9-1-2004

# Heterogeneous Self-Reconfiguring Robotics

Robert Charles Fitch
*Dartmouth College*

# Heterogeneous Self-Reconfiguring Robotics

**Dartmouth Technical Report TR2004-519**

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Robert Charles Fitch

DARTMOUTH COLLEGE

Hanover, New Hampshire

September, 2004

Examining Committee:

(chair) Daniela Rus

Chris Bailey-Kellogg

Zack Butler

Bruce Donald

Alfred Rizzi

Charles K. Barlowe
Dean of Graduate Studies

# Abstract

Self-reconfiguring (SR) robots are modular systems that can autonomously change shape, or *reconfigure*, for increased versatility and adaptability in unknown environments. In this thesis, we investigate planning and control for systems of non-identical modules, known as *heterogeneous* SR robots. Although previous approaches rely on module homogeneity as a critical property, we show that the planning complexity of fundamental algorithmic problems in the heterogeneous case is equivalent to that of systems with identical modules. Primarily, we study the problem of how to plan shape changes while considering the placement of specific modules within the structure. We characterize this key challenge in terms of the amount of free space available to the robot and develop a series of decentralized reconfiguration planning algorithms that assume progressively more severe free space constraints and support reconfiguration among obstacles. In addition, we compose our basic planning techniques in different ways to address problems in the related task domains of positioning modules according to function, locomotion among obstacles, self-repair, and recognizing the achievement of distributed goal-states. We also describe the design of a novel simulation environment, implementation results using this simulator, and experimental results in hardware using a planar SR system called the Crystal Robot. These results encourage development of heterogeneous systems. Our algorithms enhance the versatility and adaptability of SR robots by enabling them to use functionally specialized components to match capability, in addition to shape, to the task at hand.

# Acknowledgements

I would first like to thank my advisor, Daniela Rus, for all her ideas, support, and encouragement here at Dartmouth. Thanks also to my committee for all their time and energy!

The members of the Robot Lab and Robotics Journal Club have been endless sources of both technical assistance and fun. Double thanks go to Zack Butler for collaboration and help with recalcitrant robots, algorithms, and code, in addition to service on the committee – and for just being an all-around great guy to work with. Marty Vona built the first Crystal robot and simulator, and developed the MeltGrow algorithm on which MeltSortGrow is based. Yuhang Wang assembled the second Crystal prototype and wrote initial low-level code. Paul Hansen helped put together our favorite Mars-terrain simulations.

Thanks, as well, to everyone in the CS Dept. who has helped me brain-storm ideas and watched me scrawl diagrams on any available writing surface. Ramgopal Mettu has provided invaluable algorithmic discussions, especially in working out the details of the CTS swap-sequence algorithm. Dave Wagner and I spent many late nights with MBP planning. Chris Langmead has always been there to listen to my crazy idea *di giorno*.

Finally, many thanks go to my family - Bob, Pat, Eric, and Bradley - and all the wonderful friends I've been fortunate enough to share time and experiences with during my Hanover years. You know who you are and know I wouldn't be here without you!

The work in this thesis was performed in the Dartmouth Robotics Lab at Dartmouth College. Funding was provided by NASA and the NSF. I am grateful for their assistance.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

*Self-reconfiguring* (SR) robots are robots that can change shape to match the task at hand. These robots comprise many discrete modules with simple functionality such as connecting to neighbors, limited actuation, computation, communication and power. Orchestrating the behavior of the individual modules allows the robot to approximate, and *reconfigure* between, arbitrary shapes. This shape-changing ability enables SR robots to respond to complex situations better than fixed architecture robots. Common examples of reconfigurability include transforming between snake shapes for moving through holes and legged locomotion for traversing rough terrain. Alternatively, an SR system can employ waterfall-like locomotion as illustrated in Figure 1.1. Here, we see a robot with cube-shaped modules divide into smaller groups to explore its environment in parallel. These examples illustrate the main promise of shape-changing machines: extreme versatility and adaptability in unknown environments.

The modules that compose SR systems can be identical, as they are in *homogeneous* systems, or non-identical, as in *heterogeneous* systems. In heterogeneous systems, modules are grouped into various classes, where modules are the same within a given class. Differences between classes are generally based on function, such as sensor payload,

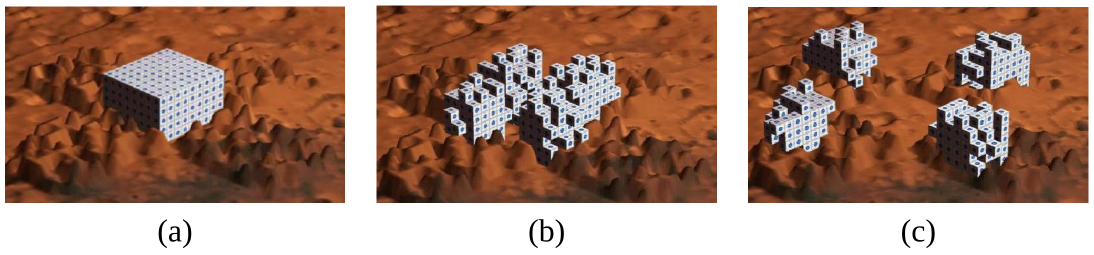


Figure 1.1: Simulation of SR robot on Mars terrain. The robot begins as a single cube in (a), but divides into four in (b) and (c) for parallel exploration using a distributed, waterfall-like locomotion algorithm.

computational power, energy storage, etc. Class differences can also encompass other variations such as module shape, actuation, and connection mechanism. Systems that are heterogeneous retain the shape-shifting advantages of homogeneous systems, but also offer increased capabilities due to functional specialization. The benefit is that heterogeneous SR robots can match not only *structure*, but also *capability* to task.

Many interesting planning and control problems arise in the study of SR robots. The central problem, *reconfiguration planning*, is how to compute a global shape change composed of local module movements. For heterogeneous systems, the key challenge is that the placement of individual modules within the structure must be considered. In this thesis, we concern ourselves with the development, analysis, and implementation of planning and control algorithms for heterogeneous SR robots, focusing on a class known as *lattice-based* systems. In a lattice-based system, module location is confined to discrete positions in space, as if the modules were embedded in a lattice[1].

[1]We do not consider the other main class, *chain-based* systems, where modules contain continuous degrees of freedom and aggregate in chains.

## 1.1 Heterogeneous Systems

As most SR robots research to date focuses on homogeneous systems[2], our reasons for studying the implications of heterogeneity warrants discussion. In research on homogenous systems, contributions are numerous and many algorithmic and hardware challenges have been met. But, the natural limitations of homogeneous systems lead us to investigate heterogeneity. For example, it would be difficult to argue for building a robot with 1000 radio transmitters, 1000 deployable wheels, and 1000 infrared cameras, as would be required by a homogeneous system, when significantly fewer resources are sufficient. One potential solution is to construct complex modules from very small homogeneous parts, but then the granularity of module size must be extremely high versus the size of the overall robot.

Castano and Will [16] compare homogeneous and heterogeneous designs in terms of a trade-off between software complexity and hardware complexity. They observe that a common module design benefits from economies of scale in design and manufacturing, and from simpler software requirements. Choosing multiple module types increases the cost of both hardware and software production. However, as desired capabilities increase, the complexity and cost of the base module also increases. The trade-off is that as desired functionality increases, homogeneous systems give up simple hardware design for less software complexity, whereas heterogeneous systems maintain low hardware complexity through specialized modules but require complex software to control them.

The belief that planning is more difficult in heterogeneous systems is one factor that has led researchers to focus exclusively on homogeneous systems thus far, even though the complexity of heterogeneous variants of common problems was unknown until this thesis. This was an obvious gap because even the addition of trivial heterogeneity from a

[2]It is interesting to note, though, that the original vision of SR robots, outlined by Fukuda in 1987 [38] describes a heterogeneous system with dedicated wheel and joint modules, among others.

hardware perspective incapacitates homogeneous reconfiguration algorithms, which cannot distinguish a specialized module from any other. It is impossible to, for example, maintain a particular sensor module on the surface of the robot after reconfiguration. If any module differences, however small, are desired, a heterogeneous reconfiguration algorithm is required.

The results we report show that the complexity of many planning problems is, in fact, asymptotically equivalent in homogeneous and heterogeneous versions. Our hope is to encourage other researchers to explore the benefits of heterogeneous systems. To begin, a fundamental issue is the degree to which modules are different from each other. There are many possible dimensions of heterogeneity, such as size and shape differences, various sensor payloads, or different actuation capabilities. We illustrate these differences and motivate the specific problems we study via the following example applications.

## 1.2 Example Applications

Our vision for the potential of heterogenous systems is illustrated with the following representative motivating applications. Consider a future scenario in which exploration tasks are carried out by heterogeneous SR robots. When necessary, the robot reconfigures into a legged walker to move across rough terrain or rubble, or transforms into a snake shape for moving through small holes. The robot can also take advantage of smooth terrain by deploying a special module type containing wheels for fast, efficient locomotion. A variety of sensors are onboard, contained as modules within the structure of the robot. The sensor modules are surrounded by other modules for protection during locomotion, but reconfigure to the surface when needed for better performance. Power is provided by dedicated battery modules also stored in the structure of the robot. Since they are relatively heavy, these non-actuated battery modules remain close to the base of the robot during reconfigurations, but are maneuvered closer to modules that draw a large amount of current. Long range communication with human users is accomplished through a small number of radio modules. Communications algorithms work for any number of radios to provide fault tolerance against single radio failure. This property, that performance degrades gracefully in response to module failure, is true of most of the robot's resources. For example, if enough wheel modules fail then the robot locomotes using legs. If further failure occurs the robot employs an inchworm gait requiring very few operational actuators.

Another example application is self-assembly. Large SR modules of varying sizes and shapes are deployed in small groups to a remote construction site, where they rapidly self-assemble into scaffolding, buildings or temporary structures. This construction technique is utilized at Martian outposts or other sites that are difficult to reach. If a building is no longer needed, it simply reconfigures into a different structure. Alternatively, SR robots can form a kind of reconfigurable factory. An assembly task is divided into a number of subassembly tasks, which are completed by an SR robot configured specifically for that subassembly. There are specialized components for peg-in-hole type assemblies, gluing, welding or other tasks. The SR robot is simple and task-specific for the given subassembly, yet can accommodate a variety of different subassemblies through reconfiguration. Or, heterogeneous systems can be used in an underground mining application. Special drilling modules remove material, while structural modules form supporting structures on-the-fly.

## 1.3 Research Issues

The motivating examples outlined above suggest suggest a number of important research problems, both in hardware design and in planning and control. We do not address the issue of designing hardware capable of self-reconfiguration, but this topic has received

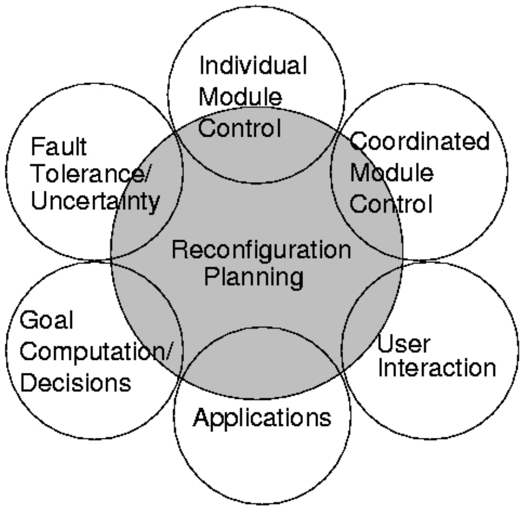

Individual Module Control
Fault Tolerance/ Uncertainty
Coordinated Module Control
Reconfiguration Planning
Goal Computation/ Decisions
User Interaction
Applications

Figure 1.2: Selected research issues in heterogeneous SR robotics. Circles represent general problem areas. Reconfiguration planning includes many problem variations, such as in-place and out-of-place reconfiguration, and reconfiguration among obstacles. This problem involves coordinating the control of individual modules to form the goal shape. User interaction includes the important human-robot interaction problem. Goal computation includes problems of configuration determination, determining the best configuration for a given situation, and also goal recognition, recognizing when the robot has reached the goal state. Fault tolerance includes issues of uncertainty in module-level actuations, as well as total module failure. Problem definitions, along with interactions between reconfiguration and peripheral problems, are discussed in Section 1.3.

extensive attention and we survey results in Chapter 2. Instead, we are interested in algorithmic planning and control problems. A set of these problems is illustrated in Figure 1.2. This set is not intended to be a complete list, but is provided to frame the problems studied in this thesis within a representative context of other important issues in this area. For example, *human-robot interaction* (HRI) is a necessary precursor to deploying SR robots in the field, but we leave this problem to other researchers. This is the also the case for the *configuration determination* problem, which asks how to compute the goal configuration best suited to a given situation. We detail these issues along with other ideas for future work later in Chapter 7.

What we do study in this thesis is the reconfiguration planning problem and the three

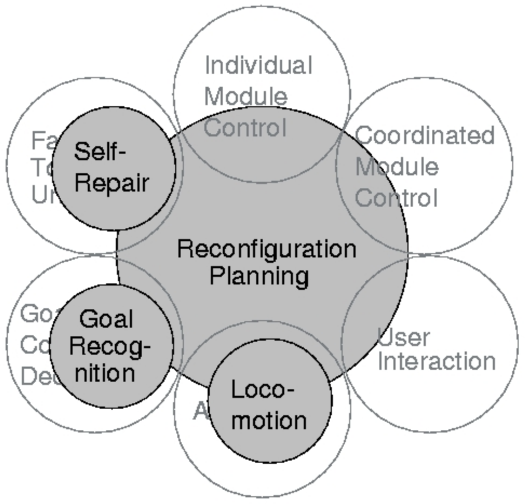


Figure 1.3: Research problems considered in this thesis, within the context of overall research issues. Shaded circles indicate problems studied; placement within unshaded circles indicates relationship to the larger problem area. Reconfiguration algorithms presented in the thesis include MeltSortGrow, TunnelSort, ConstrainedTunnelSort, and PositionConstraints. Locomotion solutions include Inchworm Locomotion, and locomotion through reconfiguration. A minimum-bends planner, 3D-MBP, is used in both locomotion and self-repair algorithms. Finally, our distributed goal recognition algorithm addresses the problem of recognizing achievement of goal state.

core related problems of *locomotion*, *fault-tolerance*, and *global goal-state recognition*. These are placed in context in Figure 1.3. Because of the inherently distributed nature of SR robots, our goal is to develop decentralized solutions to these problems that avoid the use of centralized controllers. We briefly characterize this set of problems here, and then give full discussion in the remainder of this section.

Reconfiguration is central because it underlies one of the fundamental claims of SR systems, the ability to match structure to task. We develop a sequence of three heterogenous reconfiguration algorithms that operate in environments with progressively more complex constraints on the free space available, and one algorithm that enables constraints on the position of specified module types given by relative location instead of by specific coordinates. Reconfiguration among obstacles is clearly an important challenge, and our free space constraints model this case. These are the first algorithms for reconfiguration in heterogeneous systems. We characterize this problem more fully and compare it to related problems later in this section (Section 1.3.2).

Less obviously, lower-level operations required by reconfiguration planning are inherently related to other core problems. One of these is the locomotion problem. In order to explore the unknown environments in our examples above, SR robots require mobility. They cannot employ methods utilized by other mobile robots, such as wheeled or legged locomotion, because SR systems do not necessarily have access to the required hardware mechanisms. In addition, lattice-based systems cannot perform the undulating gaits possible with the continuous degrees-of-freedom of chain-based robots. Therefore, the most natural approach is to use reconfiguration to construct novel locomotion modalities. We propose a locomotion algorithm called *Inchworm Locomotion*, and a supporting path planning algorithm called *3D-MBP*. These are introduced in Section 1.3.3.

The second of the reconfiguration-related problems we address is how to deal with uncertainty stemming from module failures. More specifically, the issue is how to use the inherent redundancy of SR systems for improved fault tolerance and reliability. Throughout our algorithmic work, we assume that uncertainty in low-level module actuation will be handled by low-level control in the modules themselves. Total failure of these controllers will be passed up to a higher layer, which is the level of abstraction at which our algorithms operate. Dealing with these failures, called *self-repair*, can be viewed as a type of heterogeneous reconfiguration. We treat failed modules as a special class of modules that are unactuated, and apply reconfiguration techniques to repair the robot by replacing failed modules with spares stored within the robot structure (see Section 1.3.3).

The final problem we study is how to compute *global* state information based on the exclusively *local* information available directly to individual modules. This is called global state estimation. The problem arises due to the dynamic connection topology of modules during reconfiguration. As the robot reconfigures to move around its environment, it may need to recognize the achievement of some goal state, such as a particular shape. We refer to this problem as *goal recognition*, and introduce it more fully in Section 1.3.3.

We now define this set of problems in detail, compare them to similar problems, and summarize our algorithmic results. But first, to begin the algorithmic presentation, it is necessary to introduce the module abstraction we assume: the Sliding-Cube.

### 1.3.1 Sliding-Cube Model

Algorithms for SR systems necessarily make assumptions about the SR systems on which they operate. These include such properties as actuation type, primitive motions and connection mechanisms. Together, these assumptions form a model of the underlying system. Since many types of SR systems have been conceived and built, the algorithm designer must make choices as to what model to support. We would like to design algorithms not for one specific robot, but for a class of module types. In other words, we would like our algorithms to be as general as possible and to be useful in a maximum number of systems. General models have been proposed for lattice-based systems, but since previous algorithms are exclusively homogeneous, a new model is necessary.

A model used previously by our lab can be instantiated by various lattice-based systems [7]. We will extend this model and define the *Sliding-Cube*. The module is a cube with connectors on all faces. Modules connect face-to-face to any other module, and have two motion primitives: sliding across another modules, and making a convex transition. These primitives allow a single module to move across the surface of a robot shape, or through the robot's volume using tunneling techniques. All modules are of the same size, but to support simple heterogeneity we assign each module a type, or *class ID*.

Although the level of heterogeneity in the Sliding Cube only consists of unique IDs, it is possible to extend the model to encompass a greater degree of heterogeneity. Each module can be given connectivity constraints based on type (e.g., class $x$ can only connect to class $y$), or more complicated motion primitives can be defined. We limit the heterogeneity of the model to unique IDs throughout the thesis, however.

Such an abstraction is useful because resulting algorithms can then be executed on any system that can instantiate the model. Many module type can do this. Most often the instantiation is accomplished through the use of *meta-modules*, or groups of atomic modules that act together as a unit. Examples are the Crystal, the Telecube, and the Molecule (see Chapter 2). A few module designs do implement the model directly, as well [19, 45, 53].

### 1.3.2 Reconfiguration Problem

After specifying the Sliding-Cube model, we can now define algorithmic problems more clearly. First, we discuss reconfiguration planning. An example reconfiguration is shown in Figure 1.4. There are many variations of this problem, but most frequently we assume fixed start and goal configurations sharing a common reference frame. In other words, the problem is to compute a feasible plan that, when executed from an initial configuration

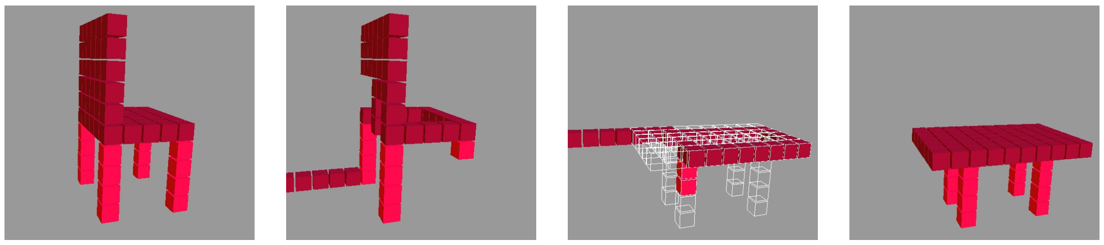


Figure 1.4: Simulation of chair-to-table reconfiguration with two module types. Light modules forming legs of the chair map to legs of the table. Dark modules in the chair form the top of the table. White outlines indicate the goal shape. This reconfiguration was planned with the MeltSortGrow algorithm.

$C$, results in a specified goal configuration $C'$. A plan is *feasible* if it maintains the *connectivity* property and consists of valid primitive motions for the module actuation model specified. Connectivity requires that no group of modules be separated from any others. This is defined precisely in Chapter 3. A configuration, assuming lattice-based systems, is represented as a list of modules defined by type and relative position. Position is defined by coordinates within a common reference frame. It is convenient to think of lattice-based systems graph theoretically; a configuration can also be thought of as an adjacency-list representation. Often, we assume that modules in adjacent lattice positions are connected, reducing the graph representation to a list of vertices.

Another important property of reconfiguration problems is the total amount of space occupied by intermediate configurations during shape-changing. We speak of this as the *free space* required by the algorithm. If free space is unlimited, the algorithm is *out-of-place*. *In-place* algorithms, alternately, are subject to a constrained amount of free space proportional to the union of the start and goal configurations. Finally, free space can also be defined as an arbitrarily-shaped bounding region surrounding the start and goal configurations. This variation is useful in performing reconfiguration among obstacles, or in preventing intermediate configurations from colliding with the surface on which the robot sits.

There is a continuum of approaches for solving the reconfiguration problem, ranging from fully centralized solutions where a plan is produced based on complete knowledge of global state, to purely distributed reactive solutions where each module has access to local information exclusively. Hybrid approaches mix elements of both extremes, using communication to selectively transmit partial state. An example of a local approach is the *cellular automata* (CA) model, where modules select motions by comparing local neighborhood to preconditions defined in a fixed set of rules. Centralized approaches have the advantage of easily provable properties; distributed approaches appeal to the notion of self-organizing systems [93]. Centralized planners require the use of a single centralized processor, while decentralized planners execute on systems with one processor per module.

This thesis focuses primarily on planning-based approaches. We report three primary heterogeneous reconfiguration planning results and one preliminary result for a problem variation for heterogeneous reconfiguration by function. Algorithms MeltSortGrow (MSG), TunnelSort (TS), and ConstraintedTunnelSort (CTS) form a sequence reducing free space requirements from out-of-place in MSG, to in-place in TS, to reconfiguration within a defined bounding region in CTS. The CTS algorithm supports reconfiguration among obstacles by representing obstacles as constraints on free space. All three assume Sliding-Cube modules and fixed start and goal configuration with or without holes. We provide both centralized and decentralized versions for each.

MSG is based on a homogeneous planner for unit-compressible modules called Melt-Grow (MG) [83]. Both algorithms reconfigure the start shape into a regular intermediate structure and then form the goal configuration. MSG supports heterogeneity by modifying this intermediate structure so that module types are correctly placed. This strategy forces all modules to move, and so is appropriate for large changes in shape. The free space requirements, though, are large. MSG is the first heterogeneous reconfiguration planner, and represents a clear example of how heterogeneous and homogeneous reconfiguration can both be solved in asymptotically optimal worst-case time, $\Theta(n^2)$, where $n$ is the number of modules in the system. We also implemented a portion of MSG in hardware.

TS is in-place and uses a different approach. Whereas module trajectories in MSG traverse the surface of the structure only, in TS we create trajectories that move through the volume of the structure. This technique is called *tunneling*. We show that short, straight tunnels are sufficient for solving the problem in-place. CTS uses more complex tunnel paths but allows for more complex free space constraints. The worst-case running time of TS is optimal ($\Theta(n^2)$), but CTS has the slower worst-case bound of $O(n^4)$ time and moves. We discuss later how this bound is much improved in practice.

The fourth algorithm is for a heterogeneous reconfiguration variation called *position constraints* (PC). PC abandons the assumption of exact goal configuration specification. This enables applications such as maintaining a sensor module at the top or front of the robot during locomotion. The goal configuration, in this formulation, constrains the location of a particular module type to a position in a goal configuration represented as a set of cubes or other orthohedral objects of varying sizes. This problem is an advance over standard heterogeneous reconfiguration because it decouples reconfiguration by shape from reconfiguration by function.

The work most related to ours is MG, as mentioned, and also distributed planners by Butler and Rus [11], and by Vassilvitskii, Yim, and Suh [99]. These algorithms are for homogeneous reconfiguration and are specified for unit-compressible modules. MG is centralized; the others are decentralized. In unit-compressible systems, modules are moved using *virtual module relocation*. Due to homogeneity, one unit can be virtually relocated to another position by shifting modules along a path between the start and goal locations. This generally requires the use of meta-modules, which simplifies the difficulty of maintaining connectivity. In contrast, our results support heterogeneous reconfiguration and are specified for the surface-moving Sliding-Cube modules. Virtual module relocation is not generally possible in heterogeneous systems, since the configuration specifies the placement of individual modules by type. Although MG and MSG share the use of an intermediate configuration, the module trajectory planning is completely different. Also, MSG encompasses both centralized and decentralized versions.

TS and CTS develop a powerful, novel method for motion the volume of the structure in inherently surface-moving systems. Maintaining connectivity is a significant challenge here since module relocation involves the removal of all intervening modules long the intended path.

## Complexity Analysis and Relationship to Coordinated Motion Planning

The lower bound for homogeneous reconfiguration is $\Omega(n^2)$, counting either moves or time steps, where $n$ is the number of modules in the system. This bound also applies to the heterogeneous case. The lower bound construction is explained in Chapter 3. Algorithms exist for homogeneous reconfiguration with an upper bound of $O(n^2)$ time and moves. Surprisingly, we show the upper bound of the heterogeneous case to be equivalent.

This result is interesting because reconfiguration belongs to the general family of *coordinated motion planning* problems. This family of problems involves the motion of multiple objects within a bounded region, avoiding collisions. These problems are often intractable, but also have special cases solvable in polynomial-time. One example is the *Warehouse Problem*, which is $PSPACE-$hard in the general case [100, 44]. The Warehouse Problem involves motion planning for multiple objects among obstacles and is intractable even with the restriction of 2D rectangular objects within a rectangular bounding region. The intractability proof relies on non-square objects. There are no connectivity constraints in this problem; objects can move freely in the available space. Furthermore, the Warehouse Problem has been shown to admit polynomial-time solutions given sufficient free space [87]. Our reconfiguration algorithms must solve a problem similar to the Warehouse Problem, with the added difficulty of maintaining connectivity. In Warehouse Problem terms, our solutions are tractable because they exploit the special cases of both free space and unit-square module shape.

The unit-square module assumption, however, is not always sufficient. In another related problem, the $n^2-1$-puzzle, a solution is not guaranteed to exist [43]. In this problem, also known as the Sliding Block Puzzle, labeled square tiles must be repositioned within a tight square region with limited free space such that tile labels form a sequence. Polynomial-time solutions exist, but finding optimal solutions is $PSPACE-$complete. Here, free space is severely limited and some instances have no solution. Our conclusion is that free space is the most important factor in the solvability of heterogeneous reconfiguration problems. Our results support this claim in that the complexity increases as available free space decreases.

The reconfiguration problem also has strong connections to the well-studied problem of sorting. Heterogeneous reconfiguration can be decomposed into two interrelated tasks: 1) forming overall shape, and 2) positioning modules correctly according to type. These tasks interact since operations that correct module positioning discrepancies most often disturb the global shape. However, it is possible to consider these as distinct since shape errors can be corrected ignoring type, assuming same-sized modules. Correcting type errors, then, involves relocating modules among a predetermined set of positions. This can be thought of as a variation of the sorting problem. The connection is most clear in the case of same-size modules with unique types; each module has exactly one valid destination. This destination is known *a priori* and is not determined by comparisons of module types. The $\Omega(n \log n)$ lower bound on comparison-based sorts, therefore, does not apply. Instead, this problem falls into the category of linear-time sorting. It is not generally possible to correctly position a single module in constant time due to feasibility constraints, however.

The above properties are summarized in Table 1.3.2. The table lists asymptotic complexity and common properties of reconfiguration variations, in addition to other coordinated motion planning problems.

| **Problem** | **Connectivity Constraints** | **Environment** | **Module (Object) Type** | **Internal Obstacles (holes)** | **Worst-Case Planning Complexity (time)** |
|---|---|---|---|---|---|
| Reconfiguration, out-of-place [this thesis] | Yes | Unlimited | Distinct IDs | Yes | $\Theta(n^2)$ |
| Reconfiguration, in-place [this thesis] | Yes | Crust | Distinct IDs | Yes | $\Theta(n^2)$ |
| Reconfiguration, with obstacles [this thesis] | Yes | Bounding region | Distinct IDs | Yes | $O(n^4)$ or no solution |
| Warehouse [44] | No | Rectangular bounding region | Rectangular (not square), distinct IDs | No | PSPACE-hard |
| $(n^2 - 1)$-Puzzle [43] | No | Square bounding region | Square, unique IDs | No | Polynomial |
| Sorting (not comparison-based) [25] | No | N/A | Distinct IDs | N/A | $\Theta(n)$ |

Table 1.1: Comparison of reconfiguration to related coordinated motion planning problems. Planning complexity listed counts time steps in an $n$-module robot. Tighter bounds for reconfiguration problems are given in Chapter 4.

### 1.3.3 Locomotion, Self-Repair, and Goal Recognition

Having fully introduced the reconfiguration problem, we now return to the other three problem areas studied in the thesis. It may be useful to refer again to Figure 1.3. Here we define the specific problem variations studied, list assumptions, and summarize results.

#### Locomotion

In planning reconfigurations, the goal can be specified in such a way as to effect a translation of the robot as a whole. It is straightforward to extend reconfiguration algorithms to address the problem of locomotion. Broadly, this approach divides locomotion into two levels. Low-level planning moves the robot in a specified direction, driven by a high-level path planner. The low-level planning is implemented using reconfiguration. An example is the rule-based locomotion approach of Butler, Kotay, Rus, and Tomita [9]. This is the algorithm used to generate the motion shown earlier in Figure 1.1. Individual modules control their motions by comparing local neighborhood information to a set of pre-compiled rules. This is a powerful method that is easy to implement, but the rule-base is tedious to build and care must be taken to prevent global disconnection. Restrictions such as fixed configuration height or width are used to ensure connectivity. Our approach requires additional planning during execution but guarantees connectivity, has no restrictions on shape, and can handle arbitrary obstacles.

We also investigated a low-level locomotion gait specific to unit-compressible systems, called Inchworm Locomotion. This is a purely distributed algorithm but requires a fixed connection topology. The unit-compressible actuation allows the robot to minimally change shape without altering module connectivity. We implemented this algorithm in hardware and performed extensive experiments.

Sliding-Cube modules, which are embedded in a square lattice, have fixed rotation. Likewise, the robot itself cannot actually turn. It can follow a rectilinear path through space, but rotation is constant. The low-level algorithms we described generally are more efficient at planning and controlling the straight segments of the path than the turns. Therefore, the high-level path planner should find a rectilinear path, among obstacles, that minimizes the number of bends. In two dimensions, this problem is called *2D-MBP* (two-dimensional minimum bend path) and has been studied in the context of VLSI wire-routing [60]. Our robots are intended to move over complex terrain, so we developed a solution to the 3D problem which we call 3D-MBP. The 2D-MBP algorithms generally assume rectilinear obstacles; we equivalently assume obstacles to be orthohedral in three dimensions. This is the first published solution to 3D-MBP, but other researchers are currently developing asymptotically faster algorithms.

### Self-Repair

As the robot performs locomotion or other types of reconfigurations, modules are likely to fail. Dealing with the eventuality of module failure is known as the self-repair problem. Our approach is a three-step strategy: 1) detect module failure, 2) eject the failed module, and 3) replace the failed module with a spare stored within the robot's structure. We do not consider the failure detection step, but the eject and replace steps can be cast as heterogeneous reconfiguration problems and addressed by extending reconfiguration planning methods. The 3D-MBP problem arises here as well, since the algorithms for cooperatively moving the failed module require extra steps at turns. We developed an algorithm that is the first planning-based approach to self-repair; earlier work achieved self-repair using distributed self-assembly [109].

### Goal Recognition

Another issue faced by an SR robot during reconfiguration is collective coordination and recognition of the current global state. Having no central controller, the robot operates as a tightly coupled distributed system where each module has access to local information only. The problem of how the robot collectively recognizes the achievement of some goal is called goal recognition. The goal can be the achievement of some desired configuration during reconfiguration or a desired location during locomotion. The reconfiguration problems we study subsume this problem and guarantee termination when the goal configuration is reached. However, it is useful to decouple goal recognition from reconfiguration so that different methods can be combined independently. This allows distributed reconfiguration methods to choose from existing goal recognition algorithms, for example.

We developed a heterogeneous goal recognition algorithm for detecting whether the current configuration matches a given heterogeneous shape in a distributed manner. This algorithm works in 2D or 3D, for configurations with or without holes, in linear time. We also implemented the algorithm in hardware and performed a number of validating experiments.

## 1.4 Contributions

In this thesis, we study reconfiguration and three related problems. The main contribution is an algorithmic basis for reconfiguration planning in heterogeneous SR systems. This includes a number of algorithms, simulation results, and hardware experiments. Simulations were performed using a software environment that we designed and constructed. Hardware experiments made use of an existing prototype platform built by other members of the Dartmouth Robotics Lab. Detailed contributions are summarized subsequently:

- First published solutions to the heterogeneous planning problem, presented in both centralized and decentralized versions.
- MeltSortGrow: An algorithm for heterogeneous reconfiguration planning that solves the problem out-of-place in asymptotically optimal time.
- TunnelSort: An in-place reconfiguration planning solution introducing motion through the volume in systems with native surface-motion. TunnelSort also runs in asymptotically optimal time.
- ConstrainedTunnelSort: An algorithm for reconfiguration planning within a given bounding region, allowing for disconnected free space. ConstrainedTunnelSort supports the case of reconfiguration among obstacles.

- Position Constraints: An algorithm for maintaining the relative position of specified module types during arbitrary reconfiguration.
- Compliant locomotion for systems of surface-moving modules.
- 3D-MBP: First published solution to the rectilinear minimum-bend-path problem among rectilinear obstacles in three dimensions. 3D-MBP is useful both for path planning during locomotion and module-level trajectory planning.
- Self-Repair: An algorithm for ejecting and replacing failed modules. Addresses issues of uncertainty at the level of abstraction assumed by other algorithms.
- Goal Recognition: An algorithm for recognizing the achievement of a given heterogeneous goal configuration in a distributed fashion.
- SRSim: A simulation engine for implementation, visualization, and animation of SR systems in three dimensions.
- Implementations and experiments using SRSim.
- Hardware experiments with the Crystal robot, a robot constructed previously in the Dartmouth Robotics Lab. These experiments are the first hardware experiments performed with a heterogeneous SR robot.

## 1.5 Outline

The thesis is organized as follows. Chapter 2 discusses previous work in SR robotics and related areas. In Chapter 3, we provide technical background material prerequisite to later chapters.

New results are presented in Chapters 4 through 6. We begin in Chapter 4 with a collection of solutions for heterogeneous reconfiguration planning. This includes both centralized and decentralized algorithms that solve the reconfiguration problem out-of-place, in-place, and in-place within a bounding region, as well as the position constraints algorithm. We discuss solutions for distributed heterogeneous goal recognition, locomotion, 3D-MBP, and self-repair in Chapter 5. In Chapter 6, we present implementations of our algorithms in simulation and in hardware, along with descriptions of the simulation environment we built and hardware platform we used. Chapter 7 concludes the thesis with a general discussion of our methods and suggestions for future work.

# Chapter 2

# Related Work

Research in modular and SR robots spans more than 15 years [38] of active investigation. During this time, researchers have obtained significant theoretical and practical results, and hardware prototypes have progressed from 2D tethered units such as the Fracta to sleeker, 3D robots such as MTRAN and the Molecule (see Figure 2.1). Work relevant to this thesis falls into the categories of hardware design, algorithms and theory, and cooperative robotics.

## 2.1 Hardware Design

Building reconfiguring robots in hardware involves designing and constructing the basic modular units that combine to form the robot itself. Such modules differ from the wheels, arms, and grippers of fixed architecture robots in that they are functional only as a group as opposed to individually. Because we are interested in developing general algorithms for classes of robots instead of particular systems, familiarity with the entire spectrum of existing systems is valuable. Current systems can be divided into classes based on a number of module properties.

