Universiteit Leiden
The Netherlands

Opleiding Informatica

Solving Fillomino:

An Algorithmic and SMT-Based Approach

Ryan Behari

Supervisors:
Jeannette de Graaf & Rudy van Vliet

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

31/07/2025

Abstract

Fillomino is a Japanese logic-based puzzle. The goal of the puzzle is to divide a given grid into polyominoes, such that each polyomino contains exactly the number of cells as indicated within those cells. This thesis presents an algorithmic approach to solving Fillomino puzzles, using a combination of local deduction rules and backtracking. Furthermore, we use Satisfiability Modulo Theories (SMT) solvers to find valid solutions for a given puzzle by translating Fillomino rules into a format that SMT solvers understand. Additionaly, we apply both the SMT-based method and the algorithmic approach on two widely known variants of the puzzle, demonstrating how they can be used to find valid solutions.

Contents

1 Introduction

Fillomino is a logic-based puzzle created by Japanese puzzle magazine publisher Nikoli. Nikoli is best known for popularizing puzzles such as Sudoku and Numberlink. The objective in Fillomino is to partition a grid into connected regions, known as polyominoes. Each region contains exactly the number of cells as indicated by the number within that region. Multiple regions are allowed to contain the same number, as seen in Figure 1. Additionally, two or more regions of the same size are not allowed to touch each other along their edges, though diagonal contact is allowed. A complete overview of the rules can be found in Section 3.

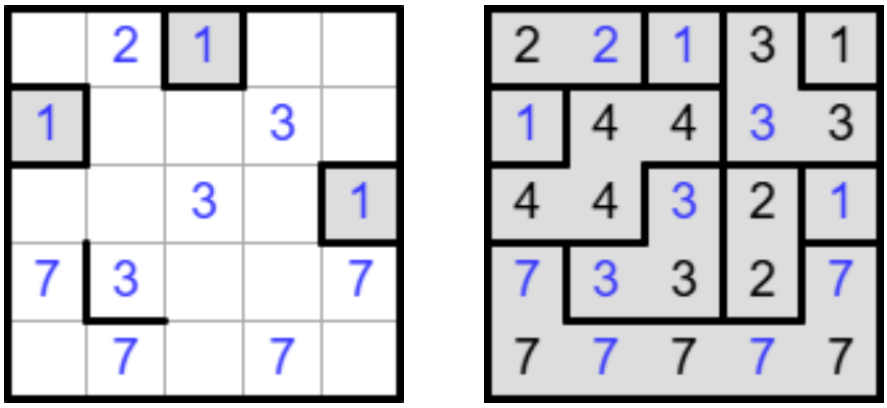


Figure 1: An example of a Fillomino puzzle and its corresponding solution.

Our research question is the following: **How can Fillomino puzzles be solved using algorithmic strategies and SMT?**

2 Related work

Fillomino is a logic-based puzzle that until recently, lacked a direct proof of its computational complexity. According to the comprehensive work of Robert Hearn and Erik Demaine on puzzle complexity [HD09], Fillomino is known to be $\mathcal{ASP}$-complete, which implies $\mathcal{NP}$-completeness, although it does not provide a polynomial-time reduction from a known $\mathcal{NP}$-complete problem. However, in a recent study by Thijs van de Griendt [vdG25], Fillomino was proven to be $\mathcal{NP}$-complete by a polynomial-time reduction from the Hamiltonian Path problem ($HP3G$), which is a well-known $\mathcal{NP}$-complete problem. This puts Fillomino among other $\mathcal{NP}$-complete logic-based puzzles like Sudoku, Slitherlink and Cross-Sum [YS03].

The class $\mathcal{NP}$[1] is the class of problems that can be verified, where solutions of the problem can be verified in polynomial time. If every problem in $\mathcal{NP}$ can be reduced to a certain decision problem in polynomial time, that problem is called $\mathcal{NP}$-hard, meaning that it is at least as difficult as all other problems in $\mathcal{NP}$. A decision problem is classified as $\mathcal{NP}$-complete if it is both in $\mathcal{NP}$ and $\mathcal{NP}$-hard, meaning informally that $\mathcal{NP}$-complete problems are among the most challenging problems in $\mathcal{NP}$. This classification indicates that Fillomino is part of the most computationally complex problems, as an efficient solution to it would allow other difficult problems to be solved efficiently, as well. More detailed information about complexity classes can be found in [Sip13].

To solve puzzles like Fillomino, methods like local deduction rules and backtracking [RN10] are used. To make solving more efficient, constraint propagation can be used to eliminate invalid choices early on by applying local constraints to reduce the search space.

More recently, translating puzzles into satisfiability problems has gained interest. Satisfiability Modulo Theories (SMT) solvers, like Yices [Dut] and Z3 [dMB] can be used to handle complex constraints, such as connectivity and region sizes, that are important for puzzles like Fillomino. Several puzzles with connectivity constraints have previously been studied by Gerhard van der Knijff [dK21], who implemented connectivity constraints as SMT problems for six puzzles, including Slitherlink, Hitori, Nurikabe and Hashi.

Previous works from students of Leiden University such as Niels Heslenfeld's [Hes25] thesis "Solving and generating Fobidoshi puzzles", explored algorithmic and SMT-based techniques for a puzzle with comparable connectivity constraints. This research provided inspiration for our project and helped shape the structure of this thesis.

Finally, the basic ideas from computational complexity theory [AB06] offer insight into the difficulty of puzzles and the effectiveness of various solving techniques, for example by classifying them into complexity classes and describing how time and space requirements grow with input size.

[1] More formally, $\mathcal{NP}$ is the class of decision problems that can be solved in polynomial time by a non-deterministic Turingmachine

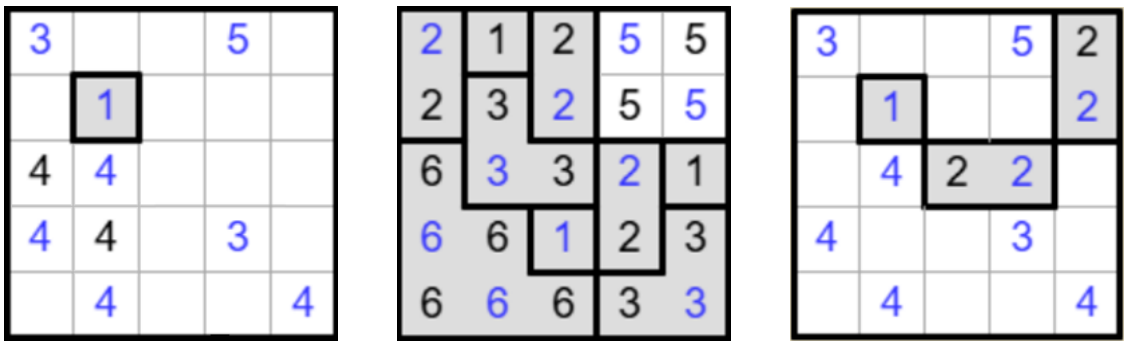
3 Puzzle Rules

Fillomino is played on a rectangular board that contains some pre-filled numbers and empty cells. The goal is to fill in all empty cells with numbers such that the board is divided into connected regions of cells with the same number, called groups or blocks, each forming the shape of a polyomino that contains exactly the number of cells as the number written inside the cells.

To better understand the mechanics of Fillomino, consider the following:

- The initial board includes some pre-filled numbers that indicate the exact size of the connected block each number belongs to.
- The player's task is to determine the correct positioning of blocks on the board while following the puzzle's rules, including that blocks of the same size are not allowed to touch each other horizontally or vertically. An example of a puzzle and its solution can be seen in Figure 1 in Section 1.
- In principle, the player is allowed to create new blocks to complete the puzzle, though some variants restrict this. The two primary variants are discussed in Section 4.

To illustrate these rules, Figure 2 presents several examples of Fillomino boards, with invalid and valid configurations based on the puzzle's requirements.



