On the Complexity of Generalized Roller Splat!

Sebastian Zhu, William Yue, and Vincent Fan

Abstract. In the popular mobile game *Roller Splat!*, the player controls a paintball rolling around a rectangular board, with a number of blocked out regions. On any given move, the paintball moves in any of the four cardinal directions until it hits a barrier or the edge of the board. As the paintball rolls, it paints the board underneath it, and the objective is to paint every open square. In this paper, we show that if given an arbitrary board, it is computationally "easy" to determine whether or not the whole board can be covered by showing there exists a polynomial time algorithm (the problem is in P). We also explore three other similar 2-dimensional variants and show they are also in P, presenting code implementations. Then we consider the more interesting 3-dimensional case and prove it is computationally "hard" by constructing a novel reduction from 3-SAT to show it is NP-complete.

Keywords: Ice-sliding game, Complexity theory, Algorithms, Turing reduction

.

Contents

1 Introduction

The ice-sliding game is a recurring motif in video games, as seen in the popular mobile games *Roller Splat!* (which involves coloring in blank squares with a ball of paint) and *Niwashi* (which involves cutting squares of grass with the namesake tool). These games have been heavily featured on Facebook and Instagram advertisements. Furthermore, the theme is also present in puzzle elements of various RPG games, such as those of the popular and world-renowned Pokémon franchise. For example, the player must play a version of the ice-sliding game at the seventh gym in the Sinnoh region, to battle the Ice-type Gym Leader; the player must also solve a similar puzzle to catch a legendary Pokémon, Regigigas, at Snowpoint Temple [snoa, snob]. In all of these games, the user is controlling a maximal sliding agent, such as the ball of paint, the grass cutting tool, or the player's sprite. Furthermore, players are trying to accomplish a set of tasks on a gridded game board, whether it be moving from one point to another, or traversing the entire board. This ice-sliding theme also manifests itself in real life: Tilt-Mazes, a physical puzzle where the player tries to maneuver a small ball, also feature maximal sliding agents [cli]. In fact, these tilt puzzles exist in both 2-dimensional and 3-dimensional forms. In the former, the bead rolls around on one plane, while in the latter, the bead is confined to a box. Examples of both devices are shown in Figures 3 and 4, respectively [3dt].

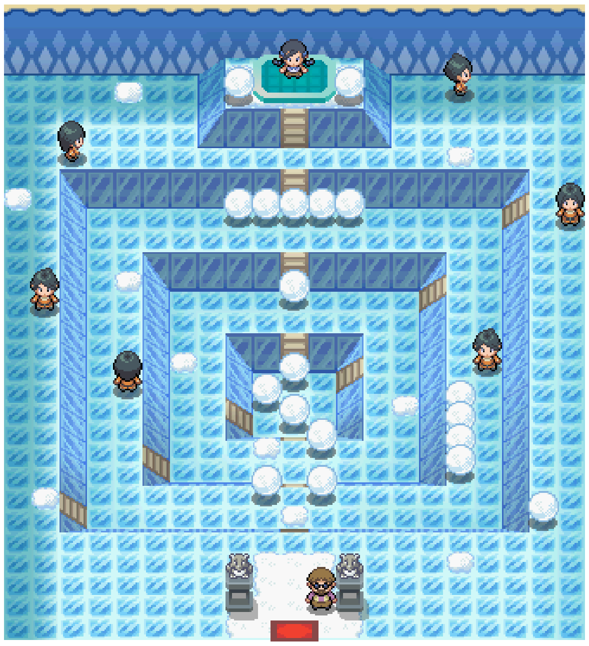


Fig. 1: The player's goal is to reach the gym leader (on raised podium) from the red entrance. However, the player cannot stop when moving on ice. The player can only stop moving in one direction when halted by the well, another sprite, a set of stairs, or a patch of dry snow. [snoa]

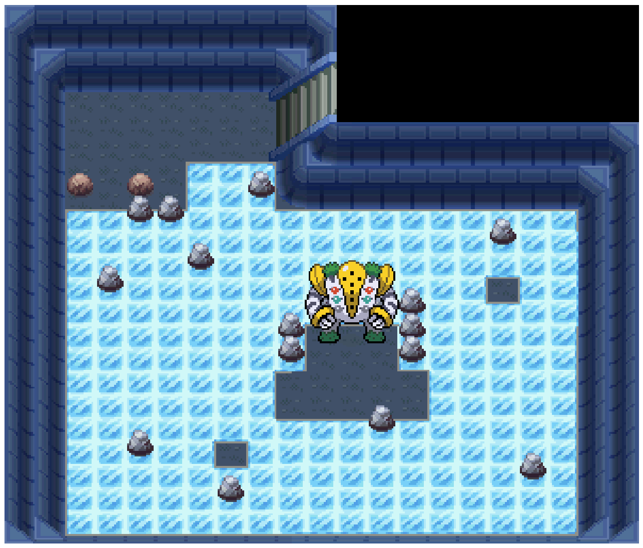


Fig. 2: The Player's goal is to reach the direct front of the statue, in order to summon a legendary Pokémon. However, once again, the player's avatar will not stop sliding on ice unless stopped by a wall or a rock. [snob]

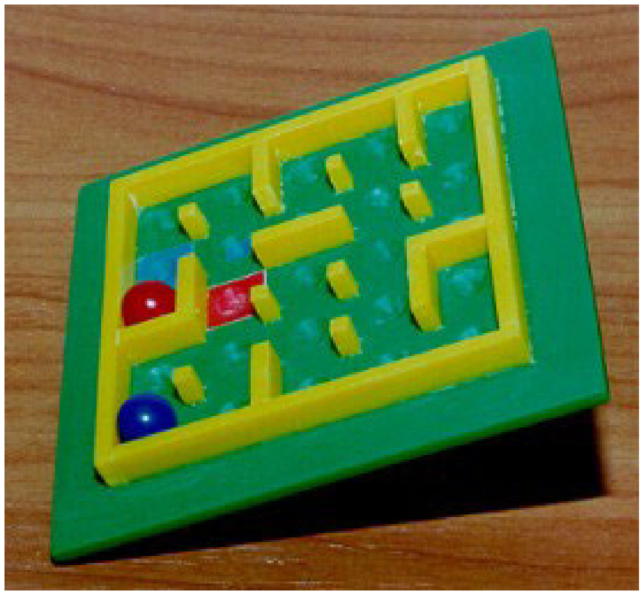


Fig. 3: This figure shows a real life tilt puzzle realized by famous puzzle maker Oskar van Deventer. The player tries to roll the bead to a desired location by tilting the board. Unless the user is very dextrous, the bead also acts as a maximal sliding agent, as it can only be stopped when it hits a wall. [cli]

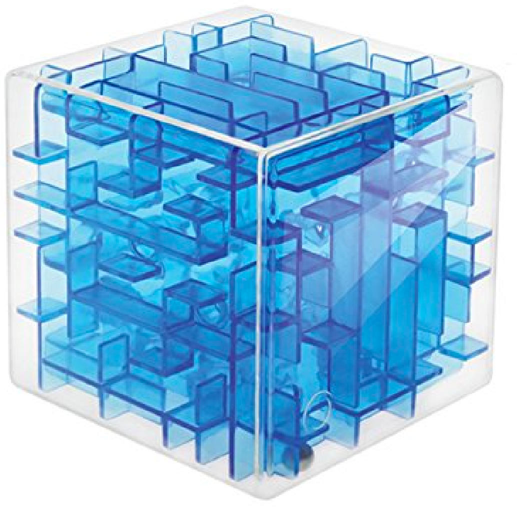


Fig. 4: The above shows a 3d bead puzzle sold by the ABBYFRANK store on Amazon. Once again, the bead is considered to be a maximal sliding agent as it will only be stopped by the presence of a barrier. [3dt]

There have been significant mathematical studies into the realm of recreational games. Certainly, many mathematicians have long been captured by the beauty of Chess [BC39]. More recently, Demaine et al. have analyzed the complexity of numerous video games, such as the classic *Tetris* and *Super Mario Brothers* [DHLN03,ADGV15]. Specifically, many puzzles involving moving agents on grid-based game boards have received thorough mathematical treatment. In the category of sliding puzzles, these include the famous *15-puzzle* and childhood favorite *Rush Hour* [RW90,FB02]. There are also a variety of well-studied games where the player actively pushes around obstacles around on the game board, including *Push* and *PushPush*. Many of these games have been shown to be NP- or PSPACE-complete [DHH04].

On the other hand, games featuring maximal sliding agents have not been nearly as well studied through the years. Tejeda published a 2014 paper about the complexity of *collecting items* with maximal sliding agents [Tej14]. In this paper, we will instead explore the complexity (among other aspects) of *covering the board* in various ways. Note that while in two dimensions, the P time algorithm is equivalent for Pass Coverage, proving NP-completeness in three dimensions is strictly harder, as we tackle a more specific problem, and therefore have less freedom when it comes to creating a reduction. For example, in Tejada's reduction, the maximal sliding agent chooses between two paths to denote true or false for each variable, and then takes corresponding paths to clause gadgets to reach a target square with a desired item, before continuing through all the other variables. This method does not work for full coverage of the grid; after all, even if you manage to reach the target square, how can you cover the other paths not taken? If you choose true for a variable, how do you cover the false path without allowing the agent to take both paths? Is there a way to cover all squares except for the special clause squares? We solve these issues and many more in Section 3 by creating a mostly open grid that allows us to cover paths from a different direction. Our reduction also requires many more layers than Tejada's.

Broadly, complexity theory is concerned with the intrinsic difficulty of a problem, as well as making comparisons between the difficulties of two problems. As such, complexity theory features various categories, such as P and NP. We will present some important definitions, and then explain their connections to our work. [TW06]

Definition 1 (Language). *Any set of strings that can be input to a Turing Machine is a language. Languages, which often represent problems, are members of various complexity classes, which we introduce below. For example, if the problem at hand asks if a number is prime, we then associate that problem with the language* $L_{\text{PRIMES}} =\{$*string which is a natural encoding of the problem of* n *being prime or not* $\forall n \in \mathbb{N}\}$ *However, we will often abuse notation and simply say that a problem is in a complexity class, instead of the language associated with this problem.*

Definition 2 (P). *The complexity class* P *contains all languages such that there exists a DTM* L *which decides* P *in time that is bounded by a polynomial function of the input size (the length of the string that represents the problem). If* Π *is a decision problem for which* $L_{\Pi} \in \mathsf{P}$ *we say that there is a polynomial time algorithm for* Π*.*

Definition 3 (NP). *When a decision problem* Π *has a succinct certificate which can be used to check that a given instance is true in polynomial time then we say that the associated language,* $L_{\Pi} = \{x | x$ *is a natural encoding of an instance of* $\Pi\}$ *is accepted in non-deterministic polynomial time, or it is in* NP*. In other words, given a proposed "solution" to a problem in* NP*, there exists a polynomial time algorithm to check the validity of this solution.*

Definition 4 (Polynomial Time Reduction). *If* A *and* B *are both languages and a function* f *that runs in polynomial time of input size that satisfies* $x \in A \iff f(x) \in B$*, then* f *is a polynomial time reduction from* A *to* B *and we write* $A \leq_m B$*. Importantly, if* $A \leq_m B$ *and* B *is 'easy' then so is* A*. This allows us to compare the difficulty of problems in some instances.*

Definition 5 (NP-hard). *We define a language* L *to be* NP*-hard if* $A \in \mathsf{NP}$ *implies that* $A \leq_m L$*. Such a language* L *is at least as difficult as any other language in* NP*.*

Definition 6 (NP-complete). *We define a language* L *to be* NP*-complete if* L *is both in* NP *and is* NP*-hard. Such a language* L *can be thought of the "hardest" problems in* NP*.*

Definition 7 (Ice-sliding game). *The 2-dimensional version of the Ice-sliding game is played on an* $m \times n$ *grid of squares. Each square is either empty, or occupied by a barrier. The player begins by occupying some designated empty square. The player may then move in the directions up, down, right, or left. However, the player is a maximal sliding agent, that is, once it begins moving in one direction, it will not stop until stopped by a barrier (the player stops in the square directly adjacent to the first barrier it encounters). Given these rules for movement, the objective of the game is to accomplish various tasks on the game board, which we will specify below. The 3-dimensional version is similar: it is played on an* $a \times b \times c$ *array of cubes (for continuity, we still refer to these cells as "squares" below), either empty or filled with a barrier. In a similar maximal sliding fashion, the player can move around, except it has six directions to choose from: up, down, right, left, front, or back. This definition can be extended similarly for all dimensions at least 2.*

We note now that there is no intrinsic relationship between the difficulty of a problem in NP versus one in P; problems in NP are not automatically harder than those in P (after all $\mathsf{P} \subseteq \mathsf{NP}$). However, mathematicians do focus on studying the hardest problems in NP, which we have defined to be NP-complete above. While it is generally accepted that NP-complete problems are not in P, nobody has succeeded in providing a proof. After all, this is one of the Millenium Problems [Jaf06]. As it is generally thought that $\mathsf{P} \neq \mathsf{NP}$, we call such NP-complete problems *intractable*; in other words, unless $\mathsf{P} = \mathsf{NP}$, there does not exist an algorithm that accepts the problem in polynomial time.

In this paper we will study the complexity of decision problems involving the *Roller Splat!*, or Ice-Sliding motif as defined above. Specifically, we show that, given an initial game board with obstacles, it is in P to determine whether the desired goal can be reached. There are four variants on the goal, which we enumerate in section 2. The four decision problems are in P by virtue of the polynomial time algorithms we provide below. Furthermore, we show that the 3 dimensional analogue of the Pass Coverage variant is in fact NP-complete by constructing a novel reduction from 3-SAT, a famous problems which is known to be NP-complete [Coo71].

This problem has natural applications in robotics. Consider the Roomba series of autonomous robotic vacuum cleaners sold by iRobot. Robots like this rely on simple sensors, and act as maximal sliding agents trying to achieve Pass Coverage over a designated area in order to clean it. Navigational challenges by small, cheap robots in organized storage areas can be solved by Stop Coverage as well. While more complex methods of autonomous travel are possible, they rely on more complicated, expensive, and bulky equipment. Thus, this simple model of maximal sliding agents is still highly relevant given the current constraints of technology. Furthermore, rigorously studying the computational complexity of a game can potentially improve the quality of future games and puzzles as well. For example, given our analysis, futher game-makers may be able to determine if they have created the hardest ice-sliding puzzles, and may also be able to determine if their puzzles are too hard/impossible to solve.

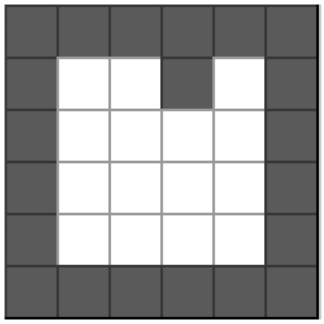
2 Polynomial-time Algorithms for 2-Dimensional Variants

In this section, we provide polynomial-time algorithms for four 2-dimensional ice-sliding problems to show they are in P:

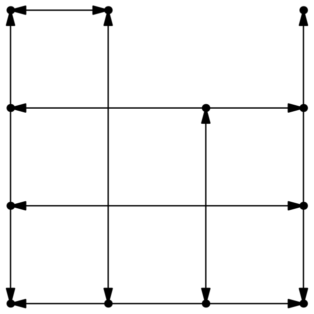
1. Stop Reachability: whether or not the agent can stop on some target square;
2. Pass Reachability: whether or not the agent can pass through (traverse) some target square;
3. Stop Coverage: whether or not the agent can stop on every square;
4. Pass Coverage: whether or not the agent can pass through (traverse) every square.

