Hardness of Packing, Covering and Partitioning Simple Polygons with Unit Squares

Mikkel Abrahamsen Jack Stade

April 2024

Abstract

We show that packing axis-aligned unit squares into a simple polygon P is NP-hard, even when P is an orthogonal and orthogonally convex polygon with half-integer coordinates. It has been known since the early 80s that packing unit squares into a polygon with holes is NP-hard [Fowler, Paterson, Tanimoto, Inf. Process. Lett., 1981], but the version without holes was conjectured to be polynomial-time solvable more than two decades ago [Baur and Fekete, Algorithmica, 2001].

Our reduction relies on a new way of reducing from Planar-3SAT. Interestingly, our geometric realization of a planar formula is non-planar. Vertices become rows and edges become columns, with crossings being allowed. The planarity ensures that all endpoints of rows and columns are incident to the outer face of the resulting drawing. We can then construct a polygon following the outer face that realizes all the logic of the formula geometrically, without the need of any holes.

This new reduction technique proves to be general enough to also show hardness of two natural covering and partitioning problems, even when the input polygon is simple. We say that a polygon Q is *small* if Q is contained in a unit square. We prove that it is NP-hard to find a minimum number of small polygons whose union is P (covering) and to find a minimum number of pairwise interior-disjoint small polygons whose union is P (partitioning), when P is an orthogonal simple polygon with half-integer coordinates. This is the first partitioning problem known to be NP-hard for polygons without holes, with the usual objective of minimizing the number of pieces.

Contents

1 Introduction **1**
1.1 Other related work on packing . . . 2
1.2 Technical overview . . . 3

2 Packing in simple polygons **4**
2.1 Schematics of the construction . . . 4
2.2 Reference centers . . . 6
2.3 Components and gadgets . . . 7
2.4 The variable component . . . 7
2.5 Rows and alignment . . . 8
2.6 The PUSH gadgets . . . 9

2.7 The OR gadgets . . . 10
2.8 Verifying the construction . . . 13

3 Covering and partitioning **14**
3.1 Components and gadgets . . . 14

4 Packing in orthogonally convex polygons **17**
4.1 Overview of the construction . . . 17
4.2 Clover-3SAT . . . 18
4.3 Stacks, reference centers, and crossings . . . 23
4.4 Verification order and membrane rows . . . 25
4.5 Variable components . . . 28
4.5.1 Helper rows . . . 29
4.5.2 PUSH gadgets . . . 30
4.5.3 Redundancy columns . . . 31
4.5.4 The complete variable component . . . 33
4.6 Verification of the variable components . . . 35
4.7 Clause components . . . 40
4.7.1 Tester stacks and SWITCH gadgets . . . 40
4.7.2 Defining the clause components . . . 43
4.7.3 Packing the clause components . . . 45
4.8 Verification of the clause components . . . 46
4.9 Proof of Theorem 1 . . . 54

5 Concluding remarks **54**

1 Introduction

Packing is a large field in computational geometry, operations research and pure mathematics concerned with arranging certain geometric shapes without overlap in a space efficient way. The importance of the area is emphasized by numerous industrial settings where packing problems appear, such as in shipping, manufacturing, VLSI design and clothing production.

One of the simplest packing problems is to decide if k axis-aligned unit squares can be placed in a given polygon P without overlap. In this paper, we shall be mainly concerned with the equivalent problem 2×2-Square-Packing: Given a polygon P and an integer k, decide if P has room for k axis-aligned squares of size 2×2. Focusing on 2×2 squares makes it more convenient to state our results and explain our constructions.

It has been known for more than four decades that 2×2-Square-Packing is NP-hard if P can have holes. This was shown in 1981 independently by Fowler, Paterson and Tanimoto [31] and by Berman, Leighton and Snyder [16], later expanded in the paper [15]. The reduction in [15, 16] made use of the at the time recent discovery that Planar-3SAT is NP-hard [42], and the authors thus didn't have to develop a "crossing gadget". The reduction in [31] reduced directly from 3SAT, using crossing gadgets. The reductions are otherwise quite similar in how they get from a 3SAT instance Φ to a polygon: The edges of Φ are turned into corridors that can be packed in two optimal ways, corresponding to the values of a binary variable. Each clause is realized as a carefully designed "room", where one more square can fit if the packing in one of the connected corridors corresponds to a value that makes the clause satisfied.

These works constitute the first published NP-hardness proofs of problems where the input is a polygon that we are aware of.[1] The technique of "building a polygon" on top of (some version of) 3SAT has since been used to show hardness of a great variety of problems, for instance the Art Gallery Problem and other covering problems [47], as well as problems concerned with triangulations [27, 44, 45], partitions [29, 43], tool paths for milling [9, 10], Voronoi games [28], facility location [30], separation of point sets [22], and motion planning [40]. However, the method has the downside that it necessarily leads to a polygon with holes, since each bounded face of the plane embedding of Φ will lead to a hole in the resulting polygon.

While the complexity of 2×2-Square-Packing for polygons with holes was settled early, the complexity remained unknown in the case of *simple* polygons, i.e., polygons without holes. This is Problem 56 in The Open Problems Project [24]. Baur and Fekete [12] conjectured in 2001 that for any fixed integer $s > 1$, there is a polynomial-time algorithm to pack a maximum number of $s \times s$ squares in a simple *grid polygon*, i.e., an orthogonal polygon with vertices at integer coordinates. There have apparently also been some other attempts to resolve the problem, leading to algorithms for special cases and other results [25, 38, 50]. Our main result is that an even more restricted version of the problem is NP-hard. A polygon P is *orthogonally convex* if, for any vertical or horizontal line ℓ, the intersection $P \cap \ell$ is connected. Note that an orthogonally convex polygon is necessarily simple.

Theorem 1. *The problem* 2×2-Square-Packing *is NP-hard, even for orthogonally convex grid polygons.*

[1] The only preceding work with a result of this type seems to be a manuscript by Masek cited by Garey and Johnson [32, p. 232], proving NP-hardness of the problem of describing an orthogonal polygon as a union of a minimum number of rectangles. The manuscript was never published and is now apparently lost.

Allen and Iacono [8] mentioned that this special case of 2×2-Square-Packing was the simplest packing problem with unknown complexity, and that the problem was "most likely in P."

Like the known reductions for polygons with holes, we also reduce from Planar-3SAT. Interestingly, our geometric realization of a planar formula is non-planar, where binary values are represented by configurations of horizontal rows and vertical columns of squares, and these often intersect each other in the interior of the polygon. The crucial observation is that movement in one direction does not influence movement in the other direction, so binary values can be "transported" independently in both directions through a crossing.

The technique proves to be general enough to also show hardness of some other problems. We say that a polygon Q is *small* if Q is contained in an axis-aligned 2×2 square. We show that it is NP-hard to find optimal covers and partitions of a simple polygon using small polygons. In the problem Small-Cover, we are given as input a polygon P and an integer k and want to decide if there exists k small polygons whose union is P. The problem Small-Partitioning is similar, but where we require the k small polygons to be pairwise interior-disjoint. We show that both of these problems are NP-hard, even when P is simple.

Theorem 2. *The problem* Small-Cover *is NP-hard, even for simple grid polygons.*

Theorem 3. *The problem* Small-Partitioning *is NP-hard, even for simple grid polygons.*

There have been many prior examples of covering problems that are intractable for simple polygons, and some are even $\exists\mathbb{R}$-hard like the Art Gallery Problem [3] and covering with convex polygons [2]. It is more remarkable that Theorem 3 gives the first example of a *partitioning* problem that is hard already for simple polygons (with the usual objective of minimizing the number of pieces). For other partitioning problems, like partitioning into convex [19] or star-shaped pieces [4] or trapezoids [11], there are known polynomial-time algorithms. Partitioning problems usually become hard in the presence of holes, which is also the case for the here mentioned problems [11, 46] (a notable exception is partitioning into rectangles, which can be done optimally in polynomial time even for polygons with holes [35]). Like packing, polygon decomposition forms a large subfield in computational geometry, with several books and survey papers that give an overview of the state-of-the-art at the time of publication [18, 20, 36, 37, 46, 48, 51]. Abrahamsen and Rasmussen [6] recently described an 13-approximation algorithm for finding a partition of a simple polygon into a minimum number of small pieces. The problem is motivated by various settings in manufacturing and shipping.

1.1 Other related work on packing

Several other packing problems have been shown to be NP-hard. Here we mention the problem of packing axis-aligned squares of varying sizes into a square [41], packing segments into a simple polygon [39], packing disks into a square [23] and packing 1×3-rectangles (that can be rotated) into an orthogonal polygon with holes [13].

Allen and Iacono [8] showed that it is NP-hard to pack identical simple (small) polygons Q into a simple (larger) polygon P. Here, both Q and P are specified as part of the input. We show that this problem is hard even when Q is the 2×2 square and P is orthogonally convex.

Some packing problems are even known to be $\exists\mathbb{R}$-complete, and thus likely not in NP. Abrahamsen, Miltzow and Seiferth [5] showed that when the pieces can be rotated, the problem of packing convex polygons into a square is $\exists\mathbb{R}$-complete. When the pieces can only be translated,

the problem is $\exists\mathbb{R}$-complete if arcs from hyperbolae can appear on the boundaries of the pieces or the container.

On the positive side, Hochbaum and Maass [34] gave a PTAS for 2×2-Square-Packing in grid polygons with holes. Faster schemes have since then been described by Agarwal, van Kreveld and Suri [7] and Chan [17]. El-Khechen [38] and van Renssen and Speckmann [50] described families of simple grid polygons where 2×2-Square-Packing can be solved optimally in polynomial time. El-Khechen, Dulieu, Iacono and van Omme [25] showed that 2×2-Square-Packing into grid polygons with holes is in NP. This is not immediately clear since, if P is given in the standard representation as the coordinates of the vertices in binary, then the number of squares that fit in P can be exponential in n (the number of vertices), so specifying the placements of the squares is not a valid certificate.

Aamand, Abrahamsen, Ahle and Rasmussen [1] proved that packing dominoes, i.e., rectangles of size 1×2 that can be rotated, into a given grid polygon P with holes is polynomial-time solvable. Already when we go to 2×2-squares or 1×3-rectangles, the problem becomes NP-hard, and as we show in this paper for 2×2 squares, this is even the case for *simple* grid polygons.

1.2 Technical overview

We present two constructions for packing. The first construction shows that 2×2-Square-Packing is NP-hard for simple grid polygons. The second construction shows that this remains true if the polygon is restricted to be orthogonally convex. The second result is strictly stronger than the first, but the proof is much more complicated, so we believe it is justified to give the constructions separately.

Our constructions are made possible by two key ideas. The first idea is a way to constrain what a packing can look like locally, even in parts of the polygon that are far from the boundary. We show that we can define a certain set of 1×1 squares called *reference centers* in the polygon in such a way that any 2×2 square in the polygon with integer coordinates must contain a reference center. A packing is *perfect* if there are as many squares as there are reference centers. It is then straightforward to verify how the individual parts of our construction can be packed, since for each reference center, there are just four relevant 2×2 squares to consider, namely one containing the reference center in each of the quadrants.

The second idea is a new way of drawing a planar graph where the vertices become rows and the edges become columns. We reduce from Planar-3SAT, and our drawing provides a schematic for our construction. Rows and columns are allowed to cross in these schematics, but the planarity of the graph ensures that the endpoints of all rows and columns are incident to the outer face of the drawing. This makes it possible to construct a simple polygon, following the boundary of the outer face, that "implements" the functionality of the rows and columns.

Variables are represented by neighbouring rows of squares that can be packed in essentially only two different ways, corresponding to the truth values of a variable. Dependency between variables is created with so-called PUSH gadgets, where a column of *push squares* will be pushed up or down depending on the value of a variable. A crucial observation is that a column of push squares can cross any number of other variable rows without interacting with them. Only the variable where the column ends will be affected by the push squares.

Our first construction, described in Section 2, shows that 2×2-Square-Packing is NP-hard for simple grid polygons. The construction is not so involved and quite straightforward to verify.

In Section 3, we study the problems Small-Cover and Small-Partitioning. A construction very similar to the one for packing proves that Small-Cover is NP-hard, establishing Theorem 2. A main difference is that in packing, information propagate by squares *pushing* each other, because they are not allowed to overlap, while in covering, the squares are *pulling* each other, because they must cover everything. We also show that in a satisfiable instance, an optimal covering has a nice property that allows it to be converted into a partition of the same cardinality, hence proving Theorem 3.

In Section 4, we then turn our attention to packing 2×2 squares in an orthogonally convex grid polygon. This is a strictly stronger result than the one from Section 2, that requires a different, much more complicated construction. We develop the problem Clover-3SAT, which is a modification of Planar-3SAT under which our schematics have an orthogonally convex shape. This builds on work by Pilz [49], who showed that a related problem called Variable-Clause-Linked-Planar-3SAT is NP-hard.

The variable and clause components are built out of a large number of criss-crossing rows and columns. Since it is harder to isolate the different pieces, the dependencies between different parts of the construction are much more complicated. We need a long technical verification to ensure that the packings behave as we intend.

2 Packing in simple polygons

2.1 Schematics of the construction

We reduce from Monotone-Planar-3SAT, as introduced by de Berg and Khosravi [14]. An instance of 3SAT is *monotone* if in each clause, all literals are positive or all literals are negative. An instance $\Phi = (F, G)$ of Monotone-Planar-3SAT consists of a monotone instance F of 3SAT with variables $x_1, \ldots, x_k$ and clauses $c_1, \ldots, c_\ell$ and a plane graph G where

- the vertices of G are $\{x_1, \ldots, x_k, c_1, \ldots, c_\ell\}$,
- the edges of G are $\{x_i c_j \mid x_i \in c_j \vee \neg x_i \in c_j\} \cup C$, where $C = \{x_1x_2, \ldots, x_{k-1}x_k, x_kx_1\}$,
- the cycle C separates all positive clauses from all negative clauses.

The following lemma was proved in [14].

Lemma 4. Monotone-Planar-3SAT *is NP-complete.*

Let Φ be an instance of Monotone-Planar-3SAT with variables $x_1, \ldots, x_k$. We now describe schematically the overall construction of a grid polygon P, so that P has a packing with a certain number of 2×2 squares if and only if Φ is satisfiable. The first step is shown in Figure 1 (middle). We make a horizontal segment for each variable $x_1, \ldots, x_k$ in this order bottom-up, where the x-coordinates of the right endpoints are non-decreasing, as are those of the left endpoints. Furthermore, the right endpoint of x_1 is to the right of the left endpoint of x_k, so that all the segments have a common horizontal overlap. These are the *main* variable rows.

The positive clauses are represented as rows above x_k and the negative as rows below x_1. In the embedding of G, a clause can be *nested* inside another, defining a partial order on the clauses. For instance, in Figure 1, c_2 and c_3 are nested inside c_1. If a positive clause c_i is nested inside c_j, then we draw the row of c_i above that of c_j in the schematics. If instead the clauses are negative, the

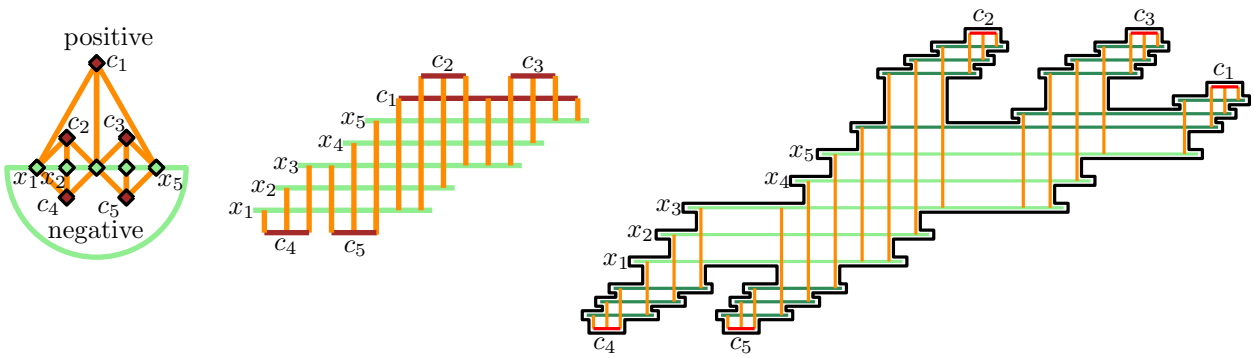

Figure 1: The figure shows schematically how to convert an instance of Monotone-Planar-3SAT into a polygon. Left: The instance of Monotone-Planar-3SAT that we start with. Middle: We realize variables and clauses as rows. Right: We replace the clause rows with dark green auxiliary variables connected to red OR gadgets. The boundary of our constructed polygon is sketched in black.

row of c_i is below that of c_j. Each edge $x_i c_j$ is realized as a vertical segment connecting the rows of x_i and c_j. All vertical segments to the positive clauses are placed to the right of all those to the negative clauses. The left-to-right ordering of the edges to the positive clauses along the variables $x_1, \ldots, x_k$ in G is preserved by the corresponding vertical segments, as well as the ordering of the edges to the negative clauses. The row of a positive (resp. negative) clause c_j starts at the top (resp. bottom) endpoint of the vertical segment corresponding to the leftmost edge incident to c_j in G and ends at the top (resp. bottom) endpoint of the rightmost edge.

In the next step, as shown in Figure 1 (right), we replace each clause row by three *auxiliary* variable rows, one for each of the variables connected to the clause. In the right side of the auxiliary rows, we connect them using vertical segments to an OR gadget, represented by a red horizontal segment. As sketched in the figure, we construct a polygon whose boundary approximately follows the outer face of the resulting arrangement, but the finer details will be given later.

The use of auxiliary variables is necessary to make sure the OR gadgets can be placed appropriately, so that they are not crossed by other columns and without the need of creating holes in the final polygon. In order to not introduce any holes in our polygon, it is important that every endpoint of a segment be incident to the outer face of the arrangement induced by the segments. Indeed, the endpoints correspond to some "functionality" that we need to realize using an appropriate polygon boundary.

Lemma 5. *There is a schematics as described, where the segments representing the OR gadgets are not crossed by any edges, and all segment endpoints are incident to the outer face of the drawing.*

Proof. We proceed by induction on the number of clauses. The claim is trivial with no clauses. Consider a formula Φ with n clauses and suppose inductively that the claim holds for $n-1$ clauses. Let c be a clause of maximum depth of nestedness in Φ and let Φ' be Φ with c removed. By the induction hypothesis, we can consider a schematics of Φ' with the stated properties. Without loss of generality, we consider the case where c is a positive clause, so in the schematics, c should be realized with three auxiliary variable rows connected to an OR gadget, and the auxiliary variables should be above the other OR gadgets. We can stretch the schematics of Φ' horizontally to make enough room for the columns connecting the main variable rows and the new auxiliary variable

rows. These columns then appear consecutively along the main variable rows. We can then also draw the auxiliary variable rows and the OR gadget. We thus avoid crossing through other OR gadgets and we keep all segment endpoints incident to the outer face of the drawing. □

Algebraically, we make use of the following equivalence when introducing auxiliary variables:

$$(x_i \vee x_j \vee x_k) \iff \Big(\exists y_i, y_j, y_k : (y_i \implies x_i) \wedge (y_j \implies x_j) \wedge (y_k \implies x_k) \wedge (y_i \vee y_j \vee y_k)\Big) \tag{1}$$

Here, x_i, x_j, x_k are the original variables and y_i, y_j, y_k are the auxiliary variables introduced for that particular clause. For a negative clause, we use an analogous equivalence:

$$(\neg x_i \vee \neg x_j \vee \neg x_k) \iff \Big(\exists y_i, y_j, y_k : (\neg y_i \implies \neg x_i) \wedge (\neg y_j \implies \neg x_j) \wedge (\neg y_k \implies \neg x_k) \wedge (\neg y_i \vee \neg y_j \vee \neg y_k)\Big) \tag{2}$$

2.2 Reference centers

By the following lemma, we can restrict our attention to packings where the squares have integer coordinates. The lemma appears to be folklore, but is also proved in [12].

Lemma 6. *If a grid polygon can be packed with k axis-aligned 2×2 squares, then such a packing can be chosen where the coordinates of the vertices of all the squares are integers.*

Proof. Consider a packing with k squares that minimizes the sum over all squares of the sum of both coordinates of the square center. □

We are going to construct a grid polygon P based on the formula Φ. The unit squares of the form $[2k-1, 2k] \times [2\ell-1, 2\ell]$ for $k, \ell \in \mathbb{Z}$ that are contained in P are called the *reference centers*.

Lemma 7. *In a packing of a grid polygon using 2×2 squares with integer coordinates, each square covers exactly one reference center.*

Proof. The statement follows by inspection of the four combinations of the parity of the coordinates of a 2×2 square with integer coordinates. □

We get from Lemma 7 that there can be at most as many squares as reference centers in any packing. We say that a packing is *perfect* if it has squares of integer coordinates and consists of as many squares as there are reference centers. This means that for each reference center, there are only 4 possible ways for a square in a perfect packing to cover the reference center, i.e., the reference center must be in one of the four quadrants of the square. This makes it straightforward to verify the individual parts of our construction, at least in principle.

A square in a perfect packing *pushes up* (resp. *down*, *left*, *right*) if the reference center is contained in one of the lower (resp. upper, right, left) quadrants. For clarity, we will draw the reference centers in our diagrams. We use a system of color coding that describes some of the constraints on how that tile can move, as shown in Figure 2. Squares with pink reference centers will always push down and left, while squares with red reference centers will always push up and left. Squares with orange reference centers can move vertically but will always push left. Squares with green reference centers can move horizontally but will always push down. Squares with dark blue reference centers will always push up. Squares with light blue reference centers will have some freedom in both horizontal and vertical directions.

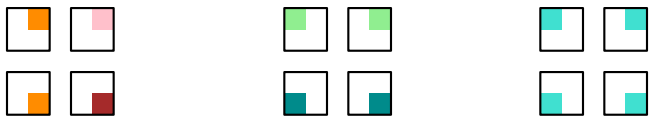

Figure 2: The color key.

2.3 Components and gadgets

We informally distinguish between *gadgets* and *components* in our constructions, where a gadget is a few edges of the boundary creating some simple functionality, and a component is thought of as a whole region of the polygon that can involve an arbitrary number of gadgets.

First, we describe a *variable component* in Section 2.4, which has two possible positions representing the values of a binary variable. The variable components are represented in these schematics as green horizontal rows. The two positions of a variable component are *plus* and *minus*, corresponding to the values true and false of a binary variable.

Next, we create *PUSH gadgets* in Sections 2.5 and 2.6. A PUSH gadget is a section of the polygon boundary that interacts with a variable component. Each PUSH gadget pushes on a *push column*, which is shown as an orange column in our schematics. We describe a PUSH-UP-IF-MINUS and a PUSH-DOWN-IF-PLUS gadget. Consider a variable component x that is below another y. We can make a PUSH-UP-IF-MINUS gadget on x and a PUSH-DOWN-IF-PLUS on y and a connection between these using a push column. Since a push column cannot be pushed both up and down at the same time, we have ensured that x is plus or y is minus, so we have made the implication $y \implies x$.

Note that variable components only push on push columns via the presence of PUSH gadgets. Without a PUSH gadget, a push column and a variable component will cross each other without interacting.

The last gadget is an OR gadget, as described in Section 2.7. We represent this as a red row in our schematics. An OR gadget interacts with the top of three push columns and always pushes down on at least one of them. Unlike the variable components, a push column *cannot* pass through an OR gadget; any push column that touches an OR gadget should terminate there. This is why we have attached the OR gadget using auxiliary rows, as in Figure 1 (right).

2.4 The variable component

Consider a variable row x in the schematics. We make a corresponding variable component that contains a pair of rows of squares. At the ends of the rows, the polygon boundary constrain the rows to always be unaligned, as shown in Figure 3. In between the ends, there is a PUSH gadget for each edge to the variable row x in the schematics. The PUSH gadgets are described in Section 2.6 below.

Lemma 8. *In any perfect packing, one row of the variable component pushes right and the other pushes left.*

Proof. We first consider the two leftmost squares in the variable component and note that the polygon boundary forces at least one of them to push to the right. Similarly, (at least) one of the two rightmost squares must push to the left. So all the squares in one of the rows must push to the left and all the squares in the other row must push to the right. □

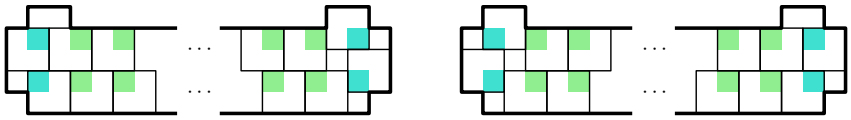

Figure 3: The ends of the variable component. The plus position is shown left and the minus is shown right. Some squares in the middle may be pushed up.

