

Compacting Squares: Input-Sensitive In-Place Reconfiguration of Sliding Squares

Hugo A. Akitaya  



University of Massachusetts Lowell, MA, USA

Erik D. Demaine  

Massachusetts Institute of Technology,
Cambridge, MA, USA

Matias Korman 

Siemens Electronic Design Automation,
Wilsonville, OR, USA

Irina Kostitsyna  

Eindhoven University of Technology,
The Netherlands

Irene Parada  

Technical University of Denmark, Lyngby,
Denmark

Willem Sonke  

Eindhoven University of Technology,
The Netherlands

Bettina Speckmann  

Eindhoven University of Technology,
The Netherlands

Ryuhei Uehara  

Japan Advanced Institute of Science and
Technology, Ishikawa, Japan

Jules Wulms  

Technische Universität Wien, Austria

Abstract

Edge-connected configurations of square modules, which can reconfigure through so-called sliding moves, are a well-established theoretical model for modular robots in two dimensions. Dumitrescu and Pach [Graphs and Combinatorics, 2006] proved that it is always possible to reconfigure one edge-connected configuration of n squares into any other using at most $O(n^2)$ sliding moves, while keeping the configuration connected at all times.

For certain pairs of configurations, reconfiguration may require $\Omega(n^2)$ sliding moves. However, significantly fewer moves may be sufficient. We prove that it is NP-hard to minimize the number of sliding moves for a given pair of edge-connected configurations. On the positive side we present Gather&Compact, an input-sensitive in-place algorithm that requires only $O(\bar{P}n)$ sliding moves to transform one configuration into the other, where \bar{P} is the maximum perimeter of the two bounding boxes. The squares move within the bounding boxes only, with the exception of at most one square at a time which may move through the positions adjacent to the bounding boxes. The $O(\bar{P}n)$ bound never exceeds $O(n^2)$, and is optimal (up to constant factors) among all bounds parameterized by just n and \bar{P} . Our algorithm is built on the basic principle that well-connected components of modular robots can be transformed efficiently. Hence we iteratively increase the connectivity within a configuration, to finally arrive at a single solid xy -monotone component.

We implemented Gather&Compact and compared it experimentally to the in-place modification by Moreno and Sacristán [EuroCG 2020] of the Dumitrescu and Pach algorithm (MSDP). Our experiments show that Gather&Compact consistently outperforms MSDP by a significant margin, on all types of square configurations.

2012 ACM Subject Classification Theory of computation \rightarrow Computational geometry

Keywords and phrases Sliding cubes, Reconfiguration, Modular robots, NP-hardness

Digital Object Identifier 10.4230/LIPIcs.SWAT.2022.4

Related Version *Full Version*: <https://arxiv.org/abs/2105.07997>

Supplementary Material *Software*: <https://alga.win.tue.nl/software/compacting-squares/>

Funding Irene Parada was supported by Independent Research Fund Denmark grant 2020-2023 (9131-00044B) “Dynamic Network Analysis” and Jules Wulms was supported partially by the Austrian Science Fund (FWF) under grant P31119 and partially by the Vienna Science and Technology Fund (WWTF) under grant ICT19-035.



© Hugo A. Akitaya, Erik D. Demaine, Matias Korman, Irina Kostitsyna, Irene Parada, Willem Sonke, Bettina Speckmann, Ryuhei Uehara, and Jules Wulms;
licensed under Creative Commons License CC-BY 4.0

18th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2022).

Editors: Artur Czumaj and Qin Xin; Article No. 4; pp. 4:1–4:19



Leibniz International Proceedings in Informatics

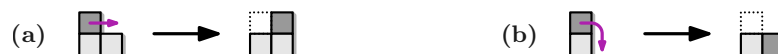
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Acknowledgements Parts of this work were initiated at the 5th Workshop on Applied Geometric Algorithms (AGA 2020) and at the 2nd Virtual Workshop on Computational Geometry. We thank all participants for discussions and an inspiring and productive atmosphere. We thank Fabian Klute for discussions on the computational experiments.

1 Introduction

Self-reconfigurable modular robots [20] promise adaptive, robust, scalable, and cheap solutions in a wide range of technological areas, from aerospace engineering to medicine. Modular robots are envisioned to consist of identical building blocks arranged in a lattice and are intended to be highly versatile, due to their ability to reconfigure into arbitrary forms. An actual realization of this vision depends on fast and reliable reconfiguration algorithms, which hence have become an area of growing interest.

One of the best-studied paradigms of modular robots is the *sliding cube model* [11]. In this model, a robot *configuration* is a face-connected set of cubic modules on the cubic grid. The cubes can perform two types of (*sliding*) moves, illustrated in two dimensions in Figure 1.



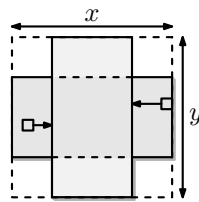
■ **Figure 1** Moves admitted by the sliding cube model: (a) slide, (b) convex transition.

First, a module can *slide* along two face-adjacent cubes to reach a face-adjacent empty grid cell. Second, a module m can make a *convex transition* around a module m' to end in an edge-adjacent empty grid cell. For this second move to be feasible, also the grid cell (not occupied by m') face-adjacent to both the starting and the ending positions must be empty. Moves need to maintain connectivity, that is, a cube c is movable only if the configuration without c would still be face-connected. There are several prototypes of modular robots that realize the sliding cube model in 2D [6, 9, 12]. Units of multiple other prototypes, including expandable and contractible units [17, 18] as well as large classes of modular robots [4, 16], can be arranged into cubic *meta-modules* consisting of several units such that the meta-module can perform slide and convex transition moves. Thus, algorithmic solutions in the sliding cube model can be applied to modular robot systems realizing other models.

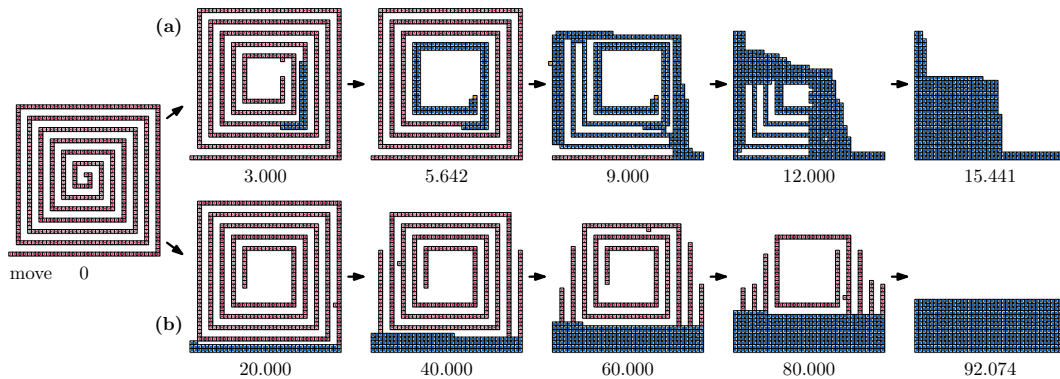
Another well-studied model for modular robots is the *pivoting cube model* [7, 19]. This model strengthens the free-space requirements for each move and has also been realized by some existing prototypes. Previous work [2] showed very recently that in the pivoting cube model in two dimensions it is PSPACE-hard to decide whether it is possible at all to reconfigure one configuration into the other. However, if one allows six auxiliary squares in addition to the input configuration, then there is a worst-case optimal reconfiguration algorithm [1]. Other models for squares relax the face-connectivity condition [8], restrict or enlarge the set of allowed moves [13], or relax the free-space requirements [5].

In this paper we study the reconfiguration problem for the sliding cube model in two dimensions (the *sliding square model*). Given two configurations of n unlabeled squares (each describing the relative positions of squares), we compute a short sequence of moves that transforms one configuration into the other, while preserving edge-connectivity at all times. Dumitrescu and Pach [10] described an algorithm which transforms any two configurations of n squares into each other using $O(n^2)$ moves. This bound is worst-case optimal: there are pairs of configurations (a horizontal and a vertical line) which require $\Omega(n^2)$ moves for any transformation. However, for other pairs of configurations, significantly fewer moves suffice.

