Project : Generating and Solving 3D Tetracube Puzzles with Answer Set Programming

Nicolas Melaerts
Umons, Faculté des Sciences, Belgique

Kenza Khemar
Umons, Faculté des Sciences, Belgique

Abstract

This report presents an approach based on *Answer Set Programming* (ASP) to solve the problem of placing tetracubes in a 3D volume. We explore how to model and solve the problem of placing 8 different tetracubes (pieces composed of 4 unit cubes) in a volume of 32 unit cubes. We present three configurations: a $2 \times 4 \times 4$ rectangular prism, a $2 \times 2 \times 8$ rectangular prism, and two regular prisms of size $2 \times 2 \times 4$. Our solution uses the Clingo solver and includes an interactive 3D visualization in Python of the solutions found. We analyze the performance and discuss the challenges encountered in modeling this combinatorial problem.

2012 ACM Subject Classification Theory of Computation → Constraint Satisfaction

Keywords and phrases Answer Set Programming, Tetracubes, 3D Puzzle, Clingo, Comibnatorics

Acknowledgements We want to thank Jef Wijsen

1 Introduction

Puzzles involving the placement of pieces in a given volume constitute a classic challenge in recreational mathematics and computer science. In this report, we focus on the specific problem of placing tetracubes in a 3D volume.

A tetracube is a geometric figure composed of 4 unit cubes connected face-to-face. There are 8 different types of tetracubes: I, T, L, Pyramid, O, N, Z, and Z-mirror. Our goal is to place these 8 tetracubes in a volume of 32 unit cubes, without overlap and completely filling the space.

We explore three volume configurations:

- A rectangular prism of dimensions $2 \times 4 \times 4$
- A rectangular prism of dimensions $2 \times 2 \times 8$
- 2 regular prisms of $2 \times 2 \times 4$

To solve this problem, we use Answer Set Programming (ASP), a declarative programming paradigm particularly suited for combinatorial and constraint satisfaction problems.

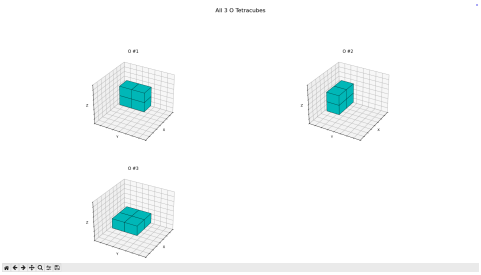
2 Problem Modeling in ASP

2.1 Tetracube Representation

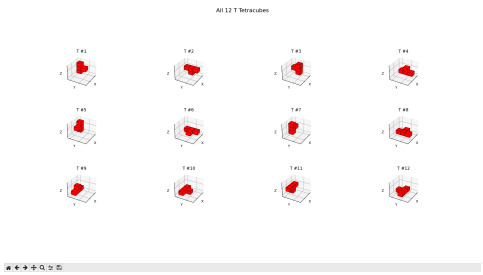
Each tetracube is represented by a set of 4 unit cubes positioned in 3D space. For each type of tetracube, we define all its possible orientations (rotations). For example, the I tetracube can be oriented in 3 different directions (aligned with the X, Y, or Z axis).

The number of unique orientations varies significantly between tetracube types. The O tetracube, with its compact shape, has only 3 unique orientations, while the T tetracube has 12 different orientations due to its asymmetric structure. This variation in the number of orientations affects the complexity of the placement problem.

- The I tetracube has 3 unique orientations
- The L tetracube has 24 unique orientations
- The N tetracube has 12 unique orientations
- The O tetracube has 3 unique orientations
- The Pyramid tetracube has 10 unique orientations
- The T tetracube has 12 unique orientations
- The Z tetracube has 12 unique orientations
- The Z-mirror tetracube has 12 unique orientations



(a) The O tetracube with its 3 unique orientations



(b) The T tetracube with its 12 unique orientations

Figure 1 Examples of tetracubes with their possible orientations

2.2 How to represent tetracubes in ASP

In our ASP model, each tetracube is represented as a collection of 4 unit cubes positioned in a relative coordinate system. We define a predicate `cube(Type, Rotation, DX, DY, DZ)` where:

- `Type` is the tetracube type (I, T, L, Pyramid, O, N, Z, or Z_mirror)
- `Rotation` is a unique identifier for each possible orientation
- `DX, DY, DZ` are the relative coordinates of each unit cube

For example, the O tetracube (shaped like a 2x2x1 rectangle) has 3 unique orientations, corresponding to its alignment with each of the three axes:

Listing 1 ASP representation of the O tetracube

```
% O tetracube
% Rotation 1
cube("O",1,0,0,0).cube("O",1,0,0,1).cube("O",1,0,1,0).cube("O",1,0,1,1).
% Rotation 2
cube("O",2,0,0,0).cube("O",2,0,0,1).cube("O",2,1,0,0).cube("O",2,1,0,1).
% Rotation 3
cube("O",3,0,0,0).cube("O",3,0,1,0).cube("O",3,1,0,0).cube("O",3,1,1,0).
```

Each rotation is defined by specifying the coordinates of the 4 unit cubes that make up the tetracube. The first rotation places the O tetracube in the XY plane, the second in the XZ plane, and the third in the YZ plane.

More complex tetracubes like the L shape have many more possible orientations (24) in total, due to their asymmetric structure. Each orientation must be explicitly defined in our ASP model to allow the solver to consider all possible placements.

This representation allows us to efficiently model the placement of tetracubes in 3D space while ensuring that all geometric constraints are properly enforced.