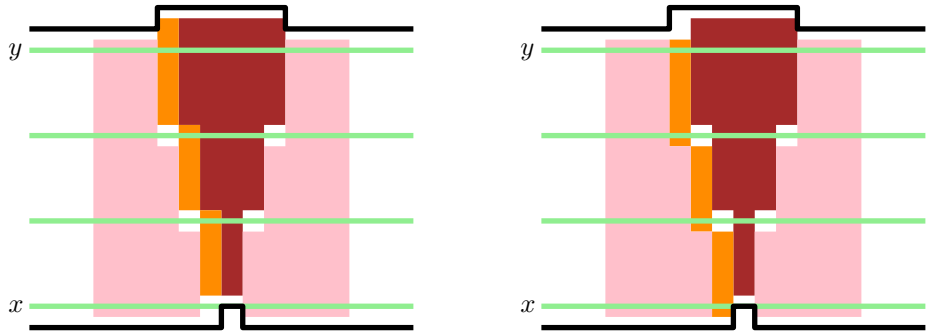

Figure 4: Schematic of how to make a dependency between two variable components. When crossing each variable component in between, the width of the pyramid grows by two squares. The push column is shown in orange. Depending on the position of the bottom variable component x, this column may be pushed up (left). Depending on the position of the top variable component y, it may be pushed down (right). Since the column can't be pushed both up and down, this creates a constraint between x and y.

As we have seen, a perfect packing of the variable component has two possible positions, and they correspond to the values of a binary variable. The transition from one position to the other corresponds to all the squares rotating one step either clockwise or counterclockwise around the cycle formed by the squares. We say that *plus* is the position where the squares are rotated in the positive (i.e., counterclockwise) direction, and *minus* is the position where they rotated in the negative direction.

2.5 Rows and alignment

We describe a way to make a dependency between two variable components x and y, where x is below y, as shown schematically in Figure 4. This is done by an upside down pyramid of squares that are raised by 1 unit. The squares are raised by a PUSH gadget of the lower variable x. The pyramid should be able to cross variable rows of other variables in between x and y without interacting with them.

One extra layer of squares on the left edge of the pyramid may move up or stay down depending on the positions of x and y. The squares in this layer are called a *push column* or *push* squares. This layer acts like a wire connecting the bottom and top edges of the polygon that bound the variable components x and y, respectively. The gadgets described in Section 2.6 work by pushing on the push squares. This structure is pyramid shaped because a stack of raised squares has to increase in width every time it passes between a pair of rows with opposite horizontal alignments.

Recall that each variable component consists of one pair of neighbouring unaligned rows. We want to control the width of the pyramid at the top, which means that the number of times it crosses

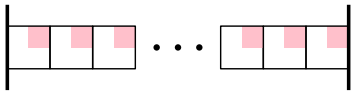

Figure 5: A static row. Some squares may be pushed up, but all push left.

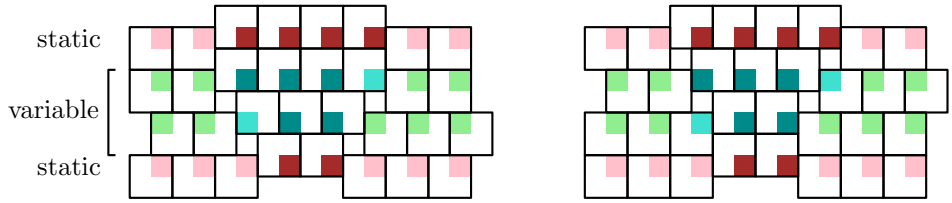

Figure 6: A pyramid crossing a variable component. The size of the pyramid always grows by 2 squares, regardless of the position of the variable component.

a pair of unaligned rows should not depend of the positions of the individual variable components crossed by the pyramid. To do this, we place each variable component between two *static* rows such that, in either position of the variable component, one of the static rows is aligned with the adjacent row in the variable component and the other is unaligned. Figure 5 shows a static row. In each static row, the rightmost square is pushed to the left by an edge of the polygon, so the whole row of squares are in a left position.

Figure 6 shows in detail how the width of a pyramid grows by two squares whenever it crosses a variable component. This is expressed by the following lemma.

Lemma 9. *Consider a perfect packing. If a block of consecutive squares in a static row below a variable component pushes up, then the corresponding squares and their two horizontal neighbors in the static row above the variable component must also push up. If a block of consecutive squares in an upper static row pushes down, then the corresponding squares and their two horizontal neighbors in the static row below must also push down.*

Proof. We prove the first claim; the other one is analogous. A static row is always left-aligned with respect to the reference centers. One of the rows of the variable component is right-aligned, which causes one more square to the left of that row to rise, as compared to the row below. In the next row (which is either the upper variable row or the static row above the variable), one more square to the right is then also pushed up. □

2.6 The PUSH gadgets

Consider a variable x that is part of a positive clause in Φ. We then have an auxiliary variable y above x, and we need to make the implication $y \implies x$. We create an upside-down pyramid from x to y that has a column of push squares on the left. This creates the constraint that both gadgets cannot simultaneously push on the push column, i.e.,

$$x \text{ does not push up on its push square} \vee y \text{ does not push down on its push square.}$$

It hence suffices to make gadgets ensuring

$$\begin{aligned} x \text{ pushes up on its push square} &\iff x \text{ is minus}, \quad \text{and} \\ y \text{ pushes down on its push square} &\iff y \text{ is plus} \end{aligned}$$

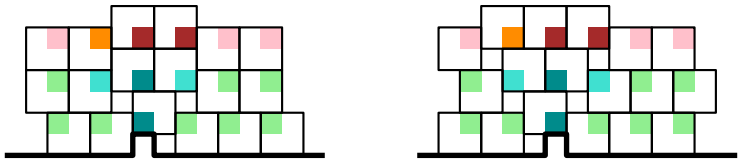

Figure 7: The PUSH-UP-IF-MINUS gadget. The gadget forces some number of squares in the static row (top, red) to rise, and requires an additional push square to the left to rise if the variable component (bottom two rows, green) is minus. The plus position is shown on the left and the minus position is shown on the right.

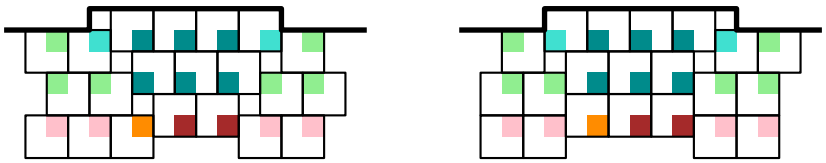

Figure 8: The PUSH-DOWN-IF-PLUS gadget. The gadget allows some number of squares in the static row (bottom, red) to rise, and *allows* the square to left of these to rise only if the variable component (top two rows, green) is plus. Note that the top gadgets may need to be much wider since the constraint pyramid could increase in width many times.

To this end, we make the two gadget *PUSH-UP-IF-MINUS* and *PUSH-DOWN-IF-PLUS*, respectively. Figure 7 shows the PUSH-UP-IF-MINUS gadget, which simply consists of small indent in the polygon wall that forces some squares to rise. Figure 8 shows the PUSH-DOWN-IF-PLUS gadget, which consists of an *out*dent in the polygon wall allowing some squares to rise. The width of this outdent is adjusted according to the width of the push pyramid, which, as stated by Lemma 9, depends on the number of variable components it crosses. This also ensures that the pyramids don't collide in the interior of the polygon.

Suppose now that x is part of a negative clause. We then have an auxiliary variable y below x, and need to realize the implication $\neg y \implies \neg x$, or equivalently $x \implies y$. We can thus realize this implication using the gadget PUSH-DOWN-IF-PLUS on x and PUSH-UP-IF-MINUS on y.

Figure 9 shows an example of how to make a dependency using these two gadgets. Out of the four possible combinations of positions of the top and bottom variable components, three are allowed by the constraint. Two of these have one gadget pushing on the push column, while the third has neither gadget pushing its push square.

In conclusion, we have described gadgets with the properties stated by the following lemma.

Lemma 10. *In any perfect packing, the following holds. If a variable is minus, the push square of any PUSH-UP-IF-MINUS gadget pushes up. If a variable is plus, the push square of any PUSH-DOWN-IF-PLUS gadget pushes down. As a consequence, combining a PUSH-UP-IF-MINUS gadget and a PUSH-DOWN-IF-PLUS gadget, we can realize the implications $y \implies x$ and $\neg y \implies \neg x$, i.e., ensure that the values of variables encoded by the packing satisfy the implications.*

2.7 The OR gadgets

We create two types of OR gadgets, namely positive OR gadgets for the positive clauses and negative for the negative clauses. Figure 10 shows the positive OR gadget. The OR gadget is connected to

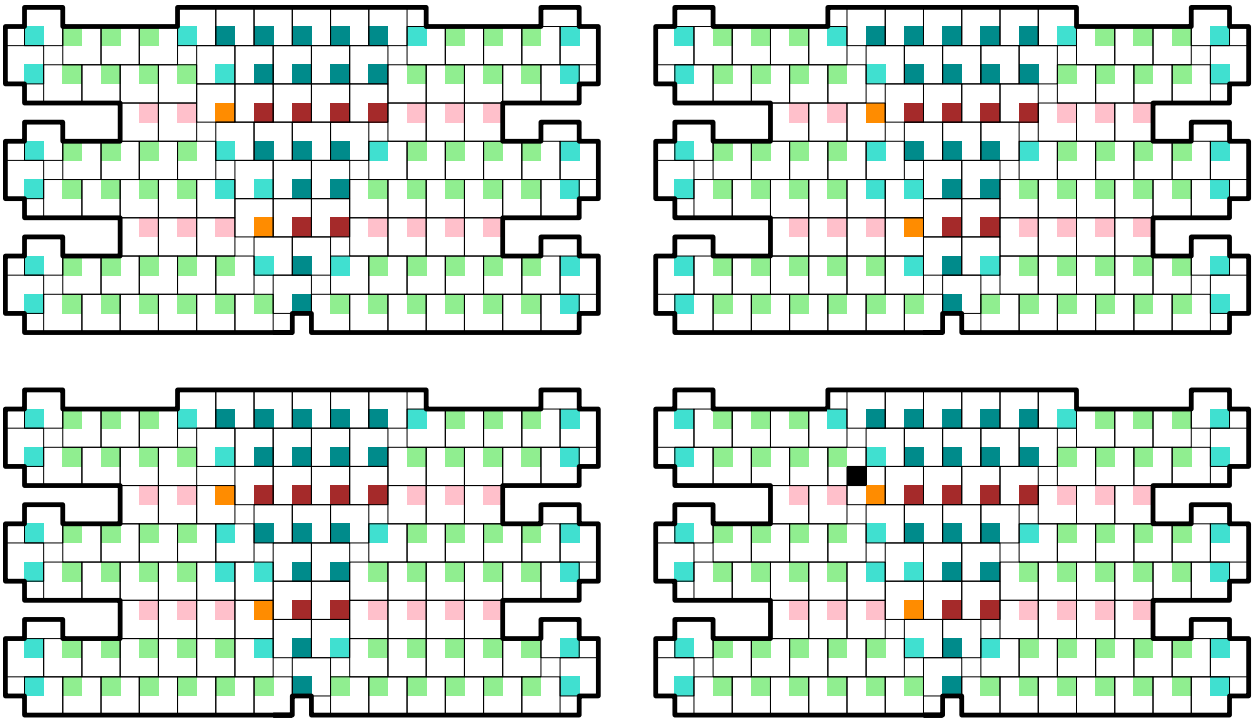

Figure 9: Example of a dependency using the PUSH-UP-IF-MINUS gadget on the bottom variable and the PUSH-DOWN-IF-PLUS gadget on the top variable. Top left: The bottom and top variables are both minus. Top right: The bottom and top variables are both plus. Bottom left: The bottom variable is plus and the top is minus. Since no gadget pushes on the push square, this introduces some slack so there are more than one perfect packing. Bottom right: The bottom variable is minus and the top is plus, which causes an overlap at the black square.

three auxiliary variables that are below the gadget, using upside down pyramids of raised squares with push columns, as shown in Figure 11. These pyramids are created by PUSH-UP-IF-MINUS gadgets on the auxiliary variables. If all three push columns push up, then it is impossible to pack the OR gadget. On the other hand, Figure 12 shows that it is possible to pack the OR gadget if one or more of the columns is not pushed up. In this way we can think of the OR gadget as being a gadget that pushes down on one of three input columns. Hence, one of the auxiliary variables must be plus.

Figure 13 shows the negative OR gadget. Here we connect the auxiliary variables using PUSH-DOWN-IF-PLUS gadgets, as shown in Figure 14. We use a slightly different variant of the PUSH-DOWN-IF-PLUS gadget than what is shown in Figure 8: We shift the outdent one unit square to the right, which causes the column of push squares to be on the right side of the pyramid instead of the left. This allows for a slightly simpler OR gadget than if we were to use the usual push column on the left side of the pyramids. By inspection, we obtain the following lemma.

Lemma 11. *In a positive OR gadget, at least one of the connected auxiliary variables is plus. In a negative OR gadget, at least one of the connected auxiliary variables is minus.*

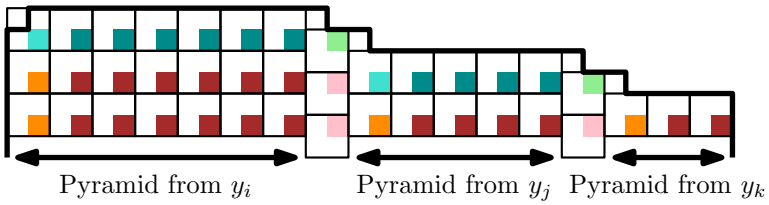

Figure 10: The positive OR gadget. In this figure, all the orange push columns are pushed up, meaning that we can't pack all the squares into the OR gadget. The labels y_i, y_j, y_k refer to Equation (1).

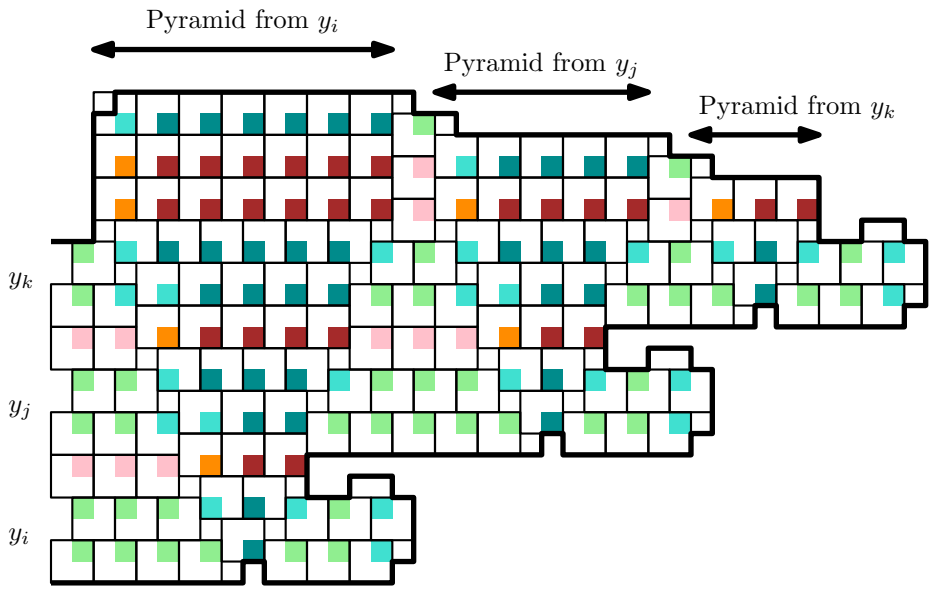

Figure 11: The figure shows how the auxiliary variable components are connected to the OR gadget using pyramids. Here is shown the situation where all variables are minus, so there is no perfect packing and the top left square sticks out.

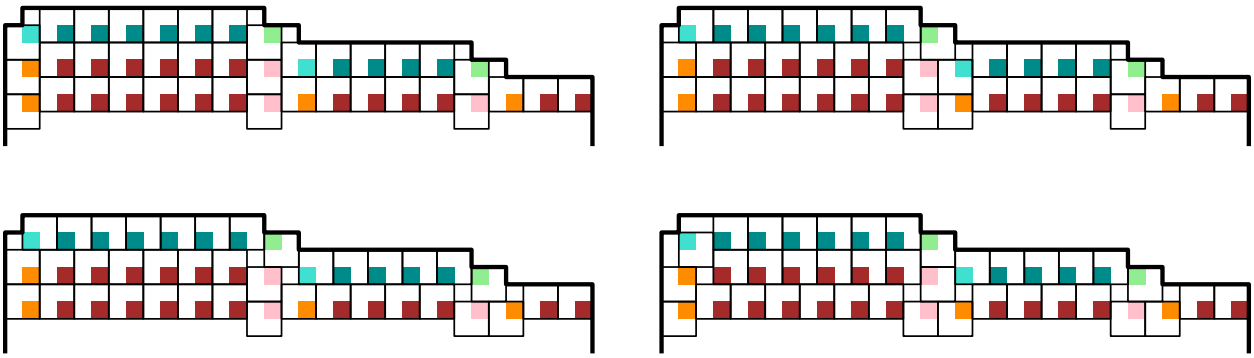

Figure 12: Satisfying assignments of the OR gadget. As in Figure 10, we say that the push columns represent values y_i, y_j, y_k from left to right, respectively. The top two and the bottom left diagrams show packings where the push squares of y_i, y_j, y_k are down, respectively. To the bottom right is shown the situation that all three are down, which creates some slack so that there are more perfect packings.

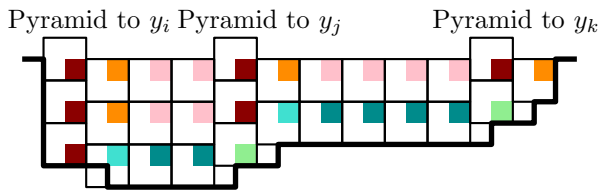

Figure 13: The negative OR gadget. In this figure, all the orange push columns are pushed down, meaning that we can't pack all the squares into the OR gadget. The labels y_i, y_j, y_k refer to Equation (2).

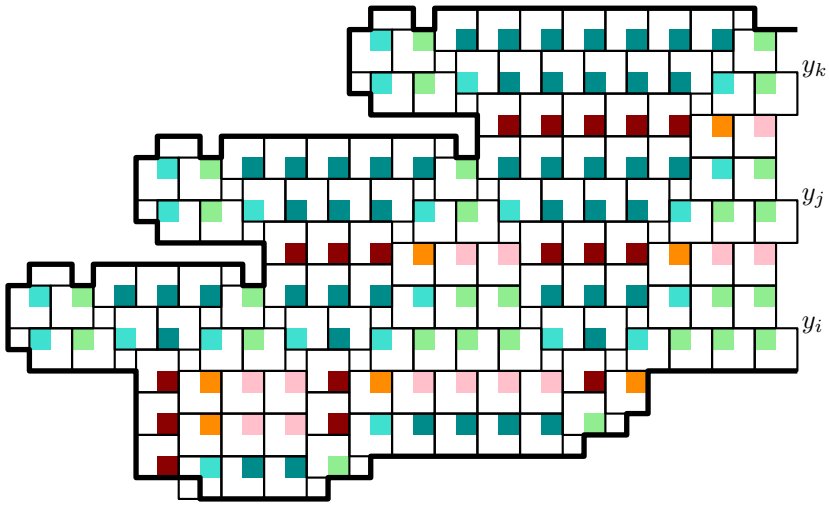

Figure 14: The figure shows how the variable components are connected to the negative OR gadget. Here is shown the situation where all variables are plus, so there is no perfect packing, and the bottom left square sticks out.

2.8 Verifying the construction

It is clear from our diagrams that a perfect packing exists when the instance is satisfiable, so it remains to check that every perfect packing corresponds to a satisfying assignment of Φ, as established by the following lemma.

Lemma 12. *If there exists a perfect packing of P, then Φ is satisfied.*

Proof. By Lemma 8, each variable component encodes a value of one of the variables x_i of Φ or an auxiliary y_i used for an OR gadget. Consider a clause $C(x_i, x_j, x_k)$ of Φ. By Lemma 11, the packing encodes values of the auxiliary variables y_i, y_j, y_k that satisfy the corresponding clause $C(y_i, y_j, y_k)$. It now follows from Lemma 10 that $C(x_i, x_j, x_k)$ is also satisfied. Hence, the full formula Φ is likewise satisfied. □

It is clear that our polygon can be constructed in polynomial time from the instance Φ. We have thus proven the following theorem.

Theorem 13. *The problem* 2×2-Square-Packing *is NP-hard, even for simple grid polygons.*

3 Covering and partitioning

The proof of Theorem 2 is almost analogous to that of Theorem 13, but we need to modify the gadgets and replace "push" by "pull" in many places.

Let us first show a connection between covering and partitioning. Recall that we define a polygon Q to be *small* if Q is contained in an axis-aligned 2×2 square.

Lemma 14. *Suppose a polygon P is contained in the union of k axis-aligned 2×2 squares in such a way that the intersection of P with each square is a star-shaped polygon with the kernel containing the center of the square. Then P can be partitioned into k small polygons.*

Proof. Consider the L_∞ distance Voronoi diagram for the set of centers of squares. For each square S with center c and Voronoi region V, we use $V \cap P$ as a piece. We need to show that $V \cap P$ is small, and it suffices to show that (i) $V \cap P \subseteq S$ and (ii) $V \cap P$ is connected. Part (i) follows since the squares cover P, so every point in P is within a distance of 1 from the center of some square. For part (ii), note that since V is a Voronoi cell, V is a star-shaped polygon with a kernel containing c, and $P \cap S$ is also a star-shaped polygon with a kernel containing c by assumption. Hence, $V \cap P = V \cap (P \cap S)$ is also a star-shaped polygon and therefore connected. □

Note that axis-aligned unit squares could be replaced by circles, or indeed any symmetric convex shape (as long as rotations are not allowed), and the same result can be obtained by changing the metric used. The optimal coverings considered in the proof of Theorem 2 satisfy the conditions of the lemma, implying Theorem 3.

Consider a polygon P and a set $\mathcal{S}$ of axis-aligned 2×2 squares so that $P \subseteq \bigcup \mathcal{S}$. We say that $\mathcal{S}$ is a *square cover* for P. In our construction, we adopt the use of reference centers, as described in Section 2.2. Since the area of reference centers covered by one 2×2 square is exactly 1, we know that $|\mathcal{S}| \geq k$, where k is the number of reference centers. We say that the cover $\mathcal{S}$ is *perfect* if $|\mathcal{S}| = k$.

We construct a polygon P based on a Monotone-Planar-3SAT instance Φ so that P has a perfect square cover if and only if Φ is satisfiable. For our polygon P it holds that if a perfect square cover $\mathcal{S}$ exists, then for each square $S \in \mathcal{S}$, the polygon $P \cap S$ is either S or three quadrants of S, in particular $P \cap S$ is connected. Hence, Φ is satisfiable if and only if there are k small polygons whose union is P, so Small-Cover is NP-hard. As the polygons $P \cap S$ have that form, we also know by Lemma 14 that Φ is satisfiable if and only if there is a partition of P into k small polygons. Hence, Small-Partitioning is also NP-hard. In the following, we therefore analyze how a perfect square cover for P must look.

Analogous to Lemma 6, we have the following.

Lemma 15. *If an orthogonal polygon with integer coordinates has a square cover of size k, then such a square cover can be chosen where the coordinates of the vertices of all the squares are integers.*

This again allows us to consider only four possible squares for each reference center, with the reference center in each of the quadrants.

3.1 Components and gadgets

Due to the similarity with our construction for packing, we shall give a less detailed description of the construction. Figure 15 shows the variable component, and we have the following analogue of Lemma 8.

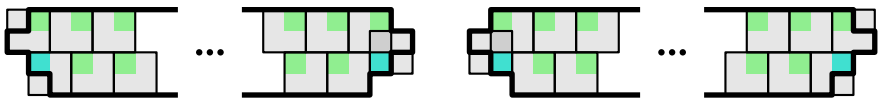

Figure 15: Variable component for the covering problem.

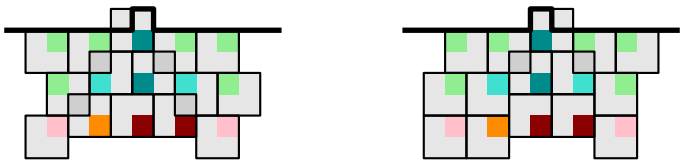

Figure 16: The PULL-UP-IF-PLUS gadget. The gadget pulls up a pyramid of squares, and the orange pull square is also pulled up in the plus position.

