges directly connected to this node is M.

N There exists a non-self-intersecting path starting from this node where N is the sum of the weights of the edges on that path. Multiple numbers indicate multiple paths that may overlap.

Once the graph is filled, find the shortest (weighted) path from ✸ to ✸ and convert it to letters (1=A, 2=B, etc.) to find a secret message.



25.1 Solution

Jane Street puzzles have a reputation for being fiendishly difficult so I didn't want to tackle them by hand. I wanted to squash this recreational "bug" using heavy duty computational machinery 😀. Given that this is a puzzle involving **Graph Theory** and **Constraint Programming**, I immediately got to work using my favourite Python libraries in this space, **networkx** and the venerable **z3**.

25.1.1 First constraint

There are 24 edges in the graph and each edge has to have a distinct weight between 1 and 24. As the weights of 4 edges have already been provided, we only need 20 variables for the weights of the remaining edges. The code below shows how the above constraint can be implemented using **z3**.

```
w = [Int("w%d" % i) for i in range(20)]
s = Solver()
s.add(Distinct(w))
for i in range(20):
    s.add(And(w[i] >= 1, w[i] <= 24))
```

25.1.2 Second constraint

The sum of edges directly connected to white nodes with a green border is equal to the number inside the node. Once we assign the weight variables to each of the edges, the code below shows how to implement this straightforward constraint in **z3**.

```
s.add(w[0] + w[1] == 17)
s.add(w[0] + w[2] == 3)
s.add(w[3] + w[4] + w[6] + w[7] == 54)
s.add(24 + w[7] + w[8] + w[9] + w[13] == 60)
s.add(w[5] + w[9] + w[10] + 20 == 49)
s.add(w[13] + w[14] + w[15] + 20 == 75)
s.add(w[11] + w[6] + w[12] + 7 + 24 == 79)
s.add(w[12] + w[16] + w[17] == 29)
s.add(w[14] + w[17] + w[19] == 25)
s.add(7 + w[16] + w[18] == 39)
```

25.1.3 Third constraint

For each green node, the number inside the node represents the sum of the edge weights of a simple non intersecting path starting from that node. This is the trickiest constraint of the three but thankfully we can get **networkx** to do the heavylifting.

25.1.4 Graph Creation

We first create a weighted graph using the code below.

```
G = nx.Graph()
G.add_edge(1, 2, weight=w[0])
G.add_edge(1, 4, weight=w[1])
```

```
G.add_edge(2, 5, weight=w[2])
G.add_edge(4, 5, weight=12)
G.add_edge(4, 7, weight=w[4])
G.add_edge(3, 7, weight=w[3])
G.add_edge(5, 9, weight=w[5])
G.add_edge(6, 9, weight=w[10])
G.add_edge(7, 11, weight=w[7])
G.add_edge(8, 11, weight=w[8])
G.add_edge(9, 11, weight=w[9])
G.add_edge(7, 10, weight=w[6])
G.add_edge(10, 18, weight=w[11])
G.add_edge(10, 14, weight=7)
G.add_edge(10, 13, weight=w[12])
G.add_edge(11, 10, weight=24)
G.add_edge(11, 12, weight=w[13])
G.add_edge(9, 12, weight=20)
G.add_edge(13, 14, weight=w[16])
G.add_edge(12, 16, weight=w[14])
G.add_edge(12, 17, weight=w[15])
G.add_edge(13, 16, weight=w[17])
G.add_edge(14, 15, weight=w[18])
G.add_edge(16, 15, weight=w[19])
```

25.1.5 Implementing path constraints

From each green node, we find all the simple paths to all other nodes using the **all_simple_paths** function from **networkx** and calculate the weight of each path in terms of the weight variables. The thing here to note is that we have to use an **"Or"** constraint as the number inside each green node has to match the total path weight for one of the paths. The other insight is that the above logic doesn't change whether there there is one number or multiple numbers in each green node. These insights lead to the simple and elegant code below.

```
for start_node, total in [(3, 31), (6, 8), (4, 19), (4, 23), (8, 6),
(8, 9), (8, 16)]:
    constraints = []
    for node in set(range(1, 19)) - set([start_node]):
        for path in nx.all_simple_paths(G, start_node, node):
            constraints.append(nx.path_weight(G, path, weight="weight")
== total)
    s.add(Or(constraints))
```

25.1.6 Checking the model for satisfiability

The last part involves checking the model for satisfiability, finding the shortest path between the two nodes containing the stars, mapping the edge weights in the shortest path to alphabets. The code to do that is given below.

```
if s.check() == sat:
    m = s.model()
    for u, v in G.edges():
        if not (isinstance(G[u][v]["weight"], int)):
            G[u][v]["weight"] = m.evaluate(G[u][v]["weight"]).as_long()
```

```
sp = list(nx.shortest_path(G, 5, 15, weight="weight"))
for u, v in zip(sp, sp[1:]):
    print(chr(ord("@") + G[u][v]["weight"]))
```

25.2 Python code

Putting all the above code together, you will see that the answer is **LINKED**.

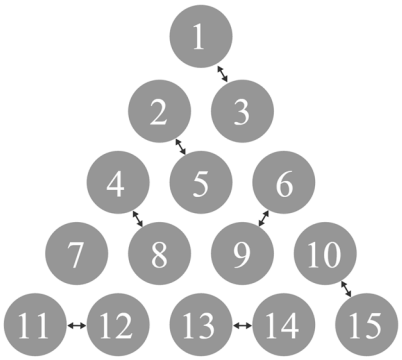
```
from z3 import Int, Distinct, And, Or, Solver
import networkx as nx


w = [Int("w%d" % i) for i in range(20)]
s = Solver()
s.add(Distinct(w))
for i in range(20):
    s.add(And(w[i] >= 1, w[i] <= 24))

s.add(w[0] + w[1] == 17)
s.add(w[0] + w[2] == 3)
s.add(w[3] + w[4] + w[6] + w[7] == 54)
s.add(24 + w[7] + w[8] + w[9] + w[13] == 60)
s.add(w[5] + w[9] + w[10] + 20 == 49)
s.add(w[13] + w[14] + w[15] + 20 == 75)
s.add(w[11] + w[6] + w[12] + 7 + 24 == 79)
s.add(w[12] + w[16] + w[17] == 29)
s.add(w[14] + w[17] + w[19] == 25)
s.add(7 + w[16] + w[18] == 39)

G = nx.Graph()
G.add_edge(1, 2, weight=w[0])
G.add_edge(1, 4, weight=w[1])
G.add_edge(2, 5, weight=w[2])
G.add_edge(4, 5, weight=12)
G.add_edge(4, 7, weight=w[4])
G.add_edge(3, 7, weight=w[3])
G.add_edge(5, 9, weight=w[5])
G.add_edge(6, 9, weight=w[10])
G.add_edge(7, 11, weight=w[7])
G.add_edge(8, 11, weight=w[8])
G.add_edge(9, 11, weight=w[9])
G.add_edge(7, 10, weight=w[6])
G.add_edge(10, 18, weight=w[11])
G.add_edge(10, 14, weight=7)
G.add_edge(10, 13, weight=w[12])
G.add_edge(11, 10, weight=24)
G.add_edge(11, 12, weight=w[13])
G.add_edge(9, 12, weight=20)
G.add_edge(13, 14, weight=w[16])
G.add_edge(12, 16, weight=w[14])
G.add_edge(12, 17, weight=w[15])
G.add_edge(13, 16, weight=w[17])
```

```
G.add_edge(14, 15, weight=w[18])
G.add_edge(16, 15, weight=w[19])


for start_node, total in [(3, 31), (6, 8), (4, 19), (4, 23), (8, 6),
(8, 9), (8, 16)]:
    constraints = []
    for node in set(range(1, 19)) - set([start_node]):
        for path in nx.all_simple_paths(G, start_node, node):
            constraints.append(nx.path_weight(G, path, weight="weight")
== total)
    s.add(Or(constraints))

if s.check() == sat:
    m = s.model()
    for u, v in G.edges():
        if not (isinstance(G[u][v]["weight"], int)):
            G[u][v]["weight"] = m.evaluate(G[u][v]["weight"]).as_long()

sp = list(nx.shortest_path(G, 5, 15, weight="weight"))
for u, v in zip(sp, sp[1:]):
    print(chr(ord("@") + G[u][v]["weight"]))
```

