

Reconfiguration Planning Among Obstacles for Heterogeneous Self-Reconfiguring Robots

Robert Fitch[†]
Department of Computer Science
Dartmouth College
Hanover, NH USA
rfitch@cs.dartmouth.edu

Zack Butler
Department of Computer Science
Rochester Institute of Technology
Rochester, NY USA
zjb@cs.rit.edu

Daniela Rus
CSAIL
MIT
Cambridge, MA USA
rus@csail.mit.edu

Abstract—Most reconfiguration planners for self-reconfiguring robots do not consider the placement of specific modules within the configuration. Recently, we have begun to investigate heterogeneous reconfiguration planning in lattice-based systems, in which there are various classes of modules. The start and goal configurations specify the class of each module, in addition to placement. Our previous work presents solutions for this problem with unrestricted free space available to the robot during reconfiguration, and also free space limited to a thin connected region over the entire surface of the configuration. In this paper, we further this restriction and define free space by an arbitrarily-shaped bounding region. This addresses the important problem of reconfiguration among obstacles, and reconfiguration over a rigid surface. Our algorithm plans module trajectories through the volume of the structure, and is divided into two phases: shape-forming, and sorting the goal configuration to correctly position modules by class. The worst-case running time for the first phase is $O(n^2)$ with $O(n^2)$ moves for an n -module robot, and a loose upper bound for the second phase is $O(n^4)$ time and moves. However, we show this bound to be $\Theta(n^2)$ time and moves in common instances.

I. INTRODUCTION

Reconfiguration planning in self-reconfiguring (SR) robots is the problem of how to transform the robot's configuration to match a goal. This problem has been studied mainly in the context of homogeneous systems. Recently, we have investigated reconfiguration in heterogeneous SR robots and examined how the free space available to the robot during reconfiguration affects planning complexity. The benefits of heterogeneous systems come from functional specialization of modules. For example, a heterogeneous robot can include individual sensor, power, or communications modules and control their placement within the configuration. Planners for homogeneous systems cannot guarantee such placement. In our previous work, we showed that the worst-case planning complexity for reconfiguration in heterogeneous lattice-based robots is equivalent to the homogeneous case, given a small amount of connected free space [6], [7]. In this paper, we continue this line of inquiry and examine the case where free space is defined by an arbitrarily-shaped bounding region. This addresses the important practical problem of reconfiguration among obstacles, and also the interest-

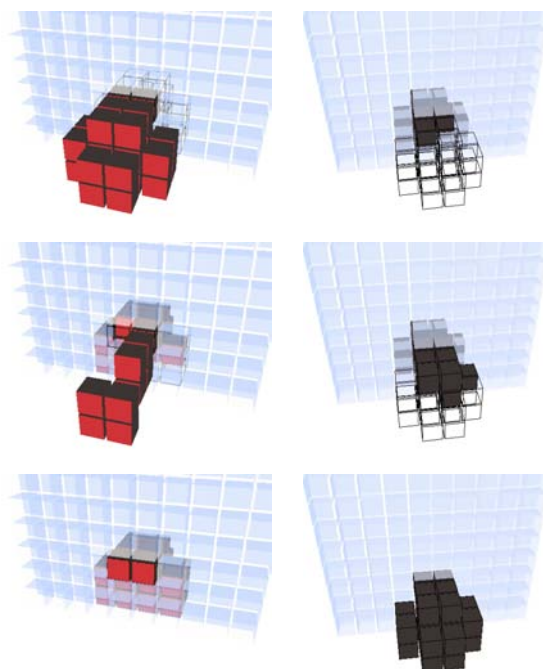


Fig. 1. Example of heterogeneous SR robot facing a large obstacle. Left column shows view from front of obstacle; right column shows corresponding view from rear of obstacle. Wireframe indicates goal shape. Robot moves through small hole to form goal configuration, which specifies positioning of special sensor modules within the structure.

ing theoretical question of reconfiguration planning with severely constrained, possibly disconnected free space.

A motivating example for this problem is reconfiguration on a surface. Unless the robot is floating in space, reconfiguration must be able to avoid planning paths through impenetrable surfaces such as the ground. Also, such an algorithm can allow the robot to move tightly against obstacles, in a type of compliant reconfiguration for locomotion. Fig. 1 shows an example where a heterogeneous SR robot reconfigures to move through a small hole in a wall-shaped obstacle, placing sensor modules in desired locations in the goal configuration.

The challenge of disconnected free space is that the constraints can make it very difficult, or even impossible, to move modules to a desired position. For example, consider the 2D homogeneous instance in Fig. 2(a). If free

[†]Current affiliation is National ICT Australia (<http://nicta.com.au>).

space surrounds the configuration, a greedy method can be applied that repeatedly searches for a mobile module and moves it over the surface of the structure to any unfilled position in the goal configuration. But, as the example shows, a greedy planner can fail if some space adjacent to the goal configuration is not free. Even worse, Fig. 2(b) shows a locked configuration where no moves are possible. A heterogeneous example is in Fig. 2(c). Here, the only available free space is the size of one module. With uniquely-typed modules, this problem is an instance of the $(n^2 - 1)$ -Puzzle, which is not solvable for all instances [8].