2.3 Problem Definition

Our ASP model defines:

- The 3D grid (2x4x4, 2x2x8 or two 2x2x4)
- The 8 types of tetracubes and their possible rotations
- Constraints ensuring that:
  - Each tetracube is placed exactly once
  - Tetracubes do not overlap
  - All unit cubes in the grid are occupied
  - Tetracubes remain within the grid boundaries

(a) 2x4x4 configuration **(b)** 2x2x8 configuration **(c)** Two 2x2x4 configurations

Figure 2 Different volume configurations for tetracube placement. Source: `https://puzzler.sourceforge.net/docs/polycubes.html#tetracubes`

2.4 Puzzle Generation

Our system generates 3D tetracube puzzles with varying levels of difficulty by first finding a complete valid arrangement of all 8 tetracubes, then selecting a subset of these tetracubes as "hints" that will be shown to the puzzle solver.

The puzzle generation mechanism is implemented directly in our ASP encoding. We use a choice rule to select exactly `num_hints` tetracubes as hints, and a heuristic directive with a seed value to ensure deterministic selection for reproducibility. This approach allows us to control the difficulty of the puzzle by adjusting the number of pre-placed pieces - fewer hints result in more challenging puzzles.

For each tetracube, we determine whether it should be shown as a hint using the `hint(P)` predicate. The output includes both the complete solution (using `fullPosition` predicates) and the partial puzzle with only hint pieces (using `hintPosition` predicates). This dual output makes it easy to generate both the puzzle and its solution in a single solver run.

The random seed ensures that we can generate different puzzles with the same number of hints, providing variety while maintaining reproducibility. This is particularly useful for creating multiple puzzles of similar difficulty levels.

Our approach guarantees that every generated puzzle has at least one valid solution, as the hints are derived from a complete solution found by the ASP solver. This ensures that all puzzles are solvable, regardless of the difficulty level chosen.

The flexibility of our puzzle generation system allows for creating a wide range of 3D tetracube puzzles across different volume configurations and difficulty levels, making it suitable for both casual puzzlers and serious enthusiasts.

2.5 ASP Encoding

Our ASP encoding for the tetracube puzzle consists of several interconnected components that work together to model the problem constraints and generate solutions. The complete source code is available in Appendix A.

The program begins by importing the tetracube definitions from the `tetracubes.lp` file using the `#include` directive. As described in Section 2.1, these definitions provide all possible orientations for each tetracube type using the predicate `cube(Type, Rotation, DX, DY, DZ)`.

The program supports three different volume configurations, controlled by the `grid_type` constant:

- **Type 1**: A $2 \times 4 \times 4$ rectangular prism (32 unit cubes)
- **Type 2**: A $2 \times 2 \times 8$ rectangular prism (32 unit cubes)
- **Type 3**: Two separate $2 \times 2 \times 4$ prisms (16 unit cubes each)

For each grid type, we define the valid coordinate ranges using conditional facts. For example, for grid type 1, we define `width(0..3)`, `height(0..3)`, and `depth(0..1)`. This approach allows us to use a single program for all three configurations.

For grid type 3 (two separate prisms), we introduce additional predicates to track which tetracubes are assigned to which prism. The predicate `typeGrid(Type, G)` assigns each tetracube type directly to exactly one grid `G`. We also add a constraint to ensure a balanced distribution of 4 tetracubes per prism.

The program defines cells based on the grid type. For grid types 1 and 2, cells are represented as `cell(X,Y,Z)`, while for grid type 3, they are represented as `cell(X,Y,Z,G)` to include the grid identifier.

We define the valid orientations for each tetracube type using the `validOrientation/2` predicate. For example, the `I` tetracube has 3 possible orientations, while the `L` tetracube has 24. We also define the set of all tetracube types using `tetracubeType/1`.

The core of the program uses choice rules to select positions and orientations for each tetracube type directly. For grid types 1 and 2, we use the predicate `position(Type, R, X, Y, Z)`, indicating that a tetracube of type `Type` is placed with rotation `R` at position (X, Y, Z). For grid type 3, we use `position(Type, R, X, Y, Z, G)` to include the grid identifier.

Based on these positions, we compute which cells are occupied by each tetracube using the `occupied/4` predicate (or `occupied/5` for grid type 3). This predicate takes into account the relative coordinates of each unit cube within the tetracube.

The program includes several constraints to ensure valid solutions:

- Tetracubes must stay within the grid boundaries.
- Tetracubes cannot overlap (no cell can be occupied by more than one tetracube).
- All cells in the grid must be occupied by exactly one tetracube.

For puzzle generation, we introduce a mechanism to select a subset of tetracubes as "hints" that will be shown to the solver. The number of hints is controlled by the `num_hints` constant. We use a choice rule `{ hint(Type) : tetracubeType(Type) } num_hints` to select exactly `num_hints` tetracubes, and a heuristic directive with a seed value to ensure deterministic selection for reproducibility.

The output predicates are designed to show both the complete solution (`fullPosition`) and the partial puzzle with only the hint pieces visible (`hintPosition`). This makes it easy to generate and visualize both the puzzle and its solution. Additional output predicates show which pieces are hints and, for grid type 3, the grid assignments.

This direct approach, which uses tetracube types rather than numeric IDs, provides a more intuitive and readable encoding. It eliminates the need for an intermediate mapping between IDs and types, resulting in a more streamlined representation of the problem. The encoding provides a flexible framework for generating and solving tetracube puzzles across different volume configurations and difficulty levels.

The complete source code for all components is available in Appendix A.

