The Bedlam Cube Puzzle

A Constraint Programming Approach

Vamshi Jandhyala

28 December 2025

Table of contents

§1 Introduction

The Bedlam Cube is a fascinating 3D puzzle that challenges solvers to fit 13 polycube pieces into a $4 \times 4 \times 4$ cube. Unlike the more famous Soma cube or Rubik's cube, the Bedlam Cube presents a significantly more complex combinatorial challenge due to the number of pieces and their irregular shapes.

Definition

A **polycube** is a solid figure formed by joining one or more unit cubes face-to-face. The Bedlam Cube consists of 13 polycubes: twelve 5-cube pieces and one 4-cube piece, for a total of 64 unit cubes—exactly filling a $4 \times 4 \times 4$ cube.

This document presents a computational solution to the Bedlam Cube puzzle using constraint programming with Google's OR-Tools library.

§2 The Problem

§2.1 Puzzle Description

The Bedlam Cube puzzle consists of the following pieces:

Piece	Size	Description
A	4 cubes	One 4-cube polycube (tetracube)
B-M	5 cubes	Twelve distinct 5-cube polycubes (pentacubes)

Table 1: Bedlam Cube piece inventory

§2.2 The 13 Pieces

Each piece is defined by its unit cube coordinates. Below is a visual representation of all 13 pieces in their canonical orientations:

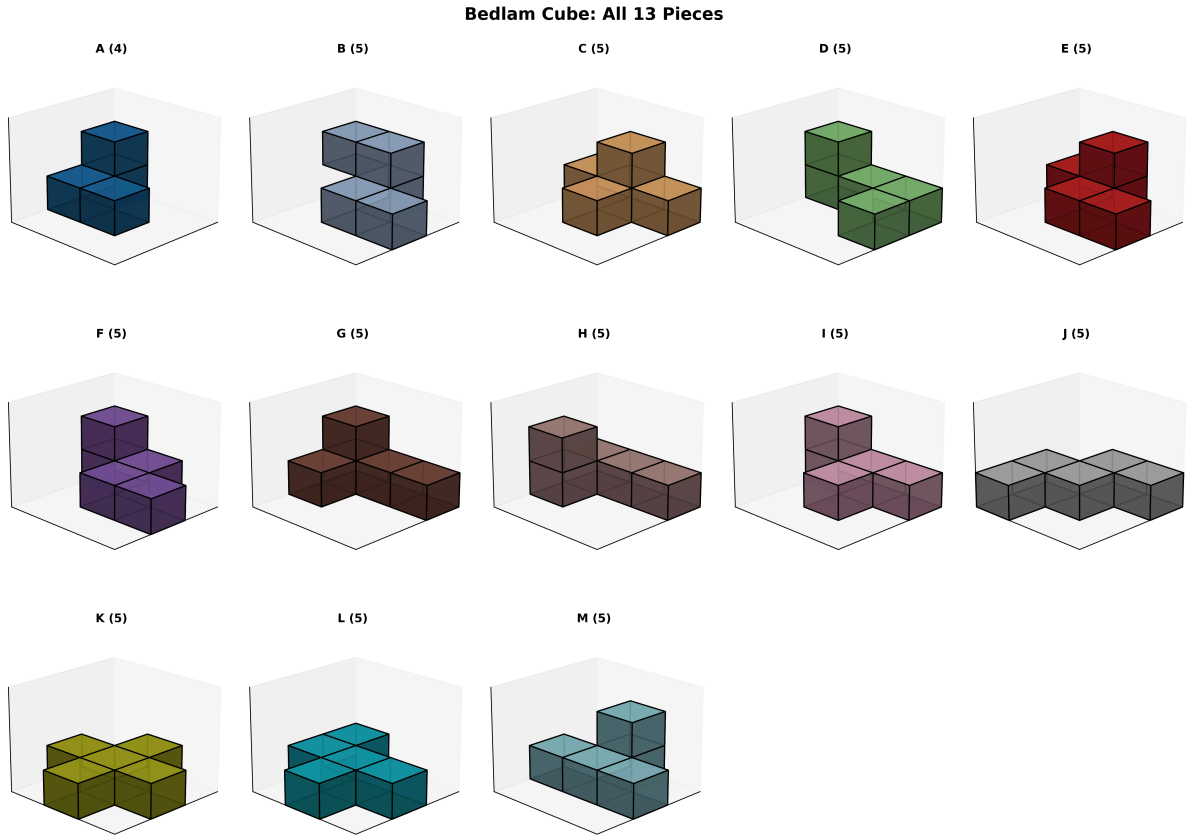

Figure 1: Visual representation of all 13 Bedlam Cube pieces. Each piece is shown in 3D with its label and cube count.