Our approach to these challenges is to move modules through the volume of the structure in a planned order. Before discussing this approach precisely, we first discuss related results. Then, we define the problem and our approach, summarize results, and give an outline for the rest of the paper.

A. Related Work

The SR systems considered in this paper are lattice-based. Many groups have designed and constructed hardware prototypes [3], [9], [11], [13], [14], [16], [18], [21]. Instantiation of the Sliding-Cube motion primitives with several hardware designs is described by Butler [1].

Complexity of the reconfiguration problem was first studied by Pamecha and Chirikjian [15]. Homogeneous reconfiguration in unit-compressible systems is well-studied [2], [19]. Other related work in reconfiguration planning is by Yim et al. [21], Walter, Welch, and Amato [20], and Christensen, Østergaard, and Lund [3].

In our work, we maintain connectivity during reconfiguration by explicitly planning paths. An alternative method for maintaining connectivity is to consider special classes of shapes. One idea is the notion of scaffolding – building sparse configurations with many internal holes. This is used in planning by Kotay [10] and by Stoy [17].

Planning for heterogeneous systems with unique module IDs was introduced by our group. An out-of-place (unlimited free space) algorithm is presented in [6], and an in-place (limited free space) algorithm is presented in [7].

B. General Approach and Summary of Results

The heterogeneous reconfiguration planning problem is how to compute a feasible plan that, when executed from an initial configuration C , results in a specified goal configuration C' . A plan is feasible if it maintains connectivity and consists of valid primitive motions. A configuration is a list of modules defined by type and position defined by coordinates within a common reference frame. Alignment between start and goal configurations is given. We assume modules defined by a module abstraction called the *Sliding-Cube* [6]. This abstraction encapsulates architecture-specific details; most existing lattice-based hardware can instantiate this model using a constant number of modules and moves. Sliding-Cube modules are cube-shaped, all have the same size, and belong to classes, or types, identified by unique class (type) IDs. Primitive motions consist of 1) sliding motion over the surface of

adjacent modules, and 2) convex transition from the face of a neighbor module to an adjacent face of the same module. Modules in adjacent lattice positions are assumed to be connected at their faces.

Heterogeneous reconfiguration can be decomposed into two interrelated tasks: 1) forming overall shape, and 2) positioning modules correctly according to type. It is possible to separate these tasks since shape errors can be corrected ignoring type, assuming same-sized modules. The first is called the *homogeneous phase*. Correcting type errors involves relocating modules among a predetermined set of positions. This can be thought of as sorting the modules in the goal shape, and is called the *heterogeneous phase*. Our previous algorithm, *TunnelSort (TS)*, uses the simple greedy method for the homogeneous phase, and solves the heterogeneous phase using a technique we call *tunneling*. A tunnel is a means of unlocking an interior module by creating a path between it and the surface of the structure. Modules along this *tunnel path* are temporarily displaced and stored in available free space. Two interior modules are swapped by creating a tunnel for each, swapping positions, and replacing displaced modules. In TS, we assume that there is a one-module-thick crust of free space on all sides of the system, and so swap order can be arbitrary and all paths are straight lines.

The algorithm we present here, *ConstrainedTunnelSort (CTS)*, defines a more general tunneling procedure that allows tunnel paths with bends. CTS uses the greedy algorithm for the homogeneous phase, but replaces surface-moving module trajectories with tunneling. This avoids the failure described in Fig. 2(d) because previously unreachable interior positions can be reached via tunneling. For the heterogeneous phase, CTS swaps modules by tunneling to reachable free space. Swap order can no longer be arbitrary (see Fig. 2(d)), so CTS plans a valid swap sequence by searching a graph that describes which modules can reach which free space regions. The worst-case running time of the homogeneous phase is $O(n^2)$, with $O(np)$ primitive moves. The heterogeneous phase requires $O(mn + m^2 + 25m^2n + 4m^2t^2)$ time and $O(22m^2p + 4m^2t^2)$ moves. The size of the robot in modules is n , m is the number of modules with invalid type, t is an upper bound on the length of the longest tunnel, and p is an upper bound on the length of a surface path. Since path lengths are bounded by the size of the robot, $t \leq p \leq n$. Because we are interested in systems with no central controller, we present both centralized and decentralized versions of CTS.

C. Outline

We now present the CTS algorithm. We define the tunnel procedure in Sec. II, and the heterogeneous phase in Sec. III. Analysis is given in Sec. IV, improvements to the algorithm are suggested in Sec. VI, and the paper concludes with discussion and future work in Sec. VII.