3 Implementation

3.1 Project Structure

Our project consists of the following components:

- `tetracubes.lp`: Contains the definitions of all tetracube types and their possible orientations.
- `PUZZLE.lp`: The main ASP program that encodes the puzzle constraints and generation logic.
- `place_tetracubes.py`: A Python script that shows the puzzle and the solution for the visualization.
- `draw_tetracubes.py`: A visualization script that renders the 3D tetracube configurations using matplotlib.
- `solutions.txt`: A file containing the solver output, which includes the complete solution and the puzzle with only hint pieces.

3.2 Solving with Clingo

We use Clingo, a modern ASP solver, to find solutions to our tetracube puzzle and generate new puzzles. The general command structure for running our program is:

```
clingo PUZZLE.lp -c grid_type=X -c num_hints=Y --seed=Z > solution.txt
```

Where:

- `grid_type` specifies the volume configuration (1, 2, or 3)
- `num_hints` controls how many tetracubes are shown as hints
- `seed` provides a random seed for reproducible hint selection

For example, to generate a puzzle using the $2 \times 4 \times 4$ configuration with 6 hint pieces:

```
clingo PUZZLE.lp -c grid_type=1 -c num_hints=6 --seed=42 > solution.txt
```

For the $2 \times 2 \times 8$ configuration with 3 hint pieces:

```
clingo PUZZLE.lp -c grid_type=2 -c num_hints=3 --seed=123 > solution.txt
```

For the two separate $2 \times 2 \times 4$ cubes with 4 hint pieces:

```
clingo PUZZLE.lp -c grid_type=3 -c num_hints=4 --seed=42 > solution.txt
```

We can also generate puzzles with no hints at all, creating a completely blank starting configuration:

```
clingo PUZZLE.lp -c grid_type=1 -c num_hints=0 --seed=42 > solution.txt
```

The flexibility of our approach allows us to create puzzles of varying difficulty levels. Generally, fewer hints result in more challenging puzzles. The output is redirected to a solution.txt file, which contains both the complete solution and the partial puzzle with only the hint pieces visible.

Our Python scripts can then process this output to visualize both the puzzle and its solution, providing a complete workflow from puzzle generation to visualization.

By adjusting the grid type and number of hints, we can generate a wide variety of 3D tetracube puzzles, each with a unique configuration and difficulty level, while ensuring that every puzzle has at least one valid solution.

3.3 Interpreting Clingo Output

When we run our ASP program with Clingo, it produces output like the following (for grid type 3 with 6 hint pieces):

```
clingo version 5.7.1
Reading from PUZZLE.lp
Solving...
Answer: 1
hint("I") hint("T") hint("L") hint("Pyramid") hint("O") hint("N")
fullPosition("I",2,1,0,0) fullPosition("T",4,2,1,0) fullPosition("L",16,0,1,1)
fullPosition("Pyramid",6,2,2,0) fullPosition("O",2,2,0,0) fullPosition("N",2,0,1,0)
fullPosition("Z",8,2,1,0) fullPosition("Z_mirror",1,0,0,0)
hintPosition("I",2,1,0,0) hintPosition("T",4,2,1,0) hintPosition("L",16,0,1,1)
hintPosition("Pyramid",6,2,2,0) hintPosition("O",2,2,0,0) hintPosition("N",2,0,1,0)

SATISFIABLE
```

This output contains several key pieces of information:

- `hint(Type)`: Indicates which tetracube types are selected as hints
- `fullPosition(Type, R, X, Y, Z)`: Shows the complete solution with each tetracube type `Type` placed at position (X, Y, Z) with rotation `R`
- `hintPosition(Type, R, X, Y, Z)`: Shows the partial puzzle with only the hint pieces visible, positioned similarly

In this example, the hints are tetracube types “I”, “T”, “L”, “Pyramid”, “O”, and “N”. The remaining pieces (“Z” and “Z_mirror”) must be placed by the puzzle solver.

3.4 Solution Visualisation

To visualize the puzzle and its solution, we use a Python script that processes the Clingo output:

```
python place_tetracubes.py solution.txt
```

This script offers two visualization modes:

- **Puzzle mode**: Displays only the hint pieces, showing the initial configuration that a human solver would start with. This allows the solver to see which pieces are already fixed and which ones need to be placed.

- **Solution mode**: Shows the complete solution with all tetracubes properly placed in their respective positions, filling the entire volume(s).

The visualization script uses matplotlib to create 3D renderings of the tetracubes, with different colors for each tetracube type. For grid type 3 (two separate prisms), both prisms are displayed side by side, making it easy to see how the tetracubes are distributed between them.

This visualization capability is crucial for both puzzle design and solving, as it transforms the abstract ASP solution into a tangible 3D representation that can be easily understood and manipulated.

4 Results and Analysis

4.1 Complete Solutions Found

Our ASP-based approach successfully found solutions for all three volume configurations. For each configuration, Clingo was able to find at least one stable model, confirming that it is possible to pack all 8 tetracubes into the given volumes without gaps or overlaps.