In particular, Pass Coverage describes the game of *Roller Splat!*, while Stop Reachability and Pass Reachability are common in RPG games.

A natural way to express the board is as a directed graph, where the vertices represent possible stopping points of the agent, and directed edges represent possible movements between them. Note that the edges must be directed since, as in the case below, the agent may only be able to travel from one point to another. We depict examples of these graphical representations from the example board shown below.



Definition 8 (Stopping Point Graph Representation). *Given a board, assign it a graph G. Its vertices are all potential stopping points, which are squares directly next to barriers. We connect two vertices x, y of G with a directed edge if an agent placed on the square corresponding to vertex x can make a move to the square corresponding to vertex y.*



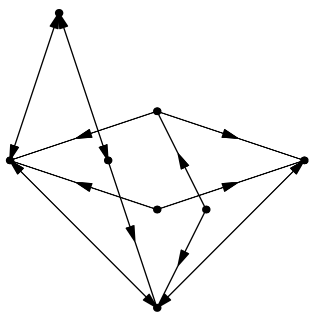
The construction of this graph can be completed in $\mathcal{O}(N^4)$ time, with $N = \max\{m, n\}$ where m and n are the dimensions of the grid, since it takes $\mathcal{O}(N^2)$ time to choose the valid vertices, then at most $\mathcal{O}(N^4)$ time to connect directed edges.

For our code implementation, we will create a new class instance `Coordinate` for each point in the grid, defined by its row and column values, x and y.

The method `findStoppingPoints` (**Algorithm 1** in Appendix A) takes in the inputs of m and n, as well as a list of the Coordinates B_1, consisting of all the coordinates of squares with barriers in them. For the example board, $m = 4$, $n = 4$, and B_1 is a list containing one element: the Coordinate $(1, 3)$. The method outputs the list of Coordinates S, containing all the stopping points.

Definition 9 (Movement Graph Representation). *Given a board, assign it a graph H. Its vertices are all possible sets of back and forth movements. We connect two vertices x, y of H with a directed edge if it is possible for an agent undergoing the back and forth movement of x to, after reaching a stopping point, change direction and enter the back and forth movement of y.*

The key observation motivating this representation is that all the arrows in the movement graph representation come in pairs, so we can assign each pair a vertex in H. In the following depiction, we put the vertices at the midpoints of each of these back and forth movements. Note that the number of vertices in H matches the number of back and forth movements in G.



The construction of this graph can also be completed in $\mathcal{O}(N^4)$ time, as there are at most $\mathcal{O}(N^2)$ back and forth movements, so it takes $\mathcal{O}(N^4)$ time to find the valid directed edges. This graph is relevant for Tejada's proof of Pass Coverage, discussed in section 2.4, as well as for showing the membership of d-dimensional case of Pass Coverage in NP.

2.1 Stop Reachability

Theorem 1. *Given a board, a starting square, and an ending square, the problem of determining whether there exists a path from the starting square that stops on the ending square is in* P.

Proof. Consider the stopping point graph representation G, which can be constructed in $\mathcal{O}(N^4)$ time. Potentially, the starting square is not a stopping point in the graph, so in this case, we add it to the graph, with an indegree of 0 and outdegree of at most 4 (four cardinal directions). From this starting square, we can perform a breadth first search (BFS) in $\mathcal{O}(V+E) = \mathcal{O}(N^4)$ time to see if there exists a directed path from the starting square to the ending square. □

First we define the method `isReachable` (**Algorithm 2** in Appendix A) for performing a BFS through the graph representation. It takes an input of a starting coordinate s, an ending coordinate e, and a directed adjacency list D representing the graph.

In order to utilize this method, we must construct the stopping point graph representation. The method `stopReachability` (**Algorithm 3** in Appendix A) constructs it before running the previous `isReachable` method on it.

2.2 Pass Reachability

Theorem 2. *Given a board, a starting square, and an ending square, the problem of determining whether there exists a path from the starting square that passes through the ending square is in* P.

Proof. This is very similar to the above proof, except we add in a special red vertex v corresponding to the ending square into the stopping point graph representation G. We connect directed edges from the possible stopping points that, when a paintball is placed there, may make one move and pass over the ending square. Now perform another breadth first search from the starting square in $\mathcal{O}(N^4)$ time. □

In `passReachability` (**Algorithm 4** in Appendix A), we construct the modified stopping point graph representation with the added red vertex; we just always add the coordinate e to the adjacency list of each vertex if it can reach it with one of the four directional moves.

2.3 Stop Coverage

Theorem 3. *Given a board and a starting square, the problem of determining whether it is possible to stop on every square in the board is in* P.

Proof. Consider the stopping point graph representation G of the board. We can easily check in polynomial time whether or not this graph contains every vertex; if not, we're done. It suffices to find a traversal through these vertices from the starting vertex.

This directed graph can be uniquely maximally decomposed into its strongly connected components (SCCs), which are subgraphs C of G such that for any two vertices $x, y \in C$, there exists a directed path from x to y. Using Tarjan's algorithm [Tar72], we can find SCCs using depth first search in $\mathcal{O}(N^4)$ time. Now consider the reduced graph G' with vertices as SCCs (also called the *condensation* of G), and connect any two vertices $u, v \in G'$ with a directed edge if there exists a directed edge in the graph G between *any* vertex in the SCC corresponding to u and *any* vertex the SCC corresponding to v.

There exists a path through G if and only if there is a path through G'. To see if a desired path exists through G', consider a topological sort of its vertices; that is, put its vertices in an order $u_1, u_2, \ldots, u_k$ such that there exists a directed edge $u_i \to u_j$ only if $i < j$. This may also be thought of as a linear extension of a partially ordered set: we dictate a full ordering on some partial ordering.

Now, we must move through the vertices in the order $u_1, u_2, \ldots, u_k$, otherwise we have to go backwards at some point. Then it is clearly $\mathcal{O}(k) = \mathcal{O}(N)$ time to check if there exists an edge between u_i and u_{i+1} for every i, and also that the starting square is in the SCC corresponding to u_1. $\square$

The main structure of Tarjan's algorithm is a recursive depth first search (DFS), where we define a stack for the order for the vertices to be checked in (first in, last out), then run the iteration on each of a vertex's unvisited neighbors. Here, we also label each vertex with an index and a lowlink. The indices are labelled in the order in which the DFS uncovers new vertices, and the lowlinks tell us the lowest index (including its own) we can reach from that vertex in its DFS subtree. The end result is that points that all share the same lowlink become strongly connected components. A subtle nuance is the lowlink tells us the lowest index *still on the stack*.

The key idea is that the stack ensures that we only examine one "branch" of the DFS tree at a time, and if we ever find an edge going across branches in the tree, we ignore it since they can't form an SCC.

For example, consider a simple example of three vertices A, B, C, with directed edges $A \to C$ and $B \to C$. Let's say out DFS begins on vertex A. We assign A an index of 0 and lowlink of 0, then move to C and assign it an index of 1. Since C cannot go anywhere, it must have a lowlink of 1 as well. Now when we continue the DFS with B, we assign it an index of 2, and *want* to assign it a lowlink of 1, since it can reach C. This is incorrect however; B and C are not strongly connected. The difference is that whenever we encounter a situation where we cannot move to an unvisited vertex (for example when we reach C), we check if its lowlink matches its index. If so, we remove it from the stack. If its lowlink is less than its index, we keep moving up the graph until we find a vertex whose index does math its lowlink, then remove all vertices above it in the stack, who must now have the same lowlink and form an SCC. Since C has been removed, we would proceed up the graph, checking A and remove it as well. Then B must be assigned a lowlink of 2 and given as its own SCC.

This stack method also gives the interesting property that it doesn't matter where the DFS starting points are chosen, or in what order. For example, if we instead began on vertex C in the previous example, we would assign it an index of 0 and lowlink of 0, before immediately removing that vertex as an SCC. Then we would proceed to either A or B.

Luckily, Tarjan's algorithm also outputs SCCs in reverse topological order [Har11], so we can skip the topological sort step and directly check the directed edges. Note that in the case above, both orders CAB and CBA are valid topological sorts. Stop Coverage (**Algorithm 5** in Appendix A) is an implementation of Tarjan's algorithm on the directed adjacency list D, followed by a check of whether or not there exists an edge from u_i to u_{i+1} for all i.

2.4 Pass Coverage

We end this section by discussing the following result.

Theorem 4. *Given a board and a starting square, the problem of determining whether it is possible to pass through every square in the board is in* P*.*

As mentioned before, this follows as a direct corollary of the result of Tejada:

Theorem 5 (Tejada). *Given a board, of which some (specified) squares have a collectible object in them, and a starting square, the problem of determining whether it is possible to collect every object by passing through its squares is in* P*.*

The particular case of filling every square with a collectible object directly proves Theorem 4. His proof involved using the Movement Graph Representation H of the board and contracting all SCCs to the reduced graph H'. Then, we can associate every pearl with two movements, vertices in H', and construct a 2-SAT formula that is satisfiable if and only if all the pearls can be collected. Further details can be found in section 2 of his paper.

3 Intractability of Pass Coverage in Higher Dimensions

For three of the four problems discussed in the previous section, Pass Reachability, Stop Reachability, and Stop Coverage, the polynomial time solution depends on the efficient calculation of the stopping point graph G. Therefore, we may extend the graph argument in section 2 to any number of dimensions: in fact, given a grid in d dimensions, we may, in the worst case, calculate the potential stopping points in $\mathcal{O}(N^d)$ time, and subsequently calculate the entire stopping point graph in $\mathcal{O}(N^{2d})$ time by considering each pair of potential stopping points at a time. As the algorithms are fundamentally polynomial-time, each of these three problems will remain in P in higher dimensions. However, the same is not the case with the fourth problem, Pass Coverage. On a 2-dimensional board, each square is associated with at most two movement paths: a horizontal path passing through it and/or a vertical path passing through it. This allowed Tejada to construct a polynomial-time reduction from Pass Coverage to 2-SAT, thereby proving the tractability of the pass coverage problem in two dimensions. This argument fails in higher dimensions: in k dimensions, there may be as many as k movement paths associated with a square, and the k-SAT problem has been shown to be NP-complete for $k \geq 3$. In the same paper, Tejada showed that the problem of collecting objects in boards of dimension greater than 2 is NP-complete.

The fact that each square in higher-dimensional grids can belong to more than two movement paths hints at their NP-hardness through possible reductions from k-SAT. In this section, we investigate the problem of determining, given a grid of dimension 3 or greater and a starting square, whether every square of the grid can be traversed:

Theorem 6 (d-Dimensional Pass Coverage). *Given a d-dimensional grid and a starting square, the problem of determining whether it is possible to pass through every square in the grid is* NP*-complete when $d \geq 3$.*

First, we will show that this problem is in NP, before performing a reduction from 3-SAT to show it is NP-hard. It suffices to prove that for any starting point, the optimal path to traverse as many squares as possible is bounded in length by polynomial time, since this means we can check that path in polynomial time. Consider the movement graph representation H of the grid, which can be constructed in $\mathcal{O}(N^{2d})$ time. Suppose that from the starting location, the particle can travel to $v = \mathcal{O}(N^d)$ vertices in H, corresponding to back and forth movements. Then, the particle takes $\mathcal{O}(v)$ moves to get to any vertex it can reach. Therefore, it takes only $\mathcal{O}(v^2) = \mathcal{O}(N^{2d})$ moves to traverse all squares that can be reached, and so it can be checked in polynomial time.

We now show that the problem of Pass Coverage in 3 dimensions is NP-complete by performing a reduction from 3-SAT, which implies NP-completeness for all boards of dimension at least 3 by setting all the other dimensions to 1. This is notably a different problem from the problem formulation found in Tejada's paper, that collecting given objects on a board of dimension greater than 2 is NP-complete. As his construction provided insight only for collecting specifically placed objects, his solution cannot be generalized to provide a solution for traversing every square of a given board. This gives rise to the following proposition, which we prove in this section:

Theorem 7 (3-Dimensional Pass Coverage). *Given a 3-dimensional grid and a starting square, the problem of determining whether it is possible to pass through every square in the grid is* NP*-complete.*

Proof. We perform a reduction from 3-SAT. Suppose that (V, C) is a specific instance of 3-SAT, where $V = \{v_1, v_2, \dots, v_n\}$ and $C = \{c_1, c_2, \dots, c_m\}$ are the variables and clauses in the instance. We create a grid with a variable gadget for each variable $v_i \in V$ and a clause gadget for each clause $c_j \in C$. As we are required by the problem statement to fill in every square, however, we will have four other gadgets responsible for passing through the remainder of the grid after traversing the variable and clause gadgets.

We now begin our proof with a few definitions and an overview of our construction.

3.1 Definitions and Structure of the Grid

A few definitions are in order before starting.

Definition 10 (Grid). *The grid refers to the entire 3-dimensional object that we construct in this section.*

Definition 11 (Board). *A board refers to any of nine 2-dimensional layers of the grid. More specifically, since any grid will always measure* $9 \times m \times n$ *squares, a board refers to any of the nine* $m \times n$ *layers.*

Definition 12 (Particle). *The particle is the* $1 \times 1 \times 1$ *moving object within the grid that serves as the "maximal sliding agent".*

Definition 13 (Block). *A block is any* $1 \times 1 \times 1$ *region in the grid that is occupied, so that the particle cannot pass through it.*

Definition 14 (Square). *A square is any* $1 \times 1 \times 1$ *region in the grid that is unoccupied, so that the particle may pass through it. For the sake of consistency from Section 2, we choose to keep the name "square" over the technically more correct "cube".*

Definition 15 (Movement). *Describing movement in the grid can be difficult, so we provide some terminology. An "upward (or downward) movement between layers" signifies that the particle is sliding from one board to another board, while maintaining the same coordinates within the overall* $m \times n$ *configuration. An "upward (or downward) movement", without reference to layers, signifies that the particle is sliding within a single board in the vertical direction. A "left (or right) movement" signifies that the particle is sliding within a single board in the horizontal direction. Orientations for movement can be found with the figures in Appendix B.*

Definition 16 (Variables). *For this paper,* $\overline{v}_i$ *represents the negation of the variable* v_i*.*

Given a 3-SAT instance, our construction of the grid consists of nine 2-dimensional boards stacked on top of each other, the first and last of which are completely filled "covers" and act as boundary layers. The construction of the nine boards is shown in Figure 19, and discussed in Section 3.3. We provide an illuminating example as a means to discuss the proof for Theorem 7, namely the 3-SAT instance $(v_1 \vee v_2 \vee v_3) \wedge (\overline{v}_1 \vee \overline{v}_2 \vee \overline{v}_3)$. The construction of the grid for this 3-SAT instance features nine 2-dimensional boards, each measuring 70 blocks long by 89 blocks wide; however, as will be seen, only a small minority of the grid is used in setting the variable truth assignments. The remaining space is necessary for covering the entirety of the board after traversing all the variable and clause gadgets. All six gadgets are described in detail in Section 3.2, and all nine layers are described in detail in Section 3.3.

One notable feature of the grid is that, no matter what truth assignments the variables v_i have, every square in the grid can be reached except for at most m squares, one for each clause $c_j \in C$. These m squares can only be reached by a suitable choice of truth assignments. Therefore, a suitable variable assignment is both necessary and sufficient for the completion of the grid. By assigning truth values to the v_i so that the given 3-SAT instance is satisfied, one will be able to use the grid to reach all of the m clause squares and therefore traverse every square in the board. Conversely, by assigning truth values to the v_i that do not satisfy the instance, one will not be able to reach all m clause squares and traverse every square.

