Reconfiguration of Unit Squares and Disks: PSPACE-Hardness in Simple Settings

Mikkel Abrahamsen
University of Copenhagen, Denmark

Kevin Buchin
TU Dortmund, Germany

Maike Buchin
Ruhr-Universität Bochum, Germany

Linda Kleist
Universität Potsdam, Germany

Maarten Löffler
Utrecht University, The Netherlands

Lena Schlipf
Universität Tübingen, Germany

André Schulz
FernUniversität Hagen, Germany

Jack Stade
University of Copenhagen, Denmark

Abstract

We study well-known reconfiguration problems. Given a start and a target configuration of geometric objects in a polygon, we wonder whether we can move the objects from the start configuration to the target configuration while avoiding collisions between the objects and staying within the polygon. Problems of this type have been considered since the early 80s by roboticists and computational geometers. In this paper, we study some of the simplest possible variants where the objects are labeled or unlabeled unit squares or unit disks. In unlabeled reconfiguration, the objects are identical, so that any object is allowed to end at any of the targets positions. In the labeled variant, each object has a designated target position. The results for the labeled variants are direct consequences from our insights on the unlabeled versions.

We show that it is PSPACE-hard to decide whether there exists a reconfiguration of (unlabeled/labeled) unit squares even in a simple polygon. Previously, it was only known to be PSPACE-hard in a polygon with holes for both the unlabeled and labeled version [Solovey and Halperin, Int. J. Robotics Res. 2016]. Our proof is based on a result of independent interest, namely that reconfiguration between two satisfying assignments of a formula of Monotone-Planar-3-Sat is also PSPACE-complete. The reduction from reconfiguration of Monotone-Planar-3-Sat to reconfiguration of unit squares extends techniques recently developed to show NP-hardness of packing unit squares in a simple polygon [Abrahamsen and Stade, FOCS 2024]. We also show PSPACE-hardness of reconfiguration of (unlabeled/labeled) unit disks in a polygon with holes. Previously, it was known that unlabeled reconfiguration of disks of two different sizes was PSPACE-hard [Brocken, van der Heijden, Kostitsyna, Lo-Wong and Surtel, FUN 2021].

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Complexity theory and logic

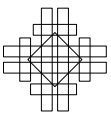
Keywords and phrases reconfiguration, unit square, unit disk, unlabeled, labeled, simple polygon, polygon

Digital Object Identifier 10.4230/LIPIcs.SoCG.2025.1

Related Version *Full Version*: `https://arxiv.org/abs/2412.21017`

Funding *Mikkel Abrahamsen*: Supported by Independent Research Fund Denmark, grant 1054-00032B, and by the Carlsberg Foundation, grant CF24-1929.
Jack Stade: Supported by Independent Research Fund Denmark, grant 1054-00032B, and by the Carlsberg Foundation, grant CF24-1929.

Acknowledgements This work was initiated at the Dagstuhl seminar 24072: Triangulations in Geometry and Topology. We thank the organizers and the participants for the fruitful atmosphere.



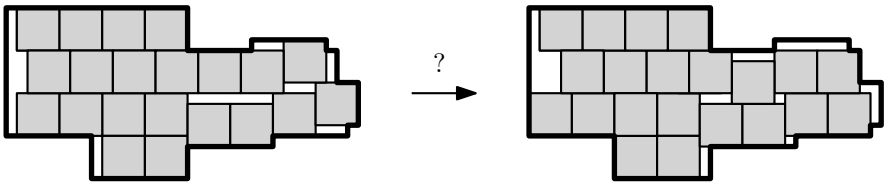


Figure 1 Can the left configuration of unit squares in a simple polygon be reconfigured to the right configuration of unit squares?

1 Introduction

In *geometric reconfiguration* we are given a start configuration of geometric *objects* and a desired target configuration. A particularly well-studied geometric configuration problem is *multi-robot motion planning*. The goal is to move the robots (or objects) from the start position to the target configuration while avoiding collisions. Problems of this type have been considered since the early 80s by roboticists and computational geometers. Hopcroft, Schwartz and Sharir [14] showed already in 1984 that reconfiguration of labeled rectangles moving in a rectangle is PSPACE-hard. The motivation of their work was "to find moving systems (necessarily involving arbitrarily many degrees of freedom) whose geometry is *as simple as possible*, but for which the motion planning problem is still PSPACE-hard".

In this paper, we study some of the simplest possible variants where the objects are unlabeled or labeled axis-aligned congruent squares or unit disks. In case of squares, the objects have to stay axis-aligned during the reconfiguration. For an example, consider Figure 1. Obstacles are modeled by the workspace in which the objects lie. At any time, the objects have to avoid the complement of the workspace. Depending on the variant, the workspace is given as a polygon, a simple polygon, or a grid polygon, that is an orthogonal polygon whose vertex coordinates are integers. In unlabeled reconfiguration, the objects are indistinguishable, so that any object is allowed to end at any of the target positions. In labeled reconfiguration the target position for every object is explicitly given. Table 1 presents an overview of the known results and the new results of this paper.

Table 1 Summary of results in motion planning for multiple objects.

Contribution	Labeled	Complexity	Objects	Workspace
[14]	yes	PSPACE-hard	rectangles	rectangle
[12]	yes	PSPACE-hard	1×2 rectangles	rectangle
[19]	yes	PSPACE-hard	unit squares	polygon (with holes)
this work	yes	PSPACE-hard	unit squares	simple polygon
[16]	yes	P	unit squares	grid polygon
[19]	no	PSPACE-hard	unit squares	polygon (with holes)
this work	no	PSPACE-hard	unit squares	simple polygon
easy exercise	no	trivial "yes"	unit squares	grid polygon
[21]	yes	strongly NP-hard	disks of three sizes	simple polygon
this work	yes	PSPACE-hard	unit disks	polygon (with holes)
[4]	no	PSPACE-hard	disks of two sizes	polygon (with holes)
this work	no	PSPACE-hard	unit disks	polygon (with holes)

Recent results of PSPACE-hardness of geometric reconfiguration problems have been proven by a reduction from Nondeterministic Constrained Logic (NCL) [11, 13]. This holds for instance for the works [4, 5, 12, 19]. Here, a polygon is built "on top of" an instance of NCL, which is in itself a plane graph (the problem NCL will be described in detail in Section 2). This technique inevitably leads to a polygon with holes, because the bounded faces of the NCL-instance are turned into holes of the polygon.

A similar phenomenon appears in many proofs of NP-hardness of two-dimensional geometric problems where the input is a polygon: Such reductions are often from a variant of Planar-3-Sat and work by building a polygon on top of the Planar-3-Sat-instance we are reducing from. Again, this leads to a polygon with a hole for every bounded face of the graph representing the Planar-3-Sat-formula.

Abrahamsen and Stade [1] recently overcame this obstacle by making a novel reduction from Planar-3-Sat in order to obtain NP-hardness of the well-known problem of packing unit squares in a simple polygon; a problem that had been known to be NP-hard for polygons with holes since 1981 [8]. Interestingly, Abrahamsen and Stade [1] used a non-planar geometric realization of a planar formula, where variables are represented as horizontal rows and edges are vertical columns. These rows and columns are allowed to cross, 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. The rows and columns of the drawing are represented by rows and columns of squares, and these likewise cross 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.

