Robotica (2011) volume 29, pp. 59–71. © Cambridge University Press 2011
doi:10.1017/S026357471000072X

Efficient constant-velocity reconfiguration of crystalline robots**

Greg Aloupis†*, Sébastien Collette‡, Mirela Damian§, Erik D. Demaine¶, Robin Flatland‖, Stefan Langerman††, Joseph O'Rourke‡‡, Val Pinciu§§, Suneeta Ramaswami¶¶, Vera Sacristán‖‖ and Stefanie Wuhrer†††

†*Université Libre de Bruxelles, Belgique. E-mail: aloupis.greg@gmail.com* (*Supported by the Communauté française de Belgique - ARC*)
‡*Chargé de Recherches du FRS-FNRS, Université Libre de Bruxelles, Belgique. E-mail: sebastien.collette@ulb.ac.be*
§*Villanova University, Villanova, PA, USA. E-mail: mirela.damian@villanova.edu*
¶*Massachusetts Institute of Technology, Cambridge, MA, USA. E-mail: edemaine@mit.edu* (*Partially supported by NSF CAREER award CCF-0347776, DOE grant DE-FG02-04ER25647, and AFOSR grant FA9550-07-1-0538*)
‖*Siena College, Loudonville, NY, USA. E-mail: flatland@siena.edu*
††*Maître de Recherches du FRS-FNRS, Université Libre de Bruxelles, Belgique. E-mail: stefan.langerman@ulb.ac.be*
‡‡*Smith College, Northampton, MA, USA. E-mail: orourke@cs.smith.edu*
§§*Southern Connecticut State University, New Haven, CT, USA. E-mail: pinciu@scsu.ctstateu.edu*
¶¶*Rutgers University, Camden, NJ, USA. E-mail: rsuneeta@camden.rutgers.edu.* (*Partially supported by NSF grant CCR-0204293*)
‖‖*Universitat Politècnica de Catalunya, Barcelona, Spain. E-mail: vera.sacristan@upc.edu* (*Partially supported by projects MCI MTM2009-07242 and Gen. Cat. DGR 2009SGR1040*)
†††*Carleton University, Ottawa, Canada. E-mail: swuhrer@scs.carleton.ca*

(Received in Final Form: October 18, 2010)

SUMMARY

In this paper, we propose novel algorithms for reconfiguring modular robots that are composed of n atoms. Each atom has the shape of a unit cube and can expand/contract each face by half a unit, as well as attach to or detach from faces of neighboring atoms. For universal reconfiguration, atoms must be arranged in $2 \times 2 \times 2$ modules. We respect certain physical constraints: each atom reaches at most constant velocity and can displace at most a constant number of other atoms. We assume that one of the atoms has access to the coordinates of atoms in the target configuration.

Our algorithms involve a total of $O(n^2)$ atom operations, which are performed in $O(n)$ parallel steps. This improves on previous reconfiguration algorithms, which either use $O(n^2)$ parallel steps[10,25,30] or do not respect the constraints mentioned above.[3] In fact, in the settings considered, our algorithms are optimal. A further advantage of our algorithms is that reconfiguration can take place within the union of the source and target configuration space, and only requires local communication.

KEYWORDS: Control of robotic systems; Mobile robots; Modular robots; Motion planning; Path planning.

**A short version appeared at WAFR 2008,[2] with title *Realistic reconfiguration of crystalline (and telecube) robots*
*Corresponding author. E-mail: aloupis.greg@gmail.com

1. Introduction

1.1. *Self-reconfiguring modular robots*

Robots designed with inflexible structures tend to have a unique purpose. They can be very efficient but often lack versatility, in the sense that they might not perform unexpected tasks efficiently, or might have problems adapting to new environments.

For this reason, since the late 1980's, much research has been concentrated on the design of modular robots that can self-reconfigure.[14] Modular robots are theoretically capable of reaching any shape that has the same mass/volume (restricted to the size of their finer components, or modules). Thus, they not only become capable of seemingly limitless uses but also have the ability to self-repair (by replacing damaged modules), and navigate through new environments.

Several new prototypes of modular robots appear each year. Developers strive to solve hardware design challenges involving strength, precision, bonding, energy efficiency, and flexibility of modules. An important goal is also to reduce module sizes. On the other hand, a significant problem in the field is to design efficient algorithms for self-reconfiguration.

Modular robots are often classified into distinct categories. Crystalline robots, which we deal with in this paper, are an example of a cube *lattice model*. Early square and hexagonal lattice examples, as well as 3-dimensional rectangular grid robots, are found in refs. [12, 15, 21, 22, 29]. Recent efforts have been made to define and characterize lattice robots.[6] Examples of *chain robots* are *Polypod* and *Polybot*.[32,33] Finally, one can also find mixed (chain and lattice) models,

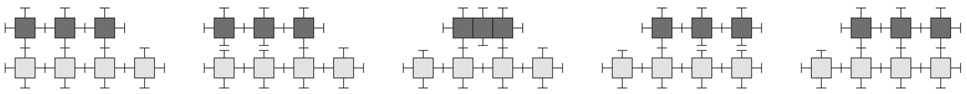
Fig. 1. Example of reconfiguring Crystalline atoms.

such as *M-Tran*.[35] Various types of self-reconfiguring robots, as well as related algorithmic issues, are surveyed in refs. [20, 34]. In this paper, we focus on the (modular) *crystalline*[8,25] and *telecube*[28] robots, which are designed on a square lattice.

1.2. *Crystalline and telecube robots*

The atoms* of these robots are cubic in shape, and are arranged in a grid configuration. Each atom is equipped with mechanisms allowing it to extend each face out one unit and later retract it back. Furthermore, the faces can attach to or detach from faces of adjacent atoms; at all times, the atoms should form a connected unit. The default configuration for a Crystalline atom has expanded faces, while the default for a Telecube atom has contracted faces.

When groups of atoms perform the four basic atom operations (expand, contract, attach, and detach) in a coordinated way, the atoms move relative to one another, resulting in a reconfiguration of the robot. Fig. 1 shows an example of a reconfiguration.

To ensure that all reconfigurations are possible, atoms must be arranged in $k \times k \times k$ modules, where $k \geq 2$.[3,30] In the 2D setting that we focus on, we assume that modules consist of 2×2 atoms. Our algorithms can easily be extended to 3D setting.

We refer the reader to refs. [3, 25, 30] for a more detailed introduction to these robots.

1.3. *The model*

The problem we solve is to reconfigure a given connected source configuration of n modules to a specified, arbitrary, and connected target configuration T in $O(n)$ parallel steps. We allow modules to exert only a constant amount of force, independent of n. In particular, each module has the ability to push/pull one other module by a unit distance (the length of one module) within a unit of time. Simply bounding the force may still lead to arbitrarily high velocities and thus rather unrealistic motions. On the other hand, in some situations where maximal control is desired (e.g., treacherous conditions, dynamic obstacle environment, and minimally stable static configuration of the robot itself), it may be desirable to strictly limit velocity. Thus, we also bound maximum velocity (and so the momentum) by a constant (module length/unit time). One important advantage of our algorithm is that we can rely on local communication between atoms. That is, atoms can decide to move based only on the state of neighbors and instructions received from them.

*In the literature, the term “module" is often used to describe an individual hardware unit, and small collections of such units are referred to as *meta-modules*. Individual cubic components in the Crystalline model have also been named *atoms* by their designers. We will avoid using the term *meta-module* entirely. Instead we will call individual cubic components *atoms*, and larger groups will be referred to as *modules*.

Our model permits separating algorithmic issues from design considerations, at the same time as retaining the fundamental physical constraints of velocity and force. We assume that if a module decides to move a short distance and dock with another module, it will do so accurately and within a specific allowed time. Our algorithms are susceptible to such issues as motion uncertainties and errors, since we perform many parallel operations. It may be possible to limit such errors by only allowing a constant number of simultaneous operations, or motions along the perimeter of the robot. Such approaches have been used before, but at the cost of increased time complexity. Our approach takes advantage of the Crystalline model, which permits traveling through the interior of the robot, something which has been done in many previous reconfiguration algorithms (see the Section 2).

Under the constant-velocity model, there exist configurations that require a quadratic number of overall basic moves to reach their goal. To attain the linear bound proved in this paper, one must rely on (more physically unstable) massively parallel algorithms. Having said this, we note that any instability that an atom may have (for instance, a docking delay) remains a local problem in our algorithms: moving components are isolated into small individual groups, each traveling on a fixed substrate frame. A mechanical problem may create a bottleneck, or “traffic jam," but this should only affect the time-complexity, not the mechanical difficulty of recovering. In other words, a problem encountered by one module should not cause any problems to another module.

Our algorithms rely on “parallel tunneling," which has been used in the *PacMan* algorithm.[7,10] The *PacMan* designers showed a tunneling experiment and discussed the troubles encountered in such physical experiments, such as misalignments that may cause problems for atom connections. In order to improve this issue, a new Crystal connector was proposed,[8] allowing a significant amount of lateral and vertical misalignment.

