Vrije Universiteit Amsterdam

Bachelor Thesis

Solving Shape-Fitting Puzzle with SAT

Author: Charanan Mahakijpalach (2750155)

1st supervisor: Joerg Endrullis
2nd reader: Femke van Raamsdonk

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

June 27, 2025

Abstract

This thesis explores the use of a SAT solver to solve shape-fitting puzzles by encoding them as Boolean satisfiability problems. Users define an arbitrary board size and a list of shapes, which remain fixed throughout the solving process. Each valid placement of a shape is represented as a unique propositional variable, and the puzzle's constraints related to the shapes' positions are translated into a conjunctive normal form (CNF) formula. This formula is then solved using the MiniSat solver through the PySAT library. The implementation identifies valid shape arrangements and determines whether a solution exists, successfully handling puzzles of small to moderate complexity. This work demonstrates the effectiveness of SAT-based techniques in solving spatial logic problems.

1 Introduction

A shape-fitting puzzle is a tiling puzzle in which a certain number of shapes need to be assembled into a larger given shape without overlaps. This larger given shape is referred to as the board, which is a quadrilateral of arbitrary size. The puzzle does not include gravity, meaning shapes do not fall when being placed. The shapes can range from one unit to as many units as the board size and are primarily considered in a single orientation. Rotated versions of the shapes are included as an additional extension. The board does not need to be completely filled, but all shapes must be placed on the board; thus, the total units of shapes should not exceed the board size. An example of the puzzle is shown in Figure 1, where a set of five shapes is arranged on a 5×4 board.

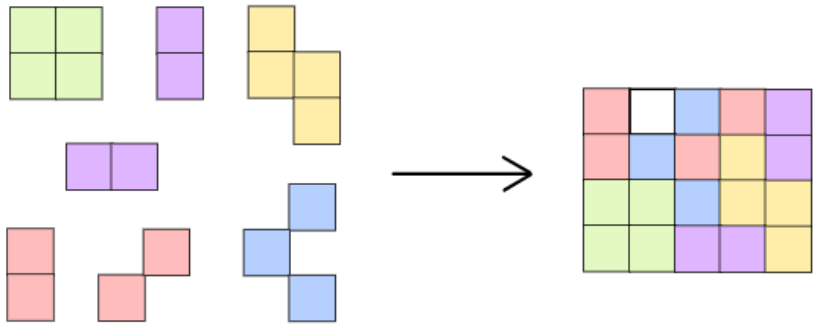


Figure 1: *An example of five shapes on a* 5×4 *board*

The goal of the shape-fitting puzzle is to find a suitable position for each shape such that no two shapes occupy the same cell on the board. This thesis aims to accomplish the goal using a *satisfiability solver* program, also known as a SAT solver. Components of the puzzle, such as the board, the shapes, and the placement of each shape, are encoded into a propositional logic formula. The formula is considered *satisfiable* (SAT) if the SAT solver can find at least one satisfying assignment, returning the valuations of the formula that corresponds to the right position of each shape.

The shape-fitting puzzle bears similarity to *polyomino puzzles*, which involve placing specific shapes—composed of square units—on a board without overlap. However, while polyominoes are limited in how these units are connected, following specific geometric patterns [6], the shape-fitting puzzle permits arbitrary user-defined shapes without restrictions on connectivity, offering greater flexibility in shape construction. To solve such combinatorial puzzles efficiently, SAT solvers like *MiniSat*—which determine the satisfiability of Boolean formulas in *conjunctive normal form* (CNF)—have shown strong performance across domains such as puzzle solving, DFA identification, and program verification [4][5][7]. Their key strength lies in their speed and scalability, especially when problems are encoded compactly into CNF [8], with modern solvers even supporting parallelism across subproblems [2]. Although *Satisfiability Modulo Theories* (SMT) solvers such as Z3 and CVC5 offer more expressive power by incorporating background theories like integer arithmetic or arrays [1][3], SAT solvers often remain more efficient when the problem structure suits pure Boolean logic. In this work, we adopt a SAT-based approach and solve the encoded puzzle using MiniSat, leveraging its efficiency while aligning with current trends in logic-based modeling [1][4].

This thesis begins by illustrating how the three core elements of the shape-fitting puzzle are mathematically represented. It then explains how two key constraints of the puzzle—using every shape exactly once and preventing overlaps—are encoded into a CNF propositional formula. To clarify the encoding process, two puzzle examples are presented: one with a valid solution and one without. The implementation section explains how the mathematical concepts are translated into Python code, including an extended version that supports rotated shapes. The thesis concludes with key takeaways and discusses potential improvements for future work.

2 Representing The Shape-Fitting Puzzle

This section will present each element of the shape-fitting puzzle—including the board, the shapes, and the positions of each shape—through both mathematical descriptions and visual illustrations.

2.1 The Board

The board is a 2-dimensional grid $W \times H$, which is user-defined. It can be treated as a 2D list or a list of lists; however, for ease of SAT encoding later on, the board is represented as a set of coordinate pairs.

Definition 2.1.1. Let $W, H \in \mathbf{N}$ be the dimensions of the board. We define the board as follows:

$$\boldsymbol{Board} \subseteq \mathbf{N} \times \mathbf{N},\ \boldsymbol{Board} = \{(x, y) \in \mathbf{N}^2 \mid 0 \leq x < W,\ 0 \leq y < H\}$$

- W is the width of the board, and x denotes the column index from left to right.
- H is the height of the board, and y denotes the row index from top to bottom.

This means that the board size must be at least 1×1, but there is no requirement for W and H to be equal. The number of coordinates is equal to the number of board units, and all coordinates are unique under the condition in Definition 2.1.1. An example board is depicted in Figure 2.

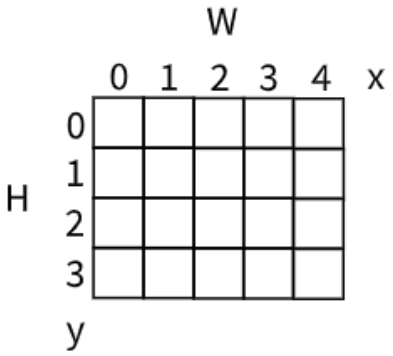


Figure 2: *A 5×4 board*

2.2 The Shapes

Each shape in the shape-fitting puzzles is represented similarly to the board, which is a set of coordinates (x, y) given by the user. These coordinates do not indicate the exact position of the shape on the board. They represent how the shape looks. In other words, x represents the horizontal structure of the shape (left to right), while y represents the vertical structure (top to bottom). For example, $\{(0,0),(0,1),(1,0)\}$ and $\{(2,3),(2,4),(3,3)\}$ both illustrate the shape below.