(a) (b) (c)

Figure 2: Examples of Fillomino boards.

- (a) is invalid, since there exists a group of 4s that contains 5 cells.
- (b) is invalid, since the board is completely filled, but there is a group of 5s that only contains 4 cells instead of 5.
- (c) is valid even though some blocks of size 2 touch diagonally, which is allowed in Fillomino.
- Figure 1 is valid, despite having a group (of size 4) formed without using any pre-filled number.

4 Variants

Fillomino puzzles come in different variants that affect the difficulty and solving process. There are two main variants of the puzzle that are commonly recognized.

4.1 Basic variant

In the basic version of the game, the puzzle provides at least one pre-filled cell for each group in the final solution. These clues guarrantee that every group is pre-determined: each cell's number represents the size of the group it is part of. No additional blocks other than those directly implied by the clues are needed, and every block contains an initially given number, so no new groups have to be created. The solution consists of expanding the initially given numbers into valid groups.

This variant simplifies the process of solving the puzzle, since all groups are represented from the start, eliminating the need to deduce the placement of missing groups. Figure 3 is an example where no new groups need to be created.

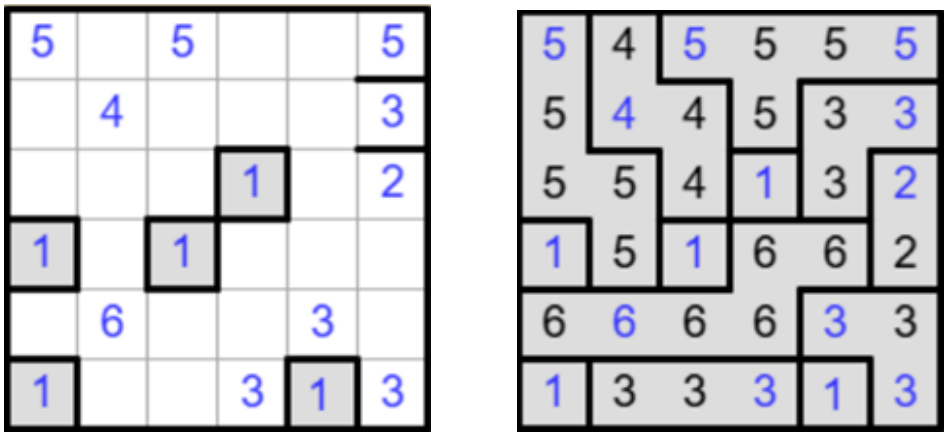


Figure 3: Basic variant, initial configuration and corresponding solution.

4.2 Challenging variant

In the more challenging version of Fillomino, there may exist groups in the final solution of the puzzle that do not contain a clue given at the start of the game. This means that the solver of the puzzle, does not only have to expand the existing numbers into valid groups, but also deduce the size and placements of possible hidden groups. For example, in Figure 4, two groups of size 3 have to be created to solve the puzzle. These hidden groups add a layer of complexity in order to solve the puzzle. This variant is widely seen as more difficult than the basic variant, as it requires the player to identify and create new blocks without any initial clues.

There is no predefined upper bound on the size of newly created groups, other than the size of the largest contiguous region of empty cells on the board that exists after the initial clues have been accounted for. As a result, the solving space is larger and the solver has to consider a broader range of possible group sizes.

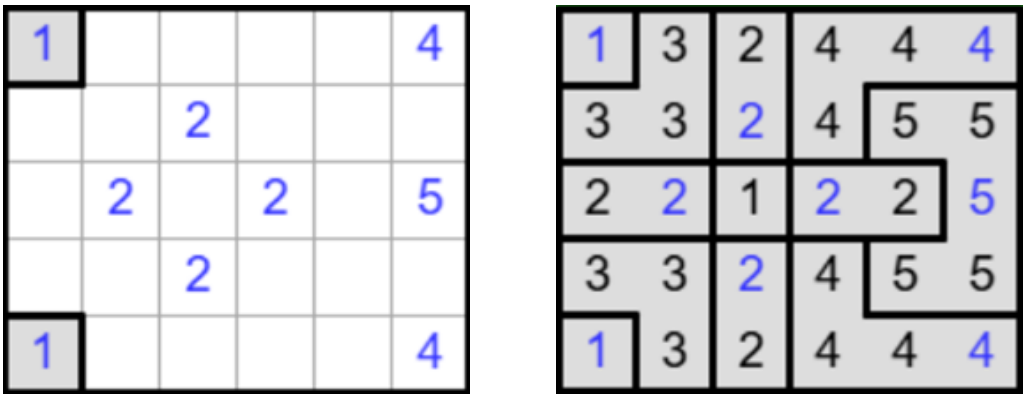


Figure 4: Challenging variant, initial configuration and corresponding solution.

These variants highlight the different aspects of the puzzle's complexity and affect what strategies we can use to solve the puzzle. Each variant requires different thinking skills and strategies, presenting new challenges that require different approaches in order to solve the puzzle. While the basic version focuses on expanding groups, the challenging version involves an additional feature, i.e. the possibility to create new groups, that changes how the puzzle is approached and solved.

5 Deterministic Solving Strategies for Fillomino

In order to solve Fillomino puzzles effectively without the immediate use of backtracking, a set of deterministic techniques can be used that are based on logical reasoning and local constraints derived from the puzzle's rules. These methods are especially useful for making safe moves early on in the solving process and for significantly reducing the search space before applying backtracking.

5.1 Loading Fillomino Puzzles

The Fillomino puzzle must be loaded into the program before any solving strategies can be applied. A JavaScript tool was developed to avoid manual input and copy puzzle configurations from online sources such as Angela and Otto Janko [JJ] and Puzzle Baron [Puz] and to transform them into a plain text file format that can be loaded by the program.

A C++ program was developed to read board configurations from a given file and save them into memory, to apply solving strategies in a consistent and automated way. The program serves as a main component of the solving process, enabling the user to experiment with different strategies and allowing for detailed tracking of the solving process.

A file begins with two numbers representing the height and width of the puzzle, followed by height $\times$ width numbers representing the board. Each number in the grid indicates the initial value of that cell in the puzzle. A `0` indicates an empty cell, while other numbers indicate a pre-filled value, specifying that the cell must belong to a group of that size.

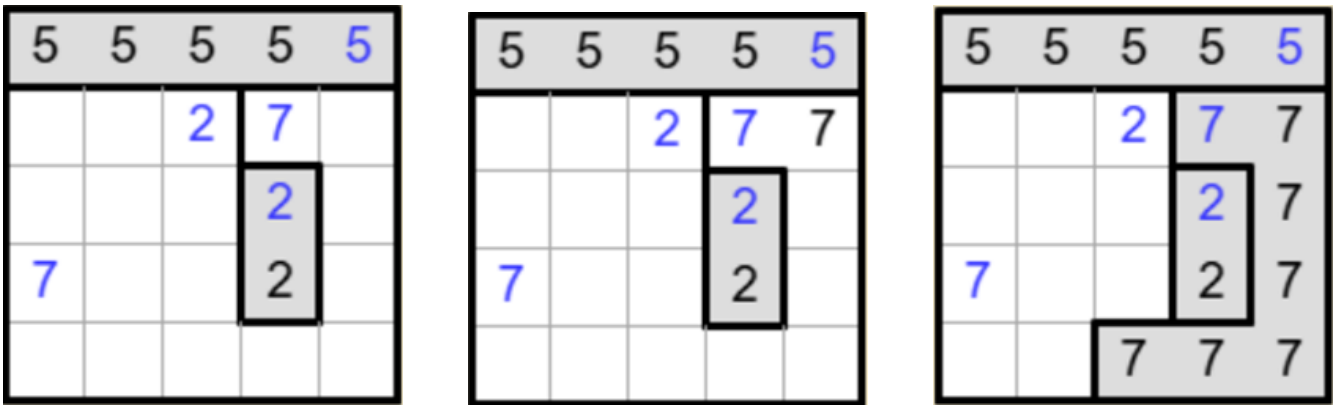
When a file is loaded, the program stores the values in a dynamically sized 2D vector. All non-zero entries are treated as fixed cells to ensure they remain unchanged throughout the entire solving process. Once the board is loaded, the existing groups are identified by using Breadth-First Search to find cells with the same non-zero number adjacent to one another (i.e., sharing an edge). These cells are then stored into a vector of group structures, each containing the group number and the coordinates of its cells. These initial groups may later be expanded or merged during the solving process. See Appendix A for the complete algorithm.