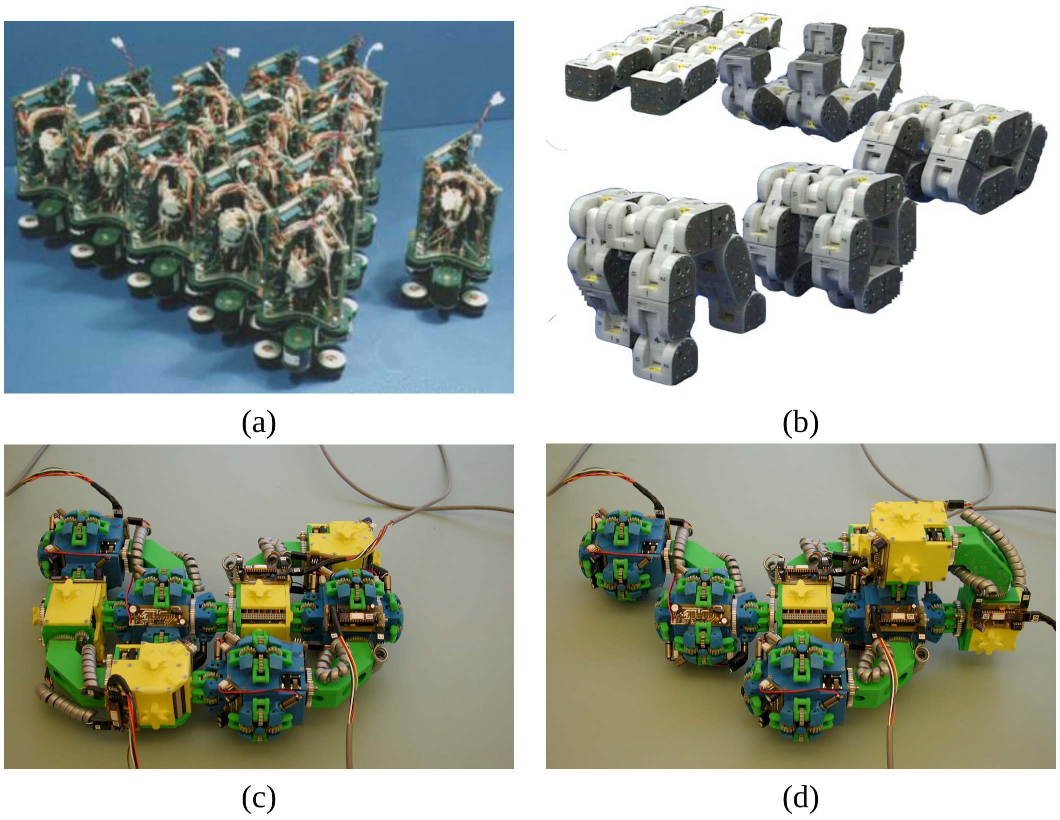


Figure 2.1: Examples of SR robots. The *Fracta* robot of Murata, Kurokawa and Kokaji [68] is shown in (a), tethers unplugged. In (b), the newer *MTRAN* robot of Murata et al. [70] is shown reconfiguring into a legged walker. The Molecule robot from the Dartmouth lab is shown in two configurations in (c) and (d) (Molecule robot photos courtesy of Dartmouth Robotics Lab).

Recall the following definitions. Systems composed of a single module type are known as homogeneous systems, and those with multiple module types are called heterogeneous systems. In lattice-based systems, modules move among discrete positions, as if embedded in a lattice. Chain-based systems attach together using hinge-like joints and permit snake-type configurations that connect to form shapes such as legged walkers and tank treads. See Figure 2.2 for typical examples from each class. Another class of modular systems cannot self-reconfigure, but can reconfigure with outside intervention. This class is called *manually reconfiguring* systems. In this section, we survey robots in each of these categories.

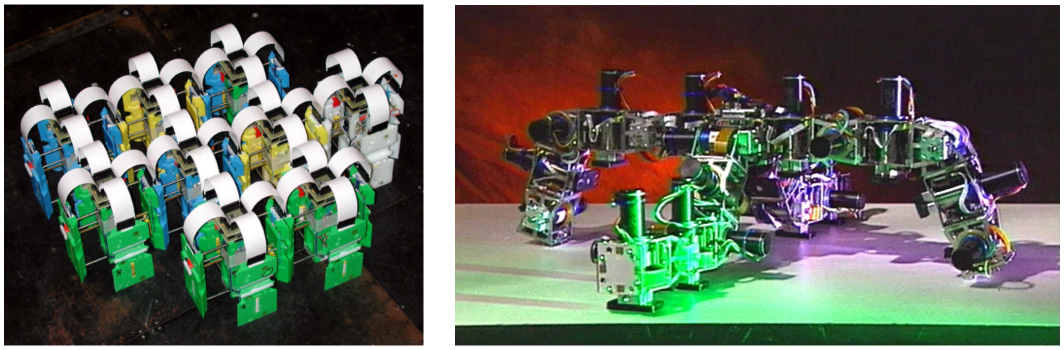

(a) (b)

Figure 2.2: Examples of (a) a lattice-based system, the Crystal, and (b) a chain-based system, Polybot (Crystal photo courtesy of Dartmouth Robotics Lab; Polybot photo courtesy of PARC).

### 2.1.1 Pioneering Research

*CEBOT* (cell structured robot) is the first proposed SR robot, introduced by Fukuda et al. in 1988 as an implementation of their 1987 idea of a *Dynamically Reconfigurable Robotic System* (DRRS) [39, 38]. The definition of DRRS parallels our current conception of SR robots – the system is made up of robotic modules (cells) that can attach and detach from each other autonomously to optimize their structure for a given task. The idea is directly inspired by biological concepts and this is reflected in the chosen terminology. It is interesting that this proposed SR robot is heterogeneous: cells have a specialized mechanical function and fall into one of three "levels." Level one cells are joints (bending, rotation, sliding) or mobile cells (wheels or legs). Linkage cells are part of Level two, and Level three contains end-effectors such as special tools. Communication and computation are assumed for all cells.

CEBOT is the physical instantiation of DRRS. Various versions range from reconfigurable modules [37] to "Mark-V," which more closely resembles a mobile robot team [13].

### 2.1.2 Lattice-based Robots

In lattice-based systems, modules are constrained to occupy positions in a virtual grid, or lattice. One of the simplest module shapes in a 2D lattice-based system is a square, but more complex polygons such as a hexagon (and a rhombic dodecahedron in 3D) have also been proposed. Because of the discrete, regular nature of their structure, developing algorithms for lattice-based systems is often easier than for other systems. However, the grid constraint makes implementing certain rolling motions, such as the tank-tread, more challenging since module attachment and detachment is required. We would like our algorithms to be implementable by most lattice-based systems, so a complete review of their properties is essential.

One of the first lattice-based SR robots proposed and constructed in hardware is the *Fracta* robot [68] (Figure 2.1(a)). The homogeneous, 2D *Fractum* modules connect to each other using electromagnets. Communication is achieved through infrared devices embedded in the sides of the units, and allows one fractum to communicate with its neighbors. Computation is also on-board; each fractum contains an 8-bit microprocessor. Power, however, is provided either through tethers or from electrical contacts with the base plane. This system was designed for self-assembly, and can form simple symmetric shapes such as a triangle, as well as arbitrary shapes [96]. Self-assembly and self-reconfiguration have a strong relationship, since self-assembly can be used for reconfiguring between different shapes. Other lattice-based robots from the same group include a smaller 2D system [107], and a 3D system [69].

The Metamorphic robot [77] is one of the first hardware prototypes for lattice-based modular SR robots in the USA. It was developed in the Robot and Protein Kinematics Lab at Johns Hopkins University and consists of modules designed as deformable hexagons. The modules are aggregated in a planar lattice by attaching neighbors on an adjacent

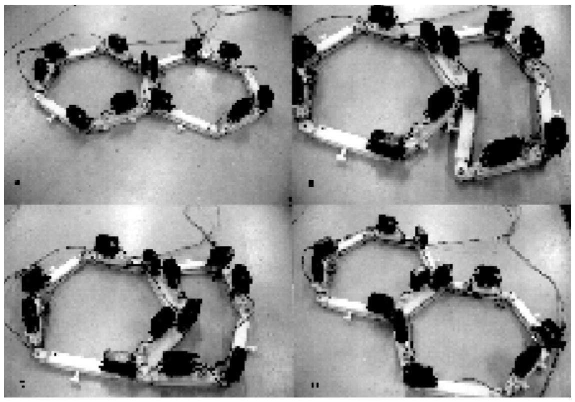


Figure 2.3: Two hexagonal robots moving relative to each other. Courtesy of the Robot and Protein Kinematics Lab at Johns Hopkins University.

hexagonal edge. Magnetic fields are used to swap connections, thus causing units to roll around each other as shown in Figure 2.3. The basic unit of this robot is a six-bar linkage forming a hexagon. The kinematics of this shape were investigated when the design was proposed [20], and hardware prototypes were constructed later [77]. A unique characteristic of this system is that it can directly implement a convex transition; a given module can move around its neighbor with no supporting structure. The hexagon deforms and translates in a flowing motion. A square shape with this same property was also proposed. This motion primitive is important since it is required by many general reconfiguration algorithms, but many systems can only implement it using a group of basic units working together.

The Metamorphic robot module is a hexagon in two dimensions, but a similar 3D module, the *Rhombic Dodecahedron*, was later proposed by Yim et al. [105]. The Rhombic Dodecahedron has 12 faces, and each face is a rhombus. Rhombic Dodecahedron modules pack tightly in a 3D grid and locomote by rolling around each other. This shape is an example of a class called *Proteo* modules [106].

The first module proposed by the Dartmouth group is the *Molecule* [55]. This 3D module consists of two *atom* units connected by a 90-degree *bond*, forming the overall shape of an elbow with a connection mechanism at each end (shown earlier in Figure 2.1). Each atom has five inter-Molecule connection points and two degrees of freedom. One degree of freedom allows the atom to rotate 180 degrees relative to its bond connection, and the other degree of freedom allows the atom to rotate 180 degrees relative to one of the inter-Molecule connectors at a right angle to the bond connection. The current design uses R/C servomotors for the rotational degrees of freedom. A feature of this prototype is the use of a gripper-type connection mechanism.

A variation on this actuation scheme is AIST's *Modular Transformer* (MTRAN) [70], shown in Figure 2.1. Like the Molecule, the MTRAN design is also based on two components connected by a link. The difference is that MTRAN's semi-cylindrical end components each can rotate 180 degrees, resulting in a variable bond angle. This actuation allows MTRAN to behave as both a lattice-based and a chain-based system. The modules can be closely packed in a 3D grid, or they can form chains such as legs in a legged-walker configuration. Communication is on-board the MTRAN, and power from an external source is transmitted between units through connections on the faces. Experiments demonstrating various modes of locomotion have been performed, including crawling and quadruped gaits and reconfiguration between modes. The latest prototype includes on-board power, distinguishing this robot as the first major untethered 3D system [58].

Another variation on this design is the *I(CES)-Cubes* robot [98]. The link component on this module, however, is separated from the end components. Therefore, the system is termed *bipartite*. The end units, the *cubes*, are passive and connect to *Link* units, which are actuated. In this way, the I(CES)-cubes implement motion primitives similar to the Molecule robot [97], although there are fewer constraints since each cube can be placed independently.

Motion of the previously discussed modules is primarily achieved by movement over the surface of the robot. Unit-compressible systems, alternatively, use modules that move through the volume of the robot. The actuation method of unit-compressible modules is termed *scaling-based* because the modules expand and contract in multiple dimensions. The basic idea of using extendable arms for actuation to construct a reconfigurable robot was patented by Tanie and Maekawa in 1993 [95]. The Crystal robot was constructed by the Dartmouth group as a 2D physical realization of a unit-compressible system [100, 82, 83]. Crystal units are squares that attach to each other at each of the four faces, and expand and contract in two dimensions. Communication in the Crystal robot is between neighbor units only, and is implemented using infrared devices mounted in the faces. Both power and computation for the Crystal are on-board each unit.

A unit-compressible system extended to three dimensions was developed at Xerox PARC [57]. This implementation, the *Telecube*, is similar to the Crystal but has an added degree of expansion and contraction for the third dimension [94]. An external power source is required but the Telecube avoids a large number of tethers by routing power between modules.

Other actuation models have also been proposed for lattice-based systems. Hosokawa et al. [45] designed and constructed a system based on cubes with actuated arms. The arms attach to other units and allow one cube to pull another cube from its side to its top. This robot is two-dimensional, but in the vertical plane. Another vertical 2D system, CHOBIE (Cooperative Hexahedral objects for Building with Intelligent Enhancement) was recently proposed [48, 53]. The CHOBIE modules are square and interlock with a system of rails and grooves for a high degree of mechanical rigidity. A system using pneumatic actuators, along with hardware experiments, was presented by Inou, Koseki and Kobayashi [47]. Guéganno and Duhaut designed and prototyped a module called MAAM (Molecule = Atom — Atom + Molecule) consisting of a sphere (the atom) with six legs (bonds) [41]. Hydron, a system designed to operate underwater, was described by Konidaris, Taylor, and Hallam [52]. The Hydron module uses water jets to control its position.

A new robot currently under development by the HYDRA EU project [29] is called the ATRON [75, 24, 76]. The ATRON is a spherical 3D module with arm-like connectors. The upper and lower hemispheres rotate with respect to each other, allowing for module locomotion. A number of prototypes have been developed, with a goal of producing 100 modules. If successful, the ATRON would be the largest SR robot constructed thus far.

### 2.1.3 Chain-based Robots

In chain-based systems, modules aggregate as connected 1D strings of units. This class of robots easily implements rolling or undulating motions as in snake robots or legged robots. However, control is much more difficult for chain-based systems than for lattice-based systems because of the continuous nature of the actuation: modules can move to any arbitrary position as opposed to a fixed number of neighbor positions in a lattice. Our work does not consider chain-based systems, but it is important to understand their characteristics in the interest of developing more generalized algorithms.

The first prominent chain-based system is *Polypod*, proposed by Yim in 1993 [103, 104] and developed at the Palo Alto Research Center (PARC). Polypod is made up of *Segments*, which are actuated 2-DOF 10-bar linkages, and *Nodes*, which are rigid cubes housing batteries. Polypod modules consist of two square parts that can rotate by 90 degrees relative to each other by a standard hobby servomotor. The inter-module connection mechanism is provided by two connection plates on either side of the module which are identical and have a four-way rotational symmetry at 90 degree increments. Four grooved pins enter four holes and are grabbed by a latching mechanism that is released by a shape-memory alloy actuator. Multiple gaits for locomotion, including rolling, legged, and even Moonwalk gaits, were demonstrated with Polypod.

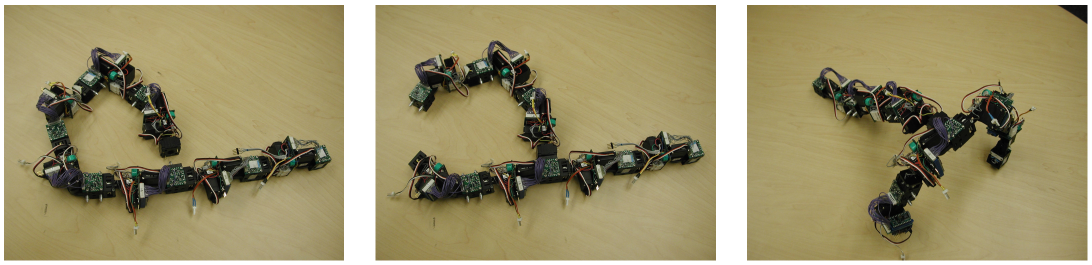


Figure 2.4: CONRO in three configurations. Photos courtesy of USC Information Sciences Institute (copyright ISI).

*Polybot* succeeds Polypod, sharing the same bipartite structure (see Figure 2.2). Although Polypod is manually reconfigurable, Polybot is self-reconfigurable. Segments in Polybot abandon the 10-bar linkage in favor of a 1-DOF rotational actuator. The latest generation of Polybot prototypes has on-board processing and CANbus (controller area network) hardware for communication. Two CANbuses on each module allows the chaining of multiple module groups to communicate without running into bus address space limitations.

A system that uses a similar actuation design is CONRO (CONfigurable RObot) [89], shown in Figure 2.4. The CONRO module has two rotational degrees of freedom, one for pitch and one for yaw, and was designed with particular size and weight considerations [15]. Considerable attention has been paid to the connection mechanism, which is a peg-in-hole connector with SMA (shape-memory alloy) latching mechanism that can be disconnected by either face. Computation is on-board each module, so unlike Polypod, CONRO has only one module type. Power can be provided externally or via batteries on later prototypes [16]. Examples of manually configured shapes are the snake and the hexapod, and the current CONRO system is designed for self-reconfiguration.

The DRAGON is a snake robot with torsion-free (constant-velocity) joints [72]. A sophisticated connector has been developed for the DRAGON, designed for strength[1] and

[1]The author demonstrated this by suspending himself from the connector, hanging from the sixth floor of a building.

tolerance for docking [73].

### 2.1.4 Heterogeneous Systems

SR systems with significant levels of heterogeneity are few. The most prominent heterogeneous system is CEBOT [38], although bipartite systems such as I(CES)-Cubes and Polybot are minimally heterogeneous. Heterogeneity results from adding various sensors to any existing system, as well. Another source of heterogeneity is module failure; failed actuators lead to a system with immobile modules. Differential battery drain in systems with on-board power can also cause actuation speed differences between modules. This heterogeneity might seem trivial from a mechanical perspective but is significant algorithmically since it leads to violations of the assumption of module interchangeability. The algorithmic results in this thesis encourage the development of new heterogeneous systems.

### 2.1.5 Manually Reconfigurable Robots

Modularity, the concept of designing a system divisible into discrete and relatively independent pieces, is often considered a desirable characteristic in designing systems. Manually reconfigurable modular systems share many design issues with self-reconfigurable systems, so a brief review is given here. Many groups have developed modular robotics [22, 61] and manipulators, such as the *Reconfigurable Modular Manipulator System* (RMMS) of Paredis, Brown and Khosla [79], and Goldenbergs modular arm [46]. The general problem of reconfigurable modular design is explored by Chen and Burdick [18], and Farritor and Dubowski [30].

## 2.2 Planning and Control

In attempting to devise new algorithms, it is useful to understand and compare techniques developed previously. In this section, we survey early theoretical work, approaches to the reconfiguration problem, and also work addressing locomotion, self-repair, and division tasks.

### 2.2.1 Early Algorithms

In SR research, CEBOT work is prescient in that a majority of current research issues in SR robotics are identified in early CEBOT papers. Communication and docking problems are described in [35]. Distributed decision making [36], hierarchical control [40, 13] and analysis of how the number of modules affects performance [49] are also studied. Other early theoretical work in planning and control for modular systems comes from the perspective of cellular automata theory. Gerardo Beni proposed the idea of a "cellular robotic system" around the same time as the first CEBOT papers [3]. Here, the familiar idea of large numbers of mobile cooperating autonomous units is reiterated, but physical reconfiguration is not included. Distributed control with no synchronized clock is emphasized. Further papers discuss theoretical [4] and practical [42] issues. For additional information on early cellular and cooperative robotics research, see survey papers by Sandini [86] and Cao, Fukunaga and Kahng [14].

### 2.2.2 Reconfiguration Planning

Solving the reconfiguration planning problem is fundamental to SR systems research. In some approaches, explicit start and goal configurations are given, and in others the goal shape is defined by desired properties. Recall that centralized algorithms require global system knowledge and compute reconfiguration plans directly, whereas decentralized algorithms compute solutions in a distributed fashion without the use of a central controller. Reconfiguration algorithms can be designed for specific robots, or for classes of modules. Often, a centralized solution is more obvious and is developed first, followed by a distributed version. Not all decentralized algorithms are guaranteed to converge to a solution, or are correct for arbitrary goal shapes. We review relevant reconfiguration algorithms in this section.

CEBOT reconfiguration was planned by a central control cell known as a *master* [37]. Master cells were later intended to be dynamically chosen, blurring the distinction between centralization and decentralization [40]. Later CEBOT control is hierarchical (behavior-based) [13].

A common technique used in reconfiguration algorithms for lattice-based systems is to build a graph representation of the robot configuration, and then to use standard graph techniques such as search to compute motion plans. Planning for the Molecule robot developed by the Dartmouth group is one example [55, 56]. Another example from the Dartmouth group is planning for unit-compressible systems such as the Crystal [100, 82]. This planner, named MeltGrow, uses the concept of a meta-module, where a group of modules are treated as a single unit with additional motion capabilities. The Crystal robot implements convex transitions using meta-modules called *Grains*. Graph-based algorithms are also used by the MTRAN planner to compute individual module trajectories [108].

Centralized planners can also store pre-computed data structures such as gait-control tables. Once a gait is selected by the central controller, it is executed by local controllers on the individual modules. This type of algorithm is used by Polypod [104]. The division between central and local controllers is also used in by RMMS [79], and I-Cubes [98].

Important work in decentralized planning begins with the Fracta system [68]. Individual units used a precompiled set of rules to self-assemble into various shapes, and eventually into arbitrary shapes [96]. The randomized component of these algorithms limits convergence guarantees, however, as the algorithms are similar to simulated annealing. A similar algorithm is presented by Hosokawa et al. [45]. This algorithm uses simpler rules and is deterministic, but is more limited in the classes of shapes it can form. Shapes with "overhangs" are disallowed, for example. Yim, Duff and Roufas [106] present a distributed controller for Proteo modules that achieves arbitrary shapes, but again without convergence guarantees. An interesting method for distributed control of hexagonal modules is given by Walter, Welch and Amato [101]. Recent work by Stoy [90] uses a gradient-based method coupled with a scaffold structure to perform concurrent reconfiguration. A multi-resolution representation of the goal configuration is also proposed [91].

A successful approach in distributed planners is the use of message-passing. Shen, Salemi and Will [88] and Salemi, Shen and Will [85] propose a control system for CONRO using a message-passing scheme called *Digital Hormones*. The problem of distributed reconfiguration for unit-compressible modules was solved by a combination of the *Pacman* algorithm developed by the Dartmouth group [5] and later modifications and analysis by Vassilvitskii, Yim and Suh [99]. This distributed algorithm is correct and complete for arbitrary shape reconfigurations of 3D unit-compressible cubic systems using metamodules, but makes explicit use of module homogeneity.

In addition to algorithmic solutions, other theoretical issues related to the reconfiguration problem have been addressed. Chirikjian and Pamecha [21] discuss bounds for self-reconfiguration. Metrics for reconfiguration planning have also been studied [78, 19], and Vassilvitskii, Yim and Suh provide complexity analysis for their distributed reconfiguration algorithm [99].

### 2.2.3 Other Tasks

Aside from the reconfiguration problem, the locomotion problem for reconfigurable robots has also received much research attention. Generally, chain-based systems locomote without module detachment, while lattice-based systems require reconfiguration to perform locomotion. Yim [104] demonstrates control for rolling and legged locomotion gaits for Polypod. Stoy, Shen and Will [92] present distributed locomotion algorithms for CONRO. Other distributed locomotion work, based on cellular automata-style algorithms, is presented by Butler, Kotay, Rus and Tomita [7].

When modules in an SR system fail, it is desirable for the robot to fix itself, or *self-repair*. The self-repair problem was first studied by Yoshida et al. [109].

Another useful capability of SR robots is self-replication, the ability of a large robot to divide itself in several independent smaller robots with the same basic functionality (but not identical size). For example, a system consisting of 100 modules could function as one large robot or 10 smaller robots each consisting of 10 modules, or any number of other configurations. Self-replicating robots are useful in tasks where the overall effectiveness and task completion time is improved by parallelism, such as distributed surveillance or exploration. Solutions to this problem are in their infancy, but some first results are described in [10].

## 2.3 Other Related Results

Results from communities other than modular robotics relate to our algorithmic questions, as well. This work comes from the fields of heterogeneous robot teams, self-assembly and formation control.

There is a large body of literature investigating teams of mobile robots. Often, researchers are interested in how teams of varying types of robots, or heterogeneous teams, can cooperate in accomplishing a common task. SR and modular robots can be thought of as a special case of a tightly coupled robot team. See Balch and Parker [12] for an excellent review of the field.

Another related field of research is self-assembly. An interesting recent result is by Nagpal [71], who presents a programming language for constructing global shapes from independent agents using “Origami Mathematics.” Other work studies how free-floating square tiles can self-assemble into global shapes [81, 2]. Inspired by biological processes, Saitou and Jakiela [84] investigate how to self-assemble subassemblies with conformational switches. More recent work based on conformational switching is by Klavins [51], focusing on the synthesis of graph grammars to control the generation of a specified assembly. Important work by Lipson and Pollack [62] studies automatic design and manufacturing of novel robot structures using genetic algorithms and rapid-prototyping technology. Lipson investigates stochastic methods for self-assembly and self-reconfiguration in later work [102].

Vehicle formations are assemblies of self-controlled mobile vehicles, such as a squadron of aerial robots. The controls community has recently investigated distributed controllers for such formations. This is similar to modular robot control, without the physical coupling of modules. Representative work is presented by Fax and Murray [32, 31], Klavins [50], and Olfati-Saber and Murray [74].

# Chapter 3

# Basic Techniques in Planning for Lattice-Based Systems

Here, we briefly review a collection of common definitions and concepts for convenience. We begin with relevant simple properties of graphs, outline existing techniques for planning in homogeneous lattice-based systems, and then briefly discuss complexity of reconfiguration planning. A summary of the cellular-automata model, included for comparison to other models, concludes the chapter.

We observe that the techniques in this chapter, as well as the new results presented in the thesis, do not consider issues of dynamics. Extensions that do include dynamics are suggested in Chapter 7. Likewise, we assume that uncertainty in module-level actuation is handled by low-level controllers and is thus not present at this level of abstraction. Total failure of low-level controllers is handled by our self-repair approach, presented in Chapter 5. Lastly, methods for surface-moving modules are specified for systems embedded in a square lattice but also are applicable to other types of lattices. Motion through the volume does not generally extend in this manner.

## 3.1 Connectivity Graph Representation

Given a lattice-based modular robot, we can represent it with a *module connectivity graph*, or equivalently, *connectivity graph*, which is an undirected graph $G = (V, E)$ where the set of vertices $V$ contains exactly one vertex for each module and the set of edges $E$ contains an edge connecting the vertices corresponding to each pair of adjacent stationary modules. Throughout this thesis, we assume $G$ is connected. All module operations must preserve connectivity.

An *articulation point* is a vertex $v$ such that removal of $v$ causes disconnection. Therefore, a module corresponding to an articulation point may not be moved. We can label all articulation points in a graph using the standard algorithm based on *depth-first search* (DFS). Every finite connected graph contains at least two vertices that are not articulation points [28].

A module $m$ is considered *mobile* if and only if its corresponding vertex is not an articulation point and $m$ can execute a primitive motion. In other words, $m$ is not locked in place by surrounding modules.

A meta-module is a group of modules that are treated as one. In the connectivity graph, there is a single node for each meta-module. Meta-modules have enhanced motion primitives that are composed of the native motions of component modules. This technique is normally used as an aid in reconfiguration planning. For example, a *tile* is such a structure for modules of the Molecule robot [56]. Meta-modules of unit-compressible systems are discussed in the next section.

## 3.2 Internal Paths: Unit-Compressible Systems

Module trajectories can be categorized as motion over the surface or through the volume of the structure. Sliding-Cube systems support only surface motion natively, but unit-compressible systems move through the volume [83]. *Virtual module relocation* refers to the idea that, in a homogeneous structure, a module can be "virtually" moved to a distant location by shifting modules along an internal path. Each module along the path moves by one position. This can be thought of as shifting the unoccupied position, or hole, to the position of the original module.

In unit-compressible systems, a single module has limited motion capabilities. Often, meta-modules called *grains* are used for locomotion and trajectory planning purposes. For a unit-compressible system organized into grains, an individual grain can follow any path through the connectivity graph via virtual module relocation. Such a path is rectilinear due to the geometry of the modules. By repeatedly relocating individual modules, the system as a whole can reconfigure to match a given goal. Algorithm MeltGrow is an example. See [83] for a full exposition of MeltGrow and general reconfiguration planning for unit-compressible systems.

## 3.3 Surface Paths: Sliding-Cube Systems

Sliding-Cube modules primitively move over the surface of other modules. A mobile module can reach any position on the surface of the structure. Homogeneous configurations with a single surface, and thus no holes, can be reconfigured into a goal configuration using a simple greedy method. The algorithm is to iteratively find a mobile module in the current configuration whose position is unoccupied in the goal configuration, and relocate it to a free position that is occupied in the goal. This operation can always be performed until the goal configuration is reached [54]. We prove this property for configurations with holes in Chapter 4.

## 3.4 Message-Passing

*Message-passing* refers to the transmission of a discrete unit of data, a *message*, from one module to another via some communication system. In Sliding-Cube systems, communication is possible between connected modules only. Therefore, message-passing occurs only between connected neighbors. *Broadcast* communication sends a message from a single source module to multiple destination modules simultaneously. Sliding-Cube systems have no native broadcast facility, but we can simulate broadcast through message-passing. When a module first receives a message, it resends the message to all other neighbors. If a module receives a message it has already seen, it does nothing. Messages can be identified by unique sequence numbers to implement this. Alternatively, the initial receipt of a message identifies the sender as the *parent*. A module only resends messages received from its parent. These concepts are described in detail in [63].

This broadcast method results in messages traversing the connectivity graph according to pre-order traversal of an induced spanning tree. We can use this technique to perform computation, as well, with the addition of a *return message*. When a module receives a message it has already seen, or has no neighbors other than its parent, it performs some computation and sends the result in a message back to its parent. After a module has received return messages from all children, it similarly computes a result and sends a return message. The module that sent the original message, the root of the spanning tree, can then compute a final result. This method represents post-order traversal of the spanning tree. We use this technique frequently in our decentralized algorithms. For example, we can easily identify the module with the minimum relative coordinate by passing the current minimum as message data. The computation performed by a module is to compare its coordinate with the current minimum. The root module broadcasts the final result. This can be thought of as a form of *leader election*.

## 3.5 Complexity of Reconfiguration

The number of primitive motions required for some instance of reconfiguration in homogeneous lattice-based systems is $\Theta(n^2)$, where $n$ is the number of modules. In other words, the Reconfiguration problem has a worst-case lower bound of $\Omega(n^2)$. Consider reconfiguring a horizontal line into a vertical line. Regardless of where the two lines intersect, each module in the horizontal line must travel lattice-distance proportional to the length of the line. For $n$ modules, total lattice-distance traveled is $\Theta(n^2)$. Assuming a single motion primitive moves a module by a lattice-distance of one, the number of moves is also $\Theta(n^2)$. Bounds for reconfiguration are discussed in detail in [21]. Comparison to other coordinated motion planning problems is given in Chapter 1.

## 3.6 Cellular Automata Model

In contrast to the above planning-based methods that search the entire configuration, reconfiguration also can be approached using local information only. We include a summary of one of these methods, inspired by cellular automata, for comparison. A general discussion of how our results compare to other methods is given in Chapter 7.

The *cellular automata model* is a reconfiguration approach in which modules use local configuration only in deciding when to move [9]. Each module has access to a set of rules that maps local configurations to actions such as actuations and internal state updates. Various evaluation models are possible, dictating how frequently a given module queries the rule set. These models range from round robin to totally asynchronous evaluation order. The rule sets have been shown to produce global behaviors such as locomotion in a straight line, turns, and locomotion among obstacles. These rule sets are currently manually constructed, but automated methods have been developed for proving correctness.

# Chapter 4

# Reconfiguration Algorithms

The versatility and adaptability of SR robots hinges on their ability to plan and control shape changes. The reconfiguration planning problem intuitively seems as though it should be fundamentally less complex in homogeneous systems since any module can substitute for any other. However, this intuition turns out to be misleading. In this chapter, we present the main results of the thesis: the worst-case planning complexity of common formulations of the reconfiguration problem in the heterogeneous case is equivalent to that of the homogeneous case. We show this with a sequence of three algorithms for reconfiguration planning under increasing free space constraints. MeltSortGrow (MSG) is an out-of-place solution, TunnelSort (TS) is a corresponding in-place solution, and ConstrainedTunnelSort plans reconfigurations among obstacles modeled as arbitrary and possibly disconnected free space bounding regions. We conclude the chapter with our approach to a novel problem formulation that is more closely tied to heterogeneity, reconfiguration according to relative module position. This problem is addressed by the PositionConstraints (PC) algorithm.

To begin, recall the following problem specification defined in Chapter 1. The heterogeneous reconfiguration planning problem is how to compute a feasible plan that, when

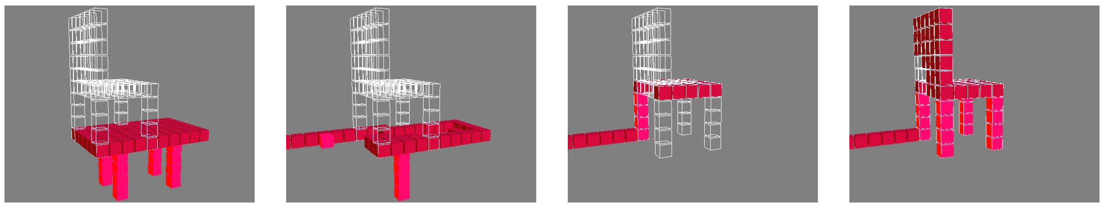


Figure 4.1: Simulation of table-to-chair reconfiguration with two module types, planned by MSG. White outline indicates the goal shape. Total number of moves in the reconfiguration is 12,143.

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 relative position. Position is defined by coordinates within a common reference frame. Start and goal configurations share this reference frame, so alignment between the two is given. We assume modules defined by the Sliding-Cube model. Modules 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 transitions from the face of a neighbor module to an adjacent face of the same modules. Modules in adjacent lattice-positions are assumed to be connected at their faces.

An example reconfiguration is shown in Figure 4.1. Notice that since intermediate configurations build a single long line of modules, this reconfiguration is out-of-place. An in-place version would only place modules in the union of the chair and table shapes plus a one module-width surface covering this union. Arbitrary free space constraints could further restrict modules from moving below the legs of the chair, for example, to model the case where the chair is resting on a rigid surface.

The Sliding-Cube model, defined and discussed in Chapter 1, is an abstraction that allows us to focus on the computational aspects of the problem by encapsulating architecture-specific details. This is a common technique in reconfiguration planning, and the Sliding-

Cube is a useful model because it can be realized by a variety of SR robot modules using a constant number of modules and moves. A number of such instantiations are given by Butler et al. [9]. Systems covered include the Crystal, Molecule, hexagonal-lattice systems like the Fracta and metamorphic robot, and M-TRAN. Planning for the Sliding-Cube model ensures that our algorithms are applicable to many different SR systems. Efficiency of our general methods versus architecture-specific methods is determined by the size of the constant factor involved in translating native motion primitives into Sliding-Cube primitives.

The notion of composing low-level motion primitives into more complex primitives can be extended as a means of understanding our general algorithmic approach. Sliding-Cube motions are composed to build module-trajectory primitives. The two types of module trajectories are motion over the surface and through the volume of the structure. These trajectory primitives are then composed into reconfiguration plans. For example, MSG uses surface motion only, and TS and CTS utilize both types of trajectory primitives. Compositions of trajectory primitives can also be used to address problems other than reconfiguration. Our approach to locomotion and self-repair, presented in Chapter 5 are examples.

Aside from actuation, the other important property of the Sliding-Cube model is that it assumes no central controller and no common communication channel. Communication is possible between connected modules only, and each module contains computational resources sufficient for processing this communication. This assumption is common in planning and control research for a number of reasons. First, a central control module creates a single point of failure. Distributing control over the entire robot helps to increase fault tolerance. Another issue is scalability; the fear is that centralized control will become too slow as the size of the robot increases to thousands of modules. Distributed control has the advantage of reduced communication cost and better support for parallelism.

The approach we adopt is termed decentralized because it combines the advantages of easily-analyzed sequential algorithms with the ability to execute on a distributed system. We first develop a centralized solution and prove correctness and running time bounds. Then we convert this into a decentralized version using message-passing. We assume that all messages are delivered eventually by the underlying network protocol. The advantages and drawbacks of this approach compared to other possibilities are discussed in Chapter 7, but the most important benefit offered by this decentralized approach is that we can provide performance and correctness guarantees unavailable in other distributed methods.

In order to specify decentralized algorithms in pseudocode, we invented a format loosely based on message-passing algorithms by Lynch [63]. Pseudocode is divided intro 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. See Algorithm 5 for an example.

Results for our three main algorithms are summarized in Table 4.1. The table includes assumptions made by each algorithm, and worst-case complexity analysis for the number of time steps required in planning and the number of moves in the plan produced. The analysis listed is parameterized by a number of factors, but can also be stated in simpler terms. Briefly, the plan size complexity for MSG and TS is $\Theta(n^2)$, and for CTS is $O(n^4)$. The bound for CTS is not tight; we discuss better bounds when CTS is presented in Section 4.3.

