Sliding Cubes in Parallel

Hugo A. Akitaya
Miner School of Computer and Information Sciences, University of Massachusetts Lowell, MA, USA

Joseph Dorfer
Institute of Algorithms and Theory, Graz University of Technology, Austria

Peter Kramer
Department of Computer Science, TU Braunschweig, Germany

Christian Rieck
Institute of Mathematics, University of Kassel, Germany

Gabriel Shahrouzi
Miner School of Computer and Information Sciences, University of Massachusetts Lowell, MA, USA

Frederick Stock
Miner School of Computer and Information Sciences, University of Massachusetts Lowell, MA, USA

Abstract

In the classic ***sliding cube model*** for programmable matter in three dimensions, the task is to find a reconfiguration sequence between two connected configurations of n indistinguishable unit cube ***modules*** by sliding modules along their neighbors' faces. Depending on the objective, this sequence should minimize either the total energy expended (the number of ***moves***) or the total elapsed time (the ***makespan***). We give a number of results for the three-dimensional setting, including (i) the first algorithm that achieves worst-case optimal makespan under parallel motion in three dimensions, (ii) a proof of log-APX-hardness to decide either the optimal makespan or the optimal number of moves, which is the strongest known inapproximability bound in any related model, and (iii) a proof of NP-hardness to decide the optimal makespan under parallel motion, even if the two configurations differ only by one module and the optimal makespan is at most two. Our results strengthen the inapproximability claim from [2] and answer a question of [4] in the negative.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Sliding squares, parallel motion, reconfigurability, three dimensions, constant makespan, log-APX-hardness, NP-hardness, worst-case optimality

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.26

Related Version *Full Version*: `https://arxiv.org/abs/2603.08537` [3]

Funding *Hugo A. Akitaya, Gabriel Shahrouzi, and Frederick Stock*: Supported by the National Science Foundation (NSF), grant CCF-2348067.
Joseph Dorfer: Austrian Science Fund (FWF) 10.55776/DOC183.
Peter Kramer: Partially funded by a fellowship of the German Academic Exchange Service (DAAD) and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), grant 530918134.
Christian Rieck: Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), grant 522790373.

1 Introduction

Programmable matter systems, composed of large numbers of simple modules capable of self-reconfiguration, have attracted significant attention for applications in robotics, manufacturing, and distributed computing. From an algorithmic perspective, a fundamental challenge is to transform one connected configuration of modules into another while preserving

connectivity and avoiding collisions. This problem has been studied extensively [1, 2, 4, 5, 10, 11, 16, 17, 19, 20] in the Sliding Cube model introduced by Fitch, Butler, and Rus [15], where modules move via ***slides*** or ***convex transitions*** on an integer grid, as shown in Figure 1.

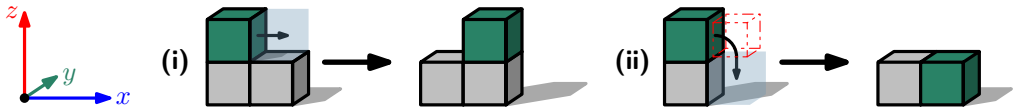


Figure 1 Two types of legal move, (i) slide and (ii) convex transition.

Early work focused on minimal energy expenditure, that is, minimizing the number of moves while moving one module at a time; we refer to this as Sequential Sliding Cubes. There exist a number of algorithms that achieve asymptotic worst-case optimality, performing reconfiguration in $\mathcal{O}(n^2)$ moves [1, 2, 5, 10, 17]. However, to the best of our knowledge, all existing algorithms for efficient reconfiguration have leveraged some notion of ***compaction***: rather than attempting to build the target configuration directly, a canonical intermediate shape is built. Notably, these techniques cannot exceed worst-case optimality.

As a first complexity result, the decision version in two dimensions was proven NP-complete via a reduction from Planar Monotone 3Sat in [2], where each produced instance is a pair of configurations whose symmetric difference is linear in the number of variables. The authors claim that the same reduction can be adapted to show APX-hardness of the minimum makespan in three dimensions, but they do not give a proof.

More recently, the authors of [4] introduced the Parallel Sliding Squares model in two dimensions, showing that parallel motion can significantly reduce the time to completion (the ***makespan***) compared to sequential schedules. They gave an algorithm that performs compaction in worst-case optimal makespan $\mathcal{O}(P)$, where P is the bounding box perimeter. In addition, they proved that it is NP-complete to decide whether an instance can be solved in makespan 1, ruling out an FPT algorithm parameterized by only the makespan (an algorithm with running time $f(k) \cdot \mathsf{poly}(n)$, where f is a computable function of the makespan k). However, as in [2], their reduction produces instances with linear-size symmetric difference, leading to the question whether the makespan minimization problem becomes tractable if the input configurations have a constant-size symmetric difference.

Our contributions. We give a number of results for three-dimensional Sliding Cubes, including an asymptotically worst-case optimal approach to parallel compaction, a negative answer to the question of [4] in three dimensions, and the strongest known inapproximability bound for both parallel and sequential movement. In particular, we prove the following:

- It is NP-hard to distinguish between optimal makespan 1 and 2 for Parallel Sliding Cubes, even if the configurations differ only by one module, answering a question of [4].
- The makespan minimization problem for both Parallel and Sequential Sliding Cubes is log-APX-hard: It cannot be approximated within a factor better than $\Theta(\log n)$.
- Asymptotically worst-case optimal schedules can be computed efficiently. In particular, one can efficiently compute compaction schedules of makespan $\mathcal{O}(A + h)$, where A is the area of a configuration's projection onto the base of its bounding box with height h.

Due to space constraints, proofs of statements marked with ($\star$) can be found in the full version of our paper [3].

Further related work

The sliding cube model, introduced by Fitch, Butler, and Rus [15] in 2003, is a central abstraction for algorithmic studies of modular reconfiguration. In the sequential model, universal compaction algorithms with quadratic worst-case output complexity are known (for squares, cubes, and hyper-cubes [1]). A desirable characteristic of such algorithms is that reconfiguration occurs ***in-place***, that is, at any given time, at most one module exceeds the bounding box of the input configuration. Notably, the algorithm proposed in [17] achieves optimal in-place compaction in a number of moves proportional to the sum of coordinates of the cubes in the input. The in-place algorithms in [1, 5] use scaffolding techniques (a substructure that provides connectivity between steps of the algorithm, proposed independently by [18] and [21]) to produce a sweep-plane that compacts the input configuration.

Parallel reconfiguration has received increasing attention. In the context of distributed algorithms for two dimensions, Dumitrescu et al. [11] first studied the problem providing linear makespan between two y-monotone shapes. Michail et al. [19] and Wolters [23] also provided algorithms that work on special input. Hurtado et al. [16] showed that a less restrictive model achieves universality within linear makespan. Using ***meta-modules***, small groups of cooperating modules that behave like a single entity, this result extends to the sliding square model for configurations that are already made of meta-modules. Akitaya et al. [4] used meta-modules and scaffolding techniques (similar to [1, 5]) to obtain a worst-case optimal algorithm for general input. While our algorithm follows similar principles, generalizing such ideas to three dimensions is highly nontrivial and required several new structural proofs.

Literature on algorithmic reconfiguration frequently focuses on worst-case bounds as it is typically much more challenging (and more often computationally intractable) to obtain optimal solutions. A classical example is the $(n^2 - 1)$ puzzle, which is famously NP-hard, but can always be solved in $\mathcal{O}(n^3)$ slides [22] if both configurations have equal parity.

In a more closely related setting to ours, Demaine et al. [8] obtained a constant factor approximation for parallel motion planning of a *swarm* of robots, where robots move between adjacent integer grid cells (similar to our slide move) without any connectivity constraint. They prove that the problem is strongly NP-hard, and thus, does not allow for an FPTAS unless P=NP. This problem has been shown APX-hard in the sequential setting [6]. To the best of our knowledge, APX-hardness is not known for parallel motion.