5.2 Single-Exit Group Strategy

One of the most effective deterministic rules involves identifying groups with only one possible cell to grow. If a region has not reached its required size (i.e, the number of connected cells does not match the number within those cells) and the group of cells only has one adjacent empty cell where it can expand, then that cell must logically be part of that same group. See Algorithm 1 for the pseudocode implementation of this strategy: The algorithm returns the single exit cell, if there is any, as well as the corresponding group number.

This technique does not require speculative reasoning, making it especially reliable and ideal to apply before attempting any other methods. We continuously scan the board for single-exit groups and expand them deterministically, which can often trigger further usage of this same rule repeatedly (see Figure 5).

This strategy does not take potential group changes into account that happen after expanding a cell, meaning it only considers the current adjacent empty cells for expansion. The strategy does not identify cases where a group appears to have multiple expansion paths, but logically only one is valid after considering how the board would change. For example, the incomplete group with number 2 in Figure 5 must expand to the left, since expanding downwards would result in an invalid board, but the single-exit group strategy fails to detect that because it ignores such future changes. Such cases are detected in the basic variant of Fillomino by the reachability-based number deduction strategy mentioned in Section 5.4 and after a single backtracking step in the challenging variant as described in Section 6, where future consequences are accounted for.



(a) (b) (c)

Figure 5: Repeated application of the single exit strategy.

Algorithm 1: Check for a Single-Exit Group

```
input  : exitCell (reference parameter to store the single exit cell)
output : group number with a single exit, or -1 if none found
foreach group in allGroups do
    tempExitCell ← invalid position
    exitCount ← 0
    foreach cell in group do
        foreach direction in {up, down, left, right} do
            neighbor ← cell moved in direction
            if neighbor is within bounds and is empty and tempExitCell is invalid then
                exitCount ← exitCount + 1
                tempExitCell ← neighbor
                if exitCount > 1 then
                    break
                end
            end
        end
        if exitCount > 1 then
            break
        end
    end
    if group is not full and exitCount = 1 then
        exitCell ← tempExitCell
        return group's number
    end
end
return −1
```

5.3 Structurally Forced Cells

Another deterministic technique that can be used is identifying cells that must belong to a certain group in every possible way the group could be completed. An incomplete group may have multiple ways to expand and meet its target size, but a particular empty cell may be included in every one of those configurations. When such cell exists, we can safely fill it with that group's number. The procedure for finding structurally forced cells for a given group can be seen in Algorithm 2. Single exits are a special case of structurally forced cells. Since they do not require any additional constraint checking, we treat them as a separate strategy as seen in Section 5.2.

To find these cells, we explore all possible expansions of groups using breadth-first search (BFS), marking which empty cells appear in all valid completion paths. These cells can then filled, as they are included in every possible scenario. Figure 6 provides an example of this strategy, with the letters in Figure 6b indicating the possible ways to complete the top-right group. The cells in columns 3 and 4 in the first row are filled with a 5, since they appear in every possible completion of the group.

(a) (b) (c)

Figure 6: Application of finding forced cells

Algorithm 2: Find structurally forced cells

Input: Incomplete group G, target size k
Output: Set of structurally forced cells F

$F \leftarrow \emptyset$
$S \leftarrow$ All cells currently in group G
$StartCells \leftarrow \{s \in S \mid s$ has at least one empty adjacent cell$\}$
$CompletionPaths \leftarrow \emptyset$
foreach *cell* $s \in StartCells$ **do**
  $Paths \leftarrow BFS(s, k)$; `// Finds all possible ways to complete the group to size` k`, starting from` s `and stores the required cells for each path`
  $CompletionPaths \leftarrow CompletionPaths \cup Paths$
end
$F \leftarrow$ Intersection of all sets in $CompletionPaths$
return F

5.4 Reachability-Based Number Deduction

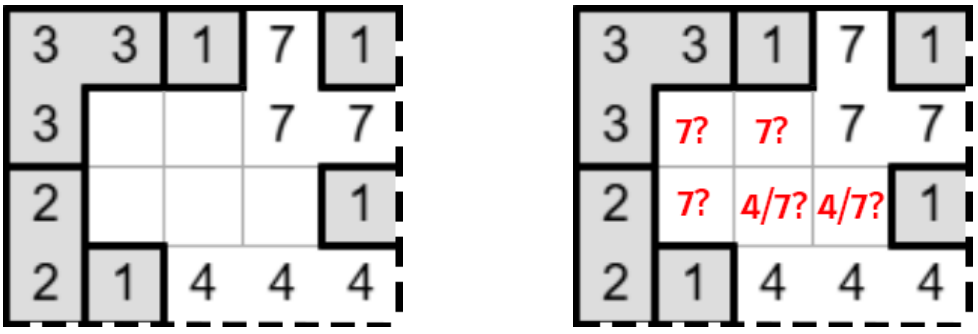
This strategy only works in the basic version of Fillomino, where no new blocks can be created and each cell must belong to an already existing numbered group. In this case, a powerful technique can be used that involves analyzing which groups can logically reach an empty cell. Specifically, for each empty cell, we determine which numbers could possibly be contained in this cell without violating any of the puzzle's constraints.

We define **reachability** as the ability of a group to expand from its currently filled cells to a specific empty cell on the board, without exceeding its available remaining size. Each group on the board has a fixed size. Since the group already occupies some cells, it can only expand into a limited number of additional empty cells until its target size has been met.

To check if a group can reach an unfilled cell, we can use breadth-first search (BFS) starting from any cell that currently belongs to the group. During this process, we move through empty cells as well as cells containing the same group number that may not yet be directly connected, since these separate groups could potentially become part of the same group in the solution. If the number of cell expansions needed to reach the target cell from the group's existing cells is less than or equal to the number of remaining cells that group can fill (i.e., the group's target size minus its current group size), then that cell is considered reachable by that group.

Alternatively, one could start in an empty cell, perform a BFS to find which groups (and hence what numbers) it can reach. In this case the maximum BFS-depth for each empty cell equals the largest number originally on the board. It seems that which method is more efficient depends on how many empty cells there are. In our implementation we chose to start from the group.

For each empty cell, all possible groups that can reach the cell are stored. If an empty cell is reachable by only one group number (i.e., by one or more groups sharing the same number), and assigning that number neither causes any group to exceed its allowed size nor prevents another group from being completed, then that number must be assigned to the cell. We explicitly exclude all other numbers that could reach the cell but would lead to an invalid board, such as causing a group to exceed its target size or making a group on the board impossible to complete.



(a) A snippet of a Fillomino puzzle

(b) Cells that are reachable by groups

Figure 7: Visualization of cells reachable by groups

6 Backtracking

While deterministic solving strategies such as forced cells, single-exit expansion and reachability-based number deduction are often useful to solve big sections of a Fillomino puzzle, some puzzles use configurations that cannot be solved by only using those strategies. In those situations, we can backtracking in order to try different number placements and backtrack from incorrect moves.

6.1 Approach

The algorithm starts by repeatedly trying the previously mentioned deterministic strategies in a loop until no further definitive moves can be made. When the Fillomino puzzle reaches such a state, the algorithm tries to find an empty cell on the board by going through the cells row by row and selecting the first empty cell it comes across. It iteratively tries every possible number for that cell that could fill it without violating the puzzle's constraints. Before each guess, the current board configuration is stored. The algorithm then places the guessed number and continues solving the board recursively, starting with the local strategies.

