Distributed reconfiguration of hexagonal metamorphic robots

Jennifer E. Walter, Jennifer L. Welch, and Nancy M. Amato

Abstract— **The problem addressed is the distributed reconfiguration of a metamorphic robotic system composed of any number of two dimensional hexagonal robots (modules) from specific initial to specific goal configurations. The initial configuration considered is a straight chain of robotic modules, while the goal configurations considered satisfy a more general "admissibility" condition. A centralized algorithm is described for determining whether an arbitrary goal configuration is admissible. We prove this algorithm correctly identifies admissible goal configurations and finds a reconfiguration surface, or "substrate path" within any admissible goal configuration. The main result of the paper is a distributed algorithm for reconfiguring a straight chain into an admissible goal configuration. Different heuristics are proposed to improve the performance of the reconfiguration algorithm and simulation results demonstrate the use of these heuristics.**

Keywords—**Metamorphic robots, distributed reconfiguration**

I. INTRODUCTION

A topic of recent interest in the field of robotics is the development of motion planning algorithms for robotic systems composed of a set of robots (modules) that change their position relative to one another, thereby reshaping the system. A robotic system that changes its shape due to individual robotic motion has been called *self-reconfigurable* [5] or *metamorphic* [2].

A self-reconfigurable robotic system is a collection of independently controlled, mobile robots, each of which has the ability to connect, disconnect, and move around adjacent robots. Metamorphic robotic systems, a subset of self-reconfigurable systems, are further limited by requiring each module to be identical in structure, motion constraints, and computing capabilities. Typically, the modules have a regular symmetry so that they can be packed densely, i.e., packed so that gaps between adjacent modules in a configuration that densely packs the plane are as small as possible. In these systems, robots achieve locomotion by moving over a substrate composed of one or more other robots. The mechanics of locomotion depends on the hardware and can include module deformation to crawl over neighboring modules [3], [9] or to expand and contract to slide over neighbors [10]. Alternatively, moving robots may be constrained to rigidly maintain their original shape, requiring them to roll over neighboring robots [6], [13], [14].

Shape changing in these composite systems is envisioned as a means to accomplish various tasks, such as bridge building, satellite recovery, or tumor excision [9]. The complete interchangeability of the robots provides a high degree of system fault tolerance. Also, self-reconfiguring robotic systems are potentially useful in environments that are not amenable to direct human observation and control (e.g., interplanetary space, undersea depths).

The motion planning problem for a metamorphic robotic system is to determine a sequence of robot motions required to go from a given initial configuration (I) to a desired goal configuration (G).

Many developers of self-reconfigurable robotic systems [5], [6], [7], [9], [10], [12], and [13] have devised motion planning strategies specific to the hardware constraints of their prototype robots. Most of the existing motion planning strategies rely on centralized algorithms to plan and supervise the motion of the system components [1], [3], [5], [9], [10], [12]. Others use distributed approaches which rely on heuristic approximations and require communication between robots in each step of the reconfiguration process [6], [7], [13], [14].

We focus on a system composed of planar, hexagonal robotic modules as described by Chirikjian [3]. We consider a distributed motion planning strategy, given the assumption of initial global knowledge of G. Our distributed approach offers the benefits of localized decision making and the potential for greater system fault tolerance. Additionally, our strategy requires less communication between modules than other approaches. We have previously applied this approach to the problem of reconfiguring a straight chain to an intersecting straight chain [11].

In this paper we address the problem of distributed reconfiguration from a straight chain of robots to goal configurations that satisfy a more general "admissibility" condition. A centralized algorithm is described for determining whether an arbitrary goal configuration is admissible, and if so, finding a path with certain properties. The main result of the paper is a distributed algorithm for reconfiguring a straight chain into an admissible goal configuration, which uses the path found by the previous algorithm. Different heuristics for choosing the path are proposed to improve the performance of the reconfiguration algorithm and the performance of these heuristics is explored through simulation.

II. RELATED WORK

Chirikjian [3] and Pamecha [9] discuss centralized algorithms for planar hexagonal modules that use the distance between all modules in I and the coordinates of each goal

Department of Computer Science, Texas A&M University, College Station, TX 77843-3112. FAX: 847-8578 Walter (contacting author): E-mail: jennyw@cs.tamu.edu, Phone: (979)845-3937. Welch: E-mail: welch@cs.tamu.edu, Phone: (979)845-5076. Amato: E-mail: amato@cs.tamu.edu, Phone: (979)862-2275

position to accomplish the reconfiguration of the system. Pamecha et al. [9] define the distance between configurations as a metric and apply this metric to system self-reconfiguration using a simulated annealing technique to drive the process towards completion. Upper and lower bounds on the number of moves for reconfiguration between general shapes are given by Chirikjian [3]. Lower bounds for the general case are obtained by finding a perfect matching between modules in I and positions in G such that the sum of the distances between pairs is minimized.

Centralized motion planning strategies for systems of two dimensional robotic modules are examined by Nguyen et al. [8] and analysis is presented for the number of moves necessary for specific reconfigurations. The authors show that the absence of a single excluded class of initial configurations is sufficient to guarantee the feasibility of motion planning for a system composed of a single connected component.

A centralized motion planning strategy for three dimensional cubic robots is presented by Rus and Vona [10]. In this paper, the proposed modules incorporate an actuator mechanism that causes module expansion and contraction, resulting in the sliding movement of a module over its neighbors. A centralized algorithm which takes $O(n^2)$ time to reconfigure a system of n modules is presented.

Centralized algorithms for decomposing a system of modules into a hierarchy of two dimensional substructures are presented by Casal and Yim [1]. Reconfiguration of the system involves connectivity changes within and between these substructures, along with substructure relocation. The paper concentrates on the decomposition algorithms and does not present algorithms for motion planning within substructures.

A distributed approach is taken by Murata et al. to reconfigure a system of two dimensional hexagonal modules [6], and a system of three dimensional cubic modules [7]. In these approaches, each module senses its own connection type and classifies itself by the number of modules that it physically contacts. Modules use a formula that relates their connection type to the set of connection types in the goal configuration to quantify their *fitness* to move. Modules communicate with physical neighbors to ensure that only the modules that have fitness greater than the local fitness average move in the same time step, choosing a direction at random. These distributed algorithms use random local motions to converge toward the goal configuration, a slow process that appears impractical for large configurations. These schemes also ignore the consequences of module collision and do not distinguish the relative location of modules in the plane, i.e., two configurations are the same if the modules composing them have the same connections.

Another distributed reconfiguration algorithm, for three dimensional rhombic dodecahedral modules, is presented by Yim et al. [13] In this strategy, each module uses local information about its own state (the number and location of its current neighbors) and information about the state of its neighbors obtained through inter-module communication to heuristically choose moves that lower its distance to the goal configuration.

Several heuristic approximation algorithms for distributed motion planning of three dimensional rhombic dodecahedral robots are presented by Zhang et al. [14] In this two phase approach, modules use neighbor-to-neighbor communication in the first phase to achieve a semi-global view of the initial configuration, using as many rounds as necessary to avoid violation of module motion constraints prior to each phase of movement.

III. Our Approach

This paper will examine distributed motion planning strategies for a planar metamorphic robotic system undergoing a reconfiguration from a straight chain to a goal configuration satisfying certain properties. In our algorithms, robots are identical, but act as independent agents, making decisions based on their current position and the sensory data obtained from physical contacts with adjacent robots. Our purpose is to seek an understanding of the necessary building blocks for reconfiguration, starting with algorithms in which no messages need to be passed between participating robots during reconfiguration. Reconfiguration in certain scenarios, like the ones presented in this and our earlier paper [11], can be accomplished using algorithms that do not require any message passing. Therefore, our algorithms are more communication efficient than the distributed approaches of [6], [13] and [14]. Another contribution of our work is how our system model abstracts from specific hardware details about the robots.

In this paper, we consider two dimensional, hexagonal robots like those described by Chirikjian [2], using the same definition of lattice distance between robots in the plane. Our proposed scheme uses a new classification of robot types based on connected edges similar to the classification used by Murata et al. [6] for connected vertices. In the algorithms presented in this paper, each robot independently determines whether it is in a movable state based on the cell it occupies in the plane, the locations of cells in the goal configuration, and on which sides it contacts neighbors. Robots move from cell to cell and modify their states as they change position. Since the robots know the coordinates of the goal cells, we show that each of them can independently choose a motion plan that avoids module collision.

In this paper we also present precise conditions for admissible goal configurations based on the motion constraints of our robots. We present an algorithm that ensures these admissibility conditions and prove that this algorithm correctly identifies admissible goal configurations. The admissibility conditions presented in this paper differ from those presented by Rus and Vona [10] and Nguyen et al. [8]. In the first case, this difference is due to module shape and motion constraints, and, in the second case, the difference is due to assumptions on module motion, as will be explained in Section V.

In Sect. IV we describe the system assumptions and the problem definition. Section V contains a centralized al-

gorithm that determines whether or not a given configuration is admissible. Section VI presents and analyzes a distributed algorithm for reconfiguring a straight chain to an admissible goal configuration. In Sect. VII we present simulation results comparing the performance of our algorithm using different heuristics. Section VIII provides a discussion of our results and future work.

IV. SYSTEM MODEL

A. Coordinate System

The plane is partitioned into equal-sized hexagonal cells and labeled using the coordinate system shown in Fig. 1, as in Chirikjian [2].