Fekete et al. [13, 14] studied a less restrictive model for connectivity-constrained reconfiguration. Their setting considers connected swarms of robots in the integer grid, allowing all robots to move in parallel and requiring connectivity only upon completion of each discrete time step, not during movement. Using similar techniques to [8], they showed that reconfiguration schedules with asymptotically optimal makespan can be computed even when connectivity is required, given a *fatness* assumption on the input: A configuration is c-fat if every module is contained in a fully occupied square of size $c \times c$ for some constant $c > 1$. This assumption is similar to, but weaker than, the existence of meta-modules in the input, which requires a more regular distribution of robots.

As stated before, Akitaya et al. [2] previously claimed APX-hardness for minimizing energy expenditure in the Sequential Sliding Cubes model. Our hardness of approximation result is, to the best of our knowledge, the first log-APX-hardness for this class of motion planning reconfiguration problems. Our result highlights a significant hurdle for attempts to move from worst-case optimal methods to bounded approximation factors and motivate research into, e.g., $\log n$ or $\sqrt{n}$ approximation methods for general instances. Simultaneously, they emphasize the power of structural assumptions on the input: As shown in [4], scale assumptions like those made in [13, 14] allow for significantly improved results.

2 Preliminaries

Our work considers the coordinated motion of unit cubes in the three-dimensional ***integer grid***, following and extending the notation of [4]. For $v \in \mathbb{Z}^3$, let $x(v), y(v), z(v) \in \mathbb{Z}$ refer to its coordinates along each of the three axes. The three axes point *east*, *north*, and *up* in that order. To measure distances in the integer grid, we use the L_1 and L_∞ norms defined as

$$\|u\|_1 = |x(u)| + |y(u)| + |z(u)| \qquad \text{and} \qquad \|u\|_\infty = \max\{|x(u)|, |y(u)|, |z(u)|\}.$$

A ***cell*** c in our setting is a unit cube anchored at its minimal coordinates, we write $c \in \mathbb{Z}^3$. Each cell thus has eight vertices and six faces, each bounded by a four-cycle. Two cells are then ***face-***, ***edge-***, or ***vertex-adjacent*** if they share a common face, edge, or vertex.

Configurations. A ***configuration*** C of n cube ***modules*** is defined as a set of n cells that are occupied by one module each, and is valid exactly if the face-adjacency relation (the ***dual graph of*** C) of occupied cells induces a connected graph. For a given C, we denote by B the minimal axis-aligned ***bounding box*** containing all modules. The ***extent*** of a bounding box along the x, y, and z axes defines its ***width***, ***depth*** and ***height*** (recall that z is *up*).

Moves. Modules perform two types of ***move***: a ***slide*** and a ***convex transition***. A slide translates a module by one unit along the surface of two adjacent modules, whereas a convex transition moves it along the surface of a single module across two axes. Following the notation and conventions of [4], both moves take unit time and are executed at constant speed.

Transformations. Parallel moves are performed in discrete ***transformations*** with unit-time duration. These must preserve a connected ***backbone*** which requires moving modules to be ***free***: moving modules do not provide connectivity. A set of modules M is free in a configuration C if the removal *of any subset of* M from C produces a connected configuration. This is more strict than requiring M to be non-cut, as it forbids moves along the surface of other moving modules. An individual module $\mathsf{m} \in C$ is then free if the set $\{\mathsf{m}\}$ is. In addition, transformations must not induce ***collisions***, in which the volume of two modules overlap. For instance, modules cannot trade places during a transformation, enter the same cell, or perform a convex transition through the same cell. See Figure 2 and [4] for details.

Problem statement. The Sequential Sliding Cubes problem gives two configurations of n modules each and an integer $\ell \in \mathbb{N}^+$ and asks for a sequence of at most ℓ moves that obtains one from the other. Likewise, the Parallel Sliding Cubes problem instead gives an integer $k \in \mathbb{N}^+$ and asks for a sequence of at most k transformations that obtains one from the other. Such a sequence is then called a ***schedule*** of ***makespan*** k.

This completes the description of our model; the following are auxiliary terms and notation.

Projection. We define the ***projection*** of a configuration $C \in \mathbb{Z}^3$ along an axis $\lambda \in \{x, y, z\}$ as the two-dimensional configuration $\lambda[C] = \{(\rho(\mathsf{m}), \phi(\mathsf{m})) \mid \rho, \phi \in \{x, y, z\} \setminus \{\lambda\}, \mathsf{m} \in C\}$. Furthermore, we define the ***area*** of a two-dimensional projection as $\mathbb{A}(\lambda[C]) = |\lambda[C]|$.

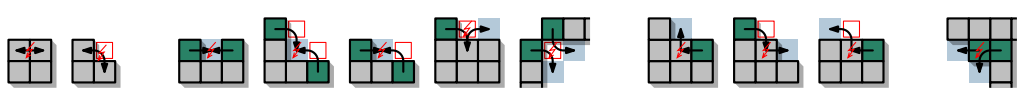


Figure 2 Examples of collisions. Figure from [4], used with the authors' permission.

Neighborhoods. We use notions of *neighborhoods* of cells and modules throughout our paper. Let $c \in \mathbb{Z}^3$. We define the open L_1 and L_∞ neighborhoods of c as

$$N(c) \coloneqq \{c' \in \mathbb{Z}^3 \mid \|c - c'\|_1 = 1\} \qquad \text{and} \qquad N[c] \coloneqq \{c' \in \mathbb{Z}^3 \mid \|c - c'\|_\infty = 1\}.$$

By an additional asterisk, e.g., $N^*[c]$, we indicate the neighborhoods which also include c. With a slight abuse of notation, these definitions extend to sets of cells $C \subset \mathbb{Z}^3$, e.g.,

$$N^*(C) \coloneqq \bigcup_{c \in C} N^*(c) \qquad \text{and} \qquad N(C) \coloneqq N^*(C) \setminus C.$$

The (closed) r-neighborhood is the rth iterate, e.g., $N^*_r(c) = N(N^*_{r-1}(c))$ with $N^*_1 = N^*$.

3 Computational complexity

In this section, we discuss two complexity results for Sliding Cubes models. In particular, we show that even seemingly "trivial" instances are computationally intractable and prove inapproximability bounds for both the Sequential and Parallel Sliding Cubes models.

Parameterized complexity. We answer a question from [4], which asked whether Parallel Sliding Cubes is FPT parameterized by the symmetric difference size, in the negative.

▶ **Theorem 1** (⋆). *Parallel Sliding Cubes is* para-NP*-hard parameterized by makespan and symmetric difference size. That is, it is* NP*-hard even if both values are constant.*

Proof sketch. We reduce from Planar Monotone 3Sat, which asks whether a given Boolean formula is satisfiable [7]. Each clause consists of at most 3 literals, all either positive or negative, and the clause-variable incidence graph must admit a plane drawing where variables are mapped to the x-axis, positive (resp., negative) clauses are mapped to the upper (resp., lower) half-plane, and edges do not cross the x-axis. Our construction places multiple instances of three types of gadget, based on a formula φ as depicted in Figure 3(a).

In particular, we place ***variable gadgets*** (blue) in the xy-plane along the x-axis, and differentiate between ***positive*** and ***negative*** modules in the variable gadgets based on their y-coordinate. Between any two successive variables, we place a ***connector gadget*** (yellow). A detailed breakdown can be seen in Figure 3(b). Also attached to each variable gadget is at least one ***clause gadget*** (green), effectively a thin comb structure that attaches to the positive or negative side of each contained variable's gadget. Finally, a single ***difference module*** (black/transparent) is located at one end of the construction in the start configuration and at the other end in the target configuration. It comprises the entire symmetric difference. To empty the difference module's initial cell and fill its target cell in the same step, we must find a path from one position to the other that does not cut the configuration apart. In particular, this path passes through each variable gadget on either the positive or negative side, separating them from clauses and thus encoding an assignment of Boolean values. ◀

▶ **Corollary 2** (⋆). *Unless* P = NP*, Parallel Sliding Cubes cannot be approximated within a factor better than* 2 *in polynomial time, even for constant-size symmetric difference.*

This explicitly rules out the existence of an FPT algorithm parameterized by the sum of makespan and symmetric difference size. We emphasize that we reduce from Planar Monotone 3Sat for clarity only: the construction remains valid when reducing from Satisfiability, as arbitrary clause-variable incidences can be embedded in the third dimension.