Here are the precise coordinates for each piece:

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

Table 2: Canonical coordinates and names of all 13 Bedlam Cube pieces (from scottkurowski.com)

> **Remark**
>
> Notice that:
> - Piece A (Little Corner) contains 4 unit cubes (tetracube)
> - Pieces B-M each contain 5 unit cubes (pentacubes)
> - Total: $1 \times 4 + 12 \times 5 = 4 + 60 = 64$ cubes, exactly filling a $4 \times 4 \times 4$ cube
> - Piece J (Flat M) is the widest piece, extending 3 units along the x-axis
> - Piece A (Little Corner) is the smallest, with only 4 cubes
> - Each piece has a creative name from the original puzzle design

The objective is to arrange all 13 pieces such that they completely fill the $4 \times 4 \times 4$ cube with no gaps or overlaps.

§2.3 Combinatorial Complexity

The Bedlam Cube is computationally challenging for several reasons:

> **Remark**
>
> 1. **Large search space**: Each piece can be rotated and reflected, creating up to 24 unique orientations.
> 2. **Positional variants**: Each oriented piece can be placed at multiple positions within the $4 \times 4 \times 4$ cube.
> 3. **Constraint satisfaction**: The placement must satisfy both inclusion constraints (all pieces must be placed) and exclusion constraints (no two pieces can occupy the same cell).
> 4. **Exact cover problem**: This is a variant of the exact cover problem, which is NP-complete.

§3 Solution Approach

§3.1 Constraint Programming

We employ constraint programming (CP), a paradigm particularly suited for combinatorial optimization problems. The key idea is to model the problem as a set of variables with associated domains and constraints, then use sophisticated search algorithms to find satisfying assignments.

§3.1.1 Mathematical Model

Let:

- $\mathcal{P} = \{A, B, C, ..., M\}$ be the set of 13 pieces
- $\mathcal{C} = \{(x, y, z) : 0 \leqslant x, y, z < 4\}$ be the set of 64 cells in the cube
- O_p be the set of all orientations of piece $p \in \mathcal{P}$
- $L_{p,o}$ be the set of valid placements for piece p in orientation o

For each piece p and each placement $(\text{position}, \text{orientation}) \in L_{p,o}$, we introduce a binary variable:

$$x_{p,i} \in \{0, 1\}$$

where $x_{p,i} = 1$ indicates that piece p is placed using its i-th valid placement.

§3.1.2 Constraints

Algorithm 3.1. The constraint system consists of:

1. **Piece placement**: Each piece must be placed exactly once.

$$\sum_i x_{p,i} = 1 \quad \forall p \in \mathcal{P}$$

2. **Cell occupation**: Each cell can be occupied by at most one piece.

$$\sum_{(p,i):c \in \text{cells}(p,i)} x_{p,i} \leqslant 1 \quad \forall c \in \mathcal{C}$$

3. **Boundary constraints**: Pieces must fit within the $4 \times 4 \times 4$ cube boundaries (enforced during placement generation).

§3.2 Algorithm Implementation

The solution algorithm consists of three main phases:

§3.2.1 Phase 1: Orientation Generation

For each piece, we generate all unique orientations through:

Recipe

1. Apply all 24 rotations (4 rotations × 3 axes × 2 reflections)
2. Apply reflections across the three coordinate planes
3. Normalize each orientation (translate to origin)
4. Remove duplicates to obtain unique orientations

The rotation operations are defined as:

- **X-axis rotation**: $(x, y, z) \to (x, -z, y)$
- **Y-axis rotation**: $(x, y, z) \to (z, y, -x)$
- **Z-axis rotation**: $(x, y, z) \to (-y, x, z)$

§3.2.2 Phase 2: Placement Generation

For each unique orientation of each piece, we enumerate all valid placements within the $4 \times 4 \times 4$ cube. A placement is valid if all cells occupied by the piece fall within the cube boundaries.