Our algorithms are designed for robots. Many other modules have been and continue to be designed for reconfigurable robots. Wikipedia lists 34 modules designed as of 2010.** Some of these modules are equivalent to Crystalline modules from the point of view of our model, and some are not. We make no attempt at a comprehensive survey. In Section 5 we discuss how our algorithms can be adjusted to work for one closely related model, the Telecube robots. In that section we also mention how our algorithm can be altered to apply to robots made of other types of modules.

1.4. *Related results*

The optimal number of moves between configurations has been studied in the context of centralized reconfiguration planning (see refs. [11, 23]). The centralized reconfiguration algorithms have been proposed for many systems (e.g., see

**`http://en.wikipedia.org/wiki/Self-Reconfiguring_Modular_Robotics`.

refs. [17, 35]). The distributed algorithms to reconfigure certain classes of hexagonal modular robot configurations are given in ref. [31]. Typically, such algorithms employ heuristics or work for particular classes of target shapes. Of particular interest are distributed algorithms in which each module actuator is based on local information received from immediate neighbors (e.g., see refs. [9, 19, 26, 27]).

Algorithms for reconfiguring Crystalline and Telecube robots in $O(n^2)$ parallel steps have been given in refs. [7, 10, 25, 30]. The quadratic bound is also implied in ref. [13], which deals with the reconfigurations of a specific class of modular robots (more restrictive than Crystalline). An algorithm that uses $O(n)$ parallel steps for reconfiguring a robot within the bounding box of source and target configurations was given in ref. [3]. The total number of individual moves is also linear. However, no restrictions were made concerning physical properties of the robots. For example, $\Theta(n)$ strength is required, since modules can carry tall towers and push large masses during certain operations. An $O(\log n)$ parallel step algorithm for 2D robots that uses a total of $O(n \log n)$ atom moves and also stays within the bounding box is given in ref. [4], and this has recently been extended to 3D.[5] However, in this algorithm, not only are modules assumed to have $\Theta(n)$ physical strength but they can also reach $\Theta(n)$ velocity. An $O(\sqrt{n})$ time algorithm for 2D robots, using the third dimension as an intermediate, is given in ref. [24]. This is optimal in the model considered, which permits linear velocities, but only constant acceleration. If applied within the model used in ref. [4], this algorithm would run in constant time. We remind the reader that, unlike refs. [3, 4, 24], we limit force and velocity to a constant level.

1.5. Contributions of this paper

We present two algorithms to reconfigure Crystalline robots in $O(n)$ time steps, using $O(n)$ parallel moves per time step. Our first algorithm (Section 3) is slightly simpler to describe. It also forms the basis of our second algorithm (Section 4), which is exactly *in-place*, i.e., it uses only the cells of the union of the source and target configurations. This is particularly interesting if there are obstacles in the environment. Both algorithms consider the given robot as a spanning tree, and push leaves toward the root with "parallel tunneling." This form of tunneling has been used in the *PacMan* algorithm,[7, 10] which also works in-place but does not guarantee linear time complexity. In our algorithm, no global communication is required. This means that constant-size memory suffices for each non-root module, which can decide how to move at each step based solely on the states of its neighbors. In the realistic model considered in this paper, our algorithms are *worst-case optimals*: transforming a horizontal row of modules to a vertical row requires a linear number of parallel moves and a quadratic number of total operations. Our first algorithm can be adapted for use with Telecube modules. However, an adaptation of our in-place algorithm seems to require more memory for non-root modules. This is discussed in Section 5. Our results are not accompanied by simulations or experiments, but from the viewpoint of mechanics and electronics, we are only requiring atoms to perform basic operations similar to what appears in the experiments reported in ref. [10].

2. Primitive Operations

We restrict our descriptions to a 2D lattice whose cell size equals the size of one robot module (i.e., 2×2 connected atoms in their expanded state). None of our techniques depend on dimension, so it is straightforward to extend to 3D robots. Given the 2×2 module size, a cell of the lattice can potentially contain two compressed modules (see Fig. 2(b)). Cells can be marked with an integer as $\{0, 1, 2\}$: a 0-cell corresponds to a node in T that has no module yet, a 1-cell contains one module, and a 2-cell contains two (compressed) modules. In a 2-cell, we sometimes distinguish between the *host* module and the *guest* module: the host module is the one occupying the 1-cell prior to becoming a 2-cell; the guest module is the one compressing itself into the 1-cell occupied by the host module, thus turning the cell into a 2-cell (see Fig. 2(a)).

Let r_0 be a specialized module that has access to a map of the target configuration, T. We compute a spanning tree S of the source configuration, rooted at r_0, and instruct modules to form attachments corresponding to tree edges in S. The spanning tree can be computed in linear time and constructed via local communication. The tree structure between cells is maintained throughout the algorithm by physical connections between the host modules; note that the guest modules are irrelevant in determining S. These modules are also responsible for the parent–child pointer structure of the tree. For each node $u \in S$, let $P(u)$ denote the parent of u in S (recall that both u and $P(u)$ are the host – not the guest – modules). A child of a cell u is adjacent either on the east, north, west, or south side of u. Let the *highest priority child* of u be the first child in counterclockwise order starting with the east direction.

Let m and q be adjacent cells. We define the following primitive operations (illustrated in Fig. 2); a cell is meant to be engaged in at most one operation at any time:

1. PUSHINLEAF(m, q): It applies when $q = P(m)$, m is a leaf, and both are uncompressed. Here, m becomes empty and q becomes compressed (i.e., q takes the module of m as a guest).
2. POPOUTLEAF(m, q): It applies when q is compressed and m is empty. This is the inverse of the PUSHINLEAF operation.
3. TRANSFER(m, q): It applies when m is compressed and q is non-empty; if q is compressed, then the guests of both cells exchange positions physically. Otherwise, the guest of m moves into (and becomes a guest of) q.
4. ATTACH(m, q): It applies when the host modules in m and q are unattached. The two modules make a physical connection.
5. DETACH(m, q): It applies when the host modules in m and q are attached. The two modules break their physical connection.
6. SWITCH(m): It applies when m is compressed. Its two modules switch positions physically (and the roles of host and guest).

In the remainder of this paper, we assume that all parallel motions are synchronized. However, due to the simple hierarchical tree structure of our robots, we find it plausible that our algorithms could be implemented so that modules may operate asynchronously.

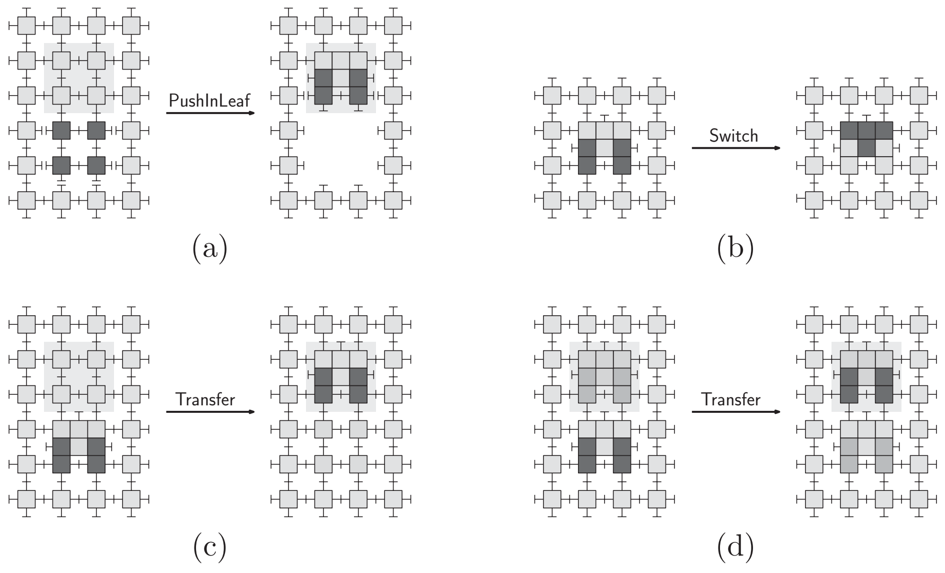

Fig. 2. (a) PUSHINLEAF. Cell q has a shaded background. In POPOUTLEAF, which is the inverse of this diagram, q would be the empty cell; (b) SWITCH: guest and host exchange their roles and positions; (c) TRANSFER, when only one module is compressed, cell q is shaded; (d) TRANSFER, when both modules are compressed. Only initial and final configurations are shown. Note that neighboring cells are not completely drawn.

Lemma 1. *Operations* PUSHINLEAF, POPOUTLEAF, SWITCH, *and* TRANSFER *maintain the tree structure of a robot and can be executed in* $O(1)$ *time.*

Proof. When each of the first three operations is performed individually, the robot remains connected. For example, in PUSHINLEAF(m, q), the atoms of the host module q temporarily displace in order to let those of m occupy intermediate spaces. However, at all times one of the two atoms along every face of q (other than the one adjacent to m) will remain in its original position so that connectivity is ensured; this property can be verified in Fig. 7 in the Appendix. Note that the activity within the guest module m does not affect connectivity in the rest of the robot.