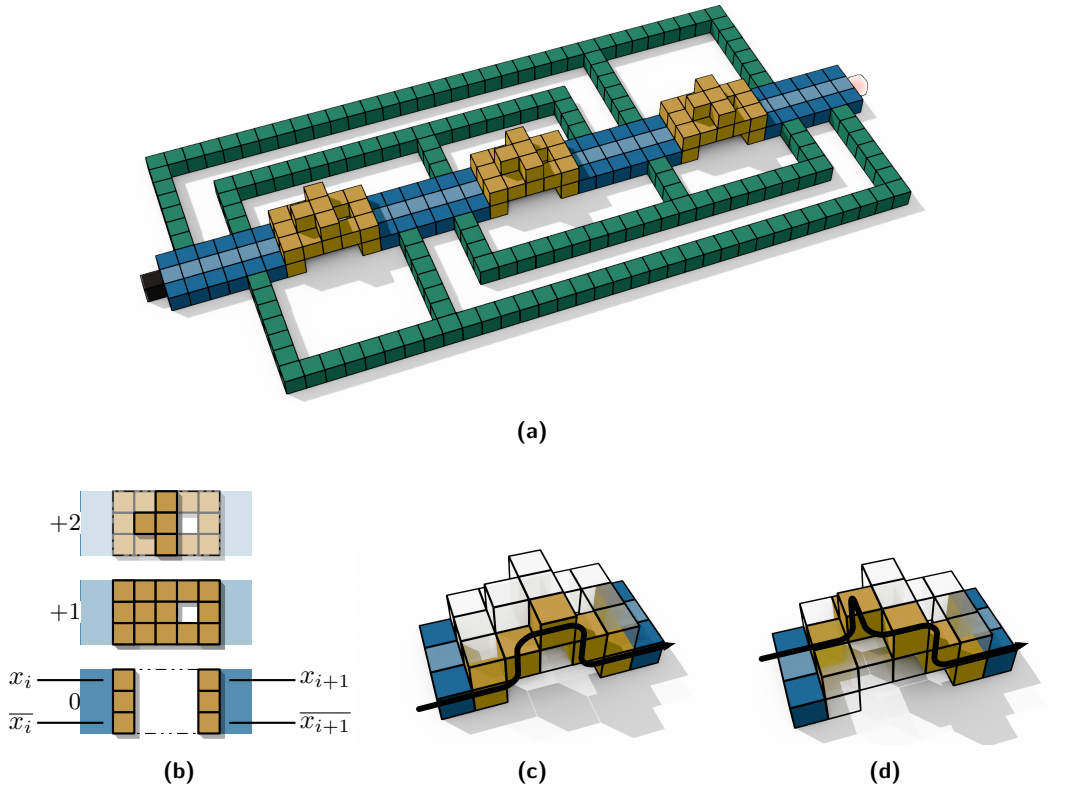


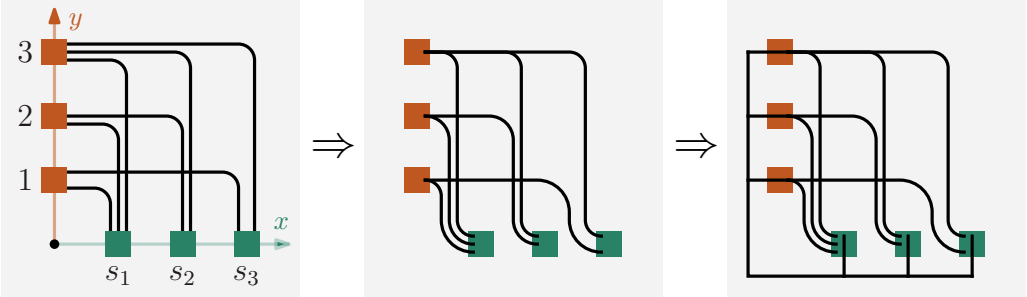
Figure 3 In (a) we show our construction, with colours indicating gadgets as outlined in the text, for $\varphi = (x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$. Variable gadgets are placed in ascending order left to right. In (b)–(d) we show the connector gadget and possible paths through.

Inapproximability. We proceed by showing that both the parallel and sequential variants of Sliding Cubes are log-APX-hard by reduction from Set Cover [9]. An instance $\Sigma_{n,k}$ consists of natural numbers $[n] = \{1, \ldots, n\}$ and a set $S = \{s_1, \ldots, s_k\}$ of subsets of $[n]$ such that $\bigcup_{i=1}^{k} s_i = [n]$. The task is then to find a minimum cardinality subset $S^\star \subseteq S$ such that $\bigcup_{s_i \in S^\star} s_i = [n]$. Our entire construction fits inside a cube of size $\mathcal{O}(n^4 k^7)$, and its reliance on large three-dimensional structure makes it unsuitable for the two-dimensional version.

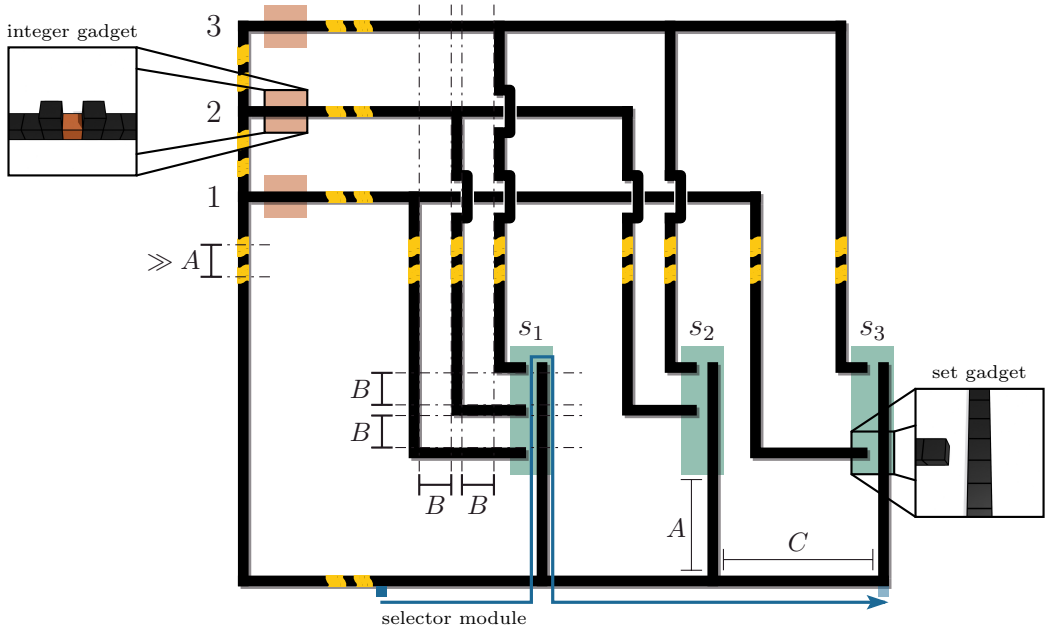
▶ **Theorem 3** ($\star$). *Both Parallel and Sequential Sliding Cubes are* log-APX*-hard and thus cannot be approximated with a factor better than* $\Theta(\log(n))$*, unless* P = NP.

Proof sketch. The high-level idea of our reduction is based on the observation that, due to the backbone constraint, movement can require cooperation of modules across great distances. We use this to design gadgets that involve moving cut (non-free) modules, which requires the existence of a large cycle in the configuration that passes through the gadget. For each gadget, such a cycle must then be created temporarily. By carefully structuring candidate cycles, we are then able to encode any instance of Set Cover as follows.

Given a Set Cover instance $\Sigma_{n,k}$, consider the bipartite graph defined by the containment relation: Its vertices corresponds to exactly $[n] \cup S$, an edge exists between $j \in [n]$ and $s_i \in S$ exactly if $j \in s_i$. This graph forms the basis of our construction. To start, we determine an embedding that maps vertices from S to the x-axis and vertices from $[n]$ to the y-axis, as shown on the left in Figure 4(a). We then "rasterize" this embedding as follows.