If we reach a state where we can not make any progress without violating the rules, the algorithm backtracks and restores the last saved configuration before we made a guess and then tries the next possible option, effectively exploring a search tree of possible states.

6.2 Measuring the Puzzle Difficulty: Backtracking Depth

A useful metric for determining the *difficulty* of a Fillomino puzzle involves measuring the depth of the backtracking process. Each time that the algorithm makes a guess and continues solving, a counter is increased to track how many successive guesses in a row the program has made before starting to backtrack.

A rough indicator of how *difficult* a puzzle is, is the maximum depth the algorithm has reached during the solving process. Puzzles that have a relatively low maximum depth can be solved by primarily using deterministic strategies for the majority of the solving process, requiring little to no backtracking at all. Deeper depths on the other hand indicate that more guessing is required in order to reach the puzzle's solution.

This metric also reflects on how a human might approach the puzzle. Puzzles with a low depth can be solved using local deductions for the most part, while puzzles with a higher depth often require trial and error when logic alone is insufficient.

7 SMT

SMT (Satisfiability Modulo Theory) is a well-known method used to solve logic and constraint problems. It builds upon SAT (Boolean Satisfiability Problem), where the objective is to assign TRUE or FALSE values to boolean variables in order to make a given propositional formula in Conjunctive Normal Form (CNF) true. The purpose of SAT is to check if there is a variable assignment that satisfies the formula. When such an assignment exists, the formula is called SATISFIABLE, otherwise it is UNSATISFIABLE.

Unlike SAT solvers, SMT solvers work with variables from various domains, including functions, integers and arrays. This allows them to solve more complex problems without translating all constraints into Boolean logic first. By combining Boolean reasoning with support for more complicated types and rules, SMT solvers can determine variable assignments that satisfy the entire formula, as long as such a solution exists.

Two well-known SMT solvers are Z3 [dMB] and Yices [Dut]. Z3 is developed by Microsoft Research, and Yices by SRI International. Both SMT solvers support a variety of logical theories, which are formal systems that define rules for reasoning about mathematical objects. These solvers are widely used to solve complicated problems involving logic and mathematics. Their reliability and efficiency make them widely used in many technology fields. Both solvers use a standard input format called **SMT-LIB**.

We construct logical formulas that capture all the puzzle's constraints, such as the group sizes and adjacency relationships between cells. These constraints are expressed in the SMT-LIB format. A `C++`-program was written to produce the formulas corresponding to the current Fillomino puzzle. Our approach is inspired by the Python implementation of Tom van Bussel [vB], who presents two solvers for the problem: one that constructs a spanning tree for each polyomino, and another one that uses a SAT-implementation of Warshall's algorithm. We use the approach that relies on constructing spanning trees for our encoder.

A **Spanning Tree** is a subgraph of an undirected graph $G = (V, E)$ that includes all of the graph's vertices. A spanning tree connects all the vertices while avoiding cycles by selecting a subset of the edges in E. Spanning trees are commonly used for algorithmic problem solving to represent connected components and simplify the connectivity constraints.

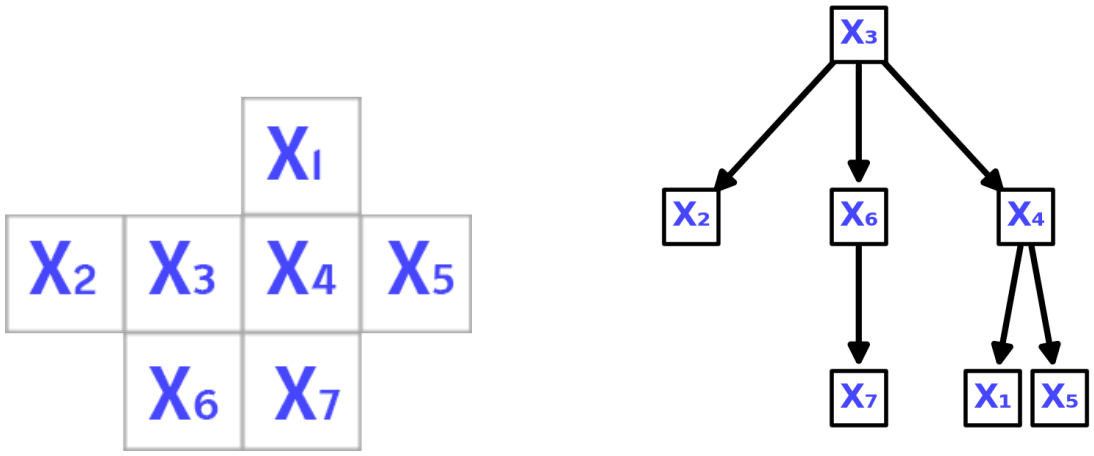
Once the formulas are generated, they will be written into a new file which can then be processed by an SMT solver such as Z3 or Yices. The solver's output can be stored and be used by the program to interpret the solution and reconstruct the completed puzzle.

8 Solving Fillomino using SMT

In order to encode a Fillomino puzzle for an SMT solver, we represent the board as a collection of cells, each having a unique identifier/index. We can define variables for these cells and capture the puzzle's constraints. We number the cells on the board starting from the top-left corner (0,0). For a board with a width of W, the cell at row r and column c has the index:

$$x = r \times W + c$$

Throughout this section, x and y are used to refer to these cell identifiers. To model the relationship within each group in the puzzle, we create a spanning tree starting at a single cell called the root. We use directed edges in the SMT encoding to represent these connections, keeping track which cells belong to the same block. Each group in the solution is represented as a tree and all these trees together form a forest over the whole board. Figure 8 illustrates a Fillomino block of size 7 alongside a possible spanning tree. The following subsections describe the variables and constraints used in the SMT encoding. The full C++ implementation of the SMT-encoder can be found in Appendix B.



(a) (b)

Figure 8: A Fillomino block and a possible spanning tree over its cells

8.1 Variable Declarations

We start by defining variables that represent the current configuration of the puzzle in order to encode the constraints of a Fillomino puzzle. Additionally, we introduce variables that describe the relationship between cells.

- **Number variables** N_x are used to store the number assigned to each cell x

  `(declare-fun` n_x`()Int)`

- **Size variables** S_x indicate the total number of cells in the subtree that cell x is the root of.

  `(declare-fun` s_x`()Int)`

- **Edge variables** $E_{x,y}$ are boolean variables that indicate whether there exists a directed edge from cell x to its adjacent cell y. Tree structures are being formed by these edges to capture how cells within the same group are connected to each other.

  `(declare-fun` $e_{x,y}$ `() Int)`

- **Root variables** R_x for each cell x, indicating which cell is the root of the tree for the group that cell x belongs to.

  `(declare-fun` r_x `() Int)`

In Figure 8 we give an example of a polyomino and its corresponding spanning tree. Assuming that this polyomino of size 7 is a completed part of a correct solution, let $a = X_3$, $b = X_4$, $c = X_6$ and $d = X_7$ we would have that $N_a = 7$ (otherwise N_a would be > 7). Furthermore, $S_b = 3$, since the subtree rooted at cell X_4 consists of 3 nodes (i.e., cell X_4, X_1 and X_5). Additionally, the root variable $R_x = a$ for all cells x, because they are all part of the same tree. Finally, the edge variable $E_{c,d} = 1$ indicating a directed edge from cell X_6 to cell X_7.