We now present our main algorithmic results in detail. The chapter is organized into four major sections. Sections 4.1, 4.2, and 4.3 present the algorithms MSG, TS, and CTS,

| | **MeltSortGrow** | **TunnelSort** | **Constrained-TunnelSort** |
|---|---|---|---|
| **Module abstraction** | Sliding-Cube | Sliding-Cube | Sliding-Cube |
| **Configurations with no holes** | yes | yes | yes |
| **Configurations with holes** | yes | yes | yes |
| **Free space requirements** | unlimited | crust | bounding region |
| **Worst-case planning complexity for shape-forming** | $O(4n^2)$ centralized, $O(3n^2 + n^3)$ decentralized | N/A | $O(n^2)$ |
| **Worst-case number of moves for shape-forming** | $O(3n^2)$ | N/A | $O(np)$ |
| **Worst-case planning complexity for sorting by type** | N/A | $O(25mn + \min(4mt^2, 4n^2))$ | $O(mn + m^2 + 25m^2n + 4m^2t^2)$ |
| **Worst-case number of moves for sorting by type** | N/A | $O(22mp + \min(4mt^2, 4n^2))$ | $O(22m^2p + 4m^2t^2)$ |

Table 4.1: Summary of heterogeneous reconfiguration planning complexity and assumptions for three main algorithms, where $n$ is the size of the robot in modules, $m$ 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, $t \leq p \leq n$. Worst-case complexity analysis is given; average-case complexity can be obtained by substituting average values of $m$, $t$, and $p$ for maximum values in the listed results. MSG has entries marked N/A since it does not separate homogeneous and heterogeneous phases. Likewise, TS has no homogeneous phase (typical greedy homogeneous reconfiguration would be $\Theta(n^2)$ time and $\Theta(np)$ moves). TS and CTS can execute tunnel moves in parallel, reducing actuation time by a factor of $t$.

respectively. The final section describes the PC algorithm. Analysis and discussion is provided for all algorithms within their individual sections.

## 4.1 Out-of-Place Reconfiguration

*MeltSortGrow* is an out-of-place algorithm that solves the heterogenous reconfiguration planning problem. This planner reconfigures the initial shape into a regular intermediate structure, and then builds the goal configuration. If cast as an instance of the Warehouse Problem, this approach would be to first move all boxes into a separate room, and then individually place them in their correct positions. The inputs to the algorithm, as defined above, are two configurations $C$ (the starting configuration) and $C'$ (the goal configuration). The output is a feasible plan that, when executed from $C$, results in $C'$. The length of the plan is $O(n^2)$, where $n$ is the number of modules in the robot, and the running time of the algorithm is also $O(n^2)$.

This solution builds on an earlier algorithm, MeltGrow, that plans reconfigurations for homogeneous systems with unit-compressible modules [83]. MeltGrow consists of two phases. The *Melt* phase builds an intermediate configuration, and the *Grow* phase builds the goal out of this intermediate shape. Different intermediate structures are specified based on dimensionality. A line is used for a 2D problem, while a plane is used in 3D. Rather than transforming from the initial configuration directly to the goal configuration, we generate an intermediate configuration that is easy to reach from any initial condition, and easy to transform into the goal configuration. In this case we choose a single line, or chain, as a simple intermediate configuration. As a homogeneous planner, MeltGrow cannot guarantee the final position of specific modules in the configuration. Our algorithm addresses this by reconfiguring the intermediate structure such that module position is correct as the goal shape is grown. The subgoal of this added *Sort* step is determined by computing a disassembly order of the goal. Another difference between the two algorithms is that MeltGrow is specific to unit-compressible systems and uses meta-modules (grains), whereas MeltSortGrow assumes Sliding-Cube actuation. Since unit-compressible modules are an instantiation of the Sliding-Cube model, MeltSortGrow can solve any instance that MeltGrow can.

In this section, we present the general approach of MeltSortGrow and detail its three major steps. We describe the centralized algorithm first, and then the decentralized version. Analysis and discussion accompany both versions. We implemented MeltSortGrow in simulation and tested the implementation with randomly generated start and goal configurations; these results are detailed later in Chapter 6. We also instantiated the sort phase of the algorithm in hardware using the Crystal Robot. These experiments are also described in Chapter 6.

### 4.1.1 MeltSortGrow Algorithm

The generic algorithm outline is listed in Algorithm 1. We first plan a reconfiguration from the start configuration to the intermediate structure. Then we sort the modules in the intermediate structure to prepare for the grow step, where we plan a reconfiguration from the modified intermediate structure to the goal shape. The grow step is basically a Melt of the final configuration, in reverse. The disassembly order computed from the goal configuration determines the sort order. This guarantees that the module chosen in Line 5 can always find a path to a free position in the goal configuration with the correct type. In other words, the module's type matches the type specified in the goal configuration at the module's final position.

**Algorithm 1** Generic centralized out-of-place algorithm for heterogeneous self-reconfiguration.

```
1: "Melt" configuration into 1D linear chain
2: Compute feasible assembly order for goal shape
3: Sort chain by assembly order
4: while Reconfiguration is not complete do
5:     Move next module into final position
```

An example reconfiguration is shown in Figure 1.4. Here, a table shape is reconfigured into a chair shape. There are two module types in this example; the legs of the table are composed of one type $t_1$, and the remaining modules have type $t_2$. In the chair configuration, the legs also are specified as type $t_1$, and all other modules are type $t_2$. Note that the intermediate structure is formed from the rear leg of the chair. The sorting step is not shown. The number of moves in this example is 3,513 for the Melt phase, 4,984 for

Sort, and 3,646 for Grow, or 12,143 in total.

### Melt

The objective of the Melt step is to compute a plan that transforms the initial configuration into the intermediate configuration, a line. As in MeltGrow, we repeatedly choose a module in the current configuration and move it to a free position in the intermediate structure. To choose a module in the current configuration, we identify all mobile modules using standard graph search techniques. Then we find a path from any mobile module to the end of the chain, called the *tail*.

**Algorithm 2** Melt algorithm builds intermediate structure (1D chain).

```
1: $I$ is intermediate configuration, initially empty
2: while Modules remain in initial configuration $C$ do
3:     Find articulation points of $C - I$
4:     Find path p from root of $I$ to non-articulation point module m in $C - I$ using BFS
5:     Move m to tail (end of $I$) using p
```

The melt step is specified in Algorithm 2. To begin, we choose one leftmost module from the initial structure and label this the *root*. The root is the minimum of all modules when sorted by x, y, and z coordinates within a coordinate frame centered at any arbitrary module. We will grow the intermediate structure to the left of the root, since by our choice of root we know there are no other modules to its left. In Line 3 we compute articulation points, and Line 4 chooses a surface module from the set of non-articulation points. If we begin at the tail, and search all possible module paths, then any module we reach must be on the surface. Observe that in the Sliding-Cube model, the only reachable positions are adjacent to existing modules. We call these *path* positions. In Line 4 of the algorithm, we search through path positions using breadth-first-search (BFS), starting from the tail position and terminating when we reach a non-articulation point module. The resulting path is transformed into a motion plan and executed in Line 5 by reconstructing

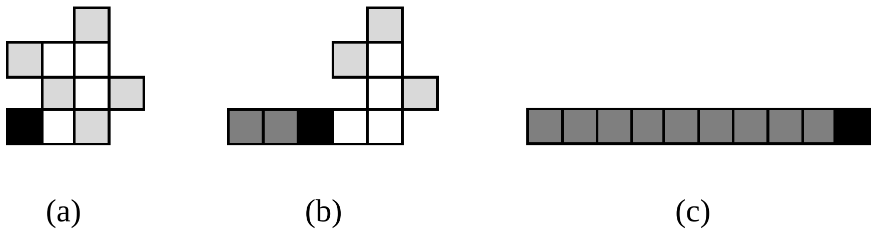


Figure 4.2: Illustration of Melt phase. In (a), the root module (lower-left) is darkened and mobile modules are shown in light grey. The state after two modules have been moved is shown in (b), with modules in the intermediate structure shown in medium grey. The complete intermediate structure is drawn in (c).

the motion primitives used in the BFS path. We repeat this procedure until all modules have been moved, which is once per module. See Figure 4.2 for an illustration of this step.

### Sort

The next phase of the algorithm is to modify the intermediate structure such that it is possible to easily grow the new configuration in the final phase. Assembly order is important since specific modules must be moved into their assigned positions. We approach this problem by sorting the modules in the intermediate chain, where the sorting corresponds to a feasible assembly order of the goal configuration. Therefore we must compute a feasible assembly order and then physically sort the modules according to this sequence.

The sort step is specified in Algorithm 3. First we compute the (dis)assembly order of the goal configuration using the melt technique described earlier, although instead of computing a path for each module we simply label it with its assembly order and mark it deleted. This can be thought of as a virtual melt. Reversing the resulting disassembly order yields a total ordering on the modules in the intermediate chain, assuming all modules have unique types. The case where modules share types can be handled by artificially labeling modules as convenient.

**Algorithm 3** Compute feasible assembly order and sort chain.

```
1: Given intermediate configuration $I$
2: for counter $c$ = 1 to $n$ do
3:     Find articulation points in goal configuration $C'$
4:     Find path *p* from root to non-articulation point module *m* in $C'$ using BFS
5:     Label *m* with $c$ and mark as deleted
6: Move left half of I on top of right half, forming rows *a* and *b*
7: while *a* is not empty do
8:     Find module *m* in *a* with minimum label
9:     Move *m* to leftmost unoccupied position in row below *b*
10: Repeat for row *b*, into row below *a*
11: Merge rows $a$ and $b$ into one sorted row above $a$, in the style of MergeSort.
```

In order to physically sort the chain, we use the simple quadratic-time sorting algorithm SelectionSort, and also one step of MergeSort. Removing a module from the middle of the line violates connectivity constraints, but the same operation on a double line is permissible. Therefore, in Line 6 we create a double line by iteratively moving half of the modules from the end of the line to form a second row connected to the original row. Now we are free to use SelectionSort to move modules in the top line into sorted position in a new row below the bottom line. Continuing in this way, we obtain two sorted rows. Now we simply merge these two as in MergeSort into a final sorted line adjacent to the double row. By carefully choosing where the new rows are created, the final sorted line is assembled at the same position as the original intermediate structure. See Figure 4.3 for an illustration of this phase of the algorithm. The one special case to consider is to ensure that the rightmost modules stay connected during the MergeSort step. This can be handled in a number of ways, but the simplest is to test the sort order of the upper rightmost module against the lower rightmost module, and swap the upper module with one from the bottom row if necessary before beginning MergeSort. This guarantees that as the rows are merged, the lower rightmost module will be merged before the upper one and the configuration remains connected. The following Lemma shows that is always possible to perform sorting:

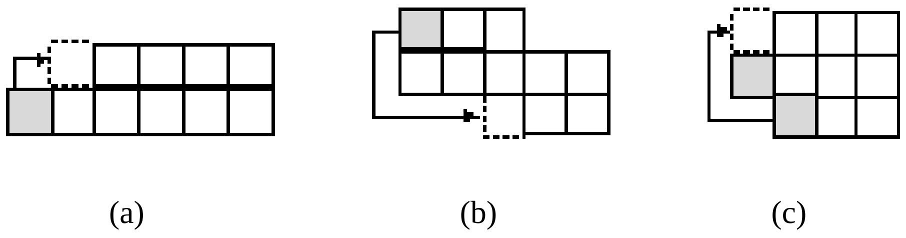


Figure 4.3: Steps of the sort phase. Modules next to move are greyed. Construction of the double row is shown in (a). In (b), SelectionSort is in progress. The final sorted order is assembled in (c) by merging the two sorted rows into a third.

**Lemma 1.** *An intermediate configuration with arbitrary module ordering can be reconfigured into a configuration with the same shape and a designated module ordering.*

*Proof.* Consider the specification of Algorithm 3. The double row can be assembled without disconnecting the structure since modules always move from the end of the top row, which can not be an articulation point since it has only one neighbor. The selection sort step maintains connectivity since all modules at all times are connected to a module in the lower row, and correctly assembles the sorted row since modules are chosen in order. The merge step is correct since we merge two sorted chains, and the three rows are always connected by the rightmost modules. Thus, the sorting algorithm is correct. □

### Grow

In the final phase of the algorithm, we build the goal configuration from the sorted intermediate configuration. Due to the sorting step, it is guaranteed that at any time, there exists a path from the tail module to its unique position in the goal configuration. We repeatedly find such a path and execute it for each module in the intermediate configuration. It is also possible to store the paths found while computing disassembly order to avoid replanning.

Pseudocode for this phase is listed in Algorithm 4. Line 3 finds a path from the tail to

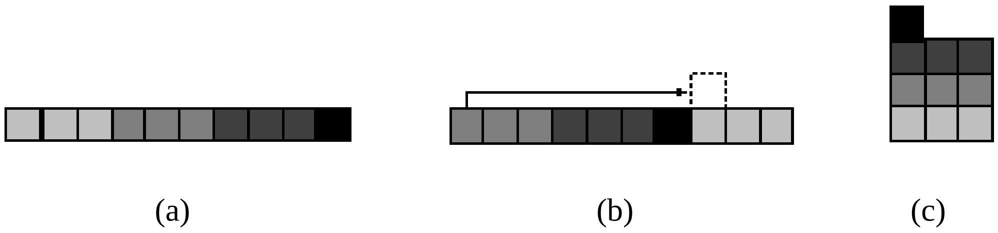


Figure 4.4: Steps of the grow phase. Shading indicates module type. In (a), a path is found from the module at the left end of the intermediate structure to its final position in the goal configuration, shown in (b). This repeats until the complete goal shape is assembled in (c).

**Algorithm 4** Grow goal configuration.

```
1: Intermediate configuration $I$
2: while Reconfiguration is not complete do
3:     Find path $p$ from tail of I to goal position using BFS
4:     Execute path $p$
```

its position in the goal configuration using the same BFS technique described in the melt phase. A motion plan based on the returned path is computed in Line 4, and this process repeats for each module. See Figure 4.4 for an example of this step.

**Analysis**

To prove correctness, we first must show that there is always a mobile module available from the original configuration to be moved into the intermediate configuration during the Melt phase. For configurations without holes, this has already been proved by Kotay [54]. Here, we prove this property for configurations including holes.

**Lemma 2.** *During the Melt phase, in a configuration with one or more internal holes, there exists at least one mobile module such that this module is not part of the intermediate structure.*

*Proof.* If any surface module has only one neighbor, it is mobile because it cannot be an articulation point. Therefore, we only need to consider the case where all surface modules have two or more neighbors. For any module on the surface, at least two of these neighbors must lie on a cycle surrounding a hole. Otherwise, the hole could not be completely surrounded by modules.

We consider a subset of surface modules and prove that at least one of these is mobile. We say a module is *extreme* if it is minimum or maximum in some direction. Suppose module $m$ is extreme, and is one of the rightmost modules. There can be no right neighbor, since $m$ is extreme in this direction. Either $m$ or one if its neighbors is mobile. To show this, we examine each of $m$'s neighbors in turn. Suppose $m$ has a top neighbor, $m'$. Module $m'$ is on the surface, since it too is extreme. By our initial assumptions, $m'$ has at least two neighbors and is connected to the structure other than through $m$. Therefore, $m'$ will not be disconnected by the removal of $m$. Aside from the left neighbor, the other cases are symmetric.

Now refer to Figure 4.5. If $m$ has no left neighbor, it must be mobile. If $m$ does have a left neighbor, this left neighbor is either connected to one of $m$'s other neighbors or not. In the first case, $m$ is mobile. Otherwise, all $m$'s remaining neighbors have no left neighbor, and therefore must be mobile by the same analysis applied to $m$. $\square$

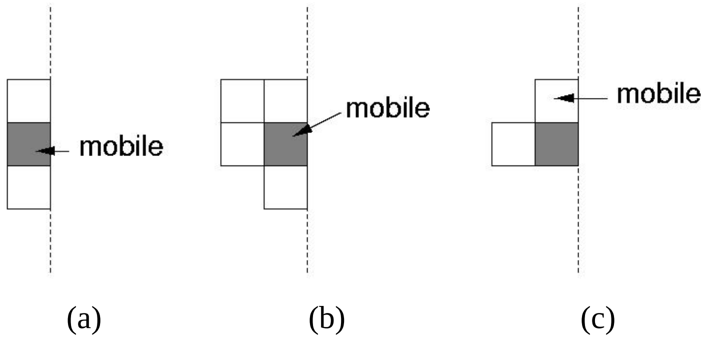


Figure 4.5: Illustration of cases described in Lemma 2. Module $m$ is shown shaded in all cases. The dotted line signifies that $m$ is extreme; there are no other modules to the right. In (a), $m$ has no left neighbor. In (b), $m$'s left neighbor is connected to the rest of the structure. In (c), $m$'s top neighbor has no left neighbor, and also is extreme.

We now show that MeltSortGrow is correct and complete. We also show the running time as claimed.

**Theorem 1.** *The algorithm MeltSortGrow computes a feasible reconfiguration plan of length $O(n^2)$ for all start and goal configurations in $O(n^2)$ time, where $n$ is the number of modules in the system.*

*Proof.* By the specification of Algorithm 4, each position in the goal configuration is filled only by a module of appropriate type. Therefore, the goal configuration is assembled correctly. It remains to prove completeness. We will show that any start configuration can be reconfigured into the intermediate configuration, and that the intermediate configuration can be reconfigured into any goal configuration.

As shown by Lemma 2, there is always a mobile module available during the Melt step. By repeatedly relocating mobile modules, we can therefore form the intermediate configuration from any start configuration. This argument also applies to computing the (dis)assembly sequence, since the same melt procedure is used. The intermediate configuration can be sorted by the assembly sequence due to Lemma 1. Now consider the module at the left end of the intermediate configuration, the tail. The tail is clearly mobile, and can move to any position on the surface of the structure without disconnection. Due to the assembly order, the destination position of the tail in the goal is unfilled and is reachable, and therefore the tail can successfully be relocated. Continuing in this way, we can relocate all modules in the intermediate structure and the goal configuration is assembled. Therefore MeltSortGrow is correct and complete for all start and goal configurations.

The running time is easy to see as each of the algorithm's three phases requires $O(n^2)$ time. First, one step of the melt phase requires $O(n)$ time for articulation point finding and $O(n)$ time for BFS. For $n$ modules, we have $O(n^2)$ time for the Melt phase in total. Sorting requires $O(n^2)$ time for computing the sort order, $O(n^2)$ time for selection sort, and $O(n)$ for merging. Finally the grow phase performs BFS $O(n)$ times or $O(n^2)$ total. Overall, the algorithm requires $O(n^2 + n^2 + n^2 + n + n^2) = O(4n^2) = O(n^2)$ time. No more than one primitive move is generated during each time step, and the sort order computation generates no moves, so the length of the resulting path is $O(3n^2) = O(n^2)$. We cannot lower the bound on the path length traveled by individual modules since the intermediate structure forces path lengths to increase towards $n$ with every module melted.

$\square$

### 4.1.2 Decentralized MeltSortGrow

In this section, we extend centralized MeltSortGrow to run in a decentralized manner. The main approach is to retain the overall structure of the algorithm, but replace centralized computation with message-passing as necessary. We assume that at the beginning, a single module receives a message to start running the algorithm. Further, we assume that all modules have a copy of the goal configuration in their local memories. If only one module knows the goal shape, then it can propagate this data to the rest of the system. This assumption appears to involve a significant communications cost, but this cost is only paid once, before the algorithm starts. Also, the problem of actually determining the desired goal configuration is still open. Recent approaches involve goal representations whose size is dependent on granularity [91]. We also begin to address inexact goal representations later in this chapter (Section 4.4). Reducing the size of the representation decreases the communication cost of propagating this information throughout the system. In any case, the planning-based approach we adopt necessitates this assumption because all modules within a class can substitute for each other and therefore need access to configuration information for all modules in that class. With neighbor-to-neighbor communication only, there is no way to selectively transmit pieces of the goal configuration without the potential for necessarily routing this information throughout the whole system.

### Melt

To begin algorithm execution, a single module, $m_{start}$, receives the initial message. We find the root by propagating a message depth-first style from $m_{start}$ that computes a relative lattice position for each module. We use depth-first search (DFS) since it is easier to distribute than BFS. Starting at $m_{start}$ $(0, 0, 0)$, each module computes positions for its children and passes this information in a message. In this way, the leftmost coordinate can be returned by comparing the return values of a module's children. Eventually $m_{start}$ will receive the answer and propagate this information to the rest of the system.

The root is the initial tail. At any time, there exists exactly one tail. To find a mobile module, the tail initiates a distributed articulation-point labeling algorithm. This is the same as the centralized version except recursive calls are replaced by message-passing to children. The tail then initiates distributed DFS to find the first non-articulation point, which then follows the path back to the tail and becomes the new tail. This ends when DFS fails. Alternatively, iterative-deepening search can be implemented if shortest paths are desired. Centralized MeltSortGrow uses BFS for this purpose. Pseudocode is shown in Algorithm 5.

### Sort

The next step is to sort the intermediate structure. We first need to determine the sort order by virtually disassembling the goal configuration and then physically sorting the modules. Disassembly in a decentralized manner can be handled in different ways, but the simplest solution is for each module to perform the computation itself to discover its own order. This solution is acceptable due to our assumption that all modules know the

**Algorithm 5** Distributed Melt. Pseudocode for single module.

```
State:
articulationPoint, am I an articulation point

Messages:
start, sent to exactly one module to begin algorithm
   Action: determine root, root executes meltOneModule()
labelArticulationPoints, labels articulation point modules
   Action: DFS-send(labelArticulationPoints), setting articulationPoint as result
findMobileModule, search to find non-articulation point
   Action: if I am mobile, follow path to tail and execute meltOneModule(), else continue
   message propagation according to DFS-send procedure

Procedures:
meltOneModule()
   DFS-send labelArticulationPoints
   DFS-send findMobileModule
   if result is false, signal start of sort phase
DFS-send(message)
   send message to first child, wait for response
   repeat for all children and compute result
   send result in return message to parent
```

goal configuration, and each module uses the same deterministic algorithm. This adds a factor of $n$ to the total computation cost, but occurs in parallel. Next, the single chain must fold in half to form a double row. The tail module initiates this by moving around to its final position below the root. Other modules follow and stop when all top row modules have lower neighbors. The last module in the top row signals its row to begin sorting, using distributed BubbleSort. When one pair has finished their comparison, they signal the next pair. This happens back and forth down the row until no more swaps are made. Then the top row signals the bottom row to performs the same sort operation. The two then merge to complete the intermediate configuration. See pseudocode in Algorithm 6.

This specification handles the case when all modules are uniquely identified. The modification to remove this assumption is simple, but requires extra memory. Instead of computing a unique sort position, the modules keep an array of all positions for their class

**Algorithm 6** Distributed Sort. Pseudocode for single module.

```
State:
goal, goal configuration
sortLabel, my order in assembly sequence
swapCount, counter to detect bubble sort termination

Messages:
sort, sent to start sort phase
    Action: propagate sort, execute computeSortOrder(), execute formDoubleRow()
bubbleSort, sent to do swap test
    Action: execute handleBubbleSort()
bubbleSortDone, sent to signal bubble sort termination
    Action: if top row, send bubbleSort to bottom row. if bottom row, execute merge()

Procedures:
computeSortOrder()
    disassemble goal to determine sortLabel
formDoubleRow()
    move around structure to form double row
    if I am last module, execute bubbleSort()
bubbleSort()
    compare sortLabel with neighbor and swap if necessary
    increment swapCount if swapped
    send bubbleSort(swapCount) to neighbor
handleBubbleSort()
    if swapCount=0 then propagate bubbleSortDone
    otherwise if I am at the end of the line, send bubbleSort in opposite direction
merge()
    tails of sorted rows propagate sortLabel. tail with minimum moves to final row and signals
    next tails. when complete, grow phase begins.
```

ID. Then when the single chain is assembled, the unique position is resolved by passing a counter down the chain.

### Grow

The grow phase reconfigures the intermediate configuration into the final shape. The basic step is that the tail module finds a path, executes it, and signals the new tail. Path planning is done using distributed DFS. When the root module becomes the tail, the algorithm terminates. See Algorithm 7 for pseudocode.

**Algorithm 7** Distributed Grow. Pseudocode for single module.

```
State:
goalPosition, my position in goal configuration
root, am I the root

Messages:
grow, sent to signal start of grow phase
   Action: if I am the current tail, execute moveToGoal()
nextModule, sent to grow next module
   Action: if I am the current tail, and I am root, signal completion. else execute moveToGoal()
findPath(goalPosition), sent to find path
   Action: if I am adjacent to goalPosition, return true. else DFS-send(findPath(goalPosition))

Procedures:
moveToGoal()
   DFS-send findPath(goalPosition)
   follow path
   propagate nextModule
DFS-send()
   specified in Algorithm 5
```

## Analysis

Theorem 1 proved correctness and completeness for the centralized algorithm. Here, we show that the decentralized version is also correct and complete using a similar argument.

**Theorem 2.** *The decentralized version of MeltSortGrow computes a feasible reconfiguration plan of length $O(n^2)$ for all start and goal configurations in $O(n^2)$ time, where $n$ is the number of modules in the system.*

*Proof.* In the melt phase, a module trajectory path is found by searching free positions adjacent to modules. Because the system is connected, we visit all free spaces eventually and a path is guaranteed to be found. Continuing in this way for all modules, the intermediate structure is assembled. Sorting is performed in the style of bubble-sort until both halves of the double row are sorted. Because each half is sorted sequentially, connectivity is maintained. Finally, the grow phase assembles the goal shape using search techniques as in the melt phase.

The running time bound is shown by observing that each atomic step in the centralized version is replaced by a message in the decentralized version. We assume a constant amount of work is performed in processing a message, so we equate time steps with messages. Path searching is done using DFS, which takes $O(n)$ messages to find a path. Therefore, melt and grow take $O(n^2)$ messages each. BubbleSort also generates $O(n^2)$ messages. Since sort order is computed by each module, this adds a factor of $O(n^3)$ time steps. The overall running time is thus $O(3n^2 + n^3) = O(n^3)$, with $O(n^2)$ moves generated as in the centralized version. $\square$

### 4.1.3 Example

A detailed centrally-planned example is shown in Figure 4.6. Here the start and goal configurations are both box shapes of 18 modules with unique types. However, the type specification of the goal configuration is such that no module in the start configuration is in a valid position, i.e. each module must move to a new position to correctly form the goal configuration.

In (a), the root module is chosen. The next picture shows the line being formed to the left of the root. The Melt phase is then completed in (c) as all modules in the initial configuration have been moved.

The next three illustrations show the Sort phase. In (d), the line configuration has been broken in half and doubled-up on itself. This enables each half row to be sorted, shown in (e). Finally, (f) shows the two sorted rows being merged back into a single line.

At this point, the tail module can always find a path to its final position in the goal shape. Since this example has unique types, each module has exactly one valid position in the goal. Pictures (g) and (h) illustrate the Grow phase, and the final configuration is completed in (i).

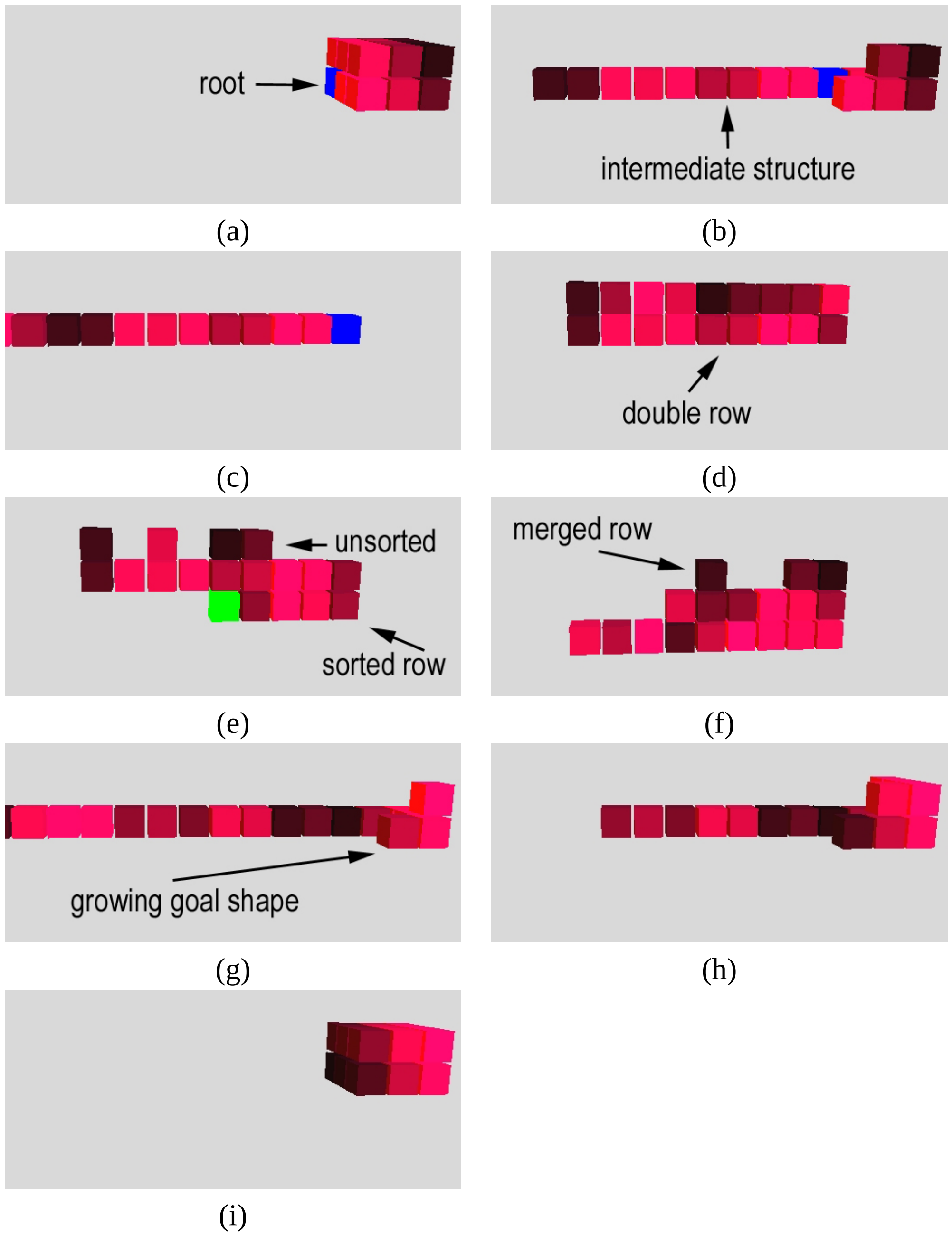


Figure 4.6: Screenshots from MeltSortGrow implementation in SRSim simulation. Module shading indicates unique labels; here all modules are unique. Cube-shape reconfigures into another cube shape with order of modules reversed.

### 4.1.4 Discussion

The significance of this algorithm is that it is the first to address the heterogeneous reconfiguration problem. It is surprising that the running time is equivalent to the fastest algorithms for homogeneous reconfiguration as well, since this contradicts the intuitive notion that module interchangeability makes reconfiguration easier. One explanation is that MeltSortGrow exploits a special case of the Warehouse Problem, where free space is relatively unconstrained. Therefore, free space appears to be the critical resource in contention, as the $O(n^2)$ worst-case running time matches the $\Omega(n^2)$ reconfiguration lower bound and so is asymptotically optimal.

Practically speaking, choosing a line as the intermediate structure is not the best alternative. It is simple to explain and analyze, but other structures would be more useful in practice. For example, a square or other planar structure would be just as easy to construct and sort. In fact, any intermediate structure amenable to sorting would suffice. This approach is particularly beneficial in cases where the robot is primarily homogeneous – for example, where a few specialized sensor modules are added to an otherwise homogeneous robot. In this case, efficient sorting could be used to easily adapt to the system's limited heterogeneity.

In comparing the centralized and decentralized versions, the decentralized version is necessary since we generally assume that a centralized controller is neither available nor desirable. The decentralized version does incur increased cost of messaging overhead, but this is primarily due to broadcast messages. Although the Sliding Cube model only includes point-to-point communication, real systems often have some sort of broadcast functionality. Utilizing this reduces the communications burden of the decentralized version. Furthermore, time spent in actuation typically dwarfs the time spent in communication, so actual running time should not be appreciably greater even in the absence of any broadcast facilities.

Although the free space required by this algorithm is large, subsequent algorithms in this section reduce this requirement significantly. The next obvious theoretical question, then, is whether the length of the plan can be reduced. The $\Omega(n^2)$ lower bound holds for general reconfiguration, but perhaps other special cases can be identified that would provide homogeneous systems an advantage.

A major area of future work for this style of algorithm lies in considering dynamics of the system during reconfiguration. Throughout this work, we generally ignore issues of the robot toppling over and basically assume the robot is floating in space. There is good potential for addressing this issue through refinement of two pieces of the algorithm: choice of modules during melt and order of constructing the intermediate shape. Only very sparse, stringy shapes contain few mobile modules. Most shapes have many choices, and here we simply choose the first encountered by BFS or DFS. A more sophisticated search procedure could be used instead. Also, the intermediate structure could be constructed such that it forms supporting structure for modules above. Addressing the issue of dynamics is, of course, critical for implementing this algorithmic approach in a real 3D system.

## 4.2 In-Place Reconfiguration

Although unlimited free space is useful for planning fast reconfigurations, the case where free space is limited also admits a polynomial-time solution. *TunnelSort* is an algorithm we developed that solves the heterogeneous reconfiguration problem in-place. Due to these constraints, algorithms that utilize intermediate goal configurations, such as MeltSortGrow, are no longer feasible. Instead, TunnelSort builds the goal configuration directly in a greedy manner, without regard to type. It then relocates modules within the goal configuration to correctly place modules according to type. This can be thought of as “sorting” modules in the goal configuration. Like MeltSortGrow, TunnelSort takes as input a start configuration and a goal configuration and outputs a reconfiguration plan. The worst-case running time and number of moves are both $O(n^2)$, which is equivalent to MeltSortGrow. This is asymptotically optimal for the reconfiguration problem.

Free space is constrained in the following way. Allowable module positions include only the union of the start and goal configurations (with alignment assumed to be given) plus empty lattice positions immediately adjacent to modules on the surface of this union. We call this extra space the *crust*, since it can be thought of as growing the union by one unit. See Figure 4.7 for an example.

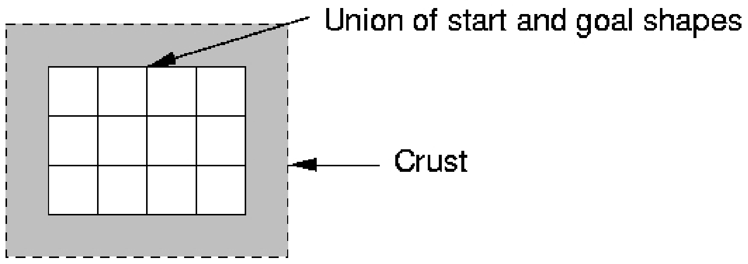


Figure 4.7: Illustration of crust around the union of start and goal shapes. This is the region of free space available to an in-place reconfiguration algorithm.

For two non-identical configurations, inconsistencies at a specific lattice position fall into two cases: 1) there is a module in the goal and not in the start, or 2) there is a module in both but they have different types. Heterogeneous reconfiguration can thus be divided into two phases: the first forms the goal shape regardless of type, and the second adjusts the goal shape to ensure type consistency. Decomposing the problem in this way is helpful since the first phase is purely homogeneous and can therefore be solved using existing greedy algorithms as described by Yim [99] or Kotay [54], in the case of configurations with no holes. The case of configurations *with* holes requires more planning; we address this in the following section. For now, we focus on the challenge of the second phase,

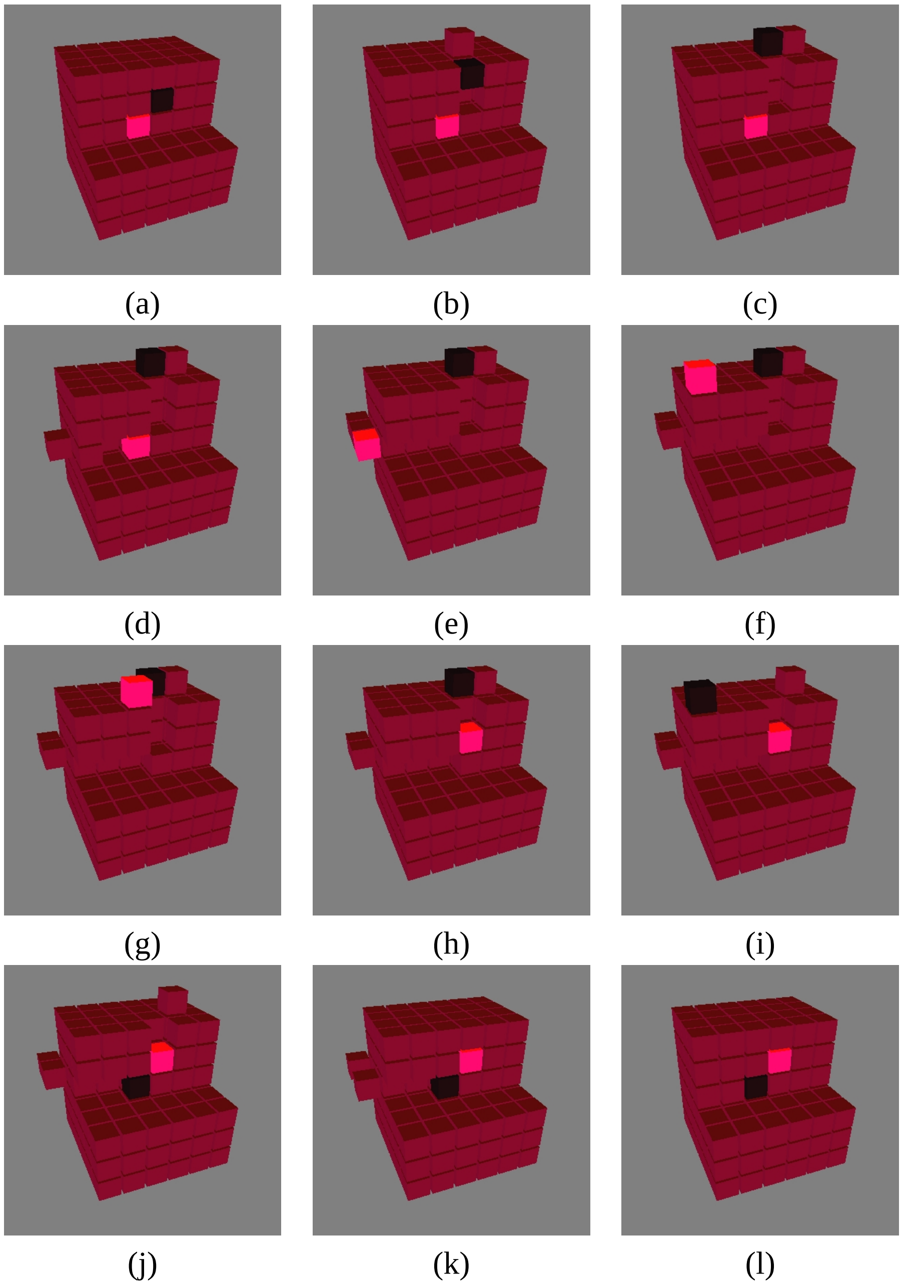


Figure 4.8: Example of TunnelSort algorithm. Some modules not shown. Initial configuration with two incorrectly placed modules is shown in (a). Modules are unlocked in (b)–(e), and swapped in (f)–(k), resulting in the final configuration (l).

where the modules must be sorted by type. TunnelSort solves this phase of the problem.