A valid traversal consists of two parts: first, the *variable truth assignment traversal* (VTA traversal), and second, the *pass coverage traversal* (PC traversal). The VTA traversal consists of the truth assignments of the variables, along with the traversal of all paths associated with the truth assignments in the grid. The PC traversal consists of traversing the remainder of the squares, except for the m clause squares, after the VTA traversal has been completed. The motivation for using these two traversals in conjunction is that every square in the grid except the m clause squares should be reachable no matter what truth assignment is given to the variables. In this way, the reachability of every square in the grid is dependent only on the truth assignments of the variables. Less rigidly, the squares other than the clause squares should not matter; only the reachability of the clause squares should determine the reachability of all the squares in the grid in order to ensure a proper 3-SAT reduction.

3.2 Gadgets

We now introduce the gadgets present in our construction. Unless otherwise stated, an instance of each gadget is contained entirely within one of the nine 2-dimensional boards.

Variable Gadget

The variable gadget ensures that each $v_i \in V$ is assigned either true or false, but not both. The relevant "variable selection" section of the gadget is shown in Figure 5. Note that the variable gadget is not reachable during PC traversal, as PC traversal, by definition, comes after the variable truth assignment phase, and therefore should not return back to the variable gadgets. Thus, it must be possible to traverse the entire variable gadget during VTA traversal. Indeed, while only one of the two paths in each gadget (the true path or the false path) is selectable, both paths can be traversed before the selection. For example, if we wish to assign "true" to a certain variable, the particle will first move downwards to the red square, then make a movement upwards between layers to a layer above, then move back down to the variable layer. Then, the particle moves back up to the green square and can make the relevant selection by moving upwards between layers to the above layer and going to the right, exiting the variable gadget and entering the clause gadgets. The opposite may be performed to assign "false" to a variable.

After exiting the clause gadget, the particle will enter the variable return section of the variable gadget. An example of variable return is depicted in Figure 6. The purpose of variable return is to provide a path for the particle to reach the next variable gadget. As before, the variable return section can also be entirely traversed during VTA traversal. After returning down to the variable layer, for example through the green "true" path, we have the opportunity once again to cover the squares above the red "false" square, by moving downwards, then back up between layers (to pass through certain squares in the "false" selection), then back down between layers and to the next variable selection. Once again, the opposite may be performed if the "false" path was selected.

It is important to note that all variable gadgets are completely traversable during VTA traversal, no matter what truth assignment is chosen; therefore, PC traversal is not necessary for traversing these gadgets.

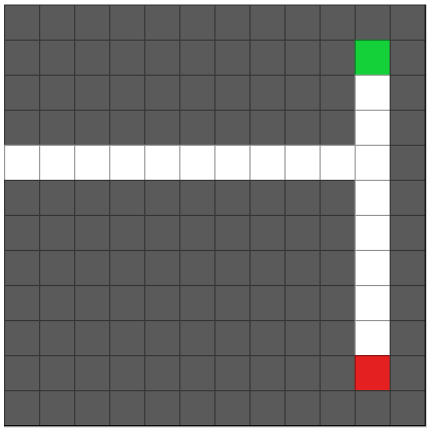


Fig. 5: The variable gadget. Choosing the green path indicates assigning "true" to the variable in question, and choosing the red path indicates assigning "false." Note that since there are layers above and below this layer, the paths are not dead-ends, and instead lead to the clause gadgets.

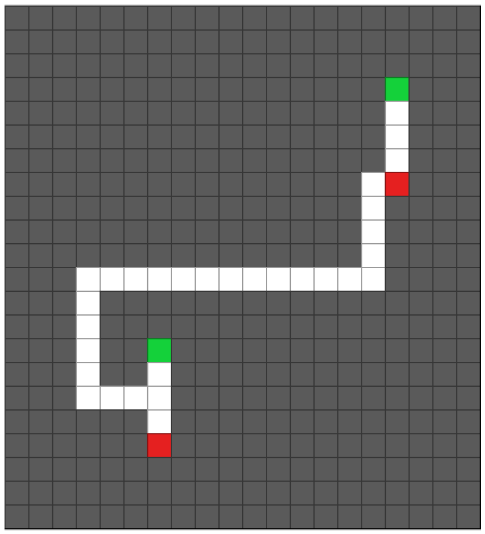


Fig. 6: A simplified version of the variable return part of the variable gadget. This part allows the particle to travel to the next variable gadget after traversing the clause gadgets for the previous variable. The corner at the red square allows the particle to travel upwards in order to cover squares that are traversable only during VTA traversal.

Clause Gadget

The clause gadget ensures that for each clause $v_i \vee v_j \vee v_k$ (negation of variables are similar), the particle can only reach the corresponding clause square through the three variable gadgets corresponding to v_i, v_j, v_k. Supposing that $i < j < k$, v_i will go up between layers to a layer above and come back down directly onto the clause square, v_j will go downwards through the clause square, and v_k will pass directly through the clause square to the right, as shown in Figure 7. Of the three, only v_i can stop on the clause square, meaning that it is technically possible to travel from a lower-indexed variable path to a higher-indexed variable path. However, there is no advantage to doing this, as the particle would be traversing a later variable truth assignment without completing the current one. It would accomplish the same task to simply finish the current truth assignment, then choose the truth assignment of the later variable once the particle reaches it. Therefore, when traversing the smallest-indexed variable v_i, it is better to travel back along the same path and continue with the variable. The paths are shown in Figure 7. Note that a particle traversing v_j will also return back to where it came in order to avoid PC traversal, and a particle traversing v_k gets no choice at all but to continue.

Clause Wall Gadget

The clause wall gadget is a wall separating adjacent clauses from each other. It is completely solid, except for 1×1 holes that allow passage to the next clause gadget after the current one is finished. These holes are mostly necessary in VTA traversal, in order to allow each variable path to continue to the next clause after the current one; in these cases, a hole is simply placed at each spot where the particle would ordinarily hit the clause wall. It is also used in PC traversal; however, its use there is very specific for type III wall gadgets, discussed later.

The clause wall gadget ensures that two problems do not occur: first, it prevents the particle from potentially reaching squares that are not part of the chosen truth assignment path of the variable that it is currently traversing, during VTA traversal; second, it prevents the particle from potentially retracing parts of VTA traversal from PC traversal. Its presence simplifies the clause gadget construction by disconnecting the clause gadgets from each, so that it suffices to focus on any one clause at a time instead of needing to consider all of them at once.

Traversal Gadget

The traversal gadget is the primary gadget in the PC traversal. Inspired largely by the minimal block construction of [PD18], it consists of an $n \times 8$ section, with four blocks placed as shown in Figure 9. Also shown in the same figure is the traversal of this area using arrows, beginning at the top left corner. Note that, by the figure, the particle may freely travel within any given traversal gadget; also, if one places two traversal gadgets side-by-side, the particle may travel from one traversal gadget to the other. By adjoining multiple traversal gadgets side-by-side, it will always be possible to reach the top left corner of each gadget, so this construction suffices to cover all possible starting positions and show that these traversal gadgets do indeed provide coverage of all the squares.

Note that in order to reach some of the squares towards the right of the gadget, it is necessary to travel leftwards from another gadget to the right in order to reach those squares. This, however, will not prove to be a problem due to the fact that while we have presented our traversal gadget as traversable from left to right and where the long paths (arrows)

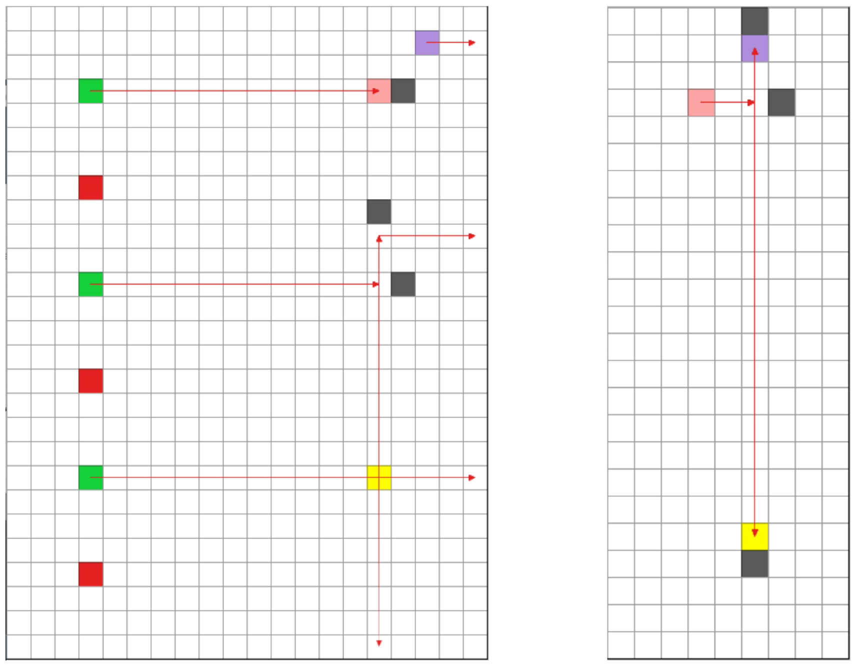


Fig. 7: A simplified version of the clause gadget. The three variable gadgets (v_i, v_j, v_k from top to bottom, with $i < j < k$), with their true and false selections, can be seen at the left side of the left figure. The particle, when traversing the topmost variable v_i, will travel to the pink square in the left figure, where it will then travel upwards between layers to a layer above. The clause gadget in this new layer, shown in the right figure, will guide the particle to the yellow clause square. Upon the particle's return to the old layer, it will drop to a new horizontal path that is above and to the right of its original path, shown by the lavender square and the arrow. The particle, when traversing the middle variable v_j, will remain in the same layer, but will pass through the clause square vertically from above. Upon the particle's return, it will enter a new horizontal path above the previous path. The particle, when traversing the bottommost variable v_k, will simply pass through the clause square horizontally and continue to the next clause gadget. Note that when the particle traverses v_i, it will stop on the clause square. This makes it possible to enter the traversal of a different variable (i.e. v_j or v_k); however, since $i < j < k$, this provides no advantage to the particle. If the particle so wished, it could choose the truth assignment for v_j or v_k later in VTA traversal that would lead to the same path, accomplishing the same task as switching variable paths.

run vertically (horizontal), the gadget can also be rotated 90° to be traversable from top to bottom and where the long paths run horizontally (vertical). Thus, all four walls of the board in which they reside will be lined with traversal gadgets, allowing all squares to be traversable. Our construction of the 3-dimensional grid will utilize both the horizontal and vertical traversal gadgets, in order to prevent the particle from passing through the clause squares present in the same layer as the traversal gadgets during PC traversal.

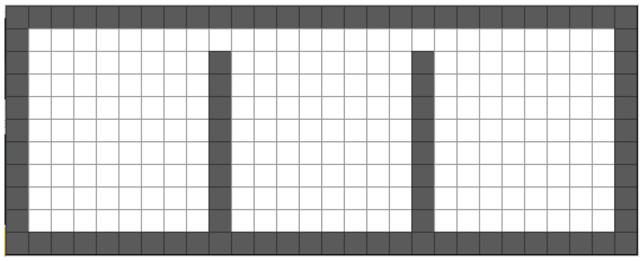


Fig. 8: The clause wall gadget, as shown by the vertical walls in the middle of the board (traversal holes in wall not shown). Each of the three boxes represents a clause. The holes at the top allow travel from clause to clause during the PC traversal.

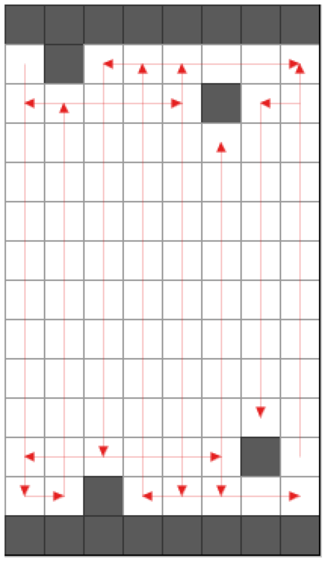


Fig. 9: The traversal gadget, with the particle beginning in the top left square. The arrows show that every square in an 8-square-long section can be traversed by placing four blocks as shown in the figure.

Drop-Down Gadget

The drop-down gadget is used to traverse single squares that cannot be covered by the traversal gadgets, within a layer. Note that ordinarily, the traversal gadgets would be sufficient for the particle to traverse an entire layer, as seen in the previous section. However, due to the fact that there are blocks present in the middle of traversal layers, suitable adjustments were made to accommodate such changes. As a result, squares that were before traversable using the traversal gadgets could no longer be reached. Any square that is part of both a vertical and horizontal movement path in VTA traversal within a single layer cannot be covered by traversal gadgets, as an attempt to do so would cause the particle to return to VTA traversal. One example of this is the square returning from the clause square in the variable v_j in a clause $v_i \vee v_j \vee v_k$ with $i < j < k$. Another example is the holes in the clause wall gadgets, since the clause wall gadgets prevent vertical traversal of these squares.

The drop-down gadget can only be accessed in one layer of the grid; however, each gadget spans as many as 7 layers. At each drop-down square in the access layer (later called the drop-down layer), the particle will stop on that square and "drop down" (i.e. move downwards between layers) to reach the target squares. Note that due to the sparsity of these squares, there is no problem with installing enough corners to be able to drop down into the required squares. (That this actually does work is shown in section 3.6.) Finally, due to the presence of a ditch layer at the bottom of the grid, dropping down any of the drop-down gadgets will cause the particle to land in a square surrounded on five sides by blocks. This forces to particle to retrace its steps; thus this gadget does not interact with any layer other than the drop-down layer and the ditch layer.

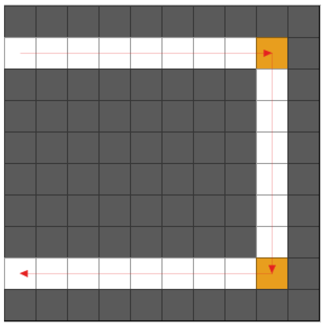


Fig. 10: Example of drop-down gadgets within a layer. The drop-down squares are shown in orange, and they are present at corners where the particle can stop and drop down into them.

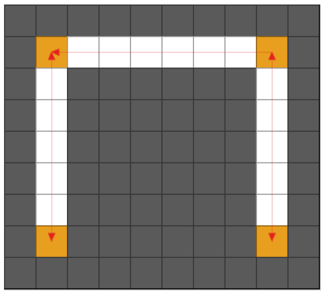


Fig. 11: Cross section of drop-down gadgets, showing all nine layers. The second row (from the top) is the layer shown in Figure 10 (the orange drop-down squares correspond to each other), and the vertical tunnels show the depth of the gadgets, passing through the entire grid down to the bottom.

Wall Gadget

The wall gadget is a modification in the traversal gadget, necessary in our construction of the grid. It consists of a 2-square-wide and 1- or 2-square-deep hole in the side of the wall to allow an extra movement path that would ordinarily not be allowed by the traversal gadget, as shown in Figures 12, 13, and 14. Note that this addition does not break the traversal gadget, as it still allows all other movement paths to remain the same.

The purpose of the wall gadget is threefold: first, to allow a transition between the vertical and horizontal traversal gadgets at a corner (two per clause gadget); second, to cover the rest of a path ordinarily induced by a traversal gadget that was partially blocked off by a block in the middle of the board (one per clause gadget); third, to cover squares that are impossible to cover by drop-down gadgets or by traversal gadgets. The wall gadgets used for these purposes are named type I wall gadgets, type II wall gadgets, and type III wall gadgets, respectively. Examples of all three uses are shown in Figures 12, 13, and 14.