26 Dancer Pairs

15 dancers are standing in an equilateral triangle formation, with every dancer standing 1 unit apart from her nearest neighbors. Each dancer chooses another who is 1 unit away to pair up with, and all but 1 dancer ends up as a part of a pair. An example of one such arrangement is presented here.

How many different sets of 7 pairs are possible?

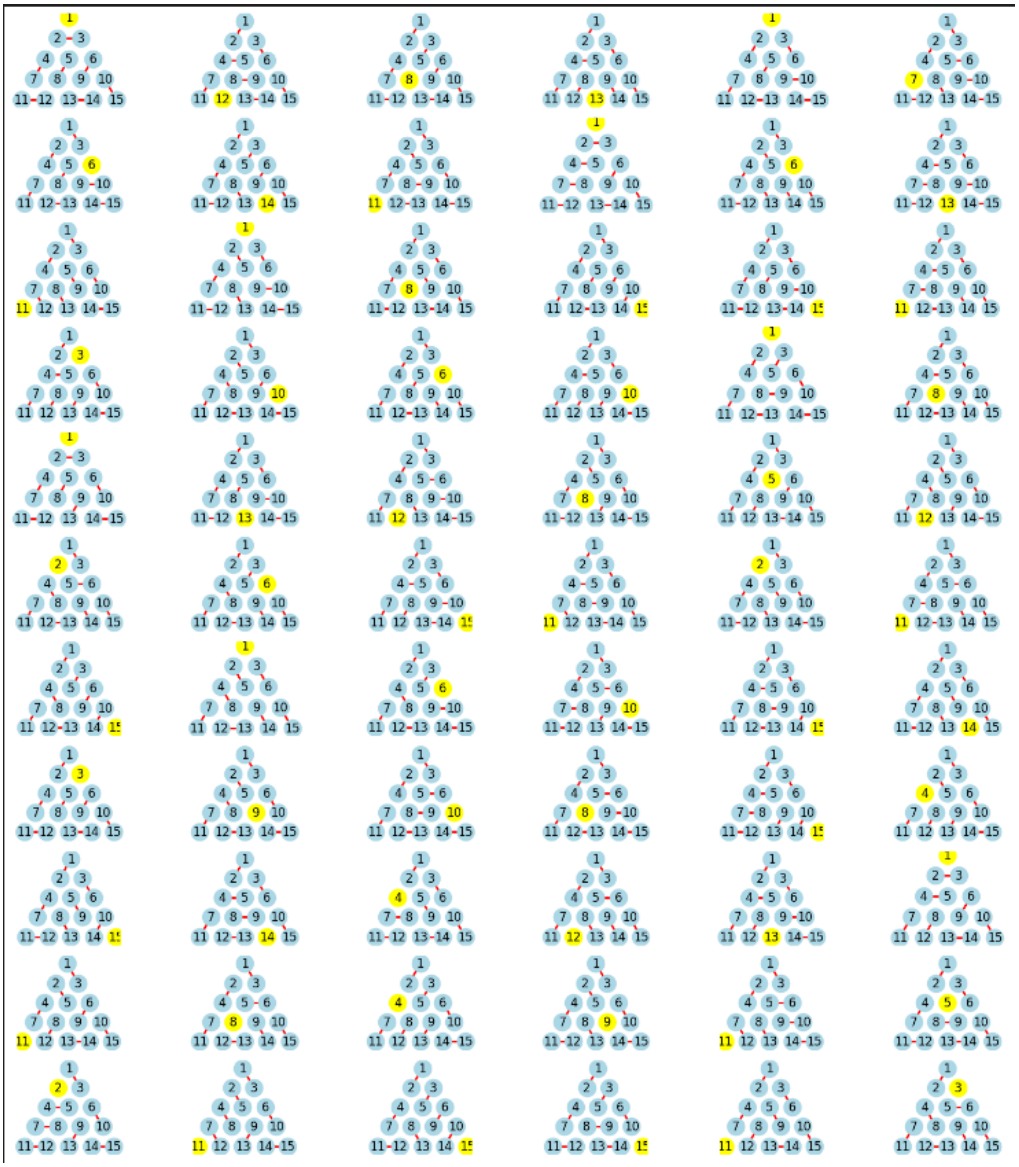


26.1 Solution

We use a backtracking algorithm in the generate_all_pairings function to generate all possible combinations. The backtrack function:

- Takes the current index, the current pairing, and the set of used nodes.
- If we have 7 pairs, we add the current pairing to our list of all pairings.
- For each node from the current index onwards:
  - If the node is already used, we skip it.
  - For each neighbor of the node:
    - If the neighbor is not used and has a higher index (to avoid duplicates), we create a new pairing with this pair.
    - We then recursively call backtrack with the updated pairing and used nodes.
- We use frozensets to represent pairs and combinations of pairs, allowing us to eliminate duplicates easily.