This temporary displacement can cause connectivity issues if the modules neighboring the host module q are also performing basic operations. To avoid any problems, we can subdivide one basic time unit into four subunits so that each host module acts when it has the right parity of row and column. For example, in the first time subunit, modules located in (odd row, odd column) lattice cells are allowed to reconfigure; in the second time unit (odd row, even column) modules reconfigure; and so on.

This way, when a leaf pushes into its parent, we can ensure that no other cells adjacent to the parent are active. This issue is even simpler to resolve in 3D, where the actuation at the atomic level can be done by 2D layers.

Now consider the TRANSFER operation. In this operation, two adjacent cells interact and it is best not to let any of the six neighboring cells perform basic operations simultaneously. A similar lattice parity solution can be applied.

Regarding the tree structure, it is straightforward to see that SWITCH and TRANSFER only affect module positions, while PUSHINLEAF removes one leaf module from S and POPOUTLEAF adds one leaf module to S.

As for the complexity of the primitive operation, note that any permutation of atoms within a lattice cell containing two modules can be realized in linear time with respect to the size of the module (i.e., $O(1)$ time for our modules). Thus, a compressed cell may transfer or push one module to any direction, and two modules within a cell can switch roles in $O(1)$ time. □

In our basic motions, modules move by one unit length per time step. The two modules involved in a primitive operation do not carry other modules. Thus, our reconfiguration algorithms place no additional force constraints beyond those required by *any* reconfiguration algorithm.

2.1. Details of the primitive operations

Refer to Figs. 7–11, located in the Appendix. Figure 7 shows the finite sequence of basic atom operations (attach, detach, compress, and expand) that implement PUSHINLEAF and POPOUTLEAF, which are inverse operations.

TRANSFER(m, q) can be described as a two-step operation. In the first step, module m is rotated within its lattice cell in order to face module q. We call this the *positioning* step. In the second step, the module m is actually sent to the lattice cell of q. Depending on whether q was compressed or uncompressed, we call this the *send* or the *exchange* step. Figure 8 shows the finite sequence of basic atom operations (attach, detach, compress, and expand) that produces the *positioning* step. Figures 9 and 10 show the analogous sequence for the *send* and the *exchange* steps. TRANSFER is the result of appropriately concatenating these steps.

SWITCH is a particular permutation of atoms within a lattice cell containing two modules, which can be realized in linear time with respect to the size of the module. Figure 11 shows the details of the SWITCH operation.

3. Reconfiguration via Canonical Form

This section describes an algorithm to reconfigure S into T via an intermediate canonical configuration. Modules follow a path directly to the root r_0, and into a canonical "storage configuration." We focus on the construction of one type of canonical form, a vertical line V. In fact, V could be any path that avoids the source configuration. Thus, the entire reconfiguration can take place relatively close to the bounding box of S. Reconfiguring from V to T is nearly the inverse procedure and is relatively straightforward. Each module m passes from the canonical form through r_0. It suffices for r_0 to provide m with just a few bits of information, indicating where m should have children. If we can afford to let m store $O(\log n)$ bits, then we can even specify the size of the subtrees rooted at m (this helps heuristically, and is described at the end of Section 3). For this task, it is assumed that r_0 has access to a map of T (perhaps stored in memory, or via direct communication with some external processor).

3.1. *Algorithmic details*

Our algorithm reconfigures S into a vertical strip V that begins at the maximum y-coordinate of S. We first move r_0 to a maximum possible y-coordinate: this involves pushing in a leaf and iteratively transferring it to r_0, so that r_0 becomes part of a 2-cell and is then able to iteratively transfer to a module of maximum y-coordinate. Note that this step might not be necessary in implementations in which all modules are capable of playing the role of r_0 (e.g., if all modules have a map of T, or if all are capable of communicating to an external processor). This initial step is followed by two main phases, during which r_0 does not move.

In the first phase, we repeatedly apply procedure ClusterStep to move modules closer to r_0. This is done by compressing leaves into their parents, and moving up S in parallel. The shape of S shrinks during this procedure, as PushInLeaf operations in ClusterStep compress leaf modules into their parent cells. At the end of this phase, all *non-leaf* cells will become 2-cells. We refer to S in this state as being *fully compressed*. It is not critical that all cells become compressed; in fact, we can proceed to the next phase as soon as the root becomes part of a 2-cell. The restriction for S being fully compressed at the end of this phase will merely simplify our analysis of the total number of parallel steps in our algorithm.

```
ClusterStep($S$)

For all cells $u$ in $S$ except for that containing $r_0$, execute
the following in parallel:
  If $P(u)$ is a 1-cell.
        If $u$ is the highest priority child of $P(u)$ and
        all siblings of $u$ are leaves or 2-cells.
            If $u$ is a 1-cell leaf then PushInLeaf($u$, $P(u)$).
            If $u$ is a 2-cell, then Transfer($u$, $P(u)$).
```

```
SourceCluster($S$)

Repeat until $S$ is fully compressed
        ClusterStep($S$).
```

The procedure SourceCluster is illustrated in Fig. 3. The task of compressing a parent cell $P(u)$ falls onto its highest priority child, u. Note that $P(u)$ first becomes compressed only when all its subtrees are essentially compressed. That is, even if u is ready to supply a module to $P(u)$, it waits until all other children are also ready. This rule could be altered, and in fact the whole process would then run slightly faster. Here, we ensure that once the root of a subtree becomes compressed, it will be able to supply a steady stream of guest modules to its ancestors.

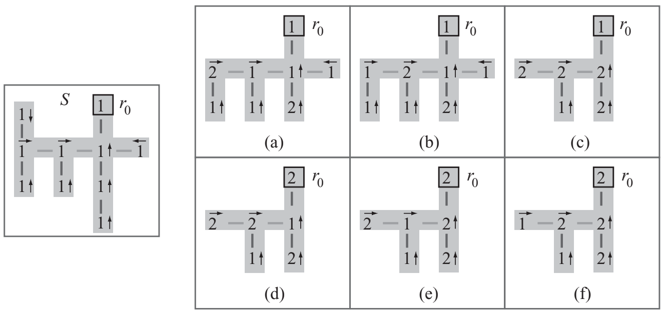

Fig. 3. An example of SourceCluster. Arrows indicate the direction of the Transfer and PushInLeaf operations. After steps (a) through (e), the source configuration (left) becomes fully compressed (f).

In the second phase, we construct V while emptying S, one module at a time. This is described in the second step of algorithm TreeToPath, and is illustrated in Fig. 4.

```
Algorithm TreeToPath($S$, $V$)

1.  SourceCluster($S$)
2.  Let $d$ be the cell containing $r_0$ as a *host*. Let $V = \{d\}$.
    Repeat until $V$ contains all modules:
        (a) For all 2-cells $u$ in $V$, execute in parallel:
              Let $c$ be the cell vertically above $u$.
              If $c$ is empty, PopOutLeaf($u$, $c$);
              Otherwise, if $c$ is a 1-cell, Transfer($u$, $c$).
        (b) ClusterStep($S$).
```

Lemma 2. *If S is a set of modules physically connected in a tree of cells, then* ClusterStep(S) *returns a tree containing the same set of modules, while maintaining connectivity. So does* SourceCluster(S).

Proof. ClusterStep invokes two basic operations, PushInLeaf and Transfer. By Lemma 1, these operations maintain a tree independently. Since u is involved in PushInLeaf or Transfer (with $P(u)$) only if it is a 1-cell leaf or a 2-cell respectively, but also only when $P(u)$ is a 1-cell, we know that $P(u)$ is not involved in any other such operation in parallel. Therefore every cell is involved in at most one basic operation at a time. The claim also follows immediately for SourceCluster. □

Define the *height* of a cell in S to be the longest path to a leaf in its subtree, plus 1. By convention, leaves have height 1.