Our contribution. In this paper we show PSPACE-hardness of reconfiguration of unit squares in a simple polygon. To this end, we first show a result of independent interest. The solution space of boolean formulas is given by an hypercube graph, where edges are formed between assignments which differ in exactly one variable. The satisfying assignments of a formula ϕ define a subgraph of this graph, which we call $G(\phi)$. We call two satisfying assignments *connected* if they are connected in $G(\phi)$. The reconfiguration problem for boolean formulas asks if two satisfying assignments are connected. We show the following result.

▶ **Theorem 1.** *The reconfiguration problem for Monotone-Planar-3-Sat formulas is* PSPACE*-complete.*

For some other versions of 3-Sat, the reconfiguration problem has likewise been shown to be PSPACE-hard, such as Monotone-Planar-Not-All-Equal-3-SAT [6]. In Table 2, we give a summary of some variants of 3-Sat and the complexity of their satisfyability and reconfiguration problems. Interestingly, most satisfiability and reconfiguration problems are hard, but there are some exceptions that can be solved in polynomial time.

Using Theorem 1, we show PSPACE-hardness for reconfiguration of squares.

▶ **Theorem 2.** *Unlabeled reconfiguration of* 8×8 *squares in simple grid polygons is* PSPACE*-complete.*

This implies PSPACE-hardness for reconfiguration of unit squares in simple polygons. We also remark that the size of the coordinates of the constructed polygon is $O(n^2)$, where n is the number of corners.

In our construction, the squares can only move locally. Hence, we exactly know the start and end position of each square and also obtain the same result for the labeled version.

Table 2 Overview of computational complexity for the satisfiability and reconfiguration problems of fundamental 3-Sat variants.

	Satisfiability	Reconfiguration
3-Sat	NP-complete [15]	PSPACE-complete [10]
Planar-3-Sat	NP-complete [17]	PSPACE-complete [6]
Positive-1-in-3-SAT	NP-complete [9]	P ('no' iff assignments differ)
Positive-Not-All-Equal-3-SAT	NP-complete [18]	PSPACE-complete [10]
Mon.-Planar-Not-All-Eequal-3-SAT	P [18]	PSPACE-complete [6]
Monotone-Planar-3-Sat	NP-complete [7]	PSPACE-complete [This work]

▶ **Corollary 3.** *Labeled reconfiguration of* 8×8 *squares in simple grid polygons is* PSPACE*-complete.*

Theorem 2 and Corollary 3 strengthen the results of Solovey and Halperin [19], who showed that the problem is PSPACE-hard for polygons with holes in the unlabeled and labeled variants. In contrast, in grid polygons, it is an easy exercise to show that reconfiguration of unlabeled unit squares is always doable (if the number of squares agrees) and reconfiguration of labeled unit squares is polynomial time decidable [16]. Hence, our result is close to the border of polynomial-time decidability.

Lastly, we consider the reconfiguration of unit disks.

▶ **Theorem 4.** *Reconfiguration of unlabeled unit disks in a polygon (possibly with holes) is* PSPACE*-hard.*

Before, it was only known that reconfiguration of unlabeled disks of two different sizes was PSPACE-hard, as shown by Brocken, van der Heijden, Kostitsyna, Lo-Wong and Surtel [4]. For disks of the same size so far only positive results were known under the assumption that there is sufficient separation between the start and target positions of the disks in a simple polygon [2, 3] and also in a polygon with holes [20]. The main technical challenge here is to artificially discretize the continuous space of disk configurations, to avoid allowing global states in which many gadgets are locally in an "in-between" state which would break the reduction. We overcome this challenge by designing a special edge gadget that enforces the consistent orientation of an edge, and argue that only one edge may be reoriented at a time. Thus, our results in particular show that an assumption such as the separation assumption in [20] is necessary if $\mathsf{P} \neq \mathsf{PSPACE}$.

In fact, our constructed polygon guarantees that no two disks may swap their positions. Therefore, hardness for the labeled version follows immediately.

▶ **Corollary 5.** *Reconfiguration of labeled unit disks in a polygon (possibly with holes) is* PSPACE*-hard.*

Previously, Spikaris and Yap [21] showed NP-hardness for labeled disks of three different sizes.

Organization of the paper. The rest of the paper is structured as follows. In Section 2, we briefly introduce NCL and and present the general idea of the NP-hardness proof for Monotone-Planar-3-Sat. In Section 3, we prove that the reconfiguration of unit squares in a simple polygon is PSPACE-hard, by reducing from reconfiguration of Monotone-Planar-3-Sat. Finally, in Section 4, we show that the reconfiguration of unit disks in a

polygon with holes is PSPACE-hard, by a reduction from NCL. We conclude with a discussion in Section 5. The proofs of all statements marked by ($\star$) and the full proof of Theorem 1 are presented in the full version.

2 Reconfiguration of Monotone-Planar-3-Sat

Here we sketch a proof of the PSPACE-hardness where we reduce from NCL. We start by reviewing the configuration-to-configuration variant of the NCL problem. An instance is given by an oriented planar cubic graph G and an reorientation G' of G. Each edge is colored red or blue, such that each vertex is incident to either three blue edges, or one blue edge and two red edge. The orientations of G and G' need to be *feasible*, that is, the indegree of every vertex is at least two, where blue edges count with multiplicity 2. An NCL-instance is positive, if and only if there is a sequence of edge reversals that transforms G to G' such that each intermediate orientation is feasible.

For the reduction we start with an embedding of G and modify G such that it becomes the vertex-clause-incidence graph of a formula ϕ. Any satisfying assignment of the variables of ϕ encodes a feasible orientation of G and vice versa. Furthermore, if two feasible orientations of G differ by the orientation of a single edge, the corresponding satisfying variable assignments differ by two variable flips. Moreover, any variable flip corresponds to two feasible orientations of G that differ by at most one edge reorientation. A key idea is to replace all vertices of G with one or two clause nodes and every edge with a variable node as sketched in Figure 2. A variable x_e for an edge e is `true`, if and only if the current orientation of e matches the orientation of e in G.

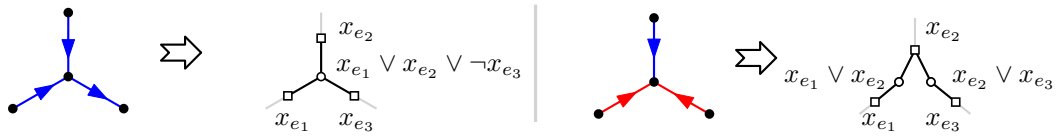


Figure 2 Mapping an NCL-Instance to an embedding of a planar 3-SAT formula. Variable nodes are drawn as squares, clause nodes are drawn as disks.

To prove that the embedding of the variable-clause incidence graph has the required properties we have to extend ϕ. In particular, we have to ensure that on every face in the planar embedding of the clause-variable incidence graph of ϕ positive and negative clauses alternate. See the full version for all details and a full proof of Theorem 1.