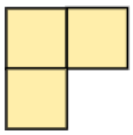


Figure 3: *A sample shape*

A shape can contain any coordinate. In fact, the values of x and y can exceed W and H, respectively, as long as the differences between the smallest values x_{min}, y_{min} and the largest values x_{max}, y_{max} of the shape do not exceed W and H; in other words, the shape does not extend beyond the dimensions of the board. However, for simplicity, it is assumed that all shapes are normalized such that the x_{min}, y_{min} of the shape's coordinates are normalized to 0, and the other x, y values are adjusted accordingly. This means that the only set of coordinates for the shape in Figure 3 is {(0,0),(0,1),(1,0)}.

Definition 2.2.1. A shape $\boldsymbol{S} \subseteq \mathbf{N} \times \mathbf{N}$ is a non-empty, finite set:

$$\boldsymbol{S} = \{(x_i, y_i) \in \mathbf{N}^2 \mid 0 \leq i < n\} \; for \; some \; n \in \mathbf{N}$$

where n is the total units of the shape. We say that:

- $w = max\{x_i\} - min\{x_i\} + 1 = max\{x_i\} + 1$ is the width of the shape.
- $h = max\{y_i\} - min\{y_i\} + 1 = max\{y_i\} + 1$ is the height of the shape.

And we require that $min\{x_i\} = 0$ and $min\{y_i\} = 0$ (normalization).

Each coordinate within a shape is unique, but there is no limitation to how each coordinate differentiates from the others. Normally, every coordinate in a shape is strongly connected. That is, a coordinate (x_i, y_i) is adjacent to another coordinate (x_j, y_j) either horizontally $(x_j = x_i \pm 1, y_j = y_i)$ or vertically $(x_j = x_i, y_j = y_i \pm 1)$. These connections may form a closed loop, resulting in a *hollow shape*, which contains a hole. In case there is a pair of weakly connected coordinates, where a coordinate (x_i, y_i) is diagonally adjacent to another coordinate (x_j, y_j) such that $(x_j = x_i \pm 1, y_j = y_i \pm 1)$, they can form an *irregular shape*. If not all coordinates in a shape are strongly or weakly connected, the shape is said to be *disconnected*. Shapes exhibiting more than one of these features are referred to as *combination shapes*.

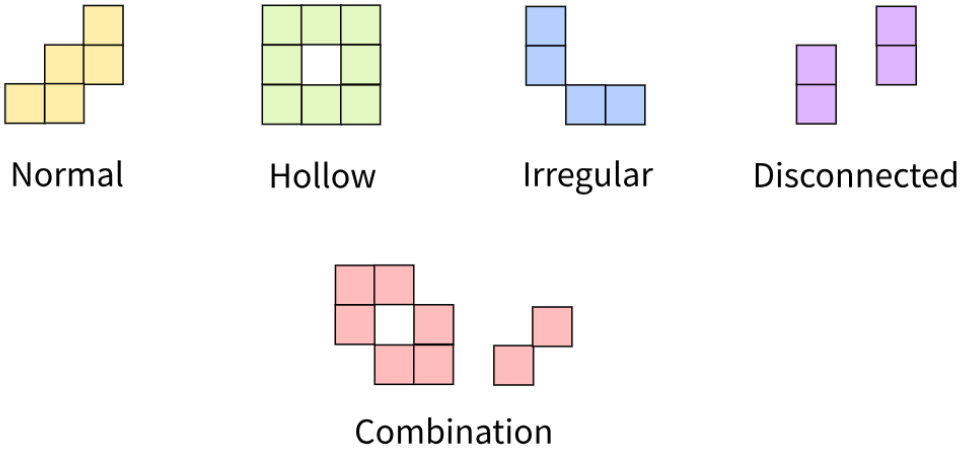


Figure 4: *All types of shape*

Since a single shape is represented as a set of coordinates, the shapes in the puzzle are given as multiple sets of coordinates or a list of shapes. These shapes can be of any type mentioned above. Duplicate shapes are allowed, as long as the total number of coordinates across all shapes does not exceed the total number of units on the board. As the primary goal is to formulate the puzzle as a SAT problem, all given shapes remain fixed throughout the puzzle. This means they cannot be rotated or flipped and retain merely a single orientation. An extended version of the puzzle will later consider rotating shapes if no solution exists under the default orientation. For now, the key adjustable attribute relevant to the solution is the position of each shape on the board.

2.3 The Shape Positions

A shape can be placed anywhere on the board. Since all shapes are assumed to be normalized, they share the same anchor point at coordinate $(0, 0)$. Their positions on the board are relative to this anchor. For instance, as shown in Figure 5, placing a shape defined by the coordinates $\{(1, 0), (1, 1), (0, 1)\}$ at position $(1, 1)$ means that its anchor $(0, 0)$ has been shifted to $(1, 1)$. The shape's coordinates are also shifted accordingly by adding $(1, 1)$ to each, resulting in $\{(2, 1), (2, 2), (1, 2)\}$. If the shape is placed at position $(0, 0)$, the anchor remains unchanged, and the shape retains its original coordinates. This implies that the coordinates of normalized shapes represent their default position at the top-left corner of the board.

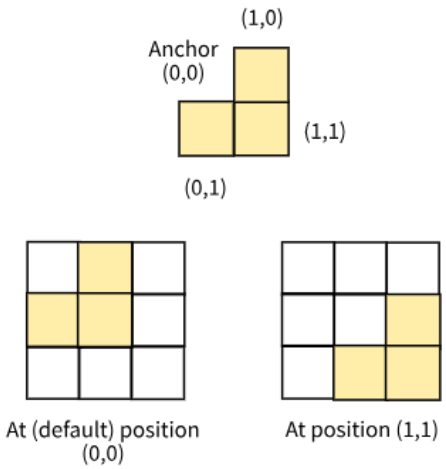


Figure 5: *Different positions of a shape relative to its anchor*

After being shifted to any position on the board, every part of the shape must remain within the board's boundaries. That is, for each shifted coordinate (x_i', y_i') derived from (x_i, y_i), it must hold that $x_i' < W$ and $y_i' < H$. It is important to note that the original coordinates of the shape remain fixed throughout the process. The shifted coordinates relative to the anchor represent the shape's placement on the board—not the shape's structure itself. While a shape's *position* refers to a single coordinate indicating where its anchor is placed, a shape's *placement* refers to the full set of coordinates resulting from shifting the shape to that position.