We show in Section 2 that it is NP-hard to minimize the number of moves for a given pair of edge-connected configurations. Due to the $O(n^2)$ upper bound on the number of moves, the corresponding decision problem is NP-complete. In Section 3, we present an input-sensitive and in-place algorithm for self-reconfiguration, based on the “compact-and-deploy” approach. Using the basic principle that well-connected components of modular robots can be transformed efficiently, our algorithm iteratively increases the connectivity within a configuration, to arrive at a single solid xy -monotone component, before deploying it into the target configuration. Hence, our algorithm builds the target configuration in such a way that the lower left corner of the bounding boxes of both configurations are aligned. Our algorithm is *input-sensitive*: it requires only $O(\bar{P}n)$ moves to transform one configuration into the other, where \bar{P} is the maximum perimeter of the two bounding boxes. Our algorithm is also *in-place*: only one square at a time is allowed to move outside the respective bounding box, and then only through cells vertex-adjacent to the bounding box.



Lower bound. Our $O(\bar{P}n)$ bound is optimal (up to constant factors) among all bounds parameterized by just n and \bar{P} . Given \bar{P} and n , consider a source and target configuration that each consist of a rectangle filled with squares, with dimensions $x \times y/2$ and $x/2 \times y$, respectively, such that $n = x/2 \cdot y = x \cdot y/2$ and $\bar{P} = \Theta(x + y)$. This is satisfied by choosing $x = \Theta(\bar{P})$ and $y = \Theta(n/\bar{P})$. Without loss of generality assume $x > y$, so that $\bar{P} = \Theta(x)$. Given the source configuration, the target configuration may be built anywhere in the grid, and for such a given *target (configuration) position*, we call all grid cells that should be filled with squares *target cells*. For any target position, every source square s that is not in a target cell requires at least as many moves as the L_∞ -distance d_s (along the grid) between s and its closest target cell. Given the set S of source squares not on target cells, the configuration then requires at least $\sum_{s \in S} d_s$ moves. For any target position, S contains at least an $x/4 \times y$



■ **Figure 2** A spiral configuration in a 40×40 bounding box. (a) Gather&Compact: gathering done after 5.642 moves; total 15.441 moves. (b) MSDP [10, 15]: total 92.074 moves. Video: ■ <https://tinyurl.com/algaspiral>.

rectangle R of squares, either to the left or to the right of the vertical strip occupied by the target configuration. Each robot r in the i -th column of R has $d_r \geq i$, hence the total required movement is at least $\sum_{i=1}^{x/4} iy = \Theta((x/4)^2y) = \Theta(xn) = \Theta(Pn)$.

Comparison with Dumitrescu and Pach. The algorithm by Dumitrescu and Pach [10] constructs a canonical shape from both input configurations. In the original paper this canonical shape is a strip that extends to the right of a rightmost square and hence, necessarily, their algorithm always requires $\Omega(n^2)$ moves. Moreno and Sacristán [14, 15] modify the algorithm of Dumitrescu and Pach to be in-place; their canonical shape is a rectangle within the bounding box of the input. For either type of canonical shape their algorithm roughly proceeds as follows. If there is a square which is a leaf in the edge-adjacency graph, then the algorithm attempts to move this square along the boundary towards the canonical configuration. If this leaf square “gets stuck” on the way, and hence increases its connectivity, or if there is no leaf in the first place, then the algorithm identifies a 2-connected square on the outside of the configuration which it can move towards the canonical configuration. Hence, if configurations are tree-like (such as the spiral in Figure 2), then each square moves along all remaining squares, for a total of $\Omega(n^2)$ moves (see Figure 2 bottom row). However, the width and the height of this spiral configuration is $O(\sqrt{n})$. Our algorithm gathers $\Theta(\sqrt{n})$ squares from the end of the spiral and then compacts in a total of $O(n\sqrt{n})$ moves.

The in-place modification by Moreno and Sacristán of Dumitrescu and Pach (henceforth MSDP) has the potential to use fewer than $\Theta(n^2)$ moves in practice. In Section 4 we compare our Gather&Compact to MSDP experimentally; Gather&Compact consistently outperforms MSDP by a significant margin, on all types of square configurations.

2 Hardness of optimal reconfiguration

In this section we sketch the proof of Theorem 1, details can be found in the full version [3].

► **Theorem 1.** *Let \mathcal{C} and \mathcal{C}' be two configurations of n squares each and let k be a positive integer. It is NP-complete to determine whether we can transform \mathcal{C} into \mathcal{C}' using at most k moves while maintaining edge-connectivity at all times.*

We provide a reduction from PLANAR MONOTONE 3SAT. In particular, we start from a rectilinear drawing of a planar monotone 3SAT instance \mathcal{C} with l variables and m clauses (see Figure 3). We create a problem instance of the reconfiguration problem whose size is polynomial in l and m ; we show that \mathcal{C} can be satisfied if and only if the corresponding reconfiguration problem can be solved in at most $66m + 24l$ moves.

We replace each variable with a variable gadget, highlighted by an orange-shaded area in Figure 4 and summarized in Figure 5a. Consecutive variable gadgets are connected by a horizontal line of squares forming a central path of cycles through all variable gadgets (pink squares). More precisely, the gadget associated with variable x_i has k_i O-shaped cycles in the path of cycles, where k_i is the number of times that the variable x_i appears in \mathcal{C} . Each O-shaped cycle has two *prongs* (yellow squares). The spacing between gadgets is large enough for different variable gadgets not to interact in an optimal reconfiguration process.

The source and target configurations in the variable gadgets are very similar. The only difference is that the positions marked with \times in Figure 5a must be emptied and the positions marked with \circ must be occupied. Using an earth movers argument one can argue that a square must be transferred from the right of the gadget to the left, and the minimum number of moves required to do so is the horizontal distance between the positions, in this case

$20k_i + 20$. There are two paths that achieve this bound, namely a path along the top (shown in Figure 5a) or one along the bottom. These correspond to setting the variable to **true** or **false**, respectively. The remaining required changes need four additional moves.

► **Lemma 2.** *At least $20k_i + 24$ moves are necessary to reconfigure the variable gadget x_i . Moreover, this number of moves can be achieved only by moving one of the right \times squares to the left \circ position at the same height, following the path shown in Figure 4 or the equivalent path connecting the other \times and \circ pair along the bottom.*

Lemma 2 shows that to reconfigure using as few moves as possible we must transfer one of the two \times squares on the right to a \circ position on the left. During the process, the \times square creates cycles involving the central path of cycles and either all upper or lower prongs.

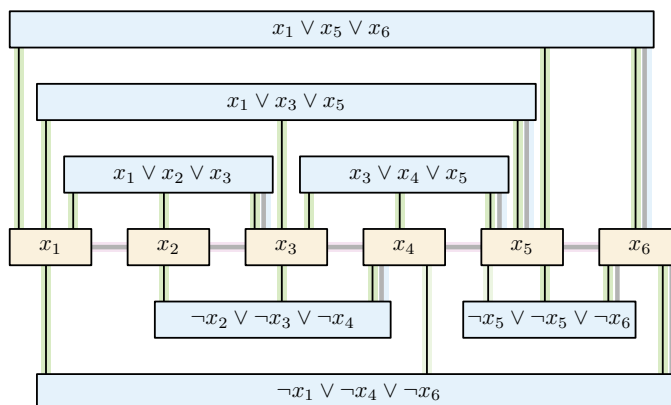
The clause gadget mainly consists of a set of squares forming a *pitchfork* (\pitchfork) shape (blue squares in Figure 5c). The pitchfork has three *tines* consisting of two squares each. Each tine corresponds to a literal in the clause. We add a path of squares connecting the \pitchfork shape to the central path of cycles so that the source configuration is connected (gray squares). Most squares are in both the source and the target configurations. The only exception is one square (marked with \times) that wants to be transferred to a nearby position (marked with \circ). However, the move is initially not possible as it would disconnect the \pitchfork part.

The wire gadgets are connected to the variable gadgets and part of them is placed very close to each of the tines of a pitchfork. A wire gadget is a path of squares that form a \sqcap shape for positive literals and a \sqcup shape for negative ones (see Figure 5b, green squares). Each wire gadget is attached to a different prong of the corresponding variable gadget. This associates each literal in a clause to a wire gadget and a prong (note that there can be spare prongs). The goal is to allow creating a different connection between the \pitchfork of a clause gadget and the central path of cycles in two moves as long as the prong is in a cycle.