3 Unit squares in simple polygons

In this section, we prove Theorem 2 by reducing MONOTONE-PLANAR-3-SAT to the unlabeled reconfiguration of squares in simple grid polygons. We start with a MONOTONE-PLANAR-3-SAT formula ϕ and two satisfying assignments σ_0 and σ_1 of ϕ. In our reduction, we create a grid polygon P and use the assignments σ_0 and σ_1 to define two configurations c_0 and c_1 of 8×8 squares in P. We then show that there is a reconfiguration from c_0 to c_1 in P if and only if there is a reconfiguration from σ_0 to σ_1 in ϕ. The squares are axis-aligned and may move by translations only.

3.1 Reference centers

As a preliminary step we provide a helpful tool for verifying the correctness of the reduction. Let P be a grid polygon. A set $\mathcal{R}$ of k non-overlapping 4×4 squares, each contained in P, is called a set of *reference centers* for P if, for any packing of k 8×8 squares in P, each square in the packing contains one square in $\mathcal{R}$. Such a configuration of 8×8 squares in P is called *perfect*. For every $y \in \mathbb{Z}$, let b_y be an integer in $[0, 8)$. For $x, y \in \mathbb{Z}$, let $S_{xy} = [8x + b_y, 8x + b_y + 4] \times [8y, 8y + 4]$. Suppose that each S_{xy} is either fully contained in P or is interior-disjoint with P. Let $\mathcal{R}$ be the set of those S_{xy} that are fully contained in P. An example is shown in Figure 3. We will show that $\mathcal{R}$ is a set of reference centers for P.

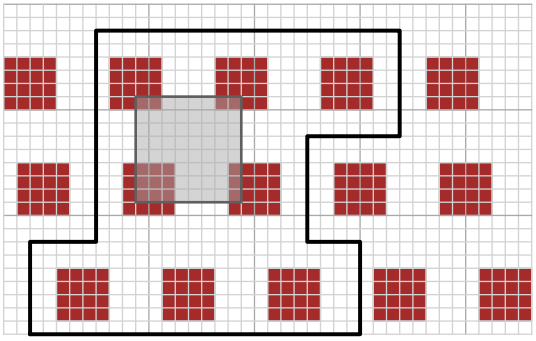


Figure 3 A grid polygon that has reference centers with $b_0 = 1, b_1 = 6, b_2 = 5$ and an 8×8 square intersecting four reference centers; the area of intersection is 16. (Note that in this example there exists no perfect configuration.)

▶ **Lemma 6.** ($\star$) *Let S be an 8×8 square placed in P. Then the area of the intersection of S with $\mathcal{R}$ is* 16.

▶ **Lemma 7.** ($\star$) *$\mathcal{R}$ is a set of reference centers for P.*

By Lemma 7, in any reconfiguration from one perfect configuration to another, each square contains the same reference center the whole time, i.e., the squares merely shift slightly sideways and up and down around the same reference center. In order to describe the mechanics of our intended reconfigurations, we only need to consider positions of squares that are extreme or centered in each direction with respect to their reference centers. This leaves us with the nine possible positions shown in Figure 4a. In Section 3.7, we prove that also arbitrary continuous reconfigurations in P correspond to reconfigurations of ϕ, but for understanding the key ideas of the reduction, it suffices to consider these nine discrete positions.

When the top (resp. bottom) edge of a square aligns with that of its reference center, we say that the square is *down* (resp. *up*). Otherwise, the midpoints of the reference center and the square have the same y-coordinate, in which case we say that the square is *vemid* (abbreviation of vertically in the middle). When the left (resp. right) edge of the square aligns with that of the reference center, the square is *right* (resp. *left*). Otherwise, the midpoints of the reference center and the square have the same x-coordinate, in which case we say that the square is *homid* (abbreviation of horizontally in the middle).

For most squares, several of the nine possible positions can be excluded, for instance due to being pushed by the boundary of P or another square. In our figures, we use a color scheme for the reference centers, to make the reader aware of what possible positions a square

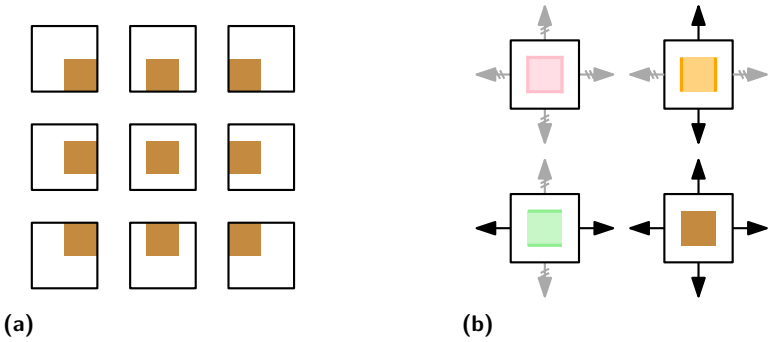


Figure 4 (a) The up to nine important placements of a square for a given reference center. (b) Examples of colored reference centers and how the squares containing them may move.

has. The colors have the following meanings: the square can not move (*pink*), only move vertically (*orange*), only move horizontally (*green*), move vertically and horizontally (*brown*). There might exist additional restrictions on the positions that are not indicated by the colors.

3.2 Schematics of the construction

On a big scale, our construction follows the ideas of Abrahamsen and Stade [1]. Therefore, we start by reviewing the high level ideas. Let ϕ be an instance of Monotone-Planar-3-Sat with variables $x_1, \ldots, x_k$ and consider the variable-clause-incidence graph G with a suitable embedding. We now describe schematically the construction of a grid polygon P. Each feasible solution to ϕ corresponds to a configuration of 8×8 squares in P, and there is a reconfiguration between two configurations if and only if there is a reconfiguration between the two solutions of ϕ.

In the first step, we introduce a *segment representation* that consists of horizontal segments for vertices and clauses, and of vertical segments for the edges of G, see in Figure 5b. The idea behind this representation is as follows: The horizontal segments will correspond to one or several rows of squares which transmit information horizontally. Similarly, the vertical segments will correspond to columns of squares that transmit information vertically. The endpoints of each segment should be on the outer face of the drawing, so that the smallest enclosing orthogonal polygon runs through all of the endpoints.

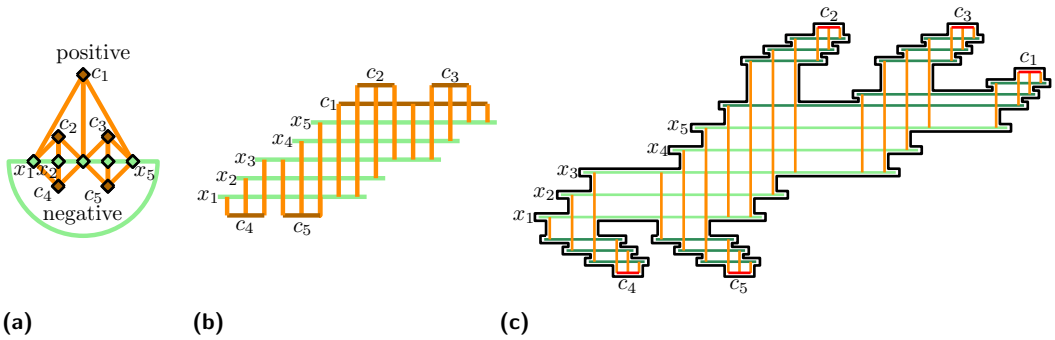


Figure 5 Schematic construction for transforming an instance of Monotone-Planar-3-Sat to a polygon. (a) a variable-clause incidence graph. (b) first step for constructing the segment representation of G (c) second step for constructing the segment representation of G: introducing the auxiliary variable rows and OR gadgets.