Lemma 3. *Let r be a cell in S with height $h \geq 2$. In iteration $h-1$ of* SourceCluster(S), *r becomes a 2-cell for the first time.*

Proof. Prior to the first iteration, S contains only 1-cells. The proof is by induction on h. For the base case when $h = 2$, the children of r are leaves. Therefore, in the first iteration,

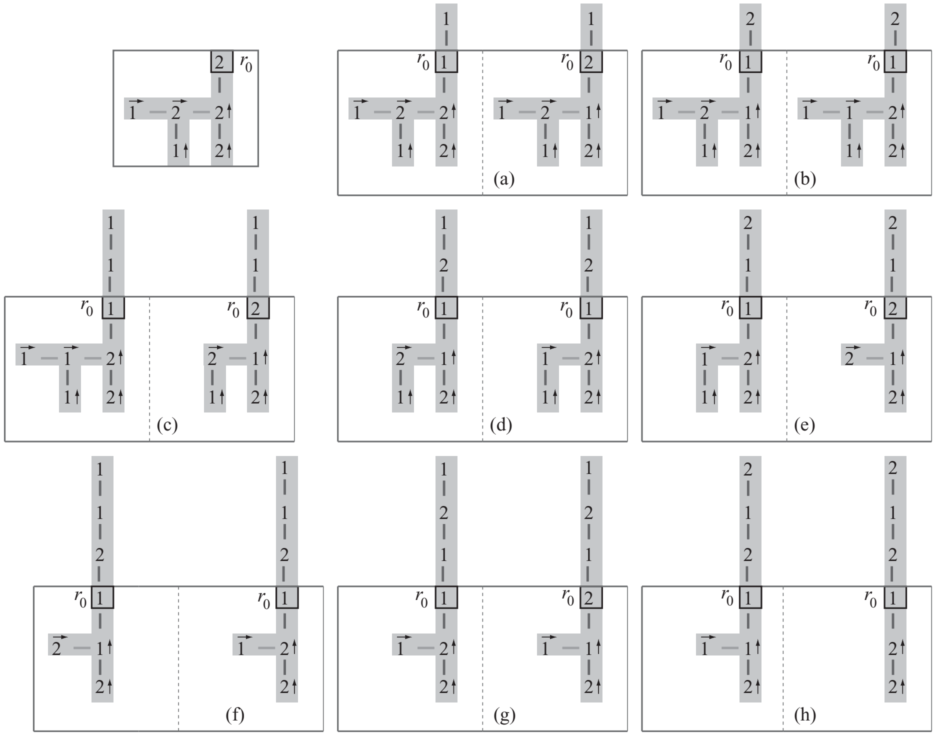

Fig. 4. An example of TREETOPATH. For each step, the two phases (POPOUTLEAF or TRANSFER, and CLUSTERSTEP) are shown.

during CLUSTERSTEP, the highest priority leaf compresses into r.

Now assume inductively that the lemma is true for all subtrees of height smaller than h. Cell r must have at least one child c with height $h-1$. By the inductive hypothesis, c becomes a 2-cell in iteration $h-2$, and all of r's other than non-leaf children are 2-cells by the end of iteration $h-2$. Therefore, at iteration $h-1$, for the first time the conditions are satisfied for r to receive a module from its highest priority child during CLUSTERSTEP. □

Lemma 4. *In iteration i of* SOURCECLUSTER, *let r be a 2-cell with height h that transfers its guest module to $P(r)$. Then at the end of iteration $i+1$, r is either a leaf or a 2-cell again.*

Proof. First note that at the beginning of iteration $i+1$, r is a 1-cell and $P(r)$ is a 2-cell. Thus, if r is a leaf after iteration i, it remains so. On the other hand if r has children ($h \geq 2$), it will become a 2-cell. We prove this by assuming inductively that our claim holds for all heights less than h. Consider the base case when $h = 2$. At the end of iteration i, all children of r are still leaves and thus one will compress into r (note that r might *also* become a leaf in this particular case).

For $h > 2$, consider the iteration $j < i$, in which r received the guest module that it later transfers to $P(r)$ in iteration i. At the beginning of iteration j, all of r's children were leaves or 2-cells, because that is a requirement for r to receive a guest. Let c be the child that passed the module to r. If c used the PUSHINLEAF operation, then at the end of iteration j, r has one fewer children (but at least one). The other children remain leaves or 2-cells until iteration $i+1$, when r again becomes a 1-cell. Thus, in iteration $i+1$, conditions are set for r to receive a module.

On the other hand, if c used the TRANSFER operation, we apply the inductive hypothesis: at the end of iteration $j+1 \leq i$, c is either a leaf or a 2-cell. During iterations j and $j+1$, in which r is busy receiving or transferring a module, all other children of r (if any) remain leaves or 2-cells. Therefore, in iteration $j+2 \leq i+1$, the conditions are set for r to receive a module. □

Define the *depth* of a cell in a tree to be its distance from the root. Hence, the root has zero depth. Let the root of S be at height h_0.

Lemma 5. SOURCECLUSTER *terminates after at most* $2h_0-1$ *iterations of* CLUSTERSTEP.

Proof. We claim that at the completion of iteration h_0-1+d of SOURCECLUSTER, all non-leaf modules at depth less than or equal to d in S are 2-cells (here, we use S to refer to the current instance of the dynamically changing tree, not the original S). The proof is by induction on d. The base case is the root of S at depth $d = 0$. By Lemma 3, the root becomes a 2-cell in iteration $h_0 - 1$. Assume inductively that our claim is true for all values of d', where $0 \leq d' < d$.

Now consider a cell p at depth $d-1$ that has children. By the inductive hypothesis, p and all its ancestors are 2-cells by the end of iteration $i = h_0 - 1 + (d-1)$, and p is the last of this group to become a 2-cell. Thus, at the beginning of iteration i, all children of p are either leaves or 2-cells. During iteration i, only p's highest priority child c changes, either by transferring a guest module into p (if c is a 2-cell), or by pushing into p (if c is a 1-cell leaf). In the first case, by Lemma 4, c will be a 2-cell leaf or a leaf by the end of iteration $i+1$. In the second case, c is not part of S anymore.

As p will not accept new guest modules after iteration i (because all its ancestors are 2-cells), all siblings of c remain leaves or 2-cells during iteration $i+1$. Thus, at the end of this iteration, our claim holds for depth d. By setting $d = h_0$, our result follows. □

Let a *long gap* consists of two adjacent 1-cells that are not leaves. A tree is *root-clustered* if it has no long gaps. Observe that a fully compressed tree is a special case of a root-clustered tree.

Lemma 6. *Let S be a root-clustered tree. Then after one application of* ClusterStep(S), *S remains root-clustered.*

Proof. This follows from claims in the proof of Lemma 4. Specifically, consider any 2-cell u. If ClusterStep keeps u as a 2-cell, then u is not part of a long gap. Otherwise, if u sends a module to $P(u)$, none of the children of u attempt to transfer a module to u. Now consider any 1-cell non-leaf child y of u. As there was no long gap in S, all children of y were either 2-cells or leaves. Thus y will become a 2-cell during this iteration of ClusterStep. Again we conclude that u cannot be part of a long gap. □

Theorem 1. *Algorithm* TreeToPath *terminates in linear time.*

Proof. By Lemma 5, SourceCluster terminates in linear time. In fact, by treating the final top position of V as an implicit root, our claim follows.

More specifically, however, we analyze the transition from S into V. When SourceCluster terminates, S is fully compressed (i.e., root-clustered), and we set r_0 to be the host in cell d.

First, we mention that all basic operations are performed legally, i.e., every cell is involved in at most one operation. In step 2a, every 2-cell is involved in an operation, only if the cell above it is not a 2-cell. Therefore, both cells are only involved in this operation. Step 2b is safe, by Lemma 2.

In step 2a, d sends a module to the empty position c vertically above, if c is not a 2-cell. We may treat the position c as $P(d)$, and consider step 2a to be synchronous to step 2b. In other words, d is the only child of c, and thus d follows the same rules as ClusterStep. In fact, since S is fully compressed, after the first iteration of phase 2, the tree rooted at c will be root-clustered (only c and the highest priority child of d will not be 2-cells). Therefore, by Lemma 6, in every iteration of phase 2, S remains root-clustered. Thus, in every even iteration, d supplies a module to c, and in every odd iteration d is given a module from one of its children. Informally, when d sends a module up into V, the gap (in the sense of lack of guest module) that is created in S travels down the highest priority path of S until it disappears at a leaf. In general, a guest module on the priority path will never be more than two steps away from d, following the analysis of Lemma 4. Within V, a stream of guest modules, two units apart, will move upward. One module will pop up into an empty cell, every three iterations. Thus, compressed modules in V are always able to progress. □

We now briefly discuss the reconfiguration from V to T. If we merely wish to construct the shape of T, then we can assume that V does not intersect the cells that will be occupied by T. Otherwise, if T must occupy specific cell coordinates, it is trivial to move V to a position where our assumption will hold.

For the construction of T, let us first assume that the memory of each module suffices to count to n. All modules from V pass through the root r_0 on their way to T. Once a module m reaches r_0, the root determines the position of m in T and supplies m with three values corresponding to the sizes of the three subtrees of m in T. Then r_0 transfers m to the highest priority child c of r_0 whose subtree is not full yet. The child c, in turn, transfers m to its own highest priority child not yet full and so on, until m encounters the conditions to pop into an empty cell in T, as directed by its host module. From that point on, m simply awaits modules transferred by its parent and directs them to its children according to our priority rules (counterclockwise starting with the east child). In order to decide where to send an incoming module, m keeps count of all modules passed through; this information, along with the information collected from r_0 (the sizes of its three subtrees) suffices for m to avoid sending an incoming module into a completed subtree.

If we do not have the luxury of allowing modules to count, we do the following. The root does not tell m the size of its subtrees, but instead it just tells m if it will have a subtree in each direction. Priority rules are followed, as before. The only difference is that when m reaches its final position, it will not be able to determine when its subtrees are full. Thus, each module performs an entire Depth-First Search of the partial structure of T. This involves backtracking, which can be dealt with via Transfer. We omit details here, since this backtracking issue appears again and is clarified in Section 4.

We remind the reader that our first phase need not terminate before the second commences. By compressing leaves and sending them toward the root, while simultaneously constructing V from the root whenever it becomes compressed, the target configuration will be constructed even faster. Splitting into two distinct phases simply helps with the analysis.