(a) A step-by-step illustration of how we obtain our construction for the Set Cover instance with $n = 3$ and sets $s_1 = \{1, 2, 3\}$, $s_2 = \{2, 3\}$, and $s_3 = \{1, 3\}$, starting from the graph induced by set membership.



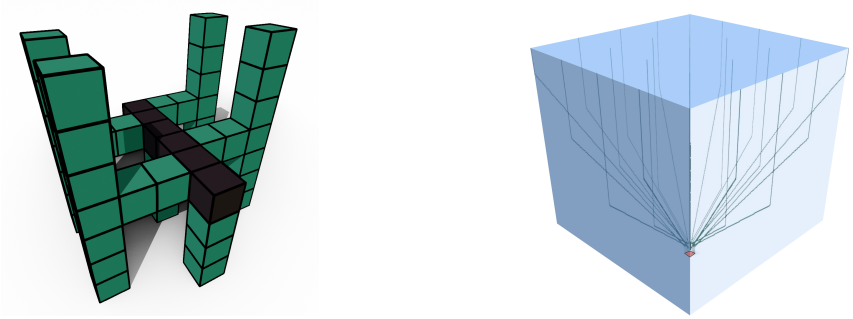
(b) Our final construction for the instance from (a) – not to scale: the dashed sections hide long distances.

Figure 4 A diagram showing an example construction for our reduction from Set Cover.

Our instance consists of start and target configurations C_I and C_F that have most modules in common. We embed each of the edges in the grid as a thin chain of modules, indicated by black lines. Crucially, we omit in this embedding the endpoints of each edge that attaches to a set vertex $s_i \in S$. This gives a disconnected configuration, so we add a "brace" through the origin that spans the configuration in both x- and y-direction. See Figure 4 for a schematic visualization, but note that for technical reasons, distances are not to scale.

For each vertex, we then place a corresponding gadget. In the ***integer gadget*** (light orange) for each $j \in [n]$, there exists a single module m_j (orange in the zoom box in Figure 4(b)) that needs to be moved up by one unit along the z-axis, but is a cut module in the configuration, and thus not free. One mechanism to make these modules free is provided by ***set gadgets*** (light green). Each set gadget contains a set of small gaps that, when filled, induce cycles in the configuration that pass through integer gadgets, freeing the corresponding module.

To fill these gaps, the construction contains a free ***selector module*** (blue) that needs to be displaced along the x-axis, but can also move to set gadgets to close cycles. For each set gadget visited, this module must move along the y-axis, incurring additional travel.



(a) Spikes alternately extend left, right, and down. **(b)** Spikes end on the surface of a large cube.

Figure 5 Spikes immobilize black modules sufficiently that an optimal solution never moves them.

Intuitively, this means that we can determine feasible schedules for our instance by visiting set gadgets with the selector module. Since each visit incurs additional moves, it is desirable to minimize the number of visits and, equivalently, minimize the corresponding solution to Set Cover. It remains to argue that the instance cannot be solved differently.

Indeed, the reader has likely noticed that the instance could be solved in significantly fewer moves by moving a small number of black modules in each integer or set gadget. To prevent this, we now briefly outline a measure that fixes all black modules in place for sufficiently many moves that the suggested approach is the only feasible one.

We call this the ***spike gadget***, which consists of polynomially many additional modules as shown in Figure 5, where spike modules are shown in green. Note that the small configuration of spikes shown in Figure 5(a) shows only the base of each spike. This construction gives all black modules at least three neighbors and makes them non-free for a chosen, bounded number of moves. Informally, each spike is a long, simply connected configuration of modules that extends from the graph embedding to evenly spaced positions on the surfaces of a cube of size $\mathcal{O}(n^4k^7)$ as shown in Figure 5(b), where spike modules are shown in green. Intuitively, since the free modules at the endpoints of spikes are so far from both one another and the graph embedding, they are not of any use to solve the underlying set cover instance, restricting all optimal solutions to moving only the selector module and integer gadgets.

As the analysis of the chosen distances and corresponding bounds on the makespan and number of moves in an optimal solution is lengthy, we refer to the full version [3]. ◀

4 Teleport

The main technical result of this section, ***teleport***, is a generalization of LocateAndFree from Abel, Akitaya Kominers, Korman, and Stock [1]. LocateAndFree searches for a free module on the outer face of a configuration C, if no such modules exist, it then locates a "nearly free" module and frees it by reconfiguring modules in the interior of C. The effect is that the located module is able to move without disconnecting C, while the exterior of C is otherwise unchanged, though internally it may be very different. Teleport moves a specified subset Q of the configuration C to new positions Q' while keeping the rest of the configuration the same, i.e., it transforms C into $C' = (C \setminus Q) \cup Q'$. In contrast to LocateAndFree, this is more powerful, as it moves multiple modules instead of just making a single module non-cut. However, teleport is more complicated as it requires that the rest of the configuration remain unchanged, not just the outermost modules like in LocateAndFree.

Given a subset Q of a configuration C we say that a schedule *teleports* Q if it transforms C into a new configuration C' with $(C \setminus Q) \subset C'$. We show the following: given two configurations C and C', with $S \subset C$ and $S' \subset C'$ such that $C' = (C \setminus S) \cup S'$ and S, S', and $C \setminus S$ are all connected subconfigurations, there exists a schedule that transforms C into C', i.e., S teleports into S'. This result is of independent interest and a version with an efficient makespan may be useful for designing simple reconfiguration algorithms. However, we do not optimize the makespan here. This suffices for our purposes, as our main algorithm only teleports constant-size subconfigurations and therefore only requires constant makespan, and allows us to present our algorithm with relatively simple proofs of correctness.

Before formally presenting teleport, we introduce definitions needed for this section and the next. A ***face*** of the configuration C is the set of module faces in the boundary of a component of the complement $\overline{C}$ (the set of cells in $\mathbb{Z}^3$ not in C). The ***outer face*** of C is the face in the boundary of the unbounded component of $\overline{C}$. Two adjacent module faces f_1 and f_2 in a face of C are ***slide-adjacent*** if either they bound the same cell $c \in \overline{C}$, or if a module in $c_1 \in \overline{C}$ containing f_1 can reach a cell $c_2 \in \overline{C}$ containing f_2 after a single move in the configuration $C \cup \{c_1\}$. Each face of C can be seen as a 2-manifold obtained by joining slide-adjacent module faces at their common edge. (See [1] for more details. Each face of C corresponds to a component of the "slide-adjacency graph" in [1].) An edge e of a configuration face is ***pinched*** if it is contained in more than two module faces in a face of C. In particular, in three dimensions a pinched edge is contained in exactly four module faces and up to two configuration faces. This happens when two modules that are not face-adjacent share an edge. We define an ***extended face*** of C, the manifold obtained by performing a topological surgery at each pinched edge joining pairs of module faces f_1 and f_2 at the common pinched edge so that f_1 and f_2 are contained in the boundary of the same module in C. Note that an extended face of C might contain many faces of C since the topological surgery might merge different faces of C. (Note that this surgery is equivalent to "smoothing" the edges of the polycube, not changing the topology of its interior. Thus, it never disconnects a face of C since this would imply that C is not connected.) The ***outer*** *extended face* of C is the extended face that contains the outer face of C. We say that a module is *in a face* F of C if one of its faces is contained in F. Intuitively, C has a single extended face if and only if its interior is homeomorphic with a 3-manifold with a connected boundary (such as the open ball or open torus).