To construct the segment representation, we introduce a horizontal segment for each variable $x_1, \ldots, x_k$ in this order bottom-up, where the x-coordinates of the left and right endpoints are increasing, see Figure 5. 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 *primary* variable rows. The positive clauses are represented by horizontal segments above x_k and the negative clauses by horizontal segments below x_1. We say that a clause c' is *nested* in a clause c, if in G the clause node of c' lies inside a cycle defined by two incident edges of the node of c closed with a path between variable nodes. For instance, in Figure 5a, c_2 and c_3 are nested inside c_1. Note that the property of being nested defines a partial order. If a positive clause c_i is nested inside c_j, then we place the segment of c_i above that of c_j in the segment representation. If instead the clauses are negative, the segment of c_i is below that of c_j. Each edge $x_i c_j$ of G is realized by a vertical segment connecting the segments 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 segment 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, illustrated in Figure 5c, we replace each clause segment by three *auxiliary* variable segments, one for each of the variables connected to the clause. In the right side of the auxiliary segments, we connect them using vertical segments to an OR gadget, represented by a red horizontal segment. As sketched in the figure, we construct (guided by the partial nested-order of the positive and negative clauses) a polygon whose boundary approximately follows the outer face of the resulting representation, but the finer details will be given later. The following lemma was proved in [1].

▶ **Lemma 8.** *There is a segment representation as described, where the segments representing the OR gadgets are not crossed by any vertical segments, and all segment endpoints are incident to the outer face of the drawing.*

Algebraically, the introduction of an auxiliary variable row for a clause c_t corresponds to the following equivalence:

$$(x_i \vee x_j \vee x_k) \Leftrightarrow \Big(\exists y_{t,i}, y_{t,j}, y_{t,k} : (y_{t,i} \Rightarrow x_i) \wedge (y_{t,j} \Rightarrow x_j) \wedge (y_{t,k} \Rightarrow x_k) \wedge (y_{t,i} \vee y_{t,j} \vee y_{t,k})\Big)$$

Here, x_i, x_j, x_k are the primary (original) variables and $y_{t,i}, y_{t,j}, y_{t,k}$ are the auxiliary variables introduced for that particular clause. We remark that the formula is not monotone anymore, but the only non-monotone clauses are implications. Note that the equivalence also preserves reconfiguration: If, say, x_i changes to `false`, then we change $y_{t,i}$ to `false` before changing x_i. If x_i changes to `true`, then we change $y_{t,i}$ to `true` after x_i. For a negative clause c_t , we use an analogous equivalence:

$$(\neg x_i \vee \neg x_j \vee \neg x_k) \Leftrightarrow$$
$$\Big(\exists y_{t,i}, y_{t,j}, y_{t,k} : (\neg y_{t,i} \Rightarrow \neg x_i) \wedge (\neg y_{t,j} \Rightarrow \neg x_j) \wedge (\neg y_{t,k} \Rightarrow \neg x_k) \wedge (\neg y_{t,i} \vee \neg y_{t,j} \vee \neg y_{t,k})\Big)$$

3.3 Components and gadgets

We informally distinguish between *gadgets* and *components* in our constructions, where a gadget is given by a set of segments on 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.

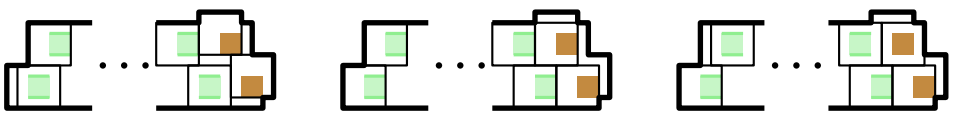
On a very high level the truth assignment of a variable of ϕ is encoded by a configuration of a so-called *variable component*; see Figure 6. The variable component (used to represent primary and auxiliary variables) has essentially three possible states, namely *plus*, *minus* and *transitional*. The first two correspond to the values `true` and `false` of a binary variable, while the truth value of the transitional state is defined to be the `true`/`false` value of the previous plus/minus state. In order to split the information represented in a variable component of a primary variable and transfer it to a variable component of an auxiliary variable we use PUSH gadgets. In particular, these gadgets are used to model the implication-clauses and they come in two versions: PUSH-UP-IF-MINUS and PUSH-DOWN-IF-PLUS. The former is used to model implications of the form $y \Rightarrow x$, the later for $\neg y \Rightarrow \neg x$. The gadgets use a vertically moving stack of square rows with growing width (called pyramid) to transmit information from primary to auxiliary variables, see for example Figure 8. For a PUSH-UP-IF-MINUS gadget representing an implication $y \Rightarrow x$ a pyramid that is raised enforces that the auxiliary variable component of y is set to minus, if the variable component of x is set to minus. Similarly, a PUSH-DOWN-IF-PLUS gadget modeling $\neg y \Rightarrow \neg x$ has the functionality that a pyramid that is lowered enforces that the auxiliary variable component of y is set to plus, if the variable component of x is set to plus. We will make sure that the vertically moving pyramid structure do not collide with the vertically operating variable components they "cross". To only allow satisfying assignments for every clause we place an *OR gadget* that is connected with the corresponding auxiliary variable components. We reuse the ideas for the PUSH Gadgets. For a clause $(y_i \vee y_j \vee y_k)$ the auxiliary variables will push a pyramid up into the clause gadget if the corresponding auxiliary variable is set to minus. The OR gadget has only room for two of these pyramids to push up. The clauses $(\neg y_i \vee \neg y_j \vee \neg y_k)$ are handled symmetrically. Here, an auxiliary variable set to plus would result in a pyramid pushing down and the OR gadget can only incorporate two lowered pyramids.

We continue with describing the gadgets and components in detail.

3.4 The variable component

The variable component has two *main* rows of squares and an arbitrary number of PUSH gadgets, ensuring dependency between the primary and auxiliary variables and between the auxiliary variables and the OR gadgets. The PUSH gadgets require the introduction of *helper rows* padded along the main rows, above the top row or below the bottom row. The construction of PUSH gadgets and helper rows will be explained in later sections.

The construction of the main rows of a variable component is shown in Figure 6. A perfect configuration of the component has three possible states, as defined by the horizontal position of the two rightmost squares (covering the brown reference centers). Note that none of the two squares can push right, and if one is homid, the other needs to be right.



(a) plus state **(b)** transitional state **(c)** minus state

Figure 6 A variable component and its three states, defined by the horizontal positions of the two rightmost squares with brown reference centers.

If the bottom is in the homid position, the state is *plus*. If the top is in the homid position, the state is *minus*. Otherwise, the state is *transitional*.

Now we describe what happens when a block of raised or lowered squares "crosses" a variable component. As discussed before, pyramids made of such blocks are used to create dependencies between two variable components, but it is important that they can cross other variable components without interacting with them.