4. In-place Reconfiguration

This section describes an *in-place* algorithm that reconfigures S into T by restricting the movement of all modules to the space occupied by $S \cup T$ as long as they intersect. If S and T do not intersect, then we also use the cells on the shortest path between them. Our description assumes intersection. If all modules were to know which direction to take in each time unit (e.g., by having an external source that synchronously transmits instructions to each module individually), then it would not be difficult to design an in-place algorithm similar to the one in Section 3. However, since we impose the restriction that all modules are only capable of communicating locally, it is up to r_0 to direct all actions.

4.1. Overview

Our algorithm consists of two phases. The first phase is identical to phase 1 of the TreeToPath algorithm from Section 3 (i.e., clustering around the root).

In the second phase, r_0 carries out a depth-first search (DFS) walk on T, dynamically constructing portions that are not already in place. Apart from modules in cells adjacent to r_0 that receive its instructions, all other modules simply try to keep up with r_0 (i.e., they follow CLUSTERSTEP). Note that if r_0 is not initially inside T, it must first travel to such a position. At any time, this "moving root" will either be traveling through modules that belong to the partially constructed tree T, or will be expanding T beyond the current tree structure, using compressed modules that are tagging along close to r_0.

4.2. *Algorithmic details*

The INPLACERECONFIGURATION algorithm maintains a dynamically changing tree S, and a connected subset S_ℓ of that tree. Each cell u maintains two links: a *physical* link corresponding to the physical connection between u and $P(u)$ in S, and a *logical* link that is only present between adjacent nodes in S_ℓ. We call the tree S_ℓ induced by the logical links as the *logical tree*.

Tree S always contains all occupied cells. Tree S_ℓ is the smallest subtree of S that contains modules not in their final position in T. Thus, at the end of the algorithm, $S = T$ and $S_\ell = \emptyset$.

The full algorithm is summarized in the following:

```
Algorithm INPLACERECONFIGURATION($S$, $T$)

Phase 1.    $S \leftarrow$ SOURCECLUSTER($S$).

Phase 2.    $S_\ell \leftarrow S$.
            Repeat until $r_0$ reaches the final position in its
            DFS traversal:
                $S \leftarrow$ TARGETGROW($S$, $T$).
```

We continue with the description of the operation of TARGETGROW.

```
TARGETGROW($S$, $T$)

$d \leftarrow$ next cell in the DFS visit of $T$.
$c \leftarrow$ current 2-cell in which $r_0$ is a guest module.

{1. DFS Root Update }
    Mechanical/Physical Operations
    1.1. If $d$ is a 0-cell,
             POPOUTLEAF($c$, $d$).
    1.2. If $d$ is not a 0-cell,
             If $c \neq P(d)$,
                 ATTACH($c$, $d$) and DETACH($d$, $P(d)$).
             TRANSFER($c$, $d$).

    Tree Structure Update
    1.3. Set $P(c)$ to be $d$. Set $P(d)$ to NULL.
    1.4. Mark $c$ as "visited."
    1.5. Add edge ($c$, $d$) to $S_\ell$.

{2. Root Clustering }
    Until $c$ and $d$ become 2-cells, repeat:
        (a) Leaf prune: For all 1-cell leaves $u \in S_\ell$, execute
             in parallel:
                 If $u$ is marked "visited," remove $u$ from $S_\ell$.
        (b) CLUSTERSTEP($S_\ell$).
    If $r_0$ is not the guest in $c$, SWITCH($d$).
```

Note that in Phase 2, we start with $S_\ell = S$. Throughout this phase, the S_ℓ structure is identical to the S structure, with the exception that some branches of the tree S_ℓ are logically trimmed off through iterative leaf prune operations. The host module in each cell uses a bit to determine if the cell is also part of S_ℓ. Pointers between cells and their parents apply for both trees.

The main idea of the target growing phase is to move r_0 through the cells of T in a DFS order. A caravan of modules will follow r_0, providing a steady stream of modules to fill in empty target cells that r_0 encounters. The algorithm repeats the following main steps:

1. DFS root update: r_0 is the guest of 2-cell c and is ready to depart. It marks c as "visited" (i.e., c now belongs to T). Then r_0 moves to the next cell d encountered in the DFS walk of T. This is accomplished either by uncompressing (popping) r_0 into d (see Figs. 5(a $\rightarrow$ b)), or by transferring r_0 to d (see Figs. 5(e $\rightarrow$ f)). Cell d is added to S_ℓ, if not already included.
2. Root clustering: Modules in S_ℓ attempt to move closer to r_0 to ensure that they are readily available when r_0 needs them. However, host modules in their final target position should never be displaced from that position, so we must carefully prevent such modules from compressing toward r_0. To achieve this, we alternate between the following two steps, until c and d both become 2-cells:
   (a) Logical leaf pruning: Remove any 1-cell leaf of S_ℓ that has been visited (i.e., in T). Note that a pruned 1-cell may end up back in S_ℓ one more time, during *root update*.
   (b) Cluster step: This step is applied to S_ℓ. Thus, only modules that are guests or unvisited leaves will try to move toward r_0.

Figure 5 illustrates the INPLACERECONFIGURATION algorithm with the help of a simple example.

4.3. *Algorithm correctness and complexity*

Lemma 7. *Algorithm* INPLACERECONFIGURATION *maintains a physically connected tree that contains all modules.*

Proof. By Lemma 2, SOURCECLUSTER produces a tree S containing all robot modules. Thus, we must show that TARGETGROW maintains such a tree when it receives one as input. We first analyze step 1 (DFS root update). In step 1.1, POPOUTLEAF maintains a tree, by Lemma 1. In step 1.2, d is already part of S. Now if $c \neq P(d)$, we attach c to d, which creates a cycle. However, we immediately break this cycle by detaching d from $P(d)$, and thus S is restored to a tree. See Figs. 5($c \rightarrow d \rightarrow e$) for an example. Regardless of the initial relationship between c and $P(d)$, and the possible restructuring of S, we proceed with TRANSFER(c, d), which maintains tree connectivity, by Lemma 1.

After steps 1.1 and 1.2 or the DFS update, no other physical connections are altered. Pointers are modified to reflect the made physical changes. As the root acts alone, our assumption that every cell is involved in one operation at a time holds.

Finally, S remains a physical tree during step 2 of TARGETGROW. This basically follows from the following two observations: (a) step 2a changes only the logical tree S_ℓ, and (b) CLUSTERSTEP maintains a tree, by Lemma 2. The only issue remaining is to prove that whenever a 1-cell leaf gets