The code below shows us that there are 240 ways of pairing the dancers. Here are a few sample pairings:



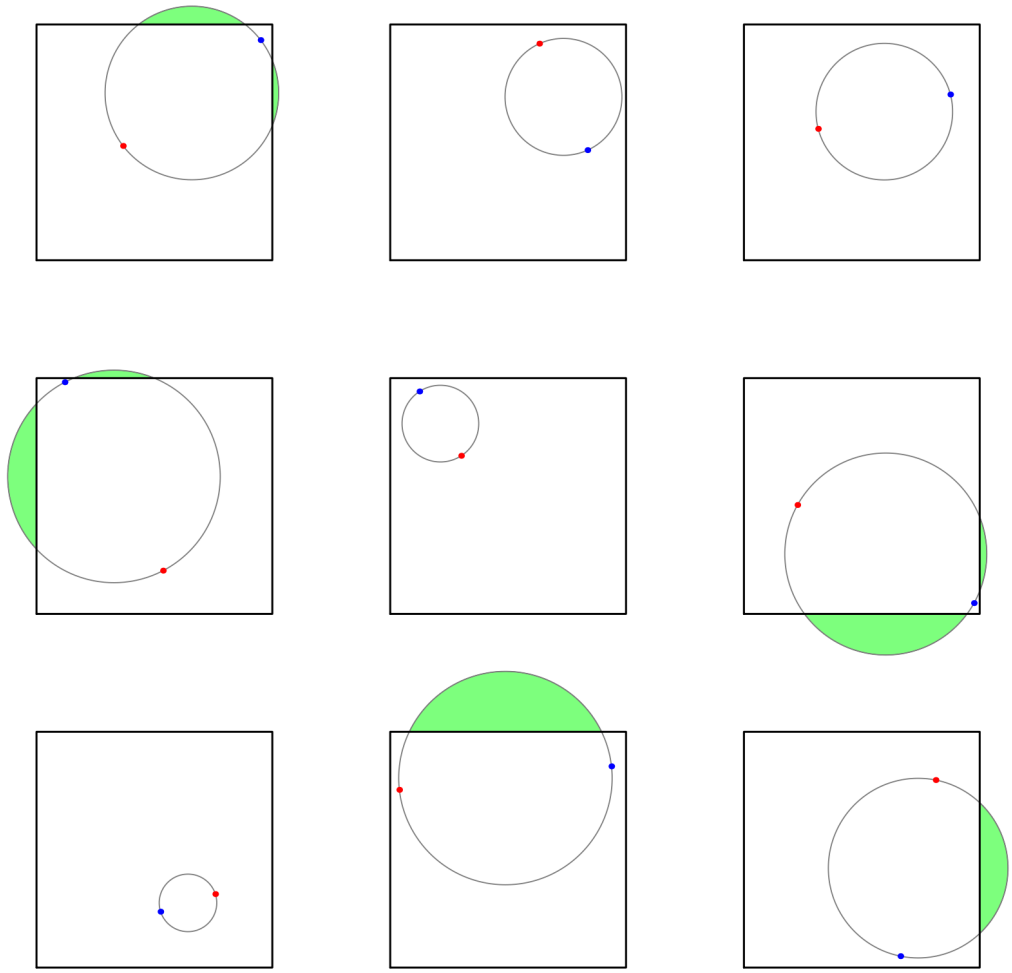
26.2 Python code

```
import networkx as nx
from itertools import combinations
import matplotlib.pyplot as plt
import math

def create_dancer_graph():
    G = nx.Graph()
    edges = [
        (1, 2), (1, 3),
        (2, 3), (2, 4), (2, 5),
        (3, 5), (3, 6),
        (4, 5), (4, 7), (4, 8),
        (5, 6), (5, 9), (5, 8),
        (6, 9), (6, 10),
        (7,8),(7, 11), (7, 12),
        (8, 9), (8, 13), (8,12),
        (9, 10), (9, 13), (9, 14),
```

```
        (10, 14), (10, 15),
        (11,12),(12,13),(13,14),(14,15)
    ]
    G.add_edges_from(edges)
    return G

def generate_all_pairings(G):
    all_pairings = []
    nodes = list(G.nodes())

    def backtrack(index, current_pairing, used_nodes):
        if len(current_pairing) == 7:
            all_pairings.append(frozenset(map(frozenset,
current_pairing)))
            return

        for i in range(index, len(nodes)):
            node = nodes[i]
            if node in used_nodes:
                continue

            for neighbor in G.neighbors(node):
                if neighbor not in used_nodes and neighbor > node:
                    new_pairing = current_pairing + [(node, neighbor)]
                    new_used_nodes = used_nodes | {node, neighbor}
                    backtrack(i + 1, new_pairing, new_used_nodes)

    backtrack(0, [], set())
    return set(all_pairings)

# Create the graph
G = create_dancer_graph()

# Generate all combinations
all_combinations = generate_all_pairings(G)

print(f"Total number of combinations: {len(all_combinations)}")
```

27 Some Off Square

A circle is randomly generated by sampling two points uniformly and independently from the interior of a square and using these points to determine its diameter. What is the probability that the circle has a part of it that is off the square? Give your answer in exact terms.



27.1 Solution

Let P_1, P_2 be the picked points and M be the midpoint of P_1P_2. Our random circle intersects the square iff the distance of M from the boundary of the square is less than the length of MP_1 or MP_2. Thus, assuming that the square is given by $[-1,1]^2$ and $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_)$, we want the probability of the event

$$\min\left(1 - \left|\frac{x_1 + x_2}{2}\right|, 1 - \left|\frac{y_1 + y_2}{2}\right|\right) \leq \frac{1}{2}\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (27.1)$$

with x_1, x_2, y_1, y_2 being independent and uniformly distributed random variables over the interval $[-1, 1]$.

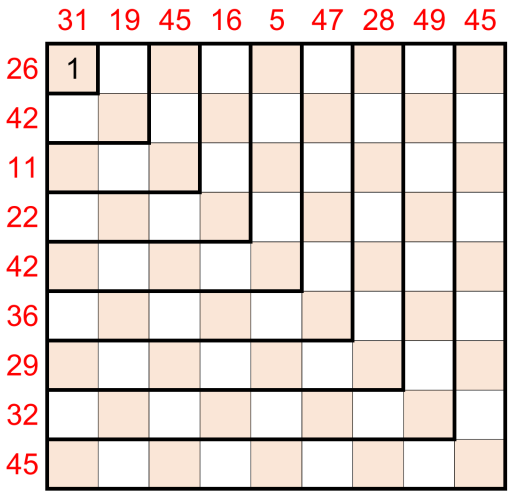
We use Monte Carlo simulation to estimate the probability. Using the Python code below, we see that the required probability is 0.476.

27.2 Python code

```
from random import uniform
from math import sqrt

runs = 10000000
cnt = 0
for _ in range(runs):
    x_1, x_2, y_1, y_2 = uniform(-1,1), uniform(-1,1), uniform(-1,1),
uniform(-1,1)
    if min(2-abs(x_1+x_2), 2-abs(y_1+y_2))<= sqrt((x_1-x_2)**2+(y_1-
y_2)**2):
        cnt += 1
print(cnt/runs)
```

28 Number hooks

In the grid below, enter nine 9's in the outermost hook, eight 8's in the next hook, then seven 7's, six 6's, and so on, down to the one 1 (already entered), so that the row and column sums match the values given along the border.