▶ **Lemma 4** ($\star$). *There exists at least one free module in the outer extended face of any connected configuration.*

▶ **Lemma 5** ($\star$). *Given a configuration C, a connected subset $S \subset C$ with $|S| \geq 2$ such that $C \setminus S$ is connected, an empty cell e adjacent to S, a free module $\mathsf{m} \in S$ so that e and m are in the same extended face of $S \setminus \{\mathsf{m}\}$, and so that both $(S \setminus \{\mathsf{m}\}) \cup \{e\}$ and $(C \setminus \{\mathsf{m}\}) \cup \{e\}$ are connected. There exists a schedule with makespan $2^{\mathcal{O}(|S|)}$ teleporting m to e.*

We require $|S| \geq 2$ and that $(S \setminus \{\mathsf{m}\}) \cup \{e\}$ be connected so we can bound the makespan as a function of $|S|$. If either of these conditions fails, there are examples that require m to walk on modules not in S, giving a runtime dependency on $|C|$. Those conditions can be dropped if we are only concerned with the existence of such schedules. Given two configurations $C \cup S$ and $C \cup S'$ we can iteratively apply Lemma 5 to grow a component of $S \cap S'$ by filling empty cells of $S' \setminus S$ adjacent to this component. We obtain the following:

▶ **Corollary 6.** *Given two configurations $C \cup S$ and $C \cup S'$ with connected C, S and S', there exists a schedule that teleports S into S'.*

5 A worst-case optimal algorithm

Equipped with the teleport subroutine from Section 4 we create an efficient compaction algorithm for the Parallel Sliding Cubes model. On a high level, our approach generalizes the compaction from [4] to three dimensions. Still, our methods and technical arguments differ significantly. A configuration C is considered ***compact*** in the literature if for every module $\mathsf{u} \in C$, the cuboid defined by its position and the origin is fully contained in C:

$$\forall \mathsf{u} \in C: \quad \forall a \in [0, x(\mathsf{u})] : \forall b \in [0, y(\mathsf{u})] : \forall c \in [0, z(\mathsf{u})] : \quad (a, b, c) \in C.$$

Universal reconfiguration by compaction has two key steps: (1) reconfiguration into a compact configuration, and (2) reconfiguration between any two compact configurations. While the second step is much easier than the first (often almost trivial), it is still crucial; an algorithm for efficient in-place reconfiguration between compact configurations of meta-modules is given in the full version [3]. The remainder of this section will focus on step (1), the compaction.

▶ **Theorem 7.** *Schedules of makespan $\mathcal{O}(A + h)$ that compact any given configuration C can be computed in polynomial time, where h is the height of C's bounding box and A is C's projection area into the base.*

Overview. Our approach to compaction consists of two phases: Gather and Compress. The goal of Gather is to obtain a ***snake***, a connected structure of $5 \times 5 \times 5$ ***meta-modules*** that can move freely through the configuration without affecting the connectivity of the remaining configuration. To guide the creation and manipulation of meta-modules, we subdivide the space into $5 \times 5 \times 5$ ***meta-cells*** made of $5^3 = 125$ unit grid cells each. Without loss of generality, we can assume that the extent of the configuration's bounding box is a multiple of 5 in each dimension, or use the smallest such box containing C (adding at most 4 units on each axis). We identify a suitable set of modules and form a constant-size solid "ball" which we then use to efficiently construct a larger snake, see Figures 6(b)–6(e).

In the Compress phase, we transform the snake into a coarse projection of the configuration onto one of the faces of its bounding box. We sweep the resulting structure through the bounding box, transforming the configuration into meta-modules and making it compact.

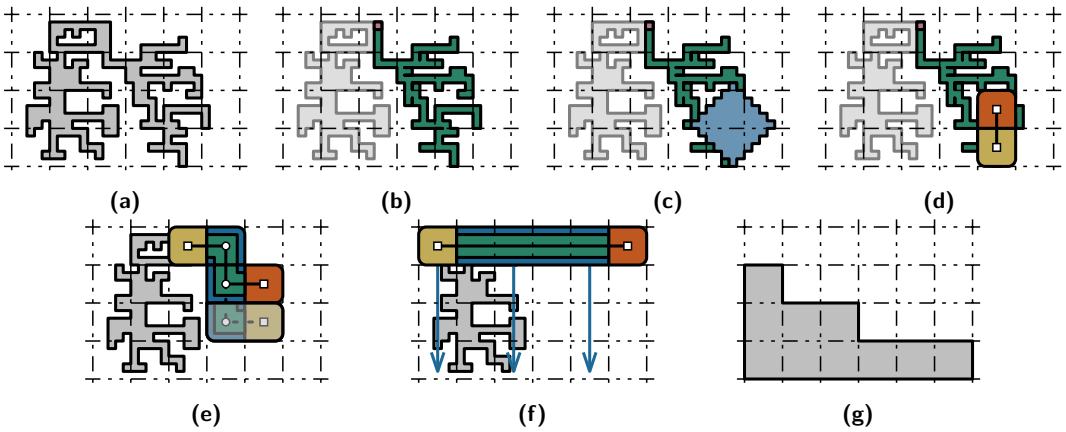


Figure 6 Overview of our algorithm (not to scale). In (a)–(d), we construct a snake, which we grow to $\Theta(A + h)$ modules in (e) and use to compact the remaining configuration in (f)–(g).

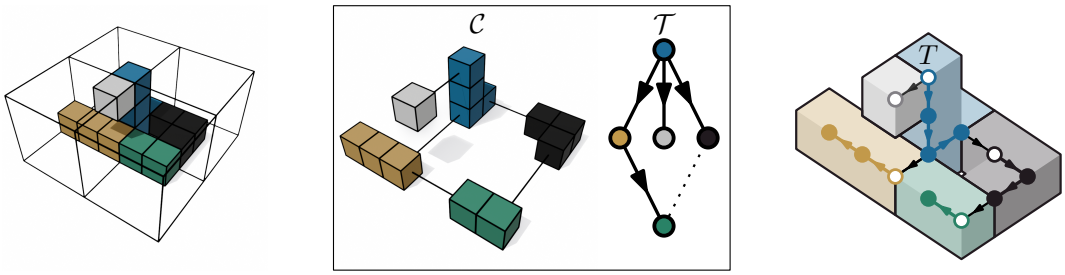


Figure 7 We divide into components, compute a spanning tree $\mathcal{T}$ of these and from this, obtain T.

5.1 Gathering phase

To build a projection of the configuration simplified into meta-cells, we need at least 5^3A modules. Since we additionally need to be able to reach the bounding box face we build the projection at, we may require an additional 5^3h modules. We thus need to gather at least $(A+h)125$ modules for the Compress phase. To achieve the makespan of $\mathcal{O}(A+h)$ we can only afford to gather at most a constant multiple of $(A+h)$ modules in the Gather phase.

Since the gathered modules will be effectively removed from the configuration, we need to find a set of modules that does not break connectivity upon removal and forms a connected shape, allowing for efficient gathering. We define a rooted spanning tree that facilitates both.

Spanning tree. Let $\mathcal{C}$ be the set of components of C induced cutting the configuration along meta-cell boundaries; C restricted to a single meta-cell may be disconnected and each of these components is an element of $\mathcal{C}$. Let $\mathcal{T}$ be an arbitrarily rooted spanning tree of the adjacency graph of $\mathcal{C}$. Based on $\mathcal{T}$, we assign each $S \in \mathcal{C}$ a root module that is adjacent to its parent component in $\mathcal{T}$. For the root component of $\mathcal{T}$, we choose the root module arbitrarily. We then compute a DFS spanning tree from the root module for each component $S \in \mathcal{C}$. To obtain our spanning tree T of C, we now connect adjacent trees until we have a single spanning tree of C, connecting the root of a component to its parent component in $\mathcal{T}$. The root of T is then simply the root of its root component in $\mathcal{C}$. See Figure 7 for an illustration.

For the remainder of this section, we denote by T the full, rooted spanning tree of C, and by $T_{\mathsf{p}} \subset T$ the subtree rooted at some module $\mathsf{p} \in C$. We further denote by T_{s} a fixed subtree of size $\Theta(A+h)$; due to T's bounded degree, T_{s} can be computed efficiently [3].

5.1.1 Building a ball

The first step to creating a snake is to obtain a meta-cell completely filled with modules. We achieve this by filling an L_1-ball of radius 12, which requires exactly 2625 modules. To this purpose, we identify a module $\mathsf{m} \in T_{\mathsf{s}}$ such that $|T_{\mathsf{m}}| \geq 2625$ and $|T_{\mathsf{m}}| \in \mathcal{O}(1)$; details are given in the full version [3]. Informally, the strategy is to move modules of T_{m} towards its root to grow a ball centered at m using teleportation (Lemma 5). This task is significantly easier to achieve in two dimensions; a similar result is obtained in [4]. The three-dimensional version requires quite intricate case analysis, even using the full power of Lemma 5. This is because in three dimensions there are more ways that the rest of the configuration $(C \setminus T_{\mathsf{m}})$ can block the advance of modules that are used to fill the desired ball.

