Realistic Reconfiguration of Crystalline (and Telecube) Robots

Greg Aloupis[1], Sébastien Collette[1] *, Mirela Damian[2], Erik D. Demaine[3] †, Dania El-Khechen[4], Robin Flatland[5], Stefan Langerman[1] ‡, Joseph O'Rourke[6], Val Pinciu[7], Suneeta Ramaswami[8], Vera Sacristán[9] §, and Stefanie Wuhrer[10]

[1] Université Libre de Bruxelles, Belgique, {greg.aloupis,sebastien.collette,stefan.langerman}@ulb.ac.be
[2] Villanova University, Villanova, USA, mirela.damian@villanova.edu
[3] Massachusetts Institute of Technology, Cambridge, USA, edemaine@mit.edu
[4] Concordia University, Montreal, Canada, d_elkhec@cs.concordia.ca
[5] Siena College, Loudonville, N.Y., USA, flatland@siena.edu
[6] Smith College, Northampton, USA, orourke@cs.smith.edu
[7] Southern Connecticut State University, USA, pinciu@scsu.ctstateu.edu
[8] Rutgers University, Camden, NJ, USA, rsuneeta@camden.rutgers.edu
[9] Universitat Politècnica de Catalunya, Barcelona, Spain, vera.sacristan@upc.edu
[10] Carleton University, Ottawa, Canada, swuhrer@scs.carleton.ca

Abstract: 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 unit velocity and (via expansion) can displace at most one other atom. We require that one of the modules can store a map of the target configuration.

Our algorithms involve a total of $O(n^2)$ such atom operations, which are performed in $O(n)$ parallel steps. This improves on previous reconfiguration algorithms, which either use $O(n^2)$ parallel steps [7, 9, 4] or do not respect the constraints mentioned above [1]. In fact, in the setting considered, our algorithms are optimal, in the sense that certain reconfigurations require $\Omega(n)$ parallel steps. A further advantage of our algorithms is that reconfiguration can take place within the union of the source and target configurations.

* Chargé de Recherches du FNRS.
† Partially supported by NSF CAREER award CCF-0347776, DOE grant DE-FG02-04ER25647, and AFOSR grant FA9550-07-1-0538.
‡ Maître de Recherches du FNRS.
§ Partially supported by projects MEC MTM2006-01267 and DURSI 2005SGR00692.

1 Introduction

Crystalline and Telecube robots. In this paper, we present new algorithms for the reconfiguration of robots composed of *crystalline atoms* [3] or *telecube atoms* [8, 9], both of which have been prototyped.

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, detach) in a coordinated way, the atoms move relative to one another, resulting in a reconfiguration of the robot. Figure 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]. In the two-dimensional setting that we focus on, we assume that modules consist of 2×2 atoms. Our algorithms can easily be extended to $3D$.

We refer the reader to [7, 9, 1] for a more detailed and basic introduction to these robots. Various types of self-reconfiguring robots, as well as related algorithmic issues, are surveyed in [6].

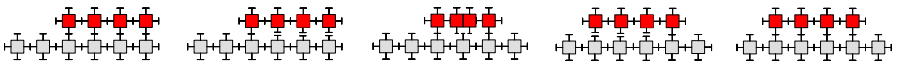

Fig. 1. Example of reconfiguring crystalline atoms.

The model. The problem we solve is to reconfigure a given connected source configuration of n modules to a specified, arbitrary, connected target configuration T in $O(n)$ parallel steps. We allow modules to be able to exert only a constant amount of force, independent of n. In other words 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, 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). Our algorithms are designed for Crystalline robots. For a discussion of the main differences for Telecube robots, see Section 5.

We restrict our descriptions to a 2D lattice. 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 contain up to two modules (see Fig. 2). Cells are marked with an integer in $\{0, 1, 2\}$: 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.

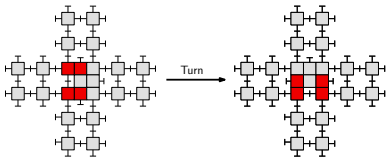

Fig. 2. The middle cell contains two modules. The red *guest* module is capable of turning orientation. Only initial and final configurations are shown.

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 cells to attach/detach so that the attachments model the tree connections 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 host modules. 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. 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.