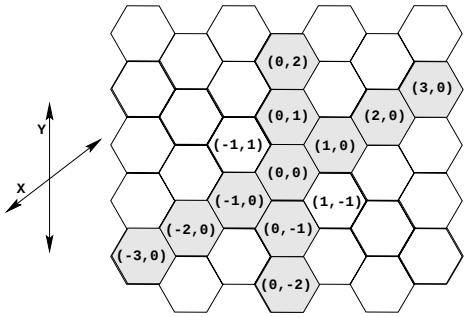

Fig. 1. Coordinates in a system of hexagonal cells.

Given the coordinates of two cells, $c_1 = (x_1, y_1)$ and $c_2 = (x_2, y_2)$, we define the *lattice distance*, LD, between them as follows: Let $\Delta x = x_1 - x_2$ and $\Delta y = y_1 - y_2$. Then

$$LD(c_1, c_2) = \begin{cases} \max(|\Delta x|, |\Delta y|) & \text{if } \Delta x \cdot \Delta y < 0, \\ |\Delta x| + |\Delta y| & \text{otherwise.} \end{cases}$$

The lattice distance describes the minimum number of cells a module would need to move through to go from cell c_1 to cell c_2.

B. Assumptions About the Modules

Our model provides an abstraction of the hardware features and the interface between the hardware and the application layer.

- Each module is identical in computing capability and runs the same program.
- Each module is a hexagon of the same size as the cells of the plane and always occupies exactly one of the cells.
- Each module knows at all times:
  - its location (the coordinates of the cell that it currently occupies),
  - its orientation (which edge is facing in which direction), and
  - which of its neighboring cells is occupied by another module.

Modules move according to the following rules.

1. Modules move in lockstep rounds.
2. In a round, a module M is capable of moving to an adjacent cell, C_1, iff (see Fig. 2 for an example)
   (a) cell C_1 is currently empty,
   (b) module M has a neighbor S that does not move in the round (called the *substrate*) and S is also adjacent to cell C_1, and
   (c) the neighboring cell to M on the other side of C_1 from S, C_2, is empty.
3. Only one module tries to move into a particular cell in each round.

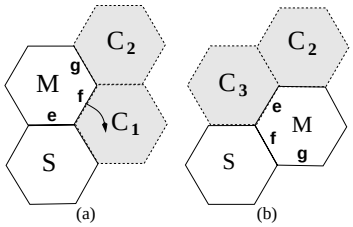

Fig. 2. Before (a) and after (b) module movement: M is moving, S is substrate, C_1, C_2, and C_3 are empty cells.

If the algorithm does not ensure that each moving module has an immobile substrate, as specified in rule 2(b), then the results of the round are unpredictable. Likewise, the results of the round are unpredictable if the algorithm does not ensure rule 3.

C. Problem Definition

We want a distributed algorithm that will cause the modules to move from an initial configuration, I, in the plane to a known goal configuration, G.

V. ADMISSIBLE CONFIGURATIONS

In this section we define admissible configurations and describe a centralized algorithm that tests whether a given configuration is admissible.

A. Definition of Admissible Configuration

Without loss of generality, assume I is a straight chain oriented north-south, no goal cell is to the west of I, and I and G intersect in the southernmost module of I and nowhere else. The number of modules in I and the number of cells in G is n. Figure 3 gives examples of orientations of I and G that satisfy these assumptions in which $n = 6$. In this figure, cells in I are numbered with solid borders and goal cells are shaded. The assumptions concerning the orientation of I and G can be made without loss of generality because, if I is a straight chain that is not oriented in this way, the algorithms presented in [11] for straight chain to straight chain reconfiguration can be used to reorient I in relation to G.

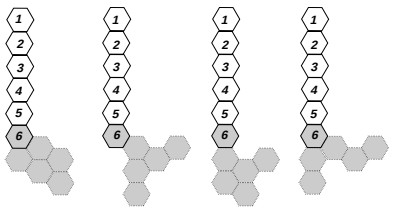

Fig. 3. Example orientations of I and G.

Let $G_1, G_2, \ldots, G_m$ be the columns of G from west to east such that each column is oriented north-south and each is composed of a contiguous chain of goal cells. Figure 4(a) shows how the columns of G are labeled and gives an example of a configuration of G in which each column is a contiguous chain of goal cells. Figure 4(b) gives an example of a configuration of G in which columns G_3 and G_5 are composed of a non-contiguous chain of goal cells. Note that it is easy to check that each column of G is composed of a contiguous chain of goal cells by scanning each column of G in a preprocessing step.

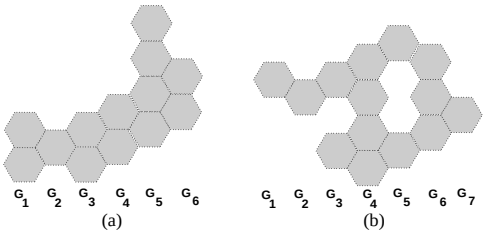

Fig. 4. Two configurations of G: (a) each column is composed of a contiguous chain of goal cells, and (b) columns G_3 and G_5 are composed of non-contiguous chains of goal cells.

Let p be a contiguous sequence of distinct cells, $c_1, c_2, \ldots, c_k$. Then

Definition 1: p is a *substrate path* if

- p begins with the cells in I, from north to south,
- subsequent cells are all in G, and
- the last cell is in the easternmost column of G (G_m).

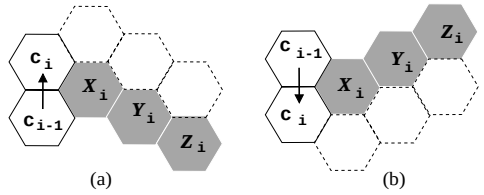

Fig. 5. Labels for north segment ending in c_i (a) and south segment ending in c_i (b) (cells that must not be goal cells are shaded).

Definition 2: A *segment* of p is a contiguous subsequence of p of length ≥ 2. In a *south segment*, each cell is south of the previous and analogously for a *north segment*.

Definition 3: p is an *admissible path* if

1. each cell in p is adjacent to the previous, *but not to the west* (i.e., consecutive higher numbered cells may not be on the northwest or southwest side of a given cell),
2. for each north segment of p ending with c_i,
   (a) cells X_i, Y_i, and Z_i are not goal cells (see Figure 5(a)) and
   (b) c_{i+1}, c_{i+2}, and c_{i+3} do not form any south segments,
   and
3. for each south segment of p ending with c_i,
   (a) cells X_i, Y_i, and Z_i are not goal cells (see Figure 5(b)) and
   (b) c_{i+1}, c_{i+2}, and c_{i+3} do not form any north segments.

In the remainder of this paper, north and south segments of p may be referred to as *vertical* segments when specific direction of the segment is not important. Segments directed to the east may be referred to as *horizontal* or *easterly* segments when specific direction is not important. Conditions 2(a) and 3(a) of Definition 3 specify where a vertical edge may be added to p relative to goal cells in the three columns to the east. Conditions 2(b) and 3(b) say that any vertical segment of p must be separated from any vertical segment in the opposite direction and to the east by at least 3 columns.

Definition 4: G is an *admissible goal configuration* if there exists an admissible substrate path in G.

Intuitively, an admissible substrate path is a chain of goal cells whose surface allows the movement of modules without collision or deadlock, provided the choices of module rotation and delay are appropriate. That is, provided the motion planning algorithm allows for adequate space between moving modules, there are no pockets or corners on the surface of the substrate path in which modules will become trapped.

The admissibility conditions for a substrate path are directly related to the degree of parallelism desired, i.e., how closely moving modules can be spaced. If moving modules are separated by only a single empty cell, they will become deadlocked in acute angle corners when running our algorithms [11]. However, acute angle intersections are very commonplace in configurations of hexagonal robots. Thus, we chose to make our algorithms applicable to a wide range of goal configurations by separating moving modules by two empty cells. Our definition of admissibility is therefore based on configuration surfaces over which moving modules with two empty cells between them can move without becoming deadlocked.

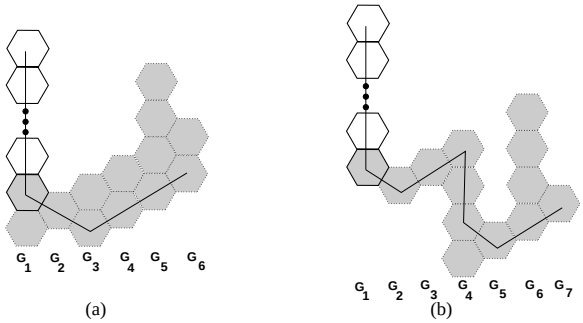

Fig. 6. Example admissible (a) and inadmissible (b) G (cells in I have solid borders and cells in G are shaded).

Figure 6(a) depicts an example of an admissible configuration of G, where the line through I and G is an admissible substrate path. Figure 6(b) depicts a configuration of G that violates admissibility condition 2. The substrate path shown is inadmissible, as is every other possible substrate path for this configuration.

Our definition of admissible classes of goal configurations differs from that presented by Rus and Vona[10] because

the modules used by these authors were cubic, with a different set of motion constraints and mode of locomotion. Even though our modules are two dimensional and hexagonal, like those of Nguyen et al. [8], our definition of admissible classes of goal configurations is different than theirs because our assumptions about module motion are different. Nguyen et al. assume that a module moves by rigid rotation around a vertex it shares with another module. Our motion constraints are similar to those presented by Chirikjian[2], where locomotion is accomplished by a combined rigid body rotation and shape transformation produced by changing joint angles.

B. Algorithms to Detect Admissible Configurations and Find Substrate Paths

Condition 1 for determining the admissibility of G can be easily accomplished by scanning G in columns from north to south, northwest to southeast, and northeast to southwest, to determine if there exists an orientation in which each G_i is contiguous. If there is no orientation in which each G_i is contiguous, then G is not admissible.