Definition 2.3.1. Let $\boldsymbol{S} = \{(x_i, y_i)\}_{i=0}^{n-1} \subseteq \mathbf{N}^2$ be a normalized shape.

- Its anchor is at coordinate $(0, 0)$.
- $\boldsymbol{S}$ at position (dx, dy) means its anchor is shifted to (dx, dy), resulting in placement:

$$\boldsymbol{S}_{(\boldsymbol{dx},\boldsymbol{dy})} = \{(x_i + dx, y_i + dy) \mid (x_i, y_i) \in \boldsymbol{S}\}$$

where $0 \leq x_i' < W,\ 0 \leq y_i' < H$ *for all* $(x_i', y_i') \in \boldsymbol{S}_{(\boldsymbol{dx},\boldsymbol{dy})}$.

Even if the anchor remains within the board boundaries after being shifted, it does not necessarily mean that all coordinates of the shapes do as well. Therefore, not all positions on the board are valid for every shape. A valid position (dx, dy) is one where the value of dx does not exceed the difference between the board width W and the shape width w, and likewise for dy in relation to the board height H and the shape height h. Including the default position on the top-left corner of the board, the total number of possible positions for each shape is the product of $((W - w) + 1)$ and $((H - h) + 1)$.

Definition 2.3.2. Let $Board = W \times H$ and $Shape$ has a size of $w \times h$. The possible positions of the shape on the board are coordinates (dx, dy) where:

$$0 \leq dx \leq W - w$$

$$0 \leq dy \leq H - h$$

Definition 2.3.3. Given $Shape$ of size $w \times h$, the total possible positions p of the shape is

$$p = (W - w + 1) \cdot (H - h + 1)$$

Additionally, as illustrated in Figure 5, a shape does not need to occupy every cell within its bounding box. In the example shown, the 2×2 shape can be placed at position $(0, 0)$ without actually covering the coordinate $(0, 0)$. This means that other shapes can also be placed in the same position, as long as their placements do not overlap with any of the coordinates occupied by the shape. For instance, a single-unit shape could also be placed at position $(0, 0)$ without conflict, provided it does not share any coordinates with the existing shape.

3 Encoding The Puzzle into SAT

There are two main requirements of the shape-fitting puzzle that must be encoded into a propositional formula for a SAT solver: *using every input shape* and *without overlaps*. Once these constraints are encoded into a formula, the SAT solver will attempt to find a satisfying assignment. The encoding is constructed such that the formula is *satisfiable* if and only if a solution to the puzzle exists. In that case, the SAT solver returns the valuations of the variables corresponding to the solution. If a propositional variable is assigned `true`, it indicates that the corresponding placement—defined by a set of coordinates for a shape—is selected.

The propositional formula used in SAT solvers is in *conjunctive normal form* (CNF)—a conjunction of one or more clauses, where each clause is a disjunction of literals (propositional variables or their negations) [8]. In this project, each clause represents one of the two constraints, and the full formula is constructed as a conjunction of such clauses, capturing all puzzle requirements.

To simplify the encoding process, each possible placement of a shape is treated as a separate entity and represented by a unique propositional variable. For example, if a shape A

is placed at position (dx, dy), its placement is denoted as $\boldsymbol{A}_{(dx,dy)}$, and is associated with a propositional variable $a_{(dx,dy)}$ that indicates whether this specific placement is selected in the solution. This approach allows each potential configuration to be captured and reasoned about within the SAT formula.

Definition 3.0.1. For every possible position (dx, dy) of a shape A on the board, its placement at (dx, dy) is $\boldsymbol{A}_{(dx,dy)}$.

- The default placement of the shape is $\boldsymbol{A}_{(0,0)}$, with the anchor at the origin.
- Each placement $\boldsymbol{A}_{(dx,dy)}$ is associated with a unique propositional variable $a_{(dx,dy)}$, which is `true` if and only if this placement is selected in the solution.

3.1 Using Every Input Shape

Given the input shapes, each shape must be used exactly once—meaning it must be used at least once and at most once. In other words, among all possible placements of a shape A, at least one placement must be selected, and no two placements can be selected simultaneously.

Definition 3.1.1. Let Shape A have a number of possible placements on the board equal to p. Each placement $\boldsymbol{A}_{(dx_i,dy_i)}$ is represented by a unique propositional variable $a_{(dx_i,dy_i)}$ where $0 \leq i < p$; $a_{(dx_i,dy_i)} = \texttt{true}$ means the shape is in placement $\boldsymbol{A}_{(dx_i,dy_i)}$ at position (dx_i, dy_i). Each input Shape A must be used exactly once.

- Shape A must be used at least once:

$$a_{(dx_0,dy_0)} \vee a_{(dx_1,dy_1)} \vee \ \ldots \ \vee a_{(dx_{p-1},dy_{p-1})}$$

- Shape A must be used at most once:

$$\neg a_{(dx_i,dy_i)} \vee \neg a_{(dx_j,dy_j)}, \ \textit{for all } 0 \leq i < j < p$$

3.2 No Overlaps

When all shapes are used and placed on the board, they must not overlap—meaning no two shapes can occupy the same cell. For example, if Shape A is in placement $\boldsymbol{A}_{(dx,dy)}$, then Shape B must not be placed in a way that includes any coordinate also occupied by $\boldsymbol{A}_{(dx,dy)}$.

Definition 3.2.1. Given Shape A having number of possible positions p, which are placements

$$\boldsymbol{A}_{(\boldsymbol{dx_0},\boldsymbol{dy_0})}, \boldsymbol{A}_{(\boldsymbol{dx_1},\boldsymbol{dy_1})}, ..., \boldsymbol{A}_{(\boldsymbol{dx_{p-1}},\boldsymbol{dy_{p-1}})},$$

and Shape B having number of possible positions q, which are placements

$$\boldsymbol{B}_{(\boldsymbol{dx_0},\boldsymbol{dy_0})}, \boldsymbol{B}_{(\boldsymbol{dx_1},\boldsymbol{dy_1})}, ..., \boldsymbol{B}_{(\boldsymbol{dx_{q-1}},\boldsymbol{dy_{q-1}})}.$$