Lemma 16. *In any perfect cover, one row of the variable component pulls left and the other pulls right.*

Proof. One row must pull left to cover the left end of the variable component, and the other must pull right to cover the right end. □

Figures 16 and 17 shows the PULL-UP-IF-PLUS and PULL-DOWN-IF-MINUS gadgets. Instead of a column of push squares, we now have a column of *pull squares*. Note that a PULL-UP-IF-PLUS gadget on a variable y forces a pyramid of squares to pull up. Similarly as in packing, we can make a matching PULL-DOWN-IF-MINUS gadget on a variable x below y, resulting in a dependency between the variables. We have the following analogy of Lemma 10.

Lemma 17. *In any perfect covering, the following holds. If a variable is plus, the pull square of any PULL-UP-IF-PLUS gadget pulls up. If a variable is minus, the pull square of any PULL-DOWN-IF-MINUS gadget is pulls down. As a consequence, combining a PULL-UP-IF-PLUS gadget and a PULL-DOWN-IF-MINUS gadget, we can realize the implications $y \implies x$ and $\neg y \implies \neg x$, i.e., ensure that the values of variables encoded by the packing satisfy the implications.*

Figures 18 and 19 show the positive OR gadget, and Figures 20 and 21 show the negative. By inspection, we get the following lemma, identical to Lemma 11.

Lemma 18. *In a positive OR gadget, at least one of the connected auxiliary variables is plus. In a negative OR gadget, at least one of the connected auxiliary variables is minus.*

We conclude with the following analogue of Lemma 12 (which has an analogous proof).

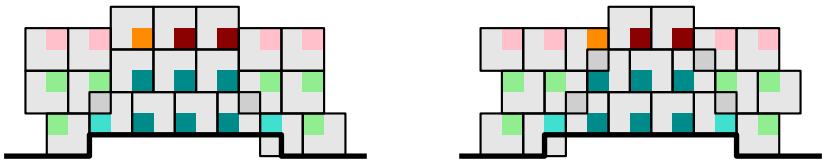

Figure 17: The PULL-DOWN-IF-MINUS gadget. The gadget allows some number of squares to pull up and requires the orange pull square to stay down in the minus position.

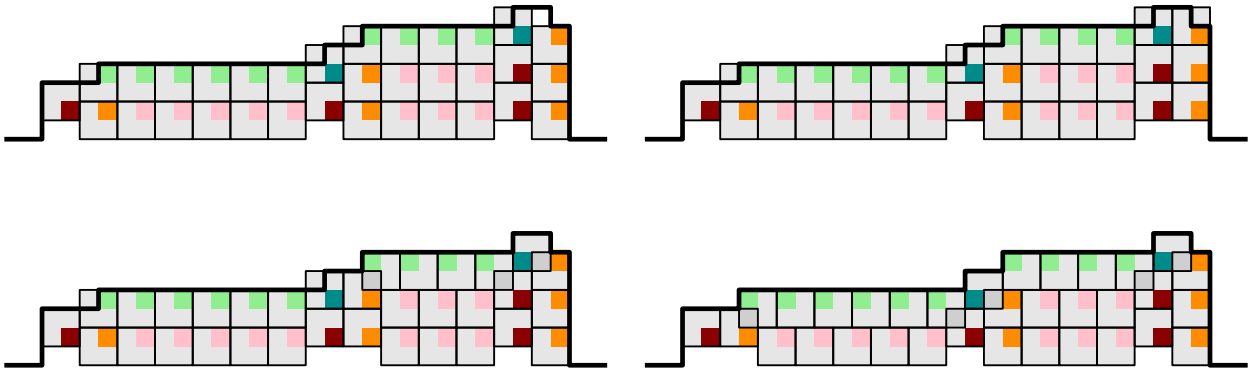

Figure 18: The positive OR gadget. Top left: The situation where all the orange pull squares pull down, and a unit square at the top is not covered. The other three: If one pull square goes up, everything can be covered.

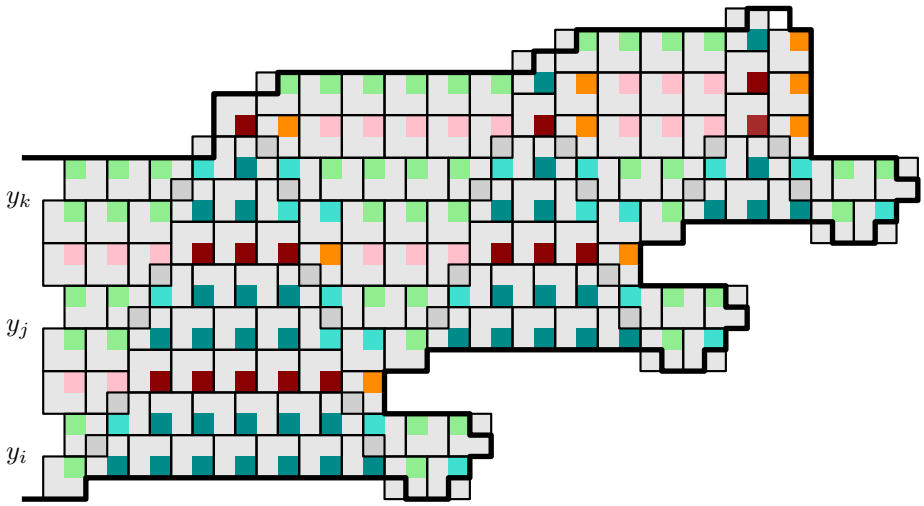

Figure 19: Connecting the auxiliary variables to the positive OR gadget using PULL-DOWN-IF-MINUS gadgets. We have made a slight adjustment to the PULL-DOWN-IF-MINUS gadgets as compared to Figure 17 in order to have the pull squares in the right side of the pyramids.

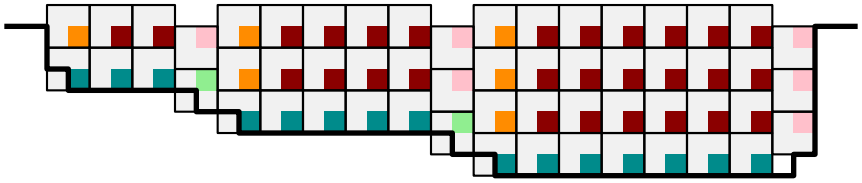

Figure 20: The negative OR gadget. We show the situation where all the orange pull squares pull up, and a unit square at the bottom is not covered. If one pull square goes down, everything can be covered.

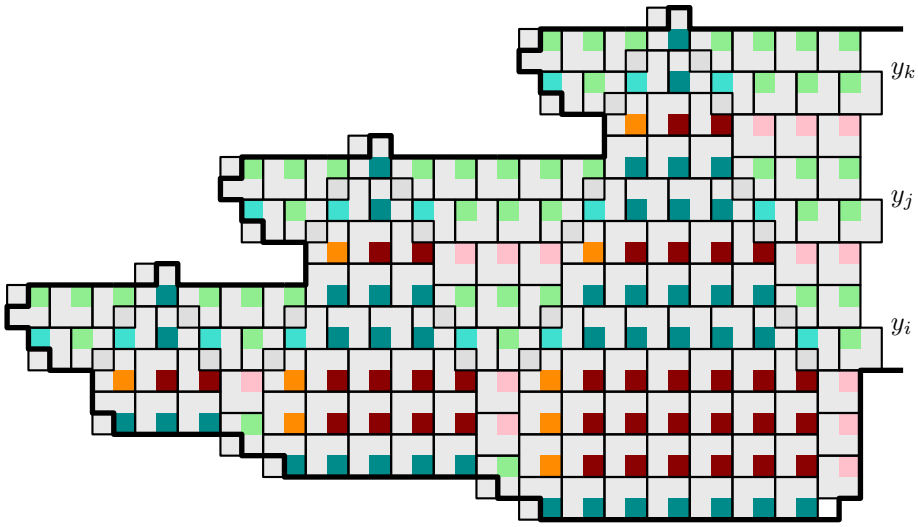

Figure 21: Connecting the auxiliary variables to the negative OR gadget using PULL-UP-IF-PLUS gadgets.

Lemma 19. *If there exists a perfect covering of P, then Φ is satisfied.*

We have then shown Theorem 2. Note that if a perfect cover exists, then so does a perfect cover where the intersection of P with any of the squares S is either S or three of the quadrants of S. Hence, we get Theorem 3 from Lemma 14.

4 Packing in orthogonally convex polygons

Finally, we show that 2×2-Square-Packing for orthogonally convex grid polygons is also NP-hard. This is a strictly stronger result than Theorem 13, but the proof is so much more difficult that it seems better to give it separately. Recall that a polygon P is *orthogonally convex* if for any horizontal or vertical line ℓ, the intersection $P \cap \ell$ is connected.

4.1 Overview of the construction

The idea is again to convert the incidence graph of an instance of 3SAT into a schematic, which then forms the structure of the polygon. The schematics that we use in Section 2.1 are usually not going to have an orthogonally convex shape. We refine Planar-3SAT in such a way that the schematics generated *are* orthogonally convex. We call this new problem Clover-3SAT. This is described in Section 4.2.

In Section 4.5, we show that we can make variable components in the orthogonally convex setting. The idea is to form parts of the "boundary" of the gadgets with squares in the packing rather than with the actual polygon boundary. Unsurprisingly, this make the verification of the variable components much more complicated. One key idea is that we can add redundancy to the constraints that form the variable component, so that one variable component can be verified without having already verified all the others.

Instead of having isolated OR gadgets, we use much more complicated *clause components*. The clause components are described in Section 4.7. The clause components create a set of constraints

that are not obviously equivalent to the 3SAT instance. Verifying the clause gadgets requires a detailed algebraic analysis, which is given in Section 4.8.

There will be two sections of border between the clause and variable components. These are formed by static rows that we call the *membrane rows*. The clause and variable components are each verified independently, with any interactions controlled by some simple properties of squares in the membrane rows.

4.2 Clover-3SAT

The first step in our construction in Section 2 for simple polygons was to convert a planar graph to a schematic that has a row for each variable and clause and a column for each edge, as shown in Figure 1. In general, it isn't possible to convert an instance of Planar-3SAT into a schematic that is suitable for generating an orthogonally convex polygon. Instead, we will have to consider a further restriction of Planar-3SAT. In this section, we define the problem Clover-3SAT, show that it is NP-hard, and show that an instance can be converted into a schematic that has an orthogonally convex shape.

An instance of Clover-3SAT consists of the following:

Input: An instance Φ of 3SAT containing variables $x_1, \ldots, x_n$, two sets of clauses $c_1, \ldots, c_p$ and $d_1, \ldots, d_q$, and a planar embedding of the graph G that contains a vertex for each x_i, c_i or d_i and:

- An edge (x_i, c_j) whenever x_i or $\neg x_i$ appears in c_j, and similarly for the clauses d_j.
- Edges (x_i, x_{i+1}) for $i = 1, \ldots, n-1$, (c_i, c_{i+1}) for $i = 1, \ldots, p-1$, and (d_i, d_{i+1}) for $i = 1, \ldots, q-1$.
- Edges (x_1, c_1), (c_1, x_n), (x_n, d_1), and (d_1, x_1).

Question: Is Φ satisfiable?

We say that an instance of Clover-3SAT is *monotone* if each clause c_i is positive (i.e., it is of form $x_j \vee x_k \vee x_\ell$) and each clause d_i is negative (i.e., it is of form $\neg x_j \vee \neg x_k \vee \neg v_\ell$). The problem Monotone-Clover-3SAT is Clover-3SAT restricted to monotone instances.

Clover-3SAT is a restriction of (variable-linked) Planar-3SAT with an additional constraint that it should also be possible to link some of the clauses as well. Figure 22 shows an example of the "clover" graph that appears in this definition.

In 2018, Pilz [49] showed that Planar-3SAT remains hard if there is a cycle that passes through all the clause vertices and then all of the variable vertices. This is called Variable-Clause-Linked-Planar-3SAT. The reduction is from Planar-3SAT, but unlike either variable-linked or clause-linked Planar-3SAT, the reduction requires modifying the original graph. By reducing instead from Monotone-Planar-3SAT, Pilz also shows that this problem is hard when all the edges inside the cycle represent positive literals and all the edges outside the cycle represent negative literals.

Variable-Clause-Linked-Planar-3SAT is closely related to Clover-3SAT, and the proof of hardness follows Pilz [49] closely. In particular, both proofs use something like what we call Layered-Planar-3SAT as an intermediate step. An instance of Layered-Planar-3SAT consists of:

Input: An instance Φ of 3SAT and a planar, integer-coordinate straight line embedding of the incidence graph G where the variable vertices have even y coordinates, the clause vertices have odd y coordinates, and each edge is between a pair of vertices whose y-coordinates differ by 1.

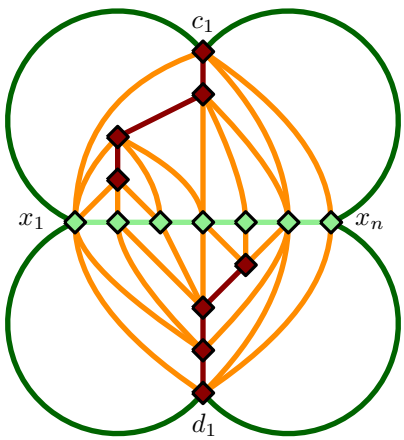

Figure 22: A graph coming from an instance of Clover-3SAT. The 4 outer edges (x_1, c_1), (c_1, x_n), (x_n, d_1), and (d_1, x_1) exist to prevent constraint edges from wrapping around to the other side of the variables. When drawn like this, all the edges for c constraints hit the variable vertices from above and all the edges for d constraints hit variable vertices from below.

Question: Is Φ satisfiable?

We call a problem instance *monotone* if clause vertices with y coordinates of the form $4k+1$ are positive and clause vertices with y coordinates of the form $4k-1$ are negative. The problem Layered-Planar-3SAT restricted to monotone instances is called Monotone-Layered-Planar-3SAT.

Lemma 20. Monotone-Layered-Planar-3SAT *is NP-hard.*

Proof. Our proof closely follows Pilz [49]. We reduce from an instance Φ of Monotone-Planar-3SAT.

We draw the incidence graph with integer-coordinate vertices, straight edges, and no crossings. Since we have a cycle through the variable-vertices, we can draw this graph in such a way that the variable-vertices all have y-coordinate 0, the clause vertices have odd y-coordinates, and the positive clauses have positive y-coordinates while the negative clauses have negative y-coordinates. The size of the coordinates needed is no more than polynomial in the size of Φ.

Next, we split each edge whenever it crosses a layer. The process of splitting edges will (temporarily) cause some clauses to contain both negated and non-negated literals. In order to end up with a monotone instance, the edge splitting process will preserve the following properties:

- Variable-vertices have even y-coordinates
- Clause-vertices have odd y-coordinates
- If a variable-vertex has y coordinate $i \equiv 0 \pmod 4$ and is connected to a clause vertex with y-coordinate k, then the sign of that variable in that clause is $\text{sign}(k-i)$
- If instead the variable vertex has coordinate $i \equiv 2 \pmod 4$, then the sign of that variable in that clause is $\text{sign}(i-k)$

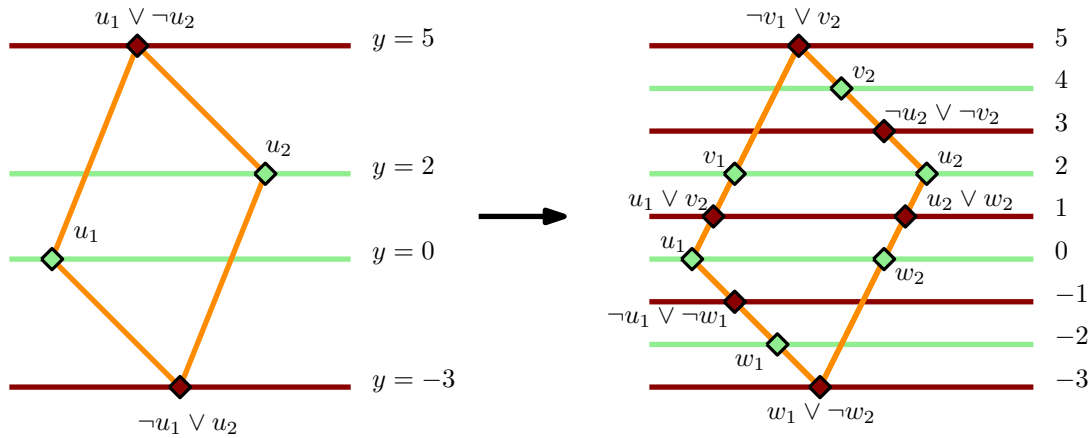

Figure 23: Splitting edges to form a layered planar graph. For our purposes, a 3SAT instance has *at most* 3 vertices per clause.

This is satisfied by the initial configuration since the variable-vertices initially all have y-coordinate 0. Whenever an edge spans a y-distance of more than 1, we can split this edge in a way that preserves this property. Figure 23 shows the 4 cases that can occur depending on the y-coordinate of the variable vertex and whether the clause vertex is above or below it. For example, suppose a variable-vertex v has y-coordinate $i \equiv 0 \pmod 4$ and is connected to a clause-vertex c with y-coordinate $k > i + 1$. We add a variable u with y-coordinate $i + 2$ and a clause $d = (v \vee u)$ with y coordinate $i + 1$. Replace v in c with $\neg u$. The new clause d is equivalent to $\neg u \implies v$, so the new 3SAT instance is equivalent, and we have reduced the the total number of times that an edges crosses a layer by 2. The other cases are shown in Figure 23.

We repeat this process until all edges are between adjacent layers. If a clause has y-coordinates of form $4k + 1$ then variables from both adjacent layers must appear non-negated, and if a clause has y-coordinates of form $4k - 1$ then variables from both adjacent layers appear negated. So the result is indeed a monotone instance. □

These layered planar graphs appear in the proof by Pilz [49]. They are then wrapped up in a spiral to obtain a graph with the variable-clause-linked property. To obtain a clover graph, we wrap a layered planar graph in a spiral in a slightly different way.

Lemma 21. Monotone-Clover-3SAT *is NP-hard.*

Proof. We can add edges to a layered planar graph from an instance of Monotone-Layered-Planar-3SAT to turn it into a clover graph, as shown in Figure 24 (left). This can be illustrated more clearly by wrapping the layered planar graph graph around in a spiral, as shown in Figure 24 (right). One of the clause paths goes through all the positive clause vertices and the other goes through the negative clause vertices, so this produces a monotone instance of Clover-3SAT. □

The planar embedding of the clover graph does not appear directly in our construction. Instead, we use the embedding to extract some combinatorial structure from the graph. The conversion from a clover graph to a schematic for the orthogonally convex construction is shown in Figure 25. Lemma 22 describes the properties that we will need.

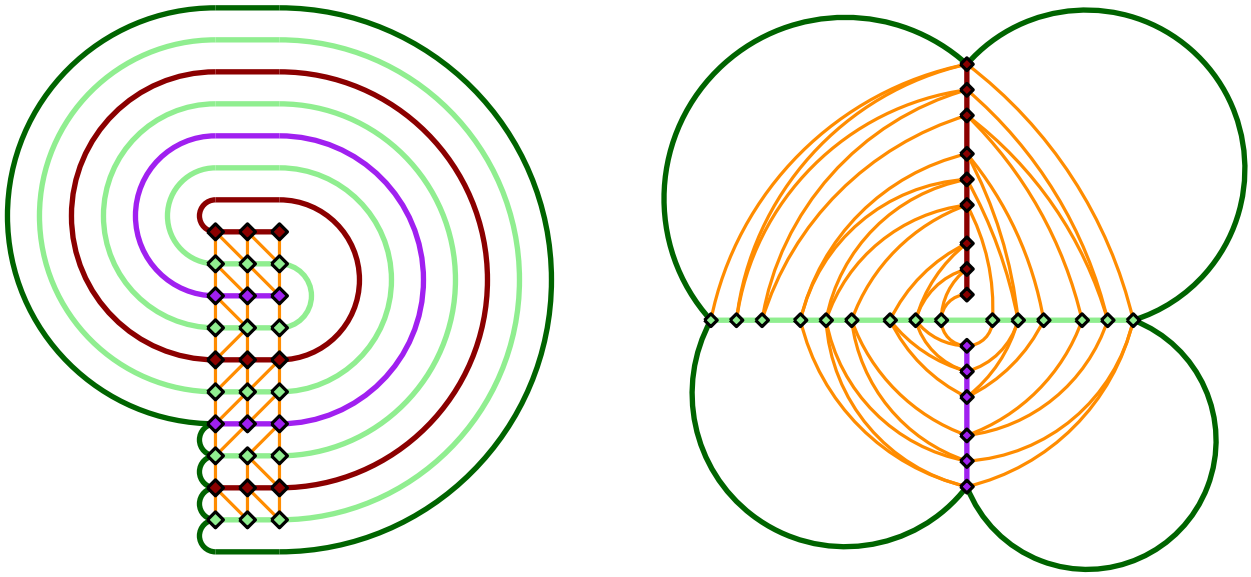

Figure 24: Left: If we order the vertices appropriately, then we can add the extra edges to convert a layered-planar graph into a clover graph. The positive clauses are shown in red and the negative clauses are shown in purple. Right: The same graph can be put into a form like that in Figure 22. The layered planar graph is wrapped around the center in a spiral.

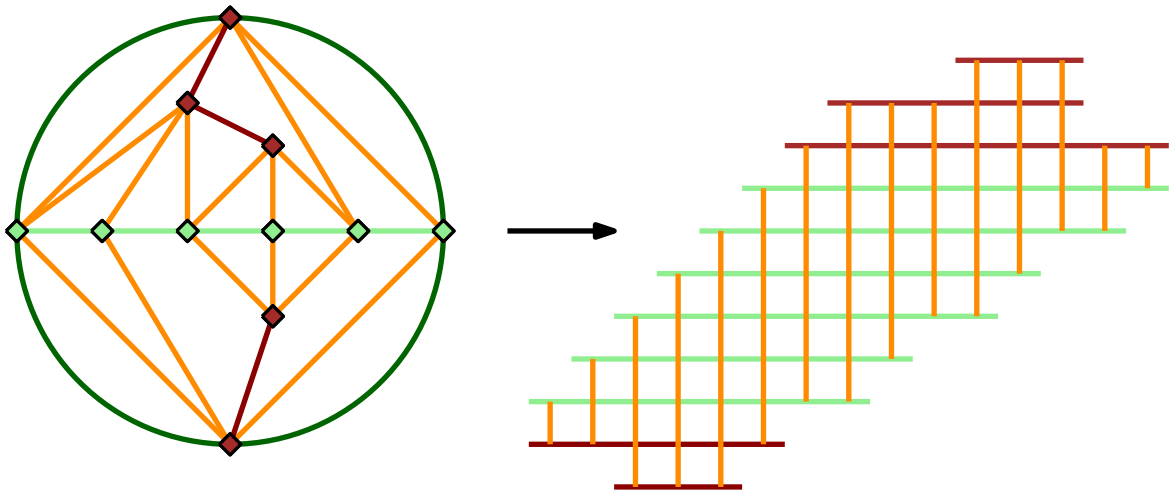

Figure 25: Converting a clover graph to a schematic. Note that the order of the clauses is reversed—the innermost clauses in the graph become the outermost clauses in the schematic.

Lemma 22. *An instance of* Clover-3SAT *can be used to produce a schematic that is orthogonally convex in the following sense: each y-coordinate represents a variable or clause row, with variable rows in the middle and sets of clause rows above and below the variable rows. Each of the upper clause rows covers a strictly smaller range of x-coordinates than the row below it, and each of the lower clause rows covers a strictly smaller set of x-coordinates than the row above it. If a variable row spans the x coordinates $[a, b]$, then the one above it spans x-coordinates $[c, d]$ with $a < c < b < d$.*

If the instance is monotone, then the lower set of clause rows represent negative clauses and the upper set of clause rows represent positive clauses. Furthermore, we have the following property:

Suppose there is a clause $y_i \vee y_j \vee y_k$ and the literal column for y_i passes through the row for another clause c, then the literal columns for y_j and y_k must also pass through the clause row for c (here the y_i are literals that could be of the form x_j or $\neg x_j$).

Proof. Start with an instance of Clover-3SAT with vertices representing variables $x_1, \ldots, x_n$ and clauses $c_1, \ldots, c_p$ and $d_1, \ldots, d_q$. The ith variable row from the bottom represents the variable x_i. The structure of the variable rows is straightforward, but we should check that it is possible to arrange the clause rows appropriately.

In our schematic, the row for c_{i+1} is directly above the row for clause c_i (and similarly d_{i+1} has a row below the row for d_i). We would like to show that it is possible to define, for a clause c_i, an interval $I_i \subseteq \{1, \ldots, n\}$ such that:

- $I_{i+1} \subset I_i$
- If c_i has variables x_j, x_k and x_ℓ, then $j, k, \ell \in I_i$ and each of j, k, ℓ is either not in I_{i+1} or is an endpoint of I_{i+1}

We show this by induction on i. First, we set $I_1 = \{1, \ldots, n\}$. We will also inductively define regions R_i in the plane. Set R_1 to be the region bounded by the path through the variable vertices in G and the edges (x_1, c_1) and (c_1, x_n).