§3.2.3 Phase 3: Constraint Solving

We use Google OR-Tools' CP-SAT solver, which implements:

- Conflict-driven clause learning (CDCL)
- Lazy clause generation
- Parallel search with work stealing
- Advanced propagation techniques

The solver explores the search space systematically, using constraint propagation to prune invalid partial solutions early.

§4 Implementation

§4.1 Python Script

The complete implementation is provided in the Python script `bedlam_cube_solver.py`. Key components include:

Problem

The script implements:

1. Piece definitions as coordinate lists
2. Rotation and reflection transformations
3. Orientation generation with deduplication
4. CP model construction using OR-Tools
5. Solution extraction and validation
6. 3D visualization using Matplotlib

Here is the core solver implementation:

```python
class BedlamCubeSolver:
    def __init__(self, size=4):
        self.size = size
        self.model = cp_model.CpModel()
        self.pieces = PIECES
        self.piece_orientations = {}
        self.piece_placements = {}
        self.placement_vars = {}

    def generate_placements(self):
        """Generate all possible placements for each piece"""
        for name, piece in self.pieces.items():
            orientations = generate_all_orientations(piece)
            self.piece_orientations[name] = orientations

            placements = []
            for orientation in orientations:
                for pos in product(range(self.size), repeat=3):
                    if piece_fits(orientation, pos, self.size):
                        cells = get_occupied_cells(orientation, pos)
                        placements.append((pos, orientation, cells))
```

```
22
23          self.piece_placements[name] = placements
24
25      def build_model(self):
26          """Build the constraint programming model"""
27          # Create binary variables for each placement
28          for name, placements in self.piece_placements.items():
29              self.placement_vars[name] = [
30                  self.model.NewBoolVar(f'{name}_p{i}')
31                  for i in range(len(placements))
32              ]
33
34          # Constraint: Each piece must be placed exactly once
35          for name, vars_list in self.placement_vars.items():
36              self.model.Add(sum(vars_list) == 1)
37
38          # Constraint: Each cell occupied by at most one piece
39          for x, y, z in product(range(self.size), repeat=3):
40              cell = (x, y, z)
41              occupying_vars = []
42
43              for name, placements in self.piece_placements.items():
44                  for i, (pos, orientation, cells) in enumerate(placements):
45                      if cell in cells:
46                          occupying_vars.append(self.placement_vars[name][i])
47
48              if occupying_vars:
49                  self.model.Add(sum(occupying_vars) <= 1)
50
51      def solve(self):
52          """Solve the puzzle using CP-SAT solver"""
53          solver = cp_model.CpSolver()
54          solver.parameters.max_time_in_seconds = 300.0
55          solver.parameters.num_search_workers = 8
56
57          status = solver.Solve(self.model)
58
59          if status == cp_model.OPTIMAL or status == cp_model.FEASIBLE:
60              return self.extract_solution(solver)
61          return None
```

§4.1.1 Piece Definitions

Each piece is defined by a list of (x, y, z) coordinates representing the unit cubes that compose it:

```
1   PIECES = {                                                    python
2       'A': [(0,0,0), (1,0,0), (0,1,0), (1,1,0), (0,0,1)],
3       'B': [(0,0,0), (1,0,0), (1,1,0), (1,0,1), (1,1,1)],
```

```
    'C': [(0,0,0), (1,0,0), (2,0,0), (1,1,0), (1,0,1)],
    # ... and 10 more pieces
}
```