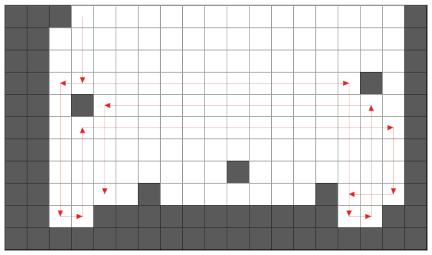


Fig. 12: The type I wall gadget, present in the lower left and lower right corners. Two paths are shown to enter the horizontal traversal gadgets from the vertical traversal gadgets (starting at the top left corner of the figure). Moving from the horizontal traversal gadgets to the vertical traversal gadgets is trivial, as the vertical traversal gadgets can be directly accessed from the horizontal traversal gadgets.

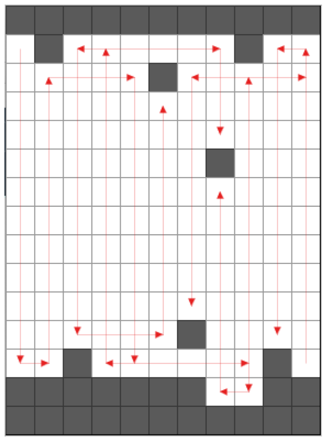


Fig. 13: The type II wall gadget, as can be seen in the lower right corner. Due to the presence of a block in the middle of the board, the wall gadget is necessary to cover the bottom part of the upper path that was blocked by the block. Note also the shrunken traversal gadget at the wall gadget, where consecutive blocks are only two spaces apart instead of three. This occurs only once when a wall gadget is present, and as it is shown that all squares are still traversable in the figure, it does not pose a problem to the overall structure.

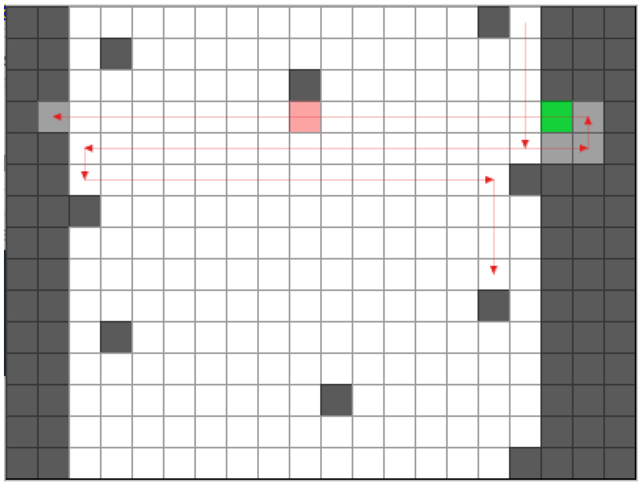


Fig. 14: The type III wall gadget, in the left and right walls. The figure shows part of the v_i clause gadget in a clause $v_i \vee v_j \vee v_k$ with $i < j < k$, after the particle has moved upward between layers. In the middle of the board is a square previously unreachable by PC traversal (pink), since the block above it prevents the particle from reaching it via the horizontal traversal gadgets, and since a drop-down gadget cannot be used due to the fact that the downward motion between layers is used by the clause gadgets; this wall gadget provides the traversal of this square. The green square shows a variable hole present in the layer below, which is why we require the two-square-deep wall gadget to avoid this square. Having only a one-square-deep wall gadget would allow the particle to drop down at the green square and return to VTA traversal. The light gray squares signify that there are blocks in the layer below this layer, but not in this layer. As with all wall gadgets, the traversal gadgets are not affected by this addition (in fact, the wall gadget provides an extra traversable horizontal row), and a path to the next traversal gadget is also shown. Finally, note the four-block gap between some blocks of the vertical traversal gadgets, contrary to the normal 3-block gap presented in the traversal gadget section. This necessary modification will be explained in Section 3.4.

3.3 Layers

There are nine layers of 2-dimensional boards that constitute our construction. In this section we will identify and explain each layer of the grid. The complete construction for the example 3-SAT instance $(v_1 \vee v_2 \vee v_3) \wedge (\overline{v}_1 \vee \overline{v}_2 \vee \overline{v}_3)$ is shown in Appendix B at the end of the paper. A general construction of a grid for any 3-SAT instance will be discussed in Section 3.5, using the terminology introduced in this section. Note that the order in which the layers are introduced is not necessarily the order in which they appear in the construction; refer to Appendix B for the order of the layers.

Note that, in the figures in Appendix B, the squares are colored with many different colors. We have included different colors in order to more clearly indicate the parts of our construction and their functions. While there is only one color to represent a block (dark gray), there are ten other colors that all represent empty squares, but may represent sections that serve different functions in the construction.

Cover Layers

These are the two boundary layers on the top and bottom of the grid, and both are identical and very boring. They are there only to ensure that the particle remains within the construction.

Variable Layer

The variable layer is where the variable assignments are made. The particle traverses the variable layer, eventually reaching a variable gadget for each variable in the 3-SAT instance. As stated before, each variable gadget is designed so that each truth path is traversable before choosing the truth assignment for the variable, so that the particle can visit each square before entering the next variable gadget. The variable gadget connects directly to the main layer, where the clause gadgets reside. At the end of each truth assignment path in the clause gadgets, the particle will return to the variable layer and traverse the next variable gadget.

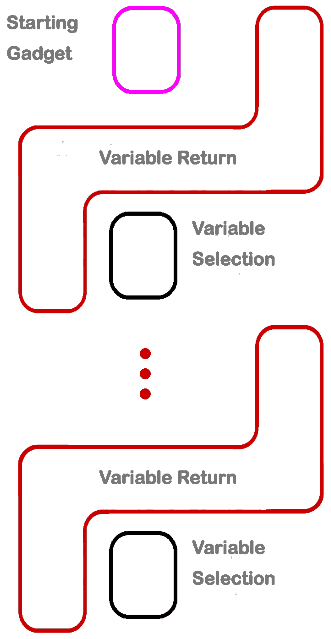


Fig. 15: A high-level overview of the variable layer.

Main Layer

The main layer is one of two layers containing the clause gadgets, the other being the main II layer. Serving as the bulk of the grid's complexity, the variable layer connects directly to this layer. After making a variable truth assignment, the particle will then traverse the clause gadgets in this layer and the main II layer (see below) in order to reach the clause squares associated with the variable. Although much of this layer is open space, the paths along the clause gadgets are accessible only from the variable layer. This accomplishes two things: first, any move leaving the clause gadget will cause the particle to move directly from VTA traversal to PC traversal, so it is in the particle's interest to not stray from the designated path. Second, it prevents the particle from returning to VTA traversal after completing it and moving to PC traversal, thus avoiding potential problems with selecting the same variable multiple times. In addition, there are a few wall gadgets present in this layer. The type I wall gadgets in the corners of each clause gadget ensure a smooth transition from the vertical traversal gadgets to the horizontal traversal gadgets, while the type II wall gadgets provide a means to traverse certain squares blocked off from the normal traversal gadget due to a block present in the middle of the board.

There are also special "elevator" squares in this layer, colored dark blue in the construction in Appendix B. Elevator squares in the corner of each clause designate squares to be used to travel from the main layer to the main II layer, and an elevator square in the upper right corner of this layer allows the particle to travel to layers above the main II layer.

Main II Layer

The main II layer is one of two layers containing the clause gadgets, the other being the main layer. As its name suggests, it serves as a supplement to the main layer. The purpose of the main II layer is to hold one of the three clause paths for each clause (specifically, the first of the three variables). As with the main layer, this layer is lined with traversal gadgets so that the entire layer can be traversed during the PC traversal. One notable difference is the impossibility of using drop-down points at certain squares, due to the fact that the clause gadgets require the downward motion between layers to successfully operate. As such, type III wall gadgets were added to be able to traverse these squares in the same layer. In addition, "variable" blocks have been placed where the variable squares would be present in the main layer, in order to ensure that the particle will stop on the variable squares in the main layer when moving upward between layers from the variable layer.

Because of the additional type III wall gadgets, there were extra complications that arose, such as the particle potentially being able to return to VTA traversal. To avoid this possibility, extra drop-down gadgets have been placed in this layer. Usually, a particle traversing a type III wall gadget will reach the left wall, in which case a 1×1 hole in the wall will suffice to force the particle to retrace its steps. However, in some type III wall gadgets, traversing them will instead cause the particle to land on the right side of a variable block. Because moving downward between layers to the main layer would then allow the particle to retrace VTA traversal, a drop-down gadget beginning in this layer is placed on such squares. This ensures that the particle cannot retrace VTA traversal.

The main II layer also has elevator squares. The elevator squares in the corners of each clause gadget designate squares to be used to travel down to the main layer, and an elevator square in the top right corner of this layer allows the particle to travel to layers above this layer.

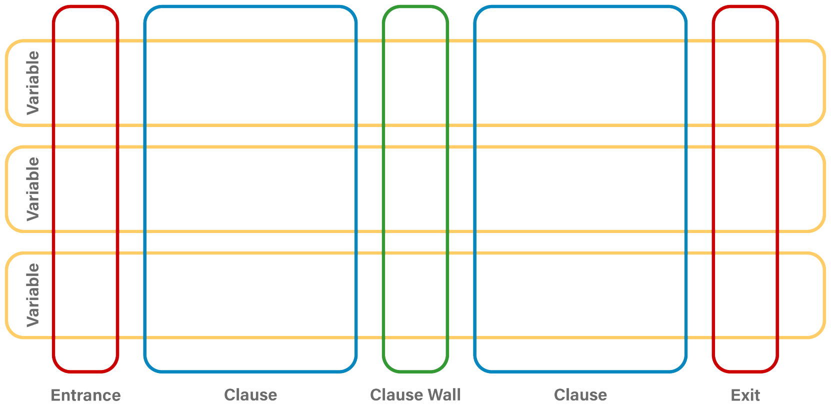


Fig. 16: A high-level overview of the main layer.

Drop-Down Layer

The drop-down layer is the layer where all the drop-down squares in the grid are present. Containing the beginning of the drop-down gadgets, the drop-down square are spaced far enough apart that the gadget does not break. Specifically, the drop-down squares beginning in this layer are all above (between layers) either the variable column, a clause column, or a clause wall gadget (which are all at least 6 squares apart). Furthermore, on each of these columns, there is at most one drop-down gadget in every vertical traversal gadget. Thus, the closest drop-down squares to any given drop-down square is at least six squares away horizontally and eight squares away vertically.

Ditch Layer

The ditch layer is the ending layer of the drop-down gadgets. It consists of only solid blocks, except for 1×1 holes at the end of each drop-down gadget. The purpose of this layer is to prevent the particle from re-entering VTA traversal from PC traversal via the drop-down gadgets by trapping them at the bottom of the grid in holes surrounded on five sides by solid block. Thus, a particle traversing a drop-down gadget is forced to retrace its steps and end up back on the square on which it started.

Slab Layers

The slab layers are two layers in the grid: one between the drop-down layer and the main II layer (the slab II layer), and one between the main layer and the variable layer (the slab I layer). They act as a "sandwich" of the main and main II layers, serving to contain the particle within the clause gadgets as it traverses these gadgets. Similar to the ditch layer, they are completely solid except for 1×1 holes at the drop-down squares; this is to allow the particle to travel all the way through each layer from the drop-down layer to the ditch layer. In addition, the slab I layer will also have holes for the variable truth selection paths, since it is between the variable layer and the main layer.

3.4 Modification of Gadgets in the Grid

In order to correctly construct the grid, it is necessary to modify the vertical traversal gadgets in the grid. Below, we explain the change, its reasoning, and the impact it has on traversal.

Vertical traversal gadgets containing horizontal movement paths that are part of VTA traversal will contain an extra square (i.e. be expanded by a square; see Figure 17). This is necessary in order to avoid passing horizontally through rows that are part of VTA traversal. However, this also causes one row per traversal gadget (specifically, the row containing VTA traversal) to be untraversable by the vertical traversal gadgets. The squares in this row will thus be covered by the horizontal traversal gadgets. Since each clause will only contain at most three variables, any general grid will function exactly the same as our example. As no modifications were made to the horizontal traversal gadgets, every column will be traversed by the horizontal traversal gadgets. Along with the wall gadgets, this ensures that all the squares in the VTA traversal in the main and main II layers (except the clause squares) are also covered by PC traversal.

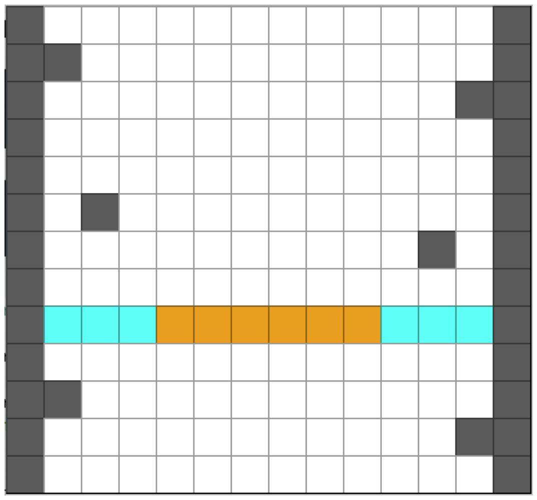


Fig. 17: The modified vertical traversal gadget. The row of colored squares shows the row in which an extra block of width has been added to the gadget. Consequently, the row of colored squares cannot be traversed by a horizontal movement in the vertical traversal gadgets. The light blue squares are traversable vertically in the vertical traversal gadgets, and the orange squares will be traversable by the horizontal traversal gadgets.

3.5 Construction of the Grid

In this section, we make a general construction of a suitable grid given any 3-SAT instance. The main ideas of the construction have already been highlighted in the gadgets and layers section, and by the use of our example 3-SAT instance. We combine all of these ideas in this section. Note that the two **cover layers** are composed of only solid block, and will not be discussed further in this section.

We begin with the **main layer**. Given a 3-SAT instance with n variables $v_1, v_2, \ldots, v_n$ and m clauses, the main layer will consist of m clause gadgets placed side-by-side. Each variable will occupy a horizontal band in the main layer across all the clause gadgets, so that no two variables will intersect with each other. (Refer to Figure 18.) Since each clause will have at most three variables associated with it, the overall structure of each clause gadget will be the same as the example. The only modification, then, to each clause gadget will be the height of each gadget. Since each variable or its negation (i.e. either of v_i or $\overline{v}_i$) is included at most m times in all the clauses, it will move upwards at most m times. (Recall that in a clause $v_i \vee v_j \vee v_k$ with $i < j < k$, the particle will move up by one vertical traversal gadget after completing either v_i or v_j, but not v_k.) Thus, we may allot a height of $m+1$ vertical traversal gadgets to each truth assignment of each variable, one for it to begin in and one for each of the possibilities of it being necessary to move up by one traversal gadget. Note that such an addition does not affect VTA traversal in any way relative to our example: the horizontal traversal gadgets will still cover the majority of the space, with the vertical traversal gadgets responsible for covering the squares along the single clause column in each clause gadget. Since each clause will still contain the same number of variables, we may ignore all variables that are not present in that clause and consider only the three that are, effectively reducing any general construction to our example. This completes the construction of the main layer.

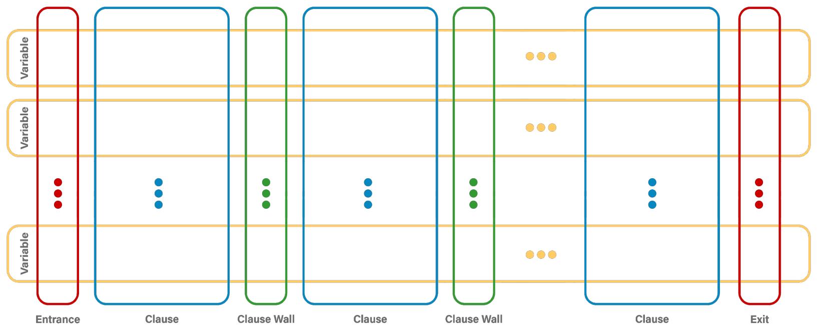


Fig. 18: A high-level overview of a general main layer.