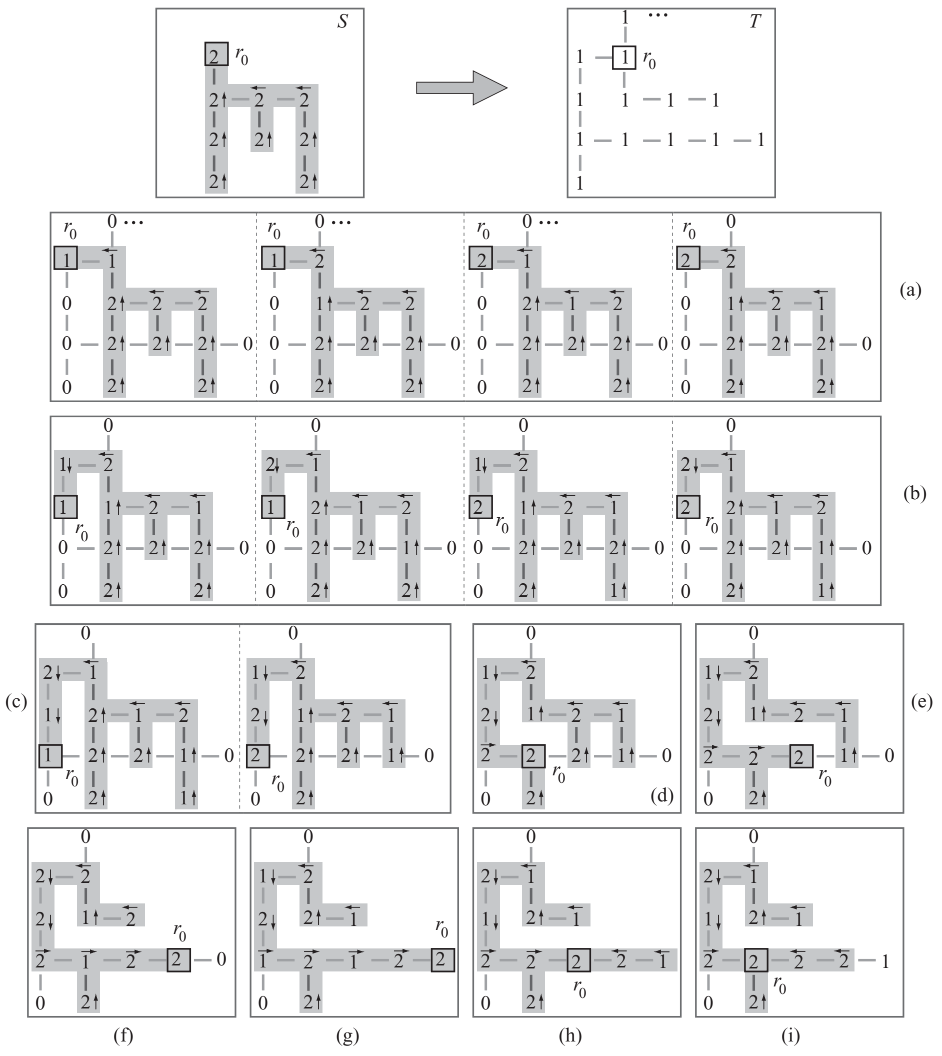

Fig. 5. Reconfiguring S into T: the top row shows S (after SourceCluster) and T. Links in the shaded S_ℓ are depicted as arrows. Subsequent figures show S (with its logical subtree S_ℓ shaded). (a) After TargetGrow, with each intermediate step illustrated (DFS root update on the left and the subsequent three clustering steps on the right); (b) after TargetGrow, with each intermediate step illustrated; (c) after TargetGrow, with its two main steps (root update and root clustering) illustrated; (d), (e), (f), and (g) show the next four TargetGrow steps; (h) after the next two TargetGrow steps; (i) after the next TargetGrow step (note the rightmost 1-cell leaf getting disconnected from S_ℓ); the process continues.

involved in PushInLeaf in ClusterStep, it is not visited. We wish to avoid moving visited 1-cells, as this operation could cause a physical disconnection if $S \neq S_\ell$. It suffices to claim that S_ℓ never contains a visited 1-cell leaf. This is in accordance with our definition of S_ℓ being the smallest tree containing unvisited modules. Given such a structure, the following situations can arise for a leaf u in step 2. The leaf u of S_ℓ could remain unaltered, or if it is a 1-cell (meaning, not visited) it could use PushInLeaf to become the guest in a new 2-cell leaf. Finally, if u is a 2-cell and becomes a 1-cell, we prune the 1-cell from S_ℓ if it is visited. □

In phase 1 of InPlaceReconfiguration, the SourceCluster call produces a root-clustered tree containing r_0 in a 2-cell. We now show that phase 2 maintains this property in constant time, regardless of how r_0 moves.

Lemma 8. TargetGrow *maintains S_ℓ as a root-clustered tree containing r_0 in a 2-cell. Furthermore, the procedure uses $O(1)$ parallel steps.*

Proof. The proof is rather similar to that of Lemma 6. Since the structure of S_ℓ is identical to the structure of S, with the exception of some branches being trimmed off, it follows from Lemma 7 that S_ℓ is connected physically. This property can also be derived from the fact that cell d is attached to one node only in S_ℓ, thus never creating a cycle.

Let S_ℓ^i denote the root-clustered tree that is input for TargetGrow at iteration i. In step 1 (DFS root update), S_ℓ^i will be modified according to any physical operation carried out (PopOutLeaf and Transfer). By Lemma 7, these changes result in a tree, which we call S_ℓ^{i+1}.

As step 1 only affects c and d, it follows that at the beginning of step 2, a long gap in S_ℓ^{i+1} must contain c, which becomes a 1-cell via PopOutLeaf (see Fig. 6(a)), or via Transfer (see Fig. 6(b)).

We now show that the loop in step 2 of TargetGrow iterates for at most four times before our claim holds. Recall that, since S_ℓ^i was root-clustered, children of c are either leaves, 2-cells, or their children have that property.

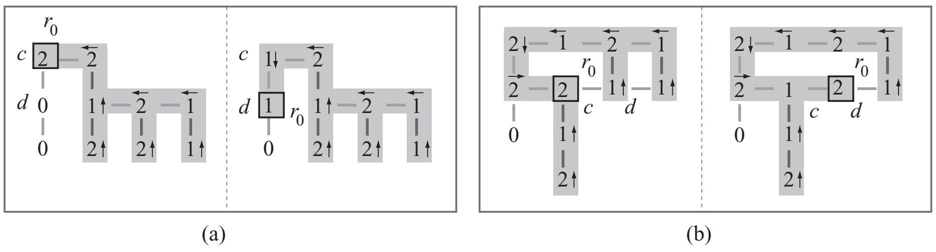

Fig. 6. DFS Root update. (a) POPOUTLEAF(c, d), (b) TRANSFER(c, d).

Any *leaf prune* operation only trims visited 1-cell leaves from the tree and thus does not affect the root-clustered property of the tree. There are the following two cases for the number of CLUSTERSTEP applications required to terminate the loop:

1. S_ℓ^{i+1} *was obtained via* POPOUTLEAF *(step 1.1)*: In this case c and d are 1-cells at the beginning of step 2. If all children of c are leaves or 2-cells, then in the first iteration of CLUSTERSTEP, c will again become a 2-cell. Otherwise, since S_ℓ^i was root-clustered, any non-leaf 1-cell child will become a 2-cell in the first iteration. Thus, in the second iteration at the latest, c will become a 2-cell. Furthermore, just as described in Lemma 6, the subtree rooted at any child of c remains root-clustered after the first application of CLUSTERSTEP (in particular, for the highest priority child, which is the only one that changes). Similarly, by the time c becomes a 2-cell, the subtree rooted at c also becomes root-clustered. The third CLUSTERSTEP makes d a 2-cell root of a root-clustered tree, since all children of c must have been leaves or 2-cells to supply a module to c. The fourth CLUSTERSTEP makes c a 2-cell, which terminates the loop.
2. S_ℓ^{i+1} *was obtained via* TRANSFER *(step 1.2)*: In this case d is already a 2-cell at the beginning of step 2 because of the TRANSFER operation in step 1.2. If c remains a 2-cell during the transfer, then S_ℓ^{i+1} is already root-clustered and the loop condition is satisfied. If c is a 1-cell, arguments similar to case 1 imply that after one application of CLUSTERSTEP, S_ℓ^{i+1} is root-clustered. A second application of CLUSTERSTEP makes c a 2-cell, which terminates the loop.

This concludes the proof. □

Theorem 2. *The* INPLACERECONFIGURATION *algorithm can be implemented in $O(n)$ parallel steps.*

Proof. By Lemma 5, phase 1 uses $O(n)$ steps. Step 2 of INPLACERECONFIGURATION has $O(n)$ iterations, since DFS has $O(n)$ complexity. By Lemma 8, each iteration takes constant time. □

5. Observations

5.1. *Matching lower bound*

Transforming a horizontal line of modules to a vertical line requires a linear number of parallel steps, if each module can only displace one other module and maximum velocity is constant.

3D: All of our techniques apply directly to 3D robots, once the top and bottom sides of cells are incorporated into our highest priority rule. None of our algorithms rely on the dimension being 2D. Once the primitive operations are set, every phase has a clear and direct extension to 3D. For instance, the 2D algorithm to construct a spanning tree of the robot can be extended in a straightforward way to 3D. The extension of our primitive operations is also trivial. Such operations still involve an interaction between two adjacent cells. The extra dimension only adds the requirement that more adjacent cells must maintain connectivity to two given cells involved in a primitive operation. This poses no difficulties.

5.2. *Reconfiguration of labeled robots*

Our algorithms are essentially unaffected if labels are assigned to modules. This is of interest if a robot is to have specialized modules, equipped with cameras, drills, etc. In TREETOPATH, assume that the partially constructed canonical path is sorted. Then a new module m can bubble/tunnel to its position by successive applications of the TRANSFER primitive operation. When it gets there, the tail of the path (from m to leaf) must shift over. This is straightforward, involving propagation of one compressed unit, and does not interfere with other modules following m. At all times, m or its replacement makes steady progress toward the leaf.

For the in-place algorithm, T can first be constructed disregarding labels. A similar type of bubble-sort can then be applied, taking place within T.

5.3. *Telecube robots*

The natural state of a Telecube robot has contracted atom arms. There is no room to compress two modules into one cell. Thus, an algorithm cannot commence with PUSHINLEAF operations, and it is not possible to physically exchange modules in adjacent cells while remaining in place. However, consider our first algorithm. We do not even need a SOURCECLUSTER phase, since all atoms are packed together. The root can transmit an instruction to a cell at maximum y-coordinate to act as root and immediately push out two of its atoms. For the construction of V, all analyses follow. It seems that labeled atoms within a module might become separated (e.g., if the module is at a junction in a tree). Thus, an extra step is used to collect the root atoms at the bottom of V.

Exact in-place reconfiguration is impossible for telecube robots if the modules are labeled. Thus, the root cannot travel to any position within S. It might be possible to deal with this issue by requiring larger modules and designing a "reduced module shape" for the root (e.g., fewer atoms, using naturally expanded links). Instead, we require that all modules have access to the map of T, which means any module can begin to expand T by filling adjacent 0-cells. Instead of backtracking or advancing physically through non-empty T cells of, the root can just tell its neighbors to take over. Eventually, a new root module would expand T at a different connected component of 0-cells.