▶ **Lemma 8** ($\star$). *Given a subtree T_m of T with size $|T_m| \geq 2625$ and $|T_m| \in \mathcal{O}(1)$, there is a $\mathcal{O}(1)$ schedule that produces a configuration where the meta-cell containing m is full, and every cell in $T \setminus T_m$ remains full.*

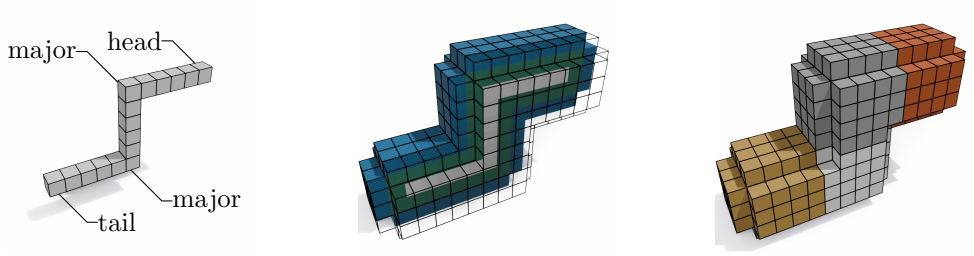


Figure 8 The spine path (white), remaining interior cells (green), and skin cells (blue) are shown in (a) and (b). In (c), we illustrate sections, with the head and tail sections in orange and yellow.

5.1.2 Finding a snake

We now give a precise description of the snake, its capabilities, and how it can be constructed.

Spine path. A ***spine path*** P of length ℓ is a set of ℓ cells that induce a simple path in the face-adjacency relation. It has endpoints (***head*** and ***tail***), ***major vertices***, and ***minor vertices***. By convention, major vertices must always be centers of meta-cells and connected by straight lines of minor vertices. The head and tail vertices are connected by induced paths of between 4 and 9 minor vertices to the closest major vertex. To prevent self-intersections, vertices with distance ≥ 5 along the spine must also have L_1-distance ≥ 5.

Cell space. Let P be a spine path. The ***interior cells*** of the snake with spine P correspond to the closed L_∞-neighborhood $N^*[P]$ of P, with the ***skin cells*** then being the open L_1-neighborhood of the interior, $N(N^*[P])$. Their union forms the snake's ***cell space*** $S(P) \not\subseteq N_2^*[P]$. We divide the cell space based on major vertices: The ***section*** of a major vertex $v \in P$ is defined as $N_2^*[v] \cap S(P)$, all remaining cells are then assigned to either the head or tail section. All of the above is illustrated in Figure 8.

Snake configuration. A ***snake configuration*** is a (sub)configuration induced by a spine path's cell space $S(P) \cap C$ in which all cells are occupied *except for the following*:

(S.1) Major vertices at which the spine bends.

(S.2) Any subset of the tail section's interior *may* be empty, except for the interior cells adjacent to three skin cells, which must be occupied by ***support modules***.

A snake by this definition always induces a connected subconfiguration, see Figure 9. In addition, keeping major vertices at bends in the spine unoccupied allows efficient teleportation of modules between the head and tail sections of any given snake. This means that the role of head and tail in a snake are fully interchangeable by filling the tail section with modules.

▶ **Lemma 9** ($\star$). *Given a snake configuration $S(P)$, the roles of head and tail vertices can be swapped in $\mathcal{O}(1)$ transformations.*

The remainder of this section assumes that we already have a snake configuration. Once C contains a large enough L_1 ball due to Lemma 8, a constant-size snake can always be found.

▶ **Lemma 10** ($\star$). *If a configuration contains a fully occupied L_1 ball with radius 12, it contains a free snake configuration with a spine path of length 11 and 198 modules.*

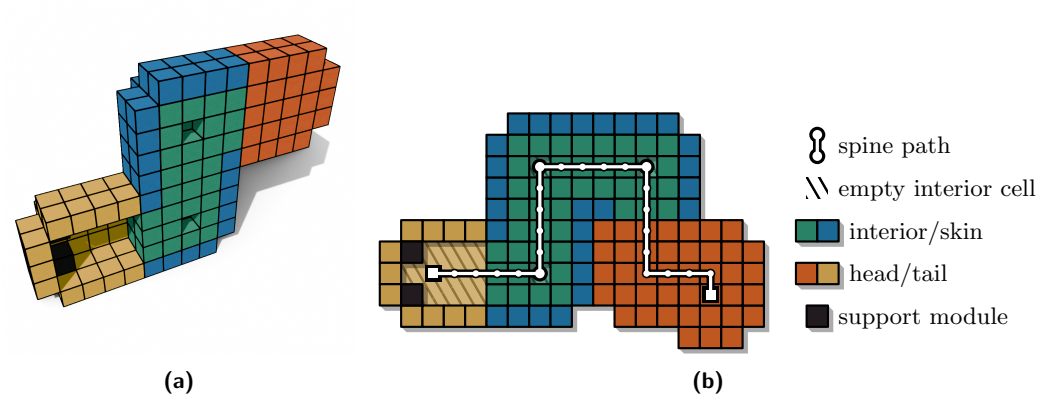


Figure 9 Structure of a snake configuration. (a) Cross-section of a valid snake configuration. (b) Two-dimensional illustration using color notation for clarity.

Having established the structure of a snake, we now define a set of operations that allow us to modify and expand a snake configuration. As a first step, we define the *push* operation and its inverse, *pull*. The push operation can be used to move the head vertex and extend the spine path by one unit. Similarly, the pull operation shortens the spine path by one unit by moving the tail vertex towards the head along the spine path.

Push. The ***push operation*** can be performed on snake configurations with at least 12 not-support modules in the tail section's interior. These are teleported to the head to fill some of the 21 newly added skin cells; note that 9 previous skin cells become interior in the same step. It moves the head in a given direction and extends the spine path by one unit, forming a valid snake configuration in the resulting cell space.

▶ **Lemma 11** (⋆). *The push operation can be performed by a schedule of makespan* $\mathcal{O}(1)$.

Each push operation expands the snake's cell space, potentially including additional modules. Some of these modules may not originally be free, meaning that we need to account for possible connectivity constraints when moving the snake. We differentiate between a snake's ***owned*** and ***held*** cells and modules, which are dynamically tracked and updated. In particular, we say that a snake configuration *holds* a cell if that cell contained a *non-free module*, which is then also held. A push operation enters 21 additional cells into the cell space, some of which must then be held. If an entered cell contains a free module before the push operation, the snake may *take ownership* of that module. The initial snake according to Lemma 10 owns all 198 contained modules. A snake will hold all cells it moves into unless otherwise stated, and will leave a module when moving away by the following operation.

Pull. The ***pull operation*** shortens the spine path by one unit. If this results in a held cell being removed from the cell space, the snake leaves a module in this cell; the number of held modules and held cells is therefore always equal. If the snake leaves a cell that is not held, it does not leave a module behind, so the number of owned modules never decreases as a result of pulling. This is strictly the reverse of Lemma 11:

▶ **Corollary 12.** *Given a snake subconfiguration with a tail that has a sufficient number of empty interior cells, the pull operation can be performed by a schedule of makespan* $\mathcal{O}(1)$.

We define two additional high-level operations, forking and joining. Both rely strictly on pushing, pulling, and the teleportation of modules within the spine of a snake.