We next construct the **main II layer**. The construction is largely the same as the main II layer in our example; however, we must take into account any type III wall gadgets that we may need upon adding more variables and clauses. We begin the construction by placing all the traversal gadgets and clause wall gadgets at the same coordinates as they were placed in the main layer. We may also place blocks directly over the variable truth assignments that feed into the main layer to ensure that the particle will stop on the desired square in the main layer when coming up from the variable layer. For each clause $v_i \vee v_j \vee v_k$ with $i < j < k$, we then build the path for v_i from the main layer up into the main II layer, then back down to the clause square, as described in the clause gadget section. This completes the clause gadgets. For each clause, we need to decide where the type III wall gadgets and drop-down gadgets will go. Two type III wall gadgets (as described in the wall gadgets section) are necessary for each clause: one to reach the unreachable square in the clause gadget at v_i as described in the wall gadget section, and one for the square immediately above the clause square in the main layer (since this is also normally unreachable in PC traversal). No wall gadget will be responsible for more than one of the former case, since the particle will always be in a higher vertical traversal gadget within the main and main II layers after traversing through v_i. The only potential issue, then, is one wall gadget being potentially responsible for more than one of the latter case. This, however, does not pose a problem, since the wall gadget is perfectly capable of allowing passage through multiple of the squares above the clause squares without causing problems; the particle will only stop once it reaches either the far left wall, in which case a 1×1 hole in the wall will force it to retreat, or once it reaches one of the blocks above the variable truth assignments, in which case a drop-down gadget will be present on that square in order to prevent the particle from potentially retracing VTA traversal by moving back down to the main layer. There is a third square that the particle can use to travel between the main II and main layers: the square that the particle initially arrives on in the main II layer after coming from the main layer. However, note that this square is covered by the horizontal traversal gadgets, since there is no vertical movement in VTA traversal passing through this square. Thus, this completes the construction of the main II layer.

The rest of the layers follow naturally from the construction of the main and main II layers. To construct the **variable layer**, connect the ending squares in the main layer (i.e. the green and red squares at the far right, indicating the end of the variable truth assignment for that particular variable) to the next variable with the variable gadget, in the same manner as in our example. One may always construct a path to the next set of variable truth assignments, which are more than 8 blocks away from the left wall. The starting variable gadget is exactly the same as our example, except that its length will vary depending on the 3-SAT instance.

To construct the **drop-down layer**, note that drop-down squares are either on the variable column, a clause column, or a clause wall gadget column in this layer. Thus, we may make a "snake" through this layer that begins at the elevator square and passes through every drop-down point on a corner, allowing the particle to stop on that square and drop down. (Our example shows an easy way of doing this.) This is achieved by making a path for each column of drop-down squares, advancing to the next column once the current column is finished.

The final three layers are now easily constructed. To construct the **slab II layer**, place holes at only the drop-down gadgets beginning in the drop-down layer and the elevator square, and cover the rest of the board with solid block. To construct the **slab I layer**, place holes at all drop-down gadgets and the variable squares, and cover the rest of the board with solid block. To construct the **ditch layer**, place holes at all drop-down gadgets, and cover the rest of the board with solid block. This completes the construction of our grid, given any 3-SAT instance.

3.6 Proof of VTA Traversal

Refer to Appendix B for the relevant construction.

The particle begins in the purple square on the **variable layer**. At any stopping point within the variable layer, the slab I and ditch layers prevent the particle from moving between layers, so it is confined to the variable selection. Then, if we want to choose true for a given variable, we *first move to the red square, go up, then come back down.* In particular, this is to cover the red square present in the slab I layer that we cannot otherwise reach. Then, we move to the green square and proceed up to the main layer, as we are stopped by the block present in the main II layer.

One of the key ideas in constructing a suitable VTA traversal is that VTA traversal must stay within the VTA traversal path. This is because it is impossible to return to VTA traversal from PC traversal, discussed in more depth in the next section.

In the **main and main II layers**, in order to continue through VTA traversal, there is a well-prescribed path that must be taken. If the current variable or its negation does not appear in a clause, the path will pass right through that clause gadget and enter the next clause gadget. If it does appear, depending on which variable it is, it will take a different path. As discussed in the clause gadget section, a particle passing through each path will pass through the clause square and then return to its original path, with correct traversal (attempting to do anything else will cause the particle to either enter PC traversal or to traverse a later variable without finishing the current one). Thus, the particle will traverse each variable, reaching the clause squares for the clause associated with that variable (or its negation), and passing through all other clauses.

Finally, note that the variable gadgets are designed so that every square can be traversed in the gadgets. This will be essential in traversing all squares in the grid, since the gadgets are not part of PC traversal. (Note that traveling upwards between layers at the variable return squares will cause the particle to end up in the main II layer; thus, traveling upwards between layers at this point doesn't allow extraneous VTA traversal.) Traversing all clause squares during VTA traversal is equivalent to choosing a truth assignment for the variables that satisfies the 3-SAT expression. Since the clause squares cannot be reached via PC traversal (discussed more in the next section), it is necessary that one choose a satisfactory truth assignment in order to traverse every square in the grid. The next section shows that it is also sufficient.

3.7 Proof of PC Traversal

This proof consists of two parts: one, that all necessary squares can be traversed (so that a suitable choice of variable truth assignments is sufficient to totally traverse the grid); two, that it is not possible to return to VTA traversal at any point (so that a suitable choice of variable truth assignments is necessary to totally traverse the grid). This ensures that the purpose of PC traversal is solely to cover every square, except for the clause squares and those necessary to be covered during the VTA traversal.

PC traversal runs through every layer that is not a cover layer; thus, we will consider PC traversal over each layer. The layers that require the most attention are the main and main II layers; in fact, the entirety of PC traversal is focused on covering every square in these layers, other than the clause squares.

We first show that all squares in the **main layer** in PC traversal lead only to other squares in PC traversal. Because of the presence of a wall between each clause gadget, it suffices to prove this for each clause gadget. At most three variables will feed into each clause gadget, meaning that each of the clauses function essentially the same way. As is shown in our example, there is no way to reach any of the paths associated with VTA traversal from the traversal gadgets. Note that the only possible ways to do so would be to travel to the right, passing through the variable squares, or to travel upwards, through the clause square. Both are designed to be impossible. The other way is to potentially pass through the spaces in the walls, but again by design that is impossible. This is elucidated in the previous section, modification of gadgets. Therefore, in the main layer it is impossible to travel from PC traversal to VTA traversal.

Next, we show the same in the **main II layer**. While this layer has many similar elements to the main layer, there are also some notable differences that complicate analyzing this layer. Most notably, in certain circumstances there will be wall gadgets in PC traversal from the right side of the layer all the way to the left side of the layer, meaning that these will need to be analyzed separately since they cross the walls in the middle, although the analysis is not particularly difficult. Simply note that these wall gadgets either land into a square surrounded on five sides by blocks (and therefore the only way out is to retrace the path, thereby remaining in PC traversal), or onto a block in the main II layer, under which block is a variable square. The former case is immediate; the latter case is solved by the existence of a drop-down gadget, beginning in the main II layer, where the particle meets the block. This ensures that any downward movement between layers by the particle will land into the ditch layer, forcing the particle to retreat its steps, and upward movement between layers is prohibited due to the presence of the slab II layer immediately above. Vertical movement at this square will lead to the horizontal traversal gadgets, as shown in our example. Finally, horizontal and vertical movement within the main II layer leads only to the traversal gadgets. This is evident by the construction in our example, and is true for any general grid.

The next step is to show that **it is not possible to travel from the main II layer into the main layer and subsequently land in VTA traversal, and vice versa**. Much of this, however, is clear from the construction. Since the main II layer traversal gadgets are identical in shape and position to the main layer traversal gadgets, any given stopping point in one layer would lead to the corresponding stopping point in the other layer, and thus this interaction does not affect traversal in any way. In addition, using only the traversal gadgets and not the wall gadgets, the particle can only stop on either the top or bottom of a block in the center of either board. (Such a block is there for VTA traversal purposes, meaning that one of the four squares directly adjacent to this block is a VTA traversal stopping point.) By construction, the wall gadgets in the horizontal traversal gadgets prevent entrance into VTA traversal vertically from PC traversal, meaning that none of the stopping points on the top or bottom will be part of VTA traversal. Finally, since horizontal VTA traversal components are at least 6 blocks apart (i.e. in different vertical traversal gadgets), traveling between the main II and main layers in this way will not cause the particle to land in a position where it may return to VTA traversal. The only other PC traversal stopping points in the main II layer are either part of a type III wall gadget or next to a variable block. In the former case, the corresponding squares which are traversable in the main II layer are blocks in the main layer, meaning that travel from the main II layer to the main layer is impossible in this case. In the latter case, the existence of a drop-down gadget ensures that attempted movement between layers from the main II layer to the main layer will only end up in the ditch layer.

Finally, we show that **all squares in both layers can indeed be covered by PC traversal** (except for the clause squares). The horizontal traversal gadgets do most of our work in the main layer. The only squares that they cannot cover are the drop-down squares and certain other squares. These other squares, all in the same column as a clause square, cannot be traversed by the horizontal traversal gadgets because it would allow the particle to return to VTA traversal. The vertical traversal gadgets provide the traversal for the majority of these squares. Note that because the vertical traversal gadgets have been modified with the extra square, one square per traversal gadget will be left out under this traversal, one of which will be a clause square; drop-down gadgets are used to cover these squares (except for the clause square). All squares are thus traversable in the main layer. The horizontal traversal gadgets do most of the work in the main II layer as well, missing only drop-down squares and certain other squares. These other squares are comprised of the same types of the squares as in the main layer (i.e. those in the same column as the clause squares). Once again, the vertical traversal gadgets and the drop-down gadgets will cover almost all of these squares. However, there is one square per clause gadget that cannot be traversed by either the traversal gadgets or the drop-down gadget; this square is the square where the particle would land on to return to the main layer after coming back from the clause square during VTA traversal. To pass through this square, wall gadgets in the right side of the board allow horizontal passage through the square. We therefore conclude that, except for the clause squares, our grid allows the particle to pass through every square in both the main and main II layers.

The remaining part of the proof is fairly easy. To reach the **drop-down layer**, the particle will use the elevator square in the top right of the main layer. The particle can then use the "snake" to pass through every square in the drop-down layer, as well as be able to descend on each of the drop-down squares. This ensures that all the squares in the **slab II layer** are traversable, and that all the squares in the **slab I layer**, except for the variable squares that are traversed as part of VTA traversal, are traversable in PC traversal. All the PC traversal squares in the **variable layer** are part of drop-down gadgets, and the drop-down gadgets all end in the **ditch layer**. This confirms that all squares in the grid, except for the clause squares and the squares exclusively traversed in VTA traversal, are traversable in PC traversal.

The combination of Sections 3.5, 3.6, and 3.7 offer the proof of the intractability of PASS COVERAGE in dimensions of three or greater. □

4 Conclusion

In this paper, we explored the recurring ice-sliding motif. In section 2, we provided pseudocode implementations to solve four 2-dimensional variants, and in section 3, we constructed a novel reduction from 3-SAT to show that the higher dimensional case of Pass Coverage is NP-complete. In the future, we would like to explore combinatorial questions such as determining the minimum number of blocks required to achieve Pass Coverage on a 3-dimensional grid, which may be related to the domination number of grid graphs. We would also like to explore further variants of the problem, perhaps involving multiple agents sliding in turns or incorporating pushable blocks. Further research in the field of the ice-sliding motif might include considering the game on a torus (allowing wrap-around movement) or king grid (allowing diagonal movement), which allow for more complex movement patterns.

5 Acknowledgements and Credits

Acknowledgements

We would like to thank:

- Our mentor, Chun Hong Lo, for his valuable feedback and guidance
- MIT PRIMES for giving us this research opportunity
- Our friend Stephanie Yang, for helping us with some graphics
- Our parents, for their continued support of this project

Credits

While every student contributed to each aspect of the project, specific emphases are noted below:

- Sebastian:
  - Section 3 write-up
  - Theoretical work for algorithms
  - Theoretical work for 3-SAT reduction
  - 2D graphics

- William:
  - Section 2 write-up
  - Theoretical work for algorithms
  - Theoretical work for 3-SAT reduction
  - Code implementation

- Vincent:
  - Introduction
  - Theoretical work for algorithms
  - Research and citations
  - Code implementation

References

3dt. `https://www.amazon.com/Sequential-Educational-Professional-Gifts-Green-Transparent/dp/B06XJLD5CH`.

ADGV15. Greg Aloupis, Erik D Demaine, Alan Guo, and Giovanni Viglietta. Classic nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015.

BC39. W. W. Rouse Ball and H. S. M. Coxeter. *Mathematical Recreations and Essays.* The Macmillan Company, 1939.

cli. `http://www.clickmazes.com/newtilt/ixtilt.htm`.

Coo71. Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing - STOC 71*, 1971.

DHH04. Erik D Demaine, Michael Hoffmann, and Markus Holzer. Pushpush-k is pspace-complete. In *Proceedings of the 3rd International Conference on FUN with Algorithms*, pages 159–170. Citeseer, 2004.

DHLN03. Erik D Demaine, Susan Hohenberger, and David Liben-Nowell. Tetris is hard, even to approximate. In *International Computing and Combinatorics Conference*, pages 351–363. Springer, 2003.

FB02. Gary William Flake and Eric B Baum. Rush hour is pspace-complete, or "why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1-2):895–911, 2002.

Har11. Paul Harrison. Robust topological sorting and tarjan's algorithm in python, 2011.

Jaf06. Arthur M Jaffe. The millennium grand challenge in mathematics. *Notices of the AMS*, 53(6), 2006.

PD18. Andre Fabbri Julien Moncel Aline Parreau et al. Paul Dorbec, Eric Duchene. Ice sliding games. *International Journal of Game Theory*, 47:487–508, 2018.

RW90. Daniel Ratner and Manfred Warmuth. The $(n^2 - 1)$-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10(2):111–137, 1990.

snoa. Snowpoint gym. `https://bulbapedia.bulbagarden.net/wiki/Snowpoint_Gym`.

snob. Snowpoint temple. `https://bulbapedia.bulbagarden.net/wiki/Snowpoint_Temple`.

Tar72. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

Tej14. Pedro J Tejada. On the complexity of collecting items with a maximal sliding agent. 2014.

TW06. John Talbot and Dominic Welsh. *Complexity and Cryptography.* Cambridge University Press, first edition, 2006.

Appendix A

This appendix contains pseudocode for the algorithms in Section 2.