Our approach is to repeatedly swap modules until all type requirements in the goal configuration are satisfied. See Figure 4.8 for an example. The challenge is to "unlock" modules from the structure while both maintaining global connectivity and avoiding backtracking caused by displacing correctly positioned modules. For example, to access a module buried deeply in the structure, it might be necessary to temporarily displace a large number of other modules. Or, consider rearranging a sparse shape such as a line. Removing a module in the interior clearly divides the line. We address these challenges as follows. To swap modules, we will unlock each by creating a path, or tunnel, to the crust. Displaced modules are temporarily stored in the crust, while the two desired modules exchange positions. Then the remaining modules reverse their motions. This idea is similar in spirit to the Warehouse Problem solution given by [87]. Throughout, the main technique is to find or construct local features that assure global connectivity.

An initial assumption we make is that the relative positions, or overlap, of the two shapes is given. This is an assumption made by other algorithms and is valid since the overlap can easily be computed according to characteristics of the underlying task.

In this section, we present the TunnelSort algorithm. We begin with basic techniques for moving modules through the structure. We then describe the algorithm itself, first in centralized form for clarity as Algorithm 8, then as a decentralized implementation in Algorithm 9. We analyze each version, and conclude the section with an example run of the algorithm and a discussion.

### 4.2.1 Module Relocation via Tunneling

For module-level trajectory planning, there are generally two strategies: motion over the surface, and motion through the volume of the structure. The Sliding-Cube model includes motion primitives for direct surface motion. We would like to implement motion through the structure as well. To do this, our approach is to create a path for a given module by temporarily displacing intervening modules. We refer to such a path as a *tunnel*.

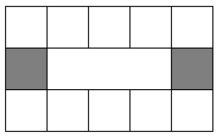


Figure 4.9: Challenge of removing modules as a group. No module is an articulation point. Either shaded module can be moved individually, but moving both causes disconnection.

The main challenge in creating a tunnel is preventing disconnection. For a single module, mobility can be tested by labeling articulation points in the connectivity graph. If a module is a non-articulation point, it can be moved without causing disconnection. However, we need to displace a group of modules. A method for identifying a mobile *group* of modules is more difficult. Consider an example configuration as shown in Figure 4.9. Here, no module is an articulation point. Either shaded module may safely be removed, but not both. Other graph theoretic techniques can he helpful in identifying a sequence of modules to be moved, but it is unclear how to extend them to be useful in path planning. For example, consider any spanning tree imposed over the connectivity graph. The leaves of the spanning tree, by definition, can be removed without disconnecting the graph. Therefore, any subtree in a spanning tree forms a set of modules that can be removed as a group, by "peeling" the leaves from the subtree. Another example by Ghrist [1] extends the idea of configuration space to support searching for sets of physically independent primitive moves. Unfortunately, this does not help in finding a path in polynomial time.

Rather, we would like to identify local properties that guarantee the global property of connectivity. First, observe that a partial cut does not disconnect the module connectivity graph:

**Lemma 3.** *Nodes along a straight-line path through a module connectivity graph can be removed without disconnecting the graph if each node along the path has both side* *neighbor positions occupied, and the side neighbors of the node at one end of the path are connected to each other.*

*Proof.* The construction insures that there is a line of modules on either side of the partial cut. Choose any removed node. All remaining neighbors stay connected since there exists a path around the cut that connects them. □

Therefore, as long as the sides of the tunnel are connected, modules along the tunnel path, *tunnel modules*, can safely be removed. The total number of motions sufficient to displace the tunnel modules is equal to the squared length of the tunnel. This can be seen by noting that if each module follows its neighbor, the deepest module traverses the entire length. So for a tunnel of length $l$, we have $l$ modules making $l$ moves or $l^2$ overall.

Interestingly, if we add holes to the tunnel path, the total number of moves does not increase. The total is still equal to the number of modules in the path, not the path length in lattice-distance. This is because some modules maybe be "stored" in the holes and do not need to traverse the entire path. The following lemma formalizes this notion.

**Lemma 4 (Amount of work done in creating a tunnel).** *For a straight line path with $l$ (not necessarily connected) modules, the amount of work required to tunnel through this path is $O(l^2)$.*

*Proof.* We have two cases. If the modules are connected, then each module makes $l$ moves to reach the crust and the total work is $l^2$. Otherwise, the tunnel is divided into segments and holes. The motion of modules adjacent to a hole is bounded by the length of the hole, since these modules remain in the hole and do not travel the entire length of the tunnel. We can bound the distance traveled by one of these modules by $h$, the sum of the lengths of all holes. There are $h$ such modules, so the total number of moves for these is $O(h^2)$. The remaining $l - h$ modules travel, at most, the entire length of the tunnel, $l + h$. The overall bound for the case with holes is thus $O(h^2 + (l - h)(l + h)) = O(h^2 + l^2 + lh - hl - h^2) = O(l^2)$. $\square$

If we assume that a tunnel must be created to every module in the structure as part of a reconfiguration algorithm, it is necessary to count the total number of moves in all tunnels. We can accomplish this by bounding the length of all tunnel paths. First, consider a simple case with no holes and paths consisting of two straight-line segments.

**Lemma 5 (Length of a two-segment tunnel path).** *There exists a tunnel path from any module to the surface of a configuration with no holes such that the path can be decomposed into two perpendicular straight-line segments where the length of each segment is $O(\sqrt{n})$.*

*Proof.* Assume not. Then the second segment can be swept up, creating a rectangle with sides $< O(\sqrt{n})$, contradicting the assumption that the total number of module is $n$. See Figure 4.10 for an illustration. $\square$

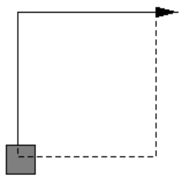


Figure 4.10: For a tunnel path consisting of two segments, the maximum required length of each segment is $\sqrt{n}$, where $n$ is the total number of modules. The shaded square is a module, and the dark line denotes a tunnel path. If either segment is longer, the rectangular region completed by the dotted line must be greater than $n$.

Now we show that the sum of the lengths of all straight-line, single-segment paths is also bounded by $O(n^2)$:

**Lemma 6 (Amortized analysis of work done in unlocking).** *For a planar robot with $n$ modules, the sum of the squares of the minimum distance to the surface over all modules is $O(n^2)$.*

*Proof.* We will show $\sum_{i=1}^{n}(\min(r_i, c_i))^2 = O(n^2)$, where $r_i$ and $c_i$ are number of modules in the row and column, respectively, that contain module $i$. We define the set $m_{long}$ as all modules $m_j$ such that $\min(r_j, c_j) > \sqrt{n}$, and similarly, $m_{short}$ as modules $m_k$ such that $\min(r_k, c_k) <= \sqrt{n}$. The total contributed to the sum by $m_{short}$, by definition, is bounded by $n(\sqrt{n})^2 = O(n^2)$.

Now consider the elements of $m_{long}$. If there are $l$ such elements, and $m_{max}$ is the maximum value of $\min(r_j, c_j)$ for all $m_j$, then $\sum_{m_{long}}(\min(r_j, c_j))^2 < lm_{max}^2$. But, we know that since there are $n$ modules in total, $m_{max}\sqrt{l} = O(n)$. See Figure 4.11. This is true because the length of $m_{max}$ is bounded by the total number of modules, minus the size of $m_{long}$, plus the maximum number of elements of $m_{long}$ that lie along the column of max length, or $n - l + \sqrt{n}$. The last term is $\sqrt{n}$ because all elements in $m_{long}$ have row and column lengths at least $\sqrt{n}$, and there are $n$ modules in total. The maximum value of $m_{max}\sqrt{l} = (n - l + \sqrt{n})\sqrt{l}$ occurs where $(l = n)$, so $m_{max}\sqrt{l} = O(n)$, and so $lm_{max}^2 = l(\frac{O(n)}{\sqrt{l}})^2 = O(n^2)$. Together, we have $\sum_{i=1}^{n}(\min(r_i, c_i))^2 = O(n^2) + O(n^2) = O(n^2)$. □

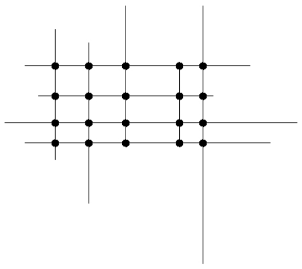


Figure 4.11: Illustration of relationship between long segments and the number of modules with high cost of tunneling. Line segments represent lines of contiguous modules in a configuration; dark circles are modules at intersection points. The number of line segments is bounded by the square root of the number of intersections, which in turn is bounded by the total number of modules.

This 2D analysis also applies to the 3D case. Any tunnel in a 3D structure lies in three

2D-planes. We showed that the total cost due to tunneling is bounded by the total number of modules in the 2D configuration. The size of any 2D slice of a 3D configuration is smaller than the size of the full 3D configuration. Therefore, the 2D upper bound shown in Lemma 6 is also an upper bound on the amortized number of moves in a 3D instance.

The analysis further extends to cases with holes:

**Lemma 7.** *For a planar robot with $n$ modules including holes, the sum of the squares of the minimum distance to the surface over all modules is $O(n^2)$.*

*Proof.* The proof of Lemma 6 proves this bound in the case with no holes. The summation counts only the number of modules in a row or column, not the lattice distance. Therefore, the proof of Lemma 6 also holds in the case of shapes with holes. □

So far we have not discussed the placement of modules displaced during tunneling. The crust is large enough to hold all tunnel modules, as shown by Lemma 8.

**Lemma 8 (Size of crust).** *A one-unit crust contains at least $4\sqrt{n}$ module positions.*

*Proof.* For a configuration with no holes and no limiting obstacles, the size of the crust is equal to the number of free lattice positions adjacent to modules. This number is minimized by the tightest packing of squares in two-dimensions. For an $n$-module robot, where $n = k^2$ for some integer $k$, the tightest packing is given by a square. This yields $4k = 4\sqrt{n}$ crust positions. Since this is a minimum, the size of the crust for any arbitrary configuration is $4\sqrt{n}$. An $n$-module shape with holes has perimeter at least as large as an $n-$module shape with no holes, so the bound holds for all connected configurations in two dimensions. □

The crust is actually large enough to hold modules from two tunnels simultaneously. Observe that the length of a tunnel is bounded by the diameter of the configuration. The

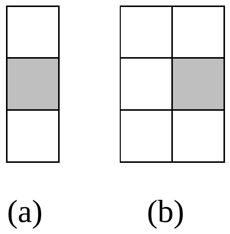


Figure 4.12: Example of bridging a module to preserve connectivity. In (a), the shaded module is to be removed, but it is an articulation point. We recruit extra modules to connect the shaded module's neighbors in (b). This temporary bridge structure allows the shaded module to be removed and swapped with another module elsewhere in the structure. Then, bridge modules can return to their original positions.

surface is at least twice as large as any diameter, so two tunnels may be created simultaneously without depleting storage positions.

Finally, we outline an algorithm for creating a tunnel. The first step is to build sufficient supporting structure around the deepest module in the tunnel, to ensure that it is not an articulation point and to connect the sides of the tunnel. We call this process *bridging*. To bridge a module, it is necessary to connect all neighbor modules not part of the tunnel. See Figure 4.12 for an example. Mobile modules in the structure are recruited for this task. Such modules are always available. Observe that if a lattice position adjacent to a module is free, then it forms the surface of a hole. This might be an internal hole, or else the crust. Since the surface forms a cycle in the connectivity graph, at least one module can be removed, followed by modules adjacent to it on the surface. Therefore the constant number of modules required for bridging are always available. After bridging is complete, the tunnel module adjacent to the crust follows an arbitrary path through the crust, and the other tunnel modules follow until the tunnel is complete.

## 4.2.2 Centralized TunnelSort

We present the algorithm in centralized fashion as Algorithm 8 in order to simply describe the algorithmic idea; Algorithm 9 shows the equivalent decentralized algorithm.

**Algorithm 8** Centralized TunnelSort.

```
1:  Reconfigure homogeneously to match goal shape
2:  Label crust
3:  while Reconfiguration is not complete do
4:      Choose modules $m_1, m_2$ where $m_2$'s type matches goal type at $m_1$'s position
5:      Find minimum-length straight-line path from $m_1$ to crust
6:      for each segment $s$ in path from $m_1$ to crust do
7:          if $s$ is tunnelable then
8:              Find path around $s$ or create bridge
9:          if $s$ was bridged then
10:             Move all modules in $s$ into adjacent free space
11:     Repeat with $m_2$
12:     Find path between $m_1$ and $m_2$ and swap
13:     Replace all other modules
```

At a high level, the outer loop (Line 3) chooses modules that are in invalid positions and swaps them. The choice of modules is done using a simple graph search. Swapping involves planning a path, creating tunnels, moving the modules, and returning all displaced modules to their original positions. See Figure 4.8, shown earlier in this section, for an example in simulation. We now go through these steps in detail.

Line 1 uses existing algorithms as noted earlier. At this point, the correct goal shape has been constructed and we can perform necessary preprocessing in Line 2. The purpose of the preprocessing step is to define the crust by labeling lattice positions that comprise it. This is straightforward in the centralized version. We simply build a data structure to store a collection of lattice positions. These are defined as follows. For a module $m_i$, the set of six adjacent lattice positions is referred to as $L_i$. For the set of all modules $M$, the set of all adjacent lattice positions $L = \bigcup_i L_i$. The crust consist of all adjacent lattice positions not occupied by another module, or $L - M$. This is easily implemented by iterating through the module list and adding or deleting items from the crust data structure as necessary.

In Line 3, the main loop commences. The choice of modules to swap (Line 4) can be done in any linear-time search. We iterate through the modules in arbitrary order and choose $m_1$ as the first module whose type does not match the goal, or $type\_of(m_1) \neq goal\_type(position\_of(m_1))$. The choice of $m_2$ is made similarly, with the added condition that $m_2$'s type matches the goal type we are looking for:

$type\_of(m_2) = goal\_type(position\_of(m_1))$. Note that since we always choose two modules whose type is invalid, the algorithm always makes progress by correctly placing at least one module. The final swap corrects the final two type errors.

Most of the complexity lies in the unlocking procedure, which begins in Line 5. The first step is to choose a path from $m_1$ to the crust. This path will form the tunnel allowing $m_1$ to access the surface of the structure. We find a minimum-length, straight-line path by searching out from $m_1$ in a straight line in all dimensions until we reach the crust. If there are no holes along this shortest path, we simply create a tunnel by moving the intervening modules into the crust, as shown in Figure 4.13(a). It might be necessary to bridge $m_1$ to initiate the tunnel. This case is handled as described earlier in Section 4.2.1. Otherwise, this path may be divided into a number of connected segments of modules. The resulting sequence of alternating segments and holes allows us to store modules in the segments in the subsequent holes instead of exclusively in the crust (Figure 4.13(b)). Considering a segment $s_i$ and hole $h_i$, if $s_i$ can be removed without disconnecting the structure, then we may relocate $s_i$ into $h_i$ until $h_i$ is filled except for free space along the tunnel we are attempting to create. Further modules in $s_i$ can then traverse this tunnel until they reach segment $s_{i+1}$, and will be handled during the next iteration.

If $s_i$ cannot be safely removed, however, we must either avoid the problem by simply moving $m_1$ around $s_i$ (Figure 4.13(c)), or else it must be possible to "bridge" the sides of $s_i$ to the rest of the structure and thus fall into the earlier case (Figure 4.13(d)). We can label the connected components that would be disconnected in the removal of $s_i$ by simply executing a depth-first search starting with modules adjacent to $s_i$. Then we find a mobile module on the perimeter of hole $h_i$, and move this module around the perimeter until it

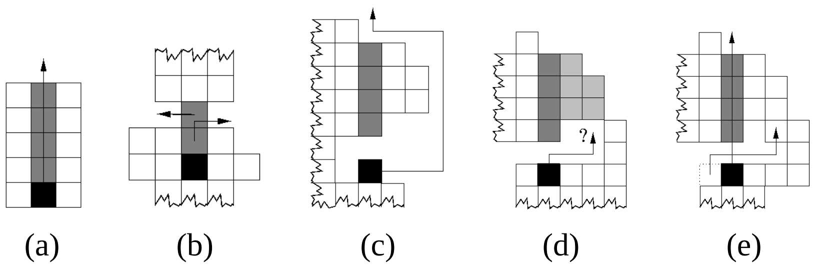


Figure 4.13: Tunneling through a segment. Module to be unlocked is shown in black; shading indicates the original tunnel path. The case with no holes is shown in (a). Shaded modules will be stored in the crust. If the path does contain holes, as in (b), shaded modules can be stored there. In (c), the tunnel segment cannot be removed but an alternate path exists. A similar case is shown in (d), but no such path exists. Rather, a bridging module allows the segment to be safely removed by connecting the lightly shaded modules to the rest of the structure, resulting in (e).

becomes adjacent to both a labeled and unlabeled modules. Such a position much exist since it was not possible to circumvent $s_i$ entirely. Repeating for all labeled connected components, it is now safe to tunnel through $s_i$.

Once a segment is identified as tunnelable, either without modification or through bridging, the tunneling proceeds in the same manner. The module closest to the crust makes a random move into the adjacent surface. It then continues to randomly move across this surface, with the remaining tunnel modules moving in follow-the-leader style. Care must be taken to not allow the string of modules to intersect itself, which would partition the crust into an unreachable region. This process terminates when the tunnel is complete. If the far endpoint of the segment is adjacent to a hole instead of the crust, then modules simply greedily move to positions on the surface of the hole until it is filled.

We repeat this procedure for each segment in the path, and thus $m_1$ can reach the crust. Note that since the modules follow one another along the tunnel, these motions can be executed in parallel for improved efficiency.

After $m_1$ and $m_2$ traverse the crust in Line 12, all other modules return to their original positions in Line 13. This is implemented by maintaining a stack of moves associated with each module. When it is time to reverse the tunnel, modules simply pop and execute motions from their stack.

**Analysis**

**Theorem 3 (Correctness of centralized TunnelSort).** *Algorithm 8 produces a reconfiguration plan that transforms any given start configuration into any given goal configuration.*

*Proof.* When the algorithm begins, the current configuration matches the goal shape. For each color $c$, the number of $c$-colored modules equals the number of $c$-colored goal positions. Therefore, suitable $m_1, m_2$ always exist in Line 4.

The unlocking procedure is correct based on case descriptions in Section 4.2.2. There is sufficient space in the crust for temporarily storing modules since the size of the surface is greater than the length of any path. A path between $m_1$ and $m_2$ through the crust always exists since temporary modules never form a path that intersects itself. All free spaces in the crust are thus reachable from each other. After swapping, the original configuration is restored except for the position reversal of $m_1$ and $m_2$. Each iteration therefore correctly positions at least one module and the loop eventually terminates with the robot in the correct final configuration. □

It remains to show the time bound of $O(n^2)$. First, the time spent in unlocking a given module $m_i$ is equal to the sum of distances traveled by modules along the tunnel path, by $m_i$ itself, and by modules used in creating bridges. The number of moves required to create bridges plus the path length of $m_i$ is $O(n + n) = O(n)$. The work done in moving all other modules is proportional to the squared length of the tunnel. Since we always choose the straight line path of minimum length, we can amortize the cost of all tunnels to $O(n^2)$. The total work done in unlocking is therefore $n * O(n) + O(n^2) = O(n^2)$.

Total time overall is $n$ iterations of $O(n)$ time for searching and swapping plus $O(n^2)$ amortized time for unlocking, or $O(n^2)$ overall.

We can give a tighter analysis by parameterizing the bound in terms of the actual number of modules that need to be swapped, $m$, the maximum path length traveled by a module during swapping, $p$, and the maximum tunnel length $t$. We first examine the number of time steps performed during planning. To unlock a module, we spend $O(n)$ steps to find a tunnel path. There are a maximum of five bridge modules, so creating and later destroying these bridges adds $O(10n)$ time steps. Then, moving tunnel modules takes $O(t^2)$ steps. Finding a path and moving the module to its final position takes $O(n)$ steps. Finally, returning tunnel modules to their original locations takes another $O(t^2)$ steps. So we have $O(12n + 2t^2)$ time for unlocking. There are two unlocking operations per swap, plus $O(n)$ time for finding modules to swap. That gives $O(25n + 4t^2)$ time steps per swap, or $O(25mn + 4mt^2)$ time total. But, we know that the sum of all tunnel lengths is $O(n^2)$. That gives a bound of $O(25mn + \min(4mt^2, 4n^2))$ time steps overall.

The number of moves in unlocking is $O(10p)$ for bridging, $O(2t^2)$ for tunnel modules, and $O(p)$ to swap. With two unlock operations per swap, that gives $O(22p + 4t^2)$ moves, or $O(22mp + 4mt^2)$ moves total. With the further bound on total tunnel lengths, we have $O(22mp + \min(4mt^2, 4n^2))$ in all.

### 4.2.3 Decentralized TunnelSort

The decentralized version of TunnelSort is listed as Algorithm 9. Our general approach is to dynamically choose modules as controllers over local operations, and to use message-passing for global synchronization. Most often, message-passing is implemented such that a message traverses modules sequentially. For example, with procedure DFS-send(), the message arrives at modules in the robot in the same order in which depth-first search would visit nodes in the module connectivity graph.

The algorithm begins as message *type_check* is sent to any module. This initial message can originate from outside the system or from another module using this algorithm as a subroutine. When a module receives *type_check*, it initiates a swap procedure if necessary and then resends *type_check*. The algorithm terminates after all modules have received *type_check*. This implements the outer loop of the centralized algorithm.

Control over the swap procedure is shared between the two modules, $m_1$ and $m_2$, that exchange positions. Module $m_1$ unlocks itself by locally creating a tunnel in procedure unlock(), and then searches for $m_2$ by sending the *swap* message. Module $m_2$ similarly unlocks itself, moves into position, and signals $m_1$. Then $m_1$ is free to move to its final position. All remaining modules return to their original positions via the *return* message and the swap is complete.

## Analysis

Correctness of decentralized TunnelSort can be proved by showing that the decentralized version performs steps equivalent to centralized TunnelSort. This is formalized in the following theorem.

**Theorem 4 (Correctness of decentralized TunnelSort).** *Algorithm 9 produces a reconfiguration plan that transforms any given start configuration into any given goal configuration.*

*Proof.* First we show that the high-level synchronization places all modules in valid positions and terminates. Initially, we have $M = M_{valid} \cup M_{invalid}$, where $M$ is the complete set of modules, $M_{valid}$ is the set of modules whose types match the goal type, and $M_{invalid}$ is the set of modules whose types conflict with their respective goal types. Procedure DFS-send() propagates a message through the system according to a post-order traversal

**Algorithm 9** Decentralized TunnelSort. Algorithm begins with message *typecheck* sent to any module. Each module executes identical code.

```
State:
type, my type label
goal_type, type in goal configuration at my current position

Messages:
type_check, sent to search for modules whose type conflicts with goal type
   Action: If type = goal_type, DFS-send(type_check). Else execute unlock(), send
   swap(goal_type, my position)
swap(t, position), sent to search for a module of requested type t
   Action: If type = t, handleSwap(position). Else DFS-send(swap(t, position))
swap_done(position), sent to synchronize swap
   Action: Find path to position, follow path, DFS-send return, send type_check
tunnel, sent to move modules out of the way
   Action: Send tunnel in same direction. Move to side of path, recording motions
return, sent to return displaced modules to original positions
   Action: While not in original position, wait for next position to be free and move there.
   Return true

Procedures:
handleSwap(p)
   unlock()
   Find path and move to position p
   send swap_done(my_original_position) to $m_1$
unlock()
   search for minimum length straight line path
   while I am not in crust do
      if current segment not tunnelable then
         if path exists around segment then
            follow path and continue
         else
            request mobile modules to bridge sides of tunnel to rest of structure
      send tunnel to segment
      move through tunnel to next segment
DFS-send(message)
   send message to first child, wait for response
   repeat for all children and compute result
   send result in return message to parent
```

of a spanning tree imposed on the module connectivity graph, as discussed in Chapter 3, Section 3.4. The message *type_check* is propagated by DFS-send() and therefore arrives at modules in sequential order. Propagation stops when *type_check* arrives at a module $m_1$ in $M_{invalid}$. The swap procedure moves $m_1$ into a valid position and some other module $m_2 \in M_{invalid}$ to a different, possibly still invalid, position. Therefore, the swap procedure results in decreasing the size of $M_{invalid}$ by at least one. Message *type_check* is then resent. Each iteration decreases the size of $M_{invalid}$ and increases the size of $M_{valid}$ until $M_{valid} = M$ and the algorithm terminates.

We now show the correctness of the swap procedure. Message *swap* is propagated until it arrives at a module $m_2$ with invalid type, so we have $m_1, m_2 \in M_{invalid}$. Module $m_2$ moves itself to the crust via the tunnel procedure, and signals $m_1$, which also creates a tunnel and moves itself to the crust. Since both $m_1$ and $m_2$ are adjacent to the same surface, a path between them exists. Module $m_1$ finds such a path and moves to the original position of $m_2$. It then sends a message to $m_2$, which similarly moves to the original position of $m_1$. Message *return* then causes all displaced modules to return to their original positions. The tunneling procedure operates locally according the case analysis described earlier. $\square$

To show the time bound, we note that the number of module movements is equal to that of the centralized version. The tunnel paths are straight lines, which Lemma 6 showed require $O(n^2)$ moves in total, for an $n$-module robot. The swap paths are bounded by the size of the robot, and so these also contribute $O(n^2)$ total moves, making $O(n^2)+O(n^2) = O(n^2)$ overall. We now must show that the addition message-passing overhead does not asymptotically increase the number of time steps. To do this, we will count messages by type. First, *type_check* is sent $O(n)$ times, or $O(n^2)$ total. Each swap consists of a constant number of synchronization messages, each arriving at $O(n)$ modules in the worst case (simulated broadcast), plus added messages for tunneling. Tunnel messages are passed along the tunnel, and the number is thus proportional to the tunnel length, which we showed earlier to be $O(n^2)$ overall. Therefore the total number of messages done in swapping is $O(n^2)$, and the total number of messages for the entire algorithm is $O(n^2) + O(n^2) = O(n^2)$. Since the decentralized and centralized versions operate equivalently, the parameterized analysis we showed for the centralized case is the same in this case.

### 4.2.4 Example

An example run of TunnelSort is given in Figure 4.14. This is an example of a difficult problem instance, a line configuration. In this section, lettered Frames all refer to Figure 4.14.

The number of mobile modules in a chain configuration is exactly two. The end modules can be moved, but all others are articulation points. Therefore, reconfiguring to or from a chain is difficult since most modules are immobile. This example shows the solution TunnelSort planned for reversing the order of a line of modules. Frame (a) shows the initial configuration, and the final frame shows the goal configuration almost complete. Module shading indicates type, and all module types are unique. Note that the order of module shading in the final frame is a reverse of the initial frame.

Since the goal is to reverse the module order, swaps must happen between outer modules, progressing inwards. The first swap, which exchanges the outer-most modules of the chain, is simple. Frame (b) shows the modules en route to their final positions, and Frame (c) shows the result. Since inner modules are immobile, they must be bridged. Frame (d) shows such a bridge. Frames (e) and (f) show the second bridge, and the modules in transit. This operation is completed in Frames (g) and (h). Another swap is shown in Frames (i) and (j).

An interesting situation is how to swap to adjacent modules. The total module count in this example is an even number, so the final swap in the reconfiguration exchanges the

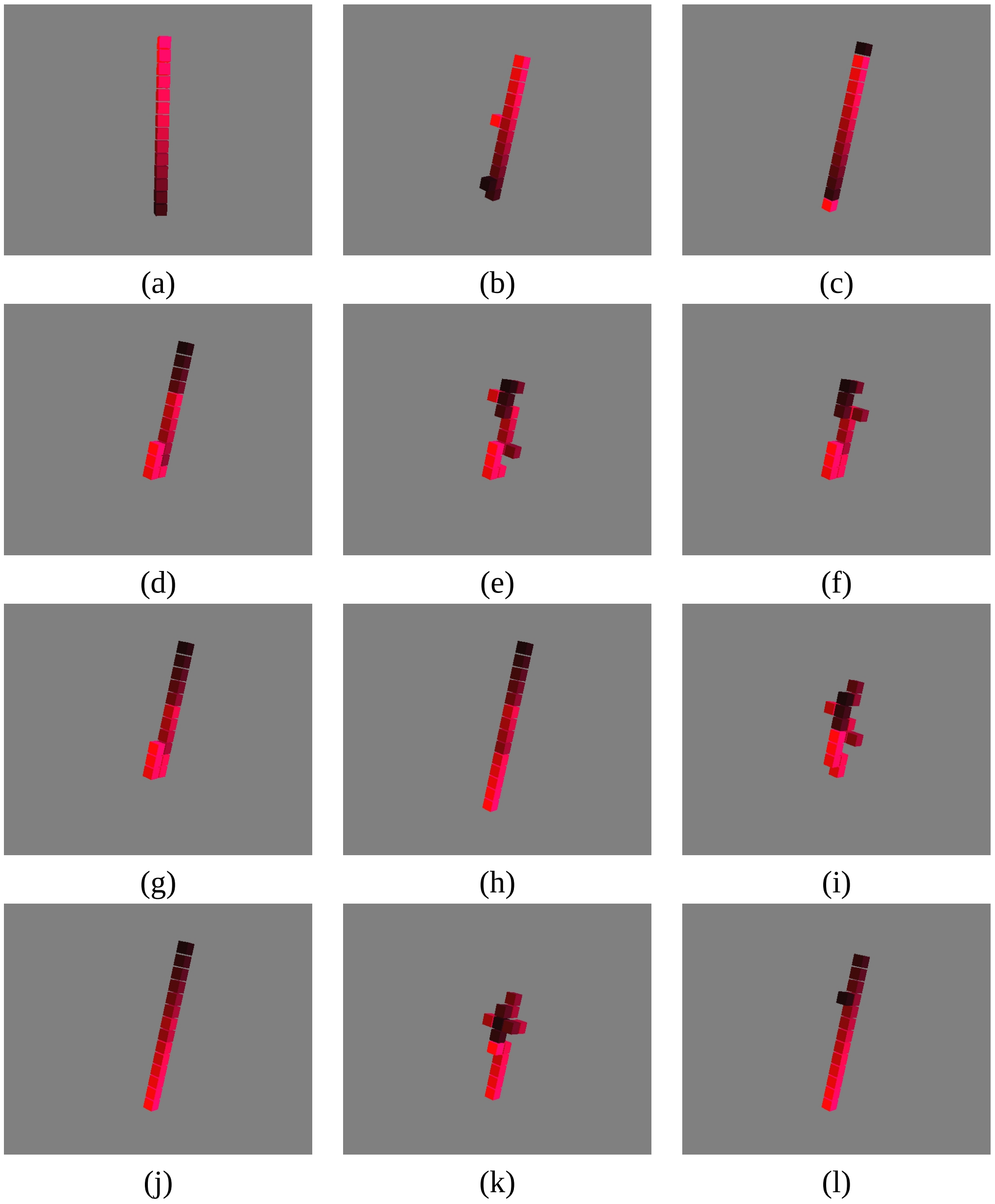

(a) (b) (c) (d) (e) (f) (g) (h) (i) (j) (k) (l)

Figure 4.14: TunnelSort example. Reversing the order of modules in a line configuration. A line is the most restrictive configuration for a Sliding-Cube system since the number of mobile modules is minimal (two). Here we see that the algorithm finds a solution by unlocking internal modules using bridging.

two modules at the midpoint of the line. The solution found by TunnelSort is in Frame (k). Note that the two modules end up sharing bridge modules in order to unlock themselves. This was not implemented as a special case; the generic bridge implementation planned it directly.

### 4.2.5 Discussion

This section presented an algorithm for sorting a configuration in-place. Coupled with a solution for in-place homogeneous reconfiguration, this provides a complete method for in-place heterogeneous reconfiguration. TS is the first published solution for this variant of the problem. The reliance on a separate method for the homogeneous phase is not limiting since 1) the greedy algorithm for the case with no holes is straightforward, 2) a similar greedy solution exists for models that provide motion through the volume of the structure, such as the unit-compressible model, and 3) we show how to simulate such motion using the Sliding-Cube model in the next section.

The theoretical importance of this result is that it proves a polynomial-time solution for heterogenous reconfiguration with constrained free-space. Earlier, we saw how heterogeneous reconfiguration exploits free space as a special case of the generally intractable Warehouse Problem. The next section of this chapter completes the picture by removing the assumption that the free space is connected, as it is in the crust.

The asymptotic worst-case number of primitive motions is not reduced in TS compared to MSG, since both are asymptotically optimal solutions in the worst-case. However, MSG forces the worst-case behavior in any configuration, even if the start and goal configurations are similar. This occurs because all configurations must proceed through the line shape chosen as the intermediate configuration, and so average path length is large ($n/2$). TS is an improvement in cases where only a small number of modules are invalid. TS can make small changes to the structure without completely disassembling it, as MSG does.

In fact, TS may be used as an alternative for the sorting procedure of MSG. In the special case of a line, decentralized TS runs quickly because all tunnels are zero-length. With a cube, TS is fast because all tunnels have length bounded by $\sqrt[3]{n}$, and there are no bridging operations. Empirical data that shows the actual number of moves for cube-shaped instances is given later in Table 6.3. We see that TS is much faster than the sort approach specified in MSG.

A single tunnel in TS can also be viewed as a local instance of MSG. A possible heuristic improvement is to re-order the tunnel modules as they are replaced. If any of these tunnel modules can be placed in valid positions, the total number of swaps is reduced.

The overall concept of tunneling is important since this is a useful technique in other variations of the reconfiguration problem. The algorithm in the next section is based on a modified tunnel technique, for example. Also, this is used as the basis for in-place homogeneous reconfiguration with holes, which has not been published thus far for the Sliding-Cube model. Along with the natural surface motion of Sliding-Cube systems, the addition of motion through the volume as provided algorithmically by the tunnel technique makes the Sliding-Cube a more powerful model. Previously, only unit-compressible systems using meta-modules had this property.

## 4.3 Reconfiguration Among Obstacles

A main theme of this chapter is the gradual reduction of free space available during reconfiguration. In this section, 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 interesting theoretical question of reconfiguration planning with severely constrained, possibly disconnected free space. Our algorithm includes a general method for homogeneous shape formation in-place as well, so it represents a full solution for heterogeneous reconfiguration. Our approach is to refine the notion of tunneling presented earlier into a set of useful algorithmic tools for module motion through the volume of the structure in the Sliding-Cube model. The only model that supports such motion primitively is unit-compressible actuation with meta-modules.