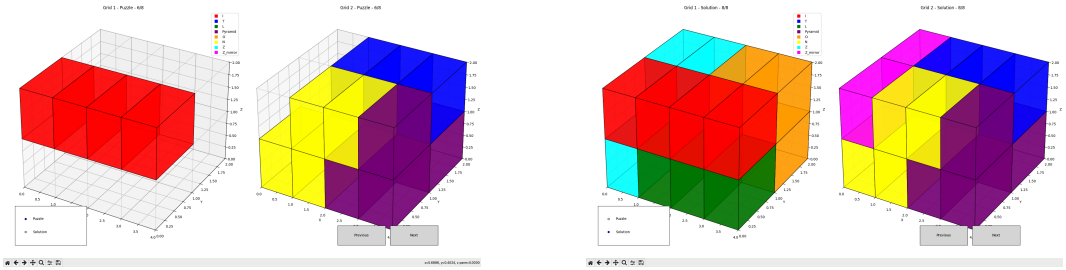
4.1.1 Two Separate $2 \times 2 \times 4$ Prisms (Grid Type 3)

Let's examine the solution found for the two separate $2 \times 2 \times 4$ prisms (grid type 3) that was presented in the "Interpreting Clingo Output" section. This solution was generated using the command:

```
clingo PUZZLE.lp -c grid_type=3 -c num_hints=4 --seed=42 > solution.txt
```

In this solution:

- Grid 1 contains the I tetracube (piece 1), L tetracube (piece 3), O tetracube (piece 5), and Z tetracube (piece 7)
- Grid 2 contains the T tetracube (piece 2), Pyramid tetracube (piece 4), N tetracube (piece 6), and Z_mirror tetracube (piece 8)



(a) Puzzle with 4 hint pieces **(b)** Complete solution with all 8 tetracubes

Figure 3 Visualization of a puzzle and its solution for the two separate $2 \times 2 \times 4$ prisms

Figure 3 shows the visualization of this solution. On the left, we see the puzzle with only the 4 hint pieces (I, T, Pyramid, and N tetracubes) displayed. On the right, we see the complete solution with all 8 tetracubes properly placed in their respective prisms.

4.1.2 $2 \times 4 \times 4$ Rectangular Prism (Grid Type 1)

For the $2 \times 4 \times 4$ rectangular prism (grid type 1), we generated a puzzle with 5 hint pieces using the command:

```
clingo PUZZLE.lp -c grid_type=1 -c num_hints=5 --seed=123 > solution.txt
```

(a) Puzzle with 5 hint pieces

(b) Complete solution with all 8 tetracubes

Figure 4 Visualization of a puzzle and its solution for the $2 \times 4 \times 4$ rectangular prism

Figure 4 shows the visualization of this solution. The puzzle (left) shows 5 hint pieces, while the complete solution (right) shows all 8 tetracubes fitting perfectly within the $2 \times 4 \times 4$ volume.

4.1.3 $2 \times 2 \times 8$ Rectangular Prism (Grid Type 2)

For the $2 \times 2 \times 8$ rectangular prism (grid type 2), we generated a more challenging puzzle with only 3 hint pieces using the command:

```
clingo PUZZLE.lp -c grid_type=2 -c num_hints=3 --seed=456 > solution.txt
```

(a) Puzzle with 3 hint pieces

(b) Complete solution with all 8 tetracubes

Figure 5 Visualization of a puzzle and its solution for the $2 \times 2 \times 8$ rectangular prism

Figure 5 shows the visualization of this solution. With only 3 hint pieces (left), this puzzle presents a more significant challenge, requiring the solver to place 5 additional tetracubes to complete the solution (right).

These examples demonstrate the flexibility of our approach in generating puzzles of varying difficulty across different volume configurations. In all cases, our ASP encoding successfully found valid solutions that completely fill the volumes without gaps or overlaps.

5 Comparative Performance Analysis of Clingo

5.1 Experimental Context

In this study, we compared the performance of two different implementations of a tetracube placement puzzle using the Clingo constraint programming tool. The two implementations differ in their complexity and the size of the problem to be solved:

- **PUZZLE.lp**: Basic version using 32 cells and 8 tetracubes.
- **PUZZLE_COMPLEX.lp**: More complex version using 64 cells and 16 tetracubes.

Both programs were executed with the same parameters:

```
-c grid_type=3 -c num_hints=3 --seed=42
```

Each program was run 20 times to obtain statistically significant measurements.

5.2 Results

Figure 6 presents the results of our comparative analysis:

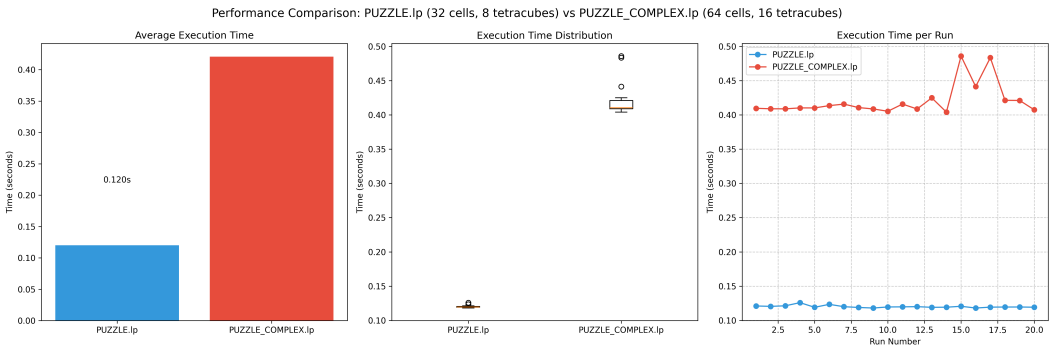


Figure 6 Performance comparison between PUZZLE.lp (32 cells, 8 tetracubes) and PUZZLE_COMPLEX.lp (64 cells, 16 tetracubes). The left graph shows the average execution time, the middle one presents the distribution of execution times as a box plot, and the right one traces the evolution of execution times for each trial.