Algorithm 1 Finds all the stopping points

```
procedure findStoppingPoints(Integer $m$, Integer $n$, List of Barrier Coordinates $B_1$)
    $B \leftarrow m \times n$ boolean array        ▷ construct array $B$ with true wherever a barrier is
    for $b \in B_1$ do
        $B[b_x][b_y] \leftarrow$ true        ▷ the Coordinate $b$ is in location $(b_x, b_y)$
    end for
    let $S$ be a new list of Coordinates        ▷ $S$ will contain all the stopping points
    for $i \leftarrow 0, m-1$ do
        for $j \leftarrow 0, n-1$ do
            if $\neg B[i][j]$ then        ▷ run through every point, make sure it's not a barrier
                $a \leftarrow$ false
                if $(i = 0) \vee (i = m-1) \vee (j = 0) \vee (j = n-1)$ then        ▷ checks if it is on the edge
                    $a \leftarrow$ true //or if it is next to a barrier
                else if $B[i-1][j] \vee B[i+1][j] \vee B[i][j-1] \vee B[i][j+1]$ then        ▷ or if it is next to a barrier
                    $a \leftarrow$ true
                end if
                if a then
                    $S$.add$(i, j)$        ▷ then add it to the list of stopping points
                end if
            end if
        end for
    end for
    return $S$
end procedure
```

Algorithm 2 Helper method that runs a BFS to find if there is a path between two points in a directed graph D

```
procedure isReachable(Coordinate $s$, Coordinate $e$, Map⟨Coordinate, List of Coordinates⟩ $D$)
    let $V$ be a map from Coordinate to Boolean        ▷ create a map of whether each vertex has been visited
    $V$.put($s$, true);
    let $Q$ be a Queue of Coordinates        ▷ create a queue for the order to check the vertices in (first in, first out)
    $Q$.add(s);
    while $Q$ is not empty do
        $c \leftarrow Q$.dequeue
        $N \leftarrow D$.get$(c)$
```

```
 9:         for $n \in N$ do
10:             if $V$ does not contain key $n$ then
11:                 if $n = e$ then
12:                     return true
13:                 end if
14:                 $Q$.add($n$)
15:                 $V$.put($n$, true)
16:             end if
17:         end for
18:     end while
19:     return false
20: end procedure
```

Algorithm 3 Stop Reachability

```
 1: procedure stopReachability(Coordinate $s$, Coordinate $e$, Integer $m$, Integer $n$, List
    of Barrier Coordinates $B_1$)
 2:     $B \leftarrow m \times n$ boolean array        ▷ create the barriers array again
 3:     for $b \in B_1$ do
 4:         $B[b_x][b_y] \leftarrow$ true
 5:     end for
 6:     $S \leftarrow$findStoppingPoints($m, n, B_1$)        ▷ use method to get list of stopping points
 7:     let $D$ be a new map from Coordinate to a List of Coordinates
 8:     for $p \in S$ do
 9:         $i \leftarrow p_x$
10:         $j \leftarrow p_y$
11:         let $N$ be a new list of Coordinates
12:         if $(i \neq 0) \vee (\neg B[i][j])$ then
13:             for $x \leftarrow i-2, -1$ do
14:                 if $x = -1$ then        ▷ if it never hits a barrier, it hits the wall so add that
    coordinate to the list of neighbors
15:                     $N$.add($(0, j)$)
16:                 else if $B[x][j]$ then        ▷ otherwise it hit a barrier, so add the stopping
    point next to it to the list of neighbors, then break out of the for loop
17:                     $N$.add($(x+1, j)$)
18:                     break
19:                 end if
20:             end for
21:         end if
22:     end for        ▷ run this procedure for the three other directions. Similarly, run this
    procedure for the coordinate $s$ to ensure it is a key in adjacency list $D$
23:     $D$.put($p, N$)        ▷ add the neighbors to the adjacency list
24:     return isReachable($s, e, D$)
25: end procedure
```

Algorithm 4 Pass Reachability

```
 1: procedure passReachability(Coordinate $s$, Coordinate $e$, Integer $m$, Integer $n$, List
    of Barrier Coordinates $B_1$)
    $B \leftarrow m \times n$ boolean array
    for $b \in B_1$ do
        $B[b_x][b_y] \leftarrow$ true
    end for
    $S \leftarrow$findStoppingPoints$(m, n, B_1)$
    let $D$ be a new map from Coordinate to a List of Coordinates
    for $p \in S$ do
        $i \leftarrow p_x$
        $j \leftarrow p_y$
        let $N$ be a new list of Coordinates
        if $(i \neq 0) \vee (\neg B[i][j])$ then
            for $x \leftarrow i - 2, -1$ do
                if $x = -1$ then
                    $N$.add$((0, j))$
                else if $B[x][j]$ then
                    $D$.add$((x + 1, j))$
                    break
                end if
                $c \leftarrow (x + 1, j)$    ▷ modification to include if it runs over the target square
                if $e = c$ then
                    $D$.add$(e)$
                end if
            end for
        end if
    end for        ▷ run this procedure for the three other directions. Similarly, run this
procedure for the Coordinate $s$ to ensure it is a key in adjacency list $D$
    $D$.put$(p, N)$
    return isReachable$(s, e, D)$
end procedure
```

Algorithm 5 Stop Coverage

```
$i \leftarrow 0$  ▷ index assigned to vertices, will increment every time a new vertex is looked at
let $I$ be a new map from Coordinate to Integer
let $L$ be a new map from Coordinate to Integer
let $O$ be a new map from Coordinate to Boolean    ▷ associate each coordinate with an
index, a lowlink, and a boolean for whether or not its on the stack
let $S$ be a new Stack of Coordinates
let $SCC$ be a new List of List of Coordinates            ▷ strongly connected components
procedure TARJANITERATION(Coordinate $c$)        ▷ recursive method, one iteration of
Tarjan
    $I$.put$(c, i)$                    ▷ assign initial values for index, lowlink, remove from stack
    $L$.put$(c, i)$
    $O$.put$(c,$ true$)$;
    $i \leftarrow i + 1$
    $S$.push$(c)$
    $N = D$.get$(c)$                    ▷ $D$ is the same adjacency list from previous sections
    for $n \in N$ do
```

```
15:         if $I$ does not contain the key $n$ then ▷ if the neighbor hasn't been looked at (no index assigned) iterate Tarjan on it
16:             tarjanIteration($n$)
17:             $m = \min(L.\text{get}(c),\ L.\text{get}(n))$ ▷ then set the new lowlink to be the min of current lowlink and lowlink of neighbor
18:             $L.\text{put}(c, m)$
19:         else if $O.\text{get}(n)$ then ▷ otherwise, if the neighbor is already on the stack
20:             $m = \min(L.\text{get}(c),\ I.\text{get}(n))$ ▷ change the lowlink to the new min lowlink
21:             $L.\text{put}(c, m)$
22:         end if
23:     end for
24:     if $L.\text{get}(c) = I.\text{get}(c)$ then ▷ if the lowlink matches the index
25:         let $nSCC$ be a new List of Coordinates ▷ create a new scc
26:         let $v$ be a new vertex
27:         do: ▷ then keep popping vertices from the top of the stack until we reach the current vertex
28:              $n \leftarrow S.\text{pop}$
29:              $nSCC.\text{add}(v)$
30:         while (c is not n)
31:         $SCC.\text{add}(nSCC)$
32:     end if
33:     for $c \in D.\text{keySet}$ do ▷ for each vertex, iterate Tarjan if it hasn't been already
34:         if $I$ does not contain key $c$ then
35:             tarjanIteration($c$)
36:         end if
37:         let $R$ be a Map from Coordinate to Integer ▷ our list sccs always returns in reverse topological sort, no matter what now map each coordinate to the integer index of the scc in the list
38:         for $i \leftarrow 0$,SCC.size$-1$ do
39:             $s \leftarrow SCC.\text{get(i)}$
40:             for $v \in s$ do ▷ if the vertex is in the scc, put it in the reduction map
41:                 $R.\text{put}(v, i)$
42:             end for
43:         end for
44:         let $R_D$ be a new map from Integers to Sets of Integers ▷ now construct the reduced graph, denoted $R_D$
45:         for $e \in D.\text{entrySet}$ do ▷ for each pair in the original directed adjacency list of coordinate to neighbors
46:             $s \leftarrow e.\text{getKey}$
47:             $N \leftarrow e.\text{getValue}$
48:             $c \leftarrow R.\text{get}(s)$ ▷ get the SCC index corresponding to the original coordinate
49:             if $R_D$ does not contain key $c$ then
50:                 $R_D.\text{put}(c,\text{new set})$
51:             end if
52:             $C \leftarrow R_D.\text{get}(c)$ ▷ add the SCC indices corresponding to each neighbor to the set of neighbors in the reduced graph
53:             for $e \in N$ do
54:                 $c \leftarrow R.\text{get}(e)$
```

```
55:             $C$.add($c$)
56:         end for
57:     end for
58:     $r \leftarrow$ true
59:     for $i \leftarrow 1, R_D$.size$-1$ do    ▷ if there ever isn't a vertex down the chain in the
        topological sort, set it to false
60:         if $R_D$.get($i$) does not contain $i-1$ then
61:             $r \leftarrow$ false
62:             break
63:         end if
64:     end for
65:     return $r$
```

Appendix B

This section gives the construction for $(v_1 \vee v_2 \vee v_3) \wedge (\overline{v}_1 \vee \overline{v}_2 \vee \overline{v}_3)$. Moving from a higher page number to a lower page number in this section signifies "upward movement between layers," and vice versa. Figure 19 below can also be referenced for the order of the layers. Moving up and down or left and right within a single page signifies "upward (or downward) movement" and "left (or right) movement," respectively.

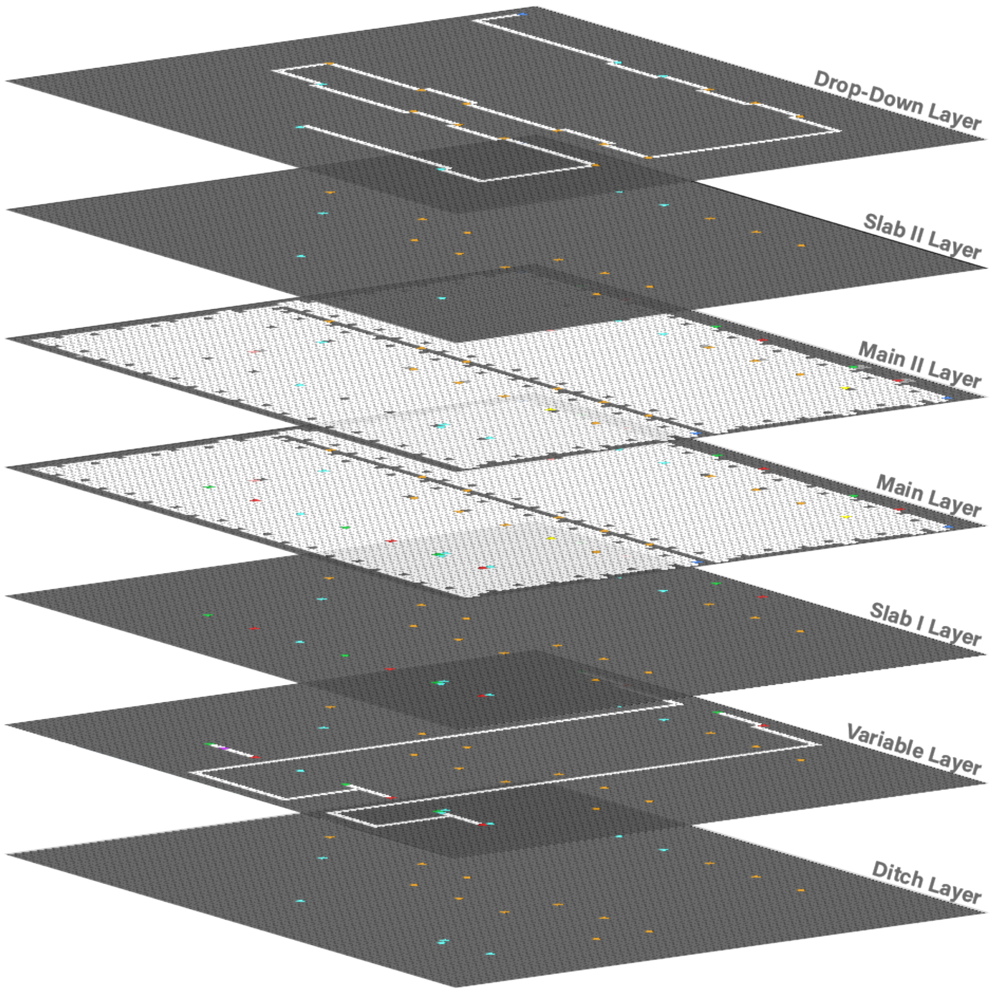


Fig. 19: The order/orientation of the layers. Top and bottom cover layers not shown.

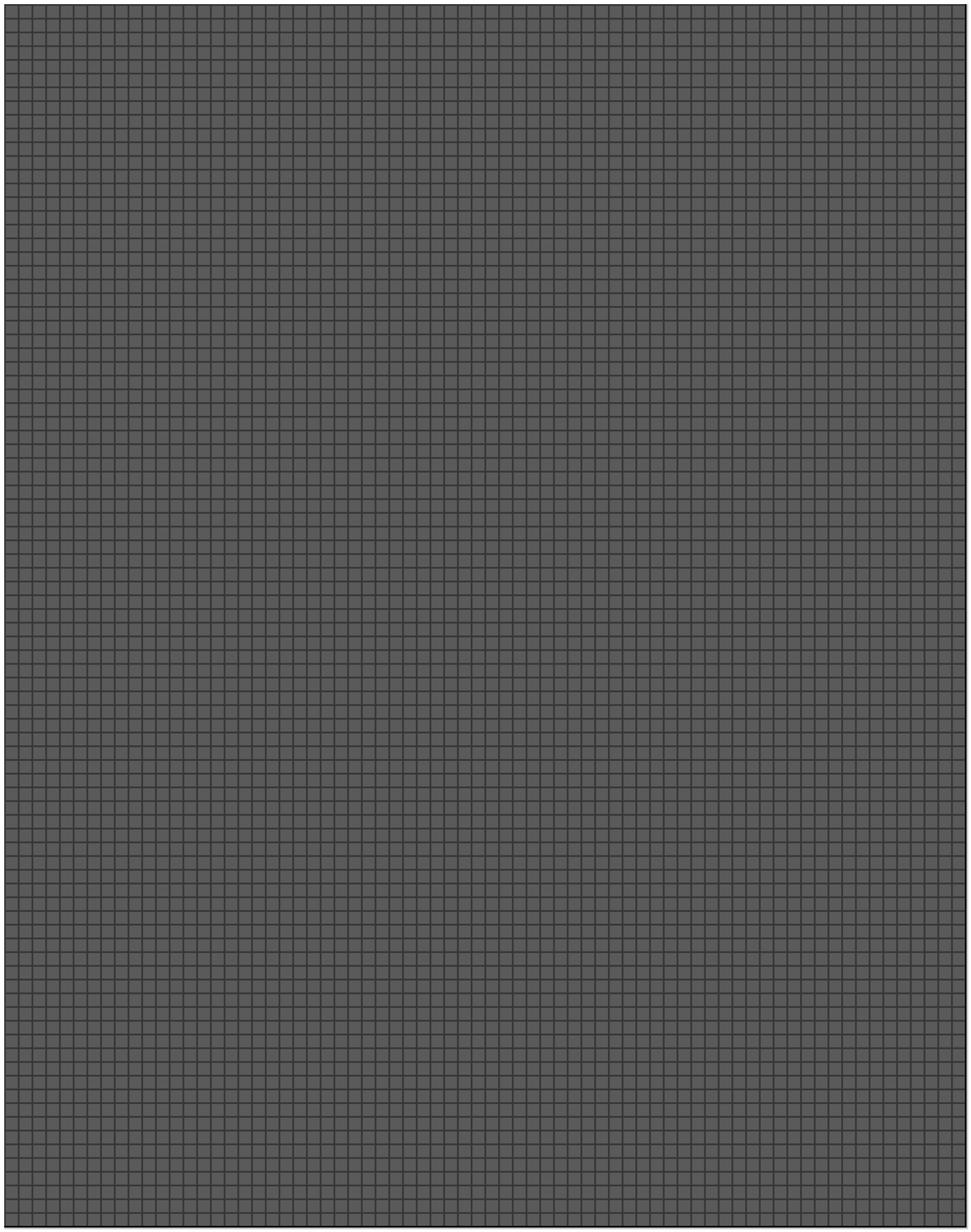


Fig. 20: Top Cover Layer. Dark gray represents a solid block.