For all i in $\{0, ..., p-1\}$ and j in $\{0, ..., q-1\}$, if $\boldsymbol{A}_{(\boldsymbol{dx_i},\boldsymbol{dy_i})} \cap \boldsymbol{B}_{(\boldsymbol{dx_j},\boldsymbol{dy_j})} \neq \emptyset$ (i.e., there is a conflict such that at least one coordinate (x, y) is in both $\boldsymbol{A}_{(\boldsymbol{dx_i},\boldsymbol{dy_i})}$ and $\boldsymbol{B}_{(\boldsymbol{dx_j},\boldsymbol{dy_j})}$), then the corresponding propositional variables must satisfy:

$$\neg a_{(dx_i,dy_i)} \vee \neg b_{(dx_j,dy_j)}$$

3.3 Examples

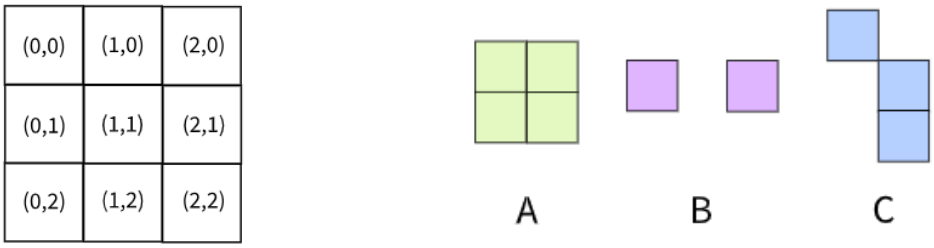
Given a board size and a set of shapes, every possible placement of each shape is encoded as a propositional variable, collectively forming a CNF formula. This formula is then passed to a SAT solver, which determines a satisfying assignment—that is, a set of variable values where each variable assigned `true` corresponds to a valid placement of a shape. The assignment ensures that all shapes are placed on the board exactly once, without overlapping.

According to Definition 3.1.1, each shape is associated with one disjunctive clause to ensure it is used at least once, and $\binom{p}{2}$ disjunctive clauses to ensure it is used at most once, where p is the number of possible placements of the shape. When a placement of a shape overlaps with another shape, the solution must also satisfy the disjunctive clauses described in Definition 3.2.1. All of these clauses are combined using conjunctions, resulting in a single formula representing the entire puzzle.

In some cases, a puzzle may have multiple valid configurations—different ways of arranging the shapes on the board such that all constraints are satisfied—where each configuration corresponds to a distinct satisfying assignment of the SAT formula. However, a SAT solver returns only one such assignment. Therefore, the formula corresponding to any puzzle with at least one valid configuration is considered *satisfiable*. Conversely, if no satisfying assignment exists, it indicates that at least one shape cannot be placed without violating the constraints.

3.3.1 Simple 3×3 Board

Assume a board is defined with dimensions $W = 3$ and $H = 3$, and a set of shapes is given as follows: $\boldsymbol{A} = \{(0,0),(0,1),(1,1),(1,0)\}$, $\boldsymbol{B} = \{(0,0),(2,0)\}$, $\boldsymbol{C} = \{(0,0),(1,1),(1,2)\}$, as illustrated in Figure 6.



(a) *3×3 board* ***(b)*** *Simple mixed shapes*

Figure 6: *Simple board with mixed shapes*

Each coordinate on the board represents a position where an anchor of a shape can be placed. The total number of coordinates across all shapes is equal to the number of units on the board, which is 9. Shape A is a 2×2, Shape B is a 3×1, and Shape C is a 2×3—none of them exceed the board boundaries. According to Definitions 2.3.2 and 2.3.3, the number of possible positions for each shape is calculated as follows:

- For Shape A: $p_A = (3-2+1)\cdot(3-2+1) = 4$, with positions $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$.
- For Shape B: $p_B = (3-3+1)\cdot(3-1+1) = 3$, with positions $(0,0)$, $(0,1)$, and $(0,2)$.
- For Shape C: $p_C = (3-2+1)\cdot(3-3+1) = 2$, with positions $(0,0)$ and $(1,0)$.

Once the possible positions for each shape are determined, their placements can be represented as propositional variables. According to Definition 3.1.1, Shapes A, B, and C are each used exactly once if the following clauses are satisfied:

- Shape A
  - At least one placement:

$$a_{(0,0)} \vee a_{(1,0)} \vee a_{(0,1)} \vee a_{(1,1)} \tag{1}$$

  - At most one placement:

$$\begin{aligned}(\neg a_{(0,0)} \vee \neg a_{(1,0)}) \wedge (\neg a_{(0,0)} \vee \neg a_{(0,1)}) \wedge (\neg a_{(0,0)} \vee \neg a_{(1,1)}) \wedge \\ (\neg a_{(1,0)} \vee \neg a_{(0,1)}) \wedge (\neg a_{(1,0)} \vee \neg a_{(1,1)}) \wedge (\neg a_{(0,1)} \vee \neg a_{(1,1)})\end{aligned} \tag{2}$$

- Shape B
  - At least one placement:

$$b_{(0,0)} \vee b_{(0,1)} \vee b_{(0,2)} \tag{3}$$

  - At most one placement:

$$(\neg b_{(0,0)} \vee \neg b_{(0,1)}) \wedge (\neg b_{(0,0)} \vee \neg b_{(0,2)}) \wedge (\neg b_{(0,1)} \vee \neg b_{(0,2)}) \tag{4}$$

- Shape C
  - At least one placement:

$$c_{(0,0)} \vee c_{(1,0)} \tag{5}$$

  - At most one placement:

$$\neg c_{(0,0)} \vee \neg c_{(1,0)} \tag{6}$$

Each placement of each shape must not overlap with any other. Therefore, the coordinates of these placements must be evaluated to ensure they do not intersect. For ease of observation, these placements are arranged in a table, shown in Figure 7, that mimics the board layout, making it clear which cells are occupied by each placement.

$\boldsymbol{A}_{(0,0)} = \{(0,0),(0,1),(1,1),(1,0)\}$
$\boldsymbol{A}_{(1,0)} = \{(1,0),(1,1),(2,1),(2,0)\}$
$\boldsymbol{A}_{(0,1)} = \{(0,1),(0,2),(1,2),(1,1)\}$
$\boldsymbol{A}_{(1,1)} = \{(1,1),(1,2),(2,2),(2,1)\}$
$\boldsymbol{B}_{(0,0)} = \{(0,0),(2,0)\}$
$\boldsymbol{B}_{(0,1)} = \{(0,1),(2,1)\}$
$\boldsymbol{B}_{(0,2)} = \{(0,2),(2,2)\}$
$\boldsymbol{C}_{(0,0)} = \{(0,0),(1,1),(1,2)\}$
$\boldsymbol{C}_{(1,0)} = \{(1,0),(2,1),(2,2)\}$