II. TUNNEL PATHS WITH BENDS

The main difficulty in removing a group of modules is maintaining global connectivity in the structure. How-

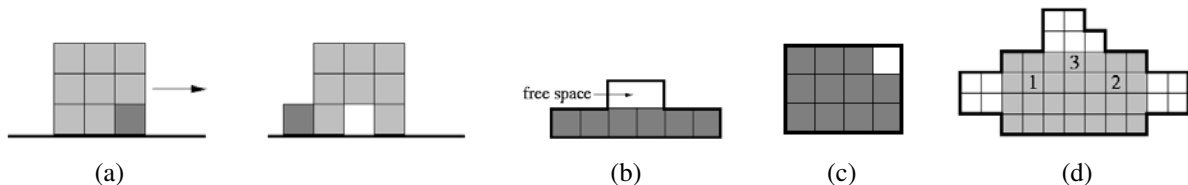


Fig. 2. Challenges of reconfiguration with free space constraints. Example (a) shows failure of greedy homogeneous reconfiguration in presence of free space constraints. Viewpoint is side view of 3D configuration. Start configuration is a cube, goal configuration is same shape translated to the right. Shaded module does not change position. Greedy algorithm fails because final free position is inaccessible via surface motion. An example of an immobile (locked) configuration is in (b). All free space is adjacent to immobile modules. Any move breaks connectivity. In (c), dark lines mark bounding region, grey squares are uniquely-typed modules, white square is free space. May not have a solution. In (d), module 1 needs to move to 3, module 2 needs to move to 1, and module 3 needs to move to 2. Shortest valid swap sequence is 2-3,2-1.

ever, a particular local substructure is sufficient to prevent disconnection. A *tunnel* is a path from a given module through intervening *tunnel modules* to the surface of the configuration. As long as all neighbors of tunnel modules are connected, no disconnection can occur when the tunnel modules are removed. We will refer to the neighbors on a given side of a tunnel as a *wall* of the tunnel. In a 3D Sliding-Cube system, a tunnel can have a maximum of four walls, but can also have less than four. A one-walled tunnel equates to surface motion. Tunnel walls are connected by using a temporary structure called a *bridge*. Bridging can be built at the mouth of the tunnel, as in Fig. 3(b), or anywhere along the tunnel walls.

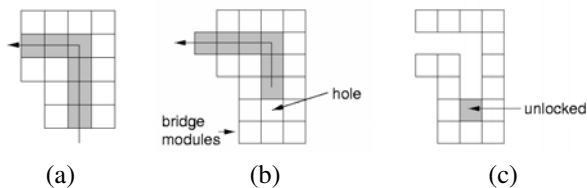


Fig. 3. Tunnel with one bend. Shaded modules are tunnel modules; unshaded modules form tunnel walls. Tunnel path is shown in (a), (b) shows virtual module relocation by shifting tunnel modules, and (c) shows unlocking a module for swapping.

In our earlier TS algorithm, all tunnel paths were straight lines. Now consider a bend in a tunnel path. The walls of the tunnel must have bends as well. See Fig. 3. Since all modules in tunnel walls are connected via bridging, tunnel modules can be removed without causing disconnection.

To find a tunnel path from a given module to free space, we can use a simple search procedure such as depth-first search (DFS). When the search begins, we determine the number of walls by examining the local neighborhood. We must maintain this number and configuration of walls during search; a search path terminates when the wall configuration changes. If the search ends without reaching the goal, it is still possible to continue searching. The portion of a tunnel with constant wall configuration is a *tunnel segment*. We will connect tunnel segments in two different ways: one for using tunnels homogeneously for virtual module relocation, and the other for using tunnels for actual module relocation as in TS.

The objective of virtual module relocation is to fill an empty lattice position by shifting modules along a path [16]. Instead of completely removing tunnel modules from

the tunnel path, we only need to move each tunnel module once. This virtually relocates the module at one end of the tunnel segment to the lattice position at the other end by shifting the tunnel modules along the tunnel path. Once a module has been virtually relocated in this manner, it is free to participate in another tunnel segment. Linking tunnel segments, we can virtually relocate a mobile module to any free lattice position.

III. ALGORITHM: CONSTRAINEDTUNNELSORT

Based on the tunneling procedure, we now present an algorithm for heterogeneous reconfiguration: Constrained-TunnelSort. As described in Sec. I-B, CTS is divided into homogeneous and heterogeneous phases. The homogeneous phase uses the greedy method, but module trajectories are planned with tunneling for virtual module relocation, which is a simpler form of the tunneling procedure in this section. We focus on the heterogeneous phase here; details of the homogeneous phase can be found in [4].

The heterogeneous phase correctly places modules by class, or in other words, sorts the goal configuration. Free space constraints are given by a set of free lattice positions. We begin by detailing a method for moving a specific module through the structure using tunneling. Because free space can be disconnected, it may not be possible to swap arbitrary pairs of modules, and so the order of swapping must be planned. We discuss a method for determining this order, and then present CTS in centralized and decentralized versions.

A. Module Relocation through Tunneling

The tunnel procedure described in Sec. II forms the basis of CTS by allowing us to exchange the position, or swap, a pair of modules. Consider a module m . To move m to a different position in the structure, we will create temporary free space in the form of a tunnel that m can traverse. As we tunnel, some number of modules will be temporarily displaced. These modules can be stored using existing free space. In some cases, a tunnel path can reach free space that is too small to hold the temporary modules. The tunnel path can continue to other free space regions, however, and is valid as long as the sum of adjacent free space is sufficient to hold all modules along the tunnel path. Once m tunnels to a given free space region, the tunnel modules can reverse

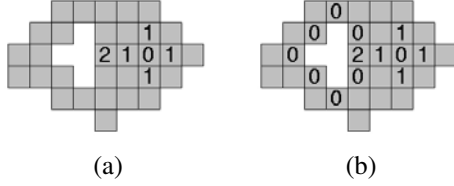


Fig. 4. Illustration of a searching for free space regions reachable via tunneling. Distance labels keep track of tunnel length, beginning at 0. When the search reaches a hole, the size of the hole is deducted from the current tunnel distance label. See (a). The next step, shown in (b), is to reset the counter to 0 and add the modules surrounding the hole as BFS children.

their movements. Repeating this for a second module m' allows m and m' to exchange positions. The remaining structure is then returned to its previous configuration.