Let $I_i = [a, b]$. Say the variables in c_i are x_j, x_k and x_ℓ with $j \leq k \leq \ell$. The indices i, j and k are in I_i by inductive assumption. In the graph G, c_i is represented by a vertex that has three "legs". These legs and the link from c_{i-1} divide R_i into four subregions (if $i = 1$, then c_i is already on the boundary of R_i, so the 3 legs are enough to produce 4 regions). The clause c_{i+1} must be in one of these regions, along with all the further clauses (see Figure 26). Each of these regions is adjacent to a variable vertices with indices in one of the intervals:

$$[a, j], [j, k], [k, \ell], [\ell, b]$$

So define R_{i+1} to be the region that contains c_{i+1} and define I_{i+1} to be the corresponding interval. The indices i, j, and k are all either outside of I_{i+1} or an endpoint of it. Since the edges corresponding to the literals in c_{i+1} can't cross boundary of the region R_{i+1}, we see that I_{i+1} contains the indices of all the variables that appear in c_{i+1}.

These intervals can then be used to draw the clauses c_i in our schematic. Since the intervals are nested, this schematic is orthogonally convex. We can do the same thing for the d_i (only upside-down).

In a monotone instance, the c_i are positive and the d_i are negative, so all the upper clauses are positive and all the lower clauses are negative.

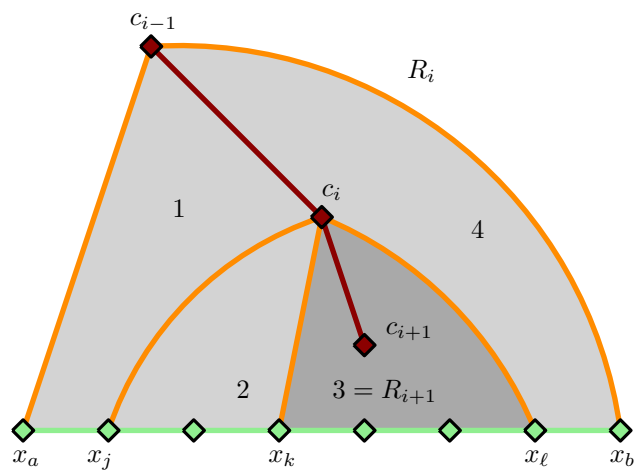

Figure 26: The region R_i is separated into 4 smaller regions, one of which is R_{i+1}.

For the last part, note that all the literal columns for clause c_i pass through the row for clause c_j for each $j < i$ and don't intersect the row for clause c_k when k is $> i$ (similarly for the d_i). Also, a literal column for a clause c_i can never intersect the row for a clause d_k (and visa-versa). □

In analogy with the common definition of Monotone-Planar-3SAT, we have so far put the positive clauses above the variables and the negative clauses below the variables. This is of course arbitrary. There are several choices like this involved in our construction, and it seems to us that it is not possible to simultaneously take the most natural choices for all of them. For this reason, we will actually use rotated schematics where the negative clauses are above the variables and the negative clauses are below.

4.3 Stacks, reference centers, and crossings

In the reduction of Section 2, the reference centers formed a regular grid. In the orthogonally convex case, a few rows will have reference centers of form $[2k, 2k+1] \times [2\ell - 1, 2\ell]$ instead of $[2k-1, 2k] \times [2\ell - 1, 2\ell]$. The following lemma generalizes Lemma 7.

Lemma 23. *Suppose that P is grid polygon, $I \subset \mathbb{Z}$ and let the reference centers be squares in P of form $[2k-1, 2k] \times [2\ell - 1, 2\ell]$ for $\ell \in I$ and $[2k, 2k+1] \times [2\ell - 1, 2\ell]$ for $\ell \notin I$. Then any 2×2 square in P that has integer coordinates contains exactly one reference center.*

Proof. A 2×2 square with integer coordinates has a span of y-coordinates that overlaps with one row containing reference centers. As the square has width 2, it contains a single reference center in that row. □

The color scheme used in Section 2 is not so useful for the orthogonally convex gadgets. The figures in this section will use a looser color-coding of the reference centers based on the function that each square serves in the construction.

Next, we introduce the concept of a *stack*. This is a generalization of the pyramids from Section 2.

A *vertical stack* is a maximal connected set of squares that share their vertical alignment. A *horizontal stack* is a maximal connected set of squares sharing horizontal alignment. We say that

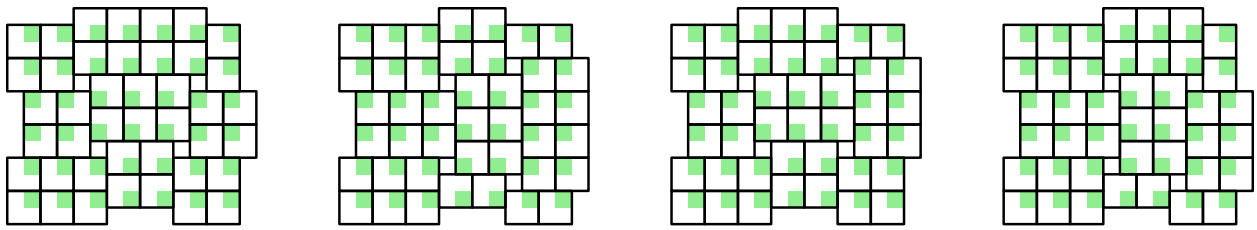

Figure 27: A vertical stack crossing a horizontal stack. First: The horizontal stack maintains the same width, while the vertical stack increases in width by 2 squares. Second: The vertical stack maintains the same width while the horizontal stack grows by 2 squares. Third: Each stack grows by 1 square. Fourth: A different way for each stack to grow by 1 square.

a vertical stack *pushes up* if all the squares in the stack push up, and *pushes down* otherwise. Similarly, a horizontal stack *pushes right* if all the squares push right, and *pushes left* otherwise.

Most of our reference centers are of the form $[2k-1, 2k] \times [2\ell-1, 2\ell]$, so in general a vertical stack always pushes either up or down and a horizontal stack pushes either right or left. However, each variable component will require exactly one row where the reference centers are of the form $[2k, 2k+1] \times [2\ell-1, 2\ell]$. A horizontal stack containing these squares can't be given a left/right direction in general. We will avoid discussing horizontal stacks that contain these squares.

A *static row* is a row of squares where an edge of the polygon pushes the rightmost square to the left or pushes the leftmost square to the right. All the static rows will have reference centers of the form $[2k-1, 2k] \times [2\ell-1, 2\ell]$. A *left static row* has squares that push left and a *right static row* has squares that push right. If a pair of neighboring static rows have opposite horizontal alignments, we call this a horizontal *static shift*.

In Section 2, we saw that a pyramid (i.e., a vertical stack) always grew when passing between two unaligned rows. The following lemma makes a similar claim.

Lemma 24. *Any vertical stack passing through a static shift grows in width by at least* 1 *square.*

Proof. Without loss of generality, we consider a vertical stack that pushes up. If the stack has width k in the row just below the static shift, then clearly, the stack pushes up $k+1$ squares in the row just above the static shift. □

We also need to understand the situation where a vertical stack crosses a horizontal stack. Figure 27 shows 4 different ways in which this can happen. As expressed by the following lemma, such a crossing always makes the stacks grow in width by at least 2 squares in total.

Lemma 25. *Consider a set of reference centers with upper right corners* $(2x, 2y)$ *for* $x = 0, \ldots, x_0+1$ *and* $y = 0, \ldots, y_0+1$. *Consider a packing of* 2×2 *squares where each of these reference centers is covered by a square, and let* $\langle x, y\rangle$ *denote the square covering the reference center with upper right corner* $(2x, 2y)$. *Suppose that*

- $\langle 1, 0\rangle, \ldots, \langle x_0, 0\rangle$ *push up, and*
- $\langle 0, 1\rangle, \ldots, \langle 0, y_0\rangle$ *push right.*

Then it holds that

1. $\langle 1, y_0+1\rangle, \ldots, \langle x_0, y_0+1\rangle$ *push up,*

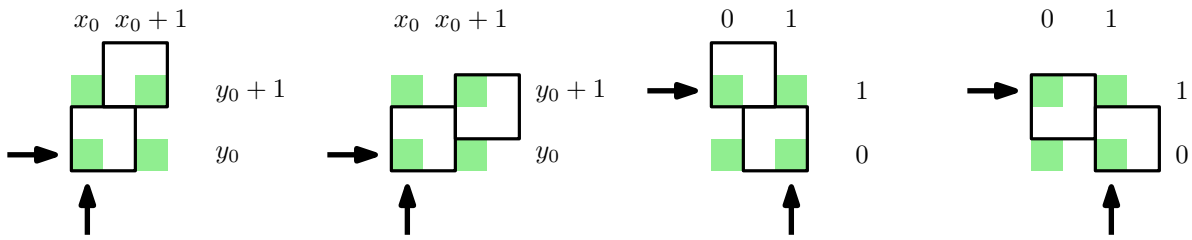

Figure 28: Two left figures: The square $\langle x_0+1, y_0+1\rangle$ pushes up or right. Two right figures: The square $\langle 0,1\rangle$ pushes up or $\langle 1,0\rangle$ pushes right.

2. $\langle x_0+1, 1\rangle, \ldots, \langle x_0+1, y_0\rangle$ *push right,*

3. $\langle x_0+1, y_0+1\rangle$ *pushes up or right, and*

4. $\langle 0, y_0+1\rangle$ *pushes up or* $\langle x_0+1, 0\rangle$ *pushes right.*

In conclusion, when a vertical stack crosses a horizontal stack in a region where the reference centers all have the form $[2k-1, 2k] \times [2\ell-1, 2\ell]$*, then the two stacks together grow in width by at least* 2 *squares.*

Proof. Statements 1 and 2 follow since the reference centers form a regular grid. Figure 28 (left) shows that 3 follows. Figure 28 (right) shows that $\langle 0,1\rangle$ pushes up or $\langle 1,0\rangle$ pushes right. Then statement 4 also follows. □

The concept of a stack was inspired by the work by El-Khechen, Dulieu, Iacono and van Omme [25] showing that these packing problems are in NP even when the coordinates of the polygon can be exponentially large. The methods in [25] seem to suggest that there could be several other types of crossings, but all of these would leave a reference center uncovered. These can be disregarded as long as the reference tiling is a regular grid. We will have to be more careful for stacks that pass through rows with differently-aligned reference centers.

4.4 Verification order and membrane rows

We are given a formula Φ of Monotone-Clover-3SAT and describe how to construct an equivalent instance of 2×2-Square-Packing with an orthogonally convex polygon. This boils down to describing variable and clause components. However, it won't be possible to fully verify the clause components until after the variable components are verified. Verifying the variable components, on the other hand, requires determining a few details about packings of squares in the clause components. The purpose of this section is to explain how the variable components interact with the clause components, allowing us to isolate the verification of the variable component from the verification of the clause component.

The variable components are separated from the clause components by two special static rows that we call the *membrane rows*. There are two such rows, one above and one below the variable components. These separate the variable components from the upper and lower clause components, respectively. All the dependence of the variable components on the clause components is through these rows—the variable components don't (directly) depend on squares in the clause components.

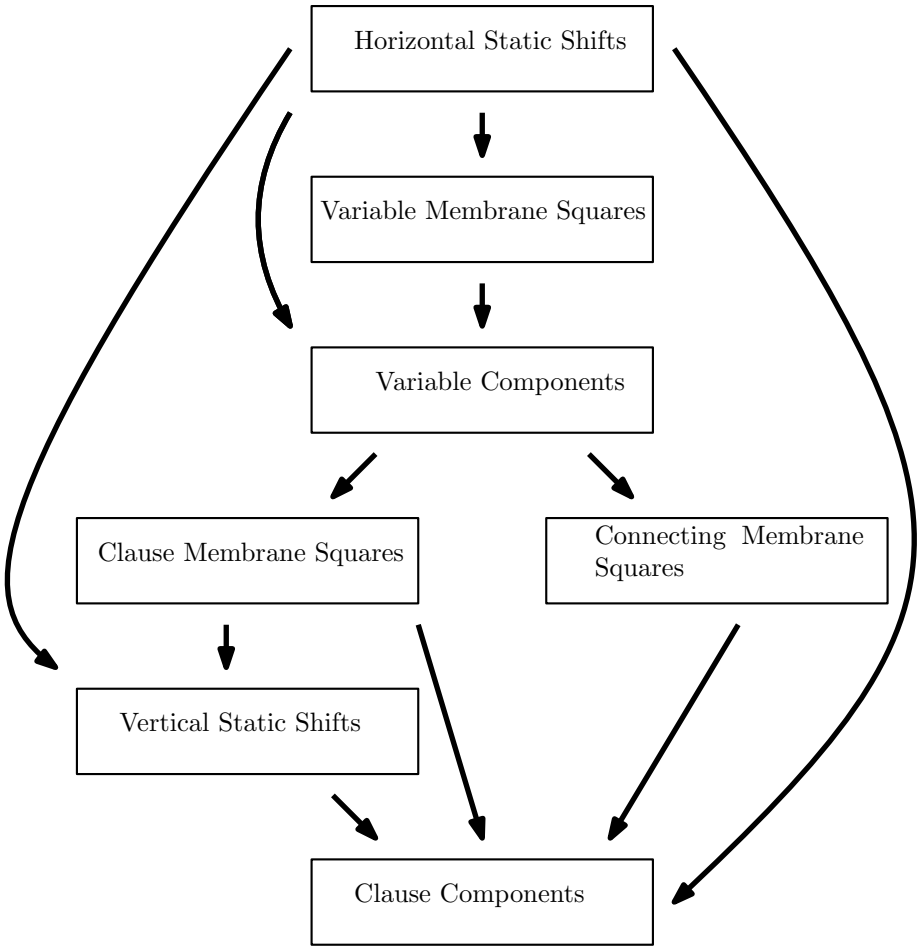

Figure 29: Logical dependence of the different parts of the construction.

Similarly, all the dependence of the clause components on the variable components is through these rows. We say a square in one of the membrane rows *pushes in* if it pushes towards the variable components—that is, it is in the upper membrane row and pushes down or is in the lower membrane row and pushes up. A square in one of the membrane rows *pushes out* if it does the opposite—that is, it pushes *away* from the variable components. We divide the squares in the membrane rows into 4 types depending on the role that they play in the construction. Each square in either of the two membrane rows will be one of these types. The types are as follows:

- *Variable membrane squares* are squares that need to always push in order to verify the variable components.
- *Connecting membrane squares* are squares that may or may not be allowed to push in depending on the position of a variable component.
- *Clause membrane squares* are squares that need to push out in order to verify the clause components.
- *Spacing membrane squares* are all the other squares on the membrane rows. We will describe

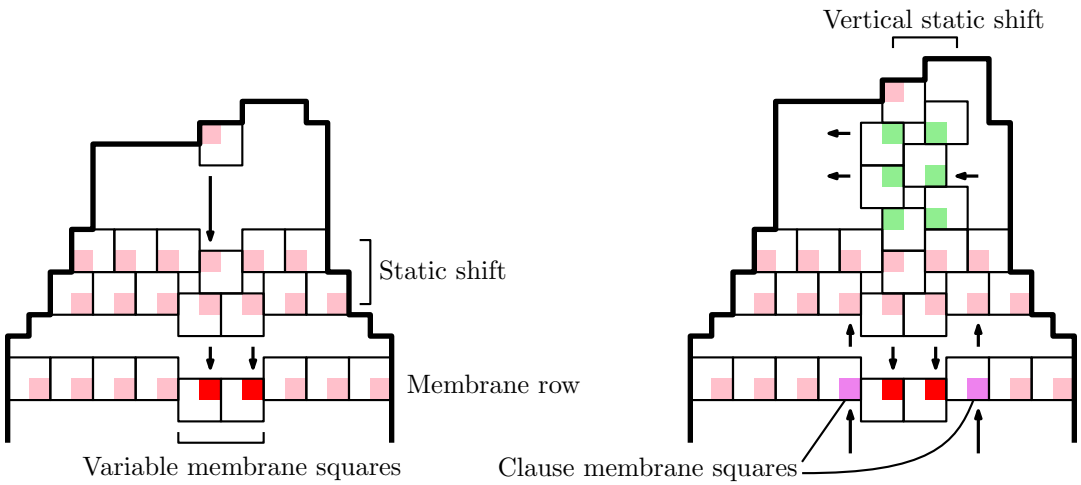

Figure 30: Left: The variable membrane squares always push down. Right: Each block of variable membrane squares has clause membrane squares on either side, creating a vertical static shift on each side.

satisfying assignments that have these pushing out, but the verification doesn't depend on this; there could be some slack in these squares.

The variable membrane squares and connecting membrane squares are organized into *blocks*, each containing some number of adjacent squares. The clause membrane squares are exactly the squares directly adjacent to a block of variable membrane or connecting membrane squares.

We will verify the variable components first, then verify the clause components (see Figure 29). In particular, we will only know that the clause membrane squares push out *after* verifying the variable components. However, we can show that the variable membrane squares push in already. Each block of variable membrane squares is part of a vertical stack created in a clause component above or below the variable components. An edge of the polygon pushes a square in, which creates the stack. By Lemma 24, the vertical stack grows each time it passes through a static shift. By the time the stack reaches the membrane row, it has grown to the full width of the block of variable membrane squares, and so all those squares push in. Figure 30 (left) shows a simplified example of this. In order to be able to make this construction, we just need to make sure that each block of variable membrane squares has a number of squares equal to 1 plus the number of static shifts above it (or below it for a block of variable membrane squares in the lower membrane row).

Lemma 26. *The variable membrane squares push in.*

Proof. Each block of variable membrane squares in the upper membrane row has width equal to 1 plus the number of static shifts above it. Each block of variable membrane squares in the lower membrane row has width equal to 1 plus the number of static shifts below it. At the top or bottom of the polygon, a section of polygon boundary forces one square to push in for each block of variable membrane squares. So by Lemma 24, the variable membrane squares push in. □

Each block of variable membrane squares will have a pair of clause membrane squares on either side. This creates *vertical static shifts* between the stack containing the variable membrane squares and the squares next to it. Figure 30 (right) shows a simplified example of this.

Figure 31: The variable rows for orthogonally convex polygons. The left figure shows what we call the *minus* position and the right figure shows what we call the *plus* position.

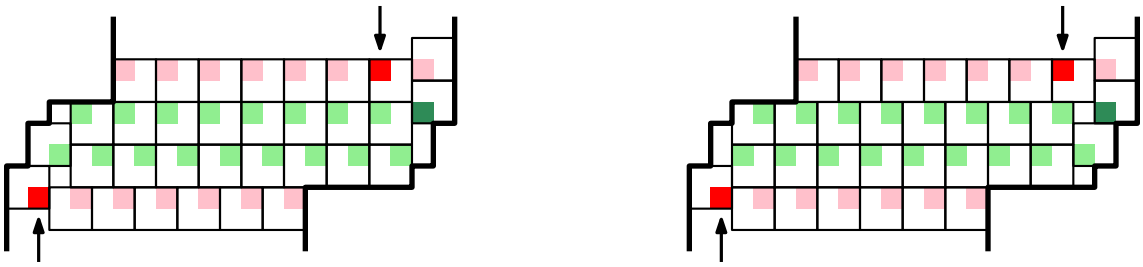

Figure 32: The squares with red reference centers are the *pinch* squares. These push in to form part of the boundary of the variable rows.

Lemma 27. *If the clause membrane squares push out, then any horizontal stack that doesn't contain a static row and passes through a vertical static shift grows in size by* 1 *square.*

Proof. By Lemma 24, the stack containing the variable membrane squares grows at each static shift (moving inwards). By the same lemma, the two adjacent stacks created by the clause membrane squares grow at each static shift when *moving out*. So the squares on opposite sides of a vertical static shift must have opposite vertical alignments, and the result follows by the same proof as Lemma 24. □

We are now ready to describe the variable components.

4.5 Variable components

The purpose of this section is to describe the variable components and show how they can be packed in a satisfying assignment. In Section 4.6, we will verify that any packing of the variable components must correspond to an assignment of variables in the Monotone-Clover-3SAT instance.

The variable components used in Section 2 involved two rows that always have alternate alignment, sandwiched between two static rows that are aligned with each other. Here, we instead use variable components where the two rows always have the same alignment; as shown in Figure 31. In order for this to work, the bottom row of reference centers needs to have a different horizontal alignment than all the other reference centers.

Similarly as in Section 2, we designate the two positions of the variable components by plus and minus, with plus corresponding to squares pushing in a counterclockwise direction around the gadget and minus corresponding to squares pushing in a clockwise direction. The advantage of this type of variable component is that parts of its boundary can be formed by other static squares, allowing the boundary of the polygon to remain orthogonally convex. This is shown in Figure 32. In order for these to be the only packings of the gadget, we need to know that the two marked squares push in. We call these the *pinch squares*.

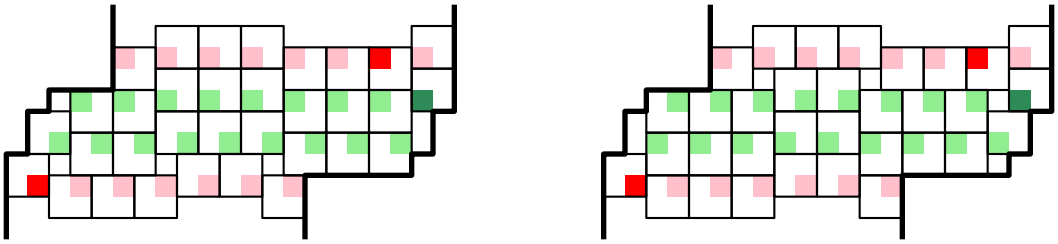

Figure 33: A stack grows once when it passes through the variable component.

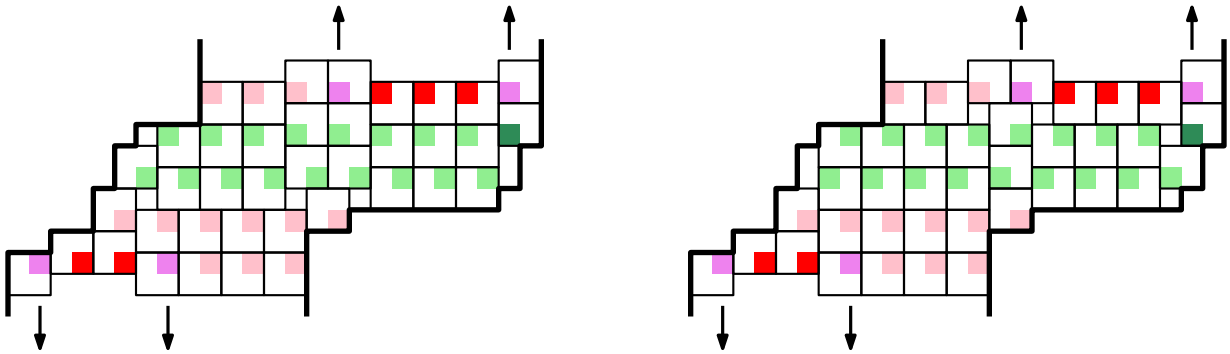

Figure 34: The blocks of pinch squares can be arbitrarily wide, and we can ensure that the squares adjacent to each block of pinch squares push out.

The static rows above and below the variable component have different horizontal alignments, with the upper static rows pushing right and the lower static rows pushing left. As long as the pinch squares push in, the variable component will always be aligned with either the top or bottom row of static squares, so a stack passing through it always grows once, see Figure 33.

The width of a block of pinch squares depends on the number of static shifts and variable components above it, so we should make sure that these blocks can be exactly as large as they need to be. In order to make sure the clause membrane squares push out, we should also make sure that the squares on either side of each block of pinch squares push out. Figure 34 shows how these can be accomplished.