$(0,0)$: $A_{(0,0)}$ $B_{(0,0)}$ $C_{(0,0)}$	$(1,0)$: $A_{(0,0)}$, $A_{(1,0)}$ $C_{(1,0)}$	$(2,0)$: $A_{(1,0)}$ $B_{(0,0)}$
(0,1): $A_{(0,0)}$, $A_{(0,1)}$ $B_{(0,1)}$	**(1,1):** $A_{(0,0)}$, $A_{(1,0)}$, $A_{(0,1)}$, $A_{(1,1)}$ $C_{(0,0)}$	**(2,1):** $A_{(1,0)}$, $A_{(1,1)}$ $B_{(0,1)}$ $C_{(1,0)}$
(0,2): $A_{(0,1)}$ $B_{(0,2)}$	**(1,2):** $A_{(0,1)}$, $A_{(1,1)}$ $C_{(0,0)}$	**(2,2):** $A_{(1,1)}$ $B_{(0,2)}$ $C_{(1,0)}$

Figure 7: *Placement coverage for each cell*

For every cell that contains one or more placement variables, at most one of those variables can be selected. According to Definition 3.2.1, the negated propositional variables corresponding to each cell are combined into pairwise disjunction clauses and joined by conjunctions. This process is similar to ensuring that at most one placement is selected per shape; therefore, any pair of placements corresponding to the same shape do not need to be considered in this part.

- $(0,0)$:

$$(\neg a_{(0,0)} \vee \neg b_{(0,0)}) \wedge (\neg a_{(0,0)} \vee \neg c_{(0,0)}) \wedge (\neg b_{(0,0)} \vee \neg c_{(0,0)}) \tag{7}$$

- $(1,0)$:

$$(\neg a_{(0,0)} \vee \neg c_{(1,0)}) \wedge (\neg a_{(1,0)} \vee \neg c_{(1,0)}) \tag{8}$$

- $(2,0)$:

$$\neg a_{(1,0)} \vee \neg b_{(0,0)} \tag{9}$$

- $(0,1)$:

$$(\neg a_{(0,0)} \vee \neg b_{(0,1)}) \wedge (\neg a_{(0,1)} \vee \neg b_{(0,1)}) \tag{10}$$

- $(1,1)$:

$$(\neg a_{(0,0)} \vee \neg c_{(0,0)}) \wedge (\neg a_{(1,0)} \vee \neg c_{(0,0)}) \wedge (\neg a_{(0,1)} \vee \neg c_{(0,0)}) \wedge (\neg a_{(1,1)} \vee \neg c_{(0,0)}) \tag{11}$$

- $(2,1)$:

$$\begin{aligned}(\neg a_{(1,0)} \vee \neg b_{(0,1)}) \wedge (\neg a_{(1,0)} \vee \neg c_{(1,0)}) \wedge (\neg b_{(0,1)} \vee \neg c_{(1,0)}) \wedge \\ (\neg a_{(1,1)} \vee \neg b_{(0,1)}) \wedge (\neg a_{(1,1)} \vee \neg c_{(1,0)})\end{aligned} \tag{12}$$

- $(0,2)$:

$$\neg a_{(0,1)} \vee \neg b_{(0,2)} \tag{13}$$

- $(1, 2)$:

$$(\neg a_{(0,1)} \vee \neg c_{(0,0)}) \wedge (\neg a_{(1,1)} \vee \neg c_{(0,0)}) \tag{14}$$

- $(2, 2)$:

$$(\neg a_{(1,1)} \vee \neg b_{(0,2)}) \wedge (\neg a_{(1,1)} \vee \neg c_{(1,0)}) \wedge (\neg b_{(0,2)} \vee \neg c_{(1,0)}) \tag{15}$$

Now that all constraints of the puzzle have been expressed in CNF form—where every clause from Equations (1) to (15) is combined using conjunctions into a single formula—the next step is to assign truth values to the variables such that each clause is satisfied, making the entire formula satisfiable. In the implementation, this step will be performed by the SAT solver.

The most straightforward approach is to begin with the shape that has the fewest possible positions, which in this case is Shape C. Suppose Shape C is placed at position $(0, 0)$, meaning $c_{(0,0)} = \texttt{true}$. Then, according to Equation (6), $c_{(1,0)}$ must be `false`. However, as seen in Equation (11), if $c_{(0,0)} = \texttt{true}$, all possible placements of Shape A must be `false`—implying that Shape A cannot be placed anywhere on the board and violating Equation (1). Therefore, $c_{(0,0)}$ must be `false`, and $c_{(1,0)}$ must be `true`, as required by Equation (5).

Consequently, based on Equations (12) and (15), $b_{(0,1)}$ and $b_{(0,2)}$ must be `false`, as well as $a_{(1,0)}$ and $a_{(1,1)}$. This leaves Shape B with only one valid placement: position $(0, 0)$. With $b_{(0,0)} = \texttt{true}$ and $c_{(1,0)} = \texttt{true}$, Equations (7) and (8) force $a_{(0,0)}$ to be `false`. The only remaining valid placement for Shape A is $a_{(0,1)}$, satisfying Equation (1). Thus, the final satisfying assignment for the formula is $a_{(0,1)} = \texttt{true}$, $b_{(0,0)} = \texttt{true}$, and $c_{(1,0)} = \texttt{true}$. Figure 8 illustrates the solution placements corresponding to these variable assignments.

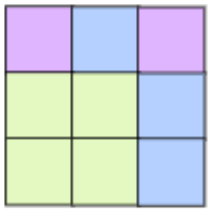


Figure 8: *The solution placements of Shapes A, B, and C*

3.3.2 No Solution Puzzle

Given the same board size as in the previous example, which is 3×3, a similar set of shapes is defined as $\{B, C, D\}$, where Shape $D = \{(0, 0), (1, 1), (2, 2)\}$. This means that the only valid position for Shape D is $(0, 0)$, since the shape's width and height match the board dimensions. The placement $\boldsymbol{D}_{(0,0)}$ directly corresponds to the shape's original coordinates.

The solution to this puzzle can be easily inferred by examining the placement coverage of the cell occupied by Shape D.