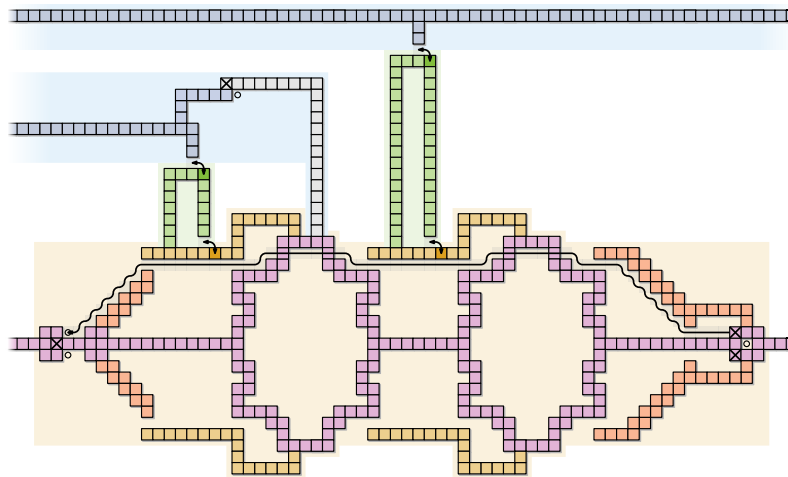
To avoid interference between different gadgets we place the clause gadgets at different heights and make the vertical separations between gadgets large enough.

► **Lemma 3.** *At least six moves are necessary to reconfigure a clause gadget, and six moves suffice if and only if a prong associated to a literal in the clause is part of a cycle.*

The six moves required by a clause gadget are in fact additional to the $20k_i + 24$ moves required to reconfigure the gadget for variable x_i and to the six moves required by any other clause gadget. If we allow only the minimum number of moves per gadget, Lemma 2 forces



■ **Figure 3** Rectilinear drawing of a PLANAR MONOTONE 3SAT instance. Our reduction attaches the variable gadgets horizontally and the clause gadgets next to the rightmost literal in the clause.



■ **Figure 4** An overview of the reduction. (Background colors match Figure 3).

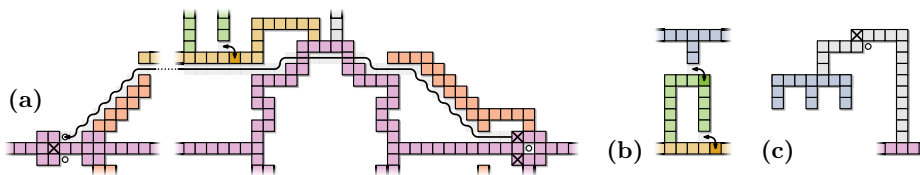
that in each variable gadget either the upper or the lower prongs become part of cycles. Moreover, Lemma 3 requires that for each clause there is a prong associated to a literal in the clause that becomes part of a cycle as part of the reconfiguration of the variable gadgets. This implies that if a reconfiguration sequence exists, the 3SAT instance must be satisfiable. In the other direction, if the 3SAT instance is satisfiable, then we show how to order the moves carefully to reconfigure with the minimum number of moves required.

► **Lemma 4.** *A PLANAR MONOTONE 3SAT instance can be solved if and only if the corresponding reconfiguration problem instance can be solved using $66m + 24l$ moves.*

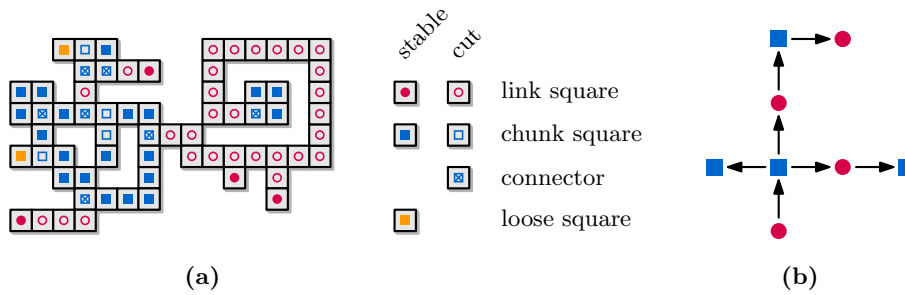
3 Input-sensitive in-place algorithm

To describe our input-sensitive reconfiguration algorithm, we first need to introduce the following definitions and notations. Let \mathcal{C} be an edge-adjacent configuration of squares on the rectangular grid and let G be the *edge-adjacency graph* of \mathcal{C} . In G each node represents a square and two nodes are connected by an edge, if the corresponding squares are edge-adjacent. With slight abuse of notation we identify the squares and the nodes in the graph. A square $s \in \mathcal{C}$ is a *cut square* if $\mathcal{C} \setminus \{s\}$ is disconnected. Otherwise, s is a *stable square*.

A configuration \mathcal{C} is *xy-monotone* if \mathcal{C} contains the entire leftmost column and bottommost row of \mathcal{C} 's bounding box, and each row or column of \mathcal{C} forms a single contiguous set of squares. A *chunk* is any inclusion-maximal set of squares in \mathcal{C} enclosed by (and including) a simple cycle σ in G of length at least 4 (its *boundary cycle*), plus any squares with degree 1 in G edge-adjacent to σ (its *loose squares*). A chunk constitutes a well-connected component that can be efficiently transformed towards an *xy-monotone* configuration.



■ **Figure 5** (a) Variable gadget. (b) Wire gadget. (c) Clause gadget.



■ **Figure 6** (a) A configuration \mathcal{C} . (b) The corresponding component tree T .

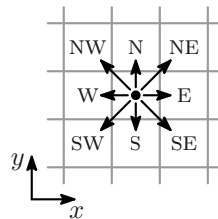
A *link* is a connected component of squares which are not in any chunk. A square is a *connector* if it is a chunk square edge-adjacent to a square in a link or in another non-overlapping chunk, or if it is the single overlapping square of two chunks. By definition a connector is always a cut square. The *size* of a chunk C is the number of squares contained in C (which includes its boundary cycle and any loose squares).

Figure 6a shows an example configuration with its chunks, links, connectors, and cut/stable squares marked. Note that a square can be part of two chunks simultaneously, in which case it must be a connector (for example, see the leftmost connector in Figure 6a). A chunk can contain both cut and stable squares.

The *component tree* T of \mathcal{C} has a vertex for each chunk/link and an edge (u, v) iff the chunks/links represented by u and v have edge-adjacent squares or share a square (when chunks are adjacent), see Figure 6b. The component containing the leftmost square in the bottom row of \mathcal{C} , the *root square*, is the root of T . Chunks in leaves of T we call *leaf chunks*.

A *hole* in \mathcal{C} is a finite maximal vertex-connected set of empty grid cells. The infinite vertex-connected set of empty grid cells is the *outside*. If a chunk C encloses a hole in \mathcal{C} , we say that C is *fragile*. Otherwise, we say that C is *solid*. The *boundary* of \mathcal{C} is the set of squares vertex-adjacent to any grid cell on the outside. The *boundary* of a hole H is the set of squares vertex-adjacent to any grid cell in H . Note that the boundary of a hole is edge-connected. We can construct T in $O(n)$ time by walking along the boundary of \mathcal{C} .

Consider now the bounding box B of \mathcal{C} on the square grid. We refer to the bottommost leftmost grid cell inside B as the *origin*. Let P be the perimeter of B , then any square in \mathcal{C} can be connected to the origin by an *xy-monotone* path of at most $P/2$ squares.



Let $c = (x, y)$ be a grid cell. We use compass directions (N, NE, E, etc.) to indicate neighbors of c . When we use grid coordinates, we assume the usual directions (the x -axis increases towards E and the y -axis increases towards N, so the N-neighbor of c is $(x, y + 1)$). Similarly, we indicate slide moves using compass directions (‘a W-move’) and convex transitions using a sequence of two compass directions (‘a WS-move’: a movement toward w followed by a movement towards s).

Algorithmic outline. In the first phase of our algorithm we ensure that the leaves of the component tree T are sufficiently large and well-connected. Specifically, we gather squares from the leaves of T until each leaf is a chunk of size at least P . In Section 3.1 we explain how to grow chunks using at most $O(P)$ moves per square that was moved. During this process, the final position of each square is chosen inside bounding box B , but squares can move through the layer of grid cells adjacent to B .

After gathering, all leaves are *heavy chunks* of size at least P . Our goal is now to make each leaf chunk contain the origin, while ensuring that all squares remain part of their chunk. A heavy leaf chunk C contains a sufficient number of squares to be transformed into a chunk containing both the connector of C and the origin: we can connect the connector with the origin by an xy -monotone path of at most $P/2$ squares; two such paths, which are disjoint, form a new boundary cycle for C . We do not explicitly construct these two paths, but instead we *compact* the configuration by filling holes and using lexicographically monotone movement towards the origin for squares in heavy leaf chunks. In Section 3.2 we explain the details of the compaction algorithm and prove that it leaves us with a solid xy -monotone component.