Our procedure for finding an admissible substrate path in G (condition 2 for the admissibility of G) proceeds by first constructing a directed graph H as follows:

- Label the columns of G as described in Sect. V-A, with the cells in each G_i labeled $G_{i,1}$, $G_{i,2}$,..., from north to south. Then cell $G_{1,1}$ is also in I, but no other goal cells are in I.
- Represent each goal cell as a node in the graph H. Add an extra node to the graph in the cell directly north of cell $G_{1,1}$ and call this node $G_{1,0}$. Initially there is an undirected edge between each pair of adjacent goal cells.
- The cells to the north, south, northeast, and southeast of $G_{i,j}$ are labeled $\mathrm{N}_{i,j}$, $\mathrm{S}_{i,j}$, $\mathrm{NE}_{i,j}$, and $\mathrm{SE}_{i,j}$, respectively (note that some of these cells might not be goal cells and thus are not represented in the graph).

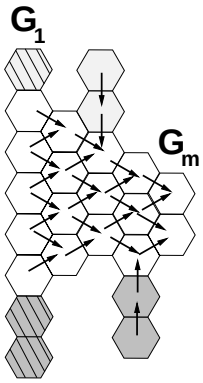

Fig. 7. Directed graph H formed by algorithm.

The first phase directs edges in the undirected graph and marks the nodes that are determined to have an admissible path to a goal cell in the easternmost column. The columns are processed from east to west. First, every node in column G_m is marked. As shown in Fig. 7, each column west of column G_m consists of three segments: (A) the north segment of nodes with no goal cells to the east (shaded light gray), (B) the central segment of nodes that have goal cells to the east (unshaded), and (C) the south segment of nodes that have no goal cells to the east (shaded dark gray). Segment (A) is initially skipped. Each node in segment (B) is given an outgoing edge to each of its *marked* east neighbors, with the exception of the situation where a NE edge would be directed toward a neighbor with an outgoing S edge or where a SE edge would be directed toward a neighbor with an outgoing N edge. Nodes in segment (C) are processed north to south. Each node is marked and given a directed edge to its north neighbor if the north neighbor is marked and if the goal cells in a local neighborhood satisfy a certain "admissibility" condition (discussed below). Finally, nodes in segment (A) are processed south to north. Similarly to segment (C), each node is marked and given a directed edge to its south neighbor if the south neighbor is marked and if the goal cells in a local neighborhood satisfy a certain "admissibility" condition (discussed below). The arrows in Fig. 7 show the edges that are directed and the direction given to the edges. The cross-hatched cells are those that remain unmarked after the algorithm has been run. The cell on the north and the two cells on the south of column G_1 do not satisfy the "admissibility" condition, so no edges are directed from these cells in the algorithm.

The *Direct_Edges* algorithm (see Figures 8 and 9) directs some of the edges in the graph, as described above. The variables used in the pseudocode are as follows:

- $onPath_{i,j}$: Boolean variable. Initially, $onPath_{i,j}$ is false for all goal cells in columns $1 \leq i \leq m-1$ and true for all nodes in column G_m. At a particular node i, the status of the $onPath_{i,j}$ variable at the nodes $\mathrm{N}_{i,j}$, $\mathrm{S}_{i,j}$, $\mathrm{NE}_{i,j}$, and $\mathrm{SE}_{i,j}$ is $onPath_{N_{i,j}}$, $onPath_{S_{i,j}}$, $onPath_{NE_{i,j}}$, and $onPath_{SE_{i,j}}$.
- x: Variable used to save the position of the southernmost cell that has not been checked by the algorithm.
- d: Direction to be checked, either N or S.
- *remove*: Set containing at most 1 goal cell coordinate. Initially, $remove = \{\emptyset\}$ at all nodes.
- *path*: List of coordinates of goal cells that are added to the substrate path.

The labels used in the *IsAdmissible* procedure are depicted in Figures 10(a) and (b).

From the *isAdmissible* procedure, we can see that if any edges at a node are directed to the east, then no edges at that node will be directed to the north or south. Also, since the cells in each column are contiguous, if an edge at a node is directed to the north, then no edge at that node will be directed to the south and vice versa. In Section V-C we will show that, after constructing H, if $onPath_{1,0} = \text{true}$, then there exists an admissible substrate path from $G_{1,0}$ to some cell in G_m because of the way H is constructed. In the next section, we show that if the algorithm fails to find an admissible substrate path with respect to $G_{1,0}$, then G does not contain such a path.

To find an admissible substrate path, we begin at node $G_{1,0}$ and move in any allowable direction (i.e., over any directed edge to a goal cell for which *onPath* is true) until reaching some goal cell in column G_m. If a node has a

For each column $i := m - 1$ downto 1 do:

```
1.  x := 1
2.  j := 1
3.  while (j ≤ |G_i|)
4.      while (G_{i,j} in north section)
5.          j++
6.      end while
7.      x := j − 1
8.      while ((j ≤ |G_i|) and
         (G_{i,j} has ≥ one adjacent node to the east))
9.          if ((G_{i,j} has node to NE) and
             (onPath_{NE_{i,j}}))
10.             if (no edge is directed S from NE_{i,j})
11.                 onPath_{i,j} := true
12.                 direct edge to NE
13.             end if
14.         end if
15.         if ((G_{i,j} has node to SE) and
             (onPath_{SE_{i,j}}))
16.             if (no edge is directed N from SE_{i,j})
17.                 onPath_{i,j} := true
18.                 direct edge to SE
19.             end if
20.         end if
21.         j++
22.     end while
23.     while (j ≤ |G_i|)
24.         if ((onPath_{N_{i,j}}) and
            (isAdmissible(S, i, j)))
25.             onPath_{i,j} := true
26.             direct edge to N
27.         end if
28.         j++
29.     end while
30.     while (x > 0)
31.         if ((onPath_{S_{i,x}}) and
            (isAdmissible(N, i, x)))
32.             onPath_{i,x} := true
33.             direct edge to S
34.         end if
35.         x−
36.     end while
37. end while
```

Fig. 8. Pseudocode for algorithm *Direct_Edges*.

Procedure *isAdmissible*(d, i, j) returns boolean

```
1.   if (X, Y, or Z is a goal cell)   //Case 1
2.       return false
3.   end if
4.   if ((A and C are goal cells) and
      ∃ an edge directed d out of Q))
5.       if (B is not a goal cell)
6.           return false     //Case 2
7.       else
8.           remove_{i,j} = {P}   //Case 3
9.       end if
10.  end if
11.  return true
```

Fig. 9. Pseudocode for Procedure *isAdmissible*.

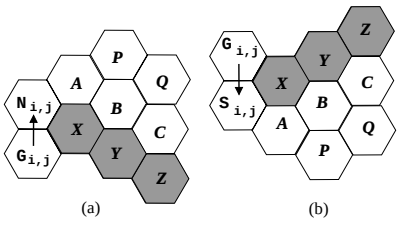

Fig. 10. Labels used in *IsAdmissible* procedure for d = S (a) and d = N (b).

directed edge to only one neighbor for which *onPath* is true (either N, S, NE, or SE), then we go in that direction. The only other possibility is that a node has two neighbors for which *onPath* is true, NE and SE. In this case, a heuristic is used to decide whether to go NE or SE. If the decision to go N or S is taken in column G_i $(i < m)$, then a particular cell in the graph two columns to the east may have *onPath* set to false (the "remove" cell calculated in *IsAdmissible*, Case 3). The choice of this edge may mean that certain later choices are no longer available.

Algorithm *Find_Path*, shown in Figure 11, is used to find and construct an admissible substrate path.

Initially, $i := 1$ and $j := 0$ and $path := \langle G_{1,0} \rangle$

```
1. while ((onPath_{i,j}) and (i < m))
2.   if G_{i,j} has an edge directed to ≥ 1
     east neighbor with onPath = true
3.       update i and j to index one such neighbor
         (heuristic choice)
4.       append G_{i,j} to path
5.   else if G_{i,j} has an edge directed to a
     north or south neighbor with onPath = true
6.       update i and j to index that neighbor
7.       append G_{i,j} to path
8.       for the goal cell in remove_{i,j}
9.           onPath := false
10.  end if
11. end while
```

Fig. 11. Pseudocode for Algorithm *Find_Path*.

C. Analysis of Algorithms Direct_Edges and Find_Path

The running time of the algorithm to find the graph H and to find an admissible substrate path is $O(n)$, since each node has a constant number of (undirected) neighbors.

Algorithm *Direct_Edges* is correct if it

- marks every cell in G that is on any admissible substrate path from cell $G_{1,0}$ to column G_m and
- directs the edges properly.

We require that algorithm *Find_Path* returns a path that ends in column G_m if and only if G is admissible.

To prove correctness, we start with some observations and claims regarding the performance of the *Direct_Edges* algorithm:

Observation 1: If a goal cell c is marked at time t by algorithm *Direct_Edges*, then c has either one or two neighboring goal cells that were marked before t and an edge is directed from c toward at least one neighbor that was marked before t.

Observation 2: If line 10 of *Direct_Edges* returns false for some goal cell in the central section, then line 16 will return true (and vice versa).

To see why Observation 2 is true, consider a cell c_i with a marked neighbor c_l to the NE (see Figure 12(a)). If c_l has an edge directed S, then c_i must have a marked neighbor c_k to the SE. It cannot be the case that c_k has an edge directed to the N, since c_k was marked before c_l, by the action of *Direct_Edges*. Therefore, line 16 of *Direct_Edges* will return true and an edge will be directed from c_i to c_k. An analogous argument can be made for a node c_i that has a marked neighbor c_l to the SE when c_l has a marked neighbor to the N.