4.5.1 Helper rows

A few rows immediately above and below the variable rows in a variable component are *helper rows*, and these will be necessary for our PUSH gadgets to work. The PUSH gadgets will be described later in Section 4.5.2. The *upper helper rows* are immediately above the variable rows, and the *lower variable rows* are immediately below the variable rows. A helper row is a row that is sometimes forced to split, with squares on the left of the split pushing left and squares on the right of the split pushing right. The squares in the helper rows are shown with light blue and dark green reference centers in our diagrams. The helper rows will satisfy the follow:

- If the variable component is plus, then the squares in the upper helper rows push away from the helper row gadgets. That is, the squares in the upper helper rows that are to the right of the helper row gadgets (dark green reference centers) push right, and the squares in upper

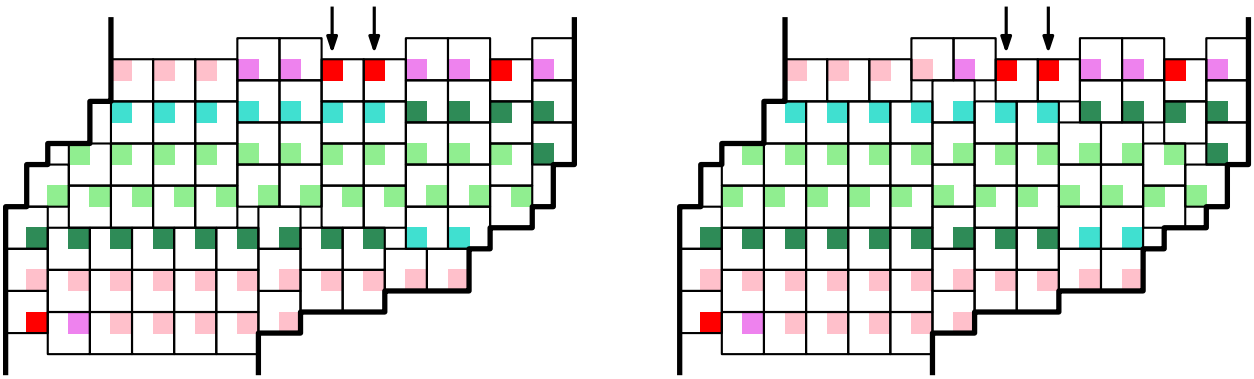

Figure 35: The marked squares are the *helper row creation squares*. The rows directly above and below the variable rows are the *helper rows*. We may need a large number of helper rows, but there will always be the same number of upper helper rows as lower helper rows.

helper rows that are to the left of the helper row gadgets (light blue reference centers) push left.

- If the variable component is minus, then the squares in the lower helper rows push away from the helper row gadgets. That is, the squares in the lower helper rows that are to the right of the helper row gadgets (light blue reference centers) push right, and the squares in upper lower rows that are to the left of the helper row gadgets (dark green reference centers) push left.

When the helper rows don't split, they can have slack, but we will carefully design the variable components so that this slack can't propagate outside the helper rows. Figure 35 shows how the helper rows are created. Each helper row gadget creates both an upper and a lower helper row, so the number of upper helper rows is the same as the number of lower helper rows.

In order for the helper rows to be created, the squares marked in Figure 35 need to push down. We call these the *helper row creation squares*. The blocks of helper row creation squares can be any size, so we can always create the helper row gadget no matter how many static shifts and variable components are above.

Since the polygon should be orthogonally convex, each helper row gadget requires adding an extra static row below it.

4.5.2 PUSH gadgets

The variable components should control the blocks of connecting membrane squares in the membrane row. Each block of connecting membrane squares forms part of a stack that terminates at a *PUSH gadget* in one of the variable components. Some of the blocks of connecting membrane squares should depend on the variable components, but others should always be allowed to push down. So we should create 4 types of PUSH gadget—these are the PUSH-UP-IF-PLUS, PUSH-UP-NEVER, PUSH-DOWN-IF-MINUS, and PUSH-DOWN-NEVER gadgets.

The PUSH-UP-IF-PLUS gadget is shown in Figure 36. If the variable component is plus, then it isn't possible for *all* the marked squares to push down. This will prevent all the squares in some block of connecting membrane squares in the upper membrane row from pushing down.

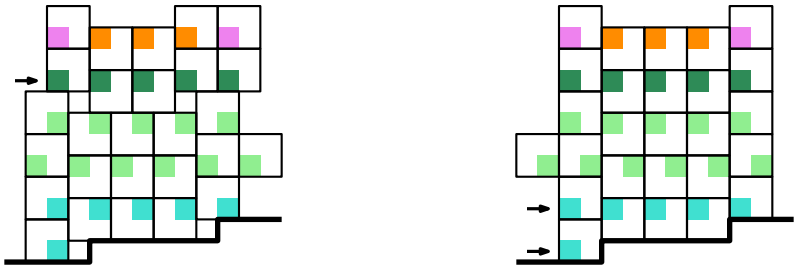

Figure 36: The PUSH-UP-IF-PLUS gadget.

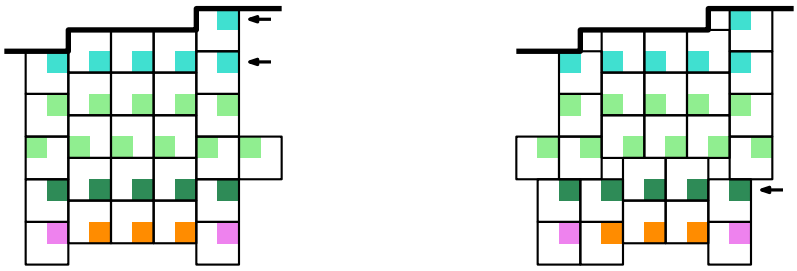

Figure 37: The PUSH-DOWN-IF-MINUS gadget.

Since the polygon is orthogonally convex, we need to add an extra row below the gadget for spacing. The part of the PUSH gadget that interacts with the polygon boundary should carry some information about the position of the variable component, so it isn't possible to use a static row for this purpose. This is why we need the helper rows. There will be helper rows on either side of the variable component, with the lower helper rows "helping" to space out the PUSH-UP gadgets and the upper helper rows helping with the PUSH-DOWN gadgets. The squares in the helper rows are shown with light blue and dark green reference centers in our figures.

The PUSH-DOWN-IF-MINUS gadget is just a mirror image of the PUSH-UP-IF-PLUS, and is shown in Figure 37. If the variable component is minus, then the connecting membrane squares can't all push up.

Some of the blocks of connecting membrane squares not supposed to depend on variable components (see Section 4.7). The squares in these blocks should all be able to push in for either position of the variable gadget. The clause membrane squares adjacent to this block still need to push out. This is accomplished with a PUSH-NEVER gadget. The PUSH-UP-NEVER gadget is shown in Figure 38. Like the true PUSH gadgets, these gadgets each consume a helper row. Note that a PUSH-NEVER gadget may have true PUSH gadgets on both its right and left. The PUSH-UP-NEVER gadget is almost identical to the PUSH-UP-IF-PLUS gadget, but there is an extra 1×1 square of space that allows the gadget to not push when the variable component is plus.

4.5.3 Redundancy columns

Each variable component relies on some squares from below pushing up and some squares from above pushing down. The main part of the verification of the variable components is done working from top-to-bottom, so a variable component may depend on the variable components above it, but shouldn't depend on the ones below it.

The first step in verifying a single variable component is to ensure that the two variable rows

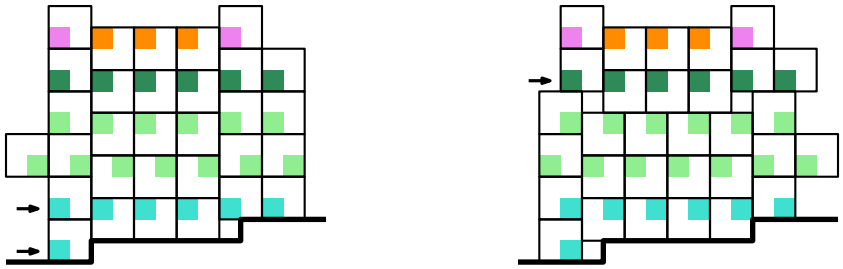

Figure 38: The PUSH-UP-NEVER gadget. The PUSH-DOWN-NEVER gadget is a mirror image.

Figure 39: The variable rows should always be aligned with each other. These are the two cases that need to be ruled out by some gadgets.

share their alignment. This is the only step that requires some squares from below to push up. Observe that there are two "bad" cases where the rows do not share an alignment. These are shown in Figure 39.

These are not symmetric. The case where both push right is excluded by knowledge about the squares packed in parts of the polygon above this variable component. The case where both push left must be ruled out by other means. Here, we use that a vertical stack crossing the unaligned variable rows would grow more in width than if the rows were aligned. We utilize this insight by introducing vertical stacks called *redundancy columns*. This is shown in Figures 40 and 41.

The redundancy columns need to pass through rows that haven't been verified yet, so we cannot be sure about its width as it reaches this variable component. We *can* have a large number of redundancy columns. As long as one of these columns has full width by the time it hits the variable component, it will enforce the necessary constraint. In Section 4.6, we will show that if the total number of redundancy columns is larger than the number of helper rows and variables rows below the variable component, then at least one of the redundancy columns needs to reach full width.

Figure 42 shows how to pack the polygon when there are multiple redundancy columns and some helper rows. Each redundancy column requires adding an extra static row above the upper

Figure 40: The dark purple squares are *redundancy squares*, which create a *redundancy column*. The redundancy column prevents both variable rows from pushing left.

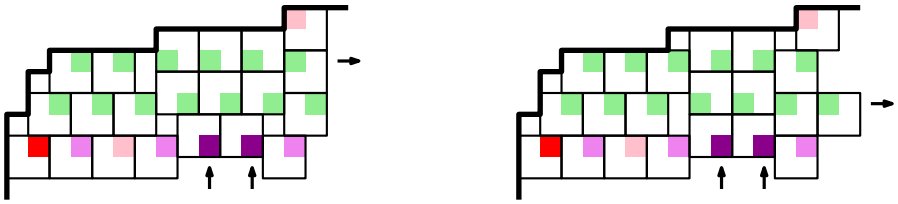

Figure 41: Even if the left pinch squares fail to push up, at least one of the variable rows still has to push right because of the redundancy column.

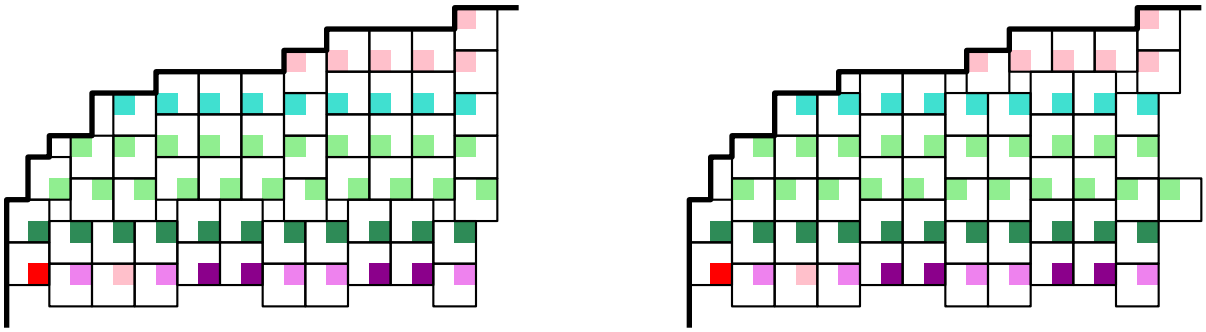

Figure 42: Multiple redundancy columns.

helper rows for spacing reasons.

The redundancy columns only help to verify the part of the variable component that is to the right of the redundancy columns. The PUSH-DOWN gadgets will be to the *left* of the redundancy columns, so we do need to eventually verify that the pinch square on the left pushes up. The left pinch squares are verified *after* the right parts of *all* the variable components have been verified.

In summary, the strategy for verifying the variable components is to first verify the parts of each gadget that are *right* of the redundancy columns working from top-to-bottom, and then to verify the remaining parts of each variable component working from bottom-to-top.

4.5.4 The complete variable component

Figure 43 shows a schematic of a full variable component. Each variable component has two variable rows and some number of helper rows above and below the variable rows. These are flanked by static rows. Because the polygon needs to be orthogonally convex, we need several static rows on each side for spacing reasons. The upper static rows are created in order to make redundancy columns, and the lower are created in order to make helper rows. The lower static rows push left and the upper static rows push right. There is a static shift between the lower static rows for this variable component (which push left) and the upper static rows for the next variable component (which push right).

There are three types of vertical stacks that are used to verify the variable components. These are pinch columns, helper row creators, and redundancy columns. The upper pinch column is always near the right edge of the gadget and the lower pinch column is at the left edge. We place the redundancy columns as far right as possible and place the helper row creators as far left as possible. So all the PUSH-UP columns are between the helper row creators and the upper pinch

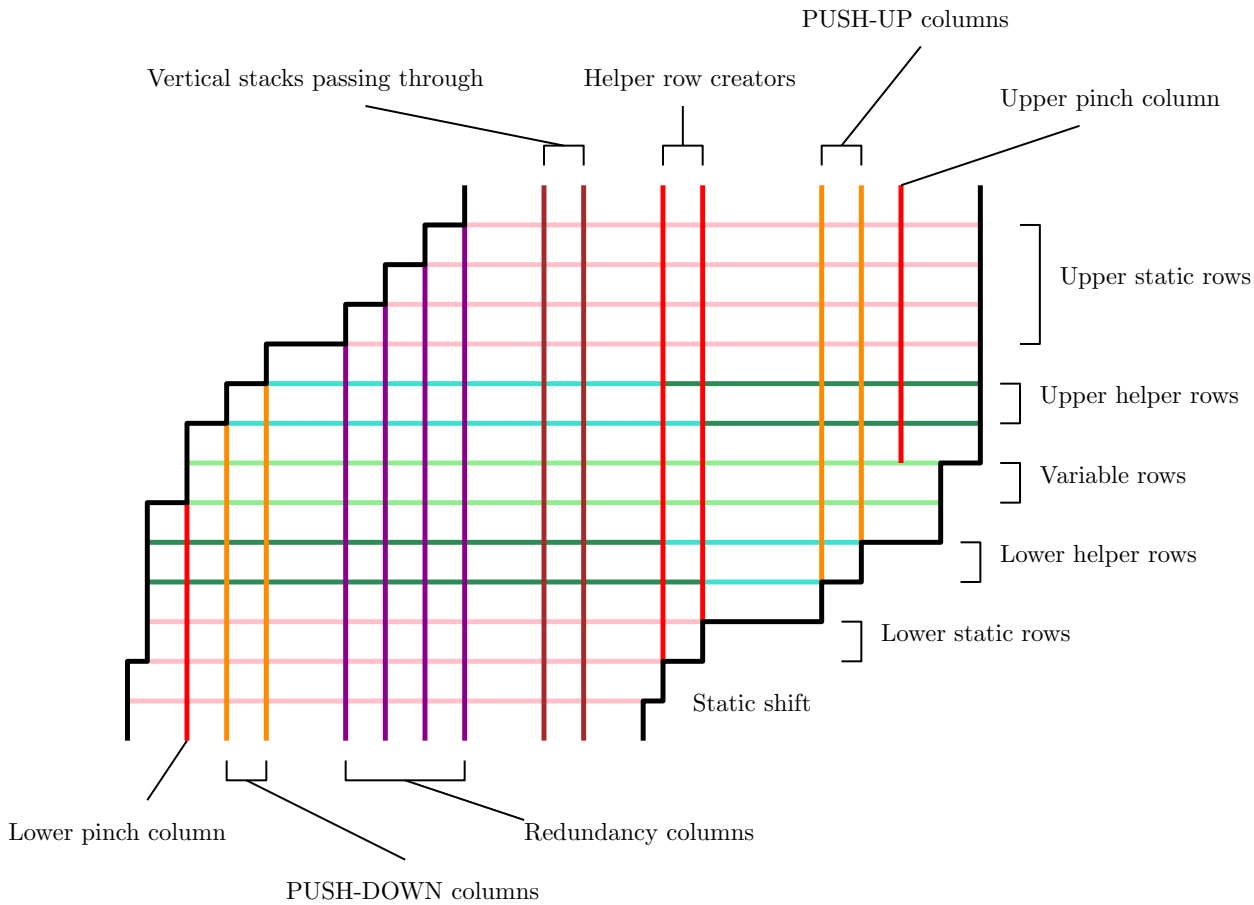

Figure 43: A schematic of a variable component.

column, while the PUSH-DOWN columns are between the lower pinch column and the redundancy columns.

Figure 44 shows a complete variable component with helper rows, a PUSH gadget on each side, and one redundancy column. It is not hard to check from the construction that there exists a packing corresponding to any assignment of variables in the formula Φ, as described by the following lemma.

Lemma 28. *For any assignment of variables, there is a packing of the variable components where the variable membrane squares push in, and:*

- *Each square in a block of connecting membrane squares in the upper (resp. lower) membrane row that is connected to a variable in the minus (resp. plus) position pushes in.*
- *Each square in a block of connecting membrane squares connected to a PUSH-NEVER gadget pushes in.*

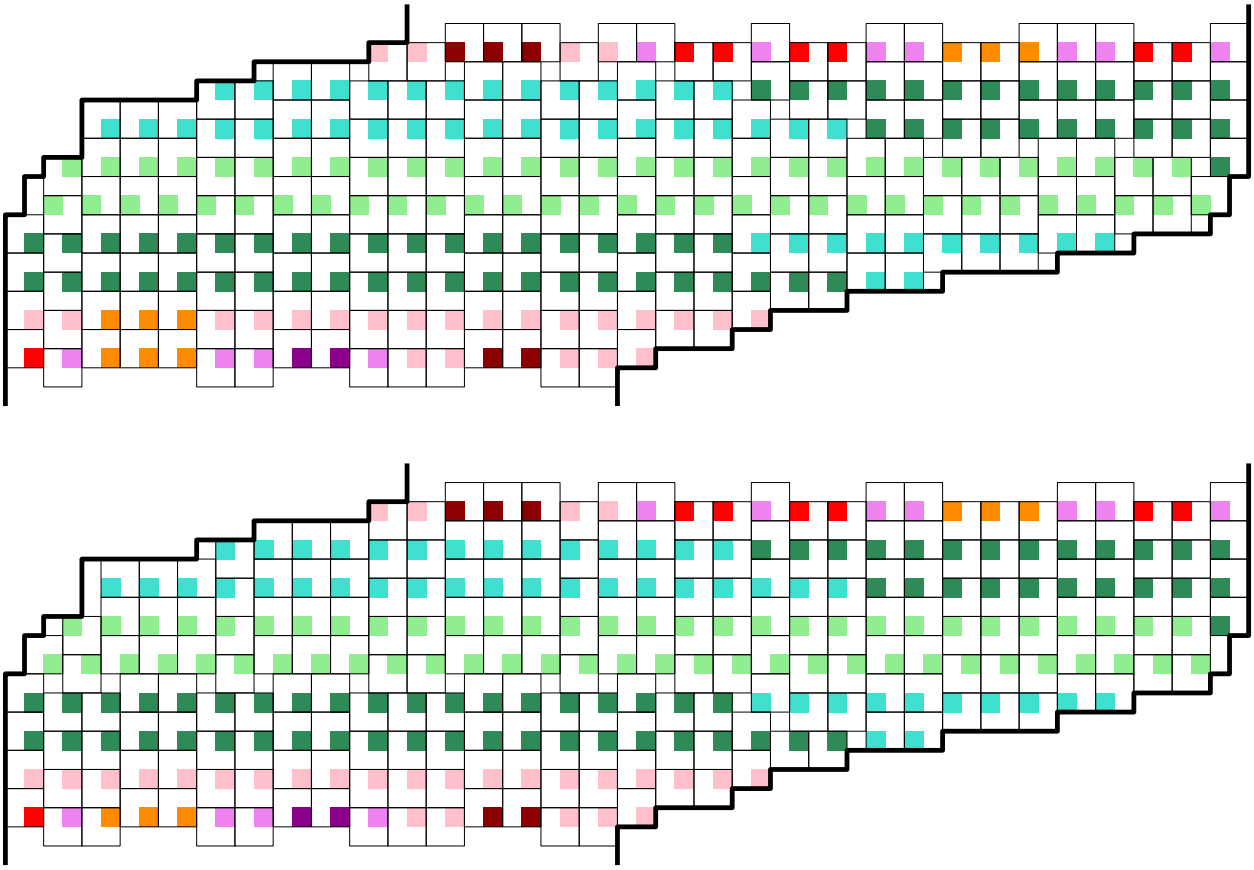

Figure 44: A variable component in the plus (top) and minus (bottom) position. There are two pairs of helper rows, one PUSH-DOWN-IF-MINUS gadget, one PUSH-UP-IF-PLUS gadget, one redundancy column, and one vertical stack passing through.

4.6 Verification of the variable components

We now give the verification of the various properties of a variable component. Throughout this section, the square $\langle x, y\rangle$ refers to the the square containing the reference center with upper right corner $(2x, 2y)$. We choose coordinates so that the squares in the upper variable row are written $\langle x, 0\rangle$ for an integer x. Squares in the lower variable row then have the form $\langle x + \frac{1}{2}, -1\rangle$. We also suppose that there are n helper rows, so the static rows on either side of the variable row are written $\langle x, n + 1\rangle$ and $\langle x, -n - 2\rangle$.

In order to verify a variable component, we need to be able to show that at least one of the redundancy columns reaches full width. We use the following observation:

Lemma 29. *Consider two rows of squares with k rows between them. Let L be the set of squares in the lower row that push up, and let x_s be the x-coordinate of the left corners of each $s \in L$. Let U be the squares of the top row where the x-coordinate of the left corners is $x_s - 1$, x_s or $x_s + 1$ for some $s \in L$. Then at most k squares in U push down.*

Proof. For each row of squares, there are three cases:

- All the squares push left.

Figure 45: There is 1 slack row between two static rows, so at least one of the vertical stacks needs to grow by 1 square.

- All the squares push right.
- Some of the squares on the left push left and the rest push right. There is a *split* separating the two sets of squares.

Each split has width 1, and the squares (and hence also the splits) have integer coordinates, so each split is directly above at most one of the squares in the bottom row.

Consider the square $s \in L$, covering the range of x-coordinates $[x_s, x_s + 2]$. If there is no split with the range $[x_s, x_s + 1]$, then all squares covering that range push up, and similarly if there is no split in $[x_s + 1, x_s + 2]$. There can be a split in the range $[x_s, x_s + 2]$ for at most k squares $s \in L$, one for each of the intermediate layers. Hence, all but at most k squares in U must push up. □

Corollary 30. *Suppose there is a left static row of squares of the form $\langle x, 0\rangle$ and $k+1$ rows above it there is a right static row of squares of the form $\langle x, k+1\rangle$. Suppose that there are blocks of squares $\langle a_i, 0\rangle$ through $\langle b_i, 0\rangle$ for $i \in \{1, \ldots, j\}$ that all push up. Then for all but k indices, all of the squares $\langle a_i - 1, k+1\rangle$ through $\langle b_i, k+1\rangle$ push up.*

This is illustrated in Figure 45. Note that Lemma 24 is a special case of Corollary 30 when $k = 0$.

Lemma 31. *Suppose that a variable component (which has possibly not yet been verified) has m helper rows in total. Suppose there are k stacks below the variable component pushing up. Then at least $k - m - 2$ of those stacks grow in size by 2 squares by the time they reach the next variable component above this one.*

Proof. There is a left static row below the lowest helper row and a right static row above the highest helper row, with $m + 2$ rows in between them. There is a static shift between this variable component and the next one above. So the result follows by Corollary 30 and Lemma 24. □

We are now ready to begin the verification of a single variable component. We first show that the variable rows share their alignment to the right of the rightmost redundancy column, as expressed by the following lemma.

Lemma 32. *Suppose that the rightmost square in the upper block of pinch squares for a variable component pushes down. Then the squares in the variable rows that are to the right of the rightmost redundancy column share their horizontal alignment. Precisely, each square in the upper row spans the same x-coordinates as a square in the lower row and each square in the lower row spanes the same x-coordinates as a square in the upper row (except for the rightmost squares at the end of the variable gadget).*

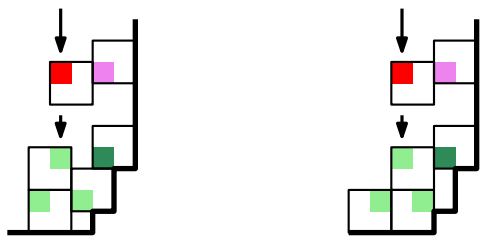

Figure 46: If rightmost of the upper pinch squares pushes down, then one of the rows in the variable component pushes left.

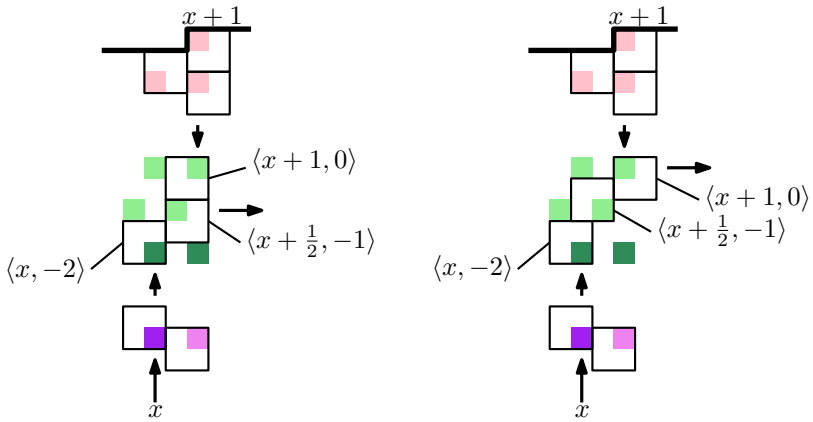

Figure 47: If the rightmost square in any block of redundancy squares pushes up, then at least one of the rows in the variable component pushes right (for squares to the right of that square).