The TunnelSort algorithm addresses heterogeneous reconfiguration in-place but relies on a separate algorithm for forming the initial shape homogeneously. Homogeneous reconfiguration in-place with a one-unit crust is straightforward in configurations of surface-moving modules with no holes. A simple greedy method suffices. Full analysis is given by Kotay [54]. In unit-compressible systems with meta-modules, a similar greedy method works in-place with no additional free space beyond the union of the start and goal shapes [99]. However, for surface-moving modules either in configurations with holes, or with free space constrained beyond a one-unit crust, this method fails. The algorithm presented in this section is useful because it addresses these problem instances.

The practicality of constraining free space is that we can model obstacles. An obvious example where this is useful 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, one of the main motivations of self-reconfiguration research is that the robot can change shape and move through complex obstacles. This algorithm can allow the robot to move tightly against obstacles, in a type of compliant reconfiguration. Finally, one method of locomotion of SR systems is to plan a serious of shapes that moves the robot to a desired location. This algorithm makes this task possible for homogeneous or heterogeneous Sliding-Cube systems.

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. 4.15(a). If free space surrounds the configuration, a

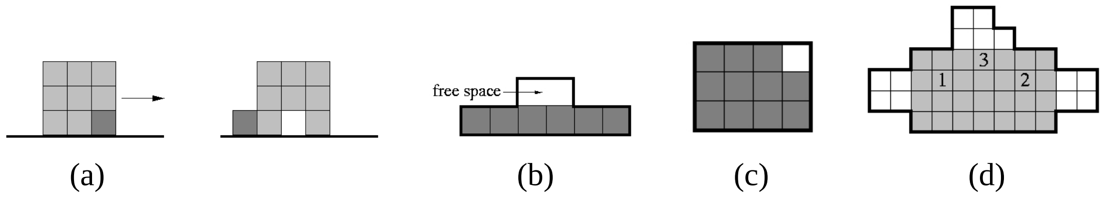


Figure 4.15: Challenges of reconfiguration with free space constraints. Example (a) shows failure of greedy homogeneous reconfiguration in presence of free space constraints. Viewpoint is side view of 3D configuration. Start configuration is a cube, goal configuration is same shape translated to the 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.

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. 4.15(b) shows a locked configuration where no moves are possible. A heterogeneous example is in Fig. 4.15(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 [43]. Finally, even if motion through the volume of the structure is provided, free space constraints can make a greedy swap order fail. Consider Fig. 4.15(d). Our earlier planner might swap modules 3 and 1, then 2 and 3. But the 2-3 swap is not possible because of free space disconnection, so we would have to first swap 2 and 3, then 2 and 1.

Our approach to these challenges is to move modules through the volume of the structure in a planned order. First, we detail a revised tunneling procedure. Then, we use this technique to develop an algorithm for homogeneous reconfiguration. The simple greedy algorithm is listed for convenience. We then present the heterogeneous component of the algorithm in both centralized and decentralized forms, along with analysis. The section concludes with an example and discussion.

### 4.3.1 Tunnel Paths with Bends

A tunnel was defined in the previous section as a straight-line path. Because free space was guaranteed to exist at the surface of the structure (the crust), a straight-line path was also guaranteed to exist that connects any module to free space. With the loss of this free-space assumption, a straight-line path is no longer sufficient. However, we can create a more intricate tunnel path by allowing bends. This requires a more sophisticated, but still linear-time, search procedure.

Recall that the main difficulty in removing a group of modules is maintaining global connectivity in the structure. Earlier we saw that a particular local substructure was sufficient to prevent disconnection during tunneling. 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. For example, if the tunnel is running along the *z*-axis, possible walls are positive *x*, negative *x*, positive *y*, and negative *y* neighbors.

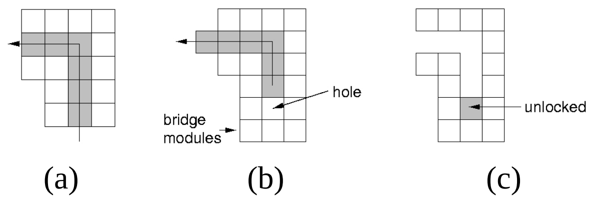


Figure 4.16: Tunnel with one bend. Shaded modules are tunnel modules; unshaded modules form tunnel walls. Tunnel path is shown in (a), (b) shows virtual module relocation by shifting tunnel modules, and (c) shows unlocking a module for swapping.

Now consider a bend in a tunnel path. The walls of the tunnel must have bends as well. See Figure 4.16. The following Lemma shows that allowing bends preserves our connectivity guarantee.

**Lemma 9 (Tunnel with bends preserves connectivity).** *Any set $M_{tunnel}$ of adjacent modules forming a chain can be removed without causing global disconnection if all neighbors of $M_{tunnel}$ are connected other than through $M_{tunnel}$.*

*Proof.* Proof is similar to Lemma 3. Since all neighbor modules are connected, removal of a module $m$ in $M_{tunnel}$ cannot cause disconnection because any two remaining modules in the structure that were connected through $m$ before removal remain connected following removal. $\square$

To find a tunnel path from a given module to free space, we can use a simple search procedure such as 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. Consider the path in Figure 4.16(a) marked by the arrow. Note that the wall configuration is the same for the first four shaded modules in the path. At this point, sufficient modules exist for both walls to bend along with the tunnel path. Using these simple local rules, we can use DFS to search for tunnel paths.

If the search ends without reaching the goal, it is still possible to continue searching. We will detail this in later sections, but for now we refer to the portion of a tunnel with constant wall configuration as 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 TunnelSort.

### 4.3.2 Homogeneous Phase

Using module relocation through tunneling, we now present an algorithm for the homogeneous (shape-only) phase of heterogeneous reconfiguration. Unlike the greedy method, this algorithm is in-place with free space constraints, and allows for configurations with holes. The inputs are a start configuration, goal configuration, and free space defined as a set of lattice positions. The output is a reconfiguration plan. Note that since this is intended as a homogeneous phase of a heterogeneous algorithm, we ignore type information in the configurations. The objective is to form the goal shape only, and a second (heterogeneous) phase will correct type errors.

We now describe an extension to our tunneling technique to support virtual module relocation. Then we present centralized and decentralized versions of the algorithm along with analysis.

#### Virtual Module Relocation through Tunneling

The objective of virtual module relocation is to fill an empty lattice position by shifting modules along a path. This idea can be implemented using tunnels. Instead of completely removing tunnel modules from the tunnel path as we did in previous sections, 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. A tunnel segment ends at the termination of one of the tunnel walls. This termination point is an open lattice position, which lies either on the surface of an internal hole or on the outside surface of the structure. The final tunnel module in the path is thus adjacent to this surface and can traverse it to reach the beginning of another tunnel segment. Linking tunnel segments in this way, we can virtually relocate a mobile module to any free lattice position.

To search for a tunnel path from a module to an open lattice position, we search the module connectivity graph beginning with the initial module. Any graph search algorithm can be used; we use DFS since it is easy to implement in a decentralized way. When we reach a module adjacent to free space, we immediately search the surface of the free space to check if we can reach the goal position, if not, we add all modules adjacent to this free space as child nodes and continue. Otherwise, the search proceeds as described earlier in Section 4.3.1. Because the underlying algorithm is DFS, we can search the entire structure in linear time.

After a path is found, motion is executed by iteratively shifting tunnel modules along the tunnel segments in the path. To initiate the shift, we must bridge the tunnel walls temporarily while the shift happens. Then the bridge modules can return to their original positions. This bridging procedure is the same as defined earlier in Section 4.2. The number of primitive motions performed during execution of a tunnel path is equal to the number of modules shifted plus the distance traveled by modules during surface motion linking tunnel segments. This is linear in the number of modules $n$, or $O(n)$.

An important feature of this search procedure is that it effectively prioritizes surface motion over internal motion. This results from adding all modules on the surface of free space as children directly at the end of a tunnel segment. Therefore, if the goal position is reachable via surface motion from a given position, this path will be found before continuing to search internally with additional tunnel segments.

### Centralized Algorithm

Algorithm 10 lists the centralized version in pseudocode. The overall structure is similar to the greedy solution. The additional capability of tunneling, however, avoids the failures of the greedy approach with surface motion only. Search for $m$ is conducted by iterating over the list of modules; search for $p$ iterates over the list of modules in the goal configuration.

**Algorithm 10** Homogeneous in-place reconfiguration with free space constraints allowing configurations with holes. Modified greedy approach that makes use of motion through the volume of the structure in addition to over the surface. Decentralized version is listed as Algorithm 11.

```
1: while reconfiguration is not complete do
2:     Choose mobile module $m$, where lattice position $m$ is empty in goal configuration
       and $m$ is bridgeable
3:     Choose lattice position $p$, where $p$ is occupied in the goal configuration but not
       filled in current configuration
4:     Search for tunnel path between $m$ and $p$
5:     Fill $p$ by executing virtual module relocation from $m$ to $p$
```

## Decentralized Algorithm

The equivalent decentralized algorithm is listed as Algorithm 11. We assume that each module has a copy of the goal configuration and knows its current position in goal configuration coordinates. Each module runs the same code, and message *start* arrives at a single module to begin algorithm execution. Algorithm steps parallel the centralized version, with centralized iterative loops replaced by message-passing. Message *findMobileModule* is propagated until it arrives at a suitable module $m$. Then $m$ begins a tunnel search to find a goal position $p$ using DFS. This search is implemented, as in our other decentralized algorithms, by message-passing. See Section 3 for a description of this technique. When a path is found, $m$ begins execution. The module that fills position $p$ signals $m$, and a new search is begun with a call to moveOneModule(). When no more mobile modules are found, the configurations are the same and the algorithm terminates.

**Algorithm 11** Decentralized homogeneous in-place reconfiguration with free space constraints allowing configurations with holes. Centralized version is listed as Algorithm 10.

```
State:
articulationPoint, am I an articulation point

Messages:
start, sent to exactly one module to begin algorithm
   Action: moveOneModule()
labelArticulationPoints, labels articulation point modules
   Action: DFS-send(labelArticulationPoints), setting articulationPoint as result
findMobileModule, search to find bridgeable non-articulation point
   Action: execute handleFindMobile()

Procedures:
moveOneModule()
   DFS-send labelArticulationPoints
   DFS-send findMobileModule
   if result is false, signal start of heterogeneous phase
handleFindMobile()
   if articulationPoint is false and there is enough adjacent free space to bridge my neighbors
   then
      findTunnelPath()
      executeTunnelPath()
      moveOneModule()
   else
      DFS-send(findMobileModule)
findTunnelPath()
   if adjacent to free space then
      search for goal position reachable with surface motion
   else
      search for tunnel path with DFS
executeTunnelPath()
   bridge start of tunnel segment
   shift modules along path
   signal start of next tunnel segment
   when goal is filled, signal return
```

## Analysis

We now show properties of correctness and completeness. We also prove the running time as claimed.

**Theorem 5.** *The homogeneous phase of CTS computes a feasible reconfiguration plan of length $O(n^2)$ for all start and goal configurations in $O(n^2)$ time, where $n$ is the number of modules in the system.*

*Proof.* We first show correctness of the centralized version. By our choice of modules to move, we never place a module in a position that is unfilled in the goal configuration. The tunnel procedure is correct by Lemma 9. Therefore, the correct goal configuration is formed without breaking connectivity or violating free space constraints.

The steps of the decentralized algorithm are equivalent to the centralized version, as shown in Section 4.3.2. Procedure moveOneModule() replaces Steps 2-5 of Algorithm 10. After a module has been correctly positioned, this procedure is called again. This replicates the loop structure of the centralized algorithm. This repeats until the goal is reached. The decentralized version thus also correctly produces the goal configuration.

Completeness is proved as follows. First we show that the tunnel procedure finds a path from any mobile module to any unfilled surface lattice position. Because the configuration remains connected, there exists a path through the module connectivity graph between any two modules. Our tunnel procedure shifts modules along any path, provided there is sufficient room to bridge the beginning of tunnel segments. If the surfaces of the start and goal configurations have the same shape, then this bridging occurs internally and is not affected by free space constraints. Otherwise, since it is always possible to greedily plan reconfigurations without holes [54] we know that relocations are possible until the surfaces of the shapes match. Then we have reached the earlier case and relocations proceed internally until reconfiguration is complete.

We now analyze the running time of the algorithm. For an $n$-module configuration, the maximum number of relocations is $O(n)$. For each relocation, we spend $O(n)$ computation steps in the centralized version (messages in the decentralized version) to find articulation points, $O(n)$ time (messages) to find a mobile module, and $O(n)$ time (steps) to find a tunnel path. Total search time is $O(n + n + n) = O(n)$. For each relocation, we spend $O(n)$ time (messages) and motions during tunneling. Running time overall for both centralized and decentralized versions is therefore $O(n^2)$ time with $O(n^2)$ primitive moves. Alternatively, if we bound the length of a module path by $p$ then plan complexity is $O(np)$ moves. $\square$

### 4.3.3 Heterogeneous Phase

The heterogeneous phase corrects type errors left by the earlier homogeneous phase. Since the goal shape as already been formed, the problem specification is the same as for TunnelSort: the inputs are a type specification for the goal configuration, and a matching start configuration with type labels. We also have free space constraints in the form of a set of free lattice positions as in the homogeneous phase.

We begin with a method for moving a given module through the structure using another refinement of tunneling. Because free space can be disconnected, the order of swapping can no longer be arbitrary. We discuss an algorithm for determining this order, and then give centralized and decentralized algorithms and analysis.

#### Actual Module Relocation through Tunneling

The tunnel procedure described in Section 4.3.1 can be extended to perform actual module relocation as in the TunnelSort algorithm. This allows us to exchange the position, or swap, a pair of modules. We now define a procedure for swapping two given 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 their movements. Now the structure, minus $m$, is returned to its previous configuration. Repeating this for a second module $m'$ allows $m$ and $m'$ to exchange positions, or swap. Given a list of modules and a list of disconnected free space regions, we present an algorithm to determine which pairs of modules can swap with each other and to generate a motion plan that executes the swap.

The path a tunnel follows consists of an alternating sequence of *segments*, sections that pass through the volume of the structure, and *holes*, sections which pass through free space. A segment does not need to be a straight line path; it can have arbitrary shape as long as its length does not exceed the lattice distance between its endpoints. This property makes sense because the only reason a path would have to double back on itself would be to avoid a hole, which in our case would simply split the segment.

To search for tunnel paths, we will use a simple graph search augmented with additional information. The idea is to begin a breadth-first 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 $m$ can use for swap space. This algorithm is listed as Algorithm 12.

We begin 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

**Algorithm 12** 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 $d = d_i -$ size of $s_i$
10:        if $d < 0$ then
11:            $d = 0$
```

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 Figure 4.17 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.

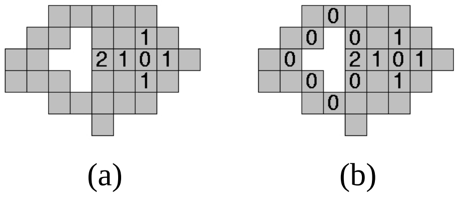


Figure 4.17: Illustration of a searching for free space regions reachable via tunneling. Distance labels keep track of tunnel length, beginning at 0. When the search reaches a hole, the size of the hole is deducted from the current tunnel distance label. See (a). The next step, shown in (b), is to reset the counter to 0 and add the modules surrounding the hole as BFS children.

### Determining Swap Sequence

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 13. 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 (Section 4.3.3). For now we will assume a method exists and continue by assigning each vertex in $V$ a color corresponding to module type.

**Algorithm 13** 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. For example, consider the graph in Figure 4.18. Choosing the specified swap order produces a solution in three operations.

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 breadth-first search (BFS) to traverse nodes until we find a match.

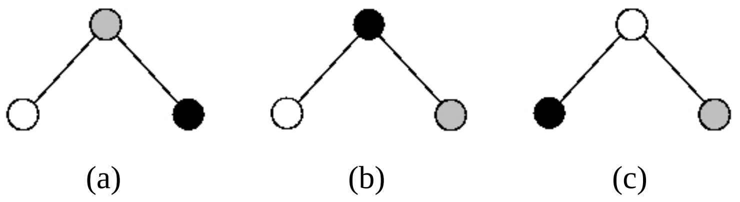


Figure 4.18: Example of a swap-graph with three nodes. Initial graph is in (a). Right child swaps with parent in (b), then left child swaps with parent in (c). Any other swap order requires more swaps to produce the goal configuration shown in (c).

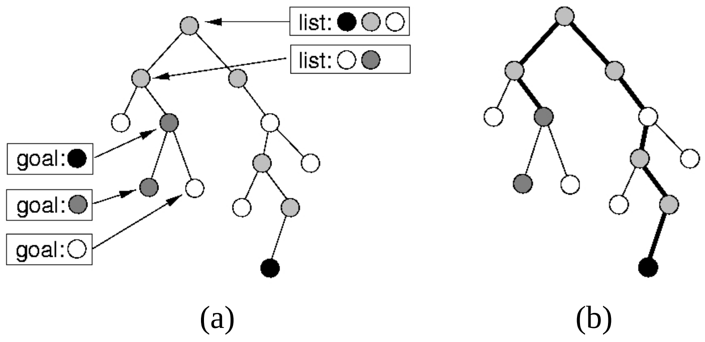


Figure 4.19: Example spanning tree. In (a), deepest nodes in left subtree already match the goal color, but their parent does not. Colors in the list associated with each node are given for selected nodes. The resulting sequence of swaps is marked by darkened edges in (b).

Then we swap colors in reverse order along the search path. This results in a sequence such as that shown in Figure 4.19.

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 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.

### Centralized Algorithm

We now combine tunneling and swap sequence to build the heterogenous phase of the reconfiguration algorithm. Initially, we find a valid sequence of swaps by building and searching 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. Swappability between each pair of modules is determined by a search procedure. We then search this graph to find a valid sequence of swaps. If no sequence exists, the algorithm fails. Otherwise, we execute the swaps using an extended tunneling procedure. This approach is summarized in pseudocode as Algorithm 14.

**Algorithm 14** Heterogeneous reconfiguration with severe free space constraints.

```
1: Build connectivity graph of swappable modules
2: Search graph for feasible swap sequence using Algorithm 13
3: if no sequence exists then
4:     Fail
5: else
6:     Execute swaps using extended Tunnel procedure
```

Assuming we have stored the tunnel paths generated earlier, we can swap using Algorithm 15. This algorithm can be easily understood as a modification of TunnelSort. Instead of only swapping using the crust, now we can swap in any free space region. Also, instead of straight tunnels, we use tunnel paths with bends. The algorithm proceeds as follows. First, we tunnel $m_1$. Using extra modules, we bridge the mouth of the tunnel. Then, we greedily move modules along the tunnel path into free space. Module $m$ can then traverse this tunnel and we reverse to reset the structure. We repeat for $m_2$, but before refilling the tunnel we move $m_1$ into $m_2$'s former position. One more tunnel allows $m_2$ to move into $m_1$'s original spot.

## 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

**Algorithm 15** 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
```

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.

Algorithm 16 defines our solution in pseudocode. A copy of this code is assumed to execute on each module simultaneously. 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.

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

**Algorithm 16** Heterogeneous reconfiguration with severe free space constraints, decentralized.

```
State:
type, my type label
goal_type, type in goal configuration at my current position
leader, true if global MST is rooted at me (initially false)

Messages:
start, sent to begin execution (arrives at exactly one module)
    Action: Broadcast MST.
MST, sent to build minimum diameter spanning tree over graph of swappable modules, rooted
at this module
    Action: Execute MST().
MST_done(cost c), sent to announce cost of MST and begin next phase
    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.
validate, sent to fix color of module
    Action: Execute validate().
done, sent to signal algorithm termination
    Action: Clean up any leftover state, start next task.

Procedures:
MST()
    if type = goal_type, broadcast MST cost as null
    search free space to find holes we can reach
    build MST
    broadcast MST cost
validate()
    search for matching color
    execute swaps along search path
    return when color is correct
DFS-send(message)
    send message to first child, wait for response
    repeat for all children and compute result
    send result in return message to parent
```

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 15. 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 Section 4.2.

## Analysis

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. Unlocking a module requires the same time as in TS, $O(25n + 4t^2)$. 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 factor of $t$.

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.

The number of swaps in the decentralized version is the same as in the centralized analysis. Therefore, the planning and plan complexity for the decentralized case is also the same as in the centralized case.

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

**Theorem 6.** *Algorithm ConstrainedTunnelSort 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. $\square$

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 6 suggests a polynomial-time decision algorithm. We can build the swappability 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.

### 4.3.4 Algorithm: ConstrainedTunnelSort

**Algorithm 17** Algorithm ConstrainedTunnelSort. Plans reconfigurations for heterogeneous systems with free space constraints, allowing configurations with holes.

```
1: Form shape using Algorithm 10 or 11
2: Correct type errors with Algorithm 14 or 16
```

We now combine the homogeneous and heterogeneous phases together as Algorithm 17. This provides a full reconfiguration solution.

### 4.3.5 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 (see Chapter 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. Such a gait is presented in Chapter 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.

### 4.3.6 Discussion

In this section we developed a novel reconfiguration 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 for this problem variation. The heterogeneous algorithm requires $O(n^4)$ time in the worst case but solves instances with complicated free space constraints. There is also a $O(n^2)$ time decision algorithm to detect whether a given instance is solvable. The tunnel procedure that enables both algorithms is a useful general method providing motion through the volume of the structure for Sliding-Cube modules.

Recall that polynomial-time solutions by Sharma [87] utilize a connected region of free space. Here we add to the list of special cases that can be solved polynomially. We still do not have an enumeration of all special cases, but can characterize instances that are solvable in terms of size and connectedness of free space.

Despite the slow worst-case bound on running time for the heterogeneous phase, this algorithm has important practical applications. The homogeneous phase can be used for locomotion among obstacles, for example. Locomotion would occur via a series of goal configurations, possibly generated on the fly, that have the effect of moving the robot along some path. The best existing locomotion algorithms, based on cellular-automata rules, are simpler but require assumptions such as constant width or height in order to prevent disconnection [8]. Our algorithm is useful as a subroutine in cases where complex obstacles prevent fast locomotion.

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.

## 4.4 Position Constraints

In all reconfiguration techniques discussed so far, we assumed the availability of a fixed goal configuration. We began to relax this assumption in the discussion of locomotion; now we investigate the idea of alternate representations of the goal configuration in the context of modules specialized by function. For example, a camera module might need to remain at the front of the robot during reconfiguration, or heavy battery modules might need to remain at the bottom. We refer to these restrictions as *position constraints*. Maintaining position constraints during reconfiguration is related to the general reconfiguration problems of this chapter in two ways. First, this approach can be thought of as an extension of the main theme of reconfiguration in heterogeneous systems. Second, the algorithmic tools enabling this task come directly from reconfiguration solutions.

The general idea of reconfiguration according to function is unexplored. In this section, we make initial steps. We propose new representations, in the data-structure sense, of configurations, and outline how constrained modules acquire and react to updates of the current configuration. We also present a simple algorithmic demonstration in simulation.

### 4.4.1 Representation

In order for a given module to maintain position constraints relative to a structure, it must have some knowledge of the current global state. This knowledge might be estimated, inexact, or out-of-date, but in the absence of global state, the module (or some module controlling it) cannot maintain its relative position in a changing structure.

Our choice of representation of this state should be small and simple, and able to be dynamically updated. We propose representing the goal shape as a sum of boxes. This is similar to the representation of a 1D signal as the sum of sinusoids. Such a representation can be complex enough to express position constraints at the required granularity, but could also be constant size, such as a bounding box, if position constraint requirements are simple. Small size is important since this is data that will be updated frequently via inter-module communication.

The question of how and when to update this global state is important since this determines communication cost, and also how quickly and accurately constrained modules retain their positions. Solutions vary in how frequently updates are sent, and in the number of modules receiving updates. At one extreme, each module broadcasts its position at every move. At the other extreme, updates are sent locally and infrequently. Hybrid approaches include partial updates punctuated by infrequent full updates, such as the idea of a key frame, or some other combination.

### 4.4.2 Constraints on a Single Module

Position constraints for a single module can be defined by specifying a box face in the goal representation. For example, to maintain a sensor module on top of the robot, the top face of a bounding box representation is chosen.

Once this constraint is expressed and a method of dynamically updating global state is provided, the problem of maintaining position can be addressed as a control problem. Standard control techniques can be adapted and applied. Low-level motion is planned by algorithms for either surface motion or tunneling.

### 4.4.3 Algorithm: Maintaining Position Constraints

We implemented a simple example to illustrate the concept. The task is to keep a module at the top center of a randomly-moving Sliding-Cube robot. We chose a single rectangle (bounding box) representation, with each module broadcasting updates at every move. A simple algorithm executed by a constrained module is listed as Algorithm 18, and screenshots are shown in Figure 4.20.

**Algorithm 18** Position constraints algorithm executed by constrained module. Assumes continuous updating of goal configuration, represented by a single box. Module attempts to maintain its position at the top center of the structure.

```
1: wait for state update
2: **if** new position is outside bounding box **then**
3:     compute new bounding box
4:     plan path to top center
5:     execute motion plan
```

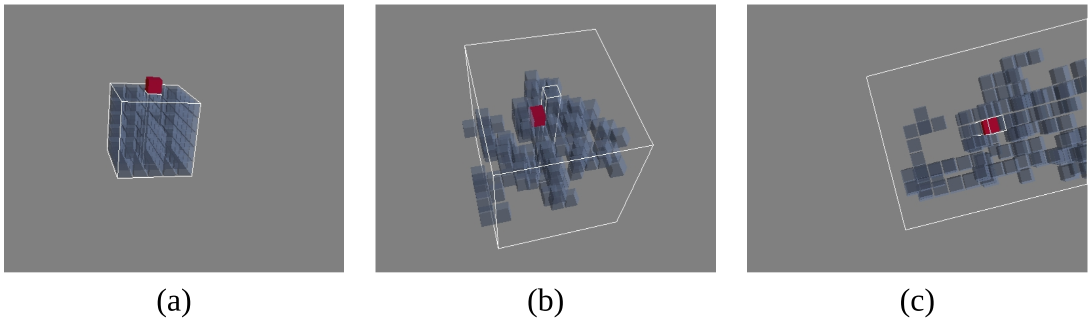


Figure 4.20: Position constraints example. The dark module attempts to maintain position in the center top of the structure. The white lines indicate the bounding box, and the center column marks the actual center. Modules move randomly, favoring motion to the right. Note that the configuration moves to the right as a result, but the constrained module maintains position.

### Analysis

Communication cost of this example is $O(n)$ per module motion, since each module broadcasts its position at every move. The constrained module can maintain its position exactly since it has complete information.

### 4.4.4 Discussion

The example presented in this section is a simple illustration of the position constraints concept. Further research is required to achieve greater communications efficiency. A good goal is to provide a parameterized algorithm that trades of communication cost with position accuracy.

# Chapter 5

# Applications

Realizing the vision of heterogeneous SR robots requires solutions to planning and control problems beyond the basics of shape-changing. In Chapter 1, we outlined a number of these related problems and identified three that we chose to study in this thesis. Other important issues, such as human-robot interaction and goal configuration determination, are left to other researchers. The three problems addressed in this chapter are distributed goal recognition, locomotion, and self-repair. We study these particular issues because of their strong relationship to our main reconfiguration planning results.

The first issue we address in this chapter is the problem of distributed goal recognition in heterogeneous systems. This is an important problem, since any type of reconfiguration requires the system to realize when it has reached its goal. Our GR algorithm compares the current state to a goal state, defined by the same heterogeneous configuration representation we use in reconfiguration algorithms, in a distributed fashion. Even though our reconfiguration algorithms are guaranteed to terminate when the robot reaches its goal configuration, it is useful to consider goal recognition as we do here, as a stand-alone problem. For example, GR can be used as a subroutine in other applications or in other types of reconfiguration algorithms.

The second issue we address in this chapter is locomotion. As previously described in Chapter 1, locomotion is an important application of reconfiguration. The ability to change shape affords SR systems versatile locomotion modalities specially suited for moving over rough terrain and through unknown environments. The issue of uncertainty arises here since modules are likely to experience connection, actuation, communication, or other failures during operation. The third issue we discuss, self-repair, is how to handle this total module failure. This is high-level, discrete uncertainty as opposed to the low-level, continuous uncertainty faced by systems that control connection, disconnection, and actuation mechanisms.

Our approach to the locomotion and self-repair problems is based on our reconfiguration planning work, which composes basic motion primitives into higher level trajectory primitives, and then further composes these into reconfiguration plans. For locomotion, we begin with a simple algorithm for unit-compressible systems that requires no changes in module connections. This algorithm, called Inchworm Locomotion, has been implemented in hardware (see Chapter 6). We then develop a general method for Sliding-Cube systems that supports compliant locomotion among obstacles. The nature of this motion is such that the robot moves along rectilinear paths, where straight segments are preferred to turns. This leads us to investigate rectilinear path planning that minimizes bends, and we describe the first published solution to the 3D minimum-bend path problem (3D-MBP).

In the self-repair problem, we view a robot with a failed module as a heterogeneous system with an unactuated component. We develop trajectory primitives that collaboratively transport a module that cannot move on its own, and use these to construct a self-repair approach. The 3D-MBP problem occurs here as well, since these trajectory primitives are more efficient for straight-line paths than for turns.

The chapter is organized into four sections. Distributed goal recognition is presented first, in Section 5.1. Section 5.2 describes the Inchworm Locomotion algorithm and general locomotion among obstacles. 3D-MBP is presented independently in Section 5.3. Self-repair concludes the chapter in Section 5.4.

## 5.1 Distributed Goal Recognition

Since modular robots operate as a tightly coupled distributed system, most usage depends on collective coordination and recognition of the current state of the system. This is especially important in the context of recognizing that the system has achieved its goal. For example, the goal can be a desired shape for a self-reconfiguring application, reaching a certain spot for locomotion or implementing a given configuration for manipulation. As discussed in Section 5.2, locomotion algorithms based on local rules do not explicitly track the global configuration centrally. Goal recognition could be used in conjunction with other algorithms for locomotion or reconfiguration tasks. The possibility of using goal recognition as a frequently called subroutine is mentioned in [106].

Distributed goal recognition is challenging because each module in the system has access to local state information only. This information has to be integrated to form a global picture, and modules must find their positions in this overall state. In order to compute the solution, the union of these configurations has to be compared against the goal.

We now present our approach for solving the goal recognition problem for heterogeneous robots. We outline the main idea of the solution, develop the algorithm for 2D robots, then propose a 3D version. Both algorithms use the same resource upper bounds, $O(n^2)$ total messages executing in (parallel) $O(n)$ time. Finally we describe the implementation of our solution in simulation.

### 5.1.1 General Approach

The general goal recognition problem asks if a modular robot's configuration matches a particular goal. Configurations can be represented as a binary matrix for homogeneous systems, with 0 corresponding to empty space and 1 corresponding to space occupied by a module, or as a matrix with value indicating module type for heterogeneous systems. This representation leads to the following problem formulation: given an oriented goal matrix $G$ and a modular robot $A$, determine if $A$'s configuration matches that specified by $G$. We assume that the robot is a purely distributed system comprising modules that have local communication, limited processing power and memory, and that form a lattice. To simplify presentation, we consider square (in 2D) and cubic (in 3D) module shapes, but the algorithm works with other shapes as well.

Our solution to distributed goal recognition is based on a technique we call a *trace*. Intuitively, a trace is a tour of the modules of the robot matched at each step against the goal matrix. Consider the situation where one module is assigned a position in the goal matrix. If the matrix value matches the module's type, then the module is considered *valid*. That module then passes a message to a neighbor, including an indication of the matrix position corresponding to that neighbor. This new module could then decide whether it is valid, and so on. If any modules are not valid, then the trace is said to fail. Conversely, if all modules are valid, then the trace is said to succeed, and under certain conditions this implies that the robot is positively in the goal configuration. In particular, if the number of modules in the robot and in the goal matrix are the same, and both are fully connected, then a successful trace is both necessary and sufficient to solve the goal recognition problem. Of course, the problem remains as to how any module knows whether all other modules are valid. We approach this differently in 2D and 3D, but in both cases the message-passing policy guarantees that the module originating the trace eventually receives the “answer.”

The main idea of our algorithm is for multiple modules to initiate traces in parallel, each testing themselves against the same well-known position in the matrix. We call this position the *anchor*, and arbitrarily define it as the upper-left corner (in 3D, forward upper-left) of the target shape. The modules find this position by scanning the matrix for the first non-zero value. In 2D, a module matching this local configuration would have no north or west neighbors, so any such physical module is called *special* and knows to initiate a trace when the algorithm begins. Neighbor detection can be performed using message-passing or through the use of some special sensor. At most one trace will succeed, and the winning module then sends a global message. If all traces fail, however, then the situation is slightly more complex since no single module has enough information to discern global failure. If at least one module knows that all traces have failed, it can then propagate a global failure message. Therefore, when a trace fails, its originator sends a failure message that acts as a “meta-trace.” Any special modules that receive this message hold it until their respective traces fail, then send as normal. When the module gets the meta-trace back, it knows that all other traces have failed and it can propagate the global failure message.

Since the algorithm is distributed, the method of signaling the algorithm’s result is not immediately clear. If the algorithm is used as a subroutine in a larger context, then the first module to attain the global answer could return the result. In this case, however, we are working with goal recognition in a stand-alone fashion. We chose to simply propagate the answer throughout the system and programmed the modules to execute a predefined behavior to signal success or failure.

This algorithm has several nice properties. First, it is distributed and avoids the concept of a supermodule. It also requires less space than simpler solutions. For example, one such simple approach would be that each module broadcasts (through local message-passing) its ID and the IDs of its neighbors. When one module has received a message from all modules, it builds a connectivity graph, converts it into a matrix, and compares against the goal matrix to compute the answer. That solution requires linear space in each module, or $O(n^2)$ space overall versus our algorithm which requires only linear space overall. Our solution also requires fewer messages. We now describe the algorithm in detail.

### 5.1.2 Goal Recognition for Homogeneous Planar Robots

We begin with the simple case of 2D homogeneous robots. If the target shape has no holes, and the number of modules in the robot and target are equal, then a trace needs only to compare the *perimeter* modules against the goal. This is easily accomplished by passing the trace messages according to the *right-hand rule*. We use the reverse order for convenience so a more accurate term would be *left-hand rule*. Passing messages along the perimeter also insures that the trace message always returns to the originator after being seen by all perimeter modules. Modules pass messages around the perimeter using a LeftHandPass function that takes the direction of the incoming message as a parameter and sends to the next module in a clockwise direction (potentially the previous sender).

The algorithm begins when a message (GR_START) is generated from an outside source, such as an outer algorithm using Goal Recognition as a subroutine, and is propagated through the system. Special modules initiate traces in response to the start message, and a successful trace generates a success message. Failed traces otherwise generate meta-traces, eventually resulting in a global failure message. Behavior to signal success or failure could be implemented as a return to the calling algorithm. Pseudocode for the algorithm, listed as a series of message handlers, is given in Algorithm 19.

Examples with only one special module are given in Figure 5.1. Part (a) shows a sim-

**Algorithm 19** 2D Goal Recognition message handlers.

```
GR_START:
PostMessage(GR_START)
if I am special then
   LeftHandPass(TRACE, receivedFrom)

TRACE(vector goalPos):
if message is from me then
   PostMessage(GR_SUCCESS)
else
   if valid(goalPos) then
      LeftHandPass(TRACE(neighborPosition, receivedFrom)
   else
      LeftHandPass(TRACE_FAIL, receivedFrom)

TRACE_FAIL:
if message is from me then
   LeftHandPass(META_TRACE, receivedFrom)
else
   LeftHandPass(TRACE_FAIL, receivedFrom)

META_TRACE:
if message is from me then
   Signal(FALSE)
   PostMessage(GR_FAILURE)
else
   if I am special then
      while my trace has not returned do
         wait for my trace message
   LeftHandPass(META_TRACE, receivedFrom)
```

ple shape. Once GR_START is received, it propagates through the robot. Meanwhile, $m_1$ initiates a TRACE message, which is valid for each module. When the TRACE returns to $m_1$, GR_SUCCESS is sent and the robot signals success. In (b), the trace fails at $m_2$ and TRACE_FAIL is sent. Module $m_1$ receives the TRACE_FAIL, sends the META_TRACE, and then follows with GR_FAILURE. Figure 5.2 is a more complex example with multiple special modules.

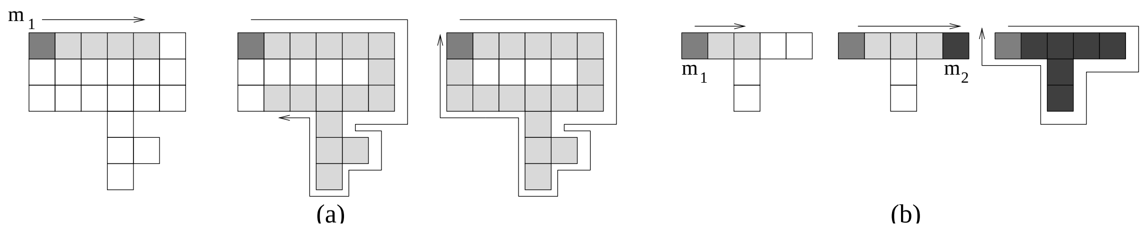


Figure 5.1: 2D goal recognition examples. In (a), module $m_1$ sends a successful trace message. Modules that have received the trace are shown in light gray. Arrows indicate direction of the trace. The goal in part (b) has no module at $m_2$, causing the trace to fail. Dark modules indicate TRACE_FAIL propagation.