To search for tunnel paths, we will use a simple graph search augmented with additional information. The idea is to begin a search from the start module m , and for every free space region we visit, we compute whether there is enough free space for the tunnel to terminate at that region. Searching the entire graph, we can compute all free space regions that m can use for swap space. This algorithm is listed as Algorithm 1.

Algorithm 1 Searching for free space reachable from a given module.

- 1: Initialize d to 0 for all modules
 - 2: BFS from m
 - 3: **for** each module m_i visited **do**
 - 4: $d_i = d_{i-1} + 1$
 - 5: **if** m_i is adjacent to free space s_i **then**
 - 6: **if** size of $s_i \geq d_i$ **then**
 - 7: add edge between m and s_i
 - 8: add modules adjacent to s_i as BFS children of m
 - 9: set $d = d_i - \text{size of } s_i$
 - 10: **if** $d < 0$ **then**
 - 11: $d = 0$
-

We begin breadth-first search (BFS) at module m . We maintain lattice distance d_i for every module m_i we visit. At m , $d = 0$. When we move from a module m_i to a child m_{i+1} , we increment d such that $d_{i+1} = d_i + 1$. When we reach a module adjacent to free space, we must perform some additional computation. First, if the size of the free space region is greater than d , then m can safely tunnel into this hole. Now, to continue the search, we adjust d by subtracting the size of the free space region. This accounts for the fact that we can deposit tunnel modules into this free space. Now we add all modules along the surface of this hole as children of m_i , and we continue until the entire graph is searched. See Fig. 4 for an example.

We repeat this procedure beginning at every invalid module. Now we have a connectivity graph of modules to free space. Our goal, however, is to determine module connectivity. This is easily computed by connecting all modules adjacent to the same free space node. After the swap sequence is determined, we can execute it by

translating each swap into a motion plan.

B. Determining Swap Sequence

As illustrated earlier in Fig. 2(d), disconnected free space can prevent us from swapping arbitrary modules since we can only swap modules that can reach the same free space region through tunneling. We must therefore plan a swap order. Pseudocode for this procedure is listed as Algorithm 2. We first build graph $G = (V, E)$ where set V is the set of all modules, and set E has an edge between each pair of modules that can swap with each other. The details of determining “swappability,” or connectivity between modules, are not relevant to the swap sequence algorithm. This abstraction allows the swap sequence algorithm to be useful in more general contexts. For this application, we give a specific method for computing swappability in the next section (Sec. III-C). For now we will assume a method exists and continue by assigning each vertex in V a color corresponding to module type.

Algorithm 2 Approximating optimal swap sequence. Our solution is a d -approximation.

- 1: Build module connectivity graph for invalid modules
 - 2: Compute minimum diameter spanning tree
 - 3: **for** each vertex v in post-order **do**
 - 4: Search from parent for vertex v' with goal color
 - 5: Exchange v with v' by swapping along search path
 - 6: Output series of swaps
-

Our goal is to modify G such that $G = G'$, where G' is isometric to G but the vertex colors correspond to types in the goal configuration. In other words, G represents the start configuration and G' represents the goal configuration. We modify G by swapping colors of adjacent vertices.

To minimize swaps, we will find a spanning tree and perform a post-order tree traversal. At each vertex we visit, we adjust the color to match G' . To do this, we begin at the parent vertex and use BFS to traverse nodes until we find a match. Then we swap colors in reverse order along the search path. This results in a sequence such as that shown in Fig. 5.

Since the colors in G are a permutation of those in G' , there exists a swap sequence that makes $G = G'$. Once a vertex is fixed, it is never modified since all searches

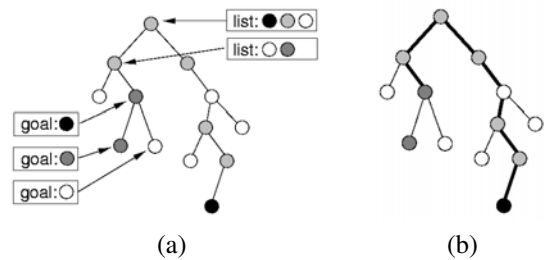


Fig. 5. Example spanning tree. In (a), deepest nodes in left subtree already match the goal color, but their parent does not. Colors in the list associated with each node are given for selected nodes. The resulting sequence of swaps is marked by darkened edges in (b).

proceed from the parent and we visit all vertices in post-order (children, then parent). Therefore, this algorithm correctly finds a solution for all connected G .