28.1 Solution

The code below provides the following solution to the above puzzle:

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

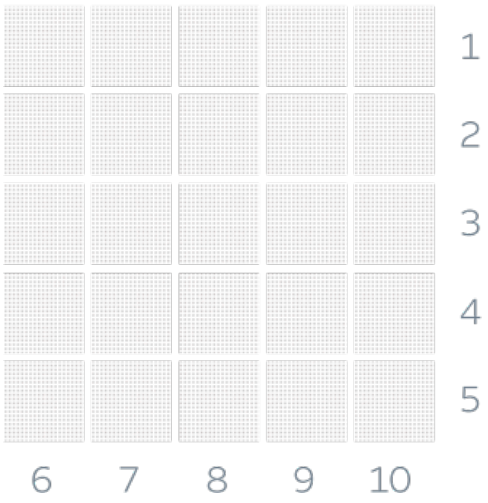
28.2 Python code

```
from z3 import *

def generate_hooks(size):
    hooks = []
    for n in range(1, size + 1):
        hook = []
        for i in range(n):
            for j in range(n):
                if i == n - 1 or j == n - 1:
                    hook.append((i, j))
        hooks.append(hook)
    return hooks


row_sums = [26, 42, 11, 22, 42, 36, 29, 32, 45]
col_sums = [31, 19, 45, 16, 5, 47, 28, 49, 45]
```

```
def solve_number_hook_puzzle():
    solver = Solver()

    # Create a 9x9 grid of integer variables
    grid = [[Int(f"cell_{i}_{j}") for j in range(9)] for i in range(9)]


    # Add constraints for row and column sums
    for i in range(9):
        solver.add(Sum(grid[i]) == row_sums[i])
        solver.add(Sum([grid[j][i] for j in range(9)]) == col_sums[i])

    # Generate hooks
    hooks = generate_hooks(9)

    # Add constraints for the hooks
    for i, hook in enumerate(hooks):
        value = i + 1  # Values from 1 to 9
        cells_in_hook = [grid[r][c] for r, c in hook]
        solver.add(Sum([If(cell == value, 1, 0) for cell in
cells_in_hook]) == value)
        for r, c in hook:
            solver.add(Or(grid[r][c] == value, grid[r][c] == 0))

    # Ensure all numbers are between 0 and 9 (0 for empty cells)
    for row in grid:
        for cell in row:
            solver.add(And(cell >= 0, cell <= 9))

    # Check if the puzzle is solvable
    if solver.check() == sat:
        model = solver.model()
        return [[model.evaluate(grid[i][j]).as_long() for j in
range(9)] for i in range(9)]
    else:
        return None

# Solve the puzzle
solution = solve_number_hook_puzzle()

# Print the solution
if solution:
    for row in solution:
        print(" ".join(map(lambda x: str(x) if x != 0 else '.', row)))
else:
    print("No solution found.")
```

29 Sum of squares

Place a digit in each of the 25 spots in the below 5×5 grid, so that each 5-digit number (leading zeroes are ok) reading across and reading down is divisible by the number outside the grid, trying to maximize the sum of the 25 numbers you enter. An example of a completed grid with sum 100 is presented on the right.



Example

1	6	2	3	5	1
5	2	4	6	0	2
0	4	8	9	3	3
2	4	8	6	8	4
4	7	0	3	0	5
6	7	8	9	10	

29.1 Solution

The code below gives the following solution

```
9 8 9 9 9
9 9 9 9 8
7 9 8 9 9
9 9 8 9 6
8 9 8 9 0
Total sum: 205
```

29.2 Python code

```
from z3 import *

row_divisors = [1, 2, 3, 4, 5]
col_divisors = [6, 7, 8, 9, 10]

def solve_grid_puzzle():
    opt = Optimize()
    grid = [[Int(f"cell_{i}_{j}") for j in range(5)] for i in range(5)]

    for row in grid:
        for cell in row:
            opt.add(And(cell >= 0, cell <= 9))

    def make_number(digits):
        return Sum([digits[i] * 10**(4-i) for i in range(5)])

    for i, row in enumerate(grid):
```

```
            opt.add(make_number(row) % row_divisors[i] == 0)

        for j in range(5):
            column = [grid[i][j] for i in range(5)]
            opt.add(make_number(column) % col_divisors[j] == 0)

        total_sum = Sum([cell for row in grid for cell in row])
        opt.maximize(total_sum)

        if opt.check() == sat:
            m = opt.model()
            result = [[m.evaluate(grid[i][j]).as_long() for j in range(5)]
for i in range(5)]
            return result, m.evaluate(total_sum).as_long()
        else:
            return None, None

solution, total = solve_grid_puzzle()

if solution:
    for row in solution:
        print(" ".join(map(str, row)))
    print(f"Total sum: {total}")
else:
    print("No solution found.")
```

30 Well Well Well

A 7-by-7′ well is dug. It has a peculiar shape: its depth varies from one 1′-by-1′ section to another, as shown below. Each section is marked with its depth. (E.g., the deepest section is 49′ deep.)

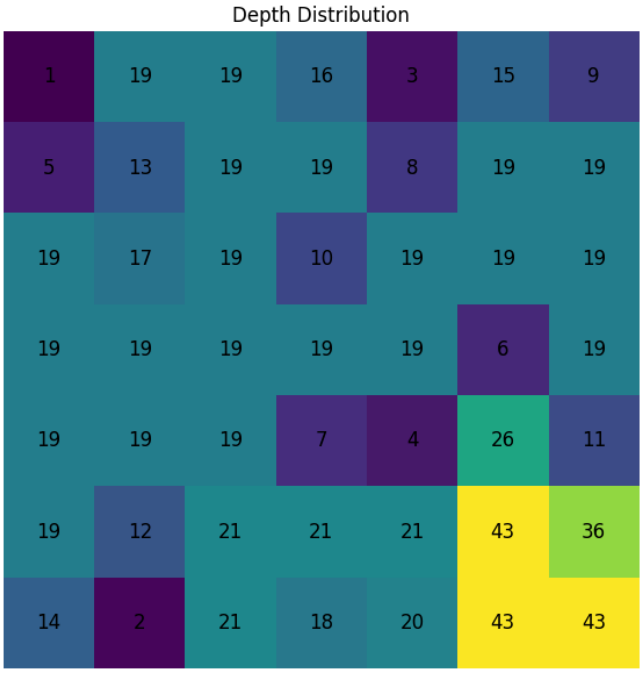
1	5	27	22	28	40	14
39	13	17	30	41	12	2
32	35	24	25	19	47	34
16	33	10	42	7	44	18
3	8	45	37	4	21	20
15	46	38	6	26	48	49
9	23	31	29	11	36	43

Water is poured into the well from a point above the section marked 1, at a rate of 1 cubic foot per minute. Assume that water entering a region of constant depth immediately disperses to any orthogonally adjacent lower-depth regions evenly along its exposed perimeter.

After how many minutes will the water level on section 43 begin to rise?

30.1 Solution

The grid below shows the depths of the well after 360 minutes. In the next minute the water in the last well will rise.