5.3 Analysis

The results clearly show that the more complex version (PUZZLE_COMPLEX.lp) requires significantly more computation time than the basic version (PUZZLE.lp). This difference can be explained primarily by:

Increased search space: Doubling the number of cells (from 32 to 64) and tetracubes (from 8 to 16) leads to an exponential increase in the search space.

6 Conclusion

In this project, we developed an Answer Set Programming approach to solve the challenging problem of placing 8 different tetracubes in a 3D volume of 32 unit cubes. Our implementation demonstrates the effectiveness of ASP for modeling and solving complex combinatorial problems with geometric constraints.

We explored three different volume configurations 2x4x4, 2x2x8, and two 2x2x4 prisms and created a flexible puzzle generation system capable of producing challenges of varying difficulty by fixing different numbers of pieces. A key aspect of our work was not just solving the placement problem, but also generating engaging puzzles that can be presented to human solvers with varying levels of complexity. The interactive 3D visualization tool we developed enhances understanding of the solutions and makes the puzzle more accessible.

This work showcases how declarative programming paradigms like ASP can elegantly handle complex spatial reasoning tasks. Future work could explore extensions to other volume shapes, additional geometric constraints, or optimization techniques to improve solving performance for larger instances of similar puzzles.

Code Availability

The complete source code for this project, including all ASP encodings and Python scripts is available on GitHub at the following address:

`https://github.com/NicolasMelaerts/KRR-Tetracubes-Puzzles.git`

This repository contains everything needed to reproduce the results, generate new puzzles, and visualize solutions as described in this report.