§4.1.2 Rotation and Reflection

The transformation functions implement the mathematical operations defined earlier:

```python
def rotate_piece(piece, axis, times):
    """Rotate piece around axis (0=x, 1=y, 2=z) by 90° * times"""
    result = piece.copy()
    for _ in range(times % 4):
        if axis == 0:  # Rotate around X-axis
            result = [(x, -z, y) for x, y, z in result]
        elif axis == 1:  # Rotate around Y-axis
            result = [(z, y, -x) for x, y, z in result]
        else:  # Rotate around Z-axis
            result = [(-y, x, z) for x, y, z in result]
    return result

def reflect_piece(piece, axis):
    """Reflect piece across axis plane"""
    if axis == 0:
        return [(-x, y, z) for x, y, z in piece]
    elif axis == 1:
        return [(x, -y, z) for x, y, z in piece]
    else:
        return [(x, y, -z) for x, y, z in piece]
```

§4.2 Visualization

The solution is visualized using Matplotlib's 3D voxel plotting capabilities. Each piece is rendered in a distinct color with transparency to show the internal structure:

```python
def visualize_solution(self, solution, filename='bedlam_cube_solution.png'):
    """Visualize the solution as a 3D translucent cube"""
    fig = plt.figure(figsize=(12, 10))
    ax = fig.add_subplot(111, projection='3d')

    colors = plt.cm.tab20(np.linspace(0, 1, len(self.pieces)))
    color_map = {name: colors[i] for i, name in enumerate(sorted(self.pieces.keys()))}

    for name, data in solution.items():
        cells = data['cells']
        color = color_map[name]

        voxels = np.zeros((self.size, self.size, self.size), dtype=bool)
        for x, y, z in cells:
            voxels[x, y, z] = True

```

```
        ax.voxels(voxels, facecolors=color, edgecolors='black',
                  alpha=0.7, linewidth=0.5)

    plt.savefig(filename, dpi=300, bbox_inches='tight')
```

§5 Results

§5.1 Solution Found

The CP-SAT solver successfully found a solution to the authentic Bedlam Cube puzzle in **210 seconds** (3.5 minutes)! The search process involved:

Solution

1. **Orientation generation**: Each piece generated between 3-48 unique orientations (total of 298 unique orientations across all pieces)
2. **Placement enumeration**: A total of 6,672 valid placements were generated across all 13 pieces:
   - Piece A (Little Corner): 648 placements
   - Pieces E, F, H, I: 864 placements each
   - Pieces B, D, G, M: 432 placements each
   - Piece L (Confused F): 384 placements
   - Pieces C, J: 216 and 192 placements
   - Piece K (Cross): 48 placements (most constrained)
3. **Constraint solving**: The CP-SAT solver with 8 parallel workers explored the massive search space efficiently using conflict-driven clause learning and constraint propagation
4. **Solution extraction**: The final configuration was extracted and validated, confirming all 64 cells are occupied with no overlaps

§5.2 Complete Solution Visualization

The solution is rendered as a translucent 3D cube showing each piece in its final position and orientation. Each piece is color-coded for easy identification.