Related results. Algorithms for reconfiguring Crystalline and Telecube robots in $O(n^2)$ parallel steps have been given in [7, 9, 4]. A linear-time parallel algorithm for reconfiguring within the bounding box of source and target is given in [1]. The total number of individual moves is also linear. However, no restrictions are made concerning physical properties of the robots. For example, $O(n)$ strength is required, since modules can carry towers and push large masses during certain operations. A $O(\log n)$ time algorithm for 2D robots that uses $O(n \log n)$ parallel moves and also stays within the bounding box is given in [2]. However, not only do modules have $O(n)$ physical strength, they can also reach $O(n)$ velocity. A $O(\sqrt{n})$ time algorithm for 2D robots, using the third dimension as an intermediate, is given in [5]. This is optimal in the model considered, which permits linear velocities, but only constant acceleration. If applied within the model used in [2], this algorithm would run in constant time. We remind the reader that, unlike [5, 1, 2], we limit force and velocity to a constant level.

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, and relatively easily adaptable to Telecube robots. 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. As far as we are aware, this is the first in-place reconfiguration algorithm. Both algorithms involve considering the given robot as a spanning tree, and pushing leaves towards the root with "parallel tunneling". 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 optimal, in the sense that certain reconfigurations require a linear number of parallel moves.

2 Primitive operations

Let m and q be adjacent cells. We define the following primitive operations:

1. PushInLeaf(m, q) – 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) – applies when q is compressed and m is empty. This is the inverse of the PushInLeaf operation.
3. Transfer(m, q) – applies when m is compressed and q is non-empty; if q is compressed, the guests of both cells physically exchange positions. Otherwise, the guest of m moves into (and becomes a guest of) q.
4. Attach(m, q) – host modules in m and q form a physical connection.
5. Detach(m, q) – the inverse of attach.
6. Switch(m) – applies when m is compressed. Its two modules physically switch positions (and roles of host and guest).

PushInLeaf, PopOutLeaf, and Transfer are illustrated in Fig. 3.

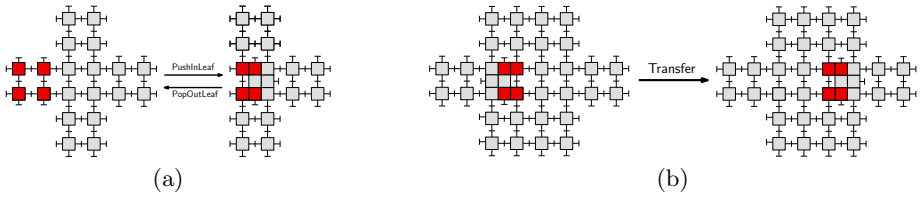

Fig. 3. (a) PushInLeaf and PopOutLeaf. (b) Transfer. Only initial and final configurations are shown.

Note that any permutation of atoms within a module 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. Details are omitted due to space restrictions.

In the remainder of this paper, we will 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. Details remain to be verified.

Lemma 1. *Operations* PushInLeaf*,* PopOutLeaf*,* Switch *and* Transfer *maintain the tree structure of a robot.*

The proof is omitted for lack of space.

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

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 relatively straightforward and not discussed here, due to space restrictions.

3.1 Algorithmic details

We first move r_0 to a maximum possible y-coordinate within S: This involves pushing in a leaf and iteratively transferring it to r_0, so that r_0 will become part of a 2-cell and then be able to iteratively transfer to its target. Note that this might not be necessary in some implementations, where all modules might be capable of playing the role of r_0 (for example, 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, by compressing in at the leaves and moving up S in parallel. The shape of S shrinks, as PushInLeaf operations in ClusterStep compress leaf modules into their parent cells. It is not critical that all cells become compressed. In fact this phase mainly helps to analyze the total number of parallel steps in our algorithm. At the end of this phase, all *non-leaf* cells will become 2-cells. In this state we refer to S as being *fully compressed*.

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).

SourceCluster is illustrated in Figure 4. 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 supply a steady stream of guest modules to its ancestors.