During compaction each square in a leaf chunk makes only lexicographic monotone moves towards the origin while staying inside B : s- and w-moves (slides), as well as sw-, ws-, nw-, and wn-moves (convex transitions). In some cases, a square in the leftmost column or bottom row can exit B , and move along the bounding box to enter the same column/row again closer to the origin. This is the only time a non-lexicographic monotone move is used, and every square can perform it at most $O(P)$ times. Hence compacting takes $O(Pn)$ moves.

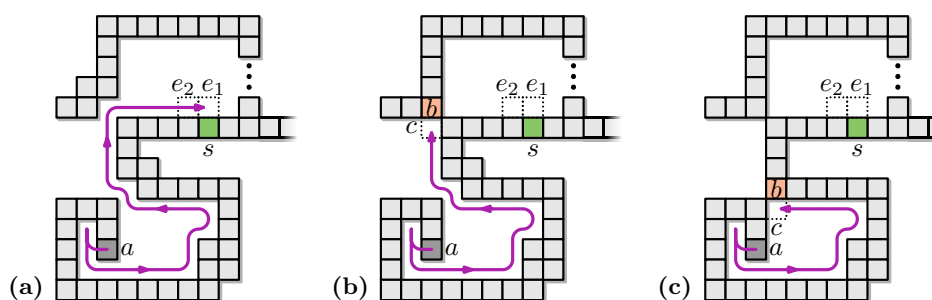
When compacting, every (heavy) leaf chunk will eventually contain a square at the origin. This means that the whole configuration becomes a single chunk, as all leaves of the component tree have merged into a single component. Therefore, once compacting has finished we arrive at an xy -monotone configuration that fits inside B . If at any point during this process the configuration becomes xy -monotone, then we simply stop. In particular, if the configuration is xy -monotone at the start, for example squares in only a single row or column, then we do not have to gather or compact, even though there are no heavy chunks. See the top row of Figure 2 for a visual impression of our algorithm.

In the special case that the input configuration \mathcal{C} contains less than P squares, we first ensure that \mathcal{C} contains the origin and then execute the gathering and compaction as before. The number of moves is trivially bounded by $O(Pn) = O(P^2)$, see Section 3.4.

Finally, in Section 3.3 we show how to convert any xy -monotone configuration into a different xy -monotone configuration with at most $O(\bar{P}n)$ moves, where \bar{P} is the maximum perimeter of the bounding boxes of source and target configurations. Thus, since all moves are reversible, we can transform the source into the target configuration via this transformation.

► **Theorem 5.** *Let \mathcal{C} and \mathcal{C}' be two configurations of n squares each, let P and P' denote the perimeters of their respective bounding boxes, and let $\bar{P} = \max\{P, P'\}$. We can transform \mathcal{C} into \mathcal{C}' using at most $O(\bar{P}n)$ moves while maintaining edge-connectivity at all times.*

Proof. For any two configurations \mathcal{C} and \mathcal{C}' of n squares each, we can apply gathering and compacting to find xy -monotone configurations M in $O(Pn)$ and M' in $O(P'n)$ moves, for \mathcal{C} and \mathcal{C}' respectively. If we want to transform \mathcal{C} into \mathcal{C}' , we first gather and compact \mathcal{C} into M , transform M into M' in $O(\bar{P}n)$ moves, and proceed by reversing the sequence of steps for \mathcal{C}' to get configuration \mathcal{C}' . In Sections 3.1, 3.2 and 3.3 we show that gathering, compacting and transforming xy -monotone configurations require the appropriate number of moves, such that the total number of moves is $O(Pn + \bar{P}n + P'n) = O(\bar{P}n)$. ◀



■ **Figure 7** Light square s (green); filling cells e_1 and e_2 makes s part of a chunk; a stable square a (dark grey) moves towards e_1 along the boundary. **(a)** a reaches e_1 . **(b)** square b (brown) part of a component outside of D , moving a to c creates a chunk containing s . **(c)** square b part of a component in D , moving a to c creates a hole; its inner boundary will not be traversed again.

3.1 Gathering

In this section we show how to gather squares from the leaves of the component tree T until we create a chunk of size at least P that is a leaf of T . In the following, let s be a connector or a cut square in a link. By definition, s lies on the boundary of \mathcal{C} . Since s is a cut square, removing s from \mathcal{C} results in at least two connected components. One of these components contains s from \mathcal{C} . We say that the other (up to three) components are *descendants* of s . Let D be the set of squares in the descendant components of s . We say that the *capacity* of s is $|D|$, and that s is *light* if its capacity is less than P and *heavy* otherwise.

► **Lemma 6.** *Let s be a light square with descendant squares D . Then s can be made part of a chunk with a sequence of $O(P)$ moves by squares in D . This procedure is in-place.*

Proof. Observe that there exist up to two empty cells e_1 (and e_2) neighboring s , such that moving squares there results in a chunk component containing s (see Figure 7). Cell e_1 (and e_2) can be chosen such that they lie inside bounding box B : these cells must exist since always at least three neighboring cells are inside B , unless B is a single row/column and the configuration was already *xy-monotone*. If such cells are already occupied by squares then s is already in a chunk. We argue that we can move squares from the descendant components of s into these empty cells with at most $O(P)$ moves. Once this is accomplished, we repeat the process in the descendant components, for the next light square of maximal capacity, until no light squares remain in the component tree below the chunk containing s .

Let $D' \subseteq D$ be a subset of boundary squares in the descendants of s of the subconfiguration $D \cup \{s\}$. Select an arbitrary stable square $a \in D'$. Such a square exists because of the following: if there is a link component in D that is a leaf in the component tree, then its degree-1 node is stable; and if there is a chunk component in D that is a leaf in the component tree, then an extremal square of the chunk in one of the NE, NW, SE, or SW directions is stable (only one of them can be a connector square).

Consider moving a along the boundary of D towards e_1 . Let E_a be the set of cells that a needs to cross to reach e_1 . If E_a is empty, then we simply move a to e_1 (see Figure 7a), and repeat the procedure for e_2 (if it exists). In this case, a takes $O(P)$ moves to get to e_1 , since a can take a simple path along the at most P descendants in D .

Now consider the case where E_a is not empty. Let b be the first square in E_a on the way from a to e_1 ; let c be the square in E_a that is just before and edge-adjacent to b . As b is not part of the boundary along which the path from a to e_1 is considered, it must be vertex-adjacent to a square that is on that part of the boundary.

There can be two cases: either $b \notin D$ or $b \in D$. In the first case, moving a to c merges a component in D with some component outside of D (see Figure 7b). Thus, a chunk is created that contains s , resulting in s no longer being a light square.

In the second case, when $b \in D$, moving a to c creates a chunk within D (see Figure 7c). In this case we select another arbitrary stable square a' in the new subconfiguration $D \cup \{s, c\} \setminus a$, and repeat the procedure. Observe that the empty squares traversed by a are now part of a hole in the new chunk. Thus the path from a' to e_1 does not overlap with the path taken by a . Let $\{a_0, a_1, a_2, \dots\}$ be the sequence of such stable squares chosen by our algorithm as candidates to be moved to e_1 . For any square a_i , its path along the boundary to e_1 does not overlap with any of the cells traversed by all a_j with $j < i$. Thus, there is some k such that a_k either reaches e_1 , or it merges the components within and outside of D into a chunk containing s . The total number of empty cells traversed by all squares a_i ($0 \leq i \leq k$) is $O(P)$.

We repeat the above procedure to fill e_2 . Note that the path taken by a (and a') may not always be inside bounding box B . When this path exits B , it will always stay adjacent to cells in \mathcal{C} , and hence it will use only the single row/column of cells adjacent to B . ◀

Using the procedure described in Lemma 6, we can iteratively reduce the number of light squares to obtain a component tree where all leaves are chunks of size at least P : any light square with capacity $P - 1$ will form a chunk of size P with all its descendants.

► **Lemma 7.** *An in-place reconfiguration of $O(Pn)$ moves exists, which ensures that all leaves in the component tree are chunks of size at least P .*

Proof. By Lemma 6, we can make a light square s part of a chunk in $O(P)$ moves by moving cubes from D , the set of descendants of s . This in-place process creates new light squares only if removal of a square a breaks a cycle in D . Thus, every new light square is part of D .