For a tree of depth d , each vertex can require up to $2d$ swaps to fix its color. Consider the vertex with the goal color. The color is swapped with its parent up to some vertex, where it then follows a path down to the final vertex. Searching for this path naively would take $O(n)$ time per search, but we can augment the graph such that each search only takes $2d$ time with $O(n)$ extra work. The extra data stored at each node is a list of all colors contained within the subtree rooted at this node. These lists can be computed simply with a post-order tree walk. The list for a parent is the merged lists of its children. With n vertices, the running time is $2d * n = O(dn)$. Note that for sparse graphs, this bound degenerates to $O(n^2)$.

So far, we have ignored the quality of the spanning tree. Now, we know the running time of our algorithm depends on the spanning tree depth. Finding the spanning tree with minimum depth therefore minimizes the running time of our algorithm. But it turns out that we can make an even stronger statement: with a minimum-depth spanning tree our algorithm approximates optimal within a factor of $2d$.

This spanning tree, more accurately the *minimum-diameter spanning tree*, can be computed in $O(n^3)$ time generally but can be done faster with unit edge weights. A quadratic-time implementation is to execute BFS from each vertex and choose the tree with minimum diameter. Since BFS finds the shortest path tree from the chosen root, some such tree is minimum overall.

To see why our algorithm is a d -approximation, consider the complete graph K^n . Our spanning tree is a root with $n - 1$ leaves. Clearly, the optimal solution (OPT) can use cross edges not present in our tree to solve this in n time, whereas we require $2dn = 2n$ swaps. The number of cross edges OPT can use, however, is limited. We are guaranteed at least one subtree of depth d with no cross edges (else we contradict the assumption that this tree has minimum depth). Therefore OPT must make the same number of swaps that we do for this subtree. The remaining m nodes can be handled in m time by OPT, but we required $2dm$. Therefore, OPT can be no more than $2d$ faster.

C. Centralized Algorithm

We now combine tunneling and swap sequence generation to build the heterogenous phase of CTS. Initially, we build a graph, where there is a node for each module, and there is an edge between two nodes if the corresponding modules can be swapped directly. Swappability between each pair of modules is determined by the tunneling search procedure as in Sec. III-A. We then find a valid swap sequence as in Sec. III-B and execute the swaps using tunneling. This approach is summarized in pseudocode as Algorithm 3.

Assuming we have stored the tunnel paths generated earlier, we swap using Algorithm 4. This algorithm can be understood as a straightforward modification of TS. Instead of only swapping using the crust, now we can swap in any

Algorithm 3 Centralized CTS: heterogeneous reconfiguration with severe free space constraints.

- 1: Execute greedy homogeneous phase with virtual module relocation via tunneling
 - 2: Build connectivity graph of swappable modules
 - 3: Search graph for feasible swap sequence using Algorithm 2
 - 4: **if** no sequence exists **then**
 - 5: Fail
 - 6: **else**
 - 7: Execute swaps using extended Tunnel procedure
-

free space region. Also, instead of straight tunnels, we use tunnel paths with bends.

Algorithm 4 Swapping modules m_1 and m_2 using free space region s_{free} .

- 1: Bridge m_1
 - 2: **for** each segment along tunnel path **do**
 - 3: Move tunnel modules along path into adjacent free space
 - 4: Move m into s_{free}
 - 5: Replace tunnel modules
 - 6: Tunnel m_2 into s_{free}
 - 7: Move m_1 into old position of m_2
 - 8: Replace tunnel modules
 - 9: Recreate m_1 's tunnel
 - 10: Move m_2 into m_1 's original position
 - 11: Replace tunnel modules
-

D. Decentralized Algorithm

We now develop a decentralized version of the heterogeneous phase, based on message-passing. Our choices of algorithms for MST and path planning generally use local information so adaption to a decentralized version is natural. We replace BFS, which is difficult to implement in a distributed way, with iterative deepening search. This allows us to compute shortest paths as in BFS and also to have the ease of distributed implementation as in depth-first search.

In order to specify decentralized algorithms in pseudocode, we invented a format loosely based on message-passing algorithms by Lynch [12]. Pseudocode is divided into three sections: *State*, *Messages*, and *Procedures*. The State section lists module-level state variables. The Messages section defines message types along with associated actions to be performed upon receipt. The term *message-handler* is equivalent to an action. The Procedures section lists code organized into procedures to be shared across message-handlers, or code that is too large to fit nicely within an action definition.

Algorithm 5 defines our solution. A copy of this code is assumed to execute on each module simultaneously, and we assume that all messages are delivered eventually by the underlying network protocol. The algorithm begins when a

start message is sent at the termination of the homogeneous phase. This initial message arrives at a single module, and algorithm execution commences. From here, the local MST computation happens in parallel, followed by sequential swapping according to a traversal of the global MST.