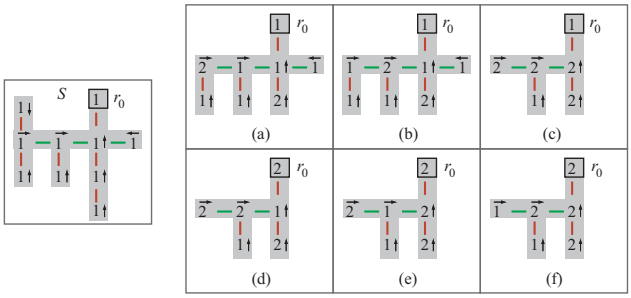

Fig. 4. An example of SourceCluster.

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 Figure 5.

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, Transfer(u, c).
   b) ClusterStep(S)

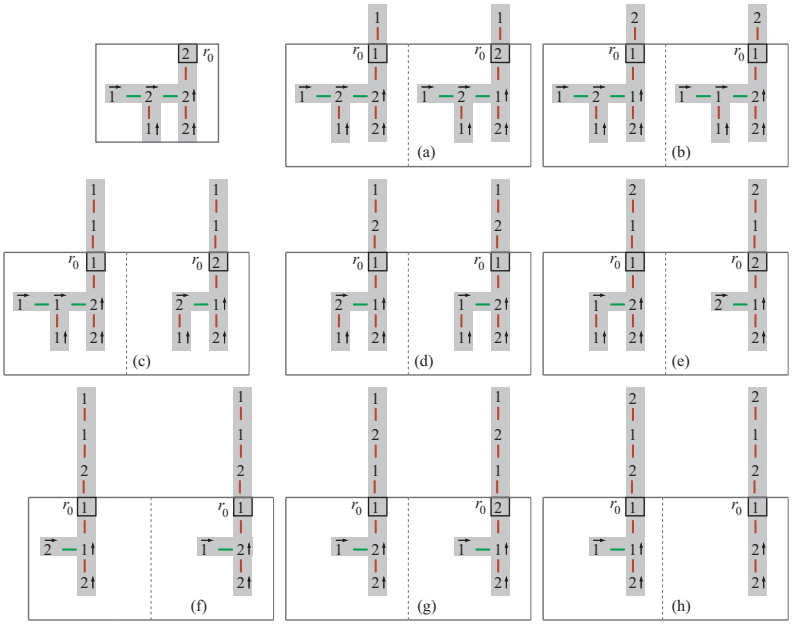

Fig. 5. An example of TreeToPath.

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. The claim follows immediately for SourceCluster. □

The height of a cell in S is the height of its subtree in S.

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, 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 its other 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. *Let r be a 2-cell with height h that transfers its guest module to $P(r)$ in iteration i of* SourceCluster*. 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 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, since 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 becomes a 1-cell again. 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. □

Let the depth of a cell in a tree be its distance from the root. Hence, the root has depth zero.

Lemma 5. SourceCluster *terminates after at most $2h-1$ iterations of* ClusterStep.

Proof. We claim that at the completion of iteration $h-1+d$ of SourceCluster, all non-leaf modules at depth less than or equal to d in S_{h-1+d} are 2-cells. The proof is by induction on d. The base case is the root of S_{h-1} at depth $d = 0$. By Lemma 3, the root becomes a 2-cell in iteration $h - 1$. Assume inductively that our claim is true for all values 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 $i = h-1+(d-1)$, 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 or a leaf by the end of iteration $i+1$. In the second case, c is not part of S anymore.

Since p will not accept new guest modules after iteration i, 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 $d = h$, our result follows. □

Let a *long gap* consist 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. Since 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.

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. □

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

4 In-place reconfiguration

This section describes an 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. We call such an algorithm *in-place*. If all modules were to know which direction to take in each time unit (for example, by having an external source synchronously transmit 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, in this section we impose the restriction that all modules are only capable of communicating locally. Thus it is up to r_0 to direct all action.

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 DFS (depth-first search) 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 first must 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, each of whose cells u maintains two links: a *physical* link corresponding to the physical connection between u and $P(u)$, and a *logical* link that could either be NULL, or identical to the physical link. We call the tree S_ℓ induced by the logical links the *logical tree*.

S always contains all occupied cells. S_ℓ is the smallest tree containing the modules that are 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. Repeat until r_0 reaches the final position in its DFS traversal:
$S \leftarrow$ TargetGrow(S, T).

We now describe the operation of TargetGrow.