We repeat the procedure, selecting a light square of maximal capacity at every step. Overall, a square can be light at most once in the process. Thus, after $O(Pn)$ moves no light squares remain, and all the leaves in the component tree are chunks of size at least P . Note that, while the root square always has capacity $n - 1$, an adjacent square can have capacity $2 \leq |D| \leq P$, and the resulting chunk will be the root component, either because the root square is on the boundary cycle, or is a loose square. In case the adjacent square is too light, namely $|D| < 2$, then the root component may stay a link. ◀

3.2 Compacting

After the gathering phase, each leaf of the component tree T is a heavy chunk, that is, each leaf is a chunk of size at least P . In this section, we describe how we compact the configuration in order to turn it into a *left-aligned histogram* containing the origin. A left aligned histogram has a vertical base, and extends only rightward. Our procedure uses three types of moves: LM-moves, corner moves, and chain moves, which we discuss below. We iteratively apply any of these moves. The correctness of the algorithm does not depend on the order in which moves are executed, as long as the moves are *valid* (defined below). Our implementation assigns priorities to the various types of moves and chooses according to these priorities whenever multiple moves are possible (see Section 4).

LM-moves. We say that a move is a *lexicographically monotone move* (*LM-move* for short) if it is either an s- or w-move (slides), or an SW-, WS-, NW-, or WN-move (convex transitions). Note that an LM-move will never move a square to the east. Squares can move to the north, but only when they also move to the west. Hence, if a square starts at coordinate (x, y) , and we perform a series of LM-moves, it stays in the region $\{(x', y') \mid x' \leq x \wedge y' \leq x - x' + y\}$.

Let s be a square in a heavy leaf chunk C of \mathcal{C} , and consider an LM-move made by s . We say that this move is *valid* if s stays inside bounding box B , and all squares $s' \in C$ are still in a single chunk after the move. While compacting, we allow only valid LM-moves, that is, we allow each chunk to grow, but a chunk can never lose any squares.

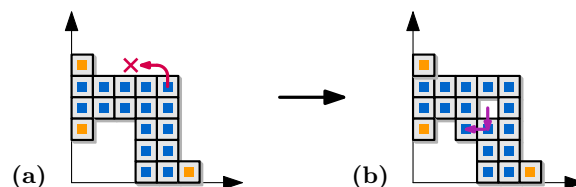
Corner moves. LM-moves on their own are not necessarily sufficient to compact a chunk into a suitable left-aligned shape. For example, consider the configuration in Figure 8a, which does not admit any valid LM-moves. However, it has a concave corner that we can fill with two moves (see Figure 8b), to expand the chunk in that direction. Repeating such corner moves allows us to make the chunk in the example left-aligned.

We define corners of a chunk C with boundary cycle σ as follows. A *top corner* (Figure 9a–d) is an empty cell with squares $b_1, b_2, b_3 \in \sigma$ as N-, NE-, and E-neighbors. Similarly, a *bottom corner* (Figure 9e–h) is an empty cell with squares $b_1, b_2, b_3 \in \sigma$ as S-, SE-, and E-neighbors. Note that a corner can be either inside a hole in C (*internal corner*), or on the outside of C (*external corner*), and we treat both of these in the same way.

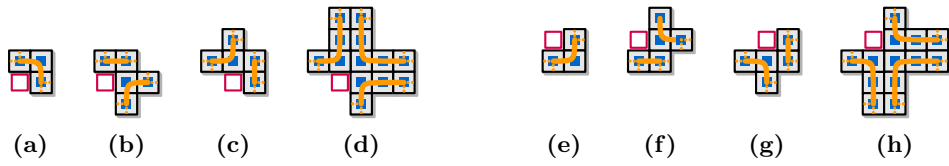
Let s be a top corner in C with neighbors b_1, b_2, b_3 as above. In the case where b_1, b_2, b_3 are consecutive squares in σ (Figure 9a), we can fill s by two slide moves: first move either b_1 or b_3 into s , and then move b_2 into the cell left empty by the first move. We call this a *top corner move*. We can fill a bottom corner with consecutive b_1, b_2, b_3 (Figure 9e) in the same way, just mirrored vertically (a *bottom corner move*). Just like for LM-moves, we say that a corner move is *valid* if all squares $s' \in C$ are still in a single chunk after the corner move. Note that all corners where b_1, b_2, b_3 are not consecutive in σ (Figure 9b–d and 9f–h) do not allow valid corner moves, as b_1 and/or b_2 becomes a connector.

Chain moves. Besides LM- and corner moves, we need a special move to prevent getting stuck when each LM-move is invalid because it would move outside the bounding box B . For example, some squares on the bottom row or leftmost column of B would be able to perform LM-moves if they were situated in any other row/column of B , as shown in Figure 10.

A *chain move* is a series of moves that is started by such an LM-move that violates validity only by leaving bounding box B . A chain move for a square s in the bottom row of B requires an empty cell $e = (x, 0)$ closer to the origin, and works as follows. Square s must be able to perform an LM-move, more precisely an SW-move that is invalid only because it leaves B . We want to place s in this empty cell e , unless it creates a link component, which happens only if the square on position $(x + 1, 0)$ is a loose square. We slide such a loose square upwards with a N-move, and identify the emptied cell as e . Note that e is again the closest empty cell in the bottom row, closer to the origin. We can then move s to e by performing an SW-move, a series of W-moves, and finally a WN-move into e . For a square s in the leftmost column, the direction of all moves is mirrored in $x = y$.



■ **Figure 8** (a) A configuration that does not admit LM-moves. For example, an NW-move (in red) of the top-right square is not valid. (b) Two slide moves expand the concave corner in SW direction.



■ **Figure 9** Empty squares shown in red are corners. The boundary cycle of the chunk is shown in orange. (a)–(d) Top corners; (e)–(h) bottom corners. Corners shown in (a) and (e) can be both external and internal. Corners shown in (b)–(d) and (f)–(h) can only be internal.

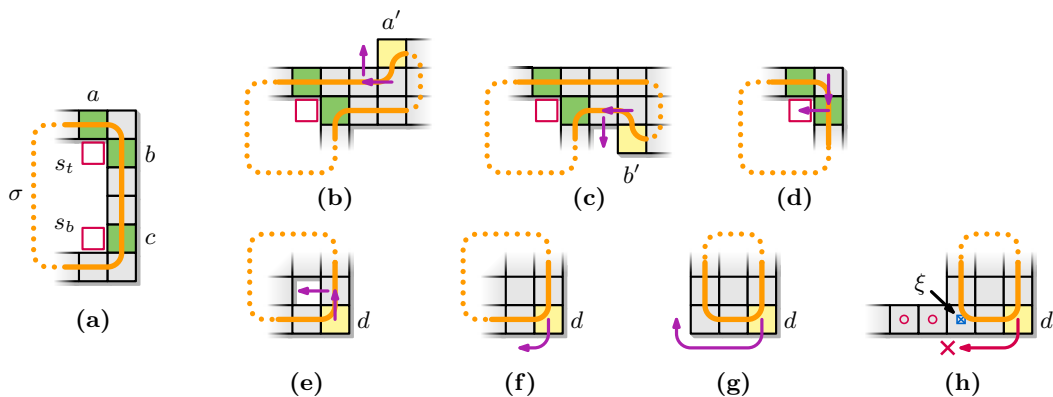


■ **Figure 10** Examples of chain moves: a square moves outside B to the first empty cell closer to the origin. This may require a loose square to move as well.

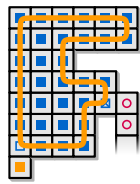
► **Lemma 8.** *Let C be a leaf chunk that does not admit valid LM-moves, corner moves, or chain moves. Then C is solid, and its boundary cycle σ outlines a left-aligned histogram.*

Proof. We first show that C is solid. Assume to the contrary that C has a hole. Consider the top- and bottommost empty squares s_t and s_b of the rightmost column of empty squares of any hole in C (s_t may be equal to s_b). Let a and b be the N- and E-neighbors of s_t , respectively, and let c be the E-neighbor of s_b (see Figure 11a). We know that $a, b, c \in \sigma$, because otherwise moving a or b into s_t , or c into s_b , would be a valid LM-move. C has at most one connector ξ , which is part of σ . We now show that ξ lies strictly between a and b on σ , and also that ξ lies strictly between c and a , to arrive at a contradiction.