A Complete Source Code

A.1 Tetracube Definitions

Listing 2: `tetracubes.lp`

Listing 2 ASP definitions for all tetracubes

```
% I tetracube
% Rotation 1
cube("I",1,0,0,0).cube("I",1,0,0,1).cube("I",1,0,0,2).cube("I",1,0,0,3).
% Rotation 2
cube("I",2,0,0,0).cube("I",2,0,1,0).cube("I",2,0,2,0).cube("I",2,0,3,0).
% Rotation 3
cube("I",3,0,0,0).cube("I",3,1,0,0).cube("I",3,2,0,0).cube("I",3,3,0,0).
% T tetracube
% Rotation 1
cube("T",1,0,0,0).cube("T",1,0,0,1).cube("T",1,0,0,2).cube("T",1,0,1,1).
% Rotation 2
cube("T",2,0,0,1).cube("T",2,0,1,0).cube("T",2,0,1,1).cube("T",2,0,2,1).
% Rotation 3
cube("T",3,0,0,1).cube("T",3,0,1,0).cube("T",3,0,1,1).cube("T",3,0,1,2).
% Rotation 4
cube("T",4,0,0,0).cube("T",4,0,1,0).cube("T",4,0,1,1).cube("T",4,0,2,0).
% Rotation 5
cube("T",5,0,0,0).cube("T",5,0,0,1).cube("T",5,0,0,2).cube("T",5,1,0,1).
% Rotation 6
cube("T",6,0,0,0).cube("T",6,0,1,0).cube("T",6,0,2,0).cube("T",6,1,1,0).
% Rotation 7
cube("T",7,0,0,1).cube("T",7,1,0,0).cube("T",7,1,0,1).cube("T",7,1,0,2).
% Rotation 8
cube("T",8,0,1,0).cube("T",8,1,0,0).cube("T",8,1,1,0).cube("T",8,1,2,0).
% Rotation 9
cube("T",9,0,0,0).cube("T",9,1,0,0).cube("T",9,1,0,1).cube("T",9,2,0,0).
% Rotation 10
cube("T",10,0,0,0).cube("T",10,1,0,0).cube("T",10,1,1,0).cube("T",10,2,0,0).
% Rotation 11
cube("T",11,0,0,1).cube("T",11,1,0,0).cube("T",11,1,0,1).cube("T",11,2,0,1).
```

```
% Rotation 12
cube("T",12,0,1,0).cube("T",12,1,0,0).cube("T",12,1,1,0).cube("T",12,2,1,0).
% L tetracube
% Rotation 1
cube("L",1,0,0,0).cube("L",1,0,0,1).cube("L",1,0,0,2).cube("L",1,0,1,0).
% Rotation 2
cube("L",2,0,0,0).cube("L",2,0,0,1).cube("L",2,0,1,1).cube("L",2,0,2,1).
% Rotation 3
cube("L",3,0,0,2).cube("L",3,0,1,0).cube("L",3,0,1,1).cube("L",3,0,1,2).
% Rotation 4
cube("L",4,0,0,0).cube("L",4,0,1,0).cube("L",4,0,2,0).cube("L",4,0,2,1).
% Rotation 5
cube("L",5,0,0,0).cube("L",5,0,0,1).cube("L",5,0,0,2).cube("L",5,0,1,2).
% Rotation 6
cube("L",6,0,0,1).cube("L",6,0,1,1).cube("L",6,0,2,0).cube("L",6,0,2,1).
% Rotation 7
cube("L",7,0,0,0).cube("L",7,0,1,0).cube("L",7,0,1,1).cube("L",7,0,1,2).
% Rotation 8
cube("L",8,0,0,0).cube("L",8,0,0,1).cube("L",8,0,1,0).cube("L",8,0,2,0).
% Rotation 9
cube("L",9,0,0,0).cube("L",9,0,0,1).cube("L",9,0,0,2).cube("L",9,1,0,2).
% Rotation 10
cube("L",10,0,0,0).cube("L",10,0,1,0).cube("L",10,0,2,0).cube("L",10,1,2,0).
% Rotation 11
cube("L",11,0,0,0).cube("L",11,0,0,1).cube("L",11,0,0,2).cube("L",11,1,0,0).
% Rotation 12
cube("L",12,0,0,0).cube("L",12,0,1,0).cube("L",12,0,2,0).cube("L",12,1,0,0).
% Rotation 13
cube("L",13,0,0,0).cube("L",13,1,0,0).cube("L",13,1,0,1).cube("L",13,1,0,2).
% Rotation 14
cube("L",14,0,0,0).cube("L",14,1,0,0).cube("L",14,1,1,0).cube("L",14,1,2,0).
% Rotation 15
cube("L",15,0,0,2).cube("L",15,1,0,0).cube("L",15,1,0,1).cube("L",15,1,0,2).
% Rotation 16
cube("L",16,0,2,0).cube("L",16,1,0,0).cube("L",16,1,1,0).cube("L",16,1,2,0).
% Rotation 17
cube("L",17,0,0,0).cube("L",17,0,0,1).cube("L",17,1,0,0).cube("L",17,2,0,0).
% Rotation 18
cube("L",18,0,0,0).cube("L",18,0,1,0).cube("L",18,1,0,0).cube("L",18,2,0,0).
% Rotation 19
cube("L",19,0,0,0).cube("L",19,0,0,1).cube("L",19,1,0,1).cube("L",19,2,0,1).
% Rotation 20
cube("L",20,0,0,0).cube("L",20,0,1,0).cube("L",20,1,1,0).cube("L",20,2,1,0).
% Rotation 21
cube("L",21,0,0,1).cube("L",21,1,0,1).cube("L",21,2,0,0).cube("L",21,2,0,1).
% Rotation 22
cube("L",22,0,1,0).cube("L",22,1,1,0).cube("L",22,2,0,0).cube("L",22,2,1,0).
% Rotation 23
cube("L",23,0,0,0).cube("L",23,1,0,0).cube("L",23,2,0,0).cube("L",23,2,0,1).
% Rotation 24
cube("L",24,0,0,0).cube("L",24,1,0,0).cube("L",24,2,0,0).cube("L",24,2,1,0).
% Pyramid tetracube
% Rotation 1
cube("Pyramid",1,0,0,0).cube("Pyramid",1,0,0,1).cube("Pyramid",1,0,1,0).cube("Pyramid",1,1,0,0).
% Rotation 2
cube("Pyramid",2,0,0,0).cube("Pyramid",2,0,0,1).cube("Pyramid",2,0,1,1).cube("Pyramid",2,1,0,1).
% Rotation 3
cube("Pyramid",3,0,0,1).cube("Pyramid",3,0,1,0).cube("Pyramid",3,0,1,1).cube("Pyramid",3,1,1,1).
% Rotation 4
cube("Pyramid",4,0,0,0).cube("Pyramid",4,0,1,0).cube("Pyramid",4,0,1,1).cube("Pyramid",4,1,1,0).
% Rotation 5
cube("Pyramid",5,0,0,1).cube("Pyramid",5,1,0,0).cube("Pyramid",5,1,0,1).cube("Pyramid",5,1,1,1).