30.2 Python code

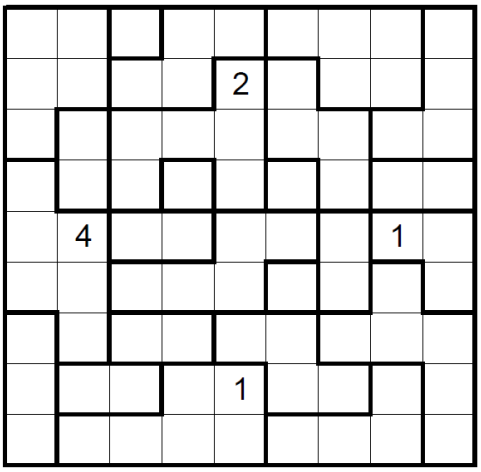
```
import networkx as nx
from fractions import Fraction
from matplotlib import pyplot as plt
import numpy as np

def find_next_deeper_neighbors(G, start):
    depth = G.nodes[start]['depth']
    equal_depth_nodes = {start}
    frontier = [start]
    while frontier:
        node = frontier.pop(0)
        for neighbor in G.neighbors(node):
            if G.nodes[neighbor]['depth'] == depth and neighbor not in
equal_depth_nodes:
                equal_depth_nodes.add(neighbor)
                frontier.append(neighbor)
    deeper_neighbors = set()
    for node in equal_depth_nodes:
        for neighbor in G.neighbors(node):
            if G.nodes[neighbor]['depth'] > depth:
                deeper_neighbors.add(neighbor)
    return list(equal_depth_nodes), list(deeper_neighbors)

def distribute_water(G, node, water_amount):
    deeper_neighbors = [neighbor for neighbor in G.neighbors(node)
                        if G.nodes[neighbor]['depth'] > G.nodes[node]
['depth']]
    if deeper_neighbors:
        water_per_neighbor = Fraction(water_amount,
len(deeper_neighbors))
        for neighbor in deeper_neighbors:
            distribute_water(G, neighbor, water_per_neighbor)
    else:
        equal_nodes, next_deeper_neighbors =
find_next_deeper_neighbors(G, node)
        if next_deeper_neighbors:
            water_per_neighbor = Fraction(water_amount,
len(next_deeper_neighbors))
            for node in next_deeper_neighbors:
                distribute_water(G, node, water_per_neighbor)
        else:
            if equal_nodes:
                water_per_neighbor = Fraction(water_amount,
len(equal_nodes))
                for node in equal_nodes:
                    G.nodes[node]['depth'] -=
Fraction(water_per_neighbor,1)
            else:
                G.nodes[node]['depth'] -= Fraction(water_amount,1)

def plot_depths(G):
```

```
    max_x = max(node[0] for node in G.nodes())
    max_y = max(node[1] for node in G.nodes())
    depths = np.zeros((max_y + 1, max_x + 1))
    for node, data in G.nodes(data=True):
        depths[node[1], node[0]] = data['depth']
    fig, ax = plt.subplots(figsize=(6,6))
    im = ax.imshow(depths)
    for i in range(depths.shape[0]):
        for j in range(depths.shape[1]):
            text = ax.text(j, i, int(round(depths[i, j],0)),
                           ha="center", va="center", color="black",
fontsize=12)
    ax.set_title("Depth Distribution")
    plt.axis('off')
    plt.tight_layout()
    plt.show()

def simulate_water_flow(G):
    t = 0
    while G.nodes[(6,6)]['depth'] == 43:
        distribute_water(G,(0,0),Fraction(1,1))
        t +=1
    print("The amount of time required to start filling well (6,6) is",
t-1)

def create_well_depth_lattice():
    depths = [ [1,5,27,22,28,40,14],
          [39,13,17,30,41,12,2],
          [32,35,24,25,19,47,34],
          [16,33,10,42,7,44,18],
          [3,8,45,37,4,21,20],
          [15,46,38,6,26,48,49],
          [9,23,31,29,11,36,43]]
    G = nx.grid_2d_graph(7,7)
    for i in range(7):
        for j in range(7):
            G.nodes[(i,j)]['depth']=Fraction(depths[i][j],1)
    return G

G = create_well_depth_lattice()
simulate_water_flow(G)
plot_depths(G)
```

31 Block Party

Block Party

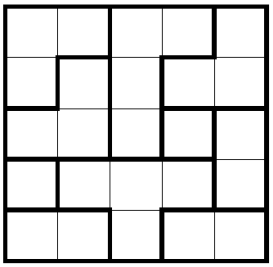
Fill each region with the digits 1 through N, where N is the number of cells in the region. If an integer K is written in a cell, that means that the nearest value of K (looking only horizontally or vertically) can be found exactly K cells away.



Once the grid is completed, take the largest "horizontally concatenated number" from each region and compute the sum of these values*. Enter this sum as your answer. Good luck!

(*This is similar to the answer computation from our May 2017 puzzle. Also, it is demonstrated in the Example below.)

Example grid



Completed grid

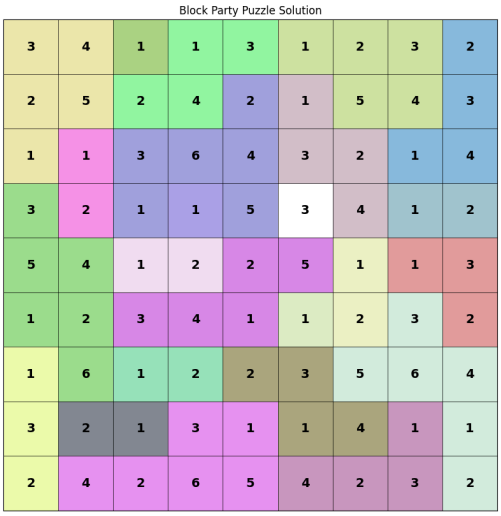
3	2	4	3	2
1	3	2	1	3
1	2	1	1	2
1	1	2	3	1
1	2	4	2	1

Example answer:

32 + 43 + 13 + 12 + 1 + 2 + 1 + 123 + 12 + 21 = **260**

31.1 Solution

Using the Python code below which makes use of the **z3** library we get the solution below



31.2 Python code

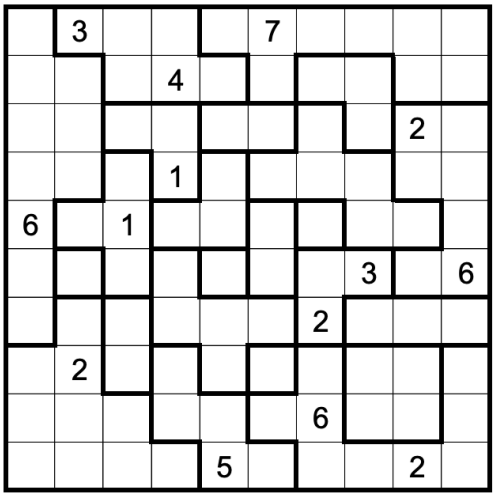
```
from z3 import *
import random
from matplotlib import pyplot as plt
import numpy as np


def solve_block_party(grid):
    solver = Solver()
    n = 9  # Grid size

    # Create Z3 variables for each cell
    cells = [[Int(f"cell_{i}_{j}") for j in range(n)] for i in
range(n)]

    for i in range(n):
        for j in range(n):
            solver.add(And(1 <= cells[i][j], cells[i][j] <= 9))

    # Process grid constraints
    for item in grid:
        if isinstance(item, tuple):  # Pre-filled number
            i, j, value = item
            solver.add(cells[i][j] == value)
        elif isinstance(item, list):  # Region
            size = len(item)
            solver.add(Distinct([cells[x][y] for x, y in item]))
            for x, y in item:
                solver.add(And(1 <= cells[x][y], cells[x][y] <= size))

    # K value rule for all cells
    for i in range(n):
        for j in range(n):
            rule_constraints = []
            for k in range(1, 10):  # k can be 1 to 9
                # Left
```