5.4. Other modular robots

It has been shown[1] that suitably constructed modules of other prototypes (e.g., M-TRAN[18] and ATRON[16]) are capable of simulating Crystalline atoms. Such modules may require 50–100 atoms to simulate one Crystalline atom, and the resulting shape is not compact. Nevertheless, the result in ref. [1] implies that our results here apply to large systems of other prototypes.

Acknowledgments

We thank the other participants of the 2008 *Workshop on Reconfiguration* at the Bellairs Research Institute of McGill University for providing a stimulating research environment. We also thank the reviewers of ref. [2] and *Robotica* for their constructive comments.

References

1. G. Aloupis, N. Benbernou, M. Damian, E. Demaine, R. Flatland, J. Iacono and S. Wuhrer, "Efficient Reconfiguration of Lattice-Based Modular Robots," *European Conference on Mobile Robots*, Mlini/Dubrovnik, Croatia, (Sep. 23–25, 2009a) pp. 81–86.
2. G. Aloupis, S. Collette, M. Damian, E. D. Demaine, D. El-Khechen, R. Flatland, S. Langerman, J. O'Rourke, V. Pinciu, S. Ramaswami, V. Sacristán and S. Wuhrer, "Realistic Reconfiguration of Crystalline and Telecube Robots. **In**: *8th International Workshop on the Algorithmic Foundations of Robotics (WAFR)* (G. S. Chirikjian *et al.* Eds.), Springer Tracts in Advanced Robotics, vol. 57, (2008a) pp. 433–447.
3. G. Aloupis, S. Collette, M. Damian, E. D. Demaine, R. Flatland, S. Langerman, J. O'Rourke, S. Ramaswami, V. Sacristán and S. Wuhrer, "Linear reconfiguration of cube-style modular robots," *Comput. Geom., Theor. Appl.* **42**(6–7), 652–663 (2009b).
4. G. Aloupis, S. Collette, E. D. Demaine, S. Langerman, V. Sacristán and S. Wuhrer, "Reconfiguration of Cube-Style Modular Robots Using $O(\log n)$ Parallel Moves," *Proceedings of the International Symposium on Algorithms and Computation (ISAAC 2008)*, volume 5369 of *LNCS*, (2008b) pp. 342–353.
5. G. Aloupis, S. Collette, E. D. Demaine, S. Langerman, V. Sacristán and S. Wuhrer, "Reconfiguration of 3D Crystalline Robots in $O(\log n)$ Parallel Steps. *Technical Report* arXiv:0908.2440 (2009c) p. 21.
6. N. Brener, F. B. Amar and P. Bidaud, "Designing modular lattice systems with chiral space groups," *Int. J. Robot. Res.* **27**(3–4), 279–297 (2008).
7. Z. Butler, S. Byrnes and D. Rus, "Distributed Motion Planning for Modular Robots with Unit-Compressible Modules," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Maui, Hawaii, USA, vol. 2, (2001) pp. 790–796.
8. Z. Butler, R. Fitch and D. Rus, "Distributed control for unit-compressible robots: Goal-recognition, locomotion and splitting," *IEEE/ASME Trans. Mechatron.* **7**(4), 418–430 (2002).
9. Z. Butler, K. Kotay, D. Rus and K. Tomita, "Generic decentralized control for lattice-based self-reconfigurable robots," *Int. J. Robot. Res.* **23**, 919–937 (2004).
10. Z. Butler and D. Rus, "Distributed planning and control for modular robots with unit-compressible modules," *Int. J. Robot. Res.* **22**(9), 699–715 (2003).
11. C.-J. Chiang and G. Chirikjian, "Similarity metric with applications in modular robot motion planning," *Auton. Robots* **10**(1), 91–106 (2001).
12. G. Chirikjian, "Kinematics of a Metamorphic Robotic System," *Proceedings of the IEEE International Conference on Robotic Automation* (May 8–13, 1994) pp. 449–455.
13. G. Chirikjian, A. Pamecha and I. Ebert-Uphoff, "Evaluating efficiency of self-reconfiguration in a class of modular robots," *J. Robot. Syst.* **13**(5), 317–338 (1996).
14. T. Fukuda and S. Nakagawa, "Approach to the dynamically reconigurable robotic system," *J. Intell. Robot. Syst.* **1**(1), 55–72 (1988).
15. K. Hosokawa, T. Tsujimori, T. Fujii, H. Kaetsu, H. Asama, Y. Kuroda and I. Endo, "Self-Organizing Collective Robots with Morphogenesis in a Vertical Plane," *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Leuven, Belgium (May 16–20, 1998) pp. 2858–2863.
16. M. W. Jørgensen, E. H. Østergaard and H. H. Lund, "Modular ATRON: Modules for a Self-Reconfigurable Robot," *Proceedings of the of the International Conference on Intelligent Robots and Systems* (2004) pp. 2068–2073.
17. K. Kotay and D. Rus, "Algorithms for Self-Reconfiguring Molecule Motion Planning," *Proceedings of the International Conference on Intelligent Robots and Systems*, Kagawa University, Takamatsu, Japan (Oct. 30–Nov. 5, 2000) pp. 2184–2193.
18. H. Kurokawa, K. Tomita, A. Kamimura, S. Kokaji, T. Hasuo and S. Murata, "Self-reconfigurable Modular Robot M-TRAN: Distributed Control and Communication," **In**: *RoboComm '07: Proceedings of the 1st International Conference on Robot Communication and Coordination*, Piscataway, NJ, USA. (IEEE Press, 2007) pp. 1–7.
19. H. Kurokawa, K. Tomita, A. Kamimura, S. Kokaji, T. Hasuo and S. Murata, "Distributed self-reconfiguration of M-TRAN III modular robotic system," *Int. J. Robot. Res.* **27**, 373–386 (2008).
20. S. Murata and H. Kurokawa, "Self-reconfigurable robots: Shape-changing cellular robots can exceed conventional robot flexibility," *IEEE Robot. Autom. Mag.* **14**(1), 43–52 (2007).
21. S. Murata, H. Kurokawa and S. Kokaji, "Self-Assembling Machine," *Proceedings of the IEEE International Conferance Robotic Automation*, San Diego, CA, USA (May 1994) pp. 441–448.
22. A. Pamecha, C. Chiang, D. Stein and G. Chirikjian, "Design and Implementation of Metamorphic Robots," **In**: *Proceedings of the ASME Design Engineering Technical Conference and Computers in Engineering Conference* (J. McCarthy, ed.), Irvine, CA, (1996) pp. 1–10.
23. A. Pamecha, I. Ebert-Uphoff and G. Chirikjian, "Useful metrics for modular robot motion planning," *IEEE Trans. Robot. Autom.* **13**(4), 531–545 (1997).
24. J. H. Reif and S. Slee, "Optimal Kinodynamic Motion Planning for Self-Reconfigurable Robots Between Arbitrary 2D Configurations," *Robotics: Science and Systems Conference, Georgia Institute of Technology*, Atlanta, GA, USA (June 27–30, 2007).
25. D. Rus and M. Vona, "Crystalline robots: Self-reconfiguration with compressible unit modules," *Auton. Robot* **10**(1), 107–124 (2001).
26. B. Salemi, P. Will and W.-M. Shen, "Distributed task negotiation in modular robots," *J. Robot. Soc. Japan, (Special Issue on "Modular Robots")* **21**(8), 32–39 (2003).
27. K. Stoy, "Using cellular automata and gradients to control self-reconfiguration," *Robot. Auton. Syst.* (Special issue IAS-04) **54**, 135–141 (2006).
28. J. W. Suh, S. B. Homans and M. Yim, "Telecubes: Mechanical Design of a Module for Self-Reconfigurable Robotics," *Proceedings of the IEEE International Conference on Robotics and Automation*, Washington, DC (May 11–15, 2002) pp. 4095–4101.

29. C. Ünsal, H. Kilite, M. Patton and P. Khosla, "Motion Planning for a Modular Self-Reconfiguring Robotic System," *Proceedings of the 5th International Symposium on Distributed Autonomous Robotic Systems*, Knoxville, Tennessee, USA (Oct. 4–6, 2000).
30. S. Vassilvitskii, M. Yim and J. Suh, "A Complete, Local and Parallel Reconfiguration Algorithm for Cube Style Modular Robots," *Proceedings of the of the IEEE International Conference on Robotics and Automation* (2002) pp. 117–122.
31. J. E. Walter, E. M. Tsai and N. M. Amato, Choosing Good Paths for Fast Distributed Reconfiguration of Hexagonal Metamorphic Robots," *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (2002) pp. 102–109.
32. M. Yim, "New Locomotion Gaits," *Proceedings of the IEEE International Conference Robotic Automation*, San Diego, CA, USA (May 1994).
33. M. Yim, D. G. Duff and K. D. Roufas, "Polybot: A Modular Reconfigurable Robot." *Proceedings of the 2000 IEEE International Conference on Robotics and Automation* (2000) pp. 514–520.
34. M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins and G. S. Chirikjian, "Modular self-reconfigurable robots systems: Challenges and opportunities for the future," *IEEE Robot. Autom. Mag.* **14**(1), 43–52 (2007).
35. E. Yoshida, S. Murata, A. Kamimura, K. Tomita, H. Kurokawa and S. Kokaji, "A self-reconfigurable modular robot: Reconfiguration planning and experiments," *Int. J. Robot. Res.* **21**(10–11), 903–915 (2002).