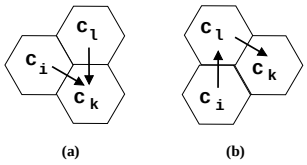

Fig. 12. Scenarios for Observation 2 and Claim 1.

Observation 3: After executing *Direct_Edges*, the following is true of the goal cells in each section of columns of H:

(a) *Central section*: Each goal cell will either be marked with at least one edge directed to the east or it will not be marked. In this section, only goal cells with no marked neighbors remain unmarked.
(b) *North section*: Each goal cell will either be marked with one edge directed to the south or it will not be marked.
(c) *South section*: Each goal cell will either be marked with one edge directed to the north or it will not be marked.

Claim 1: After executing *Direct_Edges*, no acute angle turns can be formed by any directed path from cell $G_{1,0}$ to a cell in column G_m.

Proof: The only possible acute angle turns on a directed path from cell $G_{1,0}$ to a cell in column G_m occur when

1. a N (S) edge is followed immediately by a SE (NE) edge, or
2. a NE (SE) edge is followed immediately by a S (N) edge.

Suppose, in contradiction, case 1 is allowed by algorithm *Direct_Edges* and after the execution of *Direct_Edges* there exists a cell c_i that has an outgoing edge to the N, toward cell c_l, and c_l has an outgoing edge to the SE, toward cell c_k (see Figure 12(b)). Then cell c_k must be a marked goal cell and it must be the NE neighbor of cell c_i. But then c_i would be in the central section of its column and would not have an edge directed to the N, a contradiction. An analogous argument can be made for a S edge immediately followed by a NE edge.

Case 2 is not possible because of the action of *Direct_Edges*, as stated in Observation 2. ■

Claim 2 refers to a S edge on a directed path followed to the east by a N edge. An analogous argument can be made for a N edge followed by a S edge.

Claim 2: If S and N edges occur on the same directed path in H after *Direct_Edges* finishes execution, then each S edge must be separated from the next N edge on the path by *at least* 2 easterly edges, a SE and then a NE edge.

Proof: From the action of *Direct_Edges*, we can see that a S edge cannot immediately follow a N edge on any directed path in G (or vice versa), because a single goal cell cannot be in both the north and south sections of a column. By Claim 1, any S edge can be immediately followed only by a S or SE edge and any N edge can be immediately preceded only by a N or NE edge. ■

We proceed with the proof of correctness starting with the following theorem.

Theorem 1: If G is admissible, then *Find_Path* returns a path that ends in G_m.

Proof: We begin by showing, in Lemma 1, that *Direct_Edges* will mark cells on all admissible paths leading to any cell in column G_m.

Lemma 1: For every goal cell c, if there is an admissible path from c to a cell in G_m, then algorithm *Direct_Edges* marks c.

Proof: The proof is by induction on the order in which cells are scanned by algorithm *Direct_Edges*.

Label all goal cells in G from $c_1, c_2, \ldots, c_k$, where $k = n$ as follows:

1. Start with the cells in column G_m, labeling them from $c_1, c_2, \ldots, c_q$, where $q = |G_m|$ and $q \leq k$, in increasing order from north to south.
2. For the cells in column G_j, where $1 \leq j < m$ (if $m > 1$), continue labeling the cells in increasing order from $c_{q+1}, c_{q+2}, \ldots, c_k$, in the order they are scanned by algorithm *Direct_Edges*.

For the basis of the induction, node c_1 is in column G_m. The lemma holds vacuously for all goal cells in column G_m since all goal cells in G_m are initially marked.

For the inductive hypothesis, assume the lemma holds for all cells $c_2, \ldots, c_{i-1}$ $(1 < i < k)$. We will show the lemma also holds for cell c_i. We assume c_i is a prefix of an admissible path ending in column G_m. We will show that c_i must be marked. For the remainder of this proof, refer to Figure 10(a), where $c_i = G_{i,j}$ and $c_l = N_{i,j}$.

If c_i is in column G_m, then the lemma holds vacuously, since all cells in G_m are marked.

Suppose c_i is not in column G_m. We have the following cases:

Case 1: c_i is in the south section of its column. Since we assume that there exists an admissible path from c_i to column G_m, any such path must go through c_i's north neighbor c_l, which was already scanned by algorithm *Direct_Edges* $(l < i)$. So c_l must be on some admissible path to G_m and must be marked by the inductive hypothesis. If c_i is not marked in lines 24–27 of *Direct_Edges*, it must be that *IsAdmissible* returned false in line 24, meaning that either case 1 or 2 of *IsAdmissible* was violated.

It is easy to see that case 1 of *IsAdmissible* ensures that no north edge that violates condition 2(a) of Definition 3 is directed. We need to show that if Case 2 of *IsAdmissible* returns false, then there are no paths from c_i to G_m that satisfy condition 2(b).

Referring to Figure 13, consider the pattern of segments that must exist at the start of a path from c_l to G_m if line 6 of *IsAdmissible* (Figure 9) returns false. If case 2 of *IsAdmissible* is executed, then A, Q, and C are goal cells, cells B, X, Y, and Z are not goal cells, and Q must have an edge directed south. Since Q has an edge directed south,

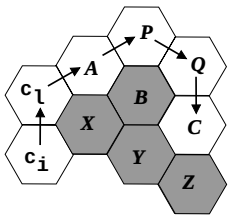

Fig. 13. Segment patterns following c_l (non-goal cells are shaded).

C was marked before Q by Observation 1. Goal cells A and C cannot have edges directed to the south or case 1 of *IsAdmissible* would not have been passed. Since A is a goal cell, c_l must have an edge directed east to A, and A must be marked or c_l would not have been marked, by Observation 3(a). Goal cell A cannot have an edge directed north, or case 1 of *IsAdmissible* would not have been passed for that edge because Q and C are goal cells. So A must have an edge directed east to P. Therefore, P must be a marked goal cell and P must have an edge directed east to marked goal cell Q, by Observation 3(a). Then every path from c_i to G_m must violate condition 2(b), since the edge from Q to C is a south edge and every path must include this edge within the third segment on any path from c_i to G_m.

Thus, if c_i is not marked, it must be that the north segment formed by c_i and c_l violates Definition 3, condition 2(a) or 2(b) for every possible admissible path from c_i to G_m. But then c_i does not form a prefix of an admissible path ending in G_m, a contradiction. Therefore, *IsAdmissible* must return true when c_i is scanned, and c_i will be marked with an edge directed toward c_l, by Observation 1.

Case 2: c_i is in the north section of its column. The argument is analogous to case 1.

Case 3: c_i is in the central section of its column. Let c_l ($l < i$) be an east neighbor of c_i through which the admissible path from c_i to G_m goes. By the inductive hypothesis, c_l is marked. Therefore, by Observation 3(a), so is c_i.

Therefore, if c_i is the prefix of an admissible path that ends in column G_m, c_i will be marked by algorithm *Direct_Edges* and, by Observation 1, edges from c_i will be directed toward the beginning of any admissible path for which it is a prefix.

■

We now continue with the proof of Theorem 1 to show that if G is admissible, then *Find_Path* returns a path that ends in G_m.

In contradiction, suppose the theorem is false and that G is admissible but *Find_Path* does not return a path that ends in G_m. Then at some point in its execution, *Find_Path* must get "stuck", i.e., it must add a cell that has no marked neighbors to *path*.

Since G is admissible, there is at least one admissible path starting with cell $G_{1,0}$ and ending in column G_m. Thus, cell $G_{1,0}$ is marked, by Lemma 1. Let $c_1, c_2, \ldots, c_j$ be the cells that are added to *path* during the execution of *Find_Path* on G, where cell c_j is in some column G_i, $1 \leq i < m$, and cell c_j has no marked neighbors.

Since c_j was added to *path*, it must have been marked. So c_j must have had at least one marked neighbor to the northeast, southeast, north, or south, in column G_i or G_{i+1}, at the time *Find_Path* started execution. Thus, at least one neighbor to the northeast, southeast, north or south of cell c_j that was marked by algorithm *Direct_Edges* must have been *unmarked* during the execution of *Find_Path*, prior to the addition of c_j to *path*.

We can see from the cell coordinates that are added to *remove* in procedure *IsAdmissible* that since c_j lost a marked neighbor to the north, south, or east, c_j's position must be within the two columns to the east of the column of G for which line 9 of *Find_Path* was executed. In other words, c_j must have lost all its marked neighboring goal cells when a north or south segment was added to *path* in column G_{i-1} or G_{i-2}.

Lemmas 2 and 3 provide useful information about the result of removing cells from an admissible path in *Find_Path*. These lemmas refer to a north segment but an analogous argument can be made for a south segment.

Lemma 2: For any north segment formed by goal cells $G_{i,j+1}$, $G_{i,j}$, if $remove_{i,j+1} \neq \emptyset$ when $G_{i,j}$ is appended to *path* in line 7 of *Find_Path*, then there are no edges directed north out of cell $G_{i,j}$ or out of any cell in columns G_{i+1}, G_{i+2}, or G_{i+3}.

Proof: The proof is by examination of the configuration of goal cells that must exist in the three columns to the east of $G_{i,j}$ when the north segment formed by $G_{i,j+1}$, $G_{i,j}$ is added to *path*, causing the cell in $remove_{i,j+1}$ to be unmarked. If $remove_{i,j+1}$ is not empty, then Case 3 of *IsAdmissible* was executed when the edge was directed from $G_{i,j+1}$ to $G_{i,j}$ in algorithm *Direct_Edges*. Refer to Figure 14 for an explanation of the labels used in this proof.