Proof. Together with the pinch squares, the edges of the polygon at the right end of the variable row force the squares in one of the rows to push left, as shown in Figure 46.

Recall that the number of redundancy columns is larger than the number of variable rows and helper rows in all the gadget below it. By Lemma 31, the number of redundancy columns that have full width decreases by $m+2$ when crossing a variable component with m helper rows. So at least one of the redundancy columns must grow by its full size (growing twice at each of the lower variable components, once at the static shift and once when crossing the variable rows).

Let $\langle x, -2-n\rangle$ be the rightmost redundancy square in a redundancy column that reaches full width. So $\langle x, -2-n\rangle$ pushes up, and therefore $\langle x, -2\rangle$ pushes up. The boundary of the polygon forces squares in column $x+1$ push down, so square $\langle x+1, 0\rangle$ pushes down. So one of $\langle x+1, 0\rangle$ or $\langle x+\frac{1}{2}, -1\rangle$ pushes left, as shown in Figure 47.

The conclusion is that, for squares to the right of the rightmost redundancy column, one of the variable rows pushes left and the other pushes right. Since these rows have differently-aligned reference centers, the two rows are aligned with each other. □

When the reference centers form a regular grid and a square pushes up, all the squares in the column above that square must also push up. This isn't *a priori* true for a column passing through a variable component since the reference centers don't form a regular grid. Figure 33 shows that a similar conclusion *does* still hold as long as the variable rows are aligned. We formulate this observation as the following lemma:

Lemma 33. *Suppose that a square $\langle x, 0\rangle$ in the upper row of a variable component is aligned with a square (either $\langle x + \frac{1}{2}, -1\rangle$ or $\langle x - \frac{1}{2}, -1\rangle$) in the lower variable row. If the square $\langle x, 0\rangle$ pushes down, then the square $\langle x, -2\rangle$ also pushes down. If the $\langle x, -2\rangle$ pushes up, then the square $\langle x, 0\rangle$ also pushes up.*

Proof. Straightforward. □

When the upper pinch squares push down, Lemma 32 lets us define the plus/minus position of the two variable rows as the position matched by the squares to the right of the rightmost redundancy column. The following lemma expresses what we need to know about the alignments of the helper rows.

Lemma 34. *Suppose that the upper pinch squares push down and all the helper row creation squares for the variable component push down. The if the variable component is in the plus (resp. minus) position, then all the squares in the upper (resp. lower) helper rows push left if they are left of the helper row squares and push right if they are right of the helper row squares.*

Proof. We show the claim about the upper helper rows when the squares in the variable rows are plus. The claim about the lower helper rows for the minus position follows by a similar argument.

Suppose that the variable rows are plus. We first show the claim that the squares in the helper rows to the left of the leftmost set of helper row creation squares push left. We proceed by induction, showing that the kth upper helper row that are to the left of the kth helper row gadget from the right must push left. The base case is $k = 0$—we think of the "zeroth" helper row as being the top row of the variable component.

The inductive step is shown in Figure 48 (left). Let $\langle x_k, n + 1\rangle$ be the rightmost square in the kth block of helper row creation squares, which pushes down by assumption. So the square $\langle x_k, k\rangle$ pushes down also. By inductive assumption, the square $\langle x_k + 1, k - 1\rangle$ pushes left. This square also pushes up because of the boundary of the polygon (Lemma 33 is used here since this column crosses over the row with differently-aligned reference centers). This means that the square $\langle x_k, k\rangle$ must push to the left, so all the squares in the kth helper row to the left of $\langle x_k, k\rangle$ push to the left. So for x-coordinates to the left of the leftmost of the helper row creation squares, the squares in all the upper helper rows push left.

By induction, we now show that the squares in the $(n + 1 - (k + 1))$th helper row that are to the right of the $(n + 1 - k)$th helper row gadget from the right push to the right. We think of the upper static row as being the $(n + 1)$st helper row, so the case for $k = 0$ is clear.

The inductive step is shown in Figure 48 (right). We label the reference centers as before. Now square $\langle x_{n+1-k}, n + 1 - k\rangle$ pushes down and right, and square $\langle x_{n+1-k} + 1, n + 1 - (k + 1)\rangle$ pushes up. So square $\langle x_{n+1-k} + 1, n + 1 - (k + 1)\rangle$ pushes right, and all the squares in the $n + 1 - (k + 1)$ helper row that are to the right of $\langle x_{n+1-k} + 1, n + 1 - (k + 1)\rangle$ push to the right.

We have now determined the horizontal positions of squares in the upper helper rows when the variable component is plus. □

We are now ready to prove a claim made early in this section and shown in Figure 33 that a vertical passing through a variable component grows in width by 1.

Lemma 35. *If all the helper row creation squares and upper pinch squares push down, then any vertical stack passing through the variable component grows in width by* 1.

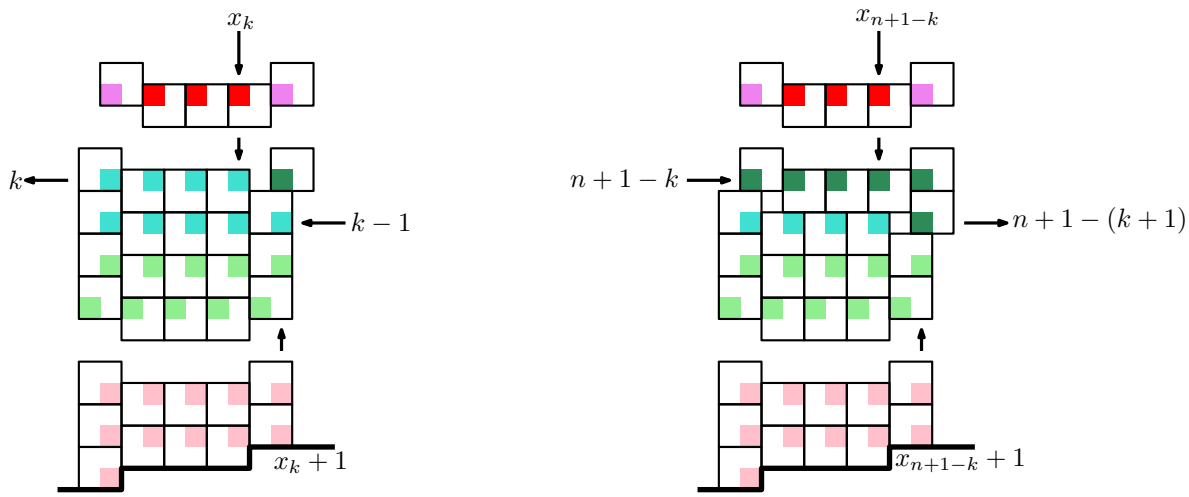

Figure 48: Left: The square $\langle x_k, k\rangle$ pushes left. Right: The square $\langle x_{n+1-k}+1, n+1-(k+1)\rangle$ pushes right.

Proof. Suppose that square $\langle x, -2-n\rangle$ in the below static row pushes up. Any stack that is the left of the redundancy columns will hit the top wall of the polygon instead of passing through the variable component, so we can assume that this square is to the right of the redundancy columns and so the conclusions of Lemmas 32 and 33 can be used. The square $\langle x, n+1\rangle$ pushes up (by Lemma 33), and we want to show that the square $\langle x-1, n+1\rangle$ also pushes up. There are two cases to consider.

If the variable component is plus, then the square $\langle x, n\rangle$ pushes left by Lemma 34. $\langle x, n\rangle$ pushes up and the square $\langle x-1, n+1\rangle$ pushes right, so $\langle x-1, n+1\rangle$ must also push up.

If the variable component is minus, then $\langle x, -2\rangle$ pushes left by Lemma 34. $\langle x, -2\rangle$ also pushes up, and so the squares $\langle x+\frac{1}{2}, -1\rangle$ and $\langle x-\frac{1}{2}, -1\rangle$ must also push up (these squares overlap the x-coordinates of $\langle x, n\rangle$ by Lemma 32). Again by Lemma 32, this means that $\langle x, 0\rangle$ and $\langle x-1, 0\rangle$ push up, so $\langle x-1, n+1\rangle$ pushes up.

So if $\langle x, -2-n\rangle$ pushes up, then $\langle x, n+1\rangle$ and $\langle x-1, n+1\rangle$ push up. Similarly, it can be shown that if $\langle x, n+1\rangle$ pushes down, then $\langle x, -2-n\rangle$ and $\langle x+1, -2-n\rangle$ push down. This gives the required result. □

Corollary 36. *Each vertical stack grows by 2 squares for each variable component that it fully passes through.*

Proof. There is a static shift between every pair of variable components, so this is clear by Lemmas 24 and 35. □

We can now verify that the push squares of the PUSH gadgets work as claimed.

Lemma 37. *If a variable component is plus, then the rightmost square of each of its PUSH-UP gadgets pushes up. If a variable component is minus, then the leftmost square of each of its PUSH-DOWN gadgets pushes down.*

Proof. The PUSH-UP gadgets are right of the helper row gadgets and the PUSH-DOWN gadgets are left of the helper row gadgets. So by Lemma 34, if the variable component is in the up position

then the upper helper row squares in a PUSH-UP gadget push right, which means that the rightmost wire square pushes up (see Figure 36). Similarly, if the variable component is in the down position, then the lower helper row squares in a PUSH-DOWN gadget push left, so the leftmost wire square pushes down (see Figure 37). □

Put together, we can now verify all the properties of the variable components.

Lemma 38. *Suppose that all of the variable membrane squares push in. The our construction has the following properties:*

- *All the clause membrane squares push out.*
- *If all the squares in a block of connecting membrane squares connected to a PUSH-UP-IF-PLUS gadget push down, then that variable component must be minus.*
- *If all the squares in a block of connecting membrane squares connected to a PUSH-DOWN-IF-MINUS gadget push up, then that variable component must be in the push position.*

Proof. The conclusions of Lemmas 32 and 34 require only that squares in the static row above a variable component push down. If the conclusions of these lemmas hold for the first k variable components from the top, then by Corollary 36 the upper pinch squares and helper row creation squares for $(k+1)$st variable component push down. By induction, we conclude that the results of Lemmas 32 and 34 hold for all variable components.

Again by Corollary 36, we can now conclude that all lower pinch squares push up. So each pair of variable rows in a variable component are aligned going all the way to the left edge (this is important because the PUSH-DOWN gadgets are left of redundancy gadgets).

Each set of variable membrane squares connects to a block of pinch, helper row creation, or redundancy squares. The two squares on either side of such a block push out by construction of the gadgets (see our diagrams). So by Corollary 36, the squares adjacent to the block of variable membrane squares also push out. This is shown in Figure 49.

By a similar principal and Corollary 36, the connecting membrane squares behave as required. That is, the clause membrane squares adjacent to the block of connecting membrane squares push out, and it is only possible for all the squares in the block to push in if the variable component is in the appropriate position (see Lemma 37). This is shown in Figure 50.

□

4.7 Clause components

We will need two sets of clause components, one above the variable components and one below. These are both created in the same way, so throughout this section we just consider the top set of components. To create the bottom set of components, just exchange "up" and "down". The clause components above the variable components represent negative clauses in the formula Φ that we reduce from, while the clause components below the variables represent positive clauses.

4.7.1 Tester stacks and SWITCH gadgets

Each set of clause components is formed by a large number of criss-crossing horizontal and vertical stacks. Each stack is limited in how wide it can grow. Figure 51 shows how the start of a stack is

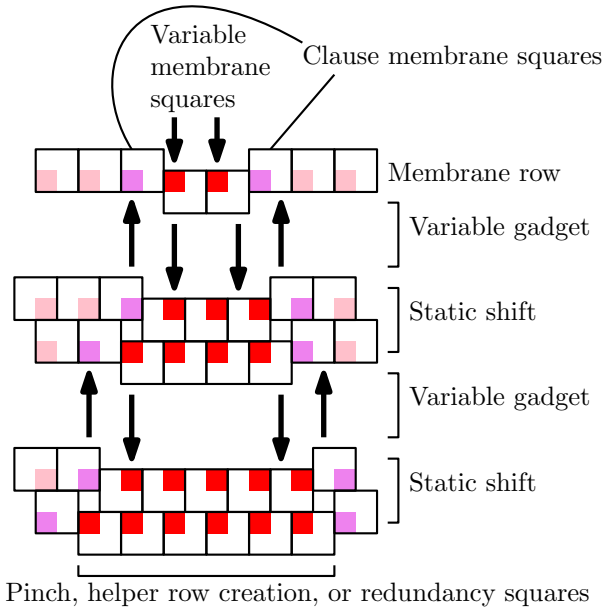

Figure 49: A block of e.g. pinch squares are created by a block of variable squares at the boundary. The squares adjacent to the block of pinch squares push out by construction of the gadget, so by Corollary 36 the squares adjacent to the set of variable membrane squares also push out.

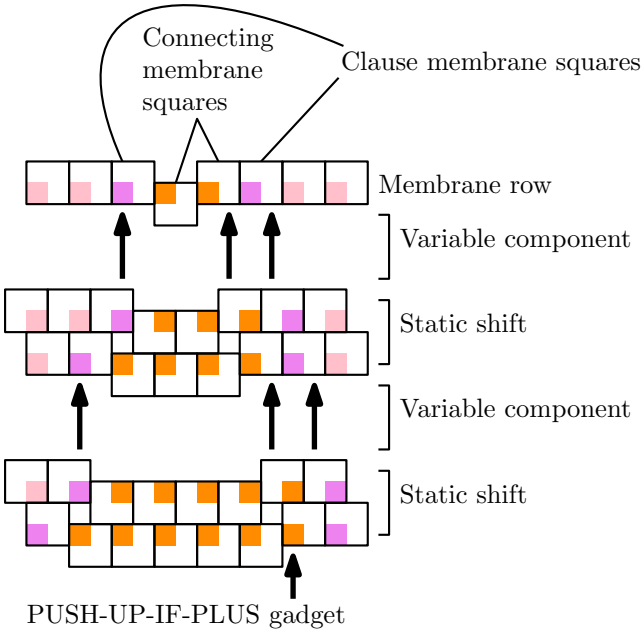

Figure 50: When the push squares are up, the rightmost connecting membrane square is also up, is if all connecting membrane squares are down, the variable component containing the PUSH-UP-IF-PLUS gadget must be minus.

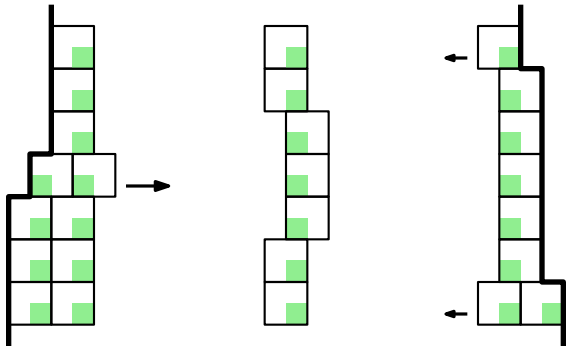

Figure 51: The squares in the middle must form part of a horizontal stack that pushes to the right, due to the edge of the polygon at the left side that pushes a square to the right. The adjacent squares could also be part of this stack. A vertical stack or a stack pushing to the left could be created in a similar way. For horizontal stacks, a section of the polygon boundary on the opposite side prevents the horizontal stack from growing more than a set amount, as shown here.

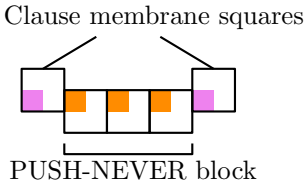

Figure 52: A block of connecting membrane squares (which are attached to a PUSH-NEVER gadget in a variable component below) allow a vertical stack created in a clause component above to grow only by up to a certain amount and no more.

created. On the other side of the polygon, the boundary prevents the stack from growing by more than a certain amount. For a vertical stack, the membrane row prevents the stack from growing more than a certain amount, as shown in Figure 52.

An important ingredient is the *SWITCH gadget*, which must create a vertical stack or a horizontal stack, but generally not both. This is shown in Figure 53. Consider a negative clause $\neg x_i \vee \neg x_j \vee \neg x_k$ of the formula Φ that we reduce from. We make a corresponding clause component in which we have three SWITCH gadgets corresponding to x_i, x_j, x_k, respectively. These define variables y_i, y_j, y_k, and for each of these y_ℓ, we define $y_\ell = 1$ if the SWITCH gadget creates a vertical stack, and otherwise we define $y_\ell = 0$. The clause component will enforce the constraint $y_i + y_j + y_k = 1$, i.e., exactly one SWITCH gadget will make a vertical stack. When $y_\ell = 1$ and the respective gadget creates a vertical stack, this stack hits a block of connecting membrane squares that is connected to a PUSH-UP-IF-PLUS gadget in the corresponding variable component x_ℓ. An example of this is shown in Figure 54. The PUSH-UP-IF-PLUS gadget only allows all these squares to push down if the variable component is minus. This creates a constraint $y_\ell \implies \neg x_\ell$. Since one of the SWITCH gadgets make a vertical stack, we conclude that the clause $\neg x_i \vee \neg x_j \vee \neg x_k$ is satisfied. This is only a one way implication—if y_ℓ is 0 then there is no constraint on x_ℓ, so even though just one SWITCH gadget makes a vertical stack, there can be more than one variable satisfying the clause.

The corresponding construction in the clause components below the variable components result

Figure 53: A *SWITCH gadget* that either creates a vertical stack or a horizontal stack.

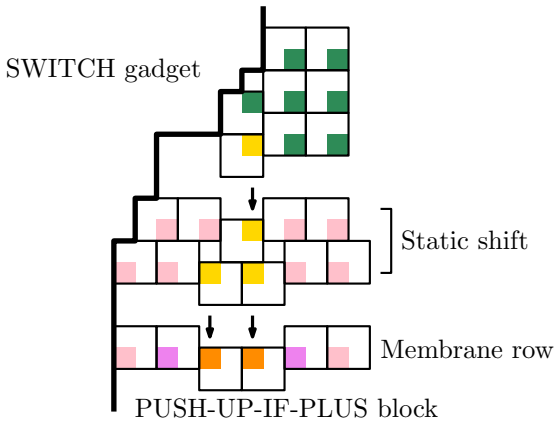

Figure 54: If the SWITCH gadget creates a vertical stack, then all the squares in a block of connecting membrane squares push down. These connect to a PUSH-UP-IF-PLUS gadget, so this can only happen if the variable component is minus.

in the implication $y_i \implies x_k$, since there we connect the SWITCH gadget to a PUSH-DOWN-IF-MINUS gadget. Hence, we can also realize a positive clause $x_i \vee x_j \vee x_k$.

The rows and columns created by the SWITCH gadgets are called *literal rows* and *literal columns*. We say that a literal row is *aligned* if there is no stack present and *unaligned* otherwise. Note that the literal row created by a SWITCH gadget on the left side of the polygon should be unaligned when pushing right, but the literal row created by a SWITCH gadget on the right side of the polygon should be unaligned when pushing left. Whenever we need a SWITCH gadget on the right side of the polygon, we will need a static shift above and below it to change the alignment of the static squares. This can be seen in Figure 60.

There are also some stacks that will always exist. These are called *tester rows* and *tester columns*. Each tester row is constrained to grow at most a fixed number of times, a constraint that is easier to satisfy the fewer vertical stacks there are. The tester columns, on the other hand, create constraints that get easier to satisfy the fewer *horizontal* stacks there are.

4.7.2 Defining the clause components

We are now ready to fully define the clause components. Figure 55 shows the setup at a schematic level. Due to the structure of the schematics produced from an instance of Monotone-Clover-3SAT, all the literal columns and tester columns from the ith clause from the top pass through the literal and testers rows from the kth clause if $k > i$.

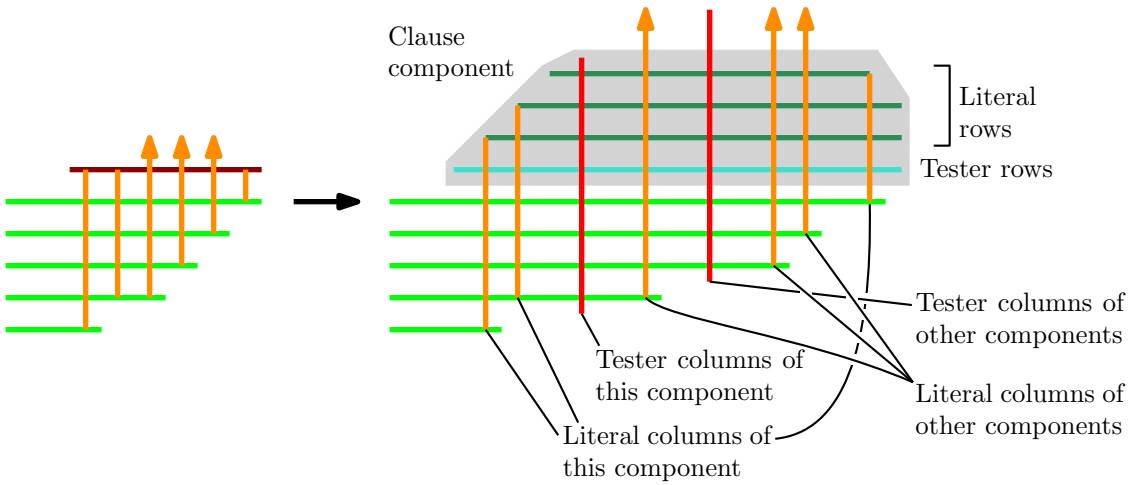

Figure 55: Expanding the schematic described in Lemma 22 to make space for the tester rows and tester columns. Each red or light blue line in this schematic represents multiple tester rows or columns.

Now we just need to specify the number of tester rows and tester columns for each clause and the amount by which each stack in the construction may grow. Here we mostly ignore horizontal static shifts (needed to change the alignment between left and right SWITCH gadgets) and vertical static shifts (needed to push down the variable membrane squares). By Lemmas 24 and 27, each stack grows exactly once when it passes through one of these shifts. When creating the final construction, the full amount that each stack can grow will be increased to account for this. The tester rows and tester columns are defined as follows:

- Starting with $i = 0$, the ith clause component from the top has $3 + i$ tester columns. These start above the literal rows for this clause component and are created by a section of the top wall of the polygon. Let $t_i = 3(i + 1) + \frac{1}{2}i(i + 1)$ be the total number of tester columns in clauses $0, \ldots, i$.
- Each tester column is allowed to grow by at most two squares on each side, so by at most 4 squares in total. Since the column starts with width 1, this means that it should be allowed to reach a size of at most 5 squares (plus the total number of static shifts that is passes through).
- The ith clause component from the top has $1 + 2t_i$ tester rows directly below its literal rows.
- A tester row below the ith clause component is allowed to grow by at most $i + 1 + t_i$ squares on each side, so by $2(i + 1 + t_i)$ squares in total.

An unaligned literal column should always push down on an entire block of connecting membrane squares, even if it only grows at static shifts. Since the clause membrane squares adjacent to the connecting membrane squares push up, this prevents an unaligned literal column from growing any further. So it just remains to specify how much the literal rows can grow. Each clause component has 3 literal rows.

- The top literal row in the ith clause component from the top (again starting at $i = 0$) can grow by at most $i + t_{i-1}$ squares on each side, so by at most $2(i + t_{i-1})$ squares in total.
- The lower two literal rows in the ith clause component can grow by an additional square on each side, so by at most $2(1 + i + t_{i-1})$ in total.

This finishes the description of the clause component and thus of the entire polygon. It remains to show that if Φ is satisfiable, then there is a perfect packing of the clause components and to verify that if there is a perfect packing, then Φ is satisfiable.

4.7.3 Packing the clause components

The following lemma describes how to pack the clause components using a satisfying assignment of Φ.

Lemma 39. *Suppose there is a satisfying assignment of the* Monotone-Clover-3SAT *formula* Φ*. Then there is a packing of the upper clause components where:*

- *The clause membrane squares push up.*
- *Each square in a block of connecting membrane squares connected to a PUSH-UP gadget pushes up unless the corresponding variable is* 0 *in the satisfying assignment.*

Proof. Each literal has a corresponding SWITCH gadget. For each clause, choose a true literal in that clause. The SWITCH gadgets corresponding to those literals are set to create a vertical stack and not a horizontal stack. The remaining SWITCH gadgets each create a horizontal stack and not a vertical stack.