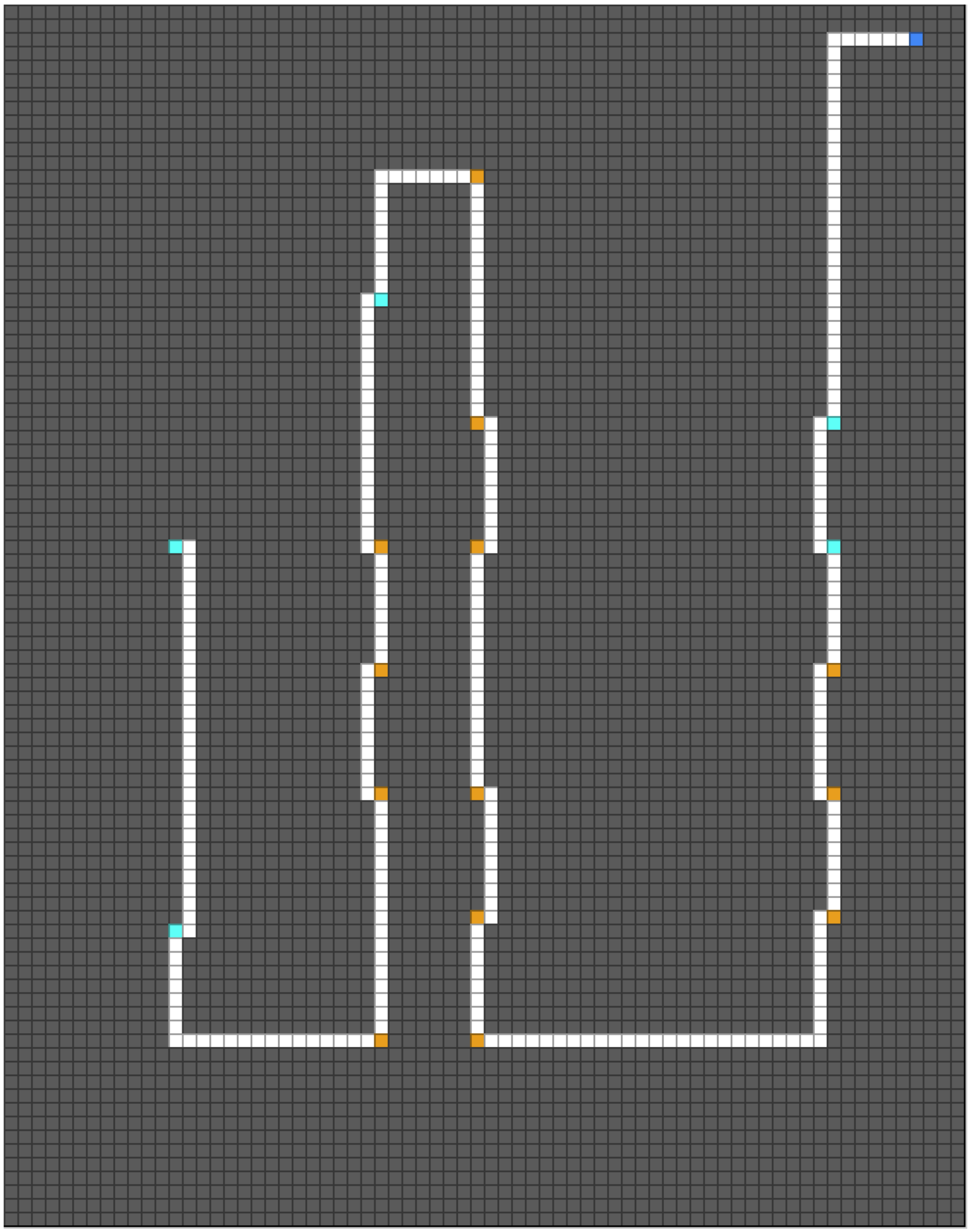


Fig. 21: Drop-Down Layer. Any square that is not dark gray is an empty square (i.e. there is no block in that square). White represents an ordinary traversable space, orange represents a drop-down point into the main layer, light blue represents a drop-down point into the main II layer, and dark blue represents an "elevator square" to reach this layer from lower layers.

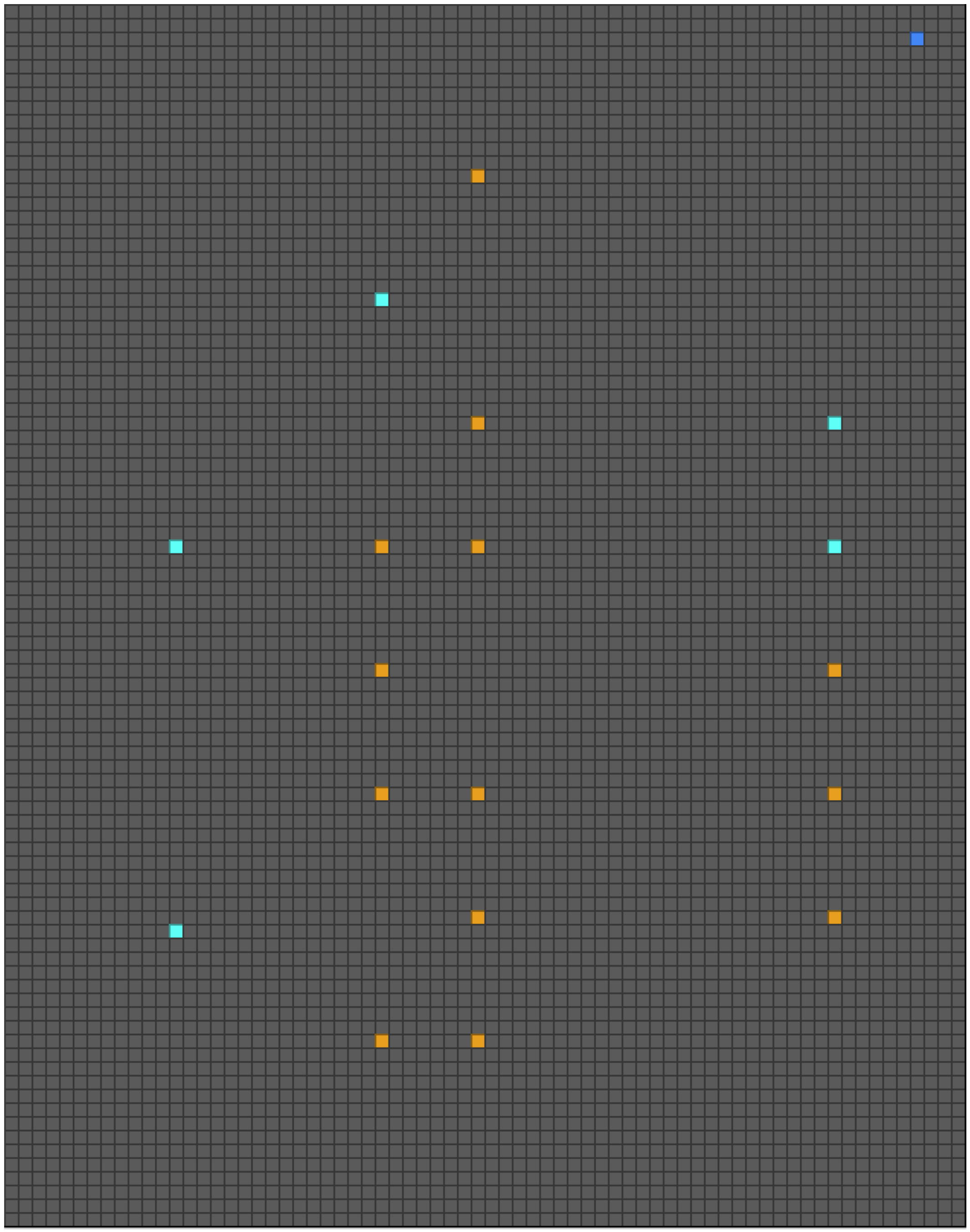


Fig. 22: Slab II Layer.

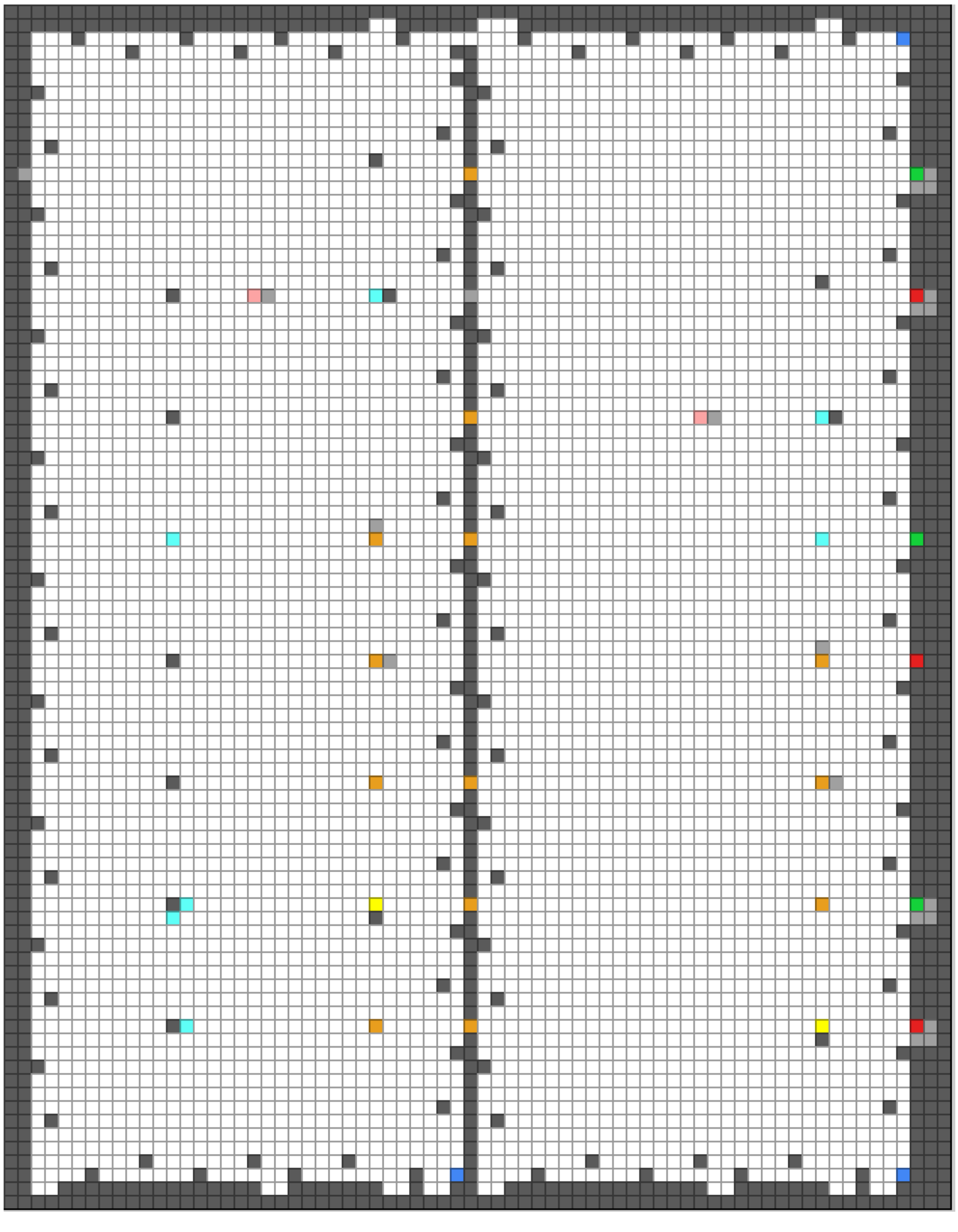


Fig. 23: Main II Layer. Light gray squares represent blocks present in the layer immediately below (main layer) but not in this layer, yellow squares represent clause squares in the layer immediately below (main layer), pink squares represent traversals along a variable truth assignment between the main and main II layers, and green and red squares represent variable squares in the layer immediately below (main layer).

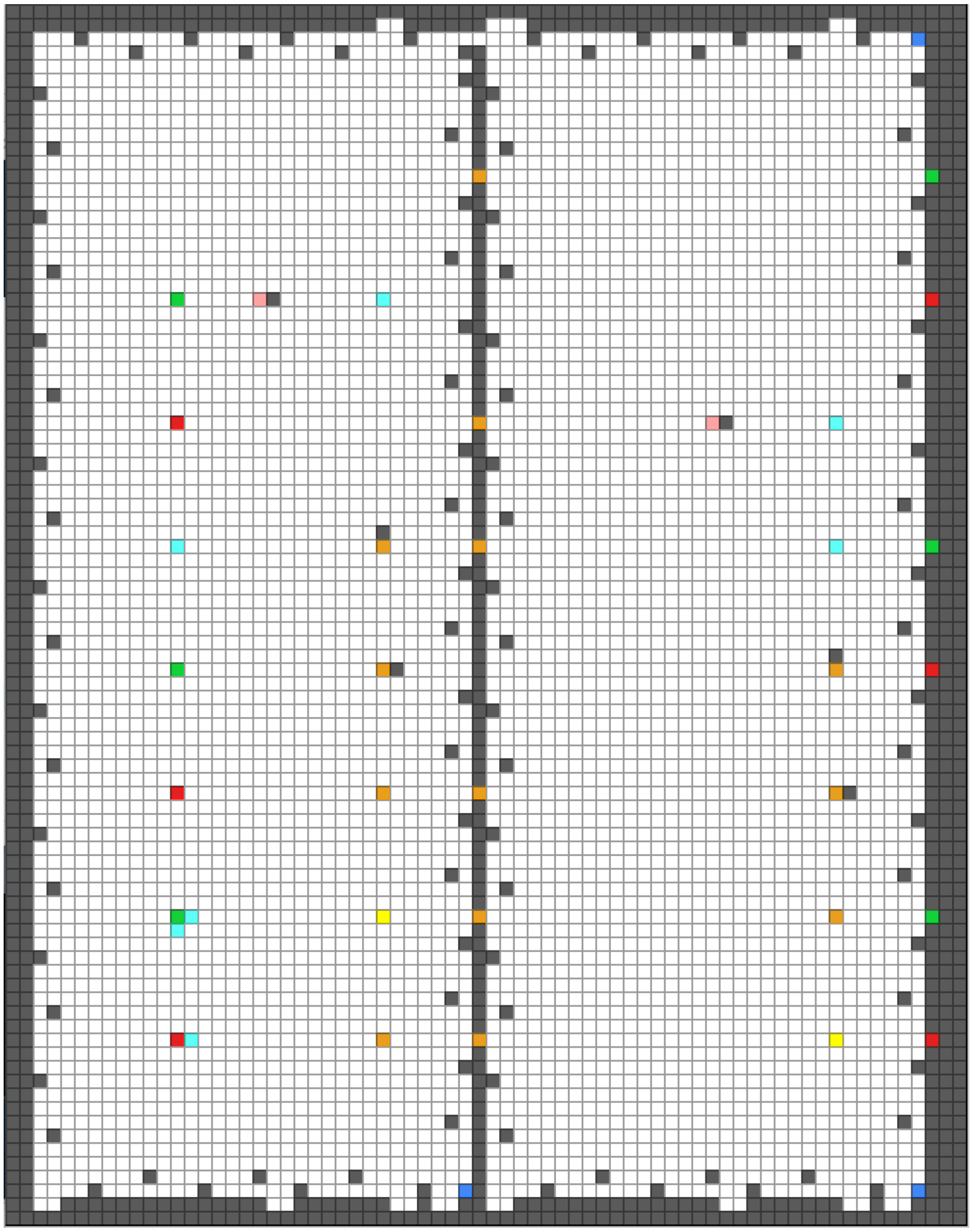


Fig. 24: Main Layer. Green squares represent the “true” state of variables from the variable layer below, red squares represent the “false” state of variables from the variable layer below, and yellow squares represent clause squares. Note that a traversal along the sides of this layer will not pass through the yellow squares, among other squares.