- $(0, 0)$:

$$(\neg d_{(0,0)} \vee \neg b_{(0,0)}) \wedge (\neg d_{(0,0)} \vee \neg c_{(0,0)}) \wedge (\neg b_{(0,0)} \vee \neg c_{(0,0)}) \tag{16}$$

- $(1, 1)$:

$$\neg d_{(0,0)} \vee \neg c_{(0,0)} \tag{17}$$

- $(2, 2)$:

$$(\neg d_{(0,0)} \vee \neg b_{(0,2)}) \wedge (\neg d_{(0,0)} \vee \neg c_{(1,0)}) \wedge (\neg b_{(0,2)} \vee \neg c_{(1,0)}) \tag{18}$$

When $d_{(0,0)}$ is set to `true`, $b_{(0,0)}$ and $c_{(0,0)}$ must be `false`, as indicated by Equations (16) and (17). Additionally, Equation (18) forces $b_{(0,2)}$ and $c_{(1,0)}$ to be `false` as well. This leaves $(0, 1)$ as the only valid position for Shape B. However, no valid placements remain for Shape C, which has only two possible positions on the board. Therefore, Shapes D and C cannot be placed together on the board without overlapping. As a result, the formula is unsatisfiable, and the puzzle has no solution.

4 Implementation

The entire puzzle was implemented in Python 3.10.12 using the PySAT library, which provides a flexible interface for CNF construction and SAT solving. The program takes as input a board size and a list of shapes, determines all valid positions for each shape, associates them with propositional variables, and constructs CNF constraints based on the formal encoding described earlier. MiniSat is then used to check whether a valid assignment exists, and the result is decoded into a human-readable board configuration. The complete source code is provided in Appendices.

4.1 Files and Functions

- *input.txt*
  This file simulates user input, containing the board size and a list of shapes.
- *input_utils.py*
  This file contains two functions:
  - **load_input()** reads the user input from *input.txt*.
  - **validate_input()** checks basic criteria of the input, such as whether all values are positive integers, whether the total units across all shapes are appropriate, and whether the shape coordinates fall within the board boundaries.

- *puzzle.py*
  This file contains the main function, **solve_puzzle()**, which handles the core logic of solving the puzzle, including constructing the two CNF constraint clauses. The complete CNF formula is then passed to MinisatGH to obtain the variable assignments, which are returned by the function.

- *print.py*
  The **print_result_board()** function in this file receives the variable assignments and prints the resulting board configuration cell by cell.

- *main.py*
  This file serves as the program's entry point. It loads the input using the **load_input()** function, solves the puzzle using the **solve_puzzle()** function, and prints the result using the **print_result_board()** function if a solution exists.

4.2 Inputs and Variable Encoding

This section describes how the program handles user inputs and internal representations.

4.2.1 User Inputs

- **Board dimensions:** Two positive integers, `W` (width) and `H` (height).
- **Shapes:** A list of shapes, where each shape is a list of coordinates (x, y) representing the shape's structure.
  - Each shape is uniquely identified by its index (ID) in the shape list.

4.2.2 Placements Representation

According to Definition 2.3.1, a placement is a set of coordinates on the board that a shape occupies when placed at a certain position. In the program, however, each placement is represented as a tuple:

`(shape_id, dx, dy)`.

This tuple indicates the shape's ID and the position (dx, dy) where its anchor $(0, 0)$ is placed. The full coordinates of the placement can be derived using the shape's structure and the offset (dx, dy).

4.2.3 Propositional Variable Mapping

SAT solvers such as MiniSat require all variables to be represented as integers, where:

- Positive integers denote variables.
- Negative integers denote negated variables.

Because of this, the program needs to maintain a mapping between placements and integer variables:

- `placement_to_var`: A dictionary that maps a placement $(shape\_id, dx, dy)$ to its corresponding propositional variable (an integer).
- `vars_for_shape`: A list that stores all propositional variables associated with a shape.

4.2.4 Cell-to-Variable Associations

To enforce constraints related to cell occupancy, the program keeps track of which placements use which board cells:

- `cell_to_vars`: A mapping where each key is a cell coordinate (x, y), and the value is a list of propositional variables corresponding to placements that occupy that cell. This structure mirrors the logic shown in Figure 7.

4.2.5 Solution Decoding

After the SAT solver returns a satisfying assignment, the solution must be interpreted:

- `cell_to_shape`: A mapping that connects each occupied board cell to the $shape\_id$ placed there. This is used to reconstruct the final board configuration.

4.3 Solving the Puzzle

After loading the input, the board dimensions `W` and `H` along with a list `shapes` are passed to the **`solve_puzzle()`** function. The function begins by initializing `cnf = CNF()` to hold the SAT formula and sets `curr_var = 1` to start assigning propositional variable identifiers. It then determines all valid positions (placements) for each shape—following Definition 2.3.2—and maps each placement to a unique propositional variable, stored in `placement_to_var`.

```
for shape_id, shape in enumerate(shapes):
    x_max = max(x for x, _ in shape)
    y_max = max(y for _, y in shape)

    for dy in range (H - y_max):
        for dx in range(W - x_max):
            placement_to_var[(shape_id, dx, dy)] = curr_var
            curr_var += 1
```

For each shape, all corresponding propositional variables are collected into `vars_for_shape`. Two types of clauses are added to `cnf` for the **using-exactly-once** constraint:

- A disjunction of all placement variables for the shape, ensuring the shape is used *at least once*.
- A set of pairwise negated disjunctions among those variables, ensuring the shape is used *at most once*.

```
for shape_id in range(len(shapes)):
    vars_for_shape = [var for (id, _, _), var in
    ↪  placement_to_var.items() if id == shape_id]

    # At least one placement
    cnf.append(vars_for_shape)
    # At most one placement
    for v1, v2 in combinations(vars_for_shape, 2):
        cnf.append([-v1, -v2])
```

To enforce the **no-overlap** constraint, the function examines all placements recorded in `placement_to_var` to build the mapping `cell_to_vars`. For each cell in the mapping, pairwise negated disjunctions of the variables are added to `cnf`, ensuring that *at most one shape can occupy any given cell.*

```
# Check which variables (placements) use the same cell
for (id, dx, dy), var in placement_to_var.items():
    for x, y in shapes[id]:
        xi, yi = x+dx, y+dy

        if (xi, yi) not in cell_to_vars:
            cell_to_vars[(xi, yi)] = []
        cell_to_vars[(xi, yi)].append(var)

# No two variables share the same cell
for cell_vars in cell_to_vars.values():
    for v1, v2 in combinations(cell_vars, 2):
        cnf.append([-v1, -v2])
```

Finally, the completed CNF formula is passed to the MinisatGH solver. If the formula is satisfiable, the solver returns a model: a list of variables assigned `True`, indicating which placements have been selected. The function uses this model to construct the `cell_to_shape` mapping.

```
with MinisatGH(bootstrap_with=cnf) as solver:
    if solver.solve():
        model = solver.get_model()
        for (id, dx, dy), var in placement_to_var.items():
            if var in model:
                print(f"Place shape {id} at position ({dx}, {dy})")

                for x, y in shapes[id]:
                    cell_to_shape[(x+dx, y+dy)] = id
```

If no satisfying assignment exists, the mapping remains empty, indicating that the puzzle has no solution. This mapping is returned as the function's output.