When a stack of raised squares crosses a row of squares with a different horizontal alignment, the stack of raised squares will grow in width. Hence, the stack will be pyramid shaped if all rows have a different alignment than the neighboring rows. It is important that we know the width of a pyramid when it reaches the target variable component where it will end. Thus, we need to know exactly how many times the width is increased and which squares are raised in the top row. To this end, and in order to create a necessary amount of vertical spacing, we introduce *static* rows in between all neighboring pairs of variable components. A static row consists of squares in the beginning and in the end, which cannot move, and some squares in the middle, which can move vertically up and down, see Figure 7. So in particular, all squares in a static row have a fixed horizontal position.

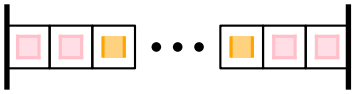


Figure 7 A static row. The squares have fixed horizontal positions, but some of the middle squares will be able to move up or down because of pyramids.

We ensure that the fixed horizontal position is never aligned with the rows of the variable component. Furthermore, the top row of a variable component is never aligned with the bottom row, see Figure 8. Hence, when a pyramid crosses a variable component, the specific horizontal positions of the rows do not affect which squares are raised/lowered on the other side of the variable. This is expressed in the following lemma.

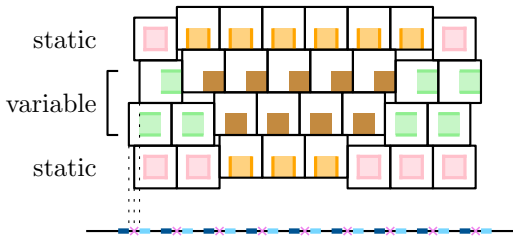


Figure 8 A pyramid crossing a variable component. The scale at the bottom shows the intervals of the possible x-coordinates of the left corners of the upper and lower squares in the variable rows (light and dark blue, respectively) and those of the squares in the static rows (pink). Since these three groups of intervals are pairwise disjoint, the pyramid grows in width by one square per row. Hence, the width of the pyramid always grows by three squares when crossing a variable component, regardless of the state of the variable component.

▶ **Lemma 9.** ($\star$) *Consider a perfect configuration. If a block of consecutive squares in a static row below (above) a variable component are up (down), then the corresponding squares in the static row above (below) the variable component must also be up (down) as well as two extra squares on the right (left) and one on the left (right).*

Recall that there may be helper rows above or below the main rows of a variable component, which will be explained in more detail in Section 3.5. Conceivably, these could be unaligned with the main rows, which would cause a pyramid to grow more in width when crossing the variable. However, if they are aligned (which is our "intended behavior" of the helper rows), then the width will not grow beyond what is expressed by Lemma 9.

3.5 PUSH gadgets and helper rows

Figures 9 and 10 show the PUSH-UP-IF-MINUS and PUSH-DOWN-IF-PLUS gadgets, respectively. The gadgets are mirror images of each other by a horizontal axis. In order to make PUSH gadgets, we need a square that is pushed up or down only when it is left. To this end, we introduce *helper rows*, which are extra rows above and below the main variable rows. Consider the PUSH-UP-IF-MINUS gadget, shown in Figure 9. If the squares of the lower main row are left, the squares of the helper row will be left as well, because the square a presses down b, which thus pushes c to the left. Therefore the square d will be up, and a pyramid of raised squares is created.

Figure 9 The plus and minus states of PUSH-UP-IF-MINUS. In minus, square d is pushed up.

The PUSH-DOWN-IF-PLUS gadget works by analogous mechanics. If the squares in the upper main row are left, two squares of the lower row are pushed down, see Figure 10.

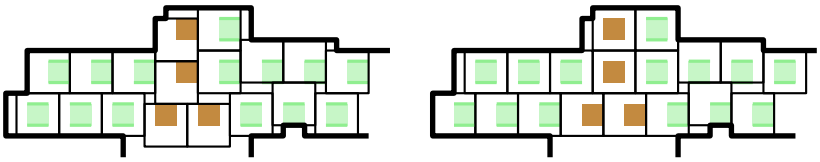


Figure 10 The plus and minus states of PUSH-DOWN-IF-PLUS. In plus, the squares with brown reference centers are pushed down.

All helper rows are created at the right end of the variable gadget and end at a PUSH gadget. Figure 11 shows schematically where the helper rows are placed, and Figure 12 shows a detailed example with multiple helper rows. Note that with multiple helper rows, the same principle is used as with a single helper row in order to ensure that when the lower main row is left, then all helper rows below are also left, so they all create pyramids of raised squares.

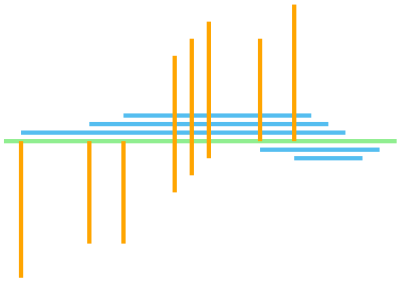


Figure 11 Schematic showing the horizontal span of the helper rows (blue) along a principal variable component. The variable connects to three auxiliary variables below and two above, so there are three helper rows above and two below. In between the two groups, pyramids from other principal variable components can cross, indicated as three orange vertical segments in the middle.

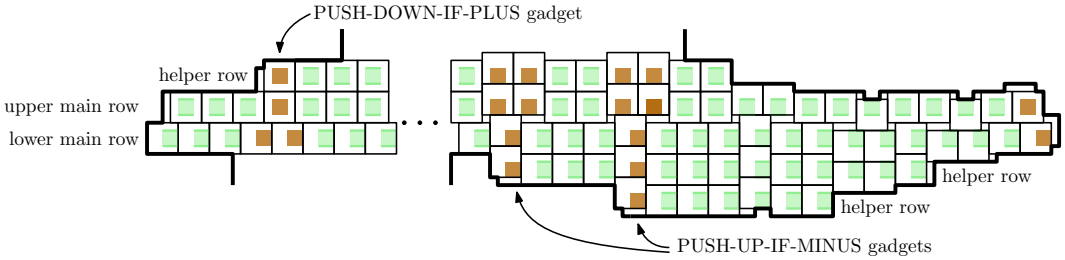


Figure 12 A variable component in the minus position with two PUSH-UP-IF-MINUS gadgets and one PUSH-DOWN-IF-PLUS gadget. Thus, there are two helper rows at the bottom and one at the top. Other pyramids may cross the variable in the region indicated by the dots.