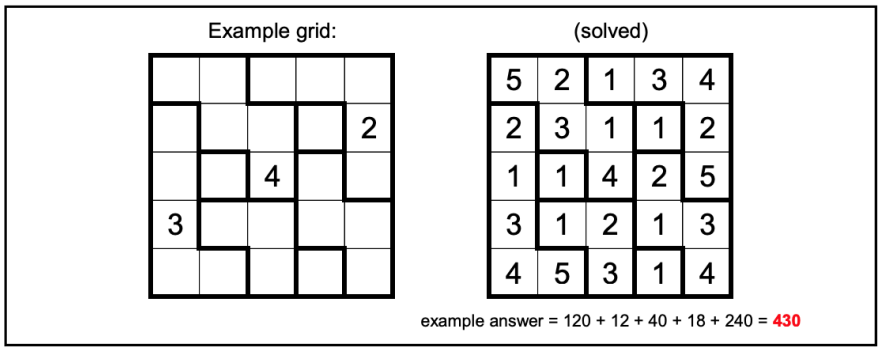
```
                if j >= k:
                    rule_constraints.append(
                        And(cells[i][j] == k, cells[i][j-k] == k,
                            And([cells[i][jj] != k for jj in range(j-
k+1, j)]))
                    )
                # Right
                if j + k < n:
                    rule_constraints.append(
                        And(cells[i][j] == k, cells[i][j+k] == k,
                            And([cells[i][jj] != k for jj in range(j+1,
j+k)]))
                    )
                # Up
                if i >= k:
                    rule_constraints.append(
                        And(cells[i][j] == k, cells[i-k][j] == k,
                            And([cells[ii][j] != k for ii in range(i-
k+1, i)]))
                    )
                # Down
                if i + k < n:
                    rule_constraints.append(
                        And(cells[i][j] == k, cells[i+k][j] == k,
                            And([cells[ii][j] != k for ii in range(i+1,
i+k)]))
                    )

            solver.add(Or(rule_constraints))

    # Solve the puzzle
    if solver.check() == sat:
        model = solver.model()
        solution = [[model.evaluate(cells[i][j]).as_long() for j in
range(n)] for i in range(n)]
        return solution
    else:
        return None

# 9x9 grid representation based on the bold lines in the image
grid = [
    # Pre-filled numbers: (row, column, value)
    (1, 4, 2), (4, 1, 4), (4, 7, 1), (7, 4, 1),
    # Regions: list of (row, column) coordinates
    [[0,0],[0,1],[1,0],[1,1],[2,0]],
    [[2,1],[3,1]],
    [[0,2]],
    [[0,3],[0,4],[1,2],[1,3]],
    [[1,4],[2,2],[2,3],[2,4],[3,2],[3,4]],
    [[3,3]],
    [[0,5],[0,6],[0,7],[1,6],[1,7]],
    [[0,8],[1,8],[2,8],[2,7]],
    [[3,0],[4,0],[5,0],[4,1],[5,1],[6,1]],
```

```
    [[6,0],[7,0],[8,0]],
    [[3,7],[3,8]],
    [[1,5],[2,5],[2,6],[3,6]],
    [[4,2],[4,3]],
    [[4,4],[4,5],[5,2],[5,3],[5,4]],
    [[5,5]],
    [[4,6],[5,6]],
    [[4,7],[4,8],[5,8]],
    [[6,2],[6,3]],
    [[6,4],[6,5],[7,5],[7,6]],
    [[5,7],[6,6],[6,7],[6,8],[7,8],[8,8]],
    [[7,1],[7,2]],
    [[7,3],[7,4],[8,1],[8,2],[8,3],[8,4]],
    [[7,7],[8,7],[8,6],[8,5]]
]

solution = solve_block_party(grid)

def generate_distinct_colors(n):
    colors = []
    for i in range(n):
        while True:
            color = "#"+''.join([random.choice('0123456789ABCDEF') for
j in range(6)])
            if color not in colors:
                colors.append(color)
                break
    return colors

def visualize_solution(solution, grid):
    n = 9
    fig, ax = plt.subplots(figsize=(10, 10))

    # Generate distinct colors for each region
    region_colors = generate_distinct_colors(len([item for item in grid
if isinstance(item, list)]))

    # Draw grid
    for i in range(n+1):
        ax.axhline(i, color='black', linewidth=0.5)
        ax.axvline(i, color='black', linewidth=0.5)

    # Color regions
    color_index = 0
    for region in grid:
        if isinstance(region, list):
            for i, j in region:
                ax.add_patch(plt.Rectangle((j, n-i-1), 1, 1,
fill=True,

facecolor=region_colors[color_index], alpha=0.5))
            color_index += 1
```

```
    # Add numbers
    for i in range(n):
        for j in range(n):
            ax.text(j+0.5, n-i-0.5, str(solution[i][j]), ha='center',
va='center', fontsize=12, fontweight='bold')

    # Set limits and remove ticks
    ax.set_xlim(0, n)
    ax.set_ylim(0, n)
    ax.set_xticks([])
    ax.set_yticks([])

    # Add title
    plt.title('Block Party Puzzle Solution')

    return fig

if solution:
    for row in solution:
        print(" ".join(map(str, row)))

    fig = visualize_solution(solution, grid)
    plt.show()
else:
    print("No solution found")
```

32 Block Party 4

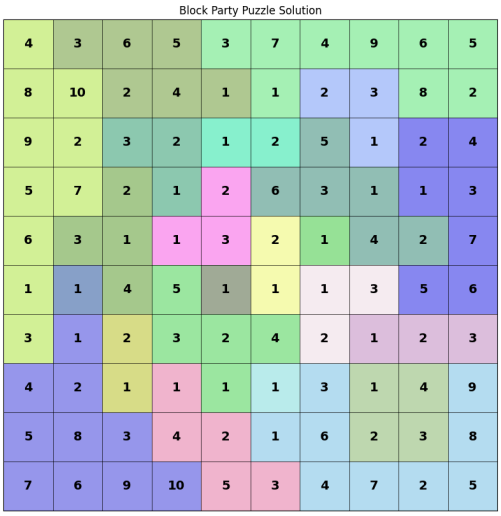




Fill each region with the numbers 1 through N, where N is the number of cells in the region. For each number K in the grid, the nearest K via taxicab distance must be exactly K cells away.