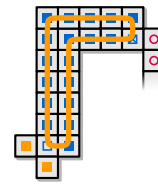
Let σ' be the part of σ strictly between a and b , walking along the boundary of C in the clockwise order. If σ' visits the row above a (leaving the row of a for the first time at square a'), then there exists a bottom corner move filling the w-neighbor of a' (see Figure 11b). This move is valid since the part of C to the right of s_t by definition does not contain any holes. Similarly, if σ' visits the row below b (returning to the row of b for the last time at square b'), then there exists a top corner move filling the w-neighbor of b' (see



■ **Figure 11** Lemma 8: (a) A chunk hole with s_t and s_b marked. (b)–(d) Possible valid moves for σ' . (e)–(g) Possible valid moves for σ'' . (h) If a chain move is not possible then ξ is left of d .



■ **Figure 12** The boundary cycle σ of the leaf chunk C outlines a left-aligned histogram.



■ **Figure 13** The chunk C forms a double- Γ shape, with one or two loose squares (orange).

Figure 11c). We conclude that the part of S attached to a and b is a protrusion of two rows tall. The protrusion is non-empty, because otherwise we can either move a loose square with an LM-move or fill s_t with a corner move (see Figure 11d). Consider the rightmost column of the protrusion. If this column contains a loose square, we can move it with an LM-move. Hence, the column contains a square in the row of a and a square in the row of b . If neither of these is ξ , then we can perform an LM-move. Hence, σ' contains ξ .

Let σ'' be the part of σ strictly between c and a in clockwise order. Walk over σ'' until encountering the first square d whose N- and W-neighbors are adjacent to d on σ . Assume that ξ is not d or one of its neighbors. If d or the N-neighbor of d have a loose square attached to them, we can perform an LM-move on this loose square. Otherwise, if the NW-neighbor of d is part of a hole in C , then this hole can be filled with a bottom corner move (see Figure 11e). Otherwise, there are three cases: either (1) we can perform an S- or SW-move on d (see Figure 11f), or (2) we can perform a horizontal chain move (see Figure 11g), or (3) the chain move is impossible because no empty square e is available to move to (see Figure 11h). In cases (1) and (2), by contradiction ξ is d or one of its neighbors; in case (3), ξ needs to be in the bottommost row. In any case, σ'' contains ξ .

As σ' and σ'' are disjoint, they cannot both contain ξ , which results in a contradiction. Hence, C is solid. To show that σ outlines a left-aligned histogram, we observe that any external corner (with neighbors b_1, b_2, b_3 as defined above) admits a valid corner move. Indeed, none of b_1, b_2, b_3 can be a loose square, as those would admit LM-moves. Furthermore, as C is solid, a boundary square cannot be a cut square for squares on the inside of σ . Finally, since C is a leaf chunk, the only cut square in σ is its connector. As only one out of b_1 and b_3 can be this connector, we can perform a valid corner move starting with the other (non-connector) square. Therefore, by our assumption that there are no corner moves in C , there cannot be external corners, and thus σ outlines a left-aligned histogram (see Figure 12). ◀

A set of squares is a *double- Γ* if it fills the top two rows and left two columns of its bounding box.

► **Lemma 9.** *Let C be a leaf chunk that does not contain the origin and does not admit valid LM-moves, corner moves, or chain moves. Then the squares of the boundary cycle of C are a double- Γ (see Figure 13).*

Proof. Let C^* be the set of squares outlined by σ . By Lemma 8, C^* is a left-aligned histogram. Let $\{r_1, r_2, \dots\}$ be the rows in C^* , ordered from top to bottom. The connector ξ of C lies on σ , and thus in C^* . Assume that ξ is in row r_i ($i \geq 3$). In that case, the leftmost square of r_1 has a valid move m : a WS-move or a vertical chain move. In particular, m is not blocked by a loose square in the leftmost column of C , because that column can contain only one loose square in the last row of C^* (otherwise one of the other loose squares would admit a valid S-move). Similarly, C cannot contain a loose square, which could block m , in its topmost row. Indeed, such a loose square would admit a W-, WS-move, or a vertical

chain move (if it has an s-neighbor in σ), or it would admit an s-move (if it has a w- or e-neighbor in σ). The existence of m leads to a contradiction, so ξ lies on r_1 or r_2 . Because C^* is 2-connected, this implies that $|r_1| = |r_2|$, forming the horizontal leg of the double- Γ .

Consider the last row r_k ($k \geq 3$) such that $|r_k| > 2$. The rightmost square in r_k has a valid sw-, s-move, or horizontal chain move (which, by a similar argument as before, cannot be blocked by a loose square). On the other hand, $|r_k| \geq 2$, again because of 2-connectivity. Therefore, for all rows r_i ($i \geq 3$), $|r_i| = 2$, forming the vertical leg of the double- Γ . ◀

► **Lemma 10.** *Let \mathcal{C} be a configuration in which each leaf is a chunk of size at least P . Let \mathcal{C}' be the result of exhaustively performing valid LM-moves, corner moves, or chain moves on \mathcal{C} . This reconfiguration is in-place and \mathcal{C}' is xy -monotone.*

Proof. In the compaction phase we iteratively apply LM-, corner, and chain moves on the configuration in which each leaf chunk contains at least P squares. After compaction, a leaf chunk can either contain the origin, or not. Consider such a chunk C that does not contain the origin. By Lemma 9, the cycle of C will outline a double- Γ , and since C is a leaf, it has at least P squares. Gathering and compacting are in-place, since squares always move to a cell inside the initial bounding box B of the configuration. Even if a square moves outside B during gathering or chain moves it always ends inside B (by Lemma 6 and by definition, respectively). Hence the connector of C will also be inside B . Since P is the perimeter of B , any double- Γ of P squares completely inside B will reach the bottom left corner of B . Thus, C must contain the origin, as one of the top two rows connects to the connector inside B . As a result, every leaf chunk contains the origin at some point during compaction.

Once every leaf chunk contains the origin, the whole configuration is one single chunk: all leaves of the component tree form a single component now. Continuing the compacting hence results in a left-aligned histogram, by Lemma 8. Finally consider the topmost row r of this histogram that is longer than the row below it. During compaction, the rightmost cube of r can perform a valid LM-move, namely a s- or sw-move to the row below it. Note that these moves cannot put cubes outside of the original bounding box B of \mathcal{C} . Thus, once the compacting phase is completed, the configuration is xy -monotone inside B . ◀

► **Lemma 11.** *There is an in-place reconfiguration of $O(Pn)$ moves of a configuration in which all leaves are chunks of size at least P to an xy -monotone configuration.*

Proof. Using Lemmata 8, 9, and 10, we can transform a configuration \mathcal{C} , in which all leaves are chunks of size at least P , to an xy -monotone configuration. Let $s = (x, y)$ be a square in \mathcal{C} . We assign to s the score $d(s) = 2x + y$, and let $d = \sum_{s \in \mathcal{C}} d(s)$. Each LM-, and bottom corner move performed in \mathcal{C} decreases d by at least 1, while every top corner move decreases d by two. Initially, $d \leq |\mathcal{C}| \cdot P$, so the total number of LM- and corner moves is also at most $|\mathcal{C}| \cdot P$. Every square s is involved in at most $P/2$ chain moves, since each chain move places s closer to the origin in the bottom row/leftmost column. Furthermore, every chain move adds at most one additional move for a loose square, which increases the above score by at most two, hence the total number of moves as a result of chain moves is also at most $O(|\mathcal{C}| \cdot P)$. ◀

3.3 Transforming xy -monotone configurations

After gathering and compacting we arrive at an xy -monotone configuration. However, this configuration is not unique and hence we need to be able to transform between such configurations. We use a potential function to guide this transformation.

► **Lemma 12.** *Let \mathcal{C}_1 and \mathcal{C}_2 be two xy -monotone configurations of n squares each, let P and P' denote the perimeters of their respective bounding boxes, and let $\bar{P} = \max\{P, P'\}$. We can reconfigure \mathcal{C}_1 into \mathcal{C}_2 using at most $O(\bar{P}n)$ moves, while remaining in-place.*

Proof. Let $\mathcal{C} := \mathcal{C}_1$. For each grid cell $c = (x, y)$, we define the *potential* of c to be $\phi(c) = x + y$. Let s be the bottommost square in $\mathcal{C} \setminus \mathcal{C}_2$ whose cell has maximum potential, and let e be the topmost empty grid cell with minimum potential that is occupied in \mathcal{C}_2 . We iteratively move s to e in \mathcal{C} until $\mathcal{C} = \mathcal{C}_2$. We first show that \mathcal{C} remains xy -monotone. Removing s cannot break this property: by definition of ϕ and since \mathcal{C}_2 is xy -monotone, s does not have N-, NE-, or E-neighbors in \mathcal{C} . Moreover, if it has a NW-neighbor then, by xy -monotonicity of \mathcal{C} , it has a W-neighbor too. Similarly, adding a square in e maintains xy -monotonicity. By the definition of ϕ and since \mathcal{C}_2 is xy -monotone, the cells neighboring e in the s, sw, and w directions must be occupied if they are inside the bounding box of \mathcal{C} . Moreover, xy -monotonicity of \mathcal{C} guarantees that e does not have N-, NE-, or E-neighbors.