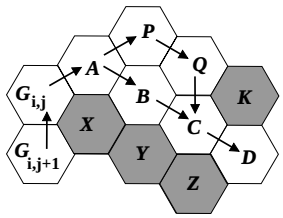

Fig. 14. Scenarios for Lemmas 2 and 3 (non-goal cells are shaded).

Since Case 3 of *IsAdmissible* was executed at the time the edge was directed from $G_{i,j+1}$ to $G_{i,j}$ in algorithm *Direct_Edges*,

- cells A, B, C, and Q are goal cells,
- X, Y and Z are not goal cells,
- P is the cell unmarked in line 9 of *Find_Path*, and
- Q has an edge directed south toward goal cell C.

Cell $G_{i,j}$ is in the central section of column G_i, with an edge directed toward A, by Observation 1. Likewise, cells A and B are in the central section of their respective columns. Since Q is in the north section, C has no marked goal cell to the northeast, in cell K. C must be in the central section of its column with an edge directed to SE neighbor D

because there are no goal cells south of C. Any cell with an edge directed north must be in the south section of its column, which is, by definition, south of the central section. Therefore, columns G_{i+1}, G_{i+2}, and G_{i+3} contain no cells with edges directed north.

Thus, if $remove_{i,j+1} \neq \emptyset$ when $G_{i,j}$ is appended to *path*, then goal cell $G_{i,j}$ cannot have an edge directed north, and neither can any goal cells in columns G_{i+1}, G_{i+2}, or G_{i+3}. ■

Lemma 3: For any north segment formed by goal cells $G_{i,j+1}$, $G_{i,j}$, if $remove_{i,j+1} \neq \emptyset$ when $G_{i,j}$ is appended to *path*, then $G_{i,j+1}$ will be a prefix of an admissible path from $G_{i,j+1}$ to column G_{i+4} after line 9 of *Find_Path* is executed.

Proof: Since *Find_Path* adds cells to *path* from west to east, then no column east of G_{i+2} will have had a cell unmarked at the time $G_{i,j}$ is appended to *path* in line 7 of *Find_Path*. Therefore, Observations 1, 2, and 3 must hold for all columns east of G_{i+2} when $G_{i,j}$ is appended to *path* in line 7 of *Find_Path*.

If $G_{i,j}$ is appended to *path*, it must be marked. Referring to labels used in Figure 14, after $G_{i,j}$ is appended to *path*, $remove_{i,j+1} = \{P\}$. Then A, B, C, and Q are goal cells and Q has an edge directed south. If Q has an edge directed south, then C must be marked. Therefore, A and B must be marked, by Observation 3(a). Since neither K nor Z are marked goal cells, C must have an edge directed east to marked cell D in column G_{i+4}. So there must be a path of marked goal cells from cell $G_{i,j+1}$ to cell D in column G_{i+4} after line 9 of *Find_Path* is executed. Thus, the lemma holds. ■

Consider the last execution of line 9 of *Find_Path* that removes a marked neighbor of c_j, either to the north (N), northeast (NE), southeast (SE), or south (S) of c_j (see Figure 15 (a)), say at time t. Recall that a cell is added to a *remove* set in procedure *IsAdmissible* only in Case 2, when vertical edges are directed. Without loss of generality, assume the removal was caused by the addition of a north segment (an analogous argument can be made for a south segment). Figure 15(b) shows a labeling on cells when a north segment O, H is added to *path* at time t, causing cell c_j to lose its last marked neighbor.

From *IsAdmissible* (Figure 9), we can see that only cell P can possibly be unmarked when segment O, H is added to *path*. In the following case analysis, all possible positions for cell c_j in relation to cell P are considered when procedure *IsAdmissible* is called with d = S. It is shown that, in each of these cases, either c_j will have a marked neighbor after P is unmarked at time t, or c_j cannot possibly have been included on *path* after time t, a contradiction.

Case 1: The last neighbor of goal cell c_j to be unmarked is N. Then c_j = B, which would still have marked neighbor C.

Case 2: The last neighbor of goal cell c_j to be unmarked is NE. Then c_j = A, which would still have marked neighbor B.

Cases 3 and 4: The last neighbor of goal cell c_j to be unmarked is SE or S. Then c_j = T or V. But neither A nor H can have edges directed to the north, by Lemma 2, and therefore there is no path from H to either T or V. Thus, $c_j \neq$ T and $c_j \neq$ V.

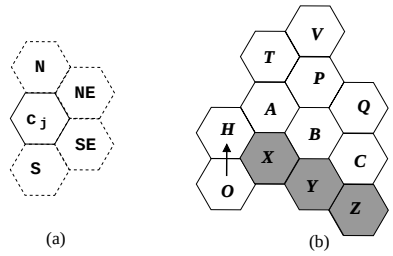

Fig. 15. Labels used in proof of Theorem 1 (in part (b), possible goal cells are unshaded).

Thus, in all cases, *Find_Path* will not get "stuck".

Therefore, if G contains an admissible substrate path p, thereby satisfying Definition 3, then *Find_Path* will return a path that ends in column G_m. ■

Theorem 2: If algorithm *Find_Path* returns a path ending in column G_m, then G is admissible.

Proof: Let the returned path be $p = c_1, c_2, \ldots, c_l$, where c_l is in G_m. Note that it is easy to show that p is a substrate path. The key thing remaining to be shown is that p is admissible, meaning that p satisfies the conditions in Definition 3. It is easy to show that p never goes west, by the way H is constructed. We will show, by induction, that for $1 \leq i \leq l$, prefix $p_i = c_1, \ldots, c_i$ of p satisfies conditions 2 and 3 for being an admissible path.

The basis, $i = 1$, is true because $p_1 = c_1 = G_{1,0}$, which can cause no violation of conditions 2 or 3 in Definition 3, since p_1 consists of a single cell.

For the inductive hypothesis, assume that $p_i = c_1, \ldots, c_i$ is admissible. Now, we will show that $p_{i+1} = c_1, \ldots, c_{i+1}$ is also admissible. References to cells A, B, C, P, Q, X, Y, and Z in this proof refer to the labels used in the *IsAdmissible* procedure (Figures 9 and 10). In the remainder of the figures used in this proof, goal cells have solid borders and unoccupied cells have dashed borders. The labels on unoccupied cells provide orientation in relation to the labels used in the *IsAdmissible* procedure.

We use a case analysis of the direction of two consecutive edges in p. Note that if each possible direction of a segment (N, S, NE, or SE) can be followed by each of four others, that there are 16 possible cases. However, the combinations of a N segment followed by a S segment and vice versa are not possible due to the assumption that p is composed of unique cells. A N segment cannot be followed immediately by a SE segment (and likewise a S segment cannot be followed immediately by a NE segment), nor can a NE segment be following immediately by a S segment (and likewise for a SE segment followed by a N segment), by Claim 1. This leaves 10 possible cases. We consider only 5 of these because the rest are vertical inversions of the cases shown.

Case 1: c_{i-1}, c_i, c_{i+1} form part of a N segment (see Figure 16(a)). There is no violation of condition 2(a) because if there are goal cells in X, Y, or Z, then the edge from c_i to c_{i+1} would not have been directed, by Case 1 of *IsAdmissible*, so this segment would not be added to *path*, a contradiction. Since the N segment c_{i-1}, c_i did not violate condition 2(b) of Definition 3 for any S segment ending to the west, no violation will be caused by the N segment formed by c_{i-1}, c_i, and c_{i+1}.

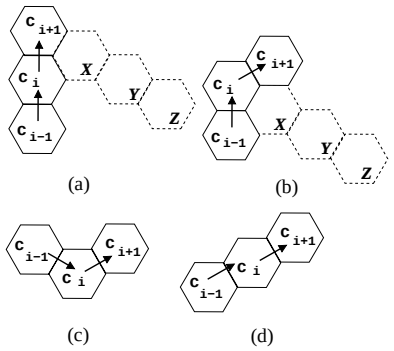

Fig. 16. Cases 1 through 4 in proof of Theorem 2.

Case 2: c_i is the end of a N segment and c_{i+1} forms part of a NE segment (see Figure 16(b)). Then the N segment c_{i-1}, c_i would not have violated case 1 of *IsAdmissible* because X, Y, and Z are not goal cells. Since the segment ending in c_{i+1} is not a S segment, p_{i+1} will satisfy all conditions of Definition 3.

Cases 3 and 4: c_i is the end of a SE segment and c_{i+1} forms part of a NE segment (see Figure 16(c)) or c_i is the end of a NE segment and c_{i+1} forms part of a NE segment (see Figure 16(d)). Since neither segment in either case is vertical, p_{i+1} would satisfy Definition 3.

Case 5: c_i is the end of a NE segment and c_{i+1} forms part of a N segment. Any violations of condition 2(a) of Definition 3 by N segment c_i, c_{i+1} occurring to the east of c_{i+1} are averted by the action of algorithm *Direct_Edges*. By Case 1 of *IsAdmissible*, if there are goal cells in positions X, Y, or Z (see Figure 17(a)), then the edge from c_i to c_{i+1} would not be directed, and therefore c_{i+1} could not be added to p, a contradiction. Condition 2(b) of Definition 3 could be violated at a later time, when a S segment is added to p to the east of c_{i+1}, but any violation will occur at the time this S segment is added to p, not when segment c_i, c_{i+1} is added.

In order for p_{i+1} to cause a violation of Definition 3 in relation to cells to the west, there must be a S segment in the segments formed by cells

(a) c_{i-4} and c_{i-3} or
(b) c_{i-3} and c_{i-2}.