**Analysis**

Our algorithm correctly solves the goal recognition problem for all instances in which the robot and goal shape have the same number of modules and no holes. This can be shown by considering two cases. First, if the robot is in the goal configuration, the anchor position in the matrix must match one special module. This module will initiate a trace that necessarily returns and is valid, at which point global success is signaled. If the robot does not match the goal shape, the perimeter of the goal cannot match the robot's perimeter. Therefore, all traces must fail, but regardless of the robot shape will return to their parent modules. A META_TRACE message is then allowed to pass through all special modules to its originator, and global failure is signaled.

The total number of messages is $O(sk)$ trace messages($s$ traces of $k$ messages each),

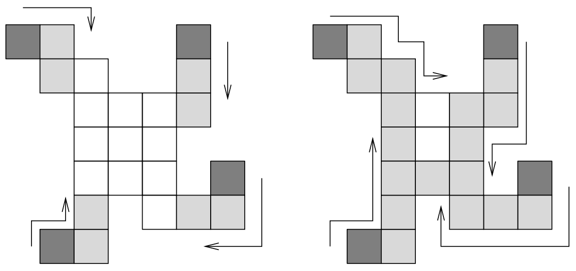


Figure 5.2: Multiple special modules. Dark modules are special and initiate traces in parallel, with paths of the traces indicated by arrows.

where $s$ is the number of special modules and $k$ is the number of modules on the perimeter, plus $O(n)$ messages for propagating the solution, or $O(n + sk)$ total. Since $s < k <= n$, the overall upper bound on number of messages is $O(n^2)$. Since messages are sent in parallel, the time requirement is linear in the number of modules. The space requirement is constant per module, or linear overall.

**Extension to General Heterogeneous Shapes**

The 2D algorithm we presented works under the assumption that the structure has no holes and is homogeneous. However, with a simple extension we can be sensitive to heterogeneous structures with holes as well. The addition is another layer of validation messages prior to the global success signal. If a trace is successful (note that even with holes present, only one trace can succeed), its originator propagates a message through the modules similar to the 3D trace described below. If any interior modules are invalid at this time, they can immediately broadcast GR_FAILURE, while if the message returns to the originator, it can signal GR_SUCCESS. This increases the number of messages by a small factor but does not affect the asymptotic analysis.

### 5.1.3 Goal Recognition for 3D Robots

The approach in Algorithm 19 can be used with a 3D robot. The major difference is in how a trace message is propagated. It is unclear how to propagate the trace over the *surface* of a 3D robot and guarantee that the special module receives the correct answer, so we must send the trace to every module instead, using a breadth-first search (BFS) scheme. We must still ensure that the special module retrieves the answer to the trace. This can be implemented with BFS on a spanning tree of the module connectivity graph. First, the special module sends the trace to all its neighbors, and waits for an answer from each. When it receives the trace back from each neighbor, its global answer is GR_SUCCESS if its trace is successful, and otherwise it broadcasts a TRACE_FAIL message. When a neighbor module receives a trace message, it checks to see if it has already received this message. If so, it simply returns TRACE_REPLY(FALSE). Else, it recursively sends the trace to all its neighbors, and so on. The algorithm is listed in pseudocode as Algorithm 20.

**Algorithm 20** 3D Goal Recognition.

```
GR_START:
PostMessage(GR_START)
if I am special then
  for all neighbor modules $m$ do
    PostMessage(TRACE)

TRACE(vector goalPos):
if already received then
  SendMessage(receivedFrom, TRACE_REPLY(FALSE))
else
  for all neighbor modules $m$ do
    SendMessage($m$, TRACE(neighborPosition))

TRACE_REPLY(Bool bSucceeded, int numSpecial)
if message is from me then
  if bSucceeded then
    PostMessage(GR_SUCCESS)
  else
    numTraces = numSpecial
    PostMessage(TRACE_FAIL)
else
  if received reply from all neighbors then
    if I am special then
      numSpecial = numSpecial + 1
    SendMessage(parent, TRACE_REPLY(answer, numSpecial))

TRACE_FAIL:
numTraceFails = numTraceFails+1
if numTraceFails == numTraces then
  Signal(FALSE)
  PostMessage(GR_FAILURE)
```

In order to detect global failure, each special module will increment a counter associated with trace messages. This way, when a special module gets its trace back, it knows how many special modules are in the current configuration. Then, if a special module receives as many TRACE_FAIL messages as traces, it sends GR_FAILURE. If a trace succeeds, the module sends GR_SUCCESS. In either case, the algorithm terminates. Note that since we send messages to all modules, this version is sensitive to holes in the structure.

The 3D algorithm is correct for all heterogeneous robot and goal shapes with equal numbers of modules. As in the 2D case, if the robot matches the goal configuration, exactly one trace will succeed. Otherwise all traces fail, and each special module knows the total number of special modules. Each special module therefore waits until all traces are known to have failed, and then correctly signals global failure.

The number of messages is now $O(n^2)$, and time is still linear. Space requirements are increased, however, to $O(n)$ per module.

### 5.1.4 Discussion

In this section, we have presented a simple algorithm for goal recognition in modular robotic systems. Our solution allows such a robot to compare its overall physical shape and distribution of module types with a given goal configuration represented as a type-matrix. We implemented the algorithm in simulation in both 2D and 3D versions, and also conducted hardware experiments using an SR system developed in our lab. These results are presented in Section 6.

An interesting extension would be, instead of computing a binary result, to measure how close the robot is to the goal, using some metric that could be computed in a cumulative fashion by the trace message. For example, the trace could carry a counter of valid

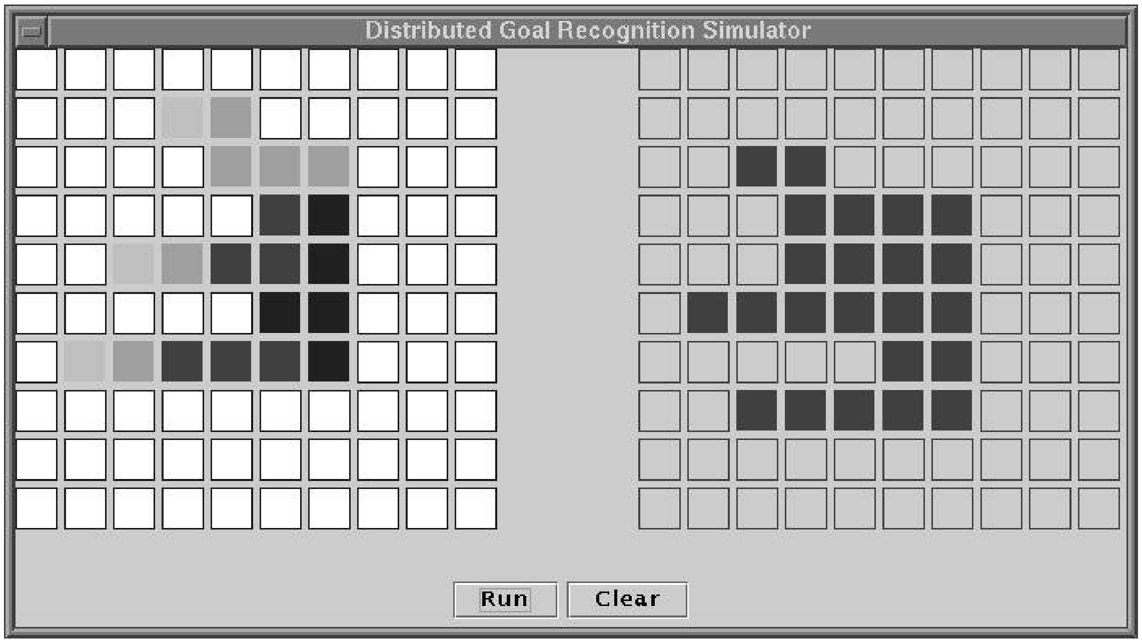


Figure 5.3: Screenshot from 2D simulator. Left pane represents robot, right pane is goal shape. Special modules are lightened. This instance will fail.

versus invalid modules, and the special modules could then use that information in some way, perhaps as input to an outer reconfiguration planning algorithm.

## 5.2 Locomotion

Most mobile robots achieve locomotion with dedicated mechanisms such as wheels, legs, or tank-treads. SR robots do not commonly include such specialized components and instead must perform locomotion through the primitive actuation capabilities of their modules. On most lattice-based systems, locomotion can be performed by moving individual modules over the surface of the group from the back to the front in a tank-tread-like pattern. In unit-compressible systems such as the Crystal, no single module can move relative to the group without help from other modules. A different specialized technique is required. We have performed extensive experiments with such a technique: a sim-

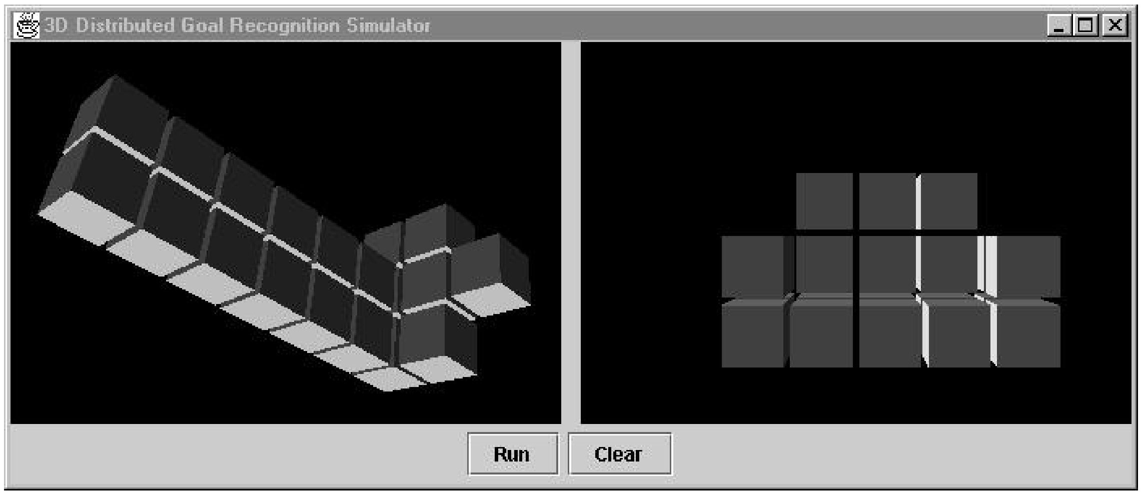


Figure 5.4: 3D simulator screenshot. The algorithm compares the robot on the left to the goal shape on the right.

ple locomotion algorithm for the Crystal robot with sensors. We describe this algorithm in this section, along with more sophisticated algorithmic techniques for locomotion in heterogeneous Sliding-Cube systems.

In general, there are two main strategies for locomotion in lattice-based systems. The first strategy is purely local and rule-based, similar to cellular automata techniques [11]. Locomotion in this strategy is fast and simple to control, but must make assumptions about the configuration, such as fixed height or fixed width. The other strategy is to use reconfiguration for locomotion. This method works in the general case, but requires planning.

We discuss both strategies in this section, beginning with a solution based on the first strategy: a simple algorithm for locomotion in a minimally heterogeneous unit-compressible system. This algorithm produces inchworm-like motion to perform locomotion on a stand-alone group of modules, taking advantage of friction with the ground to move the group forward. We then discuss a method for using heterogeneous reconfiguration algorithms for locomotion in complex environments.

### 5.2.1 Inchworm Locomotion Algorithm

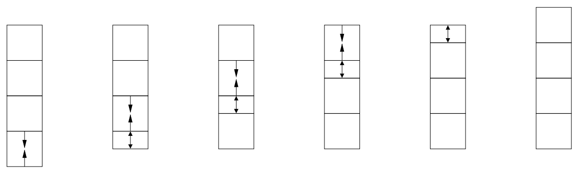


Figure 5.5: Schematic of module action under Inchworm Locomotion, in which the group is heading upward, and the series (left to right) represents progress of a single inchworm "step."

The Inchworm algorithm for locomotion in unit-compressible systems is based on a set of rules that test the module's relative geometry and generate expansions and contractions as well as messages that modules send to their neighbors. When a module receives a message from a neighbor indicating a change of state, it tests the neighborhood against all the rules, and if any rule applies, executes the commands associated with the rule. The algorithm is designed to mimic inchworm-like locomotion: compressions are created and propagated from the back of the group to the front, producing overall motion.

Pseudocode is presented as Algorithm 21. The overall idea behind Inchworm Locomotion is that at any given moment, the majority of the modules are stationary. The remaining modules move relative to the majority. In addition, the motion is specified such that two adjacent modules will move together, minimizing the net force to the other modules. A schematic storyboard of this algorithm is given in Fig. 5.5. The "tail" module contracts first and signals its forward neighbor to contract. Each module expands after contraction, so that the contraction propagates through the robot. When the contraction has reached the front of the group, the group will have moved half a unit forward (in theory; empirical results are given in Chapter 6). Depending on context, once the leader of the group has contracted and expanded, it can then send a message back to the tail to initiate another step.

**Algorithm 21** Distributed Inchworm Locomotion. Identical copies of this code execute on all modules simultaneously.

```
State:
*neighbors[]*, array of neighbors
*heading*, direction robot is moving: N,S,E,W

Messages:
*inch (direction d)*, sent to move robot in direction $d$
   Action: set *heading* state to $d$, execute TryRules()
*state (state s)*, announces state changes to neighbors
   Action: execute TryRules()

Procedures:
TryRules()
   position ← FindPosition()
   **if** position = head **then**
      **if** neighbor[opposite(*heading*)] is contracted **then**
         contract, send state
         expand, send state
         send inch
   **if** position = body **then**
      **if** neighbors[opposite(*heading*)] is contracted **then**
         contract, send state
         expand, send state
   **if** position = tail and responding to *inch* message **then**
      contract, send state
      expand

FindPosition()
   **if** rear neighbor but no forward neighbor **then**
      return head
   **else if** forward neighbor but no rear neighbor **then**
      return tail
   **else**
      return body
```

### Analysis

The Inchworm Locomotion algorithm produces locomotion in the intended direction. This is simply shown by noting that only the tail can contract at first, followed by each other module in turn. Since each contraction must be triggered by a *state* message, no module will contract until it has the proper information, but once it does contract and

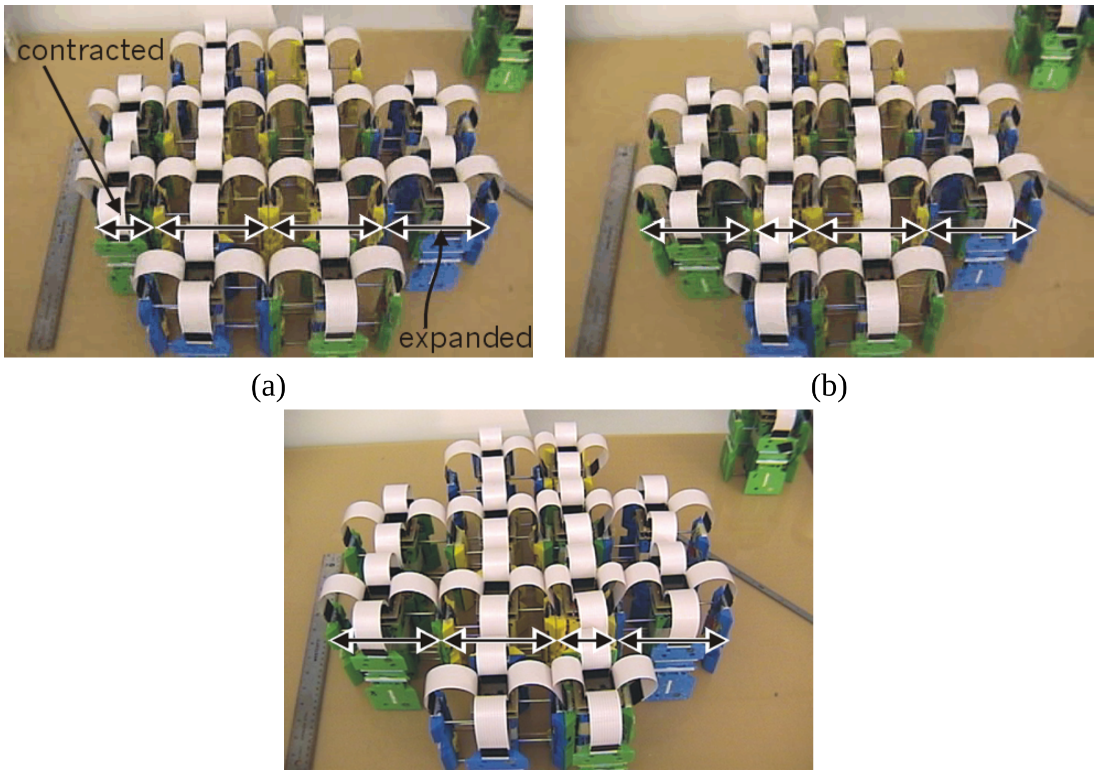


Figure 5.6: Photos of locomotion experiment for the blob shape. In (a), the leftmost column is contracted, and in (b) and (c) the following columns contract to make the group walk to the right.

sends a message forward to that effect, the contraction will always propagate. At best, this algorithm moves the robot by half the width of one module per iteration. One iteration consists of a message traveling from tail to head. In practice, the actual efficiency is less. We discuss empirical results in Chapter 6.

### Convex Shapes

Algorithm 21 is specified (and analyzed above) for a single column, but can be extended to more complex shapes by selecting one column as a master column. When a module in the master column actuates, a message is passed across its row that causes all modules in the row to actuate simultaneously. This is effective since communication is much faster than actuation, and the modules not in the master column have no other responsibilities that could cause communications lag. This allows for correct locomotion for any convex shape. An example of locomotion in such a convex shape is shown in Figure 5.6. In this way, locomotion steps can be executed in multiple directions (but only one direction at once) to allow the robot to locomote along a rectilinear path. We currently have no effective method to rotate the robot.

#### Heterogeneity

A simple form of heterogeneity results from adding sensors to an otherwise homogeneous system. In our experiments with the Crystal robot, we added touch sensors to certain units. Algorithmically, this presented an opportunity to investigate a heterogeneous version of the Inchworm algorithm by adding rudimentary obstacle avoidance. We modified the rules of the "head" module to test for touch-sensor actuation, and to reverse direction of the *inch* message as appropriate. With sensor modules on outside positions of the reconfiguration, the robot moved back and forth between obstacles. For modules without a sensor, the algorithmic changes had no affect. The software remained homogeneous.

### 5.2.2 Locomotion through Reconfiguration

The Inchworm algorithm has the advantage of simplicity and reliability. However, as specified it will work only with a planar unit-compressible system. We would like to support general Sliding-Cube systems. For example, rule-based cellular automata algorithms are also simple solutions and handle 3D systems. These can plan locomotion over certain obstacles, but in situations where the width or height of the configuration are forced to change, the rules become very complicated. Since the rules are currently hand-coded, this is problematic. It is possible, however, to leverage reconfiguration algorithms developed in the previous chapter to produce locomotion.

Recall that in MeltSortGrow, TunnelSort, and ConstrainedTunnelSort, the overlap between start and goal configurations is assumed to be given. Now, we can exploit this apparent shortcoming for the purposes of locomotion. Consider two configurations identical except for translation. Specifying minimal overlap, the execution of a reconfiguration algorithm between these two shapes results in a net translation of the robot. Continuing in this way, translation along a path can be accomplished by generating an appropriate sequence of goal configurations.

Unfortunately, generating this sequence is computationally expensive for large systems: $\Theta(n)$ per reconfiguration "step." A better strategy is to modify the reconfiguration algorithm to accept a looser goal specification. For example, the goal can be defined as a bounding box. In ConstrainedTunnelSort, the homogeneous phase can be modified such that search paths terminate at any free lattice position in the bounding box. Or, paths can be terminated at any lattice position beyond a given coordinate. This achieves locomotion in a given direction.

In the presence of sensed obstacles, this strategy results in compliant motion. The tunnel paths are designed to avoid free space constraints, and these constraints can be defined by obstacles sensed dynamically. This can be thought of as a dynamically updated goal configuration, where the updates are made by modules as they move. For example, if a module cannot make progress in the current direction because it cannot plan a path that moves it into the goal configuration, it can decide to change the robot's direction by broadcasting an updated goal configuration. Or, some policy can be implemented where multiple modules collectively decide what the update should be. The result would be compliant locomotion, as in wall-following.

### 5.2.3 Discussion

This section presented a simple algorithm for locomotion in unit-compressible systems, and a strategy for modifying reconfiguration algorithms for to produce locomotion. Both assume that specialized mechanisms such as wheel modules are unavailable. For motion on smooth surfaces, wheel modules may be suitable. But for locomotion on rough terrain, reconfiguration-based locomotion is promising.

Generating long sequences of large configurations seems inefficient and can be computationally prohibitive. The modifications to ConstrainedTunnelSort to accept loosely defined goal configurations are simple and this strategy is preferred. The potential cost is that the exact configuration at any time is no longer known, but our GR algorithm provides a method for detecting goal shapes if desired.

Another issue is that Inchworm Locomotion and specifying goal configurations as motion in a given direction both favor locomotion in a straight line. The next section addresses this issue with high-level path planning that minimizes turns in a path.

## 5.3 Rectilinear Minimum-Bend Paths Among Obstacles

The methods discussed in Section 5.2 move the robot along rectilinear paths. In general, straight-line motion is preferred over turns. In moving the robot from one point to another, we would like to find a path that has as few turns as possible. This problem, planning rectilinear paths with minimum bends among rectilinear obstacles, is called minimum-bend path (MBP). The 3D version of the problem, with orthohedral obstacles, is called 3D-MBP. MBP algorithms have numerous applications. Examples include motion planning for structured environments such as assembly lines, where it is simpler to command the robot to move along three set directions than to require arbitrary movement, and automated machining tasks where controlling a turn is more difficult and error-prone than controlling a straight line. Although the resulting rectilinear plans may be longer than the shortest path in terms of overall distance, the actual execution of such plans may be faster and more robust.

Rectilinear motion planning algorithms are well-suited to applications in planning for lattice-based SR systems. As discussed in Section 5.2, low-level locomotion control is often biased towards straight-line paths. The simple Inchworm Locomotion algorithm is a clear example. Cellular automata-style techniques and locomotion through reconfiguration are other examples. In these applications, efficient planning involves finding a rectilinear path with minimum bends among obstacles.

The general problem of finding a shortest path between two points among obstacles has been extensively studied in many contexts, and with common variations such as the rectilinearity constraint and bend-distance[1] metric we consider in this paper. Optimal solutions exist for MBP in 2D, but much less work has been done with the 3D variant, even though many applications require a 3D solution. For 2D, the optimal $O(e \log e)$ running time [27], where $e$ is the number of obstacle edges, has been achieved by a number of algorithms [60]. However, these do not extend trivially to higher dimensions, and general 3D path planning is hard. Using the Euclidean distance metric, the 3D shortest path problem among obstacles is NP-Hard [67], for example. Adding restrictions on the characteristics of paths and obstacles makes the problem tractable.

Much of the existing work on finding rectilinear paths among rectilinear obstacles[2] is motivated by applications in VLSI wire routing. The number of bends in a path affects resistance, and the total length affects cost. In some models, vertical and horizontal wires are restricted to separate layers, and minimizing connection between layers is desirable.

[1]Many authors use the term *link-distance*.

[2]Rectilinear obstacles have all edges parallel to a coordinate axis.

An excellent survey is given by Lee *et al.* [60]. Lee provides a taxonomy of problem variations along with a categorization of algorithmic techniques. In addition to MBP, *SMBP* is defined as the shortest-minimum-bend-path problem. Of all MBP solutions, the SMBP solution minimizes rectilinear distance. A more recent survey has been written by Maheshwari *et al.* [64].

In this section, we propose an algorithm to solve the 3D minimum-bend-path problem (*3D-MBP*). We describe our 3D-MBP solution, analyze its complexity and present an implementation in simulation. Our algorithm runs in $O(n^2 \log n + n^2 I)$ time, where $n$ is the number of obstacle vertices and $I$ is the maximum number of obstacles intersected by a line parallel to the $x$ axis. This result, is the first published 3D-MBP solution, and is the first to extend the wavefront technique to 3D.

### 5.3.1 2D Algorithms

Two different approaches have dominated rectilinear shortest path algorithms in 2D. In the graph theoretic approach, the method is to build and search a *path-preserving graph* [17]. The graph is constructed to contain sufficient distance information to allow standard graph search algorithms to find solutions. The second technique is the wavefront, or "continuous Dijkstra" approach, where line segments carrying distance information sweep away from the source. Mitchell [66] proposed a 45-degree wavefront algorithm, while Lee's algorithm [59] uses simpler horizontally oriented wavefronts. The horizontal wavefront algorithm is described in more detail in Section 5.3.3.

### 5.3.2 3D Algorithms

Extending these 2D solutions to higher dimensions is difficult. Piatko [80] proves various problems to be NP-Hard, and outlines approximation algorithms for 3D problems among arbitrary polyhedral obstacles. Restriction to orthohedral obstacles, which have all edges parallel to one of the coordinate axes, is useful. Mitchell [66] suggested the application of the continuous Dijkstra approach to 3D problems. Choi and Yap [23] propose a solution to 3D-SP (shortest rectilinear path) among “box” shaped obstacles with $O(n^2 \log n)$ running time, where $n$ is the number of “boxes”. Our algorithm is the first exact solution to 3D-MBP with orthohedral obstacles that has been published [33].

### 5.3.3 Horizontal Wavefront Algorithm

In developing our solution to 3D-MBP, we chose to extend Lee’s 2D horizontal wavefront algorithm to 3D. Since Lee’s algorithm is a component of our solution, we summarize it here. Given a set of rectilinear obstacles, a source $s$ and a destination $d$, the problem is to find a collision free path from $s$ to $d$ with a minimum number of bends. This is done by sweeping through the entire environment with *wavefronts*. A wavefront is defined as a horizontal line-segment that extends to obstacles on both sides, and carries distance information specifying an upper bound on the length of the MBP from $s$ to every point on the line segment. Wavefronts sweep away from $s$, stopping at *event points*, which are defined by obstacle vertices. Certain wavefront operations take place at event points to allow wavefronts to split, extend and sweep around obstacles, and merge with other wavefronts. Figure 5.7 illustrates wavefronts and enumerates cases for wavefront operations. A segment tree data structure is used to maintain distance information in the form of partitions on the wavefront.

Preprocessing is performed to allow event point calculation in constant time. The algorithm begins by inserting two wavefronts at the source, one sweeping up and the other sweeping down. Wavefronts are stored in a priority queue which is sorted by the shortest distance on the wavefront. The algorithm pops a wavefront from the front of the queue,

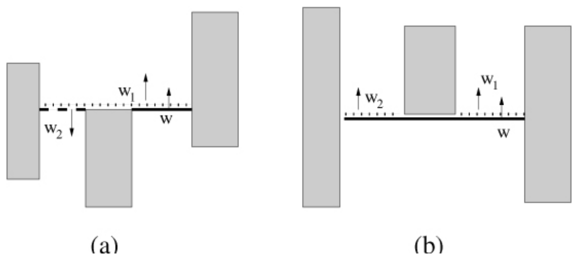


Figure 5.7: 2D wavefront operations, adapted from [59]. (a) Extend and wrap. Parent wavefront $w$ extends to $w_1$, and wraps around to create $w_2$. (b) Split. Wavefront $w$ is replaced by $w_1$ and $w_2$.

sweeps it to the next event point, applies the appropriate operations, and inserts any child wavefronts back into the queue. When a wavefront hits the destination, and all remaining wavefronts have higher distance labels, a path is reconstructed by following parent-pointers. With $e$ obstacle edges, preprocessing takes $O(e \log e)$ time. Merge operations require $O(e \log e)$ time amortized, and Lee proves that a constant number of wavefronts pass through a given obstacle vertex, for an optimal total running time of $O(e \log e)$ [27].

Note that with rectilinear obstacles in 2D, a simple cell decomposition can be constructed by dividing the space into "rows" at each obstacle vertex $y$-coordinate, and "columns" at each obstacle vertex $x$-coordinate. This cell decomposition contains $O(n^2)$ cells and can be searched using breadth-first search (BFS) to produce a solution to MBP in $O(n^2)$ time, which is slower than the optimal $O(e \log e)$ running time. This "naive" solution is easily extended to 3D, and solves 3D-MBP in $O(n^3)$ time. We are interested in a faster solution, so obviously we need to avoid explicitly calculating the entire decomposition while still exploring the entire volume of the problem. The horizontal wavefront algorithm accomplishes this by exploring multiple cells per step. Our results exploit this property in 3D.

### 5.3.4 The 3D-MBP Algorithm

We now present our solution to 3D-MBP. The goal is to find a rectilinear path with fewest bends from a source point $s$ to a destination $d$ in $\Re^3$ among obstacles. A rectilinear path is a path composed of a series of connected line segments, each parallel to one of the coordinate axes. In this case, we consider only orthohedral obstacles, which have all edges parallel to one of the coordinate axes.

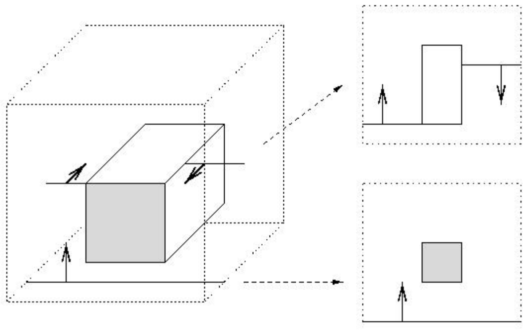


Figure 5.8: Exploring a 3D problem using 2D wavefronts. Wavefronts sweep through a 2D slice of the overall problem, as illustrated in the 2D detail views at the right of the figure.

Figure 5.8 illustrates the main idea of our solution. We explore the 3D problem using linear wavefronts that sweep in one of two orientations. As in the 2D algorithm, wave-

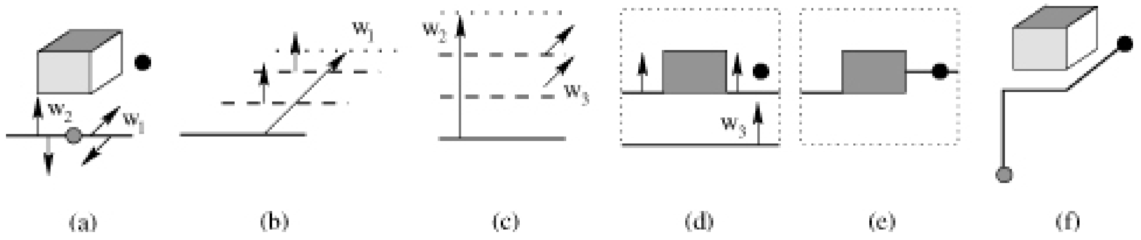


Figure 5.9: Computing a simple 3D path. In (a), four wavefronts are inserted (one in each possible direction). Part (b) shows $w_1$ being dragged, along with the resulting child wavefronts. Similarly, $w_2$ is dragged in (c). Parts (d) and (e) are 2D detail views of $w_3$ as it is dragged. The final path is shown in (f).

fronts spawn children according to certain rules, and eventually the entire problem space is searched.

Our algorithm extends Lee's 2D algorithm using the "continuous Dijkstra" approach. The basic idea is to maintain distance labels on obstacle vertices such that a label represents an upper bound on the length of a shortest path from the source to the labeled vertex. In each iteration we expand the labeled region by exploring a small area away from a vertex with a minimum distance label. In this extension of the horizontal wavefront technique, a wavefront is still a line segment but may travel in one of two possible directions and spawns child wavefronts in both directions accordingly. When a minimum-distance wavefront reaches the destination, we recreate the path by following predecessor pointers back to the source.

Conceptually, the wavefronts explore the surfaces of cells in an exact 3D cell decomposition of the environment, instead of explicitly exploring cell interiors. It is easy to see that all paths through a given series of cells are homotopic, so paths through the faces can be constructed that are just as good (in number of bends) as any path through the cell volumes. To explore all cell faces, we first take the 3D problem and create a set of 2D problems parallel to the $xy$-plane at the $z$ coordinates of all obstacle vertices. Another set of 2D problems are created that are aligned with the $xz$-plane at obstacle $y$ coordinates. Wavefronts can then be thought of as "living" in one of these 2D problems. Unlike in 2D, the sweep operation will not only generate child wavefronts in its own 2D problem, but may also spawn children in intersecting problems.

The main computation is given in Algorithm 22. To begin, we insert four wavefronts at the source: an up-going $xy$, a down-going $xy$, an up-going $xz$ and a down-going $xz$. There is no need to explicitly sweep wavefronts in the third dimension since the other wavefronts have maximal length, thereby implicitly covering this case. From there we remove a wavefront from the priority queue of existing wavefronts based on minimum bend-value, drag it, and insert all newly generated wavefronts as described below. When a wavefront hits the destination, we continue until all remaining wavefronts have minimum bend-values at least as large as the best.

A simple example is shown in Figure 5.9. In (a), the initial four wavefronts are inserted. Wavefront $w_1$ is taken from the queue and dragged in (b), generating child wavefronts as described below. Part (c) shows wavefront $w_2$ being dragged, also generating new wavefronts. Next, other wavefronts with zero minimum bends would be dragged, but for sake of illustration we skip directly to $w_3$, shown in (d) and (e) in a 2D detail. In (e), the destination is reached. The algorithm continues until all wavefronts have at least two bends (the best so far), then the path is generated, shown in (f).

As in 2D, each wavefront is a partitioned maximal-length line segment that stores the “length” (number of bends) of the shortest path from the source to any point on the line segment. Each wavefront orientation keeps its own bend-counting semantics; in other words, $xy$ wavefronts measure paths with the final segment pointing in the $y$ direction, and $xz$ problems measure paths ending in the $z$ direction. As in the standard 2D problem, this invariant is maintained until a wavefront hits the destination, when we then need to calculate which approach orientation is best. This can be done by simply adding one to the minimum partition label on the wavefront (for an approach from the $x$ direction) and comparing with the value of the partition that contains the destination point (approach from $y$ or $z$).

To drag a wavefront in 3D, we first apply all the 2D wavefront operations as usual within the wavefront’s local 2D problem. However, the wavefront now generates additional wavefronts in the other plane. As an $xy$ wavefront is dragged, it potentially sweeps past the $y$ locations of a number of $xz$ problems. This will generate both an up- and down-going wavefront in each of the $xz$ problems the wavefront cuts through. When an $xy$ wavefront is dragged from $y_1$ to $y_2$, for each obstacle vertex $v$ with $y$ coordinate

$v_y$, where $y_1 < v_y <= y_2$, wavefronts are added to the $xz$ problem at $v_y$, as shown in Figure 5.10. The $x$ coordinates of the new wavefront will equal those of the original. Thinking in terms of the $xz$ problem at $v_y$, the new wavefronts are "popping up" into the problem, not necessarily at an event point in that problem. So to find the $z$ coordinate of the new wavefronts, we use the pointers linked in preprocessing to find the nearest event points. Therefore we always generate wavefronts that are at valid event points in their respective 2D problems. All bend values are simply incremented by one, since the paths are now being measured with the final segment oriented in the $z$ direction instead of the $y$ direction. Propagation of $xz$ wavefronts is analogous; they generate $xy$ waves at every vertex $z$ coordinate they sweep through.

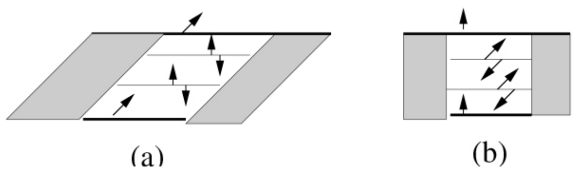


Figure 5.10: Generating new child waves. An $xy$ wavefront generating $xz$ wavefronts is shown in (a), while (b) shows an $xz$ wavefront generating $xy$ wavefronts.