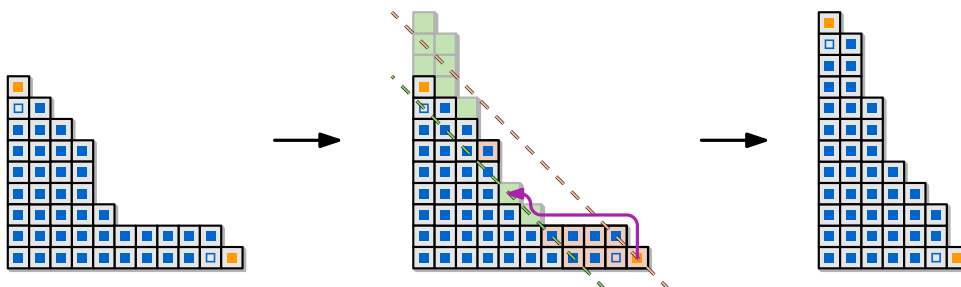
In every step we move a square from a position occupied in \mathcal{C}_1 to a position occupied in \mathcal{C}_2 , hence the perimeter of the bounding box of configuration \mathcal{C} is $O(P + P') = O(\bar{P})$. Moving one square along the boundary of \mathcal{C} is allowed for in-place reconfiguration. Since this takes at most $O(\bar{P})$ moves per square and no square is moved more than once, it takes $O(\bar{P}n)$ moves in total to reconfigure \mathcal{C}_1 into \mathcal{C}_2 . ◀

3.4 Light configurations

We say that a configuration \mathcal{C} is *light*, if it consists of fewer than P squares, where P is the perimeter of the bounding box of \mathcal{C} . Our algorithm, as explained in the main text, cannot directly handle such configurations if \mathcal{C} does not contain the origin: there are too few squares to guarantee that compacting will always result in a chunk that contains the origin. However, we can use a simple preprocessing step to ensure that \mathcal{C} will contain the origin.

For a light configuration \mathcal{C} which does not contain the origin, we select a stable square as in the gathering phase: a stable square in a link, or an extremal stable square in a chunk. We iteratively move this stable square along the boundary of \mathcal{C} to the empty cell e that is the w-neighbor of the root square. We iteratively continue to do so until \mathcal{C} contains the origin. Note that e must necessarily be empty.

At this point, we can simply gather and compact \mathcal{C} and arrive at an xy -monotone configuration, for the following reason. The gathering phase works as in the main text, since we can iteratively apply Lemma 6 on the light square closest to the root (which can be the root itself), to get a single chunk. As the root square is located at the origin, and we never move the root during gathering, we get a chunk containing the origin. Similarly, in



■ **Figure 14** Transforming between two xy -monotone configurations. The dashed lines go through cells with the same potential.

the compaction phase, this chunk will become a solid left-aligned histogram by Lemma 8. Since it already contains the origin, and we do only monotone moves towards the origin, this chunk still contains the origin. Finally the topmost row r of this histogram, that is longer than the row below it, still has valid LM-moves. Hence \mathcal{C} is xy -monotone after compaction.

There are at most $P/2$ empty cells to the left of the root. We fill each of these cells by walking along the boundary of \mathcal{C} . Since configuration \mathcal{C} consists of less than P squares, this requires at most $O(Pn) = O(P^2)$ moves. Both gathering and compacting take $O(Pn)$ moves, as proven in the main text, so including the preprocessing, we still arrive at a bound of $O(Pn)$ for the number of moves.

4 Experiments

We experimentally compared our Gather&Compact algorithm to the JavaScript implementation¹ of the in-place modification by Moreno and Sacristán [14, 15] of the Dumitrescu and Pach [10] algorithm, which we refer to as MSDP in the remainder of this section. The original algorithm by Dumitrescu and Pach always requires $\Theta(n^2)$ moves, since it builds a horizontal line to the right of a rightmost square as canonical shape. The in-place modification of Moreno and Sacristán has the potential to be more efficient in practice, since it builds a rectangle within the bounding box of the input.

We captured the output (sequence of moves) of MSDP and reran the reconfiguration sequences in our tool, to be able to verify movement sequences, count moves, and generate figures. Doing so, we discovered that MDSP was occasionally executing illegal moves, see the full version for details and for our corresponding adaptations [3]. Some of these issues could be traced to the same origin: MSDP is breaking convex transitions into two separate moves and sometimes acts on the illegal intermediate state. The number of moves we report in Table 1 counts one move both for convex transitions and for slides; hence the numbers can be lower than the numbers Moreno and Sacristán report. However, our adaptations do replace illegal moves with the corresponding (and generally longer) legal movement sequences, and hence the number of moves can also be higher than those they report.

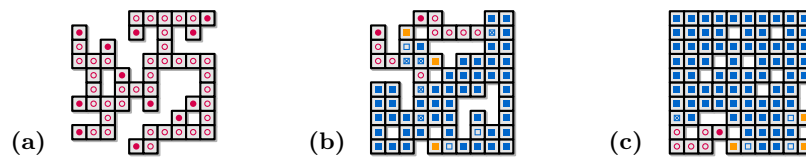
We use square grids of sizes 10×10 , 32×32 , 55×55 , 80×80 , 100×100 for our experiments. The data sets for MSDP were created by hand and are not available.² We attempted to create meaningful data sets of the same nature by starting with a fully filled square grid and

¹ <https://dccg.upc.edu/people/vera/TFM-TFG/Flooding/>

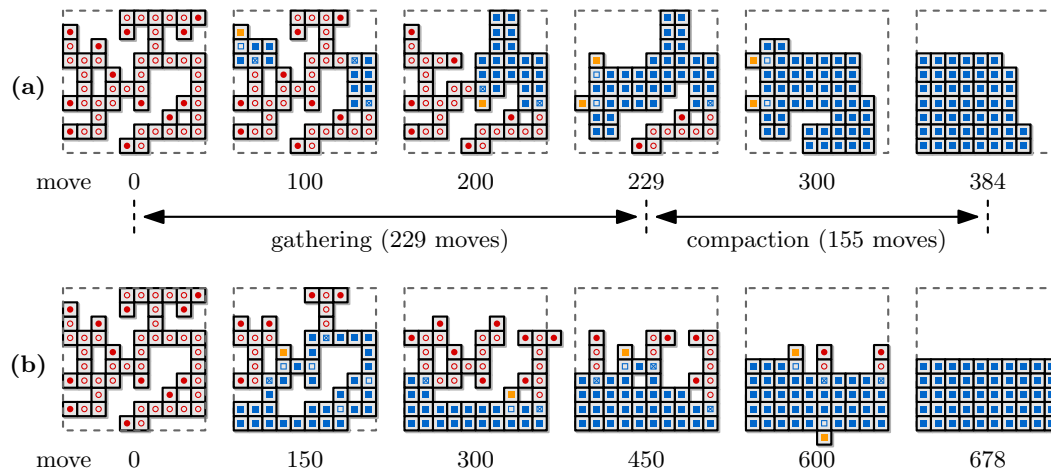
² V. Sacristán, personal communication, April 2021.

■ **Table 1** The number of moves for Gather&Compact and MSDP on various grid sizes ($D \times D$, such that $P = 4D$) and densities (in % of $D \times D$). Averages and standard deviations (in % of average) over 10 randomly generated instances are shown.

D	Gather&Compact			MSDP		
	50%	70%	85%	50%	70%	85%
10	237 31%	156 16%	95 8%	502 19%	427 21%	233 35%
32	5.395 4%	4.188 5%	2.529 8%	28.759 12%	18.447 13%	10.027 8%
55	25.916 2%	20.024 3%	12.124 4%	193.390 8%	116.431 12%	61.617 8%
80	77.745 2%	60.516 2%	36.395 3%	638.847 12%	344.529 9%	235.413 5%
100	150.666 1%	118.232 2%	69.488 3%	1.318.232 11%	743.133 17%	513.113 7%



■ **Figure 15** Example input instances on a 10×10 grid: density (a) 50%; (b) 70%; (c) 85%.