32.1 Solution

Here is the solution to the puzzle:



32.2 Python code

```
from z3 import *
import matplotlib.pyplot as plt
import random

def solve_block_party(grid):
    solver = Solver()
    n = 10  # Grid size

    # Create Z3 variables for each cell
    cells = [[Int(f"cell_{i}_{j}") for j in range(n)] for i in
range(n)]

    # Process grid constraints
    for item in grid:
        if isinstance(item, tuple):  # Pre-filled number
            i, j, value = item
            solver.add(cells[i][j] == value)
        elif isinstance(item, list):  # Region
            size = len(item)
            solver.add(Distinct([cells[x][y] for x, y in item]))
            for x, y in item:
                solver.add(And(1 <= cells[x][y], cells[x][y] <= size))

    # K value rule for all cells
    for i in range(n):
        for j in range(n):
            k = cells[i][j]

            # No cells with value k closer than distance k
            solver.add(And([
                Implies(
                    cells[x][y] == k,
                    abs(x - i) + abs(y - j) >= k
                )
                for x in range(n) for y in range(n)
                if (x != i or y != j)
            ]))

            # At least one cell with value k at exactly distance k
            solver.add(Or([
                And(cells[x][y] == k, abs(x - i) + abs(y - j) == k)
                for x in range(n) for y in range(n)
                if (x != i or y != j)
            ]))

    # Ensure all cells have values between 1 and 9
    for i in range(n):
        for j in range(n):
            solver.add(And(1 <= cells[i][j], cells[i][j] <= 10))

    # Solve the puzzle
```

```
    if solver.check() == sat:
        model = solver.model()
        solution = [[model.evaluate(cells[i][j]).as_long() for j in
range(n)] for i in range(n)]
        return solution
    else:
        return None

def generate_distinct_colors(n):
    colors = []
    for i in range(n):
        while True:
            color = "#"+''.join([random.choice('0123456789ABCDEF') for
j in range(6)])
            if color not in colors:
                colors.append(color)
                break
    return colors

def visualize_solution(solution, grid):
    n = 10
    fig, ax = plt.subplots(figsize=(10, 10))

    # Generate distinct colors for each region
    region_colors = generate_distinct_colors(len([item for item in grid
if isinstance(item, list)]))

    # Draw grid
    for i in range(n+1):
        ax.axhline(i, color='black', linewidth=0.5)
        ax.axvline(i, color='black', linewidth=0.5)

    # Color regions
    color_index = 0
    for region in grid:
        if isinstance(region, list):
            for i, j in region:
                ax.add_patch(plt.Rectangle((j, n-i-1), 1, 1,
fill=True,

facecolor=region_colors[color_index], alpha=0.5))
            color_index += 1

    # Add numbers
    for i in range(n):
        for j in range(n):
            ax.text(j+0.5, n-i-0.5, str(solution[i][j]), ha='center',
va='center', fontsize=14, fontweight='bold')

    # Set limits and remove ticks
    ax.set_xlim(0, n)
    ax.set_ylim(0, n)
    ax.set_xticks([])
```

```
    ax.set_yticks([])

    # Add title
    plt.title('Block Party Puzzle Solution')

    return fig

# 9x9 grid representation based on the bold lines in the image
grid = [
    # Pre-filled numbers: (row, column, value)
    (0, 1, 3), (0, 5, 7), (2, 8, 2),
    (1, 3, 4), (3, 3, 1), (4, 0, 6), (4, 2, 1),
    (5, 7, 3), (5, 9, 6), (6, 6, 2), (7, 1, 2),
    (8, 6, 6), (9, 4, 5), (9, 8, 2),

    # Regions: list of (row, column) coordinates
    [[0,0], [1,0], [1,1], [2,0],[2,1],[3,0],[3,1],[4,0],[5,0],[6,0]],
    [[0,1],[0,2],[0,3],[1,2],[1,3],[1,4]],
    [[0,4],[0,5],[0,6],[0,7],[0,8],[0,9],[1,5],[1,8],[1,9]],
    [[1,6],[1,7],[2,7]],
    [[2,2],[2,3],[3,3]],
    [[2,4],[2,5]],
    [[2,6],[3,5],[3,6],[3,7],[4,7],[4,8]],
    [[3,2],[4,1],[4,2],[5,2]],
    [[5,1]],
    [[3,4],[4,3],[4,4]],
    [[2,8],[2,9],[3,8],[3,9],[4,9],[5,8],[5,9]],
    [[4,6]],
    [[5,4]],
    [[4,5],[5,5]],
    [[5,3],[6,3],[6,4],[6,5],[7,4]],
    [[5,6],[5,7],[6,6]],
    [[6,7],[6,8],[6,9]],
    [[6,2],[7,2]],
    [[6,1],[7,0],[8,0],[9,0],[7,1],[8,1],[9,1],[8,2],[9,2],[9,3]],
    [[7,3],[8,3],[8,4],[9,4],[9,5]],
    [[7,5]],
    [[7,6],[8,5],[8,6],[9,6],[9,7],[9,8],[9,9],[8,9],[7,9]],
    [[7,7],[7,8],[8,7],[8,8]]
]

solution = solve_block_party(grid)

if solution:
    print("Solution found:")
    for row in solution:
        print(" ".join(map(str, row)))

    fig = visualize_solution(solution, grid)
    plt.show()
else:
    print("No solution found")
```

33 Figurine figuring

Jane received 78 figurines as gifts this holiday season: 12 drummers drumming, 11 pipers piping, 10 lords a-leaping, etc., down to 1 partridge in a pear tree. They are all mixed together in a big bag. She agrees with her friend Alex that this seems like too many figurines for one person to have, so she decides to give some of her figurines to Alex. Jane will uniformly randomly pull figurines out of the bag one at a time until she pulls out the partridge in a pear tree, and will give Alex all of the figurines she pulled out of the bag (except the partridge, that's Jane's favorite).

If n is the maximum number of any one type of ornament that Alex gets, what is the expected value of n, to seven significant figures?

33.1 Solution

Using Monte-Carlo simulation, the expected value of n is 6.87.

33.2 Python code

```
import random
from collections import Counter

def simulate_picking():
    letters = [chr(65 + i) for i in range(12) for _ in range(i + 1)]
    random.shuffle(letters)
    picks = []
    while True:
        pick = letters.pop()
        picks.append(pick)
        if pick == 'A':
            break
    return max(Counter(picks).values())

def monte_carlo_simulation(num_simulations):
    total_max_picks = 0
    for _ in range(num_simulations):
        total_max_picks += simulate_picking()
    return total_max_picks / num_simulations

# Run the simulation
num_simulations = 10000000
expected_value = monte_carlo_simulation(num_simulations)
print(f"Estimated expected value of the maximum number of a particular
figure chosen: {expected_value}")
```

34 Queens

Your goal is to have exactly one queen in each row, column, and color region. Two queens cannot touch each other, not even diagonally.

34.1 Solution

Here are solutions to two Queens puzzles using the Python code below.