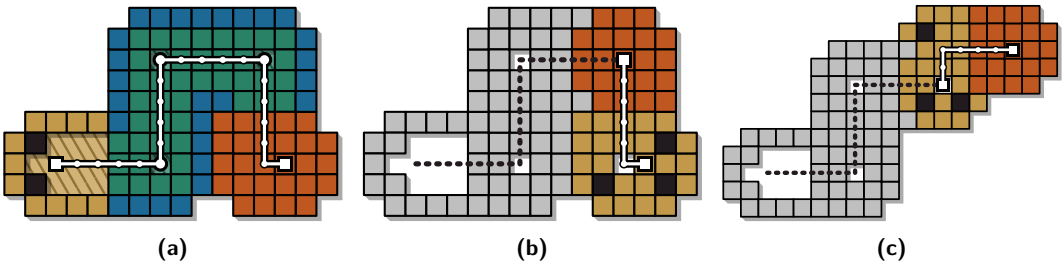


Figure 10 An illustration of forking and joining. A major vertex becomes the head vertex of a new snake in (b). If the union of their spine paths is itself a valid spine as in (c), two snakes can join.

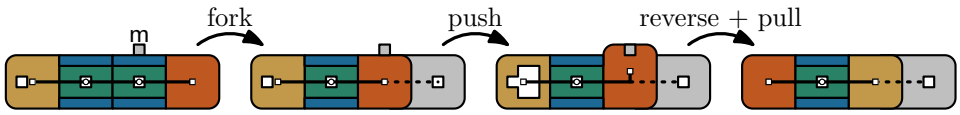


Figure 11 By forking temporarily, snakes can take ownership of free modules in close proximity.

Fork. The fork operation selects a prefix or suffix of the spine path in arbitrary orientation and teleports a constant number of interior modules along the spine to create a valid snake configuration along this spine with a chosen head vertex, as shown in Figure 10. This takes $\mathcal{O}(1)$ transformations due to Lemma 9. The new snake can then move independently.

Join. If their union constitutes a valid snake configuration, two snakes can be joined at a common endpoint to form a larger snake. This is the reverse of forking, see Figure 10.

5.2 Growing the snake

We now have the fundamental tools we need to grow the initial snake by taking ownership of additional modules. The combination of these tools enable us to universally reconfigure the snake, assuming connectivity is maintained with the rest of the configuration. Our goal is to grow the snake to own all $\Theta(A+h)$ modules in the tree $T_{\mathsf{s}} \subseteq T$. The snake configuration $S(P)$ determined due to Lemma 10 is fully contained within an L_1 ball of radius 12 that is adjacent to modules of the subtree T_{s}. We start by moving the snake, first to align with the center of a meta-cell, and then to have its head section within the same meta-cell as s. This takes $\mathcal{O}(|T_{\mathsf{s}}|)$ transformations using push and pull operations.

It remains to show that the snake can efficiently take ownership of all modules in T_{s}. We achieve this by traversing the tree in depth-first order and taking ownership of leaves.

▶ **Lemma 13** ($\star$). *Let $\mathsf{m}_1, \ldots, \mathsf{m}_\ell$ be an Euler tour representation of T_{s}. A snake configuration $\mathcal{S}$ that contains m_1 can take ownership of the entire sequence within $\mathcal{O}(\ell)$ transformations.*

As illustrated in Figure 11, constantly many snake operations allow it to take ownership of free modules close to its major spine vertices. If we keep these grid-aligned, it can thus take ownership of any free module within the same meta-cell as one of its major vertices.

▶ **Lemma 14** ($\star$). *Let $\mathcal{S}$ be a snake configuration with spine path P that has an empty cell in the tail section, and let $v \in P$ be a major vertex of the spine. A free module $\mathsf{m} \in N_2^*[v] \setminus S(P)$ can be teleported into $\mathcal{S}$'s interior by $\mathcal{O}(1)$ transformations, giving the snake ownership of it.*

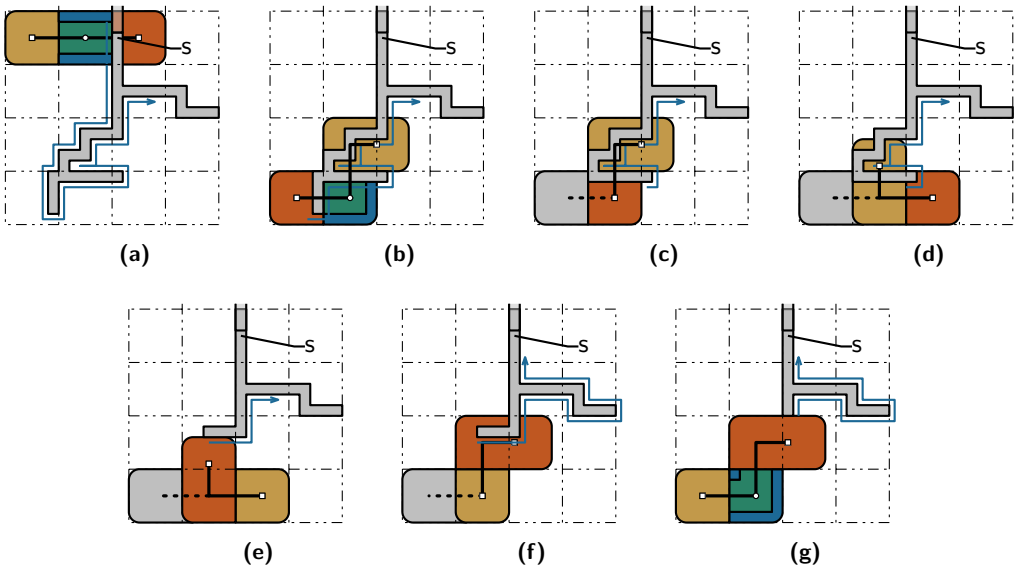


Figure 12 Example of our DFS procedure. In (a)–(b), the snake moves to a leaf of T_{s}, which it consumes before forking and pushing to the next meta-cell on its path in (c)–(d). This is repeated in (e)–(f), after which the forked spine is joined with the previous component in (g).

Algorithm 1 Procedure for growing a snake by traversing T_{s}.

Input: A snake configuration with spine $P = v_h, \ldots, v_t$ and a sequence $\mathsf{m}_1, \ldots, \mathsf{m}_\ell$ corresponding to the Euler tour of T_{s} rooted at m_1 such that $\mathsf{m}_1 \in N_2^*[v_h]$.
Output: A schedule of makespan $\mathcal{O}(\ell)$ that gives the snake ownership of the sequence.

for $i = 1$ to ℓ **do**
  if $\mathsf{m}_i \notin N_2^*[P]$ **then**
    $v_a \leftarrow$ the center of a meta-cell adjacent to m_i.
    if v_a is closer to the tail than to the head **then** `reverse` P (Lemma 9).
    `fork` such that v_a is the new head.
    $v_{\mathsf{m}_i} \leftarrow$ the center of the meta-cell containing m_i.
    while head vertex is not v_{m_i} **do**
      `push/pull` toward v_{m_i}.
      `join` with an abandoned prefix of the snake if this is possible.
  if m_i is a leaf of T_{s} **then**
    `take ownership` of m_i (either $\mathsf{m}_i \in S(P)$ or use Lemma 14).

The high-level idea for the depth-first traversal process is then to move the snake to the meta-cell containing a leaf of the tree, which it takes ownership of. If the next module in the DFS sequence is adjacent to neither the head nor the tail section, the snake needs to fork at an appropriate major vertex. We keep track of the resulting "cast off" spine paths that connect many tails with a single head; only one path in this tree is an active snake at all times. A description of this process is given in Algorithm 1 and illustrated in Figure 12.

5.3 Scaffold and compress

Due to space constraints, we refer to the full version [3] for a detailed description of this phase. Figure 13 provides intuition. After growing the snake to an appropriate size, we can move to the top of C, and then build the projection of C on this axis $z[C]$, this is our "scaffolding". We turn this projection into a large set of snakes, all of which move towards the bottom of C, taking ownership of the remaining modules C. This turns C into a series of vertical snakes, which we then turn into a compact configuration.