So each clause component has two unaligned literal rows and one unaligned literal column. This determines which rows and columns are aligned or unaligned. Now we just need to specify how the stacks grow at each crossing. This is done by assigning each stack a priority p. When two stacks cross, the stack with a larger value of p grows by 2 squares and the stack with the smaller value of p does not grow. The priorities are assigned as follows:

- The literal columns have priority 0.
- The literal rows in the ith clause component have priority $2i + 1$.
- The tester columns starting above the ith clause component have priority $2i + 2$.
- The tester rows have priority ∞.

That is to say, the literal columns only grow at static shifts. Each tester column grows at the first two unaligned literal rows that it crosses, then doesn't grow any more.

The unaligned literal rows in the ith clause component grow when they cross a tester column for one of the previous clause components, but do not grow at the tester columns for that column. The literal rows also grow at each of the unaligned literal columns. There are t_{i-1} and i unaligned literal columns coming from the clause components above. The lower of the two literal rows also cross some of the literal columns from *this* variable component, so may need to grow $1 + i + t_{i-1}$ times.

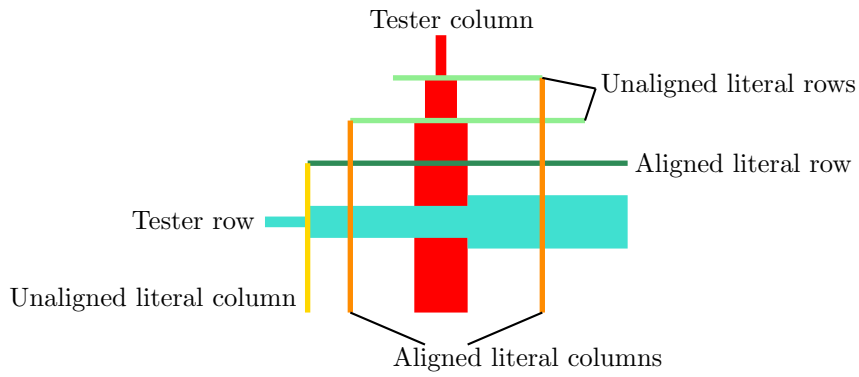

Figure 56: A simplified clause component. The tester column (red) grows twice at the two unaligned literal rows (green). There are two unaligned literal rows (light green), which have corresponding aligned literal columns (orange). There is one aligned literal row (dark green), with a corresponding unaligned literal column (gold). The tester row (light blue) grows at the tester column and at the unaligned literal column.

The tester rows grow at all the stacks that they pass through. Below the ith clause component, there are t_i tester columns and $i+1$ unaligned literal columns (recall that i starts at 0).

Figures 56 to 58 show some schematics of what this looks like.

Recall that a block of connecting membrane squares that is connected to a PUSH-UP gadget below is connected to a literal column above. Recall also that the clause above contain only negative literals. The literal columns that push down all correspond to true literals, which correspond to variables with value 0. So the only connecting membrane squares that push down are connected to variables that are 0 in the satisfying assignment. □

These are *not* the only ways to pack the clause components. Figure 59 shows a schematic of a different packing of the first clause component. That the rightmost tester column has not grown to its maximum width, creating some slack that can propagate down into the clause components below. This extra slack is the reason that the number of testers needs to increase in the subsequent gadgets. This construction may seem unnecessarily complicated, but it is actually quite difficult to limit this slack propagation without needing exponentially many testers.

Figures 60 to 63 show how the first clause component is realized and how to pack it in a satisfying assignment.

4.8 Verification of the clause components

We now formally verify that any packing of the clause components requires one of the literal columns for each clause to push down. To carefully check this, we write some algebraic constraints that must be satisfied. Say there are n clauses $c_i : y_{3i} + y_{3i+1} + y_{3i+2} = 1$, with c_0 being at the top and c_{n-1} being at the bottom.

For the ith SWITCH gadget (starting with $i = 0$), we define a variable y_i that takes values in $\{0, 1\}$. The variable y_i is 1 when the SWITCH gadget creates a vertical stack and 0 otherwise.

We give a name to each stack in the construction that can grow. The row for the literal y_i is ℓ_i. The jth tester column for the clause c_i is called $c_{i,j}$. The jth tester row for the clause c_i is called $r_{i,j}$.

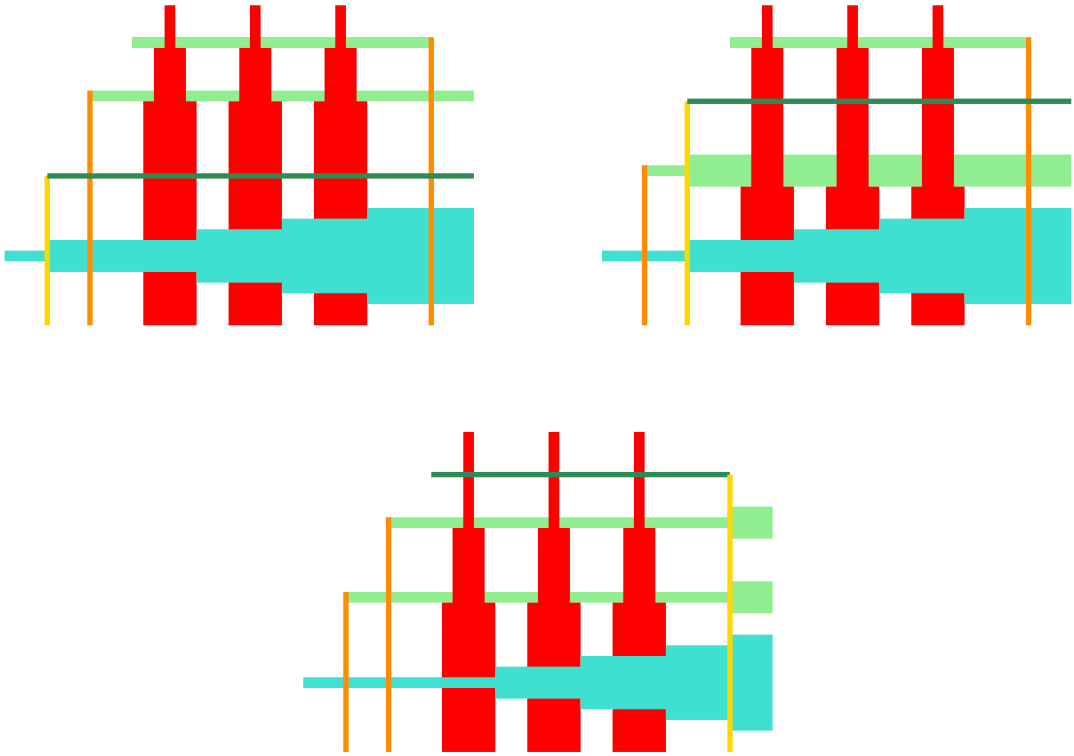

Figure 57: The three ways to pack the first clause component, depending on which literal is true. Here only one tester row is pictured, in the actual construction there would be seven. Note that in the top left figure, the middle literal row doesn't grow to its full width.

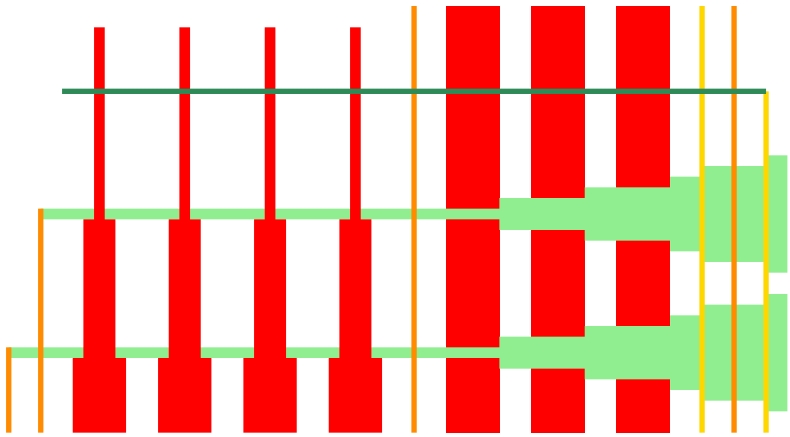

Figure 58: The second clause component. The bottom two literal rows need to be able to grow 5 times, once for the crossing with the third literal column, once for the one true literal from the first clause component, and once for each of the tester columns from the first clause component. The top literal row is only allowed to grow 4 times. The tester rows are omitted. The second clause component has a 4th tester column in case the first one sends some slack down.

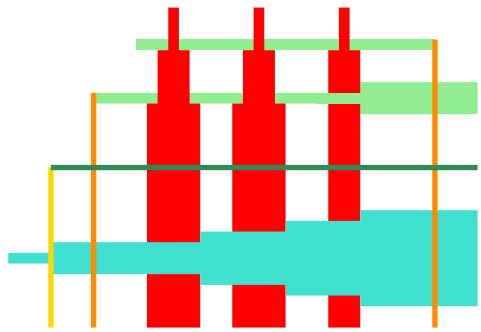

Figure 59: If the lowest literal row is the one that is aligned, then some slack can propagate.

Recall that when two stacks cross each other, combined they grow by a total of at least 2 squares (Lemma 25). For the crossing between a horizontal stack a and a vertical stack b, we designate a variable $x(a,b) \in \{0,1,2\}$ that records how many times a grows when passing through this crossing. The vertical stack b then grows $2 - x(a,b)$ times. The stacks created by unaligned literal columns can't grow, so any row that crosses an unaligned literal column grows by 2 and we don't need an x variable to keep track of this.

The full set of constraints can now be written. For $0 \leq i < n$ and $0 \leq j < 1 + 2t_i$ there is a tester row $r_{i,j}$. This is allowed to grow at each tester column and at one of the literal columns for each of the clauses above it, up to a total width of $2\,(i+1+t_i)$ (recall that $t_i = 3(i+1) + \frac{1}{2}i(i+1)$ is the number of tester columns in clauses $c_0, \ldots, c_i$). The constraint created by a tester row can then be expressed as follows:

$$2\sum_{p=0}^{3i+2} y_p + \sum_{p=0}^{i}\sum_{q=0}^{3+p-1} x(r_{i,j}, c_{p,q}) \leq 2\,(i+1+t_i) \tag{3}$$

For $0 \leq i < n$, the uppermost literal row in the ith clause is ℓ_{3i}. The literal row ℓ_{3i} is allowed to grow once at each of the tester columns for *previous* clauses, and at one of the literal columns for each of the clauses above it, for a total of $2\,(i+t_{i-1})$ times. This leads to the following inequality.

$$(1-y_{3i})\left(2\sum_{p=0}^{3i-1} y_p + \sum_{p=0}^{i}\sum_{q=0}^{3+p-1} x(\ell_{3i}, c_{p,q})\right) \leq 2\,(i+t_{i-1}) \tag{4}$$

The next two literal rows in the ith clause are each allowed to grow by an additional two squares. So for $0 \leq i < n$ and $m \in \{1,2\}$, we have the following:

$$(1-y_{3i+m})\left(2\sum_{p=0}^{3i+m-1} y_p + \sum_{p=0}^{i}\sum_{q=0}^{3+p-1} x(\ell_{3i+m}, c_{p,q})\right) \leq 2\,(1+i+t_{i-1}) \tag{5}$$

Finally, for $0 \leq p < n$ and $0 \leq q < 3+i$, we have a tester column $c_{p,q}$, which is allowed to grow 4 times. Recall that $c_{p,q}$ crosses all literal and tester rows in clauses c_{p+1} through c_{n-1}. This leads to the constraint:

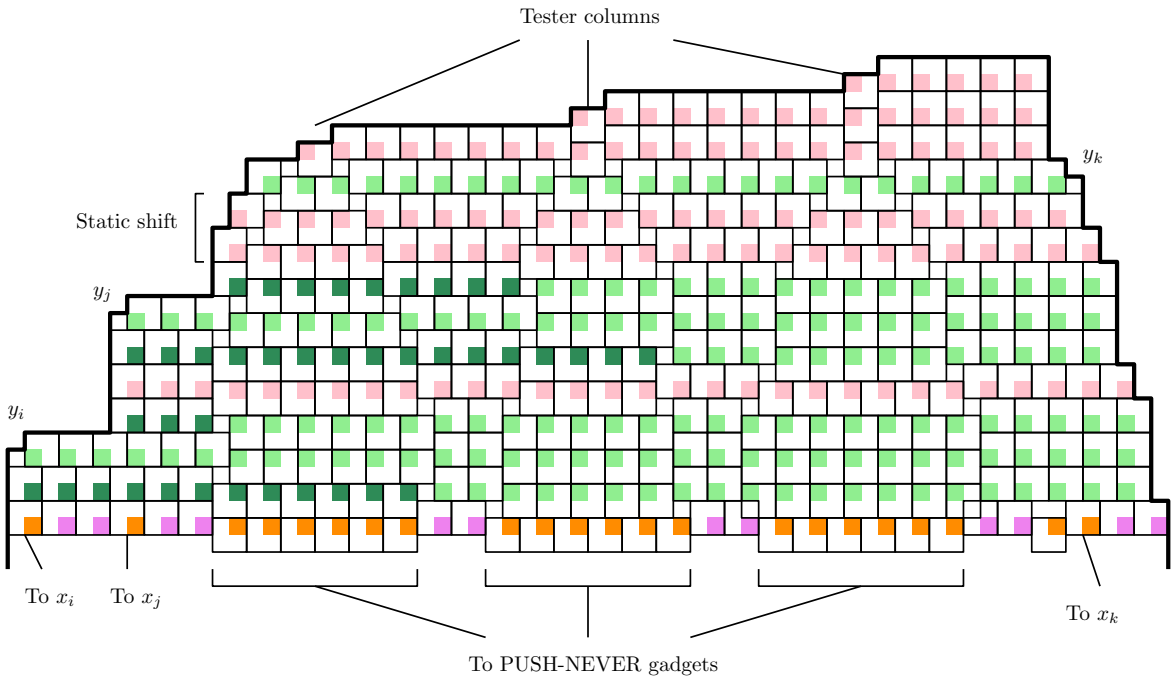

Figure 60: The clause component, showing a state where none of the literals are true and so no perfect packing exists (two squares overlap).

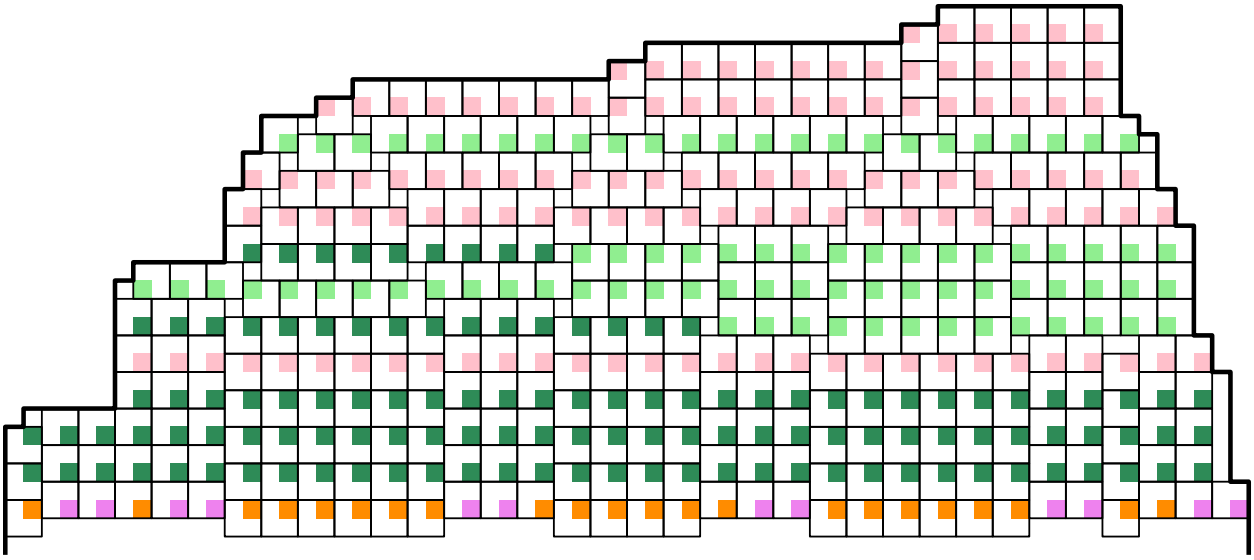

Figure 61: A perfect packing *does* exist if the first literal column is allowed to lower. The packing shown here is *not* quite the same packing described in Lemma 39. Instead we show a case where the middle tester column has not grown to its maximum width in this component, creating some slack that can propagate down.

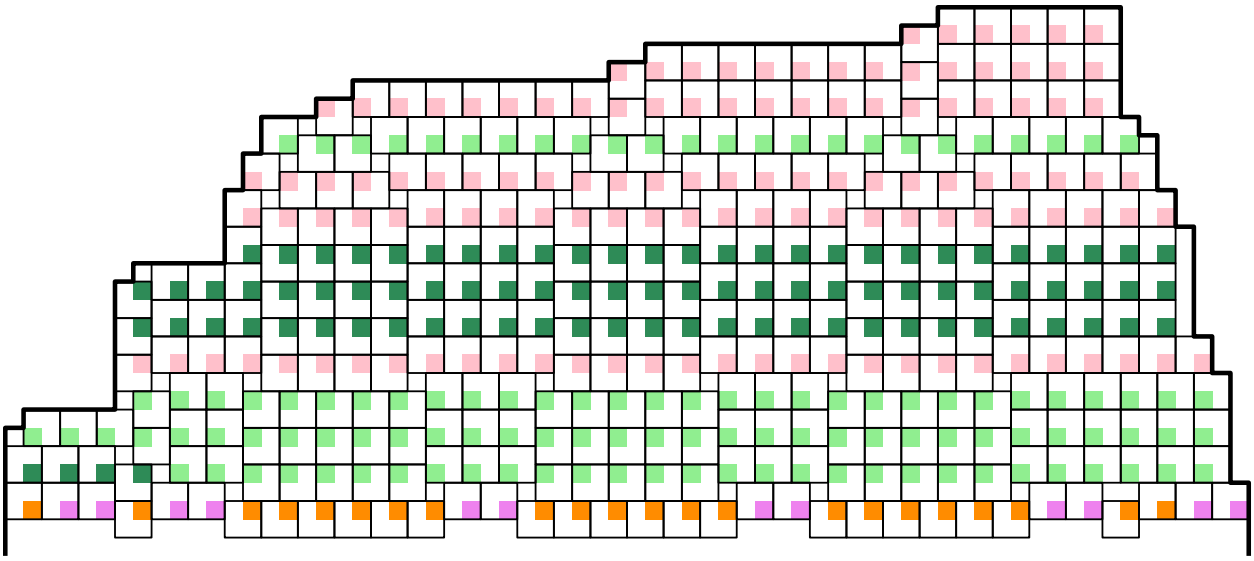

Figure 62: A perfect packing when the second literal column is allowed to lower.

$$\sum_{i=p}^{n-1}\sum_{m=0}^{2}(1-y_{3i+m})\left(2-x\left(\ell_{3i+m},c_{p,q}\right)\right)+\sum_{i=p}^{n-1}\sum_{j=0}^{2t_i}\left(2-x\left(r_{i,j},c_{p,q}\right)\right)\leq 4 \tag{6}$$

Here we are assuming that a literal row is always aligned when the corresponding literal column is unaligned. There could be both vertical and horizontal stacks starting at a single SWITCH gadget, but this could only increase the values of the left hand sides of (3)–(6).

Lemma 40. *Inequalities* (3)–(6) *imply that, for each* i, $y_{3i}+y_{3i+1}+y_{3i+2}=1$.

Proof. By induction on k, we show that for $i<k$, we have

$$y_{3i}+y_{3i+1}+y_{3i+2}=1. \tag{7}$$

We will simultaneously show (in the same induction) that:

$$\sum_{p=0}^{k-1}\sum_{q=0}^{3+p-1}\sum_{i=p}^{k-1}\sum_{m=0}^{2}(1-y_{3i+m})\left(2-x\left(\ell_{3i+m},c_{p,q}\right)\right)\geq 4t_{k-1}-2k \tag{8}$$

The left hand side of (8) counts how much the tester columns grow in total when crossing literal rows in the first k clause components. Recall that each of the t_{k-1} tester columns is allowed to grow by 4 squares, so in total they can grow by $4t_{k-1}$ squares. So what (8) says is that at most $2k$ squares worth of slack is propagating downward after the first k clause components.

The base case for the induction is $k=0$. Since there are no literals and no tester columns, the result is trivial in the base case.

For induction, suppose the above holds for some k. First, we show that $y_{3k}+y_{3k+1}+y_{3k+2}\leq 1$. By using (7) in (3), we have, for $0\leq j<1+2t_k$:

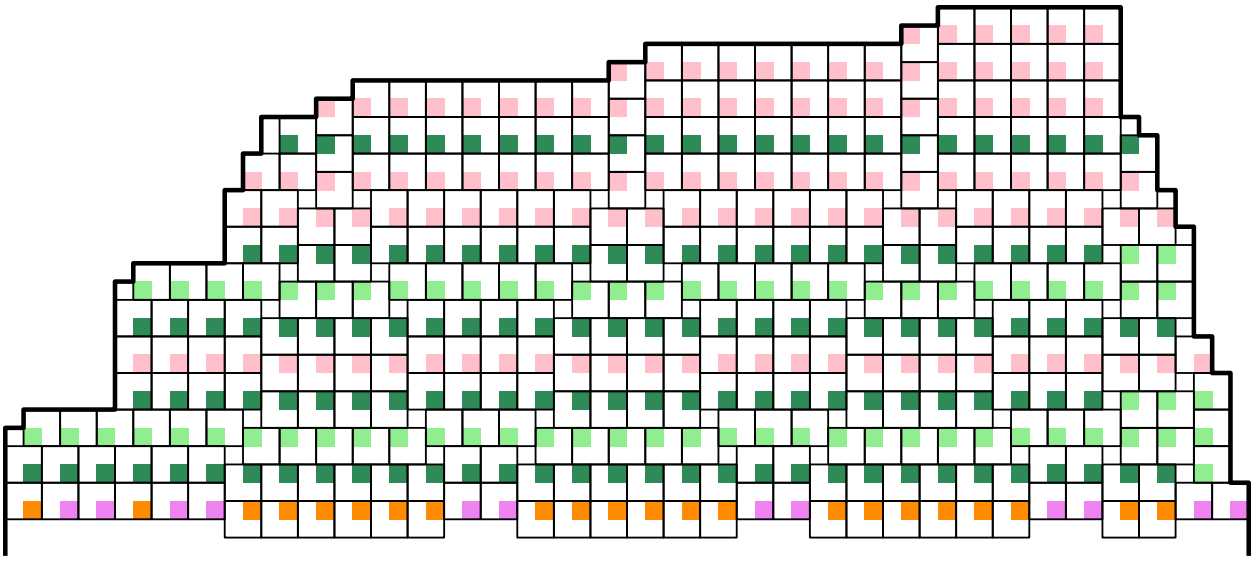

Figure 63: A perfect packing when all the squares in the third literal column are allowed to lower.

$$2k + 2(y_{3k} + y_{3k+1} + y_{3k+2}) + \sum_{p=0}^{k} \sum_{q=0}^{3+p-1} x(r_{k,j}, c_{p,q}) \leq 2\,(k + 1 + t_k) \tag{9}$$

We want to show that there is a value of j where the sum of the $x(r_{k,j}, c_{p,q})$ is at least $2t_k - 1$. All the terms in (6) are positive, so for each p, q we can extract that:

$$\sum_{j=0}^{2t_k} (2 - x\,(r_{k,j}, c_{p,q})) \leq 4$$

Summing over the t_k values of p and q:

$$\sum_{j=0}^{2t_k} \sum_{p=0}^{k} \sum_{q=0}^{3+p-1} (2 - x\,(r_{k,j}, c_{p,q})) \leq 4t_k$$

The outermost sum sums over $2t_k + 1$ positive integral values. Since $2(2t_k + 1) < 4t_k$, at least one of these values must be less than 2 (and so less than or equal to 1). That is to say, there is some j such that:

$$\sum_{p=0}^{k} \sum_{q=0}^{3+p-1} (2 - x\,(r_{k,j}, c_{p,q})) \leq 1$$

This can be rearranged to obtain:

$$\sum_{p=0}^{k} \sum_{q=0}^{3+p-1} x\,(r_{k,j}, c_{p,q}) \geq 2t_k - 1$$

Subtracting this from (9), we see that:

$$2(y_{3k} + y_{3k+1} + y_{3k+2}) \leq 3$$

Since the y_i are integer valued, this implies that $y_{3k} + y_{3k+1} + y_{3k+2} \leq 1$.