Algorithm 5 Heterogeneous reconfiguration with severe free space constraints, decentralized.

```

1: State:
2: type, my type label
3: goal_type, type in goal configuration at my current position
4: leader, true if global MST is rooted at me (initially false)
5:
6: Messages:
7: start, sent to begin execution (arrives at exactly one module)
8:   Action: Broadcast MST.
9: MST, sent to build minimum diameter spanning tree over
   graph of swappable modules, rooted at this module
10:  Action: Execute MST().
11: MST_done(cost c), sent to announce cost of MST and begin
   next phase
12:  Action: If heard from all modules and my cost is min,
   set leader to true. DFS-send(validate) over MST. When
   DFS-send returns, broadcast done.
13: validate, sent to fix color of module
14:  Action: Execute validate().
15: done, sent to signal algorithm termination
16:  Action: Clean up any leftover state, start next task.
17:
18: Procedures:
19: MST()
20:   if type = goal_type, broadcast MST cost as null
21:   search free space to find holes we can reach
22:   build MST
23:   broadcast MST cost
24: validate()
25:   search for matching color
26:   execute swaps along search path
27:   return when color is correct
28: DFS-send(message)
29:   send message to first child, wait for response
30:   repeat for all children and compute result
31:   send result in return message to parent

```

The *start* message-handler simply broadcasts a command to begin the minimum-diameter spanning tree (MST) computation. This procedure will execute on all modules in parallel. First, if the module is already valid (current type equals goal type), then there is nothing to do. Valid modules still need to broadcast this to the rest of the system, however, so later leader election will take place. Otherwise, the first step in building the MST is to compute a connectivity graph. This is implemented using iterative deepening search over all modules, using the same scheme as in the centralized version. At the end of this step, the originating module has a list of holes it can reach, along with path costs for each hole. The next step actually builds the MST. We will simulate BFS here by building a list of modules reachable in one hop, then two hops, etc. For one hop, we send out a list of reachable free space regions. Each module that can reach the same free space, and thus swap, responds with a list of its free space regions. Now we iterate through this list and request a similar list of

modules for each. Removing duplicates, this becomes the new list of two-hop modules. We repeat until all modules are visited. When done, we broadcast the maximum hop count as the MST cost.

After all MSTs are computed, the module with the minimum MST cost begins the next step of the algorithm. In the centralized version, we determined swap sequence by using a post-order traversal of the MST. Here, we can use DFS-order to implement this. The message-handler for *validate* controls the procedure to move a valid module into position, thereby validating the type. Note that as part of MST determination, we maintain the ID of the MST parent (think of this as a parent pointer), and also a list of all module colors in our MST subtree. Search proceeds by sending a message to the parent, which checks for the goal color in its subtree list. If the color exists, it sends the message to its MST children. Otherwise, it sends the message to its MST parent. This implements the same search order as in the centralized version. When a match is found, the matching module then initiates swaps following the resulting swap sequence. When it swaps with the final module, the *validate* message is then propagated to the next module to continue reconfiguration. When all swaps are done, the algorithm terminates.

Unlike the centralized version, which computes the entire swap sequence for all modules and then executes it, the decentralized algorithm interleaves these two operations. The details of controlling a single swap follow the order described in Algorithm 4 using message passing. The module to be unlocked acts as a local controller, and sends messages out along the tunnel path that cause tunnel modules to move into available free space. The messages that synchronize this process are given in [7].

IV. ANALYSIS

Correctness and completeness for the homogeneous phase are proved in [4]; we do not include this analysis here. Instead, we provide complexity analysis and discuss completeness for the heterogeneous phase. Planning and plan complexity for the decentralized and centralized versions is equivalent.

Complexity analysis is as follows. Building the swap graph takes $O(mn)$ time for m modules that are out of position and $O(n)$ search time per module. The swap sequence can then be computed in $O(m^2)$ time. Swap requires time for bridging, unlocking two modules, and then replacing tunnel modules. This is $O(25n + 4t^2)$ time for one swap, where t is the length of the longest tunnel path [4]. We have a maximum of m^2 swaps, so total time spent is $O(mn + m^2 + 25m^2n + 4m^2t^2)$.

The number of moves to swap a module is the number of time steps minus time spent searching. Search time is $O(2n)$ for finding tunnel paths plus $O(n)$ to initially find a module to swap. For a maximum surface path length p , the number of moves in a swap is thus $O(22p + 4t^2)$. Total moves with m^2 swaps is $O(22m^2p + 4m^2t^2)$. Tunneling can be parallelized by moving tunnel modules at the same time instead of sequentially, reducing actuation time by a

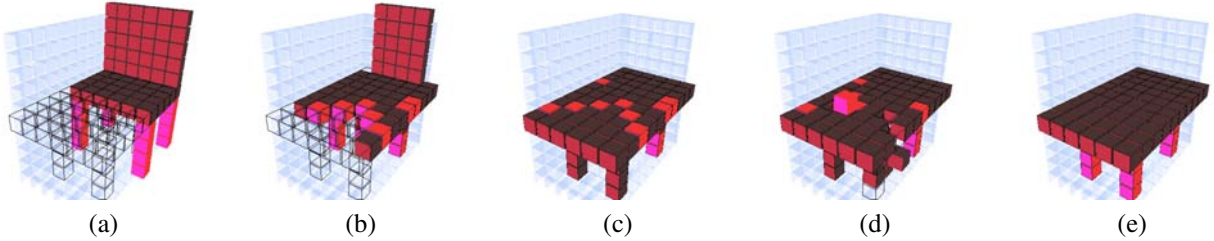


Fig. 6. Implementation example of reconfiguration with simple obstacles using SRSim simulator, planned by CTS. The start and goal configurations are positioned tightly against flat obstacles, as if placed in the corner of a room (obstacles only partially shown). Module trajectories do not penetrate rigid surfaces during reconfiguration, but our previous algorithms cannot model such constraints since they require at least one module-width of free space around the entire structure. Module shading indicates type; modules in the legs of table and chair configurations have same type. Configuration after end of homogeneous phase is shown in (c). Some modules are incorrectly placed by type. This is adjusted by the heterogeneous phase, resulting in the final configuration shown in (e).

factor of t . Incidentally, the worst-case upper bound for the homogeneous phase is $O(n^2)$ and $O(np)$ moves [4].