% Rotation 6
cube("Pyramid",6,0,1,1).cube("Pyramid",6,1,0,1).cube("Pyramid",6,1,1,0).cube("Pyramid",6,1,1,1).
% Rotation 7
cube("Pyramid",7,0,1,0).cube("Pyramid",7,1,0,0).cube("Pyramid",7,1,1,0).cube("Pyramid",7,1,1,1).
```

```
% Rotation 8
cube("Pyramid",8,0,0,0).cube("Pyramid",8,1,0,0).cube("Pyramid",8,1,0,1).cube("Pyramid",8,1,1,0).
% O tetracube
% Rotation 1
cube("O",1,0,0,0).cube("O",1,0,0,1).cube("O",1,0,1,0).cube("O",1,0,1,1).
% Rotation 2
cube("O",2,0,0,0).cube("O",2,0,0,1).cube("O",2,1,0,0).cube("O",2,1,0,1).
% Rotation 3
cube("O",3,0,0,0).cube("O",3,0,1,0).cube("O",3,1,0,0).cube("O",3,1,1,0).
% N tetracube
% Rotation 1
cube("N",1,0,0,0).cube("N",1,0,0,1).cube("N",1,0,1,1).cube("N",1,0,1,2).
% Rotation 2
cube("N",2,0,0,1).cube("N",2,0,1,0).cube("N",2,0,1,1).cube("N",2,0,2,0).
% Rotation 3
cube("N",3,0,0,1).cube("N",3,0,0,2).cube("N",3,0,1,0).cube("N",3,0,1,1).
% Rotation 4
cube("N",4,0,0,0).cube("N",4,0,1,0).cube("N",4,0,1,1).cube("N",4,0,2,1).
% Rotation 5
cube("N",5,0,0,1).cube("N",5,0,0,2).cube("N",5,1,0,0).cube("N",5,1,0,1).
% Rotation 6
cube("N",6,0,1,0).cube("N",6,0,2,0).cube("N",6,1,0,0).cube("N",6,1,1,0).
% Rotation 7
cube("N",7,0,0,0).cube("N",7,0,0,1).cube("N",7,1,0,1).cube("N",7,1,0,2).
% Rotation 8
cube("N",8,0,0,0).cube("N",8,0,1,0).cube("N",8,1,1,0).cube("N",8,1,2,0).
% Rotation 9
cube("N",9,0,0,0).cube("N",9,1,0,0).cube("N",9,1,0,1).cube("N",9,2,0,1).
% Rotation 10
cube("N",10,0,0,0).cube("N",10,1,0,0).cube("N",10,1,1,0).cube("N",10,2,1,0).
% Rotation 11
cube("N",11,0,0,1).cube("N",11,1,0,0).cube("N",11,1,0,1).cube("N",11,2,0,0).
% Rotation 12
cube("N",12,0,1,0).cube("N",12,1,0,0).cube("N",12,1,1,0).cube("N",12,2,0,0).
% Z tetracube
% Rotation 1
cube("Z",1,0,0,0).cube("Z",1,0,0,1).cube("Z",1,0,1,0).cube("Z",1,1,1,0).
% Rotation 2
cube("Z",2,0,0,0).cube("Z",2,0,0,1).cube("Z",2,0,1,1).cube("Z",2,1,0,0).
% Rotation 3
cube("Z",3,0,0,1).cube("Z",3,0,1,0).cube("Z",3,0,1,1).cube("Z",3,1,0,1).
% Rotation 4
cube("Z",4,0,0,0).cube("Z",4,0,1,0).cube("Z",4,0,1,1).cube("Z",4,1,1,1).
% Rotation 5
cube("Z",5,0,1,1).cube("Z",5,1,0,0).cube("Z",5,1,0,1).cube("Z",5,1,1,1).
% Rotation 6
cube("Z",6,0,1,0).cube("Z",6,1,0,1).cube("Z",6,1,1,0).cube("Z",6,1,1,1).
% Rotation 7
cube("Z",7,0,0,0).cube("Z",7,1,0,0).cube("Z",7,1,1,0).cube("Z",7,1,1,1).
% Rotation 8
cube("Z",8,0,0,1).cube("Z",8,1,0,0).cube("Z",8,1,0,1).cube("Z",8,1,1,0).
% Rotation 9
cube("Z",9,0,0,0).cube("Z",9,0,0,1).cube("Z",9,1,0,1).cube("Z",9,1,1,1).
% Rotation 10
cube("Z",10,0,0,1).cube("Z",10,0,1,1).cube("Z",10,1,1,0).cube("Z",10,1,1,1).
% Rotation 11
cube("Z",11,0,1,0).cube("Z",11,0,1,1).cube("Z",11,1,0,0).cube("Z",11,1,1,0).
% Rotation 12
cube("Z",12,0,0,0).cube("Z",12,0,1,0).cube("Z",12,1,0,0).cube("Z",12,1,0,1).
% Z_mirror tetracube
% Rotation 1
cube("Z_mirror",1,0,0,0).cube("Z_mirror",1,0,0,1).cube("Z_mirror",1,0,1,0).cube("Z_mirror",1,1,0,1).
% Rotation 2
cube("Z_mirror",2,0,0,0).cube("Z_mirror",2,0,0,1).cube("Z_mirror",2,0,1,1).cube("Z_mirror",2,1,1,1).
% Rotation 3
cube("Z_mirror",3,0,0,1).cube("Z_mirror",3,0,1,0).cube("Z_mirror",3,0,1,1).cube("Z_mirror",3,1,1,0).
```

```
% Rotation 4
cube("Z_mirror",4,0,0,0).cube("Z_mirror",4,0,1,0).cube("Z_mirror",4,0,1,1).cube("Z_mirror",4,1,0,0).
% Rotation 5
cube("Z_mirror",5,0,0,0).cube("Z_mirror",5,1,0,0).cube("Z_mirror",5,1,0,1).cube("Z_mirror",5,1,1,1).
% Rotation 6
cube("Z_mirror",6,0,0,1).cube("Z_mirror",6,1,0,1).cube("Z_mirror",6,1,1,0).cube("Z_mirror",6,1,1,1).
% Rotation 7
cube("Z_mirror",7,0,1,1).cube("Z_mirror",7,1,0,0).cube("Z_mirror",7,1,1,0).cube("Z_mirror",7,1,1,1).
% Rotation 8
cube("Z_mirror",8,0,1,0).cube("Z_mirror",8,1,0,0).cube("Z_mirror",8,1,0,1).cube("Z_mirror",8,1,1,0).
% Rotation 9
cube("Z_mirror",9,0,0,0).cube("Z_mirror",9,0,0,1).cube("Z_mirror",9,1,0,0).cube("Z_mirror",9,1,1,0).
% Rotation 10
cube("Z_mirror",10,0,0,1).cube("Z_mirror",10,0,1,1).cube("Z_mirror",10,1,0,0).cube("Z_mirror",10,1,0,1).
% Rotation 11
cube("Z_mirror",11,0,1,0).cube("Z_mirror",11,0,1,1).cube("Z_mirror",11,1,0,1).cube("Z_mirror",11,1,1,1).
% Rotation 12
cube("Z_mirror",12,0,0,0).cube("Z_mirror",12,0,1,0).cube("Z_mirror",12,1,1,0).cube("Z_mirror",12,1,1,1).
```