Appendix

PushInLeaf and PopOutLeaf

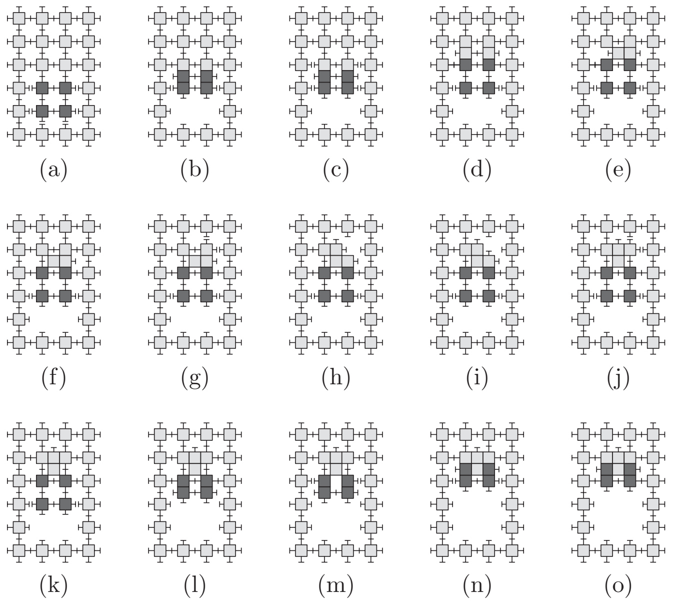

Fig. 7. Details of PushInLeaf and PopOutLeaf.

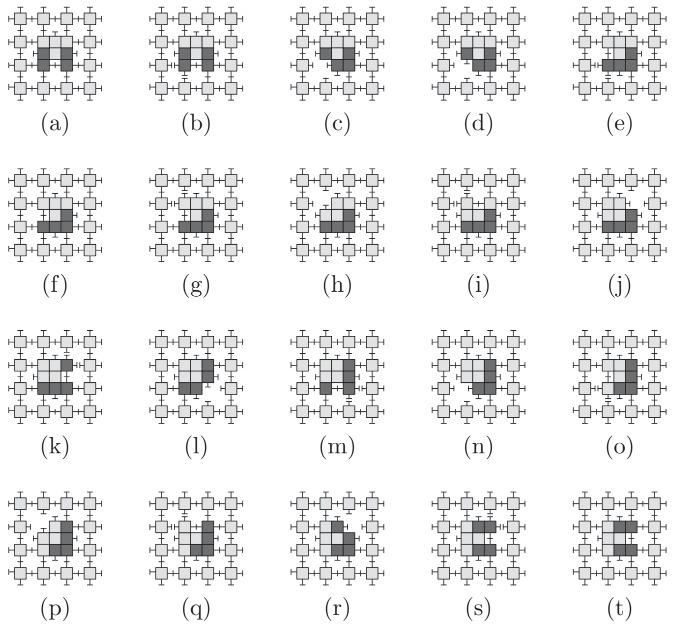

Fig. 8. Details of the *positioning* step of the primitive operation Transfer(m, q). Before being sent to the lattice position of q, the module m is rotated within its lattice cell in order to face q.

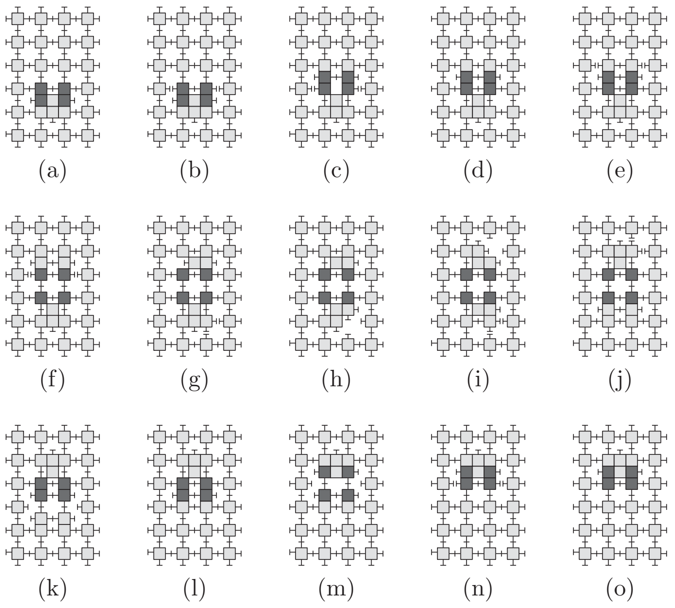

Fig. 9. Details of the *Send* step of the primitive operation Transfer.

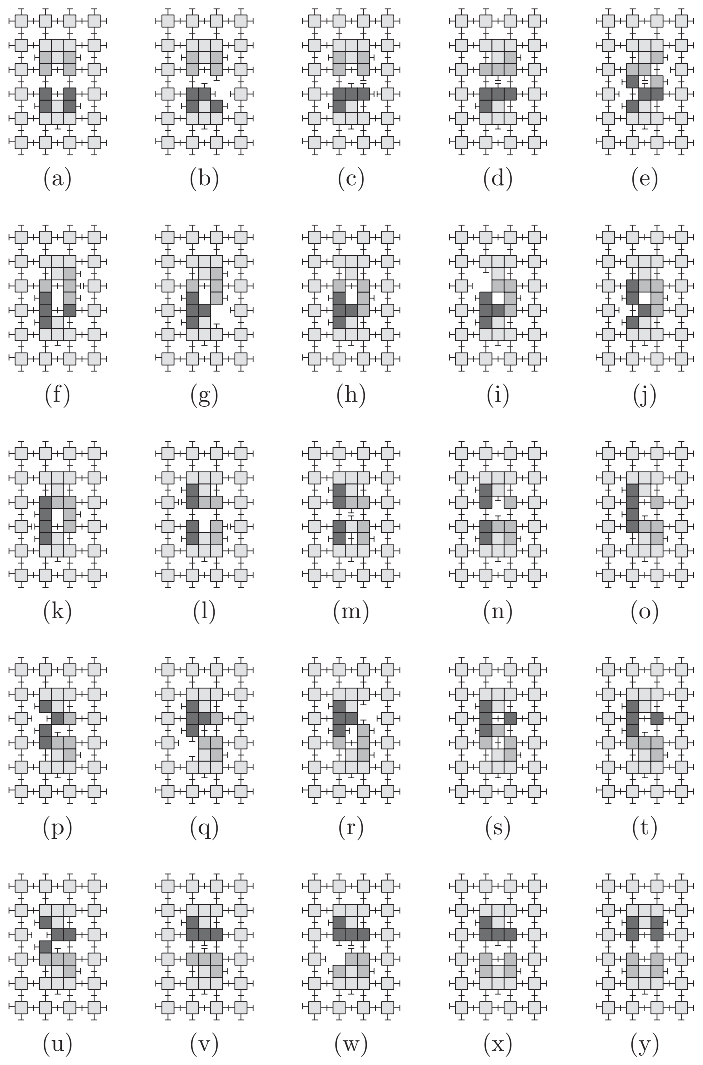

Fig. 10. Details of the *Exchange* step of the primitive operation TRANSFER.

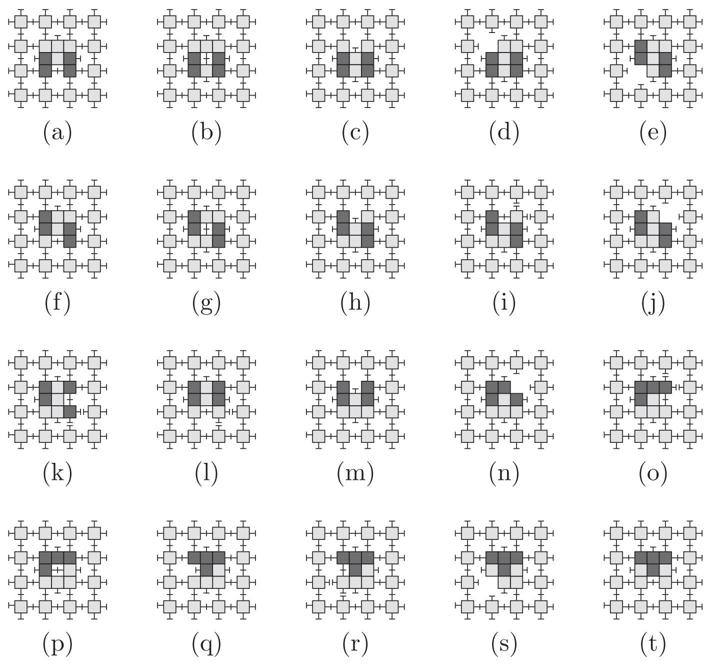

Fig. 11. Details of SWITCH.