If $m = t = p = n$, then these bounds degenerate to $O(n^4)$. For average values of t and p , and smaller m , the bound is reasonable. For example, with $m = t = p = O(\sqrt{n})$, the upper bound is $O(n^2)$ time and moves.

This algorithm is not guaranteed to solve all problem instances. However, we can solve instances with sufficient free space:

Theorem 1: Algorithm CTS produces a feasible plan for a given problem instance if the graph of free space regions is connected, and all modules can reach a free space region through tunneling.

Proof: If a module can reach free space via tunneling, it can be moved into this free space region by the specification of the tunneling procedure. If all such free space regions are connected, a module can traverse them using tunneling. Since all modules can reach some free space region, all modules can reach all possible goal positions and CTS will produce a feasible plan. ■

Specifically, instances containing one large free space region can be solved. For example, a single free space region with size equal to the diameter of the module connectivity graph is sufficient. The crust used by TS is another example.

It is important to note that the tunnel search procedure places additional constraints on the instances we can solve. A tunnel path through a single chain of modules with no surrounding free space cannot be executed without breaking connectivity. This implies a tunnel with zero walls, which would not be found by our search procedure.

Although there is no constant-time procedure for determining whether a particular instance is solvable, Theorem 1 suggests a polynomial-time decision algorithm. We can build the swap graph in $O(n^2)$ time total, since each module requires $O(n)$ time for tunnel path searching. If the graph is disconnected, the algorithm will fail.

V. IMPLEMENTATION

We are currently implementing CTS using a Java3D-based simulator we constructed called *SRSim* [1]. Initial results are shown in Fig. 1 and Fig. 6. The chair-table

reconfiguration was planned in previous work using an out-of-place algorithm [6], but here we show this reconfiguration on a rigid surface against two flat obstacles planned by CTS. The configuration at the end of the homogeneous phase (Fig. 6(c)) has correct shape but some modules are incorrectly placed by type. This is adjusted by the sorting phase to form the final configuration (Fig. 6(e)). Free space is constrained to prevent modules from colliding with the rigid surfaces below, behind, and to one side of the configurations. This type of constraint is not possible with our previous planners because they plan module trajectories through free space surrounding the union of the start and goal shapes.

VI. IMPROVEMENTS

A number of improvements can be made to reduce the number of moves at the expense of added computation steps. This is a good trade-off since actuation in SR systems is generally much more costly (in terms of time) than computation. One idea is to find minimum-bend paths for module trajectories [5]. This reduces actual moves since straight-line paths require less moves than turns in Sliding-Cube instantiations.

Another area for improvement is the bridging operation. Any position along a tunnel wall adjacent to another tunnel wall can be used for bridging. Searching from each of these positions to find the closest mobile module adds a $O(n)$ factor in planning, but potentially reduces the number of moves since it minimizes bridging moves.

During the homogeneous phase, we do not consider module type. However, the plan complexity of the heterogeneous phase is dependent on the number of type errors in the configuration. We can improve this by greedily choosing modules of the correct type during the homogeneous phase. It is not possible to do this for every module, however.

Finally, the tunneling trajectory planning can be replaced by other trajectory primitives that are specialized for certain structures. For example, a tight gait that moves modules in a local cycle can be used in dense configurations [5]. This would allow CTS to solve extremely tightly constrained instances, and a tunnel would require moves linear in the

tunnel length as opposed to quadratic. But, this does not work for sparse configurations.

VII. DISCUSSION AND FUTURE WORK

In this paper we developed a novel heterogeneous re-configuration algorithm composed of homogeneous and heterogeneous phases. The homogeneous algorithm is the first in-place solution with free space constraints and configurations with holes for modules with surface motion as the actuation primitive. The $O(n^2)$ worst-case running time is asymptotically optimal.

Not all problem instances are solvable by CTS. We do not have an enumeration of all special cases, but characterized instances that are solvable in terms of size and connectedness of free space. There is also a $O(n^2)$ time decision algorithm to detect whether a given instance is solvable.

The homogeneous phase can also be used for locomotion among obstacles. Locomotion would occur via a series of goal configurations, possibly generated dynamically, that move the robot along a path. Other locomotion algorithms, based on cellular-automata rules, are simpler but require assumptions such as constant width or height in order to prevent disconnection [1].