8.2 Edge constraints

Each edge variable $E_{x,y}$ represents whether there is a directed edge from cell x to a vertically or horizontally neighboring cell y. Since an edge can either exist or not, every $E_{x,y}$ must be a boolean value, either a 1 (edge present) or a 0 (no edge). Thus edge variables are only declared between neighboring cells.
This boolean constraint is essential as it ensures that edge values accurately represent a graph structure. Formally, the boolean constraint is expressed as:

$$\forall x, \forall y \in adj(x), \quad E_{x,y} \in \{0, 1\}$$

This condition can be asserted in the following SMT encoding:

`(assert (or (=` $e_{x,y}$ `0) (=` $e_{x,y}$ `1)))`

8.3 No bidirectional edges

In Fillomino, groups are formed by connected regions of cells. To model this connectivity using SMT, directed edges are declared between neighboring cells. However, bidirectional edges between adjacent cells must be disallowed to ensure the structure remains a tree. Meaning that if there exists an edge from x to y, then an edge in the opposite direction from y to x is not allowed to exist at the same time. This is formally expressed as:

$$\forall x, \forall y \in adj(x), \quad E_{x,y} + E_{y,x} \leq 1$$

In SMT-Lib format this is expressed as:

`(assert (<= (+` $e_{x,y}$ $e_{y,x}$ `) 1))`

8.4 At most one incoming edge

Every node in a tree structure should have at most one incoming edge. In order to enforce this characteristic, we have to make sure that each cell x has at most one incoming edge from its adjacent cells. Formally expressed as:

$$\forall x, \quad \sum_{y \in \text{adj}(x)} E_{y,x} \leq 1$$

This rule ensures that every cell is part of only one group, which is a necessary requirement for forming a tree, and that the connections between cells from the same group form a proper tree. In SMT-Lib format, this is written as:

`(assert (<= (+` $e_{y_1,x}$ $e_{y_2,x}$ `...) 1))`

where the sum is over all the cells y_i neighboring x. Each cell has at most four neighboring cells, meaning the summation involves at most four terms.

8.5 Initial clues

Fillomino puzzles contain pre-filled numbers (clues), which specify a number k assigned to a cell x. These clues fix the values of specified cells throughout the final solution:

$$\forall (x, k) \in \text{input clues}, \quad N_x = k$$

and we can create this in SMT-Lib format as:

`(assert (=` $\texttt{n}_x$ k`))`

8.6 Root cells size

For root cells (i.e., cells without any incoming edges), the size variable S_x must be equal to the number N_x assigned to that cell:

$$\sum_{y \in \text{adj}(x)} E_{y,x} = 0 \implies S_x = N_x$$

and can be denoted in SMT as:

`(assert (=> (= (+` $e_{y_1,x}$ $e_{y_2,x}$ `...) 0) (=` s_x n_x`)))`

8.7 Group size propagation

The size variable S_x denotes the number of cells that can be reached in the subtree from cell x by following the outgoing edges, including cell x itself. The root of a tree is the starting cell of the entire block, meaning that only its size S_x represents the size of that block. The size of these cells is computed recursively through the edges:

$$S_x = 1 + \sum_{y \in \text{adj}(x)} E_{x,y} \cdot S_y$$

Intuitively, the size of a block starting at root x is equal to one (counting x itself) plus the sum of the sizes of its adjacent cells y, where x points to. These neighbors can be thought of as the "children" of x in the block's structure. This rule allows the solver to compute group sizes by combining size values in a bottom-up manner throughout the tree.
This can be expressed in SMT as:

$$\begin{aligned}
&\texttt{(assert (= } \mathtt{s}_x \texttt{ (+ 1}\\
&\texttt{(ite (= } \mathtt{e}_{x,y_1} \texttt{ 1) } \mathtt{s}_{y_1} \texttt{ 0)+}\\
&\texttt{(ite (= } \mathtt{e}_{x,y_2} \texttt{ 1) } \mathtt{s}_{y_2} \texttt{ 0)+}\\
&\texttt{(ite (= } \mathtt{e}_{x,y_3} \texttt{ 1) } \mathtt{s}_{y_n} \texttt{ 0)+}\\
&\texttt{ (ite (= } \mathtt{e}_{x,y_4} \texttt{ 1) } \mathtt{s}_{y_n} \texttt{ 0)}\\
&\texttt{)))}.
\end{aligned}$$

Here, `ite` is an if-then-else construct, which includes S_y, if and only if there exists an edge from x to y.

8.8 Edge implies same numbers

When two cells are connected by an edge, regardless of the direction of that edge, they must contain the same number as they belong to the same group. Formally if there exists an edge from cell x to cell y, then their numbers must be the same:

$$y \in adj(x) \text{ and } E_{x,y} = 1 \implies N_x = N_y$$

We can assert this in SMT-Lib format with:

$$\texttt{(assert (=> (= } \mathtt{e}_{x,y} \texttt{ 1) (= } \mathtt{n}_x \texttt{ } \mathtt{n}_y \texttt{)))}$$

8.9 Root cells

Cells that do not have an incoming edge are considered roots of their respective polyomino block. The root variable R_x for such a cell x must be equal to its own index:

$$\sum_{y \in \text{adj}(x)} E_{y,x} = 0 \implies R_x = x$$

We can write this relation in SMT as:

$$\texttt{(assert (=> (= (+ } e_{y_1,x} \texttt{ } e_{y_2,x} \texttt{ ...) 0) (= } r_x \texttt{ } x\texttt{)))}$$

8.10 Same number implies same root

Finally, neighboring cells that contain the same number must belong to the same group and must therefore share the same root for that group:

$$y \in \text{adj}(x) \text{ and } N_x = N_y \implies R_x = R_y$$

This is encoded in SMT format as:

$$\texttt{(assert (=> (= } \mathtt{n}_x \texttt{ } \mathtt{n}_y \texttt{) (= } \mathtt{r}_x \texttt{ } \mathtt{r}_y \texttt{)))}$$

9 Experiments

In order to measure the performance of the previously mentioned solving methods, we conducted a series of experiments. For the basic variant of Fillomino we compared two solving methods. The first approach is a custom solver that uses the deterministic strategies mentioned in Section 5 combined with backtracking when no certain move can be found. The second method uses an SMT solver, where the puzzles are encoded and solved using the Z3-solver. For the experiments on the challenging variant of Fillomino, we only tested the performance of an SMT solver, since our custom solver could not find solutions in a reasonable amount of time. The puzzles used in the following experiments, are taken from PuzzleBaron [Puz] (for the basic variant) and from a collection of Fillomino puzzles published by Angela and Otto Janko [JJ] (for the challenging variant). All puzzles have a unique solution.

9.1 Experiments on the Basic Variant

The experiments were performed on Fillomino boards with various sizes, ranging from 5×10 up to a size of 20×20. Ten instances of the puzzles were tested for each board size, yielding sixty instances altogether. Approximately 60% of the cells on average in each board were empty. Board instances for each size were identified with a unique number. We measured the solving time for both methods. The maximum recursion depth was also recorded for the custom solver. The custom solver and board instances used in the experiments can be found at [Rya].

The solving times for different board sizes and instances are presented in Figures 9, 10, and 11. The horizontal axis in each graph represents the board number, and the vertical axis represents the time in seconds it took to solve that board instance. Two data points are plotted for every instance: one for solving the puzzle using SMT and the other for solving the board using deterministic strategies (and backtracking if necessary).