4.4 Rotating Shapes

If shape rotation is allowed, the program can continue to search for a solution even when none exists with the original, unrotated shapes. This extended version of the program, which supports shape rotations, is included in Appendix C.

The key difference in this version is the addition of a file called *rotate.py*, which defines four helper functions:

- **rotate_shape()** rotates a given shape by a specified degree. Four rotation options are supported: 0°(default), 90°, 180°, and 270°.
- **normalize_shape()** adjusts a rotated shape so that its coordinates are normalized (i.e., $min\{x_i\}$ and $max\{y_i\}$ equal to 0) and checks whether the shape still fits within the board dimensions after rotation.
- **get_rotated_shapes()** takes a list of shapes and applies a specific rotation degree to each one using the functions **rotate_shape()** and **normalize_shape()**.
- **try_rotations()** attempts different rotation combinations across all shapes if the initial (unrotated) configuration fails to produce a solution. It repeatedly calls the **solve_puzzle()** function with rotated versions of the input shapes until a solution is found or all combinations have been tried.

In the original version of the program, the *main.py* file directly calls **solve_puzzle()** with the user-provided shapes. In the rotation-enabled version, *main.py* instead calls **try_rotations()**, which handles all possible rotations and passes them to **solve_puzzle()** one by one. Importantly, the **solve_puzzle()** function remains unchanged because it treats the input list of shapes as fixed. The only addition is that it now receives the rotation degrees along with shapes, so that the correct rotation of each shape can be reported in the final solution. Once a rotated configuration leads to a valid solution, **solve_puzzle()** proceeds exactly as before to generate the result.

In both versions of the program, the number of propositional variables and clauses grows exponentially with the puzzle size and the number of shapes, leading to longer solving times and higher memory usage. Additionally, the current implementation of the **solve_puzzle()** function does not eliminate redundant pairwise comparisons in the at-most-once constraint for overlap prevention, unlike the more optimized encoding in the earlier examples. In the version with shape rotations, some shapes may remain unchanged after rotation, introducing unnecessary redundancy in the solution space. These factors can impact solver performance. Alternative solver types, such as SMT solvers (e.g., Z3), may offer better scalability by supporting higher-level constraints and more compact encodings. Further improvements to the implementation, such as refining constraint conditions or pruning equivalent shape placements, can possibly reduce unnecessary computation and improve efficiency.

5 Conclusion

This thesis explored the formulation and solution of shape-fitting puzzles as Boolean satisfiability problems using SAT encoding and the MiniSat solver. By translating the puzzle's components into logical representations and encoding shape placements and constraints as propositional variables, the implementation demonstrates how a SAT solver can effectively solve small- to medium-sized instances of the puzzle. The work highlights the flexibility of the approach in supporting custom board sizes, varied shapes, and optional shape rotation. However, one limitation of the current implementation is its reliance on propositional logic, which causes the number of variables and clauses to grow exponentially as the puzzle size and complexity increase. This scalability issue suggests the potential for future work to explore more expressive solvers such as SMT (e.g., Z3) or to develop hybrid methods that incorporate constraint simplification and symmetry reduction. Overall, the project illustrates both the practicality and challenges of SAT-based techniques in tackling combinatorial puzzle-solving.

References

1. Bofill, M., Palahí, M., Suy, J., & Villaret, M. (2012). Solving constraint satisfaction problems with SAT modulo theories. *Constraints, 17*(3), 273–303. https://doi.org/10.1007/s10601-012-9123-1

2. Burgess, M. A., Gretton, C., Milthorpe, J., Croak, L., Willingham, T., & Tiu, A. (2022). Dagster: Parallel Structured Search with Case Studies. In S. Khanna, J. Cao, Q. Bai, & G. Xu (Eds.), *PRICAI 2022: Trends in Artificial Intelligence* (Vol. 13629, pp. 75-89). Springer. https://doi.org/10.1007/978-3-031-20862-1_6

3. Davis, L., & Ji, T. (2025). *Evaluating SAT and SMT Solvers on Large-Scale Sudoku Puzzles*. arXiv.org. https://arxiv.org/abs/2501.08569

4. Florian, A. (2023). *Modeling and Solving Problems Using Propositional Logic and SAT Solvers* [Master's thesis, University of Liège]. http://hdl.handle.net/2268.2/17699

5. Gorjiara, H., Xu, G. H., & Demsky, B. (2020). Satune: synthesizing efficient SAT encoders. *Proceedings of the ACM on Programming Languages, 4*(OOPSLA), 1–32. https://doi.org/10.1145/3428214

6. Groza, A. (2021). Polyomino Puzzles. In *Modelling Puzzles in First Order Logic* (pp. 281–317). Springer. https://doi.org/10.1007/978-3-030-62547-4_12

7. Heule, M. J. H., & Verwer, S. (2010). Exact DFA Identification Using SAT Solvers. In J. M. Sempere & P. García (Eds.), *ICGI 2010* (LNAI 6339, pp. 66–79). Springer-Verlag Berlin Heidelberg.

8. Pfahringer, B. (2011). Conjunctive Normal Form. In C. Sammut & G. I. Webb (Eds.), *Encyclopedia of Machine Learning*. Springer. https://doi.org/10.1007/978-0-387-30164-8_158

Appendices

A Python Program

input.txt

```
W = 3
H = 3
shapes = [
    [(1,0), (0,1), (1,1), (1,0)],
    [(0,0), (2,0)],
    [(0,0), (1,1), (1,2)]
]
```

input_utils.py

```
def load_input(filename):
    local_vars = {}
    with open(filename, 'r') as f:
        code = f.read()
        exec(code, {}, local_vars)

    if validate_input(local_vars['W'], local_vars['H'],
    ↪  local_vars['shapes']):
        return local_vars['W'], local_vars['H'], local_vars['shapes']
    else:
        exit()

def validate_input(W, H, shapes):
    # Validate board W and H
    if not isinstance(W, int) or not isinstance(H, int) or W <= 0 or H
    ↪  <= 0:
        print("W and H must be positive integers.")
        return False

    # Validate shapes
    total_coords = sum(len(shape) for shape in shapes)
    if total_coords > W * H:
        print("Invalid total coordinates of the shapes")
        return False

    for shape_id, shape in enumerate(shapes):
        if not isinstance(shape, list):
            print("Each shape must be a list of coordinates.")
            return False
        for coord in shape:
            if not (isinstance(coord, tuple) and len(coord) == 2):
                print(f"Shape {shape_id} has invalid coordinate
                ↪  format: {coord}")
                return False
```

```
            x, y = coord
            if not (isinstance(x, int) and isinstance(y, int)) or x <
            ↪  0 or y < 0:
                print(f"Coordinates (in shape {shape_id}) must be
                ↪  positive integers: {coord}")
                return False
            if x >= W or y >= H:
                print(f"Shape {shape_id} has invalid coordinate
                ↪  against the board size.")
                return False

    return True
```