Below, each of these segments is considered as a prefix to the segments formed by c_{i-1}, c_i, and c_i, c_{i+1}. Note that there are 3 cases because we need consider only S segments followed by a SE segment in the segments (a) and (b), by Claim 2. If either segment (a) or (b) were a N segment, then it would satisfy the conditions for an admissible path by the inductive hypothesis and would not cause the N segment formed by c_i and c_{i+1} to violate these conditions either.

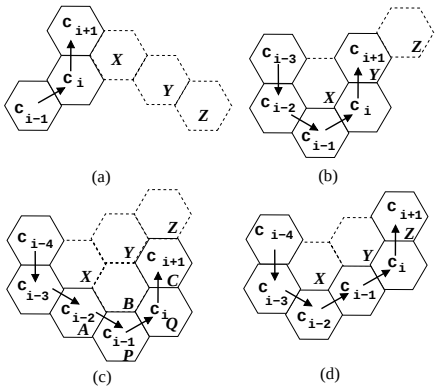

Fig. 17. Case 5 in proof of Theorem 2.

1. There is a SE segment ending in c_{i-1} and a S segment ending in c_{i-2} (see Figure 17(b)). By case 1 of *IsAdmissible* , segment c_{i-3}, c_{i-2} could not have been added to p_i because c_{i+1} is in cell Y.
2. There is a SE segment ending in c_{i-1}, a SE segment ending in c_{i-2}, and a S segment ending in c_{i-3} (see Figure 17(c)). Notice that there are goal cells in cells A and C. If there is no goal cell in B, then segment c_{i-4}, c_{i-3} could not have been added to p_i because it violates case 2 of *IsAdmissible*. If there is a goal cell in B, then cell c_{i-1} (P) would have been unmarked when segment c_{i-4}, c_{i-3} was added to p_i, by case 2 of *IsAdmissible*. Since cell P is marked by assumption, it must be that there is no goal cell in B, so segment c_{i-4}, c_{i-3} could not have been added to p_i.
3. There is a NE segment ending in c_{i-1}, a SE segment ending in c_{i-2}, and a S segment ending in c_{i-3} (see Figure 17(d)). By case 1 of *IsAdmissible*, segment c_{i-4}, c_{i-3} could not have been added to p_i because c_{i+1} is in cell Z.

These cases show that it is not possible for p_{i+1} to violate condition 2(b) of Definition 3 when c_{i+1} is added to p_i, since there can be no conflicting S segment to the west in p_i.

So p_{i+1} must satisfy all conditions of Definition 3. Since we assume *Find_Path* returns a path ending in G_m, the lemma implies that p is an admissible substrate path.

■

Theorems 1 and 2 imply that algorithm *Find_Path* will return only an admissible substrate path and will find an admissible substrate path if one exists in G. In other words, the algorithms presented in this section will correctly identify admissible configurations of G.

VI. DISTRIBUTED RECONFIGURATION ALGORITHM

In this section, we present the distributed reconfiguration algorithm that performs the reconfiguration of I to G after an admissible substrate path is found using the algorithms in the previous section.

A. Algorithm Assumptions

1. Each module knows the total number of modules in the system, n, and the goal configuration, G.
2. Initially, one module is in each cell of I.
3. I is a straight chain.
4. G is an admissible configuration.
5. I and G overlap in goal cell $G_{1,1}$, as described in Sect. V-A.

To simplify the presentation of our reconfiguration algorithm, we assume the coordinates of G are ordered at each module as follows:

- The coordinates of cells on the substrate path are stored in a list, in the order in which the cells occur on the directed path from G_1 to G_m, beginning with the cell on the substrate path which has a directed edge incoming from cell $G_{1,1}$.
- The coordinates of cells in G that are north of the substrate path are stored in a list starting with the cell adjacent to and north of the cell on the substrate path in G_m to $G_{m,1}$, followed by the cell adjacent to and north of the cell on the substrate path in G_{m-1} to $G_{m-1,1}$, and so on, ending with the northwesternmost cell north of the substrate path in G.
- The coordinates of cells in G that are south of the substrate path are stored starting with the cell adjacent to and south of the cell on the substrate path in G_m to $G_{m,j}$, where $j = |G_m|$, followed by the cell adjacent to and south of the cell on the substrate path in G_{m-1} to $G_{m-1,k}$, where $k = |G_{m-1}|$, and so on, ending in the southwesternmost cell south of the substrate path in G.

B. Overview of Algorithm

The algorithm works in synchronous rounds. In each round, each module calculates whether it is free (cf. Fig. 18). In this figure, the modules labeled *trapped* are unable to move due to hardware constraints and those labeled *free* represent modules that are allowed to move in our algorithm, possibly after some initial delay. The modules in the *other* category are restricted from moving by our algorithm, not by hardware constraints.

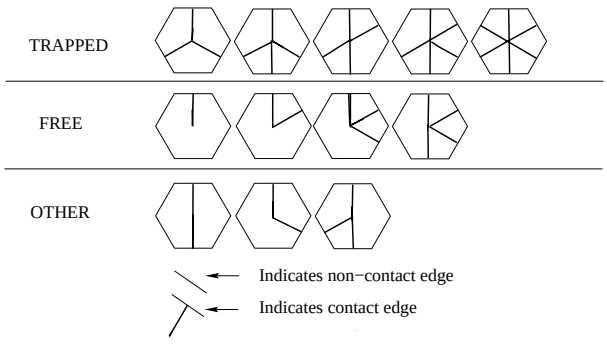

Fig. 18. Contact patterns possible in algorithm.

Modules in I initially calculate their position in I, direction of rotation, possible delay and final coordinates in G by determining their lattice distance from cell $G_{1,1}$. A module calculates the goal cell it will occupy by comparing its position in I to the length of the arrays of coordinates on, north, and south of the substrate path.

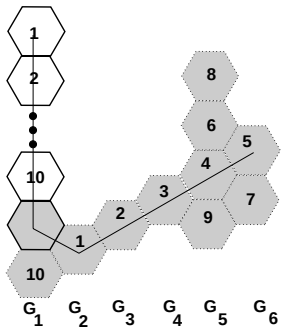

Fig. 19. Correspondence of initial module positions to final goal positions.

Let p be the substrate path, starting with the cell that has an edge incoming from cell $G_{1,1}$. Modules in positions $\leq |p|$ fill in the substrate path first. After p is filled, modules alternate rotation directions, filling the columns projecting north and south of p from east, G_m, to west, G_1. Figure 19 has numbered goal cells showing how initial module positions correspond to final goal positions.

As in our previous paper[11], modules use specific patterns of rotation and delay in our algorithm, as listed below. Note that only patterns 2 and 4 are used in the algorithm presented in this section.

1. *(0,0)-bidirectional*: modules alternate direction with no delay after free.
2. *(1,0)-bidirectional*: modules alternate direction with delay of 1 time unit after free for modules in positions > 1 rotating CW and no delay after free for modules rotating CCW.
3. *1-unidirectional*: modules rotate same direction with delay of 1 after free for modules in positions > 1.
4. *2-unidirectional*: modules rotate same direction with delay of 2 after free for modules in positions > 1.

The reconfiguration proceeds as follows:

- For modules in positions 1 through $|p|$:
  - Modules use *2-unidirectional* pattern in CW direction.
  - When a module is in the goal cell that it should occupy in p, it stops in that cell.
- For modules in positions $> |p|$:
  - Modules use *(1,0)-bidirectional* pattern until all cells on one side of p are filled. After this, modules use *2-unidirectional* pattern, with either CW or CCW direction, depending on whether there are cells remaining to be filled on the north or south side of p.
  - When a module is in the goal cell it should occupy, it stops.
- Once a module stops in the goal cell it should occupy for a round it never moves out of that goal cell.

C. Algorithm Pseudocode

The pseudocode for the distributed reconfiguration is shown in Figure 20. Figure 21 shows the pseudocode used by each module to initially determine d, *delay*, and *myGoalCoord*. Modules cannot initially determine the exact time when they will begin moving. They rely on local contact information to calculate when they are free to move. Once a module begins moving, it has only the local information about its contacts with adjacent modules and its current coordinates to guide its part of the entire system reconfiguration.

The algorithm uses the following local variables at each module:

- n: Number of cells in G and modules in I.
- *myCoord*: The coordinates of the module in the plane.
- *contacts*: Boolean array indicating which edges have neighboring modules. Assumed to be automatically updated at each round by some lower layer.
- *position*: Order of modules in I, starting at the northernmost end of I. Initially calculated as $n-LD$(*myCoord*, coordinates of $G_{1,1}$).
- d: Variable containing the direction of movement, CW or CCW.
- *diff*: Variable to hold difference between *position* and length of substrate path.
- *flips*: Counter used to determine whether the module is free.
- *delay*: Number of time units module waits after it is free and before it makes its first move. Initially set to 0.
- *myGoalCoord*: Coordinates of goal cell in which module will stop moving. Initially module in I overlapping G has *myGoalCoord* = coordinates of $G_{1,1}$ and all other modules in I calculate *myGoalCoord* after calculating their *position*.
- *substrateCoords*, *coordsN*, and *coordsS*: Arrays of coordinates of goal cells on, north, and south of the substrate path, in the order described in Sect. VI-A.

```
In round r := 1, 2, ... :
 1. if (IsFree())
 2.    if (delay = 0)
 3.       move d
 4.    end if
 5. else
 6.    delay–
 7. end if

Procedure IsFree():
 1. flips := 0
 2. for (i := 0 to 5) do
 3.    if (contacts[i] ≠ contacts[(i + 1) % 6])
 4.       flips++
 5.    end if
 6. end for
 7. return ((position - 1 is unoccupied) and
            (flips = 2) and (number of contact edges < 5))
```

Fig. 20. Pseudocode for reconfiguring modules from straight chain to admissible G.