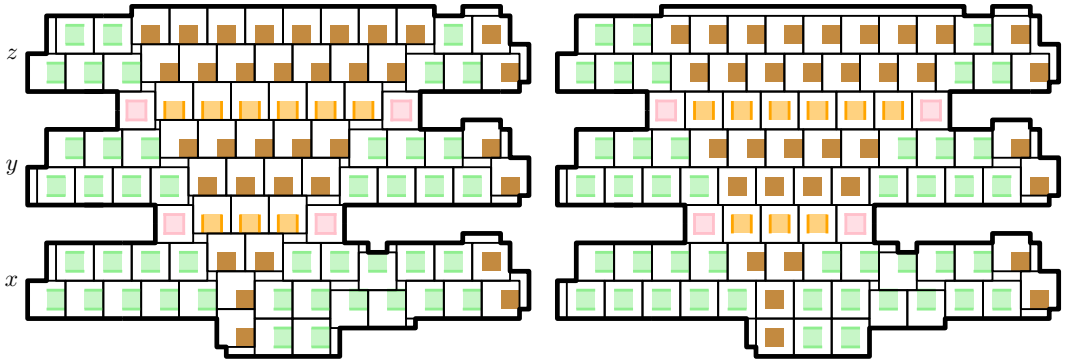


Figure 13 There is a PUSH-UP-IF-MINUS gadget on the bottom variable x, which connects x to the top variable z. When x is minus, a pyramid of raised squares is created. The boundary of the polygon at z forces z to also be minus. The state of y has no influence. When x is plus, the pyramid does not have to raise, so z is not restricted.

We now describe how a pyramid ends at an auxiliary variable component; see Figure 13 for a concrete example to support the description. When we make a PUSH-UP-IF-MINUS gadget from a primary variable x to an auxiliary variable y, we create a pocket in the boundary of P along the upper primary row of y to enable the squares of the pyramid to raise. We place the pocket so that the squares can only raise if they are homid (and if the pyramid has grown in width by exactly three squares every time it crossed another variable component). Thus, the rightmost square of the top row of y is also homid, so y must be minus. We have thus made the implication $y \Rightarrow x$.

In minus, pyramids are created to the positive auxiliary variables, which in turn create pyramids to the positive clauses. Likewise, in plus, pyramids are created to the negative clauses. In transitional, pyramids are created to all clauses. As we will see in Section 3.6, there will only be room for two pyramids for each clause. Hence, each clause ensures that one of the variables is in the plus/minus state that represents the `true`/`false` value that satisfies the clause. This gives the following lemma.

▶ **Lemma 10.** *In every perfect configuration, the following holds. If a variable is not plus (minus), a pyramid of raised squares appears from any PUSH-UP-IF-MINUS (PUSH-DOWN-IF-PLUS) gadget, and if the pyramid is received by an auxiliary variable, that variable must be minus (plus). As a consequence, we can realize the implications $y \Rightarrow x$ and $\neg y \Rightarrow \neg x$, i.e., ensure that the values of variables encoded by the configuration satisfy the implications.*

3.6 OR gadget

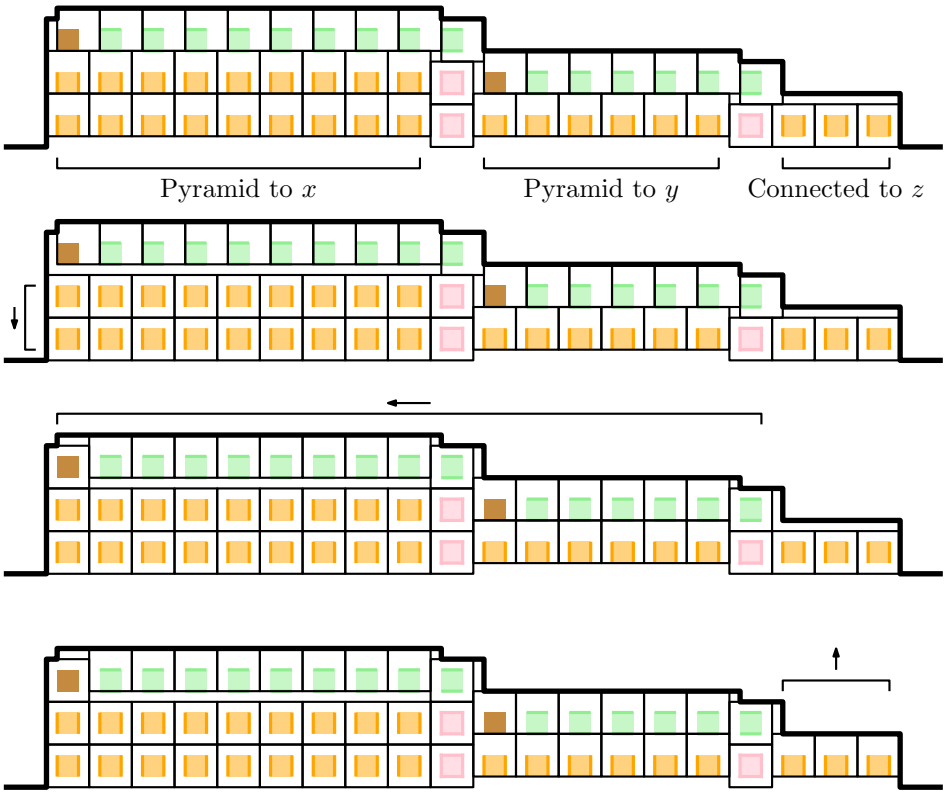


Figure 14 The OR gadget for a clause $x \vee y \vee z$. Initially, there is a pyramid from x and y. The pyramid from x is removed, and it is shown how the top squares reconfigure so that a pyramid from z can raise.

Figure 14 shows the positive OR gadget. The gadget is connected to three auxiliary variable components. Using PUSH-UP-IF-MINUS gadgets, pyramids can raise from the variables and end in the OR gadget. The gadget allows for two pyramids to be raised, but not all three at once. Hence, one variable must be plus. When at most one pyramid is raised, the squares in the top of the gadget can reconfigure so that there is room for any of the other two pyramids to raise. An example is shown in the figure – the reader can easily check that it is likewise possible in the other five cases where one pyramid is raised and we need to make room for one of the other two to raise as well. This corresponds to a reconfiguration of the formula ϕ: If $x \vee y \vee z$ is a clause and one of the variables, say z, changes from positive to negative, then x or y must have been positive before z changes (otherwise the clause would not be satisfied after z changing). Hence, there was only one pyramid, so the squares in the gadget can reconfigure to accommodate raising the pyramid from z.

Figure 15 shows how the OR gadget is connected to the variable components. The negative OR gadget is obtained by reflection along a horizontal axis.

▶ **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.*

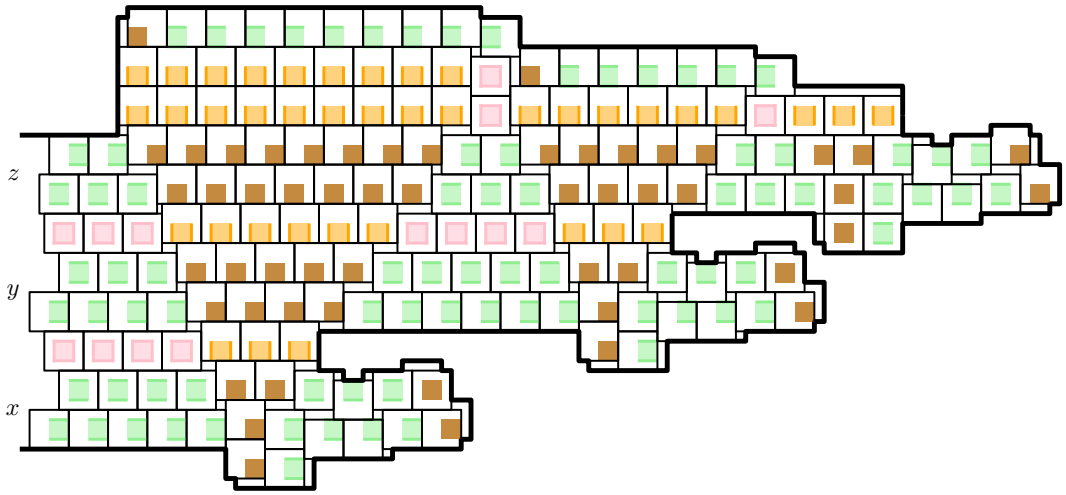


Figure 15 The positive OR gadget connected to the auxiliary variable components for x, y and z. Here, x and y are minus, so they create pyramids, but z is plus, which allows for a reconfiguration.