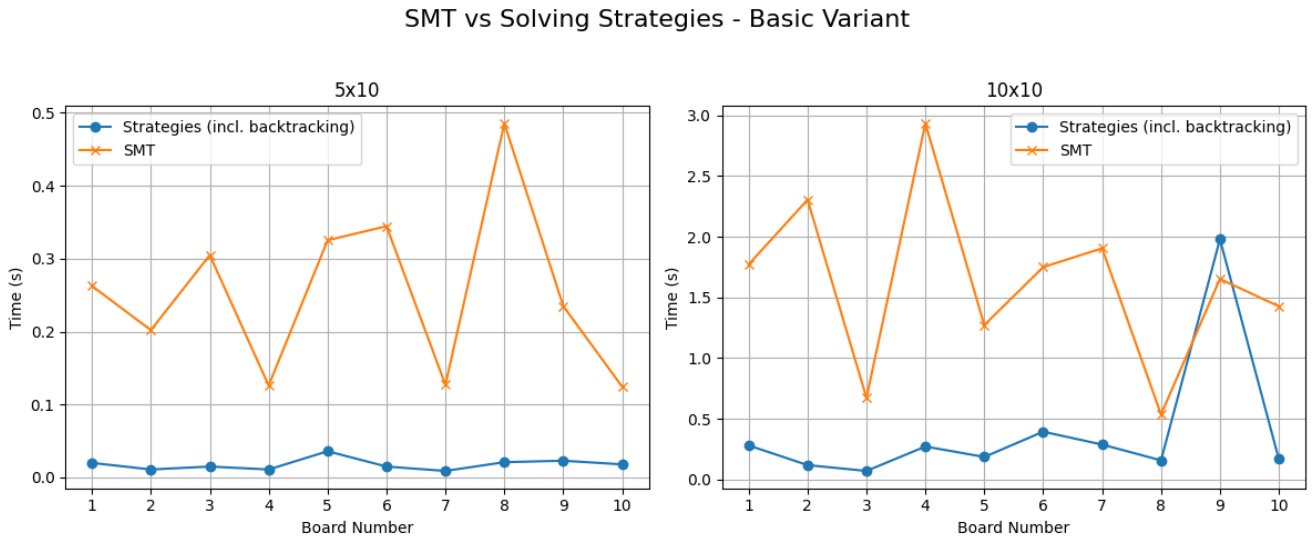


Figure 9: Solving time comparison for smaller boards (5×10 and 10×10).

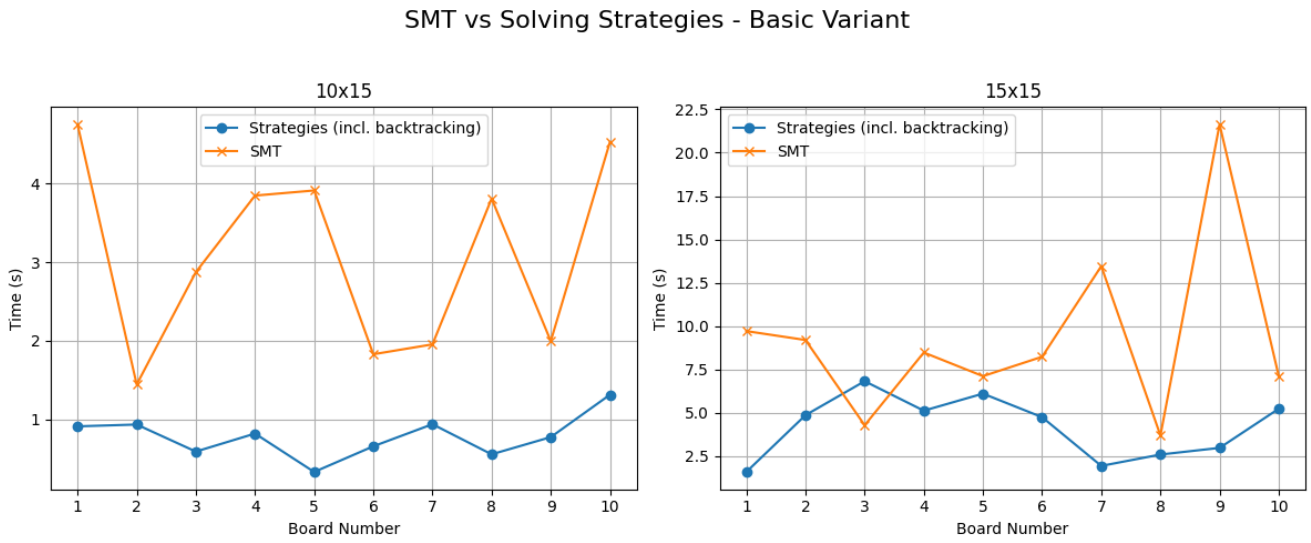


Figure 10: Solving time comparison for medium-sized boards (10×15 and 15×15).

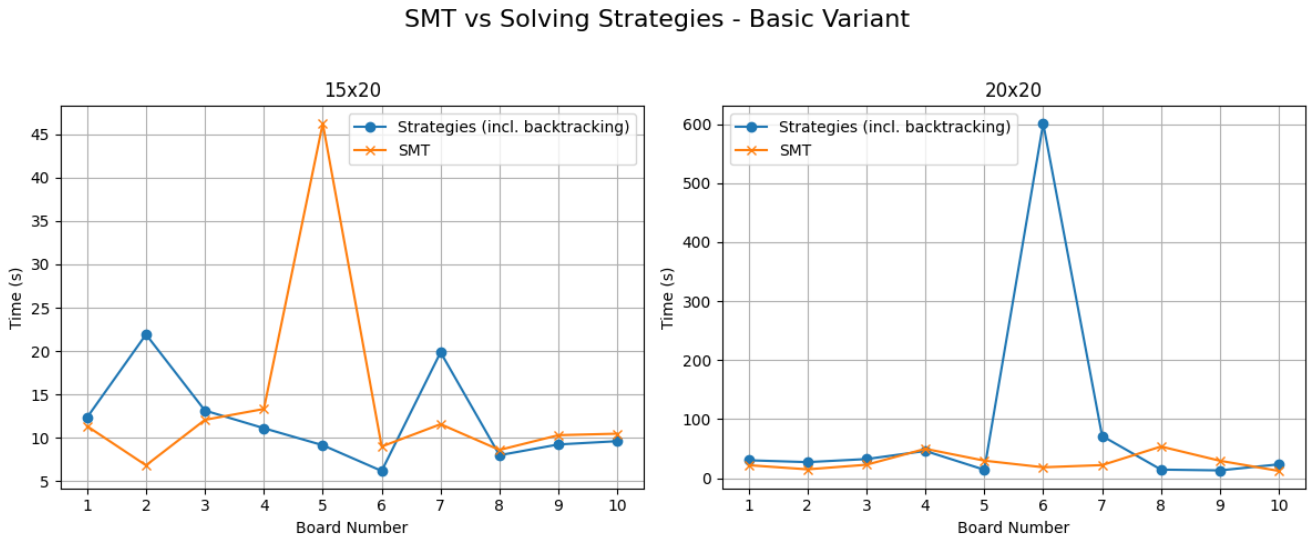


Figure 11: Solving time comparison for larger boards (15×20 and 20×20).

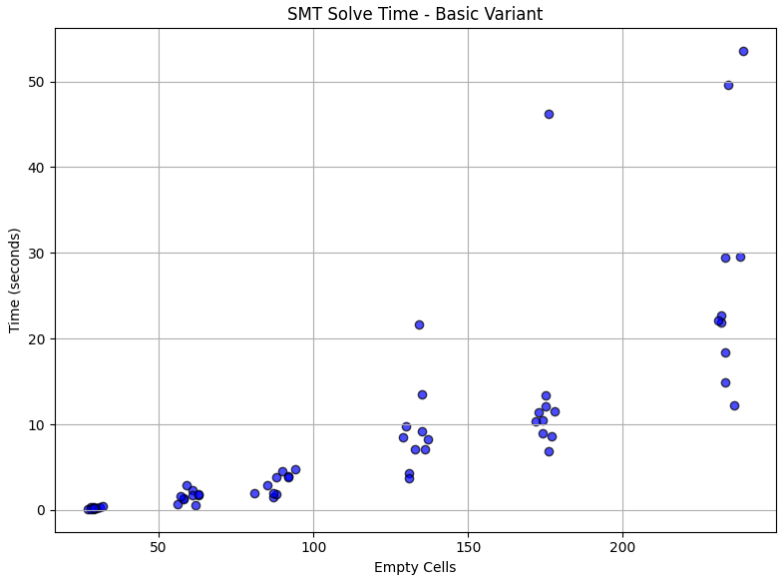


Figure 12: SMT solving times based on the number of empty cells - Basic Variant

The results show that the solver using deterministic strategies outperforms the SMT-based solver on average for small and medium sized puzzles (up to around 10×15) in terms of execution time, with only a single outlier. In 93% of all the cases, the Fillomino puzzles could be entirely solved using the deterministic strategies only, without the use of backtracking.