Code for each module where *myCoord* ≠ *myGoalCoord*:

```
Initially:
 1. position := n − LD(myCoord, coordinates of G_{1,1})
 2. diff := position −|substrateCoords|
 3. if (diff ≤ 0)    // Module goes on substrate path
 4.    myGoalCoord := substrateCoords[position - 1]
 5.    d := CW   // Go east
 6.    delay := 2
 7. else if (diff is odd)
 8.    if ((diff+1)/2 ≤ |coordsS|) // Go west
 9.       d := CCW
10.       if ((diff−1)/2 > |coordsN|)
                                  // Module to north goes west
11.          myGoalCoord := coordsS[diff −|coordsN| − 1]
12.          delay := 2
13.       else myGoalCoord := coordsS[(diff −1)/2]
                                  // Module to north goes east
14.       end if
15.    else  // Go east
16.       myGoalCoord := coordsN[diff −|coordsS| − 1]
17.       d := CW
18.       delay := 2
19.    end if
20. else if (diff is even)
21.    if ((diff/2 ≤ |coordsN|) // Go east
22.       d := CW
23.       if (diff/2 > |coordsS|)
                                  // Module to north goes east
24.          myGoalCoord := coordsN[diff −|coordsS| − 1]
25.          delay := 2
26.       else
                                  // Module to north goes west
27.          myGoalCoord := coordsN[(diff/2) − 1]
28.          delay := 1
29.       end if
30.    else  // Go west
31.       myGoalCoord := coordsS[diff −|coordsN| − 1]
32.       d := CCW
33.       delay := 2
34.    end if
35. end if
```

Fig. 21. Pseudocode for determining *myGoalCoord*, *delay*, and d.

D. Analysis of Reconfiguration Algorithm

The distributed reconfiguration algorithm presented in Section VI-C starts with a straight chain to (not necessarily straight) chain reconfiguration, with the first $|p|$ modules filling in the substrate path, p. From the proof in Section V-C, it can be seen that the moving modules (separated by 2 spaces) will not become deadlocked (i.e., by moving into a position where they have a contact pattern that is not "free" (cf. Figure 18)) prior to reaching their calculated goal positions. Since the modules filling in p all move south on the east side of I, it is not possible for them to change order and collide. Once the modules in p are in place, other moving modules are separated by p, ruling out possible collisions.

VII. Simulation Results

Our simulation experiments were inspired by the work of Pamecha et al. [9], where configurations of similar shape but varying number of modules were used to evaluate their algorithm. Direct comparison of the complexity of the algorithms presented in this paper with the results obtained by the centralized reconfiguration algorithm of Pamecha et al. is not possible due to the fact that their simulations

involved the reconfiguration of arbitrary shapes of I to arbitrary shapes of G.

We developed an object-oriented discrete event simulator to test the reconfiguration algorithms. Initially, the goal coordinates are specified and each module in I performs the calculations presented in Figure 21, depending on its initial position. During each round, the simulator checks the local status of every module, and then moves all eligible modules in the same step, thereby accurately simulating a real distributed system.

A. *Effect of Heuristics in* Find_Path

We first experimented with running our algorithm on various shapes using different numbers of modules, testing the effect on performance of varying the heuristic choice in line 3 of the *Find_Path* (see Figure 11) algorithm. Performance is measured in terms of number of rounds and number of moves needed for the reconfiguration.

The shapes experimented on included: 1) wedges of similar orientation and variable size, 2) rectangles that lengthened on the E-W axis while remaining fixed on the N-S axis, and 3) diamonds of similar orientation and variable size. These shapes were chosen because they are simple and yet illustrative of how heuristics can affect the performance of the reconfiguration algorithm.

The first heuristic (SN for "select north") chose the NE edge whenever there was a choice of NE or SE edges, biasing the substrate path to "hug" the north side of G. The second heuristic (SS) used a "seesaw" pattern, selecting the edge in the opposite direction as the edge last selected when there was a choice. The third heuristic (GR) used a greedy strategy in which the edge to the NE or SE was selected based on whichever choice most evenly divided the next column to the east.

Figure 22 illustrates the paths found by the SN heuristic, the SS heuristic, and the GR heuristic for a wedge of 29 cells, a rectangle of 21 cells, and a diamond of 26 cells. Heuristic GR was able to more evenly split G into halves for each shape when n was sufficiently large.

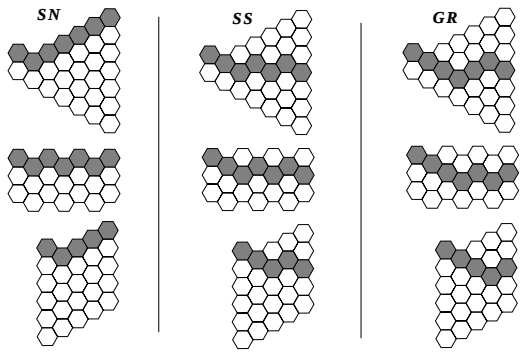

Fig. 22. Example paths found for SN, SS, and GR heuristics.

In Figs. 23 and 24(a), we depict the results obtained from experiments with wedges of similar orientation and increasing size. Figures 23 and 24(b) show the results obtained when experiments were performed on lengthening rectan-

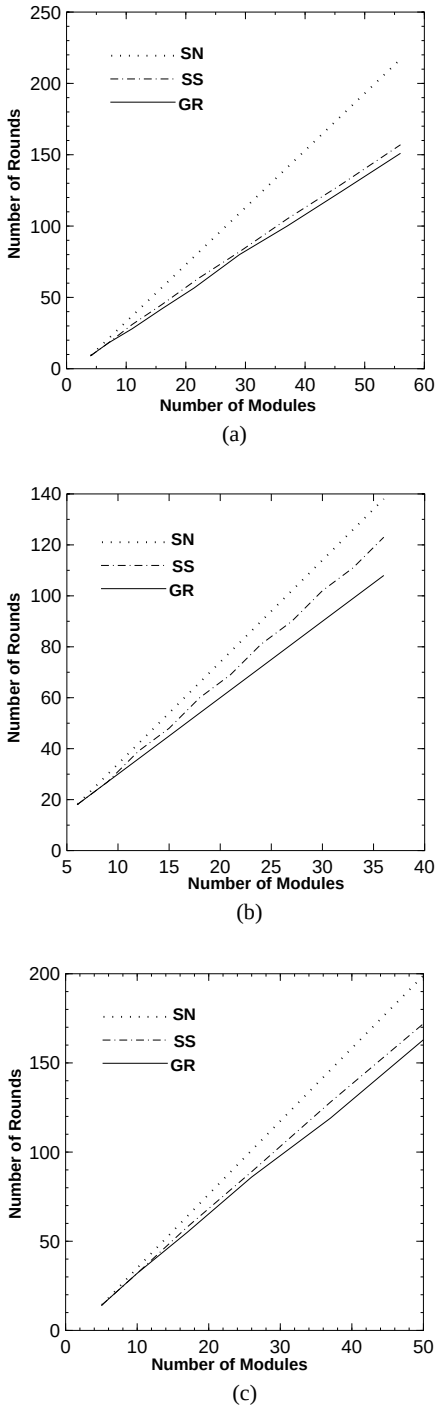

Fig. 23. Rounds used for wedge (a), lengthening rectangle (b), and diamond shaped (c) configurations. Heuristics used are "Select North" (SN), "See-Saw" (SS), and "Greedy" (GR).

gles and Figs. 23 and 24(c) show the results on diamond shapes. For each shape, the number of moves increased more than linearly for increasing values of n. Also, the number of moves was nearly the same for each heuristic for all values of n. For each shape, when $n > 9$, heuristic GR used fewer rounds than did the SN or SS heuristics. Performance, in terms of number of rounds used, improves when the substrate path evenly divides G because modules can alternate direction, allowing more modules to move in parallel.

Therefore, while any admissible directed path of marked nodes may be chosen as the substrate path, heuristics can improve the number of rounds, and, to a lesser extent, the number of moves, required for reconfiguration.

B. Simulation on Realistic Shapes

Metamorphic robotic systems need to assume different useful shapes. Possible useful shapes include bridges to span rivers or rough terrain and buttresses to support collapsing buildings or temporary constructions such as emergency flood levees (see Figure 25). In this section, we present the results of simulation experiments involving the reconfiguration of systems composed of varying numbers of modules into useful structures. Since the greedy heuristic had the best performance in terms of number of rounds when tested on simple shapes, we used the greedy heuristic when finding a substrate path in all experiments in this section.

Figure 26(a) shows three examples of the bridge shape used in our experiments, depicting the basic bridge pattern for 11, 18 and 25 modules. The shaded modules represent the substrate path chosen by our algorithm using the greedy heuristic. As shown in Table I, we continued the experiment by increasing the number of modules simulated to over 50. The extension of the bridges for higher number of modules follows the pattern depicted in Figure 26(a).

Figure 26(b) shows two examples of the tall buttress shape used in our experiments, depicting the basic pattern for 19 and 24 modules. The shaded modules represent the substrate path chosen by our algorithm using the greedy heuristic. As shown in Table I, we continued the experiment by increasing the number of modules simulated to about 50. The extension of the buttresses for higher number of modules follows the pattern depicted in Figure 26(b).