A.2 ASP Model for the Puzzle Generation

Listing 3: PUZZLE.lp

Listing 3 ASP model definition for puzzle generation

```
% Include tetracube definitions
#include "tetracubes.lp".

% Grid type selection
#const grid_type = 1. % 1: 2x4x4, 2: 2x2x8, 3: 2x2x4 (two separate grids)
#const num_hints = 6.  % Number of pieces to show as hints
#const seed = 42.      % Seed for random selection

% Definition of grids based on grid_type
width(0..3) :- grid_type == 1.
height(0..3) :- grid_type == 1.
depth(0..1) :- grid_type == 1.

width(0..7) :- grid_type == 2.
height(0..1) :- grid_type == 2.
depth(0..1) :- grid_type == 2.

width(0..3) :- grid_type == 3.
height(0..1) :- grid_type == 3.
depth(0..1) :- grid_type == 3.

% For grid_type 3, we need to track which grid a piece is placed in
grid(1..2) :- grid_type == 3.
1 { typeGrid(Type, G) : grid(G) } 1 :- tetracubeType(Type), grid_type == 3.

% Ensure balanced distribution for grid_type 3 (4 pieces per grid)
:- grid_type == 3, grid(G), #count { Type : typeGrid(Type, G) } != 4.

% Define cells based on grid type
cell(X,Y,Z) :- width(X), height(Y), depth(Z), grid_type != 3.
cell(X,Y,Z,G) :- width(X), height(Y), depth(Z), grid(G), grid_type == 3.

% Constants
#const total_cells = 32.

% Tetracube types and their valid orientations
validOrientation("I", 1..3).
validOrientation("T", 1..12).
validOrientation("L", 1..24).
validOrientation("Pyramid", 1..8).
```

```
validOrientation("O", 1..3).
validOrientation("N", 1..12).
validOrientation("Z", 1..12).
validOrientation("Z_mirror", 1..12).

% Tetracube types
tetracubeType("I").
tetracubeType("T").
tetracubeType("L").
tetracubeType("Pyramid").
tetracubeType("O").
tetracubeType("N").
tetracubeType("Z").
tetracubeType("Z_mirror").

% Choosing the position and rotation for each tetracube
% For grid_type 1 and 2
1 { position(Type, R, X, Y, Z) : width(X), height(Y), depth(Z), validOrientation(Type, R) } 1 :-
    tetracubeType(Type), grid_type != 3.

% For grid_type 3 (two separate grids)
1 { position(Type, R, X, Y, Z, G) : width(X), height(Y), depth(Z), validOrientation(Type, R) } 1 :-
    tetracubeType(Type), typeGrid(Type, G), grid_type == 3.

% Placing tetracubes on the grid
% For grid_type 1 and 2
occupied(Type, X+DX, Y+DY, Z+DZ) :- position(Type, R, X, Y, Z),
                                    cube(Type, R, DX, DY, DZ), grid_type != 3.

% For grid_type 3
occupied(Type, X+DX, Y+DY, Z+DZ, G) :- position(Type, R, X, Y, Z, G),
                                       cube(Type, R, DX, DY, DZ), grid_type == 3.

% Ensure tetracubes stay within grid limits
% For grid_type 1 and 2
:- position(Type, R, X, Y, Z), cube(Type, R, DX, DY, DZ),
   not width(X+DX), grid_type != 3.
:- position(Type, R, X, Y, Z), cube(Type, R, DX, DY, DZ),
   not height(Y+DY), grid_type != 3.
:- position(Type, R, X, Y, Z), cube(Type, R, DX, DY, DZ),
   not depth(Z+DZ), grid_type != 3.

% For grid_type 3
:- position(Type, R, X, Y, Z, G), cube(Type, R, DX, DY, DZ),
   not width(X+DX), grid_type == 3.
:- position(Type, R, X, Y, Z, G), cube(Type, R, DX, DY, DZ),
   not height(Y+DY), grid_type == 3.
:- position(Type, R, X, Y, Z, G), cube(Type, R, DX, DY, DZ),
   not depth(Z+DZ), grid_type == 3.

% Prevent overlapping tetracubes
% For grid_type 1 and 2
:- occupied(Type1, X, Y, Z), occupied(Type2, X, Y, Z), Type1 != Type2, grid_type != 3.

% For grid_type 3
:- occupied(Type1, X, Y, Z, G), occupied(Type2, X, Y, Z, G), Type1 != Type2, grid_type == 3.

% Ensure all cells are occupied
% For grid_type 1 and 2
cellOccupied(X, Y, Z) :- occupied(Type, X, Y, Z), tetracubeType(Type), grid_type != 3.
:- cell(X, Y, Z), not cellOccupied(X, Y, Z), grid_type != 3.

% For grid_type 3
cellOccupied(X, Y, Z, G) :- occupied(Type, X, Y, Z, G), tetracubeType(Type), grid_type == 3.
:- cell(X, Y, Z, G), not cellOccupied(X, Y, Z, G), grid_type == 3.
```

```
% Select exactly num_hints pieces as hints
{ hint(Type) : tetracubeType(Type) } num_hints.
:- not num_hints = #count { Type : hint(Type) }.

% Use seed for random selection
#heuristic hint(Type) : tetracubeType(Type). [seed@3,false]

% Show all piece assignments for solution
#show fullPosition(Type,R,X,Y,Z) : position(Type,R,X,Y,Z), grid_type != 3.
#show fullPosition(Type,R,X,Y,Z,G) : position(Type,R,X,Y,Z,G), grid_type == 3.

% Show positions only for hint pieces (puzzle)
#show hintPosition(Type,R,X,Y,Z) : position(Type,R,X,Y,Z), hint(Type), grid_type != 3.
#show hintPosition(Type,R,X,Y,Z,G) : position(Type,R,X,Y,Z,G), hint(Type), grid_type == 3.

% Show which pieces are hints
#show hint/1.

% Show grid assignments only for grid_type 3
#show typeGrid(Type,G) : typeGrid(Type,G), grid_type == 3.
```