3.7 Verification

Recall that we start with a Monotone-Planar-3-Sat formula ϕ and two satisfying assignments σ_0 and σ_1 of ϕ. In our reduction, we create a polygon P as described. We define start and target configurations c_0 and c_1 according to σ_0 and σ_1: We place squares in all primary and auxiliary variable components in c_i in the plus/minus state corresponding to the value of the variable in σ_i. By construction there is room for the pyramids created by PUSH gadgets. We now need to verify that there is a reconfiguration between σ_0 and σ_1 in ϕ if and only if there is one between c_0 and c_1 in P. This is expressed by the following two lemmas.

▶ **Lemma 12.** ($\star$) *If ϕ has a reconfiguration between two satisfying assignments σ_0 and σ_1, then there is also a reconfiguration between the corresponding start and target configurations c_0 and c_1 in P.*

▶ **Lemma 13.** ($\star$) *If there is a reconfiguration in P from the start configuration c_0 to the target c_1 in P, then there is a reconfiguration in the formula ϕ from the satisfying assignment σ_0 to σ_1.*

Finally, we prove the containment in PSPACE.

▶ **Lemma 14.** ($\star$) *Unlabeled reconfiguration of 8×8 squares in simple grid polygons is contained in* PSPACE.

Theorem 2 result now follows from Theorem 1 and Lemmas 12–14.

As we only consider perfect configurations in our reduction, by Lemma 7 each square always contains its reference center. Thus, we can easily label each square and obtain hardness for the labeled variant, i.e., Corollary 3.

4 Unit disks in polygons with holes

We now present the ideas for showing PSPACE-hard of reconfiguration of unit disks in a polygon possibly with holes. As a first step, we prove it for an *arc-gon*, that is, a shape whose boundary consists of straight line segments and circular arcs.

▶ **Proposition 15.** *Reconfiguration of unlabeled unit disk robots in an arc-gon is* PSPACE*-hard.*

Proof-Sketch. As in Section 2, we reduce from the configuration-to-configuration variant of Nondeterministic Constrained Logic (NCL) [11, 12, 13]. From a given NCL instance, specified by two orientations of a plane cubic graph G and a reorientation G', we construct an arc-gon and two configurations of unit disks such that the NCL instance is a yes-instance if and only of the two configurations of unit disks can be reconfigured into one another.

Similar to [4], we lay out G using five types of *tiles* as illustrated in Figure 16. Here we ensure that each edge has a at least one straight tile.

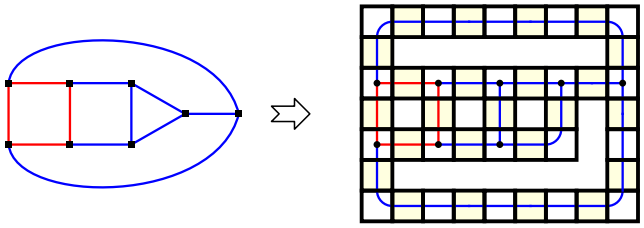
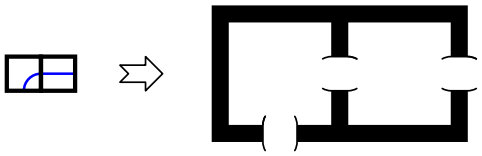
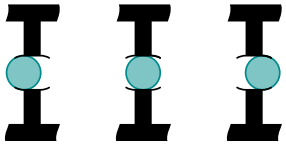


Figure 16 A layout of an NCL instance using a limited set of tiles.

Each tile is then realized as a *room*, where small "hooks" limit the movement of the disks in the opening of the room which is called the *door* of the room, see Figures 17a and 17b.



(a) Each tiles becomes a room, and an edge through tile sides implies a door.



(b) Each door has a (green) *door disk* that must stay within the door opening.

Figure 17

The interiors of the rooms mimic the behavior of the vertices and edges of G during a reconfiguration, see Figure 18. In particular, it is essential to guarantee that an edge can only have one orientation at a time; we implement this in our *straight* tile gadget where we guarantee that at least one door disks is "pushed out" and implies that the AND and OR tiles behave as required, see Figure 19. ◀

Then, we turn the arc-gon constructed in the proof of Proposition 15 into a polygon with holes and thereby complete the proof of Theorem 4. Observing that no two disks may swap within the polygon implies PSPACE-hardness for the labeled variant, i.e., Corollary 5.

5 Discussion

We proved that reconfiguration of Monotone-Planar-3-Sat is PSPACE-hard, and thereby complemented a sequence of recent results establishing the relationships between 3-Sat variants and their reconfiguration counterparts. This result was a stepping stone towards showing PSPACE-hardness of reconfiguration of 8×8-squares in a simple grid polygon. While the reduction reuses ideas from Abrahamsen and Stade [1], we also needed several new ideas. Most importantly, the reduction to the packing problem uses very rigid configurations whereas our setting inherently requires to allow for some movements. While the computational complexity of reconfiguring $k \times k$-squares in simple grid polygons is trivial for $k = 1$, the cases $k \in \{2, \ldots, 7\}$ remain an interesting open problem.