TargetGrow(S, T)

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

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 Include d in S_ℓ.

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

In Phase 2, we start with $S_\ell = S$. Throughout this phase, the logical tree is maintained as a subtree of the physical tree: $S_\ell \subseteq S$. 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 a DFS walk of T. This is accomplished either by uncompressing (popping) r_0 into d (see Fig. 6(a $\rightarrow$ b)), or by transferring r_0 to d (see Fig. 6(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 towards r_0. To achieve this, we alternate between the following two steps, until c and d both become 2-cells:
   a) Logical Leaf Detach: remove any 1-cell leaf of S_ℓ that has been visited (i.e., is in T). Note that a detached 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 towards r_0.

Fig. 6 illustrates the InPlaceReconfiguration algorithm with the help of a simple example. The top row of the figure shows the source configuration S after phase 1 completes (left), and the target robot configuration (right). Links in S_ℓ, which is shaded, are depicted as arrows.

Algorithm Correctness.

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 (see Fig. 7b). However, we immediately break this cycle by detaching d from $P(d)$, and thus S is restored to a tree. See Figs. 6($c \rightarrow d \rightarrow e$) for an example. Regardless of the initial relation between c and $P(d)$, and the possible re-structuring 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 physical changes made.

Finally, S remains a physical tree during step 2 of TargetGrow. This follows from two observations: (a) step 2a changes only the logical tree S_ℓ, and (b) ClusterStep maintains S as a tree, by Lemma 2. □

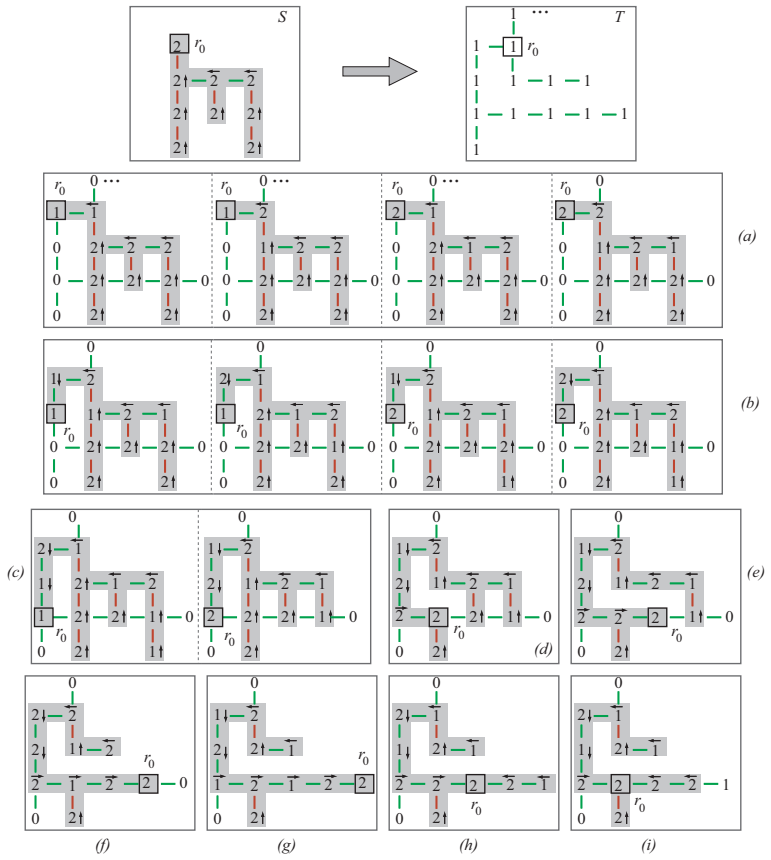

Fig. 6. Reconfiguring S into T: the top row shows S (after SourceCluster) and T. Subsequent figures show S (with its logical subtree S_ℓ shaded) *(a)* after TargetGrow, with each intermediate step illustrated (DFS root update on the left and the subsequent 3 clustering steps on the right); *(b)* after TargetGrow, with each intermediate step illustrated; *(c)* after TargetGrow, with its two main steps (root update and root clustering) illustrated; *(d,e,f,g)* show the next 4 TargetGrow steps; *(h)* after the next 2 TargetGrow steps; *(i)* after the next TargetGrow (note the rightmost 1-cell leaf getting disconnected from S_ℓ); the process continues.