■ **Figure 16** Execution of the two algorithms on one of the input instances for grid size 10×10 , density 50%. (a) Gather&Compact; (b) MSDP. Video: <https://tinyurl.com/alga10x10>.

then removing varying percentages of squares while keeping the configuration connected. We arrived at three densities, namely (50%, 70%, 85%), which arguably capture the different types of inputs well (see Figure 15). For each value, the density of the configurations generated is close to homogeneous. The configurations with 85% density are a generalization of the “dense” configurations in the data sets for MSDP. The configurations with 70% density correspond to the so-called “medium” configurations in the data sets for MSDP, which combine the two different substructures considered for that density. The edge-adjacency graphs of the configurations with 50% density are essentially trees and, especially in the larger configurations, many leaves are not on the outer boundary (resembling the “nested” configurations in the evaluation of MSDP). For both algorithms we count moves until they reach their respective canonical configurations. Our online material³ contains our code for Gather&Compact, the input instances, and the adapted version of MSDP.

The compaction step of Gather&Compact does not rely on any particular order of the available valid moves. Our implementation prioritizes squares by descending L_∞ -distance to the origin. We also prioritize downwards LM-moves (w, ws, sw, s) over upwards LM-moves (WN, NW), and top corner moves over bottom moves.

Table 1 summarizes our results and Figure 16 shows snapshots for both algorithms on a particular instance. We observe that Gather&Compact always uses significantly fewer moves than MSDP, even on high density instances where most squares are already in place. This is likely due to the fact that MSDP walks squares along the boundary of the configuration, while Gather&Compact shifts squares locally into better position. Figure 16b shows this behavior at move 600 where one can observe a square on its way along the bottom boundary.

³ <https://alga.win.tue.nl/software/compacting-squares/>

5 Conclusion

We introduced the first universal in-place input-sensitive algorithm to solve the reconfiguration problem for the sliding cube model in two dimensions. Our Gather&Compact algorithm is input-sensitive with respect to the size of the bounding box of the source and target configurations. We experimentally established that Gather&Compact not only improves the existing theoretical bounds, but that it also leads to significantly fewer moves in practice.

We showed that minimizing the number of moves for reconfiguration is NP-complete in two dimensions. The question then arises whether the problem admits approximation algorithms. Our NP-hardness proof can be adapted to show APX-hardness in the 3D sliding cube model and we conjecture that the problem is also APX-hard for sliding squares.

There may still be room to improve on the algorithm in this paper. Specifically, it may be possible to improve the hidden constants, by gathering to leaf chunks of size $P/2$ instead of P . These chunks still have enough squares to reach the origin, but they have to give up 2-connectivity to do so, and hence the algorithm becomes more complex. Once all leaves create an xy -monotone path to the origin, the configuration again consists of a single chunk, and thus the remaining parts of our algorithm still apply.

Finally, extending our algorithm to three dimensions is currently work in progress. While well-connected components can also be transformed more efficiently in 3D, the algorithm may require a higher degree of connectivity than 2-connectivity.

References

- 1 Hugo A. Akitaya, Esther M. Arkin, Mirela Damian, Erik D. Demaine, Vida Dujmović, Robin Flatland, Matias Korman, Belén Palop, Irene Parada, André van Renssen, and Vera Sacristán. Universal reconfiguration of facet-connected modular robots by pivots: The $O(1)$ musketeers. *Algorithmica*, 83(5):1316–1351, 2021. doi:10.1007/s00453-020-00784-6.
- 2 Hugo A. Akitaya, Erik D. Demaine, Andrei Gonczi, Dylan H. Hendrickson, Adam Hesterberg, Matias Korman, Oliver Kortén, Jayson Lynch, Irene Parada, and Vera Sacristán. Characterizing universal reconfigurability of modular pivoting robots. In *Proc. 37th International Symposium on Computational Geometry (SoCG)*, pages 10:1–10:20, 2021. doi:10.4230/LIPIcs.SoCG.2021.10.
- 3 Hugo A. Akitaya, Erik D. Demaine, Matias Korman, Irina Kostitsyna, Irene Parada, Willem Sonke, Bettina Speckmann, Ryuhei Uehara, and Jules Wulms. Compacting squares: Input-sensitive in-place reconfiguration of sliding squares. *CoRR*, abs/2105.07997, 2021. arXiv:2105.07997.
- 4 Greg Aloupis, Nadia Benbernou, Mirela Damian, Erik D. Demaine, Robin Flatland, John Iacono, and Stefanie Wuhler. Efficient reconfiguration of lattice-based modular robots. *Computational Geometry: Theory and Applications*, 46(8):917–928, 2013. doi:10.1016/j.comgeo.2013.03.004.
- 5 Greg Aloupis, Sébastien Collette, Mirela Damian, Erik D. Demaine, Robin Flatland, Stefan Langerman, Joseph O’Rourke, Val Pinciu, Suneeta Ramaswami, Vera Sacristán, and Stefanie Wuhler. Efficient constant-velocity reconfiguration of crystalline robots. *Robotica*, 29(1):59–71, 2011. doi:10.1017/S026357471000072X.
- 6 Byoung Kwon An. EM-Cube: Cube-shaped, self-reconfigurable robots sliding on structure surfaces. In *Proc. 2008 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3149–3155, 2008. doi:10.1109/ROBOT.2008.4543690.
- 7 Nora Ayanian, Paul J. White, Ádám Hálász, Mark Yim, and Vijay Kumar. Stochastic control for self-assembly of XBots. In *Proc. ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (IDETC-CIE)*, pages 1169–1176, 2008. doi:10.1115/DETC2008-49535.

- 8 Nadia M. Benbernou. Geometric algorithms for reconfigurable structures. PhD thesis, Massachusetts Institute of Technology, 2011.
- 9 Chih-Jung Chiang and Gregory S. Chirikjian. Modular robot motion planning using similarity metrics. *Autonomous Robots*, 10:91–106, 2001. doi:10.1023/A:1026552720914.
- 10 Adrian Dumitrescu and János Pach. Pushing squares around. *Graphs and Combinatorics*, 22:37–50, 2006. doi:10.1007/s00373-005-0640-1.
- 11 Robert Fitch, Zack Butler, and Daniela Rus. Reconfiguration planning for heterogeneous self-reconfiguring robots. In *Proc. 2003 IEEE/RSJ International Conference on Intelligent Robots and System*, pages 2460–2467, 2003. doi:10.1109/IR0S.2003.1249239.
- 12 Kazuo Hosokawa, Takehito Tsujimori, Teruo Fujii, Hayato Kaetsu, Hajime Asama, Yoji Kuroda, and Isao Endo. Self-organizing collective robots with morphogenesis in a vertical plane. In *Proc. 1998 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2858–2863, 1998. doi:10.1109/ROBOT.1998.680616.
- 13 Othon Michail, George Skretas, and Paul G. Spirakis. On the transformation capability of feasible mechanisms for programmable matter. *Journal of Computer and System Sciences*, 102:18–39, 2019. doi:10.1016/j.jcss.2018.12.001.
- 14 Joel Moreno. In-place reconfiguration of lattice-based modular robots. Bachelor’s thesis, Universitat Politècnica de Catalunya, 2019.
- 15 Joel Moreno and Vera Sacristán. Reconfiguring sliding squares in-place by flooding. In *Proc. 36th European Workshop on Computational Geometry (EuroCG)*, pages 32:1–32:7, 2020.
- 16 Irene Parada, Vera Sacristán, and Rodrigo I. Silveira. A new meta-module design for efficient reconfiguration of modular robots. *Autonomous Robots*, 45(4):457–472, 2021. doi:10.1007/s10514-021-09977-6.
- 17 Daniela Rus and Marsette Vona. A physical implementation of the self-reconfiguring crystalline robot. In *Proc. 2000 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1726–1733, 2000. doi:10.1109/ROBOT.2000.844845.
- 18 John W. Suh, Samuel B. Homans, and Mark Yim. Telecubes: mechanical design of a module for self-reconfigurable robotics. In *Proc. 2002 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4095–4101, 2002. doi:10.1109/ROBOT.2002.1014385.
- 19 Cynthia Sung, James Bern, John Romanishin, and Daniela Rus. Reconfiguration planning for pivoting cube modular robots. In *Proc. 2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1933–1940, 2015. doi:10.1109/ICRA.2015.7139451.
- 20 Mark Yim, Wei-Min Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory S. Chirikjian. Modular self-reconfigurable robot systems. *IEEE Robotics & Automation Magazine*, 14(1):43–52, 2007. doi:10.1109/MRA.2007.339623.