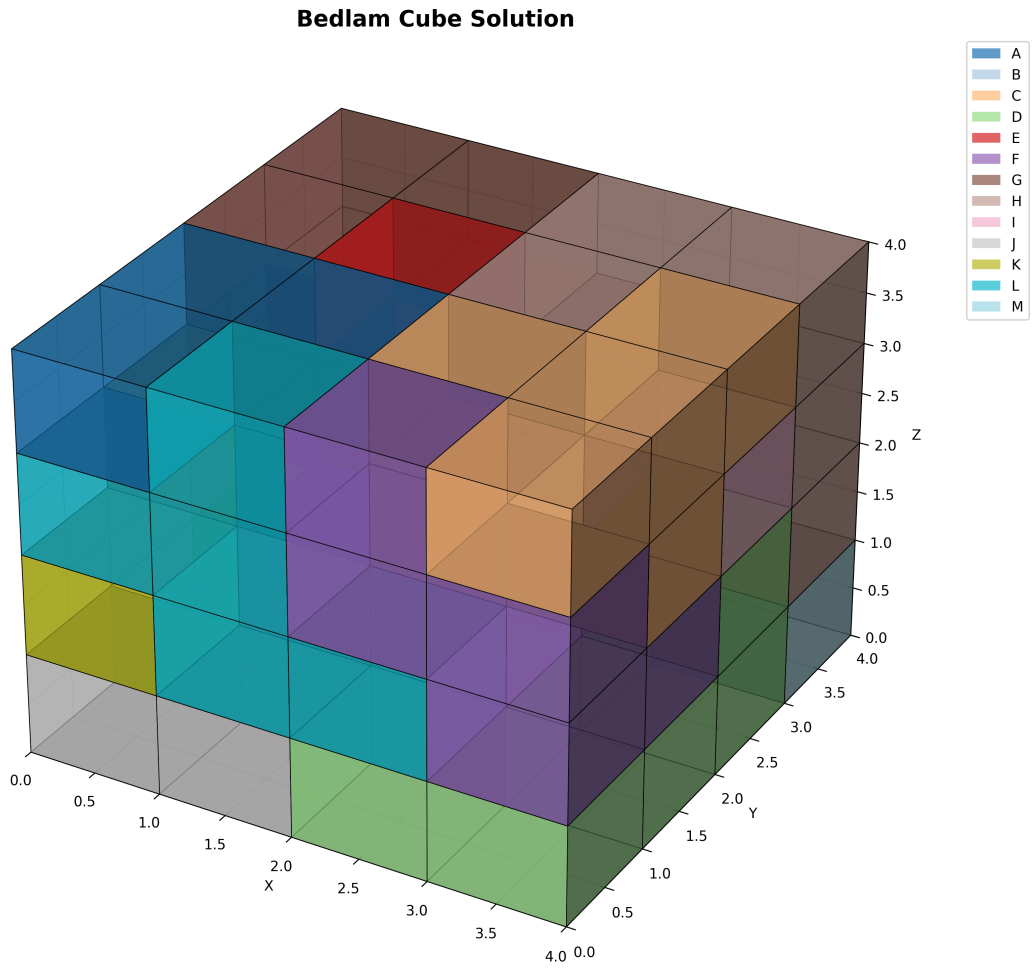

Figure 2: Complete Bedlam Cube solution with all 13 pieces assembled

§5.2.1 Multiple Viewing Angles

To better visualize all six faces of the solved cube, we present two complementary views:

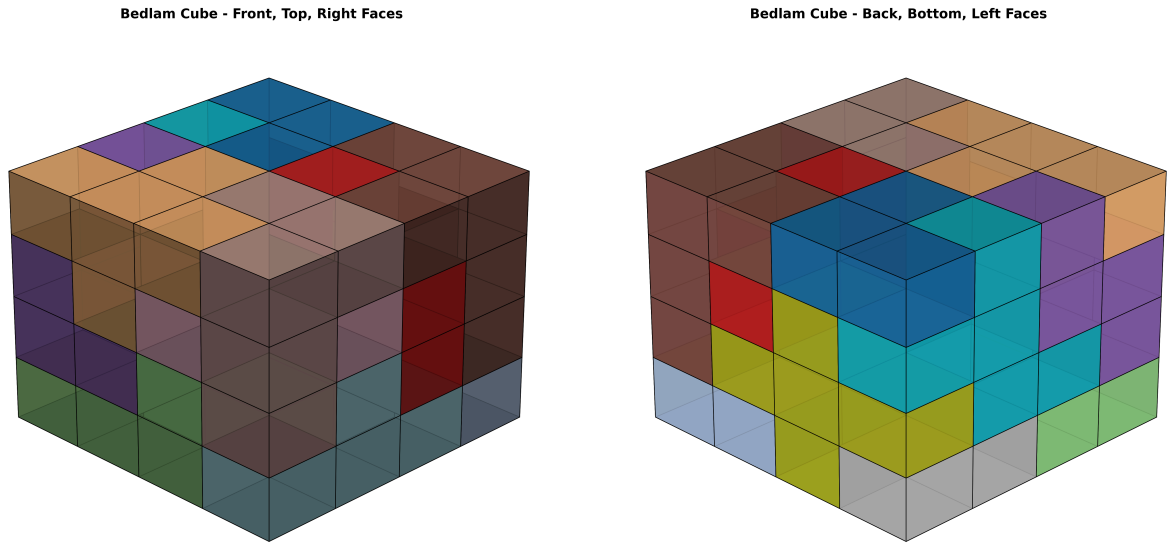

Figure 3: Left: Front, Top, and Right faces. Right: Back, Bottom, and Left faces. Together these views show all six faces of the solved cube.