The worst-case analysis presented for the heterogeneous phase is misleading when considered out of context. The worst-case running time of $O(n^4)$ occurs only when all modules must be relocated and all paths are very long (n modules). A more realistic estimate of path length is $O(\sqrt{n})$ or $O(\sqrt[3]{n})$. This reduces the tunnel cost to $O(n)$ from $O(n^2)$. Also, $O(n^2)$ swaps can only be required for uniquely typed modules and a large number of very small free space regions. Each module would have to “hop” through n free space regions before reaching a valid location, which is unlikely in practice. A better average-case measure is $O(n)$ swaps total, since the number of types in a system is likely to be constant and the number of free space regions is likely to be far less than n . This yields an average-case running time of $O(n^2)$, which is competitive with other algorithms.

The most important area for future work in re-configuration planning is minimizing the actual number of moves in a plan by using added computation time. This optimality problem is very difficult, but an interesting question is whether a polynomial-time approximation is possible. Another promising avenue is to find special cases in which the number of required moves is provably small.

ACKNOWLEDGMENT

This work was done in the Dartmouth Robotics Lab. Thanks to Ramgopal Mettu for discussion of the optimal swap sequence algorithm. Support for this work was provided through NSF awards IRI-9714332, EIA-9901589, IIS-9818299, IIS-9912193, EIA-0202789 and 0225446, and ONR award N00014-01-1-0675. National ICT Australia is funded by the Australian Government’s Backing Australia’s Ability initiative, in part through the Australian Research Council.

REFERENCES

- [1] Z. Butler, K. Kotay, D. Rus, and K. Tomita. Generic decentralized locomotion control for lattice-based self-reconfigurable robots. *International Journal of Robotics Research*, 23(9), Sept. 2004.
- [2] Z. Butler and D. Rus. Distributed motion planning for modular robots with unit-compressible modules. *International Journal of Robotics Research*, 22(9):699–716, Sept. 2003.
- [3] D. Christensen, E. Østergaard, and H. Lund. Meta-module control for the atron self-reconfigurable robotic system. In *Proceedings of the The 8th Conference on Intelligent Autonomous Systems (IAS-8)*, pages 685–692, 2004.
- [4] R. Fitch. *Heterogeneous Self-Reconfiguring Robotics*. PhD thesis, Dartmouth College, 2004.
- [5] R. Fitch, Z. Butler, and D. Rus. 3D rectilinear motion planning with minimum bend paths. In *Proc. of the Int’l Conf. on Intelligent Robots and Systems*, 2001.
- [6] R. Fitch, Z. Butler, and D. Rus. Reconfiguration planning for heterogeneous self-reconfiguring robots. In *Proc. of IROS*, 2003.
- [7] R. Fitch, Z. Butler, and D. Rus. In-place distributed heterogeneous reconfiguration planning. In *7th International Symposium on Distributed Autonomous Robotic Systems (DARS’04)*, 2004.
- [8] Robert A. Hearn and Erik D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, to appear. Special issue “Game Theory Meets Theoretical Computer Science”.
- [9] K. Hosokawa, T. Tsujimori, T. Fujii, H. Kaetsu, H. Asama, Y. Koruda, and I. Endo. Self-organizing collective robots with morphogenesis in a vertical plane. In *Proc. of IEEE ICRA*, pages 2858–63, 1998.
- [10] K. Kotay. *Self-Reconfiguring Robots: Designs, Algorithms, and Applications*. PhD thesis, Dartmouth College, Computer Science Department, 2003.
- [11] K. Kotay and D. Rus. Locomotion versatility through self-reconfiguration. *Robotics and Autonomous Systems*, 26:217–32, 1999.
- [12] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, 1996.
- [13] S. Murata, E. Yoshida, K. Tomita, H. Kurokawa, A. Kamimura, and S. Kokaji. Hardware design of modular robotic system. In *Proc. of the Int’l Conf. on Intelligent Robots and Systems*, pages 2210–7, 2000.
- [14] A. Pamecha, C.-J. Chiang, D. Stein, and G. Chirikjian. Design and implementation of metamorphic robots. In *Proc. of the 1996 ASME Design Engineering Technical Conf. and Computers in Engineering Conf.*, 1996.
- [15] A. Pamecha, I. Ebert-Uphoff, and G. Chirikjian. Useful metrics for modular robot motion planning. *IEEE Trans. on Robotics and Automation*, 13(4):531–45, 1997.
- [16] D. Rus and M. Vona. Crystalline robots: Self-reconfiguration with unit-compressible modules. *Autonomous Robots*, 10(1):107–24, 2001.
- [17] K. Stoy and R. Nagpal. Self-reconfiguration using directed growth. In *7th International Symposium on Distributed Autonomous Robotic Systems (DARS’04)*, 2004.
- [18] Cem Ünsal and Pradeep Khosla. Mechatronic design of a modular self-reconfiguring robotic system. In *Proc. of IEEE ICRA*, pages 1742–7, 2000.
- [19] S. Vassilvitskii, M. Yim, and J. Suh. A complete, local and parallel reconfiguration algorithm for cube style modular robots. In *Proc. of IEEE ICRA*, 2002.
- [20] J. Walter, J. Welch, and N. Amato. Concurrent metamorphosis of hexagonal robot chains into simple connected configurations. *IEEE Transactions on Robotics and Automation*, 18(6):945–956, 2002.
- [21] M. Yim, Y. Zhang, J. Lamping, and E. Mao. Distributed control for 3D shape metamorphosis. *Autonomous Robots*, 10(1):41–56, 2001.