puzzle.py

```
from pysat.formula import CNF
from pysat.solvers import MinisatGH
from itertools import combinations


def solve_puzzle(W, H, shapes):
    ### POSSIBLE PLACEMENTS ###
    cnf = CNF()
    curr_var = 1
    placement_to_var = {}

    for shape_id, shape in enumerate(shapes):
        # Compute boundaries of this shape
        x_max = max(x for x, _ in shape)
        y_max = max(y for _, y in shape)

        # Save all possible positions (relative to anchor) of this
        ↪  shape on the board
        for dy in range (H - y_max):
            for dx in range(W - x_max):
                placement_to_var[(shape_id, dx, dy)] = curr_var
                curr_var += 1


    ### USING EVERY SHAPE EXACTLY ONCE ###
    for shape_id in range(len(shapes)):
        # Retrieve all placement variables of this shape
        vars_for_shape = [var for (id, _, _), var in
        ↪  placement_to_var.items() if id == shape_id]
        # At least one placement
        cnf.append(vars_for_shape)
        # At most one placement
        for v1, v2 in combinations(vars_for_shape, 2):
            cnf.append([-v1, -v2])
```

```
    ### NO OVERLAPS ###
    cell_to_vars = {}

    # Check which variables (placements) use the same cell
    for (id, dx, dy), var in placement_to_var.items():
        # Retrive actual placement of this shape -> exact position
        ↪  (cell) of each unit of the shape
        for x, y in shapes[id]:
            xi, yi = x+dx, y+dy

            if (xi, yi) not in cell_to_vars:
                cell_to_vars[(xi, yi)] = []
            # Add the placement variable to this cell -> a part of the
            ↪  shape uses the cell
            cell_to_vars[(xi, yi)].append(var)

    # No two variables share the same cell
    for cell_vars in cell_to_vars.values():
        for v1, v2 in combinations(cell_vars, 2):
            cnf.append([-v1, -v2])


    ### SOLVE THE SAT PROBLEM ###
    cell_to_shape = {}

    with MinisatGH(bootstrap_with=cnf) as solver:
        if solver.solve():
            model = solver.get_model()
            for (id, dx, dy), var in placement_to_var.items():
                if var in model:
                    print(f"Place shape {id} at position ({dx},
                    ↪  {dy})")

                    for x, y in shapes[id]:
                        cell_to_shape[(x+dx, y+dy)] = id

    return cell_to_shape
```

print.py

```
def print_result_board(W, H, cell_to_shape):
    print(f"\nBoard {W} x {H}:")

    max_id = max(cell_to_shape.values(), default=0)
    digits = len(str(max_id))

    for y in range(H):
        for x in range(W):
            if (x, y) in cell_to_shape:
```

```
                id = cell_to_shape[(x, y)]
                print(f"{id:0{digits}d}", end=" ")
            else:
                print("+" * digits, end=" ")
        print()
```

B No Rotation Version

main.py

```
from input_utils import load_input
from puzzle import solve_puzzle
from print import print_result_board


### USER DEFINED ###
W, H, shapes = load_input("input.txt")

### SOLVING PUZZLE ###
result_board = solve_puzzle(W, H, shapes)

if result_board:
    ### PRINT RESULT ###
    print_result_board(W, H, result_board)

else:
    print("No solution")
```

C Rotating Shapes Version

rotate.py

```
import itertools
from puzzle import solve_puzzle


ROTATIONS = [0, 90, 180, 270]

def rotate_shape(shape, degree):
    if degree == 90:
        return [(y, -x) for (x, y) in shape]
    elif degree == 180:
        return [(-x, -y) for (x, y) in shape]
    elif degree == 270:
        return [(-y, x) for (x, y) in shape]
    return shape
```

```
def normalize_shape(shape, W, H):
    min_x = min(x for x, _ in shape)
    min_y = min(y for _, y in shape)
    normalized = [(x - min_x, y - min_y) for (x, y) in shape]

    # Check boundaries
    max_x = max(x for x, y in normalized)
    max_y = max(y for x, y in normalized)
    if max_x >= W or max_y >= H:
        return None
    return normalized

def get_rotated_shapes(W, H, original_shapes, degrees):
    rotated_shapes = []

    for shape, deg in zip(original_shapes, degrees):
        rotated = rotate_shape(shape, deg)
        normalized = normalize_shape(rotated, W, H)

        if normalized is None:
            rotated_shapes = []
            break
        rotated_shapes.append(normalized)

    return rotated_shapes

def try_rotations(W, H, original_shapes):
    solution = {}

    # Establish all degree combinations
    for degrees in itertools.product(ROTATIONS,
    ↪  repeat=len(original_shapes)):
        rotated_shapes = get_rotated_shapes(W, H, original_shapes,
        ↪  degrees)
        if not rotated_shapes:
            continue

        # Try solving
        solution = solve_puzzle(W, H, rotated_shapes, degrees)

        if solution:
            return solution

    return solution
```

puzzle.py

```
from pysat.formula import CNF
from pysat.solvers import MinisatGH
from itertools import combinations
```

```python
def solve_puzzle(W, H, shapes, degrees):

    ...

    ### SOLVE THE SAT PROBLEM ###
    cell_to_shape = {}

    with MinisatGH(bootstrap_with=cnf) as solver:
        if solver.solve():
            model = solver.get_model()
            for (id, dx, dy), var in placement_to_var.items():
                if var in model:
                    print(f"Place shape {id} rotated by {degrees[id]}
                    ↪  degree at position ({dx}, {dy})")

                    for x, y in shapes[id]:
                        cell_to_shape[(x+dx, y+dy)] = id

    return cell_to_shape
```

main.py

```python
from input_utils import load_input
from rotate import try_rotations
from print import print_result_board


### USER DEFINED ###
W, H, shapes = load_input("input.txt")

### SOLVING PUZZLE ###
result_board = try_rotations(W, H, shapes)

if result_board:
    ### PRINT RESULT ###
    print_result_board(W, H, result_board)

else:
    print("No solution")
```