The maximum recursion depth remained shallow overall in puzzles where backtracking was required, and the solver still outperformed the SMT-based solver in most cases, with only a few exceptions such as board 6 with size 20×20, which reached a maximum depth of 17 using backtracking. As the sizes of the board increases, the performance gap between the two approaches begins to narrow. On boards with sizes 15×15 and 15×20, the puzzles contain a significantly higher number of empty cells. This increase in empty cells results in a more extensive search space, since we need to explore many more possibilities to fill empty cells. This leads to longer solving times, as seen in Figure 12. Note that the number of empty cells scales roughly with the size of these puzzles. For puzzles of size 20×20, the difference in performance is less clear, with neither method consistently outperforming the other.

Overall, the experiments show the strengths of both solving methods for the basic variant. The deterministic strategy-based solver is very effective on small puzzles with few empty cells, when solutions can be found without extensive guessing. The SMT solver is generally slower on small puzzles, but is more reliable on larger puzzles and offers improved performance in worst-case scenarios.

9.2 Experiments on the Challenging Variant

Additionally, the performance of the SMT solver was evaluated on 100 (10×10) Fillomino puzzles from Janko [JJ], representing the challenging version of the game. Figure 13 shows the solving times relative to the number of empty cells in these puzzles. Not surprisingly, the results indicate that there is a correlation between the number of empty cells and the solving time, with puzzles that contain more empty cells generally taking longer to solve. Although the solving times for the challenging variant of Fillomino tend to be higher than the basic variant at equal sizes, the trend in Figure 13 suggests that the number of empty cells strongly influence the solving time regardless of the variant. Fillomino puzzles with more than 60 empty cells tend to require significantly more time to solve than those with fewer empty cells.

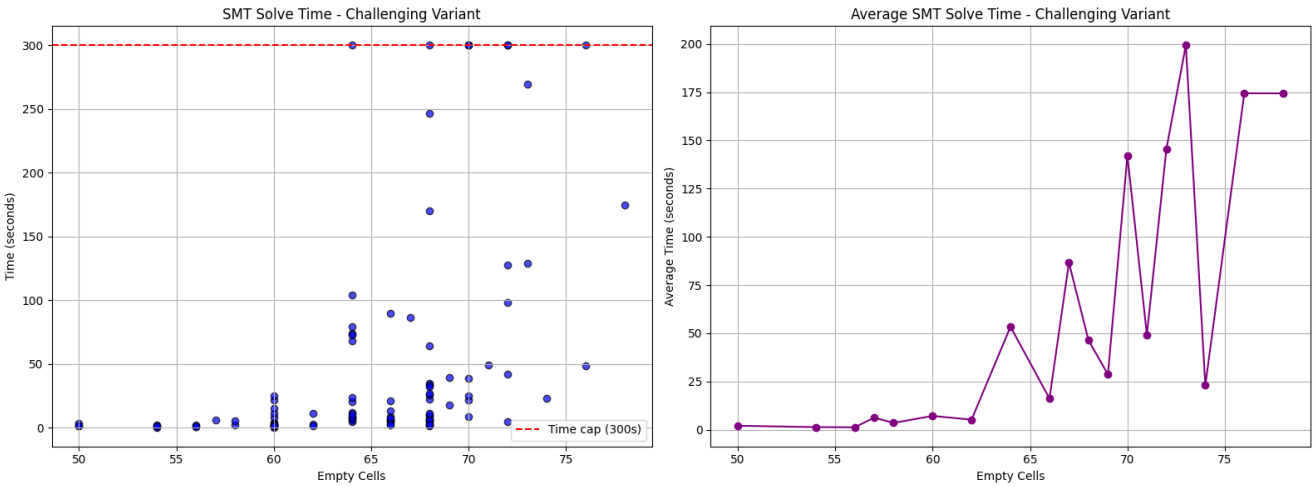


Figure 13: SMT solving times based on the percentage of empty cells for 10×10 puzzles - Challenging Variant

In conclusion, experiments on the challenging variant of the game indicate that the performance of the SMT solver is largely influenced by the percentage of empty cells. These results suggest that a solver combining the two methods, using deterministic strategies early on until no further progress can be made and then using an SMT solver on a puzzle that is already partially filled, could provide a more efficient approach for a wide range of Fillomino puzzles.

10 Conclusions and Further Research

In this thesis, we explored algorithmic strategies for solving Fillomino, focusing on deterministic methods as well as the use of Satisfiability Modulo Theories (SMT).

We implemented several deterministic strategies in order to help solve the puzzle, which are checking groups for single-exit cells, detecting structurally forced cells and deducing the values in empty cells by analyzing which groups could possibly expand to these cells. These strategies are sufficient to solve most instances of the basic variant of Fillomino. However, some of these puzzles still require the use of backtracking to be fully solved. Experiments on the challenging variant show that the performance of the SMT solver is strongly influenced by the percentage of empty cells on the board.

Additionally, an SMT encoder was created to encode Fillomino boards and the puzzle's constraints. The encoding captures adjacency rules and size constraints of polyominoes as logical formula's, enabling SMT solvers such as Z3 [dMB] and Yices [Dut] to find solutions.

We performed experiments to compare the performance of a custom solver that uses deterministic strategies and backtracking with that of an SMT solver across different puzzle instances. The results indicate that the custom solver is generally faster on smaller puzzles, while the performance difference decreases as the puzzle size increases. Furthermore, the experiments indicate that the number of empty cells on a board strongly influences the solving time of the SMT solver.

Many opportunities for future research still remain. Improving the SMT encoding could help decrease the time and resources needed to solve Fillomino puzzles. A combination of deterministic strategies and SMT solving could be used to develop a more efficient solver that takes advantage of the strengths of both approaches. Using heuristics to choose which empty cells to fill first instead of randomly choosing an option during backtracking could help speedup the solving process. Additionally, developing methods to generate Fillomino puzzles of varying difficulty and varying number of empty cells could help with the evaluation of solvers. Furthermore, the effect of the deterministic methods could be measured by trying all possible combinations of using each of the three solving strategies, in combination with backtracking.

References

[AB06] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2006.

[dK21] Gerhard Van der Knijff. Solving and generating puzzles with a connectivity constraint. Bachelor's thesis, Radboud University, Nijmegen, The Netherlands, January 2021.

[dMB] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. https://github.com/Z3Prover/z3. Accessed : 14-05-2025.

[Dut] Brandon Dutertre. Yices. https://yices.csl.sri.com. Accessed : 14-05-2025.

[HD09] Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation*. A K Peters/CRC Press, 1st edition, 2009.

[Hes25] Niels Heslenfeld. Solving and Generating Fobidoshi Puzzles. Bachelor's thesis, Leiden University, Leiden, The Netherlands, 2025.

[JJ] Angela Janko and Otto Janko. Fillomino. https://www.janko.at/Raetsel/Fillomino/index.htm. Accessed : 24-02-2025.

[Puz] PuzzleBaron. Fillomino. https://fillomino.puzzlebaron.com. Accessed : 13-03-2025.

[RN10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[Rya] Ryan Behari. Fillomino Solver: A repository of Fillomino puzzles and solvers. https://github.com/DarkSness420/FillominoSolver.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2013.

[vB] Tom van Bussel. SMT Fillomino Solvers. https://github.com/tomvbussel/fillomino. Accessed : 22-05-2025.