Table I shows that the number of rounds used in the reconfigurations from chains to more "realistic" shapes increases in a linear fashion as did the number of rounds for "simple" shapes in the last section. The increase in the number of moves as the number of modules increases is also similar to that shown for the "simple" shapes. Clearly, filling in each of these "realistic" shapes from the bottom up would be preferable if gravity were a concern in the reconfiguration. The substrate paths chosen do not attempt to follow this bottom-up pattern. The shapes considered in this section also do not lend themselves to effective use of the heuristics in *Find_path*, resulting in low overall parallelism in these simulations. However, these shapes do represent "real world" applications for metamorphic robots and

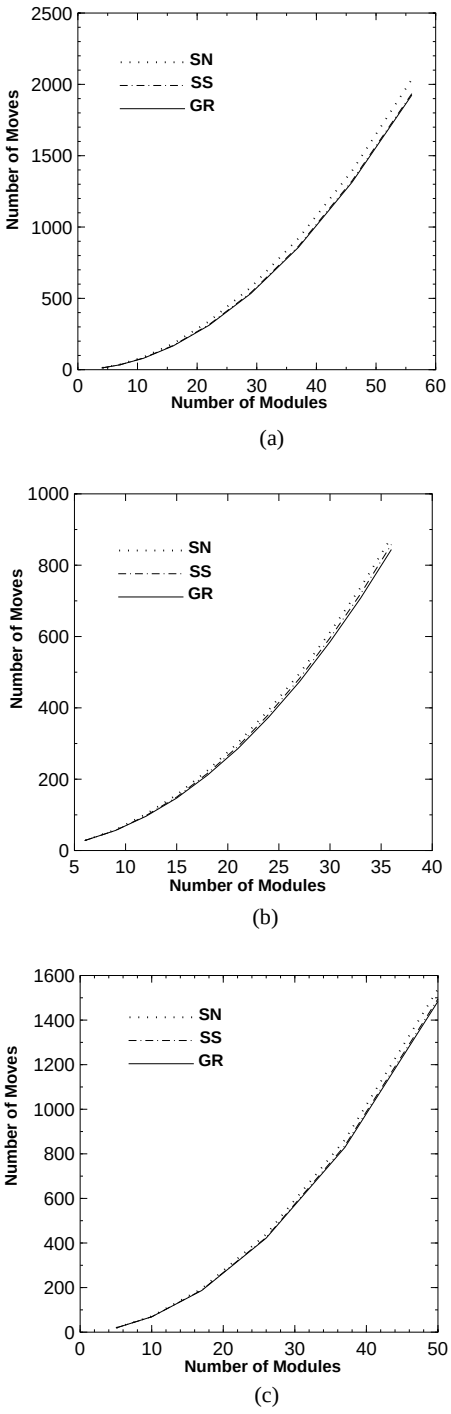

Fig. 24. Moves used for wedge (a), lengthening rectangle (b), and diamond shaped (c) configurations. Heuristics used are "Select North" (SN), "See-Saw" (SS), and "Greedy" (GR).

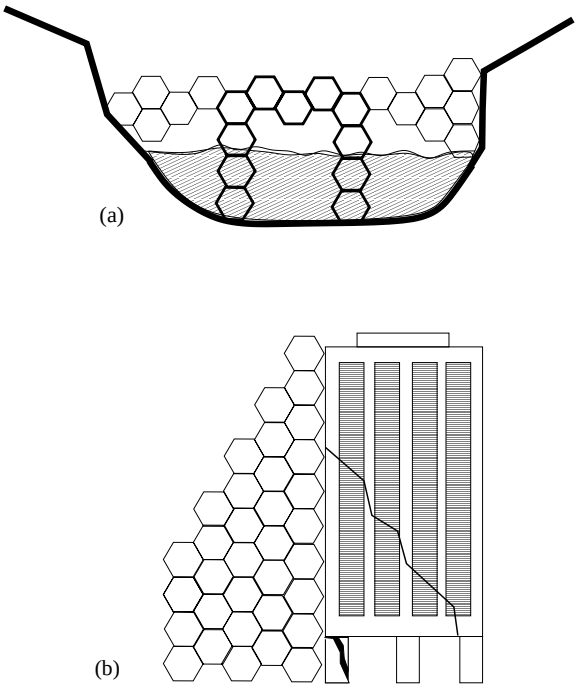

Fig. 25. Metamorphic system as (a) bridge and (b) support for building.

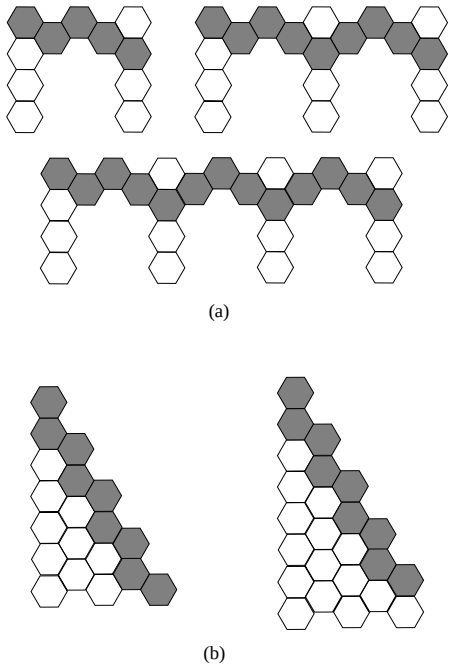

Fig. 26. Example bridge shapes (a) and tall buttress shapes (b).

TABLE I
NUMBER OF ROUNDS AND MOVES USED FOR BRIDGE AND BUTTRESS.

Shape	Modules	Rounds	Moves
Bridge	11	36	89
	18	61	239
	25	86	466
	32	111	770
	39	136	1151
	46	161	1609
	53	186	2145
Tall Buttress	19	74	269
	24	95	418
	29	116	597
	34	137	806
	44	178	1311

we have demonstrated that our algorithm can effectively perform the reconfigurations.

In [11], we showed that our distributed algorithm for straight chain to chain reconfiguration in a system of hexagonal robots, given our system assumptions, takes $O(n)$ rounds and $O(n^2)$ moves. The experimental results we present in this section suggest that our straight chain to admissible goal reconfiguration algorithms have similar complexity.

VIII. Conclusions and Future Work

The algorithms presented in this paper rely on total knowledge of the goal configuration. Each module precomputes all aspects of its movement once it has sufficient local information to reconstruct the entire initial configuration. We proved the correctness of our centralized algorithms for finding a substrate path and tested the performance of our distributed reconfiguration algorithm through simulation.

Since we restrict the initial configuration to a straight chain, it is rather simple for the modules to reconstruct the entire initial configuration. We believe that a more flexible approach will be helpful in designing reconfiguration algorithms for more irregular configurations, more asynchronous systems, and those with unknown obstacles. Part of such a flexible approach will include the ability for modules to detect and resolve collisions and deadlock situations when they occur, rather than precomputing trajectories that avoid these situations. We have some initial ideas for ways to deal with module collision and deadlock on the fly, which we leave for future work.

The orientation of the initial chain to the admissible goal shape limits the possible choices for the point of contact between the initial and goal configurations. In the future, we plan to develop algothms that do not place such a strict orientation criteria on the initial positions of the chain and the admissible goal configuration. Future work will also include simulation using different heuristics to improve the time used for reconfiguration. For example, if the substrate path is a straight chain to the SE or NE, it can be filled using a pattern in which modules alternate direction, as was done in our straight chain to straight chain algorithms[11]. Another heuristic improvement is to choose a substrate path for which all modules on the north or south

of the path meet the path at an obtuse angle, since then the distance between moving modules can be reduced to one space, increasing the overall parallelism achieved by the reconfiguration.

References

[1] A. Casal and M. Yim. Self-reconfiguration planning for a class of modular robots. In *Proc. of SPIE Symposium on Intelligent Systems and Advanced Manufacturing*, vol. 3839, pages 246–256, 1999.

[2] G. Chirikjian. Kinematics of a metamorphic robotic system. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 449–455, 1994.

[3] G. Chirikjian and A. Pamecha. Bounds for self-reconfiguration of metamorphic robots. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 1452–1457, 1996.

[4] K. Kotay and D. Rus. Motion synthesis for the self-reconfiguring molecule. In *IEEE Intl. Conf. on Robotics and Automation*, pages 843–851, 1998.

[5] K. Kotay, D. Rus, M. Vona, and C. McGray. The self-reconfiguring robotic molecule: design and control algorithms. In *Workshop on Algorithmic Foundations of Robotics*, pages 376–386, 1998.

[6] S. Murata, H. Kurokawa, and S. Kokaji. Self-assembling machine. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 441–448, 1994.

[7] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji. A 3-D self-reconfigurable structure. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 432–439, 1998.

[8] A. Nguyen, L. J. Guibas, and M. Yim. Controlled module density helps reconfiguration planning. To appear in *Proc. of 4th International Workshop on Algorithmic Foundations of Robotics*, 2000.

[9] A. Pamecha, I. Ebert-Uphoff, and G. Chirikjian. Useful metrics for modular robot motion planning. *IEEE Transactions on Robotics and Automation*, 13(4):531–545, 1997.

[10] D. Rus and M. Vona. Self-reconfiguration planning with compressible unit modules. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 2513–2520, 1999.

[11] J. Walter, J. Welch, and N. Amato. Distributed reconfiguration of metamorphic robot chains. In *Proc. of ACM Symp. on Principles of Distributed Computing*, pages 171–180, 2000.

[12] M. Yim. A reconfigurable modular robot with many modes of locomotion. In *Proc. of Intl. Conf. on Advanced Mechatronics*, pages 283–288, 1993.

[13] M. Yim, J. Lamping, E. Mao, and J. G. Chase. Rhombic dodecahedron shape for self-assembling robots. SPL TechReport P9710777, Xerox PARC, 1997.

[14] Y. Zhang, M. Yim, J. Lamping, and E. Mao. Distributed control for 3D shape metamorphosis. To appear in *Autonomous Robots Journal, special issue on self-reconfigurable robots*, 2000.