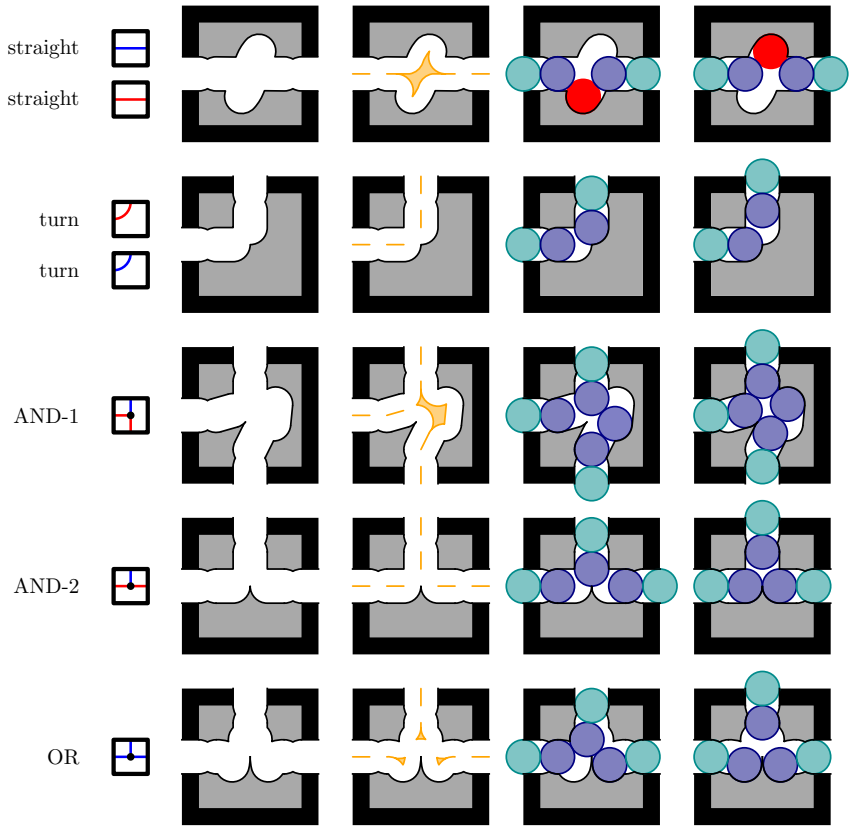


Figure 18 Geometric realizations of the five tile types. Marked in orange is the *dilation* of the shapes by 1; these are possible locations of disk centers. On the right two different valid configurations of disks with different extremal positions of the door disks.

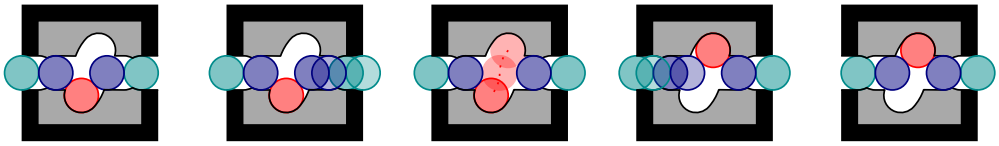


Figure 19 A sequence of moves that imitates the reorientation of an edge.

We have also shown that reconfiguration of unit disks in a polygon with holes is PSPACE-hard. A natural next question is unit disks in a simple polygon. At a first glance, it seems difficult to extend our insights in the square setting to this case, because it is troublesome to propagate movement independently in multiple directions in a large region filled with disks. Therefore, we expect that a PSPACE-hardness proof, if it exists, will likely require different techniques.

References

1 Mikkel Abrahamsen and Jack Stade. Hardness of packing, covering and partitioning simple polygons with unit squares. In *Annual Symposium on Foundations of Computer Science (FOCS 2024)*, 2024. doi:10.48550/arXiv.2404.09835.

2 Aviv Adler, Mark de Berg, Dan Halperin, and Kiril Solovey. Efficient multi-robot motion planning for unlabeled discs in simple polygons. *IEEE Trans Autom. Sci. Eng.*, 12(4):1309–1317, 2015. doi:10.1109/TASE.2015.2470096.

3 Bahareh Banyassady, Mark de Berg, Karl Bringmann, Kevin Buchin, Henning Fernau, Dan Halperin, Irina Kostitsyna, Yoshio Okamoto, and Stijn Slot. Unlabeled multi-robot motion planning with tighter separation bounds. In *International Symposium on Computational Geometry, SoCG 2022*, volume 224 of *LIPIcs*, pages 12:1–12:16, 2022. doi:10.4230/LIPICS.SOCG.2022.12.

4 Thomas Brocken, G. Wessel van der Heijden, Irina Kostitsyna, Lloyd E. Lo-Wong, and Remco J. A. Surtel. Multi-robot motion planning of k-colored discs is PSPACE-hard. In *10th International Conference on Fun with Algorithms, FUN 2021*, volume 157 of *LIPIcs*, pages 15:1–15:16, 2021. doi:10.4230/LIPICS.FUN.2021.15.

5 Josh Brunner, Lily Chung, Erik D. Demaine, Dylan H. Hendrickson, Adam Hesterberg, Adam Suhl, and Avi Zeff. 1×1 rush hour with fixed blocks is PSPACE-complete. In *10th International Conference on Fun with Algorithms, FUN 2021*, volume 157 of *LIPIcs*, pages 7:1–7:14, 2021. doi:10.4230/LIPICS.FUN.2021.7.

6 Jean Cardinal, Erik D. Demaine, David Eppstein, Robert A. Hearn, and Andrew Winslow. Reconfiguration of satisfying assignments and subset sums: Easy to find, hard to connect. *Theoretical Computer Science*, 806:332–343, 2020. doi:10.1016/j.tcs.2019.05.028.

7 Mark de Berg and Amirali Khosravi. Optimal binary space partitions for segments in the plane. *Int. J. Comput. Geom. Appl.*, 22(3):187–206, 2012. doi:10.1142/S0218195912500045.

8 Robert J. Fowler, Mike Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Inf. Process. Lett.*, 12(3):133–137, 1981. doi:10.1016/0020-0190(81)90111-3.

9 M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Mathematical Sciences Series. Freeman, 1979. URL: https://books.google.nl/books?id=fjxGAQAAIAAJ.

10 Parikshit Gopalan, Phokion G. Kolaitis, Elitza Maneva, and Christos H. Papadimitriou. The connectivity of boolean satisfiability: Computational and structural dichotomies. *SIAM Journal on Computing*, 38(6):2330–2355, 2009. doi:10.1137/07070440X.

11 Robert A. Hearn and Erik D. Demaine. The nondeterministic constraint logic model of computation: Reductions and applications. In *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002*, volume 2380 of *LNCS*, pages 401–413. Springer, 2002. doi:10.1007/3-540-45465-9_35.

12 Robert A. Hearn and Erik D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theor. Comput. Sci.*, 343(1-2):72–96, 2005. doi:10.1016/J.TCS.2005.05.008.

13 Robert A. Hearn and Erik D. Demaine. *Games, puzzles, and computation*. CRC Press, 2009.

14 J.E. Hopcroft, J.T. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects; PSPACE-hardness of the "warehouseman's problem". *The International Journal of Robotics Research*, 3(4):76–88, 1984. doi:10.1177/027836498400300405.

15 Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972. doi:10.1007/978-1-4684-2001-2_9.

16 D. Kornhauser, G. Miller, and P. Spirakis. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *Symposium on Foundations of Computer Science (FOCS)*, pages 241–250, 1984. doi:10.1109/SFCS.1984.715921.

17 David Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982. doi:10.1137/0211025.

18 Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 216–226, 1978. doi:10.1145/800133.804350.

19 Kiril Solovey and Dan Halperin. On the hardness of unlabeled multi-robot motion planning. *Int. J. Robotics Res.*, 35(14):1750–1759, 2016. doi:10.1177/0278364916672311.

20 Kiril Solovey, Jingjin Yu, Or Zamir, and Dan Halperin. Motion planning for unlabeled discs with optimality guarantees. In *Robotics: Science and Systems XI, Sapienza University of Rome*, 2015. doi:10.15607/RSS.2015.XI.011.

21 Paul G. Spirakis and Chee-Keng Yap. Strong NP-hardness of moving many discs. *Inf. Process. Lett.*, 19(1):55–59, 1984. doi:10.1016/0020-0190(84)90130-3.