[vdG25] Thijs van de Griendt. The Complexity of Two Polyomino Region Puzzles. Bachelor's thesis, Radboud University, Nijmegen, The Netherlands, 2025.

[YS03] Takayuki YATO and Takahiro SETA. Complexity and Completeness of Finding Another Solution and Its Application to Puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A, 05 2003.

A Finding Groups

```
//find groups in the board and store them
void findAndStoreGroups() {
    globalGroups.clear();
    std::vector<std::vector<bool>>
    visited(Height, std::vector<bool>(Width, false));


    for (int i = 0; i < Height; i++) {
        for (int j = 0; j < Width; j++) {
            if (board[i][j] != 0 && !visited[i][j]) {
                //BFS to find group
                queue<pair<int, int>> q;
                vector<pair<int, int>> groupCells;
                int number = board[i][j];

                q.push({i, j});
                visited[i][j] = true;

                while (!q.empty()) {
                    auto [row, col] = q.front();
                    q.pop();
                    groupCells.push_back({row, col});

                    for (const auto& dir : DIRECTIONS) {
                        int newRow = row + dir[0];
                        int newCol = col + dir[1];

                        if (isValid(newRow, newCol) &&
                        !visited[newRow][newCol] &&
                        board[newRow][newCol] == number) {
                            visited[newRow][newCol] = true;
                            q.push({newRow, newCol});
                        }
                    }
                }

                //store into global var
                Group newGroup = {number, groupCells};
                globalGroups.push_back(newGroup);
            }
        }
    }
}
```

B SMT Encoder Implementation

```
class FillominoSMTSolver {
public:
    int rows, cols;

    FillominoSMTSolver() {}

    vector<pair<int, int>> adj(int row, int col) {
        vector<pair<int, int>> neighbors;
        if (row - 1 >= 0) neighbors.emplace_back(row - 1, col);
        if (row + 1 < rows) neighbors.emplace_back(row + 1, col);
        if (col - 1 >= 0) neighbors.emplace_back(row, col - 1);
        if (col + 1 < cols) neighbors.emplace_back(row, col + 1);
        return neighbors;
    }

    string solve(int r, int c, const vector<tuple<int, int,
    int>>& nums) {
        ostringstream oss;

        oss << "(set-option :print-success false)" << "\n";
        oss << "(set-logic QF_UFLIA)" << "\n";

        rows = r;
        cols = c;

    //we define edges e_{x}_{y} between cells. 1 if there exists
    one between two cells, else 0
        for (int row1 = 0; row1 < rows; row1++) {
            for (int col1 = 0; col1 < cols; col1++) {
                int x = row1 * cols + col1;
                for (auto [row2, col2] : adj(row1, col1)) {
                    int y = row2 * cols + col2;
                    oss << "(declare-fun e_" << x << "_" << y <<
    " () Int)" << "\n";
                    oss << "(assert (or (= e_" << x << "_" << y
    << " 0) (= e_" << x << "_" << y << " 1)))" << "\n";
                }
            }
        }

    //We cant have an edge between cells in both directions,
    only one direction.
        for (int row1 = 0; row1 < rows; row1++) {
            for (int col1 = 0; col1 < cols; col1++) {
                int x = row1 * cols + col1;
```

```
                for (auto [row2, col2] : adj(row1, col1)) {
                    int y = row2 * cols + col2;
                    oss << "(assert (<= (+ e_" << x << "_" << y
<< " e_" << y << "_" << x << ") 1))" << "\n";
                }
            }
        }

        //each cell is allowed to have at most one incoming edge
(tree structure)
        for (int row1 = 0; row1 < rows; row1++) {
            for (int col1 = 0; col1 < cols; col1++) {
                int x = row1 * cols + col1;
                oss << "(assert (<= (+" << "\n";
                for (auto [row2, col2] : adj(row1, col1)) {
                    int y = row2 * cols + col2;
                    oss << "e_" << y << "_" << x << "\n";
                }
                oss << ") 1))" << "\n";
            }
        }

        //number variable for each cell
        for (int row = 0; row < rows; row++) {
            for (int col = 0; col < cols; col++) {
                int x = row * cols + col;
                oss << "(declare-fun n_" << x << " () Int)" <<
"\n";
            }
        }

        //If we already know if certain cells contain a certain
number, we assign them.
        for (auto [i, j, k] : nums) {
            int x = i * cols + j;
            oss << "(assert (= n_" << x << " " << k << "))" <<
"\n";
        }

        //size constraint for regions
        for (int row = 0; row < rows; row++) {
            for (int col = 0; col < cols; col++) {
                int x = row * cols + col;
                oss << "(declare-fun s_" << x << " () Int)" <<
"\n";
            }
        }

```

```
    //s_x =  (sum of the sizes of the neighbours connected from this cell) + 1.
        for (int row1 = 0; row1 < rows; row1++) {
            for (int col1 = 0; col1 < cols; col1++) {
                int x = row1 * cols + col1;
                oss << "(assert (= s_" << x << " (+ 1" << "\n";
                for (auto [row2, col2] : adj(row1, col1)) {
                    int y = row2 * cols + col2;
                    oss << "(ite (= e_" << x << "_" << y << " 1) s_" << y << " 0)" << "\n";
                }
                oss << ")))" << "\n";
            }
        }

    //if there are no incoming edges, s_x has to equal n_x
        for (int row1 = 0; row1 < rows; row1++) {
            for (int col1 = 0; col1 < cols; col1++) {
                int x = row1 * cols + col1;
                oss << "(assert (=> (= (+" << "\n";
                for (auto [row2, col2] : adj(row1, col1)) {
                    int y = row2 * cols + col2;
                    oss << "e_" << y << "_" << x << "\n";
                }
                oss << ") 0) (= s_" << x << " n_" << x << ")))" << "\n";
            }
        }

    //all cells in the same region must have the same number
        for (int row1 = 0; row1 < rows; row1++) {
            for (int col1 = 0; col1 < cols; col1++) {
                int x = row1 * cols + col1;
                for (auto [row2, col2] : adj(row1, col1)) {
                    int y = row2 * cols + col2;
                    oss << "(assert (=> (= e_" << x << "_" << y << " 1) (= n_" << x << " n_" << y << ")))" << "\n";
                }
            }
        }

    //declare root variable for each cell
        for (int row = 0; row < rows; row++) {
            for (int col = 0; col < cols; col++) {
                int x = row * cols + col;
                oss << "(declare-fun r_" << x << " () Int)" << "\n";
            }
```

```
126         }
127
128     //cells with no incoming edges are roots.
129         for (int row1 = 0; row1 < rows; row1++) {
130             for (int col1 = 0; col1 < cols; col1++) {
131                 int x = row1 * cols + col1;
132                 oss << "(assert (=> (= (+" << "\n";
133                 for (auto [row2, col2] : adj(row1, col1)) {
134                     int y = row2 * cols + col2;
135                     oss << "e_" << y << "_" << x << "\n";
136                 }
137                 oss << ") 0) (= r_" << x << " " << x << "))))" <<
    "\n";
138             }
139         }
140
141     //connected cells in the same region, must have the same
    root.
142         for (int row1 = 0; row1 < rows; row1++) {
143             for (int col1 = 0; col1 < cols; col1++) {
144                 int x = row1 * cols + col1;
145                 for (auto [row2, col2] : adj(row1, col1)) {
146                     int y = row2 * cols + col2;
147                     oss << "(assert (=> (= n_" << x << " n_" <<
    y << ") (= r_" << x << " r_" << y << ")))" << "\n";
148                 }
149             }
150         }
151
152         oss << "(check-sat)" << "\n";
153
154         for (int row = 0; row < rows; row++) {
155             for (int col = 0; col < cols; col++) {
156                 int x = row * cols + col;
157                 oss << "(get-value (n_" << x << "))" << "\n";
158             }
159         }
160
161         return oss.str();
162     }
163 };
```