Extra preprocessing is also required in 3D. The preprocessing pseudocode is listed in Algorithm 23. This step has two objectives: generating the canonical set of 2D problems, and linking the problems together. For convenience, we assume a bounding cube around the problem, with all obstacles and start/end points inside the cube. First a plane (parallel with the $xy$-plane) is swept through the problem, stopping at $z$ coordinates of obstacle vertices. We maintain a data structure of currently intersected obstacles, and add/remove obstacles as necessary, such that at each event point we can generate a 2D problem by ignoring the $z$ coordinates of the vertices in the data structure. Within this 2D problem we then perform 2D preprocessing [59]. This step uses a sweep line and links each vertex to *projection points* — points on the nearest obstacle on either side in the $x$ dimension. The projection points therefore define the width of a wavefront passing through that ver-

**Algorithm 22** 3D-MBP

```
1: Preprocess (Algorithm 23)
2: Let $Q$ be a priority queue sorted by minimum bend distance
3: Insert four waves into $Q$ at $s$: up $xy$, down $xy$, up $xz$, down $xz$.
4: while Q is not empty and $d$ is unmarked do
5:    $w = DeleteMin(Q)$
6:    Drag $w$ from event point $p$ to nearest event point $p'$
7:    Apply 2D wavefront operations to $w$
8:    Mark $p'$
9:    Insert into $Q$ counter-oriented child wavefronts $w_i$ at intersection points between $p$ and $p'$,
      in both UP and DOWN directions
10: if $d$ is unmarked then
11:    Fail
12: else
13:    let $b_{min} =$ distance($d$)
14:    while Q is not empty and min($Q$) $< b_{min}$ do
15:       Sweep as in lines 5-9
16:       If wavefront reaches $d$, update $b_{min}$
17: Generate path $P$ by following pointers from $d$ to $s$
18: Output $b_{min}$ and $P$
```

tex. We add pointers such that projection points may be obtained from a vertex and vice versa in constant time. The projection points are stored in linked lists that correspond to obstacle edges, so that from a projection point, the next event point can be obtained also in constant time by following the linked list. In addition to this standard 2D preprocessing, we also compute intersections with $xz$ problem locations between each pair of event points, and add these to a hash table for later constant time retrieval. Each intersection point is linked to the nearest projection point. We then repeat this process with the $xz$-plane to generate the canonical problem set. Intersection points are retrieved from the hash table as necessary and linked to the nearest $xz$ projection points. The time required to build these $2n$ problems is $O(n \log n)$ for the plane sweep operations, plus $O(n \log n)$ at each of $n$ event points. The total is $2n(2n \log n) = O(n^2 \log n)$ time and space.

**Algorithm 23** 3D-preprocessing

```
1: Generate set of all obstacle vertices $V$
2: sort $V$ by $z$ coordinate
3: for each $v \in V$ with unique $z$ do
4:     Build 2D $xy$ problem
5: sort $V$ by $y$ coordinate
6: for each $v \in V$ with unique $y$ do
7:     Build 2D $xz$ problem
```

### 5.3.5 Analysis

We now prove the correctness and running time of our algorithm.

**Theorem 7.** *The algorithm 3D-MBP correctly finds a minimum-bend path from $s$ to $d$.*

*Proof.* Let $P$ be a shortest (minimum-bend) path from $s$ to $d$. Now construct an equivalent path $P'$ as follows. Find the first $x$- or $y$-segment in $P$ and push it down (negative $z$), along with all adjacent $x$- and $y$-segments, until a segment hits an obstacle. Some segment must hit an obstacle during this pushing, otherwise we could reduce the number of turns, contradicting the assumption that $P$ is a shortest path. Continue by pushing the next $x$- or $y$-segment(s) in $P$ and so on. $P'$ is also a shortest path, since we added no additional turns in pushing, and it has all $x$- and $y$-segments in the same $xy$-plane as an obstacle face, connected by $z$-segments. Further modify $P'$ by dragging all $x$- and $z$-segments back until they hit an obstacle. For adjacent $x$- and $z$-segments, drag the segments as a unit. Now consider a line segment parallel to the $x$-axis starting from $s$ and moving along $P'$, stopping at each $x$-segment. The line segment drags in the $y$-direction only in the same plane as some obstacle vertex, which is also a 2D subproblem in our canonical set. The line segment moves in the $z$-direction only when in the same plane as some obstacle vertex, which also is included as a subproblem. Since our algorithm generates $x - y$ segments only at obstacle faces, and $z$ segments only at obstacle faces, and attempts to generate $x - y$ or $z$ segments at all possible obstacle faces, it can generate path $P'$.

Throughout the algorithm, the wavefronts properly maintain bend information. In 2D, this has already been proven [59]. What remains is to prove that bend information is properly maintained when crossing between 2D problems. This is easy to see since we simply add one bend when going from $y$-directed to $z$-directed segments, and vice versa.

Waves are dragged in best-first order, and the algorithm continues until all waves have a cost of at least $b_{min}$, the cost of the shortest path. Therefore it is not possible to have a path shorter than $b_{min}$, since all remaining waves already have at least $b_{min}$ bends and path length is nondecreasing during each sweep.

If no path is found, waves eventually explore all free space. Therefore a path is always found if one exists, and this path is shortest. $\square$

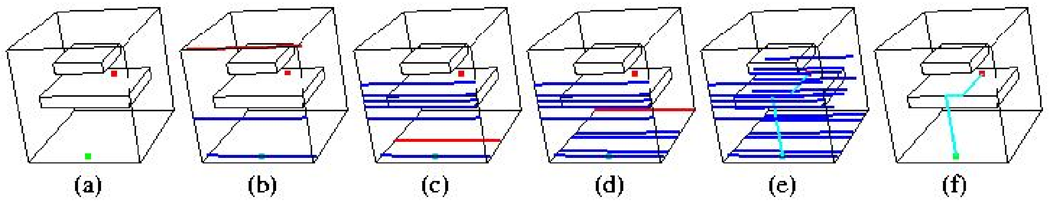


Figure 5.11: 3D-MBP implementation. In (b), the first $xz$ wavefront sweeps up and generates a child $xy$ wavefront on the front face of the bounding cube. The remaining children of this wavefront are added in (c). Part (d) shows the results of dragging the initial $xy$ wavefront. All wavefronts remaining in the queue at algorithm termination are depicted in (e), and the shortest path is given in (f).

**Theorem 8.** *The algorithm 3D-MBP runs in $O(n^2 \log n + n^2 I)$ time, where $n$ is the number of obstacle vertices, and $I$ is the maximum number of obstacles intersected by a line parallel to the $x$ axis.*

*Proof.* The preprocessing step has $O(n)$ event points, with $O(n \log n)$ time processing at each point, or $O(n^2 \log n)$ time overall.

We have $2n$ 2D problems, each of which requires $O(n \log n)$ time. Each dragging step does the same operations as in the standard 2D algorithm, plus the added wavefronts.

Charge the cost of the new wavefront to the 2D problem into which it is inserted, so the cost of each 2D problem is the $O(n \log n)$ plus the cost of adding wavefronts that "pop in" to it. The number of pops is bounded by the number of possible pop locations, which is $O(n)$ 2D problems times $nI$, where $I$ is the number of times an $x$ oriented line can be split by obstacles. Total running time is $O(n^2 \log n + n^2 I)$. The worst case scenario occurs only when there exists a line parallel to the $x$-axis that intersects all obstacles. For this class of problems, the running time is $O(n^3)$ and thus it is preferable to use a simpler algorithm such as searching the 3D cell decomposition. $\square$

### 5.3.6 Implementation

We implemented our 3D-MBP algorithm using the C++ programming language and built a graphical simulation using OpenGL graphics libraries. Figure 5.11 illustrates sample output from the simulator. The implementation takes as input a set of orthohedral obstacles, a start point and a destination point. The simulator renders the obstacles inside an artificial bounding cube and marks the start and destination points. As a wavefront is swept, it appears as a moving line segment in the simulation, and wavefronts in the priority queue are shown as static line segments. When a wavefront reaches the destination, its path is recovered and drawn. The program outputs the MBP, which can be subsequently input to other applications.

### 5.3.7 Discussion

In this section we presented a solution to the 3D-MBP problem, based on an extension of an existing wavefront algorithm for 2D-MBP. Our algorithm correctly solves 3D-MBP in $O(n^2 \log n)$ for most inputs. Although our result fails to achieve a $O(n^2 \log n)$ worst case running time for inputs where all obstacles can be intersected by a line parallel to the $x$-axis, there exists a solution to 3D-SP that does have this bound and we conjecture that a faster solution to 3D-MBP is also possible. An optimal running time for either 3D variant remains an open problem.

So far we have considered only MBP, but in 2D the SMBP solution only requires augmenting the wavefront data structure and adding a secondary sort criterion to the priority queue. Of course, if SMBP is solved then SP can also be solved by ignoring the primary metric (thinking of all paths as having equal bends, so just secondary metric is minimized). It should therefore be possible to extend our algorithm to solve 3D-SMBP and 3D-SP as well, both among orthohedral obstacles.

Our motivating application for the 3D-MBP algorithm is motion planning for SR systems. For motion along smooth surfaces, 2D planning algorithms suffice. One of the main promises of SR systems, however, is the ability to navigate over rough terrain. Here, 3D planning is useful.

The 3D-MBP solution we presented is a centralized algorithm. As in other early motion planning solutions, complete knowledge of the environment is assumed. Also, obstacles must be orthohedral. This actually reduces the necessary environmental information since we do not need to know the exact obstacle geometry; a bounding box representation is enough. Orthohedral obstacles can be true obstacle shapes or bounding boxes around more complex obstacles. Unfortunately, in real-world scenarios, knowledge about the environment is only available via sensors. Centralized motion planning approaches are not applicable in this case.

For module trajectory planning within the robot, complete knowledge of the "terrain" *is* directly available. We explore this idea as part of the self-repair task in the next section.

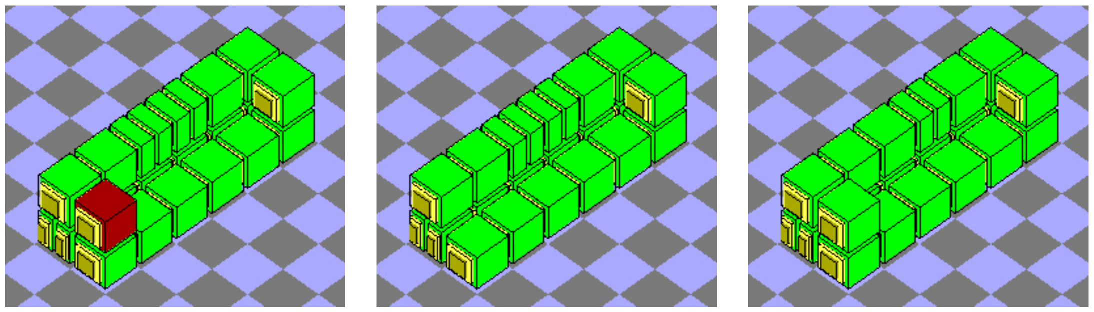

(a) (b) (c)

Figure 5.12: The left image shows a couch configuration with a defective module highlighted. The middle figure shows that couch after the bad module was ejected. The right figure shows the repaired couch. One of the compressed modules was used to fill in the gap.

## 5.4 Self-Repair

The built-in redundancy of SR systems leads to interesting fault-tolerance and self-repair properties. If an arbitrary part of a fixed-architecture robot fails, the robot cannot usually perform self-repair; a human or a different robot must perform the task. Intuitively, a self-repair system must have at least two qualities: the ability to self-modify, and the availability of new parts or resources to fix broken parts. Most extant systems lack these properties. However, self-repair behavior is prevalent in biological systems, the most notable being human tissue repair [65]. A system with failed modules is also a case of a heterogeneous system. Self-repair research is thus an investigation of another example of heterogeneity.

A self-repairing robot can, in the event of component failure, restore itself to original operation without external intervention. By carrying some additional modules, the robot may excise the failed part and replace it with the spare units. When the modules comprising the robot are identical and can move in general ways relative to one another, it is possible to detect and eliminate defective modules, while replacing their functionality in the system.

Yoshida et al [109] introduced the concept of a self-repair robot and presented a simulated-annealing algorithm for this operation. We focus instead on geometric motion planning algorithms for self-repairing robots consisting of Crystalline modules [82]. We begin by describing our approach to homogeneous self-reconfiguring robot systems, using the Crystal robot. We focus on algorithms that permit a robot to detect a failed module, eject it, and replace it with one of the extra modules on the body, so as to repair the arm-rest of the couch in Figure 5.12, for instance. The Crystal is a good candidate for studying self-repair, as it can carry redundant modules in the robot's structure—in other words, spare parts.

In many lattice-based systems, modules can travel only along rectilinear paths. Translations are easier than changing the direction of movement, therefore we can use rectilinear minimum-bend path planning to generate good module trajectory plans. This provides an appropriate situation in which to employ the 3D-MBP algorithm.

### 5.4.1 Self-Repair Approach

We start by observing that the process of self-repair consists of three phases: (1) detect failure, (2) eject the failed module, and (3) replace the failed module. We have not yet addressed detecting module failure, as in general it is highly dependent on the implementation of the system. There are a number of possible approaches, however, such as polling by a central unit or nearest-neighbor testing. Biological systems, such as human skin cells, use a "cry for help" method where the failed unit (damaged cell) sends a broadcast message by releasing its contents into the microenvironment as it dies [65].

We present algorithms for phases (2) and (3) of self-repair. Our solution assumes Crystal robots with a known failed module. Without loss of generality, we focus our discussion on the case when only one module fails in the system. Our algorithms can be iterated to cope with multiple module failures.

A bad module may not be able to move under its own power, so “live” modules in the system should manipulate the “dead” module into position for ejection. The strategy we pursue is to move the dead module to a place where it will simply fall off of the robot when released by any attached live modules. Our solution (1) identifies all the locations on the surface of the robot from where it is possible to release the bad module; (2) computes a shortest path to that region; and (3) uses a gait to propel the bad module along the shorted path.

### 5.4.2 2D Planning

Motion planning in the Crystal involves moving a module from one position to another, and exploits virtual module relocation. Therefore, path planning reduces to finding a rectilinear path through the robot structure. Each segment of the path can be executed in constant time, so an efficient motion plan requires a rectilinear path of minimum bends. Replacing a failed module (filling a “hole” in the structure) can be solved using virtual module relocation. To eject a failed module, this planning technique can not be used directly since here a particular module must be actually pushed (or pulled) to a position on the surface of the robot. However, pushing gaits to move the failed module, such as the one shown in Figure 5.13, also exhibit the property that turns are more expensive than straight line motion. Finding a minimum-bend path is therefore useful in both steps. An MBP problem is constructed by modeling the source and destination points in module coordinates, and holes and concavities in the structure as obstacles. Then, given a path, motion planning can be accomplished by iterating the appropriate gait over the path.

One such gait is shown in Figure 5.14. We use an SMBP algorithm for this particular gait, since straight line motion is easier to plan than turns but the gait actually requires

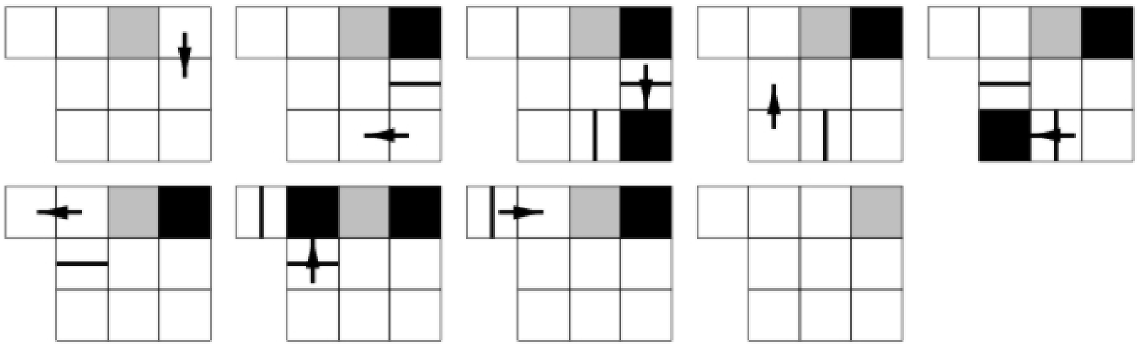


Figure 5.13: A tight gait for propagating a failed module.

constant time per unit distance. Once the dead module is ejected, we use a SMBP to plan the replacement motion, since module relocation is linear in the number of turns in the path.

The planning algorithm is summarized as follows:

---

1. Using the SMBP algorithm, compute a path from the dead unit to a point on the outside surface.
2. Iterate the pushing gait for each step in the path, eventually ejecting the module.
3. Compute the SMBP from the ejection location to the closest redundant module.
4. Use fast module relocation, following the computed path, to virtually relocate the replacement module to the ejection location.

---

**Implementation**

We conducted self-repair experiments using the *xtalsim* simulator [83]. Xtalsim was created as part of the original Crystalline robot development effort and was used to explore the self-reconfiguring abilities of the robots. The simulator reads a file-based specification of a robot and motions of individual modules. It verifies the validity of the requested motions and displays a 3D animation of the robot in motion.

The existing xtalsim version required modification to support self-repair functionality. We extended the software to allow specification of dead modules along with graphical dis-

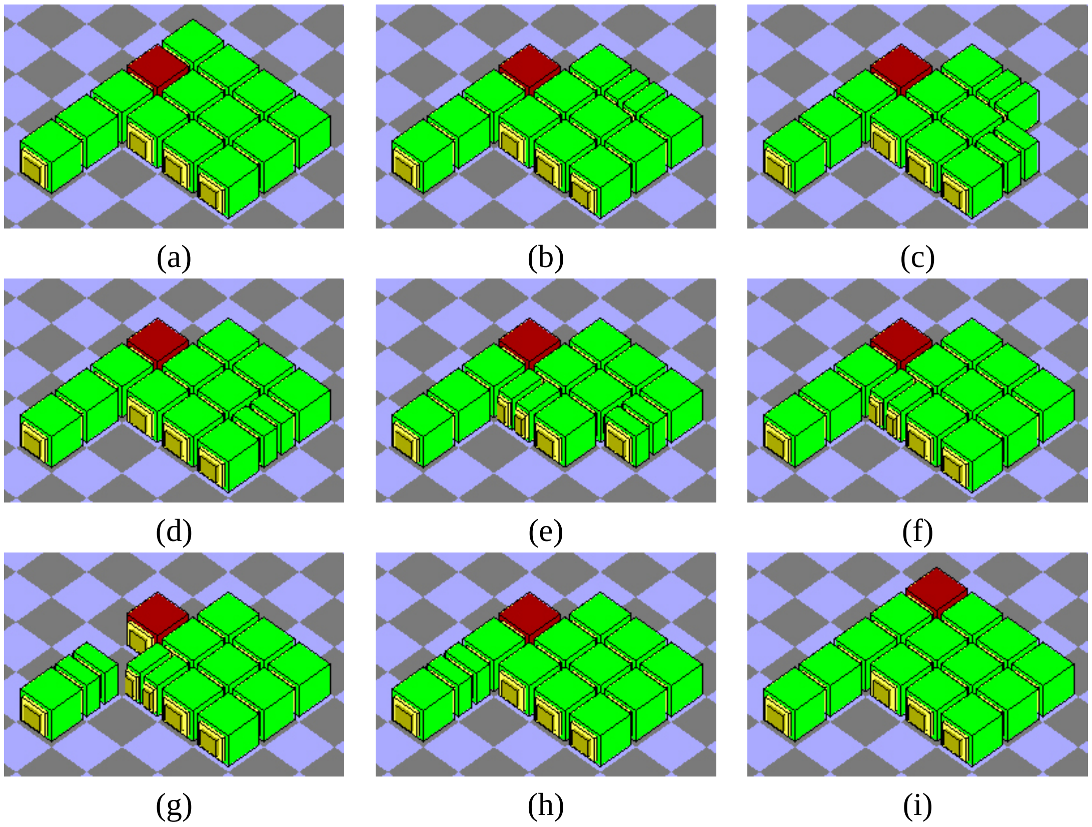


Figure 5.14: Simulation displaying sample gait to push dead module.

tinction in the animation. Also, we added removal features to simulate ejected modules. Figure 5.14 depicts snapshots from the simulator output. Our experiments demonstrate pushing gaits to move a disabled module, and module relocation to fill holes. We traverse the computed shortest path by iterating the pushing gait algorithms for straight line movement and for turns.

### 5.4.3 3D Planning

This motion planning technique easily extends to 3D given an efficient shortest path algorithm. A 3D rectilinear path can be decomposed into a sequence of 2D turns (not all of which are in the same plane). Therefore, given a 3D rectilinear path, a motion plan can be constructed by iterating the appropriate module gait over each path segment. Note

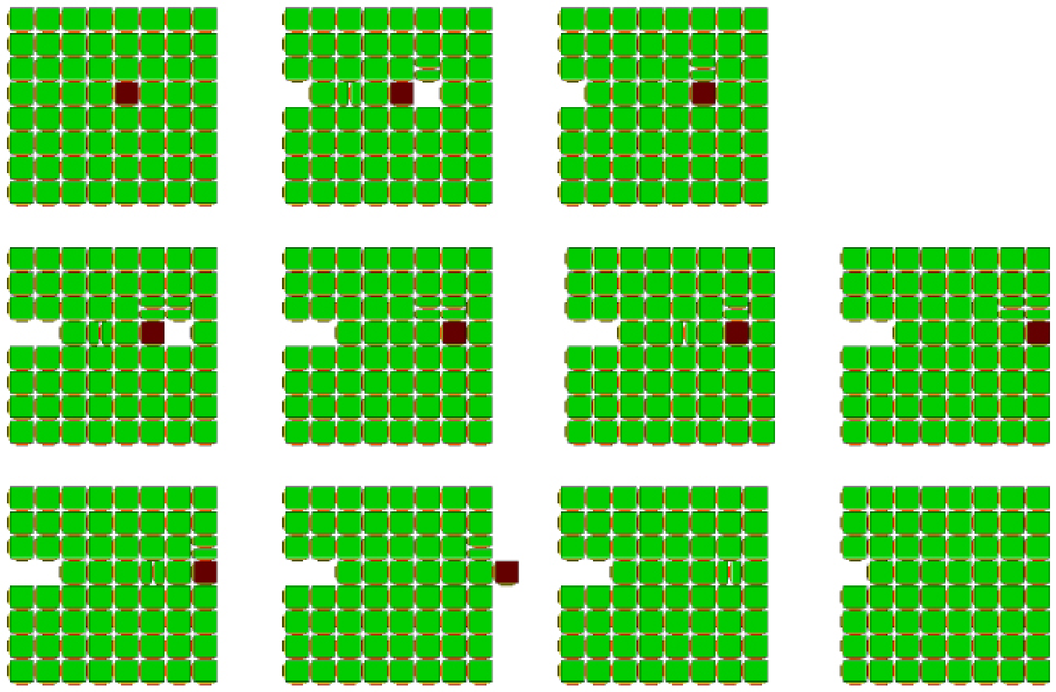


Figure 5.15: Pushing a failed module along a line. After initial setup, each push step requires four contractions/expansions. The second row illustrates two such steps. Finally, modules along the path are "reset" by shifting left.

that pushing gaits require a minimum amount of supporting structure, but we can build this into the path planning problem by growing the obstacles (holes in the structure) by the required amount. This ensures that any path returned by the algorithm is feasible. We use the 3D-MBP algorithm (Algorithm 22) to compute a path, and iterate a pushing gait along this path to eject the module. Finally, we use virtual module relocation to fill the gap in the structure left by the ejected module.

### Implementation

We have conducted experiments for 3D self-repair in Crystal robots in simulation. Output from our 3D-MBP implementation, in the form of a rectilinear path, is fed to a motion planning routine that generates motion primitives. These primitives are input in turn to our Crystal robot simulator that verifies physical feasibility and renders the simulation. Figure 5.15 outlines a pushing gait, and Figure 5.16 illustrates a sample simulation of a Crystal robot executing the pushing gait computed from a path returned by our 3D-MBP algorithm.

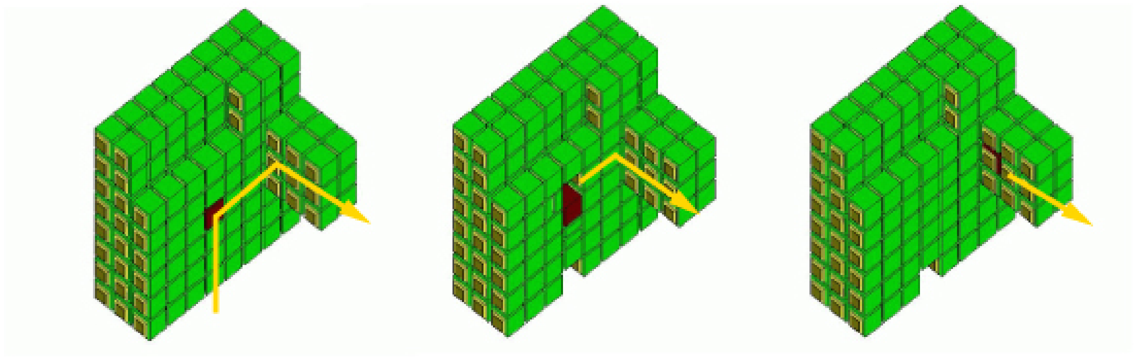


Figure 5.16: Cut-away view of 3D module ejection. The path is shown by the white arrow, and the failed module is darkened. The middle figure is previous to the first turn, and the last figure shows the failed module at the beginning of the final segment.

### 5.4.4 Discussion

In this section, we discussed the general problem of self-repair as an example task of a heterogeneous SR system and proposed a multi-step strategy for implementing self-repair in an SR system. We presented algorithms for ejecting and replacing failed modules. These algorithms analyze the robot structure to determine candidate locations for module ejection, and then compute efficient motion plans. The planning algorithm is capable of minimizing both rectilinear distance and number of turns in a path.

Although the self-repair problem is certainly difficult in existing engineered systems, we have demonstrated that the problem is addressable in experimental systems with special properties. Abstractly, the minimum requirement is the ability to self-modify, and then in the case of the Crystal the problem can be transformed into a motion planning problem for which related algorithmic tools already exist.

The final step needed to completely address our three-part strategy for self-repair is failure detection. We mentioned a selection of possible overall approaches, including pairwise testing, centralized polling, and cry-for-help. Further experience with hardware prototypes will be necessary to identify possible failure patterns in the modules, but pairwise testing seems to be the most feasible model.

# Chapter 6

# Experiments

Algorithmic research in robotics is fundamentally motivated by applications of robots in the real world. We gain insight into and can better characterize theoretical problems with algorithmic studies alone, but ultimately the purpose of this work is to use results embedded in real robots. Accordingly, this thesis includes a sizable and important experimental component. Implementation and evaluation of our algorithmic results in simulation and hardware afford a number of benefits. Experiments demonstrate proof of concept and help in identifying special cases. They also help to guide further research by providing a context in which to validate assumptions and evaluate effectiveness outside the usual method of asymptotic analysis. SR research has not yet matured to the point when it can field useful systems, but any hardware experience we can document joins other valuable first steps in this direction.

We performed experiments both in simulation and in hardware. Both are important in the context of this research. The absence of suitable existing hardware prototypes necessitates simulation work. But, simulation can never fully model the physical system and hardware experimentation is essential.

Our software implementations are constructed within a graphical simulator, *SRSim*, that we designed and developed specifically for SR robots. SRSim is intended to be easily extended to support any module type. Currently, we have implemented Sliding-Cube and Crystal modules. Centralized, multi-threaded decentralized, and cellular automata-style simulations are supported. SRSim is currently being used by other researchers and the source code is included in [9]. We implemented our reconfiguration algorithms in SRSim; other implementations are listed along with the algorithm descriptions in Chapter 5.

The hardware experiments we performed utilized the Crystal robot. This work constitutes the first thorough evaluation of the second Crystal prototype. The modules were physically designed and constructed by other members of the lab. We then developed a software architecture to support message-passing and implemented a number of algorithms. The major effort was to implement the MeltSortGrow sort phase, which is the first hardware experiment with a heterogeneous system.

The overall results indicate that the hybrid approach of provable decentralized algorithms is suitable for distributed SR systems. This approach essentially replaces centralized computation with communication, and the major concern is communication scalability. We found that communication scales sublinearly with the size of the robot, and is orders of magnitude faster than actuation time requirements. The resource bottleneck, therefore, is number of actuations, and scalability concerns with regard to communication volume are unwarranted. A legitimate issue raised by the results is how to realize the fault tolerance promises of SR systems. Our conclusion is that the current Crystal prototype does not perform to a level adequate for addressing this question.

This chapter is divided into three sections. First we describe SRSim in detail, along with implementation results. Then we present our experiments with the Crystal robot, followed by a discussion of lessons learned. Source code described in this section is included in the Appendix section for reference.

## 6.1 Experiments in Simulation

Algorithm implementation in simulation provides graphical visualization of algorithm execution and demonstrates feasibility of algorithm correctness. Commercial simulators do not support features adequate for SR systems, however. We designed and developed a simulation environment called *SRSim* in which to implement our algorithms.

In this section, we detail simulator design and common implementation techniques. Then we describe our implementation of the MeltSortGrow and TunnelSort algorithms.

### 6.1.1 SRSim Simulator

SRSim produces 3D animations in real time of Sliding-Cube and unit-compressible modules. Single-threaded (centralized), multi-threaded (decentralized), and cellular-automata style simulations are supported. The simulation engine handles all rendering tasks and is designed for extensibility. For example, new actuation types can be implemented by simply extending a module class and implementing the required interface. A terrain mapping implementation is also included. This stretches an image over a height map to create a textured landscape with obstacles. The terrain in Figure 6.1 is an example. Source code samples are included in the Appendices.

#### Design

SRSim is written in the Java programming language with the Java3D API for 3D graphics. The simulator is designed as a base class, SRSimBase.class, with a set of module classes. Currently implemented module classes are Sliding-Cube.class and Crystal.class. An algorithm implementation is a class that extends SRSimBase. The base class takes care of all Java3D initialization and draws an empty window. By overriding methods, a landscape can be added or any other modifications can be made, including adding other

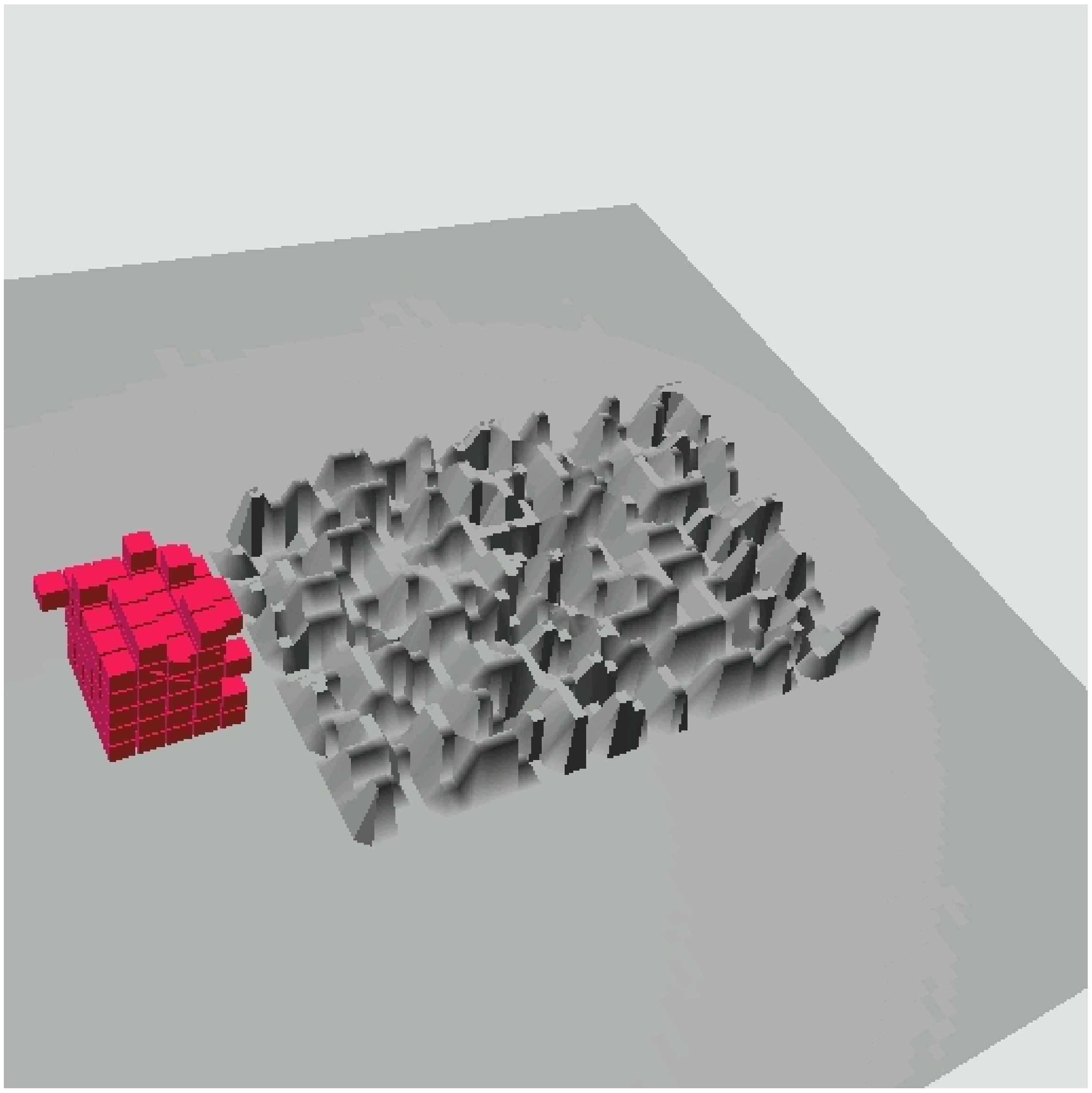


Figure 6.1: Terrain-mapping over obstacles in SRSim. A simple grey image was stretched over a terrain defined by a height map.

background elements. We designed the simulator in this way to maximize extensibility. SRSimBase extends Applet, so simulations can easily be added to web pages in addition to being called as stand-alone executables. The canvas object is actually an instance of NCSA's RecordableCanvas3D [34], so image sequences can be automatically generated for compilation into a movie.

Java3D is a scene graph-based graphics API. A scene graph is a tree data structure that holds objects to be drawn and transformations modifying those objects. The Java3D rendering thread renders the scene by traversing the scene graph. An application using Java3D is freed from all rendering responsibility. Modifications to the scene graph are handled dynamically by Java3D.

### Centralized Simulations

In a centralized simulation, the algorithm implementation runs in a single thread and has exclusive global control over moving modules and maintaining state information. To build a centralized simulation, it is enough to simply create a class that extends SRSimBase, instantiate it, and add the object as the content pane of a window object such as a JFrame. A template is provided in the Appendices (file "MyNewAlgorithm.java").

To perform initialization tasks such as adding user interface elements or background elements, a number of methods in SRSimBase can be overridden. Buttons and other UI objects are added by overriding the `init()` method. Landscapes and backgrounds can be added by overriding `createLand` and `createBackground`, respectively.

Generally, all other tasks (drawing the robot, starting the simulation) are handled in response to button clicks. Unfortunately, the direct implementation of this results in the algorithm code running inside the UI event-handling thread. This prevents any other UI events from being handled until the algorithm code finishes. To remedy this, we execute any long-running code inside a separate thread. We use the SwingWorker class to spawn the new thread for simplicity.

Algorithm implementation is accomplished by interacting with module and configuration data structures. Since graphics rendering is handled by Java3D, the algorithm needs only to interact with the robot data structures themselves and any drawing happens automatically. For example, calling a `move()` method on a module updates the geometry of the module object and also animates the module moving on the screen.

**Decentralized, Multi-Threaded Simulations**

In actual SR systems instantiating the Sliding-Cube model, each module is computationally independent. It is desirable to simulate module execution by running code for each module inside its own thread. Java provides convenient threading support, so we use Java threads. A module object extends the Thread class. Creating a multi-threaded simulation is similar to building a centralized simulation, but code for algorithm implementation resides inside the module thread class instead of in the main application class. The main application adds and starts module thread objects.

The Sliding-Cube model has neighbor-to-neighbor communication only. We implement this using a MessageQueue object belonging to each module, exposing a public method for adding messages. Modules interact with one another by adding messages to message queues. For efficiency, the message queue is implemented such that the thread blocks on receipt of a message. Most threads are sleeping at any given time, so the simulation can support a large number of modules. We observed acceptable performance with greater than 500 threads.

Algorithm-specific code in this simulation style resides in message-handling functions inside the module thread class. The benefit of this scheme is that it is very similar to writing code for the actual hardware. For example, the software architecture we built for the Crystal robot is almost identical. Simulation code can be directly ported to the Crystal hardware by converting the simulator’s java syntax to C.

In some cases, module threads need to interact with the global application. For example, a global physics module can provide information that would be directly sensed by a hardware module, such as local neighborhood configuration. To support this, we add a static reference to the application to the module class.

### Cellular Automata-style Simulations

In the previous simulation styles, essentially only a single module moves at any given time. In cellular automata simulations, however, these moves happen very rapidly and appear to be in parallel. Such simulations need to support a very large number of modules, with continuous geometry updating. Even java threads are too heavy-weight for this situation, so instead we utilize a Java3D class meant specifically for complex animations: the Behavior object.