Next we should show that $y_{3k} + y_{3k+1} + y_{3k+2} \geq 1$. Suppose for contradiction that

$$y_{3k} = y_{3k+1} = y_{3k+2} = 0 \tag{10}$$

By subtracting the inductive hypothesis (8) from the sum of (6) over $0 \leq p < k$ and $0 \leq q < 3+p$, we see that:

$$2k \geq \sum_{p=0}^{k-1} \sum_{q=0}^{3+p-1} \sum_{m=0}^{2} (2 - x(\ell_{3k+m}, c_{p,q})) = 6t_{k-1} - \sum_{p=0}^{k-1} \sum_{q=0}^{3+p-1} \sum_{m=0}^{2} x(\ell_{3k+m}, c_{p,q}) \tag{11}$$

Adding up (4) and versions of (5) for both values $m \in \{1, 2\}$ (and simplifying using (10) and (7)):

$$6k + \sum_{m=0}^{2} \sum_{p=0}^{k} \sum_{q=0}^{3+p-1} x(\ell_{3k+m}, c_{p,q}) \leq 4 + 6(k + t_{k-1})$$

Splitting the sum into the cases $p = k$ and $p < k$, we get:

$$6k + \sum_{m=0}^{2} \sum_{q=0}^{3+k-1} x(\ell_{3k+m}, c_{k,q}) + \sum_{m=0}^{2} \sum_{p=0}^{k-1} \sum_{q=0}^{3+p-1} x(\ell_{3k+m}, c_{p,q}) \leq 4 + 6(k + t_{k-1})$$

So together with (11), we see that:

$$\sum_{m=0}^{2} \sum_{q=0}^{3+k-1} x(\ell_{3k+m}, c_{k,q}) \leq 4 + 2k$$

Since $2(3 + k) > 4 + 2k$, there must be some value of q for which:

$$\sum_{m=0}^{2} x(\ell_{3k+m}, c_{k,q}) \leq 1$$

This implies that:

$$\sum_{m=0}^{2} (2 - x(\ell_{3k+m}, c_{k,q})) \geq 5$$

From (6) we can extract that:

$$\sum_{m=0}^{2} (2 - x(\ell_{3k+m}, c_{k,q})) \leq 4,$$

a contradiction. We conclude that y_{3k}, y_{3k+1} and y_{3k+2} can't all be zero, and since we already saw that $y_{3k} + y_{3k+1} + y_{3k+2} \leq 1$, we get $y_{3k} + y_{3k+1} + y_{3k+2} = 1$. It remains to check that (8) holds for $k + 1$.

For a general function f in two variables:

$$\sum_{p=0}^{k}\sum_{i=p}^{k} f(p,i) = \sum_{p=0}^{k-1}\sum_{i=p}^{k-1} f(p,i) + \sum_{p=0}^{k} f(p,k)$$

So going from k to $k+1$, the left hand side of (8) increases by:

$$\sum_{p=0}^{k}\sum_{q=0}^{3+p-1}\sum_{m=0}^{2}(1-y_{3k+m})\left(2-x\left(\ell_{3k+m},c_{p,q}\right)\right)$$

Going from k to $k+1$, the right hand side of (8) increases by $4\left(3+k\right)-2$ (since $t_k - t_{k-1}$ is $3+k$). So to show that (8) holds for $k+1$, it is sufficient to show that:

$$\sum_{p=0}^{k}\sum_{q=0}^{3+p-1}\sum_{m=0}^{2}(1-y_{3k+m})\left(2-x\left(\ell_{3k+m},c_{p,q}\right)\right) \geq 4\left(3+k\right)-2$$

There are 3 cases to check based on which of y_{3k}, y_{3k+1} or y_{3k+2} is equal to 1. Consider the case $y_{3k+2}=1$. Summing (4) for $i=k$ and (5) for $i=k$ and $m=1$, and simplifying using (7), we get:

$$4k + \sum_{m=0}^{1}\sum_{p=0}^{k}\sum_{q=0}^{3+p-1} x(\ell_{3k+m},c_{p,q}) \leq 2 + 4\left(k+t_{k-1}\right)$$

Rearranging (using that the number of pairs of (p,q) is $t_k = t_{k-1}+3+k$), we obtain:

$$\sum_{p=0}^{k}\sum_{q=0}^{3+p-1}\sum_{m=0}^{1}\left(2-x(\ell_{3k+m},c_{p,q})\right) \geq 4(3+k)-2$$

So:

$$\sum_{p=0}^{k}\sum_{q=0}^{3+p-1}\sum_{m=0}^{2}(1-y_{3k+m})\left(2-x(\ell_{3k+m},c_{p,q})\right) \geq 4(3+k)-2$$

as required. The cases $y_{3k}=1$ and $y_{3k+1}=1$ are similar. By induction, this completes the proof. □

Lemma 41. *Suppose there is a perfect packing of the clause components where the membrane clause squares push out. Then if Φ has a clause of form $\neg x_i \vee \neg x_j \vee \neg x_k$ (resp. $x_i \vee x_j \vee x_k$), then there is a block of membrane connecting squares in the upper (resp. lower) membrane row that all push in and connects to a PUSH-UP-IF-PLUS (resp. PUSH-DOWN-IF-MINUS) gadget for one of the variables x_i, x_j or x_k.*

Proof. Whenever a SWITCH gadget creates a vertical stack, the corresponding block of membrane connecting squares must all push in (by Lemma 24). Lemma 40 says that, for each clause, at least one of the SWITCH gadgets must create a vertical stack. So for clause of form $\neg x_i \vee \neg x_j \vee \neg x_k$ (resp. $x_i \vee x_j \vee x_k$) in Φ, one of the variables has a PUSH-UP-IF-PLUS (resp. PUSH-UP-IF-MINUS) gadget that is connected to a block of membrane connecting squares that all push in. □

4.9 Proof of Theorem 1

We are now ready to give the proof of our main theorem.

Theorem 1. *The problem* 2×2-Square-Packing *is NP-hard, even for orthogonally convex grid polygons.*

Proof. Let Φ be an instance of Monotone-Clover-3SAT. Construct the orthogonally convex grid polygon P and the number k of squares to be packed as above. The number k is $\mathcal{O}(m^8)$ where m is the number of variables and clauses in Φ. This size is dominated by squares in and surrounding the tester rows. The polygon P can be explicitly constructed in polynomial time, as outlines in the following.

Assume that Φ has m clauses and v variables. We start by drawing the full schematics including all the variable rows, helper rows, vertical static shifts, pinch columns, helper row columns, redundancy columns, literal rows, literal columns, tester rows, tester columns, and PUSH columns. We then compute the size that each stack is allowed to grow (taking into account any static shifts), and construct the polygon by gluing together the various gadgets appropriately. In total:

- There are v variable gadgets, which *in total* have $\mathcal{O}(m)$ helper rows. Since variables that don't appear in any clauses can be excluded, we can assume that $v = \mathcal{O}(m)$.
- Each variable gadget has an upper and lower pinch column, $\mathcal{O}(m)$ helper row columns, and $\mathcal{O}(m)$ redundancy columns. All of these grow by at most $\mathcal{O}(m)$ before terminating. Eacg if these columns has a vertical static shift, so the total number of vertical static shifts in the clause gadgets is $\mathcal{O}(m^2)$.
- There are $3m$ literal columns and $\mathcal{O}(m^2)$ tester columns, which each grow by $\mathcal{O}(m)$. These continue as PUSH columns in the variable components.
- There are $3m$ literal rows and $\mathcal{O}(m^3)$ tester rows, which each grow by $\mathcal{O}(m^2)$.

The size of the polygon is dominated by squares in and around the literal rows. There are $\mathcal{O}(m^3)$ literal rows, growing to a size of $\mathcal{O}(m^2)$. Since there are $\mathcal{O}(m^2)$ columns growing to a width of $\mathcal{O}(m)$ each, the total number of squares in the polygon is at most $\mathcal{O}(m^8)$.

Lemmas 28 and 39 say that there is a packing of P with k squares of size 2×2 whenever Φ has a satisfying assignment. Lemmas 38 and 41 say that any such packing of P must correspond to a satisfying assignment of Φ. So by the NP-hardness of Monotone-Clover-3SAT, the problem 2×2-Square-Packing is NP-hard for orthogonally convex polygons. □

5 Concluding remarks

To our knowledge, these represent the first results on NP-hardness for packing or covering a simple polygon with identical (fixed) shapes and the first NP-hardness result for partitioning simple polygons into connected pieces. There are many interesting problems that are known to be hard for polygons with holes but with unknown complexity for simple polygons. Until now, techniques for showing hardness of many of these problems have not been available.

The problem 2×2-Square-Packing in a grid polygon is equivalent to Maximum-Independent-Set on a grid graph G with diagonals added; see Figure 64. We can define G to be *orthogonally*

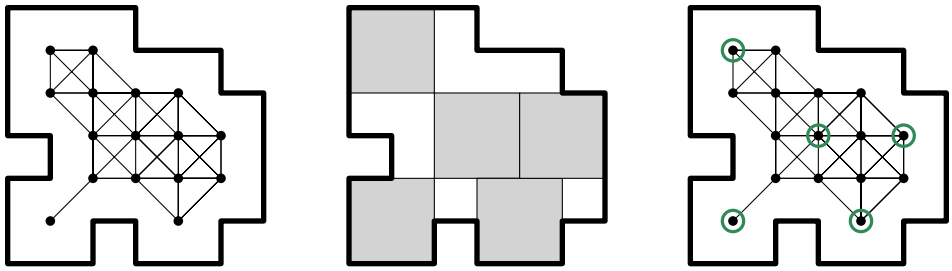

Figure 64: The equivalence between 2×2-Square-Packing and Maximum-Independent-Set. Left: A polygon and the equivalent instance of Maximum-Independent-Set. Middle: A packing with 2×2 squares. Right: The corresponding independent set.

convex if whenever two vertices $u, v \in V(G)$ are from the same row or column, then all grid points between u and v are also vertices of G. Our result has the following interesting consequence:

Corollary 42. *The maximum independent set problem for orthogonally convex grid graphs with diagonals is NP-hard.*

The most natural class of polygons for which the complexity of 2×2-Square-Packing is unresolved is *staircase* grid polygons, i.e., grid polygons where the boundary can be partitioned into two chains, both of which are simultaneously x- and y-monotone. Many of our ideas about packing in orthogonally convex polygons work here, but we do not know of a way to make clause components that could be used for this problem. Another related problem is 2×2-Square-Packing when the polygon P is convex, but not a grid polygon.

Allowing the squares to rotate arbitrarily changes the problem significantly, and although it seems obvious that it makes the packing problem no more tractable than in the axis-aligned case, we have not found a way to prove hardness. A long line of mathematical research has been devoted to this problem when the container P is also a (larger) square. This was initiated by Erdős and Graham [26] in 1975, and it is still an active research area [21]. The complicated nature of this problem is exemplified by the fact that even for a mere 11 unit squares, it is unknown what is the smallest square in which they can be packed [33]. It may be possible to apply our ideas to show that it is NP-hard to pack unit squares in a simple polygon with rotation, but this would require at minimum a much more sophisticated version of the reference center idea.

For certain geometric shapes other than squares, our techniques may be useful for showing hardness of the associated packing problems in simple polygons. The problem that seems most amenable to this approach is packing equilateral triangles (with 180-degree rotations allowed). We have not taken the effort to figure out the details, as packing equilateral triangles seems to be of limited interest. Hardness of packing unit disks may be possible, but has some of the same problems as packing squares with rotations.

It may also be possible to show hardness for packing $n \times k$ rectangles with 90-degree rotations allowed into a simple (not necessarily grid) polygon using these techniques. It seems much more difficult to apply our ideas to the problem of packing $n \times k$ rectangles into a simple *grid* polygon. This problem is in P for 1×2 rectangles, but little is known even for 1×3 rectangles.

In a forthcoming paper (with another set of authors), we show that *reconfiguration* from one packing of axis-aligned unit squares to another is PSPACE-hard, even in a simple polygon. The construction relies on a slight modification of the construction from Section 2. Until now, it was

only known that reconfiguration in a polygon with holes is PSPACE-hard [52].

References

[1] Anders Aamand, Mikkel Abrahamsen, Thomas D. Ahle, and Peter M. R. Rasmussen. "Tiling with Squares and Packing Dominos in Polynomial Time". In: *ACM Trans. Algorithms* 19.3 (2023), 30:1–30:28. DOI: 10.1145/3597932.

[2] Mikkel Abrahamsen. "Covering Polygons is Even Harder". In: *62nd IEEE Annual Symposium on Foundations of Computer Science (FOCS 2021)*. 2021, pp. 375–386. DOI: 10.1109/FOCS52979.2021.00045.

[3] Mikkel Abrahamsen, Anna Adamaszek, and Tillmann Miltzow. "The Art Gallery Problem is $\exists\mathbb{R}$-complete". In: *J. ACM* 69.1 (2022), 4:1–4:70. DOI: 10.1145/3486220.

[4] Mikkel Abrahamsen, Joakim Blikstad, André Nusser, and Hanwen Zhang. "Minimum Star Partitions of Simple Polygons in Polynomial Time". In: *Symposium on Foundations of Computer Science (FOCS 2024)*. 2024. DOI: 10.48550/ARXIV.2311.10631.

[5] Mikkel Abrahamsen, Tillmann Miltzow, and Nadja Seiferth. "Framework for ER-Completeness of Two-Dimensional Packing Problems". In: (2020). Ed. by Sandy Irani, pp. 1014–1021. DOI: 10.1109/FOCS46700.2020.00098.

[6] Mikkel Abrahamsen and Nichlas Langhoff Rasmussen. "Partitioning a Polygon Into Small Pieces". In: *CoRR* abs/2211.01359 (2022). DOI: 10.48550/ARXIV.2211.01359.

[7] Pankaj K. Agarwal, Marc J. van Kreveld, and Subhash Suri. "Label placement by maximum independent set in rectangles". In: *Comput. Geom.* 11.3-4 (1998), pp. 209–218. DOI: 10.1016/S0925-7721(98)00028-5.

[8] Sarah R. Allen and John Iacono. "Packing identical simple polygons is NP-hard". In: *CoRR* abs/1209.5307 (2012). DOI: 10.48550/arXiv.1209.5307.

[9] Esther M. Arkin, Sándor P. Fekete, and Joseph S. B. Mitchell. "Approximation algorithms for lawn mowing and milling". In: *Comput. Geom.* 17.1-2 (2000), pp. 25–50. DOI: 10.1016/S0925-7721(00)00015-8.

[10] Esther M. Arkin, Martin Held, and Christopher L. Smith. "Optimization Problems Related to Zigzag Pocket Machining". In: *Algorithmica* 26.2 (2000), pp. 197–236. DOI: 10.1007/S004539910010.

[11] Takao Asano, Tetsuo Asano, and Hiroshi Imai. "Partitioning a polygonal region into trapezoids". In: *J. ACM* 33.2 (1986), pp. 290–312. DOI: 10.1145/5383.5387. Preliminary version af FOCS 1983.

[12] Christoph Baur and Sándor P. Fekete. "Approximation of Geometric Dispersion Problems". In: *Algorithmica* 30.3 (2001), pp. 451–470. DOI: 10.1007/S00453-001-0022-X.

[13] Danièle Beauquier, Maurice Nivat, Eric Rémila, and Mike Robson. "Tiling Figures of the Plane with Two Bars". In: *Comput. Geom.* 5 (1995), pp. 1–25. DOI: 10.1016/0925-7721(94)00015-N.

[14] Mark de Berg and Amirali Khosravi. "Optimal Binary Space Partitions for Segments in the Plane". In: *Int. J. Comput. Geom. Appl.* 22.3 (2012), pp. 187–206. DOI: 10.1142/S0218195912500045.

[15] Francine Berman, David S. Johnson, Frank Thomson Leighton, Peter W. Shor, and Larry Snyder. "Generalized Planar Matching". In: *J. Algorithms* 11.2 (1990), pp. 153–184. DOI: 10.1016/0196-6774(90)90001-U.

[16] Francine Berman, Frank Thomsons Leighton, and Lawrence Snyder. *Optimal tile salvage*. Tech. rep. 81–396. Purdue University, 1981. URL: https://docs.lib.purdue.edu/cstech/322.

[17] Timothy M. Chan. "A note on maximum independent sets in rectangle intersection graphs". In: *Inf. Process. Lett.* 89.1 (2004), pp. 19–23. DOI: 10.1016/J.IPL.2003.09.019.

[18] Bernard Chazelle. "Approximation and decomposition of shapes". In: *Algorithmic and Geometric Aspects of Robotics*. Ed. by Jacob T. Schwartz and Chee-Keng Yap. Vol. 1. Advances in Robotics. 1987, pp. 145–185.

[19] Bernard Chazelle and David P. Dobkin. "Optimal convex decompositions". In: *Computational Geometry*. Ed. by Godfried T. Toussaint. Vol. 2. Machine Intelligence and Pattern Recognition. 1985, pp. 63–133. DOI: 10.1016/B978-0-444-87806-9.50009-8. Preliminary version at STOC 1979.

[20] Bernard Chazelle and Leonidas Palios. "Decomposition algorithms in geometry". In: *Algebraic Geometry and its applications*. Ed. by Chandrajit L. Bajaj. 1994, pp. 419–447. DOI: 10.1007/978-1-4612-2628-4_27.

[21] Fan Chung and Ron Graham. "Efficient Packings of Unit Squares in a Large Square". In: *Discret. Comput. Geom.* 64.3 (2020), pp. 690–699. DOI: 10.1007/s00454-019-00088-9.

[22] Erik D. Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, Henk Meijer, Mark H. Overmars, and Sue Whitesides. "Separating point sets in polygonal environments". In: *20th ACM Symposium on Computational Geometry (SoCG 2004)*. 2004, pp. 10–16. DOI: 10.1145/997817.997822.

[23] Erik D. Demaine, Sándor P. Fekete, and Robert J. Lang. "Circle Packing for Origami Design Is Hard". In: *CoRR* abs/1008.1224 (2010). DOI: 10.48550/arXiv.1008.1224.

[24] Erik D. Demaine, Joseph S. B. Mitchell, and Joseph O'Rourke, eds. *The Open Problems Project. Problem 56: Packing Unit Squares in a Simple Polygon*. Accessed: 2023-12-21. URL: https://topp.openproblem.net/p56.

[25] Dania El-Khechen, Muriel Dulieu, John Iacono, and Nikolaj van Omme. "Packing 2×2 unit squares into grid polygons is NP-complete". In: *21st Annual Canadian Conference on Computational Geometry (CCCG 2009)*. 2009, pp. 33–36. URL: http://cccg.ca/proceedings/2009/cccg09_09.pdf.

[26] Paul Erdős and Ron Graham. "On packing squares with equal squares". In: *Journal of Combinatorial Theory, Series A* 19.1 (1975), pp. 119–123. DOI: 10.1016/0097-3165(75)90099-0.

[27] Sándor P. Fekete, Tom Kamphans, Alexander Kröller, Joseph S. B. Mitchell, and Christiane Schmidt. "Exploring and Triangulating a Region by a Swarm of Robots". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX 2011)*. 2011, pp. 206–217. DOI: 10.1007/978-3-642-22935-0_18.

[28] Sándor P. Fekete and Henk Meijer. "The one-round Voronoi game replayed". In: *Comput. Geom.* 30.2 (2005), pp. 81–94. DOI: 10.1016/J.COMGEO.2004.05.005.

[29] Sándor P. Fekete and Joseph S. B. Mitchell. "Terrain Decomposition and Layered Manufacturing". In: *Int. J. Comput. Geom. Appl.* 11.6 (2001), pp. 647–668. DOI: 10.1142/S0218195901000687.

[30] Sándor P. Fekete, Joseph S. B. Mitchell, and Karin Beurer. "On the Continuous Fermat-Weber Problem". In: *Oper. Res.* 53.1 (2005), pp. 61–76. DOI: 10.1287/OPRE.1040.0137.

[31] Robert J. Fowler, Mike Paterson, and Steven L. Tanimoto. "Optimal Packing and Covering in the Plane are NP-Complete". In: *Inf. Process. Lett.* 12.3 (1981), pp. 133–137. DOI: 10.1016/0020-0190(81)90111-3.

[32] Michael R. Garey and David S. Johnson. *Computers and intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[33] Thierry Gensane and Philippe Ryckelynck. "Improved Dense Packings of Congruent Squares in a Square". In: *Discret. Comput. Geom.* 34.1 (2005), pp. 97–109. DOI: 10.1007/s00454-004-1129-z.

[34] Dorit S. Hochbaum and Wolfgang Maass. "Approximation Schemes for Covering and Packing Problems in Image Processing and VLSI". In: *J. ACM* 32.1 (1985), pp. 130–136. DOI: 10.1145/2455.214106.

[35] Hiroshi Imai and Takao Asano. "Efficient Algorithms for Geometric Graph Search Problems". In: *SIAM J. Comput.* 15.2 (1986), pp. 478–494. DOI: 10.1137/0215033.

[36] J. Mark Keil. "Polygon decomposition". In: *Handbook of computational geometry*. Ed. by Jörg-Rüdiger Sack and Jorge Urrutia. 1999. Chap. 11, pp. 491–518. DOI: 10.1016/B978-044482537-7/50012-7.

[37] J. Mark Keil and Jörg-R. Sack. "Minimum Decompositions of Polygonal Objects". In: *Computational Geometry*. Ed. by Godfried T. Toussaint. Vol. 2. Machine Intelligence and Pattern Recognition. 1985, pp. 197–216. DOI: 10.1016/B978-0-444-87806-9.50012-8.

[38] Dania El-Khechen. "Decomposing and packing polygons". PhD thesis. Concordia University, 2009. URL: https://spectrum.library.concordia.ca/id/eprint/976664/.

[39] Heuna Kim and Tillmann Miltzow. "Packing Segments in a Simple Polygon is APX-hard". In: *European Conference on Computational Geometry (EuroCG 2015)*. http://eurocg15.fri.uni-lj.si/pub/eurocg15-book-of-abstracts.pdf. 2015, pp. 24–27.

[40] David G. Kirkpatrick, Irina Kostitsyna, and Valentin Polishchuk. "Hardness Results for Two-Dimensional Curvature-Constrained Motion Planning". In: *23rd Annual Canadian Conference on Computational Geometry (CCCG 2011)*. 2011. URL: http://www.cccg.ca/proceedings/2011/papers/paper99.pdf.

[41] Joseph Y.-T. Leung, Tommy W. Tam, C. S. Wong, Gilbert H. Young, and Francis Y. L. Chin. "Packing Squares into a Square". In: *J. Parallel Distributed Comput.* 10.3 (1990), pp. 271–275. DOI: 10.1016/0743-7315(90)90019-L.

[42] David Lichtenstein. "Planar Formulae and Their Uses". In: *SIAM J. Comput.* 11.2 (1982), pp. 329–343. DOI: 10.1137/0211025.

[43] Andrzej Lingas. "The Power of Non-Rectilinear Holes". In: *9th International Colloquium on Automata, Languages, and Programming (ICALP 1982)*. 1982, pp. 369–383. DOI: 10.1007/BFB0012784.

[44] Anna Lubiw and Debajyoti Mondal. "On compatible triangulations with a minimum number of Steiner points". In: *Theor. Comput. Sci.* 835 (2020), pp. 97–107. DOI: 10.1016/J.TCS.2020.06.014.

[45] Wolfgang Mulzer and Günter Rote. "Minimum-weight triangulation is NP-hard". In: *J. ACM* 55.2 (2008), 11:1–11:29. DOI: 10.1145/1346330.1346336.

[46] Joseph O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.

[47] Joseph O'Rourke and Kenneth J. Supowit. "Some NP-hard polygon decomposition problems". In: *IEEE Trans. Inf. Theory* 29.2 (1983), pp. 181–189. DOI: 10.1109/TIT.1983.1056648.

[48] Joseph O'Rourke, Subash Suri, and Csaba D. Tóth. "Polygons". In: *Handbook of discrete and computational geometry*. Ed. by J. E. Goodman and J. O'Rourke. Third edition. 2018. Chap. 30, pp. 787–810. DOI: 10.1201/9781315119601.

[49] Alexander Pilz. "Planar 3-SAT with a Clause/Variable Cycle". In: *Discret. Math. Theor. Comput. Sci.* 21.3 (2019). DOI: 10.23638/DMTCS-21-3-18.

[50] André van Renssen and Bettina Speckmann. "The 2×2 Simple Packing Problem". In: *23rd Annual Canadian Conference on Computational Geometry (CCCG 2011)*. 2011. URL: http://www.cccg.ca/proceedings/2011/papers/paper14.pdf.

[51] Thomas C. Shermer. "Recent results in art galleries". In: *Proc. IEEE* 80.9 (1992), pp. 1384–1399. DOI: 10.1109/5.163407.

[52] Kiril Solovey and Dan Halperin. "On the hardness of unlabeled multi-robot motion planning". In: *Int. J. Robotics Res.* 35.14 (2016), pp. 1750–1759. DOI: 10.1177/0278364916672311.