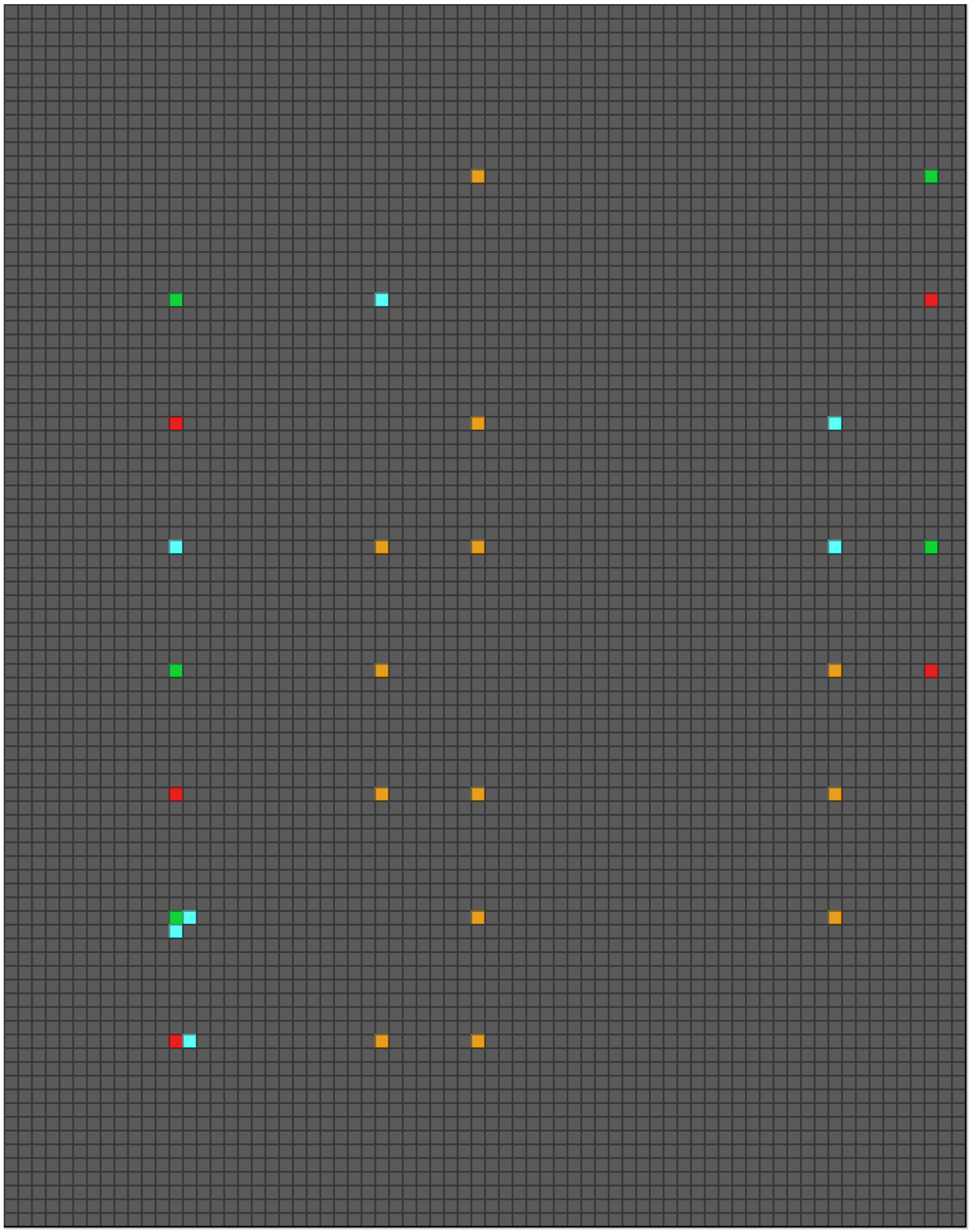


Fig. 25: Slab I Layer.

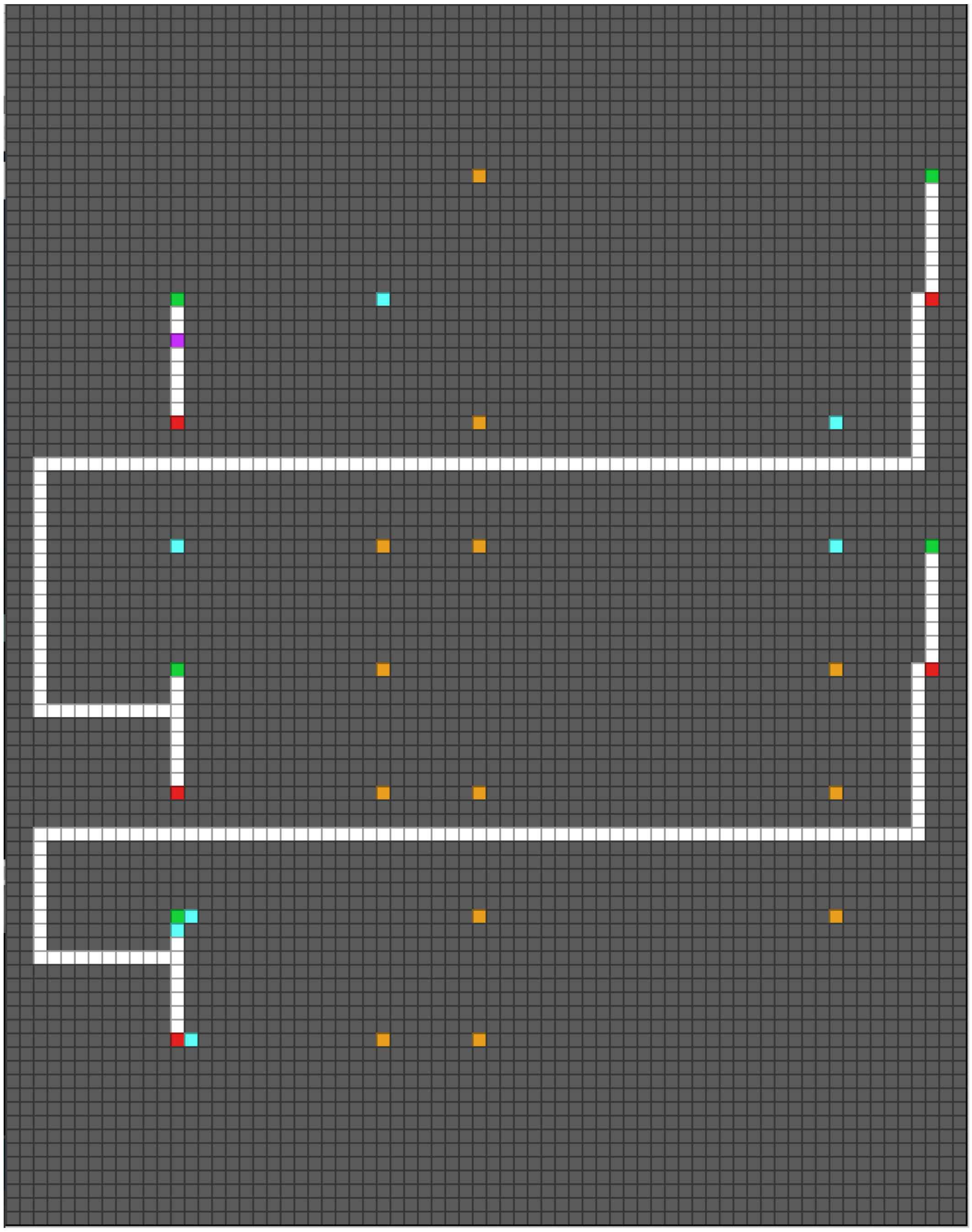


Fig. 26: Variable Layer. The purple square represents the starting square, the green squares represent the “true” state of each variable, and the red squares represent the “false” state of each variable.

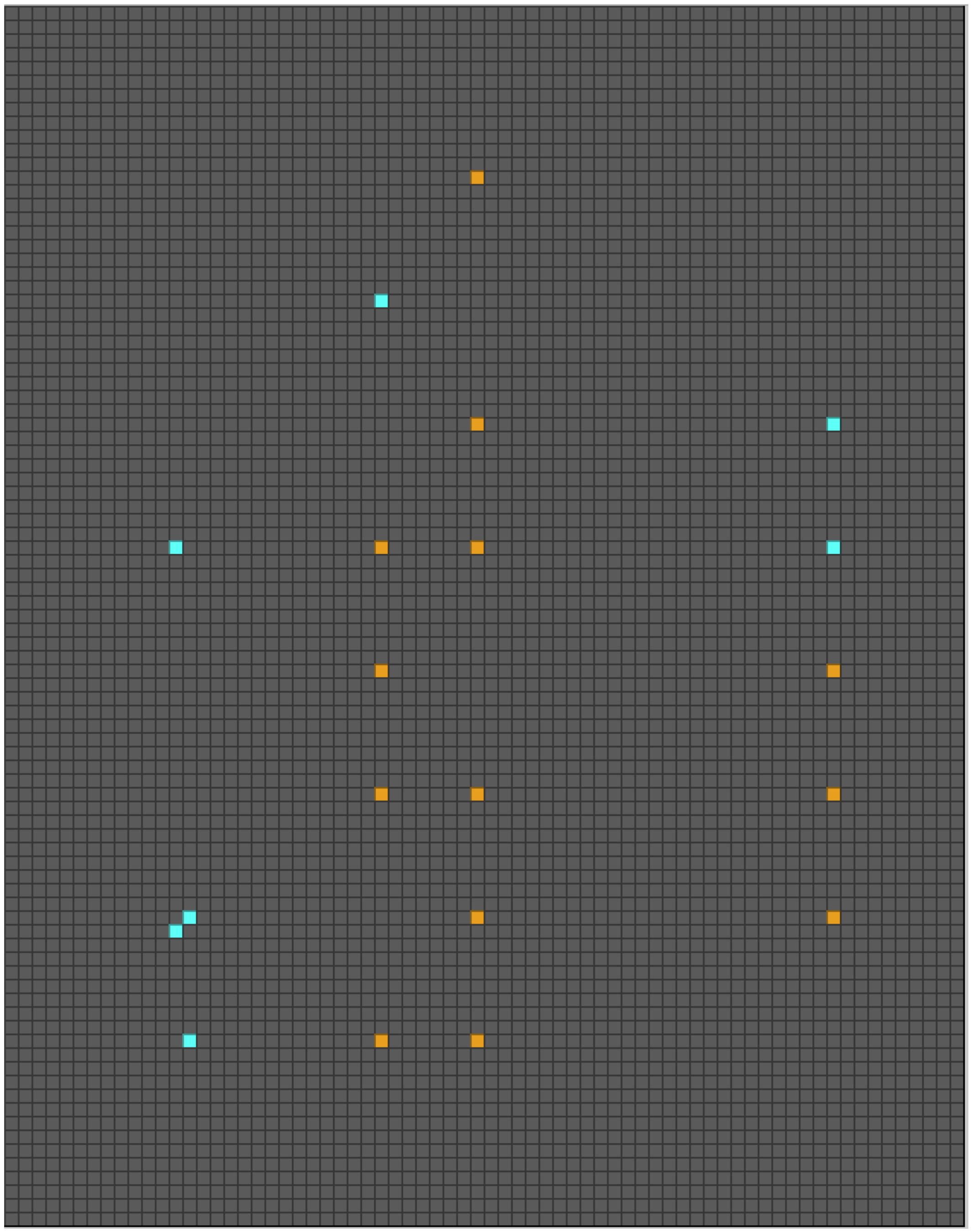


Fig. 27: Ditch Layer. Each drop-down square ends in this layer.

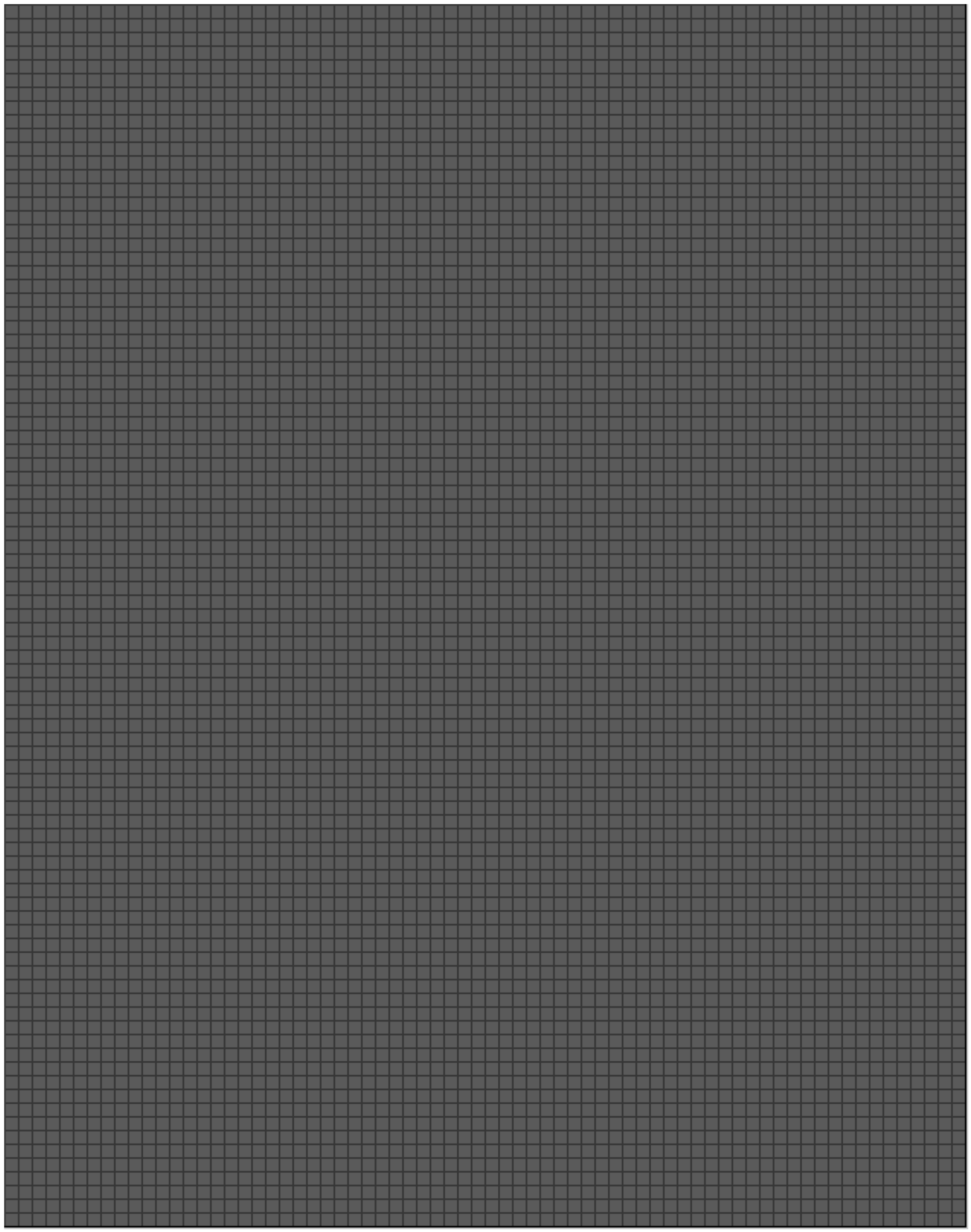


Fig. 28: Bottom Cover Layer.