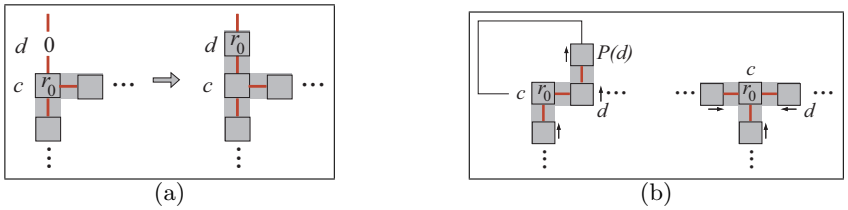

Fig. 7. (a) PopOutLeaf(c, d). (b) DFS Root update when $c \neq P(d)$ (left) and $c = P(d)$ (right).

In phase 1 of InPlaceReconfiguration SourceCluster 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 $S_\ell \subseteq S$, it follows from Lemma 7 that S_ℓ is physically connected.

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

Since 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. 8a), or via Transfer (see Fig. 8b).

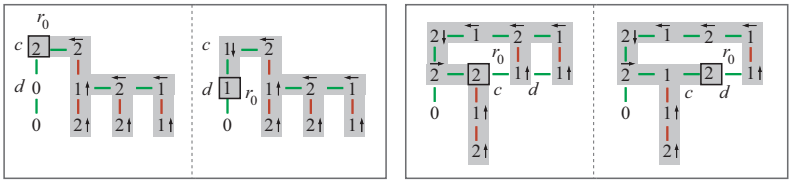

Fig. 8. DFS Root update (a) PopOutLeaf(c, d) (b) Transfer(c, d).

We now show that the loop in step 2 of TargetGrow iterates at most four times before our claim holds. Recall that, since S_ℓ^{i+1} was root-clustered, children of c are either leaves, 2-cells, or their children have that property.

Any *DetachLeaf* operation only trims visited 1-cell leaves from the tree and thus does not affect the root-clustered property of the tree. There are 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 become a 2-cell again. 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

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 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.

Labeled robots: Our algorithms are essentially unaffected if labels are assigned to modules. In TreeToPath, assume that the partially constructed canonical path is sorted. Then a new module m pushed through can bubble/tunnel to its position. When it gets there, the tail of the path must shift over, but this is straightforward involving propagation of one compressed unit, and does not interfere with other modules following m.

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.

Telecube robots: The natural state of a telecube robot has atom arms contracted. 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 analysis follows. It seems that labeled atoms within a module might become separated (for example, 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. Unavoidably, they must travel along the side of V to do this.

Exact in-place reconfiguration is impossible for labeled telecube robots. 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 could 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 through non-empty cells of T physically, 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.

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.

References

1. 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. In *Proc. Intl. Symp. on Algorithms and Computation (ISAAC 2007)*, volume 4835 of *LNCS*, pages 208–219, 2007.
2. 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 (submitted). 2008.
3. Z. Butler, R. Fitch, and D. Rus. Distributed control for unit-compressible robots: Goal-recognition, locomotion and splitting. *IEEE/ASME Trans. on Mechatronics*, 7(4):418–430, 2002.
4. Z. Butler and D. Rus. Distributed planning and control for modular robots with unit-compressible modules. *Intl. Journal of Robotics Research*, 22(9):699–715, 2003.
5. J. H. Reif and S. Slee. Optimal kinodynamic motion planning for self-reconfigurable robots between arbitrary 2D configurations. In *Robotics: Science and Systems Conference, Georgia Institute of Technology*, 2007.
6. D. Rus, Z. Butler, K. Kotay, and M. Vona. Self-reconfiguring robots. *Communications of the ACM*, 45(3):39–45, 2002.
7. D. Rus and M. Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *Autonomous Robots*, 10(1):107–124, 2001.
8. J. W. Suh, S. B. Homans, and M. Yim. Telecubes: Mechanical design of a module for self-reconfigurable robotics. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation*, pages 4095–4101, 2002.
9. S. Vassilvitskii, M. Yim, and J. Suh. A complete, local and parallel reconfiguration algorithm for cube style modular robots. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation*, pages 117–122, 2002.