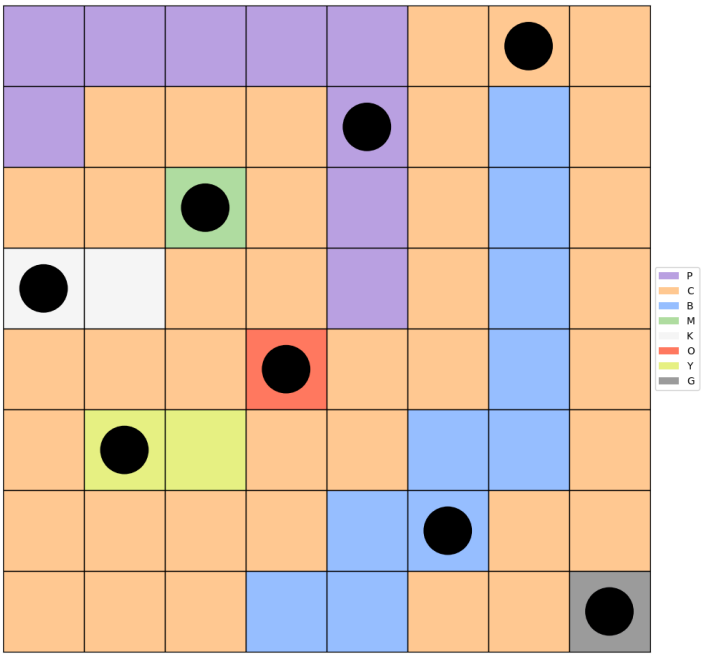


Figure 37: Puzzle 1

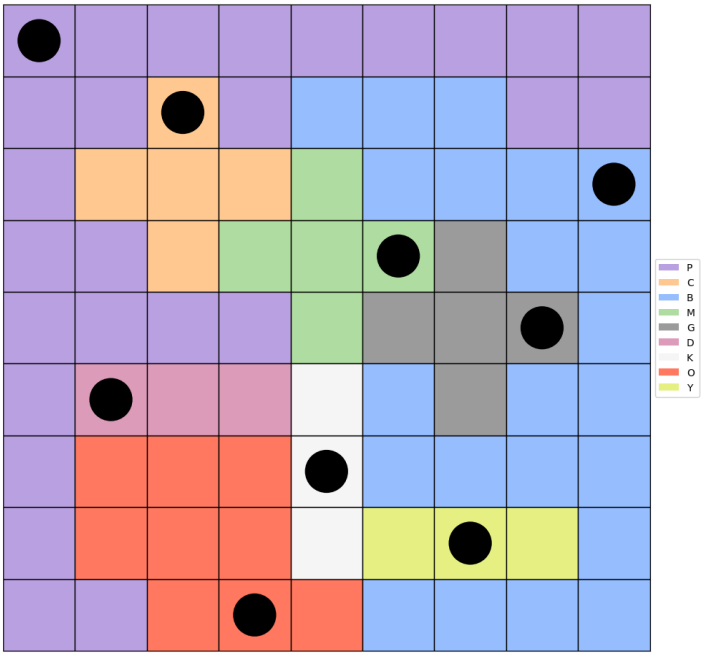


Figure 38: Puzzle 2

34.2 Python code

```
from z3 import *
import matplotlib.pyplot as plt
from matplotlib.colors import to_rgba

def parse_color_grid(color_grid):
    n = len(color_grid)
    color_data = {}
    preplaced_queens = []
    for i, row in enumerate(color_grid):
        for j, cell in enumerate(row):
            color = cell[1] if cell.startswith('Q') else cell
            if color not in color_data:
                color_data[color] = []
            color_data[color].append((i, j))
            if cell.startswith('Q'):
                preplaced_queens.append((i, j))
    return color_data, preplaced_queens

def solve_n_queens_color(n, color_data, preplaced_queens):
    solver = Solver()
    grid = [[Int(f"cell_{i}_{j}") for j in range(n)] for i in range(n)]
    for row in grid:
        for cell in row:
            solver.add(Or(cell == 0, cell == 1))
    for i, j in preplaced_queens:
        solver.add(grid[i][j] == 1)
    for row in grid:
        solver.add(Sum(row) == 1)
    for j in range(n):
        solver.add(Sum([grid[i][j] for i in range(n)]) == 1)
    directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1),
(1,0), (1,1)]
    for i in range(n):
        for j in range(n):
            for di, dj in directions:
                ni, nj = i + di, j + dj
                if 0 <= ni < n and 0 <= nj < n:
                    solver.add(Implies(grid[i][j] == 1, grid[ni][nj] ==
0))
    for color, squares in color_data.items():
        solver.add(Sum([grid[i][j] for i, j in squares]) == 1)
    if solver.check() == sat:
        return solver.model(), grid
    else:
        return None, None

COLOR_MAP = {
    'P': '#bba3e2',
    'Y': '#e6f386',
    'O': '#ff7b60',
    'B': '#96beff',
```

```
    'C': '#ffc992',
    'G': '#9e9e9f',
    'K': '#f5f6f7',
    'M': '#b3dfa0',
    'D': '#dc9fbc'
}

def visualize_solution(model, grid, color_data, n):
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.set_aspect('equal')
    ax.set_xlim(0, n)
    ax.set_ylim(0, n)
    ax.set_xticks([])
    ax.set_yticks([])
    for i in range(n):
        for j in range(n):
            color = next(c for c, squares in color_data.items() if (i,
j) in squares)
            ax.add_patch(plt.Rectangle((j, n-1-i), 1, 1,
facecolor=COLOR_MAP.get(color, 'white'), edgecolor='black'))
            if model.evaluate(grid[i][j]) == 1:
                ax.add_patch(plt.Circle((j+0.5, n-0.5-i), 0.3,
facecolor='black'))
    legend_elements = [plt.Rectangle((0,0),1,1,facecolor=COLOR_MAP[c])
for c in color_data]
    ax.legend(legend_elements, color_data.keys(), loc='center left',
bbox_to_anchor=(1, 0.5))
    plt.tight_layout()
    plt.show()

puzzle1 = [
    ['P', 'P', 'P', 'P', 'P', 'C','QC','C'],
    ['P', 'C', 'C', 'C', 'QP', 'C','B','C'],
    ['C', 'C', 'M', 'C', 'P', 'C','B','C'],
    ['K', 'K', 'C', 'C', 'P', 'C','B','C'],
    ['C', 'C', 'C', 'O', 'C', 'C','B','C'],
    ['C', 'Y', 'Y', 'C', 'C', 'B','B','C'],
    ['C', 'C', 'C', 'C', 'B', 'B','C','C'],
    ['C', 'C', 'C', 'B', 'B', 'C','C','G'],
]

puzzle2 = [
    ['P', 'P', 'P', 'P', 'P', 'P','P','P','P'],
    ['P', 'P', 'C', 'P', 'B', 'B','B','P','P'],
    ['P', 'C', 'C', 'C', 'M', 'B','B','B','B'],
    ['P', 'P', 'C', 'M', 'M', 'M','G','B','B'],
    ['P', 'P', 'P', 'P', 'M', 'G','G','G','B'],
    ['P', 'D', 'D', 'D', 'K', 'B','G','B','B'],
    ['P', 'O', 'O', 'O', 'K', 'B','B','B','B'],
    ['P', 'O', 'O', 'O', 'K', 'Y','Y','Y','B'],
    ['P', 'P', 'O', 'O', 'O', 'B','B','B','B'],
]
```

```
def solve(color_grid):
    color_data, preplaced_queens = parse_color_grid(color_grid)
    n = len(color_grid)
    model, grid = solve_n_queens_color(n, color_data, preplaced_queens)
    if model:
        print("Solution found!")
        visualize_solution(model, grid, color_data, n)
    else:
        print("No solution found.")

solve(puzzle2)
```

Bibliography