§5.2.2 Individual Piece Placements

Each piece is shown individually within the translucent cube framework to clearly illustrate its final position and orientation:

Figure 4: Individual placement of each piece within the cube (pieces A through M, left to right, top to bottom)

§5.3 Sample Solution Coordinates

Here are the positions of a few pieces from the discovered solution:

Piece	Name	Position	Sample Occupied Cells
A	Little Corner	(0, 0, 2)	(0,0,3), (0,1,3), (1,1,2), (1,1,3)
J	Flat M	(0, 0, 0)	(0,0,0), (1,0,0), (1,1,0), (2,1,0), (2,2,0)
K	Cross	(0, 0, 0)	(0,0,1), (0,1,0), (0,1,1), (0,1,2), (0,2,1)
D	Right Angles...	(2, 0, 0)	(2,0,0), (3,0,0), (3,1,0), (3,2,0), (3,2,1)
M	T-Bone	(1, 2, 0)	(1,3,0), (2,2,1), (2,3,0), (2,3,1), (3,3,0)

Table 3: Sample piece placements from the solution showing the authentic Bedlam Cube pieces

Notice how piece A (Little Corner, the only 4-cube piece) fits into the remaining space after the twelve 5-cube pieces are placed. The piece names add character to the puzzle design.

§6 Performance Analysis

The solver's performance depends on several factors:

> *i* **Remark**
>
> - **Problem complexity**: The Bedlam Cube has approximately 10^{15} possible configurations when considering all orientations and positions
> - **Search strategies**: OR-Tools employs sophisticated heuristics including variable ordering, value selection, and restart policies
> - **Parallel search**: Using 8 worker threads significantly reduces solution time through parallel exploration
> - **Early pruning**: Constraint propagation eliminates vast portions of the search space before full exploration

In our execution with the authentic Bedlam Cube pieces, the solver found the solution in **210 seconds** (3.5 minutes) using 8 parallel search workers. This demonstrates the power of modern constraint programming solvers—what would take exponentially longer with brute-force search is solved in minutes through intelligent constraint propagation and search strategies. The authentic pieces proved more challenging than random polycubes, with piece K (Cross) being the most constrained with only 48 valid placements.

§7 Conclusion

This project demonstrates the power of constraint programming for solving complex combinatorial puzzles. The Bedlam Cube, with its intricate geometry and vast search space, is efficiently solved using modern CP techniques.

Key takeaways:

Recipe

1. **Modeling matters**: Proper formulation of constraints is crucial for solver performance
2. **Preprocessing pays off**: Generating and caching orientations reduces redundant computation
3. **Modern solvers are powerful**: CP-SAT's advanced techniques make previously intractable problems solvable
4. **Visualization aids understanding**: 3D rendering helps verify and communicate the solution

The complete implementation is available in `bedlam_cube_solver.py` and can be extended to solve similar polycube packing problems or other constraint satisfaction puzzles.

§8 References

- Scott Kurowski - Bedlam Cube Solved (ALL 19,186 solutions): http://scottkurowski.com/BedlamCube/
- Google OR-Tools Documentation: https://developers.google.com/optimization
- The Bedlam Cube Puzzle: http://www.robspuzzlepage.com/bedlam.htm
- Handbook of Constraint Programming (Rossi, van Beek, Walsh)
- Matplotlib 3D Plotting: https://matplotlib.org/stable/tutorials/toolkits/mplot3d.html

Appendix: Running the Code

To run the Bedlam Cube solver:

```bash
# Install required packages
pip install ortools numpy matplotlib

# Run the solver
python bedlam_cube_solver.py
```

The script will:

1. Generate all piece orientations
2. Build the constraint model
3. Solve the puzzle
4. Display and save the solution visualization

The output includes:

- Progress messages during solving
- Solution details (piece positions and orientations)
- A PNG image of the 3D visualization