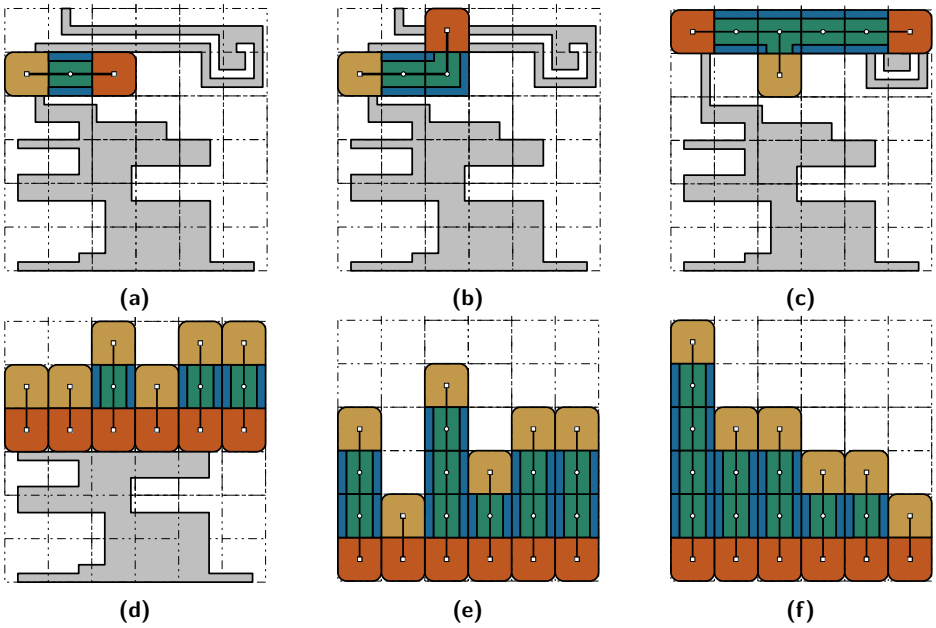


Figure 13 An overview of Scaffold and Compress. (a) We have a snake of size $\mathcal{O}(A+h)$. (b) Move the snake to the top of C. (c) Use Algorithm 1 to fill $z[C]$. (d-e) Compress the configuration with snakes. (f) Move the resulting towers towards the origin, compacting C.

6 Conclusion

We have provided several complexity and algorithmic results for Sliding Cubes in three dimensions. In particular, we proved that this optimal parallel motion is para-NP parameterized by makespan and symmetric difference size, as well as log-APX-hard in general. This is the first such inapproximability result among related models and emphasizes the relevance of input-sensitive, worst-case optimal algorithms; one such algorithm was also presented here. It also motivates future research into more efficient algorithms for instances of meta-modules, for which asymptotic optimality is known to be achievable in related models [12, 14].

Furthermore, the results of Section 4 motivate the investigation Sliding Cubes between obstacles that provide connectivity but cannot be moved. Since we wanted to keep the presentation as simple as possible, the bound for Lemma 5 can likely be improved to $\mathcal{O}(|S|^2)$. Moreover, the techniques used for our complexity results do not extend to lower-dimensional settings: Is there a significant leap between two and three dimensions?

References

1 Zachary Abel, Hugo A. Akitaya, Scott Duke Kominers, Matias Korman, and Frederick Stock. A universal in-place reconfiguration algorithm for sliding cube-shaped robots in a quadratic number of moves. In *Symposium on Computational Geometry (SoCG)*, pages 1:1–1:14, 2024. doi:10.4230/LIPIcs.SoCG.2024.1.

2 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. In *Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, pages 4:1–4:19, 2022. doi:10.4230/LIPIcs.SWAT.2022.4.

3 Hugo A. Akitaya, Joseph Dorfer, Peter Kramer, Christian Rieck, Gabriel Shahrouzi, and Frederick Stock. Sliding cubes in parallel, 2026. doi:10.48550/arXiv.2603.08537.

4 Hugo A. Akitaya, Sándor P. Fekete, Peter Kramer, Saba Molaei, Christian Rieck, Frederick Stock, and Tobias Wallner. Sliding squares in parallel. In *European Symposium on Algorithms (ESA)*, pages 28:1–28:17, 2025. doi:10.4230/LIPIcs.ESA.2025.28.

5 Hugo A. Akitaya, Matias Korman, and Frederick Stock. Input-sensitive reconfiguration of sliding cubes. In *Canadian Conference on Computational Geometry (CCCG)*, pages 216–222, 2025. URL: https://cccg.ca/proceedings/2025/CCCG2025-Proceedings.pdf#doc5B4.3.

6 Gruia Călinescu, Adrian Dumitrescu, and János Pach. Reconfigurations in graphs and grids. *SIAM Journal on Discrete Mathematics*, 22(1):124–138, 2008. doi:10.1137/060652063.

7 Mark de Berg and Amirali Khosravi. Optimal binary space partitions for segments in the plane. *International Journal on Computational Geometry and Applications*, 22(3):187–206, 2012. doi:10.1142/S0218195912500045.

8 Erik D. Demaine, Sándor P. Fekete, Phillip Keldenich, Henk Meijer, and Christian Scheffer. Coordinated motion planning: Reconfiguring a swarm of labeled robots with bounded stretch. *SIAM Journal on Computing*, 48(6):1727–1762, 2019. doi:10.1137/18M1194341.

9 Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Symposium on Theory of Computing (STOC)*, pages 624–633, 2014. doi:10.1145/2591796.2591884.

10 Adrian Dumitrescu and János Pach. Pushing squares around. *Graphs and Combinatorics*, 22(1):37–50, 2006. doi:10.1007/s00373-005-0640-1.

11 Adrian Dumitrescu, Ichiro Suzuki, and Masafumi Yamashita. Motion planning for metamorphic systems: feasibility, decidability, and distributed reconfiguration. *Transactions on Robotics*, 20(3):409–418, 2004. doi:10.1109/TRA.2004.824936.

12 Sándor P. Fekete, Phillip Keldenich, Ramin Kosfeld, Christian Rieck, and Christian Scheffer. Connected coordinated motion planning with bounded stretch. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 9:1–9:16, 2021. doi:10.4230/LIPIcs.ISAAC.2021.9.

13 Sándor P. Fekete, Phillip Keldenich, Ramin Kosfeld, Christian Rieck, and Christian Scheffer. Connected coordinated motion planning with bounded stretch. *Autonomous Agents and Multi-Agent Systems*, 37(2):43, 2023. doi:10.1007/S10458-023-09626-5.

14 Sándor P. Fekete, Peter Kramer, Christian Rieck, Christian Scheffer, and Arne Schmidt. Efficiently reconfiguring a connected swarm of labeled robots. *Autonomous Agents and Multi-Agent Systems*, 38(2):39, 2024. doi:10.1007/S10458-024-09668-3.

15 Robert Fitch, Zack J. Butler, and Daniela L. Rus. Reconfiguration planning for heterogeneous self-reconfiguring robots. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 2460–2467, 2003. doi:10.1109/IROS.2003.1249239.

16 Ferran Hurtado, Enrique Molina, Suneeta Ramaswami, and Vera Sacristán. Distributed reconfiguration of 2d lattice-based modular robotic systems. *Autonomous Robots*, 38(4):383–413, 2015. doi:10.1007/S10514-015-9421-8.

17 Irina Kostitsyna, Tim Ophelders, Irene Parada, Tom Peters, Willem Sonke, and Bettina Speckmann. Optimal in-place compaction of sliding cubes. In *Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, pages 31:1–31:14, 2024. doi:10.4230/LIPIcs.SWAT.2024.31.

18 Keith D. Kotay and Daniela L. Rus. Algorithms for self-reconfiguring molecule motion planning. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 2184–2193, 2000. doi:10.1109/IROS.2000.895294.

19 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.

20 Joel Moreno and Vera Sacristán. Reconfiguring sliding squares in-place by flooding. In *European Workshop on Computational Geometry (EuroCG)*, pages 32:1–32:7, 2020. URL: https://computational-geometry.org/abstracts/eurocg/2020.pdf.

21 An Nguyen, Leonidas J. Guibas, and Mark Yim. Controlled module density helps reconfiguration planning. *New Directions in Algorithmic and Computational Robotics*, pages 23–36, 2001.

22 Daniel Ratner and Manfred Warmuth. The $(n^2 - 1)$-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10(2):111–137, 1990. doi:10.1016/S0747-7171(08)80001-6.

23 Matthijs Wolters. Parallel algorithms for sliding squares. Master's thesis, Utrecht University, 2024.