A Behavior is a scene-graph object that is asynchronously activated in response to some repetitive event, such as the passing of time or animation frames. By adding algorithm-specific code to a Behavior object, we can implement cellular-automata simulations. This frees the application from the task of scheduling updates and allows for very fast geometry updating. In each call to the Behavior, the algorithm can decide which modules to process, depending on which execution model it assumes.

### Sliding-Cube Object

Module classes are implemented as extensions of Java3D graphical objects. They can be added to the scene graph directly. We describe the Sliding-Cube object, but other module implementations are similar. The Crystal class is another example.

The Sliding-Cube object performs two main functions. It represents the low-level module geometry and graphical appearance, and provides an interface for manipulating these properties. Geometry is stored as a set of vertices forming a cube. Java3D uses coordinates measured in meters; the default module size is 0.1 m$^3$. Methods are provided for moving the module to an absolute coordinate or by a relative coordinate offset. These methods modify the underlying geometry directly through the Java3D GeometryUpdater class. This approach is more efficient than using a Transform object for each module in the scene graph since motion happens so frequently. Appearance, such as color and opacity, can also be updated dynamically via a collection of exposed methods.

For multi-threaded simulations, we wrap a Sliding-Cube object inside a class that extends Thread. All application logic is built into this thread object. The thread implements a message loop that blocks on receipt of a message. When a message is received, it passes execution to a message-handler method appropriate to the message type. When the message-handler return, the loop continues.

## Adding Background Elements

SRSimBase provides functionality for added terrain-mapped landscapes. Various versions of the `createTerrainMappedLand` method support the creation of landscape from texture and height map files. From the height map, a triangulated surface is created. Then the image in the texture file is mapped to this surface and rendered by Java3D. Realistic terrains with obstacles are easily built using this method.

Backgrounds, such as adding a blue sky with clouds, can also easily be added by overriding the `createBackground` method. The standard procedure is to map an image onto the inside of a large sphere.

Lighting of the entire scene can be adjust by overriding the `createLights` method. Simple lighting is implemented by default.

### 6.1.2 Implementation of Common Functions

Some common implementation issues are shared by all simulations. The first is how to represent configurations. We use a file-based scheme to store configurations on disk. The file format is a simple text file. Each line of the file defines one module as a tab or space-delimited list of four integers: x coordinate, y coordinate, z coordinate, and type. A configuration file can be read by the application and interpreted as either a start or goal configuration. We consider the start configuration to define a robot. The application reads the file and instantiates a new module object for each line. We normally store a collection of modules as an array inside a simple Sliding-CubeRobot class. A goal configuration is read similarly, although the appearance of the modules is set to wireframe.

This array-based representation is usually augmented with other data structures inside the application. A three-dimensional array, referred to as a grid, can be built to store references to module instances. This provides fast look-up since it allows constant-time access to modules indexed by coordinates, but the grid must be big enough to hold the robot's entire workspace. Memory requirements quickly become prohibitively large. An alternative is to use a graph-based representation. A connectivity graph is constructed based on the module array. Both data structures must be maintained during module motion. For centralized implementations, either data structure is possible. However, in decentralized implementations only the graph-based is allowable since each module can only directly access its immediate neighbors.

These data structures are used to perform low-level trajectory planning for a single module. A module at a given location can execute the Sliding-Cube's two motion primitives, translation and convex transition, determined by its local configuration. By examining the local neighborhood, a list of possible motions can be constructed. This defines a successor function for use in search algorithms. For example, using the grid representation, a function would take a grid location as input and would output a list of child coordinates representing the locations reachable by a module at the input location. This function can be used by a BFS or DFS implementation to search for a path from a module to a point. The path, stored as an ordered list of points, is executed by the module by successively moving to each waypoint.

For a decentralized simulation, the concept is equivalent but the implementation is more complex. A given module has immediate access to its local neighborhood, so it can build a list of potential motions directly. But, path planning requires knowledge of a greater neighborhood. We address this using message-passing as follows. Any reachable location is adjacent to some module. Therefore, all possible motions exist within the local neighborhood of some module. For example, consider a module $m$ that executes a translation. After the translation, $m$ is now adjacent to neighbor $m_1$. Further motions can be computed by examining $m_1$'s neighborhood. Using message-passing, $m$ can query $m_1$ to obtain this list without actually executing the translation. Continuing in this way, $m_1$ can query its neighbors for another list of reachable locations. A search algorithm can thus be constructed that effectively searches the free space on the surface of modules. Note, however, that this does not mean that a module is marked "visited" once a search message reaches it. What is actually visited is the free space adjacent to the module. In graph terms, each module is split into a number of nodes - one for each free location in its local neighborhood. Extra bookkeeping is required but the entire free space of the robot can be searched using this technique. We use DFS for ease of implementation.

Likewise, we can search for a path from a free location to a mobile module. The only change is that the search terminates when it reaches a mobile module. One step is added to the search that takes a given free location and builds a list of modules that can reach this spot. This is simply a reverse of the previous successor function.

To test whether a module is mobile, we must decide whether removing the module disconnects the robot. A module that would cause a disconnection is an articulation point in the module connectivity graph. We use the standard DFS-based algorithm for finding and labeling articulation points. With the graph-based representation, we can execute the articulation point algorithm immediately. For the grid representation, we must first build a graph and then proceed. In decentralized simulations, this is easily implemented using message-passing.

Message-passing that routes a message according to a post-order traversal of the connectivity graph is commonly used, both for simulated broadcast and for search algorithms. We assume the connection topology to be fixed during message propagation, so the only issue is how to decide whether a given module has already received a given message. We use a unique message ID for each message, but other techniques (such as maintaining parent pointers) are possible. When a new message arrives at a module, it first tests whether the message has already been received. If so, it sends a return message back to the sender. Otherwise, it sends the message to one of its neighbors. Eventually, a return message will arrive from that neighbor, and it can send the message to a different neighbor, and so on. When finished, it sends a return message back to the original sender. This process can be thought of as distributed, asynchronous DFS.

### 6.1.3 MeltSortGrow

This section describes our implementation of our MeltSortGrow algorithm (see Section 4.1) using SRSim. We discuss centralized and decentralized versions, and also experiments with randomly generated start and goal configurations.

**Centralized MSG**

The algorithm begins by initializing data structures. Start and goal configurations are read from files and initialized as described earlier in Section 6.1.2. The start configuration is drawn as opaque modules and the goal is not drawn. We validate the configurations by checking for connectedness and equal module counts. This implementation uses a grid representation so we initialize the grid by storing module references in the appropriate array locations.

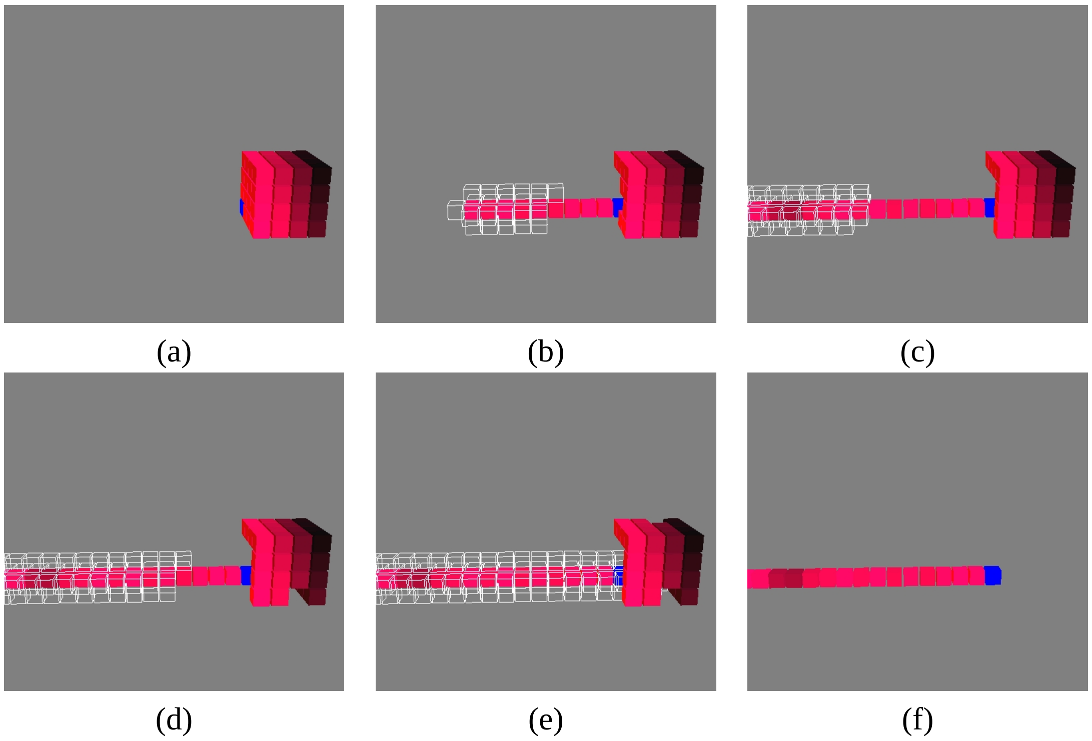


Figure 6.2: Screenshots from the melt phase of the MeltSortGrow algorithm implemented in SRSim.

The melt phase begins by iterating through the module list to find the module with minimum coordinate (the root). Path planning is implemented using BFS through the grid as described in Section 6.1.2. Screenshots illustrating this phase are shown in Figure 6.2. The search path is animated using wireframe cubes. We used BFS instead of DFS since

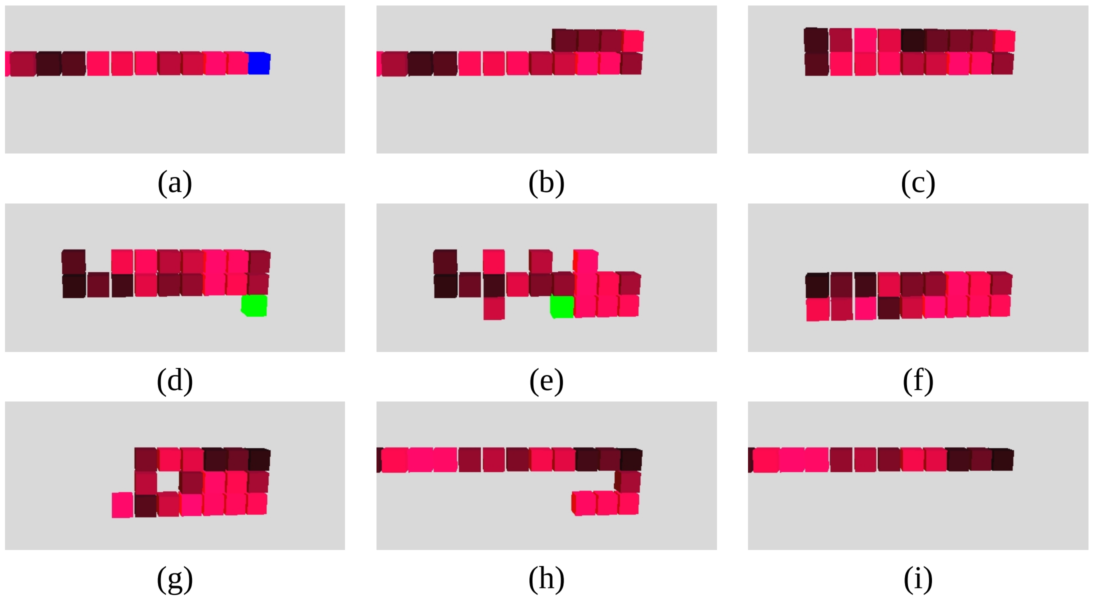


Figure 6.3: Screenshots from the sort phase of the MeltSortGrow algorithm implemented in SRSim.

it finds the shortest paths; DFS is used in the decentralized versions since it is easier to implement with message-passing.

The sort phase requires computation of an assembly order of the goal configuration. To do this, we execute the melt phase on the goal configuration. When a goal module is chosen, we find a module in the current configuration with the same type and give it an integer label. Thus, even though multiple modules can have the same type, this step essentially creates unique types by assigning unique assembly order labels. A label is stored as a property of the module object. Since we use a single grid, we need to delete all module references from the grid, and re-initialize it with goal configuration modules. During disassembly, we build a sorted list of grid locations such that a module with label $i$ originated at the coordinate at element $i$ of the list. We use this information in the grow phase. After disassembly, we again initialize the grid with current module positions. See Figure 6.3 for screenshots of the sort phase.

The grow phase is a similar to the melt phase, but in reverse. See Figure 6.4 for

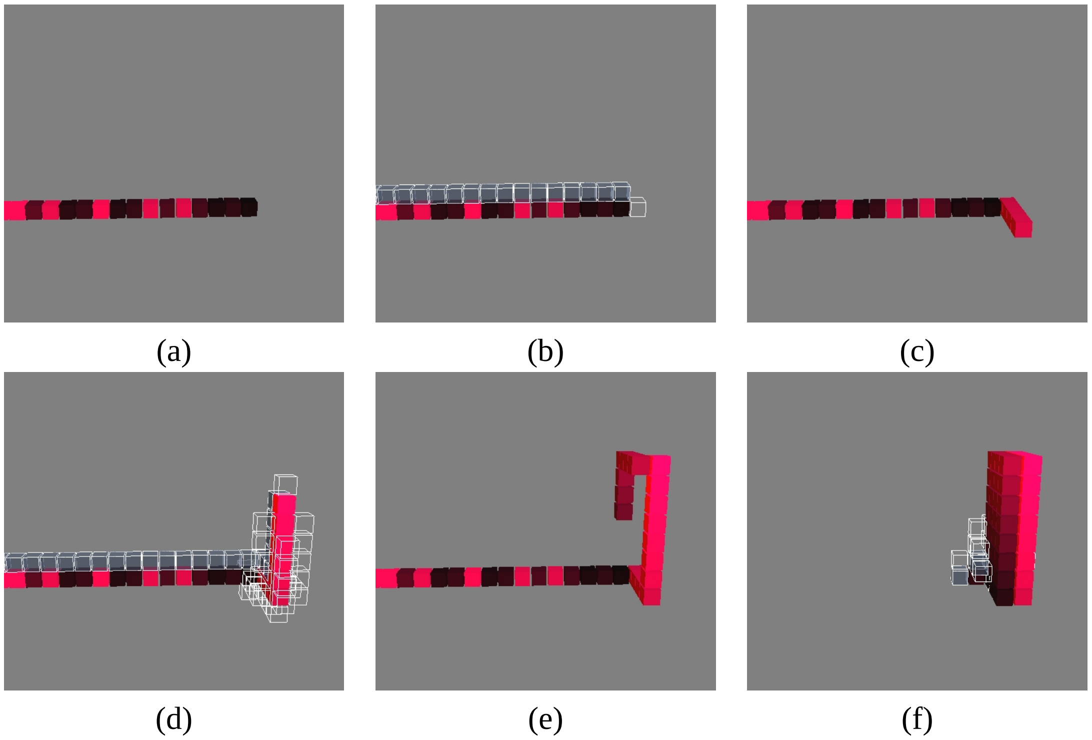


Figure 6.4: Screenshots from the grow phase of the MeltSortGrow algorithm implemented in SRSim.

screenshots.

### Decentralized MSG

The decentralized implementation generally replaces any iteration over the module array or graph search with search via message-passing. All broadcasts and DFS-style searches are implemented using DFS-style message propagation as in Section 6.1.2. Since we assume each module has a copy of the goal configuration, we read in the goal configuration and add a static reference to it inside the module thread class. To begin, module thread objects are created and initialized and the main application sends a *start* message to a random module.

The handler for the *start* message finds the root module by searching for the module with minimum coordinates using DFS-style search. When the search returns, the minimum coordinate is broadcast. The module matching this coordinate becomes the root. This can be thought of as a simple method for leader-election.

The melt and grow phases are equivalent to the centralized implementation except for path searching and added synchronization. Paths are searched using our DFS-style message-based path searching. When a module finds and executes a path, it broadcasts a message signalling completion. This causes the new tail module to start a search, and so on.

Sorting begins when the last module has been melted. At this point, the search for another mobile module fails and the current tail broadcasts a message to begin sorting. Upon receipt of this message, each module computes the disassembly order as in the decentralized version. This module compares its type to each goal module melted during disassembly, and stores a list of order labels. We need to do this because there may be multiple modules of a given type yet a unique sort order is required. We resolve this order after all modules have completed the disassembly computation by passing a message down the intermediate structure that keeps a counter for each module type. As this resolution message is propagated, modules identify their unique label by choosing an element from their label listed indexed by the counter in the message.

### Experiment with Random Configurations

To test our implementations, we built a random test configurations. The configuration generator constructed configurations additively by randomly choosing a module in the current configuration and adding a new module to a random face. We executed both centralized and decentralized implementations with configurations of up to 50 modules in size. Results are summarized in Table 6.1. Since the intermediate configuration is a line shape in all instances, we expect the average path length to be $n/2$ and the total move count to be $4n^2/2 = 2n^2$. As the table shows, the actual data track this function closely.

| Instance | Melt | Sort | Grow | Total |
|---|---|---|---|---|
| **Random shapes,** $n = 11$ | 69 | 132 | 66 | 267 |
| **Random shapes,** $n = 12$ | 84 | 150 | 84 | 318 |
| **Random shapes,** $n = 13$ | 95 | 175 | 99 | 369 |
| **Random shapes,** $n = 14$ | 108 | 202 | 111 | 421 |
| **Random shapes,** $n = 15$ | 124 | 224 | 133 | 481 |
| **Random shapes,** $n = 16$ | 155 | 256 | 145 | 556 |
| **Random shapes,** $n = 17$ | 161 | 286 | 168 | 615 |
| **Random shapes,** $n = 18$ | 183 | 312 | 192 | 687 |
| **Random shapes,** $n = 50$ | 1426 | 2302 | 1421 | 5149 |

Table 6.1: Number of moves produced by MSG for reconfiguration between random shapes of increasing size, broken down by algorithm phase. The number of modules in each instance is denoted by $n$. Each row represents an instance consisting of randomly generated start and goal shapes and unique module types. Data for instances with non-unique types is included in Table 6.3.

### 6.1.4 TunnelSort

The TunnelSort algorithm was described in Section 4.2. Its implementation is straightforward, based on common techniques described in Section 6.1.2. We also include empirical data showing the actual number of moves for each step of the algorithm.

#### Centralized TunnelSort

Initialization is similar to centralized MeltSortGrow. We read in a start configuration and populate an occupancy grid. This algorithm sorts the configuration in-place, so for the goal configuration we store the goal type in the grid along with the reference to the occupying module. To ensure the in-place property, we mark grid elements belonging to the crust and mark all other grid elements as unreachable. This is implemented by iterating over the module list and updating grid entries as necessary.

All path planning is simple BFS-based path search. To store motions of modules displaced during tunneling, we use a stack. Each tunnel module has an associated stack; to return to its original position we repeatedly pop a motion from the stack and execute it until the stack is empty.

Table 6.2 lists empirical data from a number of TS instances. We collected data from a cube shape with a varying number of modules to be swapped. We also collected data from the line example shown in Chapter 4 (Section 4.2.4). The data show that the number of moves is far less than $n$ (the number of modules) for instances with short tunnels and few swaps. The line example has many swaps but extremely short tunnels; the move count is dominated by bridging operations.

We also collected data to directly compare TS and MSG. The results indicate that MSG performs poorly for sorting alone, but is good for sorting plus shape-changing. The TS algorithm is for sorting only, and performs very well. We generated a sequence of instances with increasing heterogeneity and executed both MSG and TS on these instances. All shapes were identical (a 6x6x6 cube), but the number of module types across instances was increased. Results are shown in Table 6.3. The number of swaps executed by TS increases with the number of configuration positions whose module type is different in the start and goal configurations. In other words, TS only swaps modules that are out of place. MSG, however, builds the intermediate structure for all instances. For simple instances, TS produces very few moves compared to MSG. For more complex instances, MSG and TS perform equally well. The instances listed in Table 6.3 are all cube-shaped. This means the average tunnel path length in TS is $\sqrt[3]{n}$, and the total move count is $4k\sqrt[3]{n}$. Because $n = 216$ is fixed in these instances, the total move count for TS in the table increases linearly in $k$. The average path length in MSG is always $n/2$

| **Instance** | **Tunnel 1** | **Tunnel 2** | **Bridging** | **Path 1** | **Path 2** | **Swap Total** |
|---|---|---|---|---|---|---|
| **Cube** $n = 216$ Total Moves= $38$ | 6 | 15 | 0 | 8 | 9 | 38 |
| **Cube** $n = 216$ Total Moves= $74$ | 6 | 6 | 0 | 13 | 13 | 38 |
| | 6 | 15 | 0 | 7 | 8 | 36 |
| **Cube** $n = 216$ Total Moves= $272$ | 1 | 1 | 0 | 4 | 4 | 10 |
| | 1 | 6 | 0 | 12 | 13 | 32 |
| | 6 | 15 | 0 | 13 | 14 | 48 |
| | 6 | 15 | 0 | 7 | 8 | 36 |
| | 6 | 6 | 0 | 8 | 8 | 28 |
| | 6 | 1 | 0 | 13 | 12 | 32 |
| | 1 | 1 | 0 | 2 | 2 | 6 |
| | 1 | 6 | 0 | 12 | 13 | 32 |
| | 6 | 15 | 0 | 13 | 14 | 48 |
| **Line** $n = 14$ Total Moves= $346$ | 2 | 2 | 0 | 12 | 11 | 27 |
| | 2 | 2 | 28 | 10 | 10 | 52 |
| | 2 | 2 | 34 | 10 | 8 | 56 |
| | 2 | 2 | 30 | 8 | 6 | 48 |
| | 2 | 2 | 38 | 6 | 4 | 52 |
| | 2 | 2 | 48 | 4 | 2 | 58 |
| | 2 | 2 | 46 | 2 | 1 | 53 |

Table 6.2: Number of moves produced by TunnelSort for various instances, where $n$ is the number of modules in the instance. Each row of the table shows the number of moves for each step of a single swap operation. Columns labeled “Tunnel 1” and “Tunnel 2” are the number of moves produced in the first and second tunnel of the swap, respectively. Likewise, “Path 1” and “Path 2” are the number of moves in the paths traveled by each module being swapped. The “Bridging” column is the total moves performed in bridging for that swap.

since the intermediate configuration is fixed, so the number of moves is $4n^2/2 = 2n^2$ on average. In the table, we see that the actual move count for MSG with fixed $n$ does not vary appreciably.

With more complex shapes, the average tunnel length for TS approaches $n/2$, and the two algorithms would perform similarly. An option for improving MSG is to replace the line configuration with a cube, and then use TS to sort the cube during the sort phase. This would reduce the average path length for MSG in many instances.

| **Instance** | **Melt** | **Sort** | **Grow** | **MeltSortGrow Total** | **TunnelSort Total** |
|---|---|---|---|---|---|
| **Cube,** $k = 0$ | 25523 | 41398 | 26090 | 93011 | 0 |
| **Cube,** $k = 1$ | 25523 | 41358 | 26090 | 92971 | 18 |
| **Cube,** $k = 8$ | 25523 | 41321 | 26090 | 92934 | 229 |
| **Cube,** $k = 27$ | 25523 | 41105 | 26090 | 92718 | 635 |
| **Cube,** $k = 64$ | 25523 | 40962 | 26090 | 92575 | 1443 |
| **Cube,** $k = 125$ | 25523 | 40314 | 26090 | 91927 | 2954 |
| **Cube,** $k = 216$ | 25523 | 39495 | 26090 | 91108 | 5056 |

Table 6.3: Number of moves produced by TS versus MSG on a sequence of instances with increasing heterogeneity. Each instance has identical shape, a 6x6x6 cube, but the number of unique module types within each instance is varied. The number of positions $k$ where the module type in the start does not match the type in the goal is increased from zero (a homogeneous instance) to $n$, the total number of modules in the configuration. The value $k$ corresponds to the number of swaps performed by TS. The cube shape elicits best-case performance from TS since average path length is $\sqrt[3]{n}$. Average path length for MSG is $n/2$.

## 6.2 Experiments in Hardware

We performed experiments in hardware to demonstrate and evaluate our algorithmic ideas. These experiments use the Crystal, a robot prototype designed and constructed in the Dartmouth Robotics Lab. In the next section, we review the hardware design. Then we present the software infrastructure designed and developed to support our experiments. The first two experiments, goal recognition and Inchworm Locomotion, are implementations of algorithms described in Section 5. The final experiment instantiates the Sort phase of algorithm MeltSortGrow, described in Section 4.

### 6.2.1 Crystal Robot Hardware

The Crystalline Atom (or Crystal) is a 2-D unit-compressible self-reconfiguring modular robot. It actuates by expansion and contraction of individual modules, which together with connection and disconnection allows the robot to change shape as well as locomote. Each module consists of a central core and four faces that move in and out relative to the

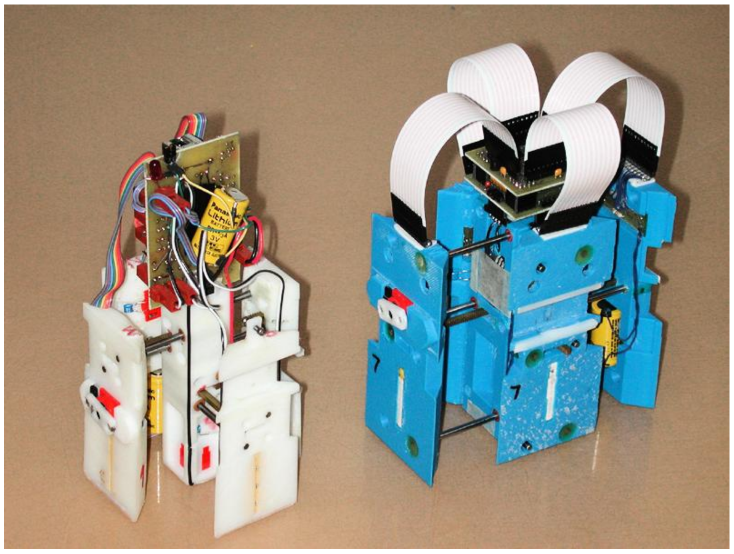


Figure 6.5: The first (left) and second (right) version prototypes of the Crystal robot. The first version is fully expanded while the second is contracted along one axis and expanded along the other.

core to perform expansion. An expanded module is exactly twice the size of a compressed module, which aids in reconfiguration and planning. It is closely related to the Telecube [94] developed at PARC, which is a tethered 3-D unit-compressible system.

The original version of the Crystal [83] had a single degree of freedom for expansion, so that all four faces expanded and contracted together. Each module had its own processor to control actuation, but synchronization was performed through sensing an external beacon — modules had no ability to communicate directly with each other. In the new version of the hardware, the expansion has two degrees of freedom as well as inter-module communications. In fact, in the current version, there is no facility for global communications, so all operations must be performed in a distributed fashion. The old and new prototypes are pictured together for comparison in Figure 6.5. In this section, we describe in detail the hardware components, electronics and fabrication of the new modules, as well as a simple communications infrastructure to allow message-passing between neighboring modules.

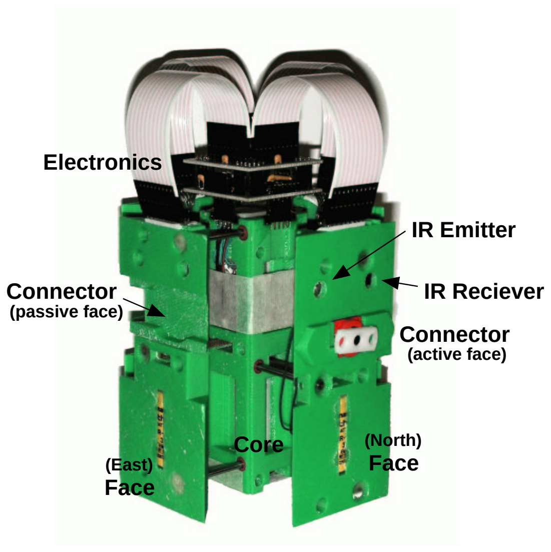


Figure 6.6: A single module of the Crystal robot.

The second-generation Crystal module, shown in Figure 6.6, incorporates several important new features including an additional degree of freedom for actuation, inter-module IR communication capability and sensor inputs. The robot control is now done in a purely distributed fashion. The North/South and East/West faces are independently actuated, so the degrees of freedom increase to four (two for expansion/contraction, and two for connectors on active faces). In addition, several features have been improved over the first version, including stiffening of the linear bearings that align the faces during actuation, more powerful motors to perform actuation, and a faster processor with more memory and I/O capability.

Each atom's on-board electronics provide computation, IR communication, sensor inputs and motor control. The processor is a Hitachi HD64F3644H running at 10 MHz, which includes 32KB of EEPROM for program storage. Analog inputs to this chip are brought out to a small connector so that various sensors (analog or digital in nature) can be attached. Digital outputs control motor drivers that perform actuation, and an additional digital output powers an LED for rudimentary debugging.

Communication is implemented with asynchronous serial over IR components on the Crystal faces. Each module face contains an IR emitter and detector that allow modules to communicate at distances of up to 10 cm. These components are each connected to a dedicated Maxim Max3100 UART in the core, so a unit can talk with all neighboring units essentially simultaneously. The UARTs communicate at 1200 Baud and have an eight-word hardware FIFO. Synchronous serial communication is used between the processor and the UARTs.

Each face is connected to the core circuit board with a flexible ribbon cable, so that the cabling exerts minimal force on the face during expansion and contraction. Four 3V Lithium batteries (one in each face) are connected in series to power the unit, enabling fully untethered operation. Code is downloaded to the processor though a serial interface. The code executes as soon as the unit is powered on.

The expansion/contraction mechanism uses a rack-and-pinion to actuate each pair of module faces. Two MicroMo motors are mounted coaxially in the module core, with pinion gears mounted directly on the motor output shafts. Racks connected to opposing faces mate on opposite sides of a pinion such that each motor drives two faces simultaneously. Shaft encoders built into each motor's housing generate interrupts that allow the processor to detect when the face is fully expanded or contracted. Three additional passive shafts on each face provide greater rigidity during expansion.

Modules attach to each other at their faces, using channel-and-key type connectors. Each module has two faces with active connectors, and two faces with passive connectors. Passive faces simply contain a channel that accepts a bar from an active face. The active face can rotate the bar a quarter-turn, locking the two modules together, and unlocking the modules by reversing the rotation. Lego mini-motors are used to actuate the active faces.

Dimensionally, this Crystal prototype is slightly larger than its predecessor. This is primarily due to the addition of the second expansion degree of freedom and the use of much stronger (but slightly larger) expansion motors. Since this is a prototype system, we are more concerned with functionality than small changes in size. The expanded size of a module is 5.2 inches square, and contracted size is 2.6 inches square. The overall height is 7.4 inches with a weight of 18 ounces. Eighteen modules have been constructed.

### 6.2.2 Crystal Robot Software Architecture

Communication is the key component for providing the system support for distributed control in Crystal robots. To this end, we developed a message-passing infrastructure on top of the Crystal's communication capabilities. Each unit maintains a message queue, and can post messages to neighbor modules. A module's program is then centered around a message loop, similar to the message loop in modern windowing systems. In each iteration, the processor polls each UART for incoming messages and adds any new messages to the queue. It then takes a message from the queue and processes it according to the appropriate message handler. Since each UART has its own FIFO, the UARTs still can receive data while the processor is busy handling messages. Because the processor speed is much faster than the UART transmission rate, the risk of the UART FIFOs filling up before they get serviced by the processor has not been an issue.

Library functions were developed to handle the synchronous communication between the processor and UARTs in both directions. Polling of the UARTs is done with a single library call, so that the creation of the message loop is trivial. For these library functions, we have assumed that all messages will be two bytes long (although the content within the two bytes is message dependent). This limit was imposed to allow the system to be able to receive four messages from each direction before the UARTs are polled, although future messaging infrastructure (and interrupt-based communication) will allow us to re-

| sender | data | parent ID | type |
|---|---|---|---|
| 15 | 13 | 7 | 3 0 |
| 0 1 | 1 1 0 0 1 1 | 0 1 1 0 | 0 1 1 0 |

Figure 6.7: Message format example. The sample message is from the goal recognition algorithm, indicating a trace message received from the west face.

lax this restriction. For all the algorithms and robot sizes presented here, this message size limitation has not been problematic.

Using this infrastructure we implemented the distributed algorithms by defining a set of message types and creating message handlers for each message type. In general we have maintained a common format for the messages, so that additional common library functions can be used. In particular, the lowest four bits of the message are reserved for the message type, and the highest two bits are reserved for the direction from which the message was received. These data can be extracted from the message with a bit mask, and providing library functions to perform the bit masking leads to less propensity for coding errors. The data can take up the remainder of the message and be in whatever format is necessary for the particular message or algorithm (in many of our algorithms, the data format is consistent over all message types). An example of a message from the distributed goal recognition algorithm is given in Figure 6.7.

## Development Environment

Since a C compiler is available for the Hitachi H8 microcontroller, programming the Crystal is done on a host computer in C, compiled, and downloaded to the unit with a serial interface. Each unit runs the same code. Heterogeneity is modeled by assigning each unit a unique integer type. To make the build process faster, we manually edited the binary image to assign types before downloading to flash memory.

## Boot Procedure

While the messages used by any given algorithm are necessarily specific to that algorithm, there is a common boot sequence that is used (with some algorithm-specific adaptations possible). A boot sequence is required since the modules must be manually switched on one at a time, so that when a module starts its program, it does not initially know whether its neighbor units are powered on. To solve this problem, we use a special message called *system_init*. This message is initially generated only after all modules have been powered on, and can be sent at later times to effect a software "reboot." This message is created by one module that has a switch. The *system_init* message handler propagates the message to all neighbors, and also recognizes each neighbor from which it received a *system_init*. Any initialization of algorithm-specific global variables is also done in this function. If further *system_init*s are received, they are ignored. However, to enable later soft reboots, there is also a *pre_init* message to set a state bit that is cleared by *system_init*. This allows a module to realize when a *system_init* (or *pre_init*) is a duplicate and when it indicates a new reboot of the system.

## Attaching and Detaching from Neighbors

Attaching and detaching from neighbors is a critical primitive operation in the Crystal. This is an asymmetrical procedure since only one face is active. To connect or disconnect a passive face, a module must make a request to its active neighbor. The challenge is that the connection mechanism is particularly error-prone and subject to jamming. Hall sensors in the mechanism detect when the connector is fully open or fully closed, so three states are observable at the software level: open, closed, and intermediate. A connector in the intermediate state for greater than a certain time threshold is considered stuck.

We implemented software control of connection/disconnection as a function that takes two arguments: 1) requested action (connect or disconnect), and 2) which face to act upon. This function acts differently depending on whether the requested face is active or passive. For an active face, we open the connector, send a message to the neighbor to notify it of disconnection, and wait for an acknowledgement message from the neighbor. Each operation is subject to timeout, defined as five seconds. If timeout occurs, the module turns on its LED to signal failure and halts. For a passive face, we send a message to the active neighbor to request disconnection, and wait for an acknowledgement message. A similar timeout scheme used.

To implement the timeout, we use a busy-waiting scheme that polls the system clock. During this busy-waiting, we must continuously check for acknowledgement messages since the main message loop is blocked. To accomplish this, we peek directly into the hardware FIFOs to search for messages of the correct type.

The connect/disconnect function, therefore, blocks until either the operation is successful or an error occurs. We provide a non-blocking version as well that returns before receiving an acknowledgement message.

### Neighbor Detection

After locomotion, a module may need to discover its local neighborhood. Since no hardware is available to sense directly the presence or absence of a neighbor, we implemented neighbor detection in software. This procedure sends a *ping* message to a given face and waits for an acknowledgement. Busy-waiting, FIFO searching, and timeouts are utilized as in the connect/disconnect function. The neighbor detect function returns true if it received an acknowledgement, and false if it reached timeout.

| Name | Function |
|---|---|
| PRE_INIT | Prepare for initialization |
| SYSTEM_INIT | Perform system initialization |
| GR_START | Begin goal recognition algorithm |
| TRACE | Tests validity in goal matrix |
| TRACE_FAIL | Trace has failed |
| META_TRACE | Check if all traces failed |
| GR_SUCCESS | Return true |
| GR_FAILURE | Return false |

Table 6.4: List of all message types used in goal recognition implementation.

### 6.2.3 Goal Recognition Experiment

We implemented the 2D Goal Recognition algorithm in hardware using the Crystal. To enable the algorithm, we developed message-handling routines as described below, and performed experimental studies with several different configurations of the modules. Results are described in this section.

#### Algorithm Implementation

The message handlers for the required message types are summarized in Table 6.4. We hard-coded the goal matrix, since the message size we chose is so small. We would like to transmit goal matrices on the fly, but that requires a more complex protocol to allow for arbitrarily large messages. When a start message is received, the goal recognition algorithm is started and trace messages cause the module LEDs to blink. When the algorithm finishes, the modules signal success or failure with a given pattern of LED blinks. A sample robot is shown in Figure 6.8.

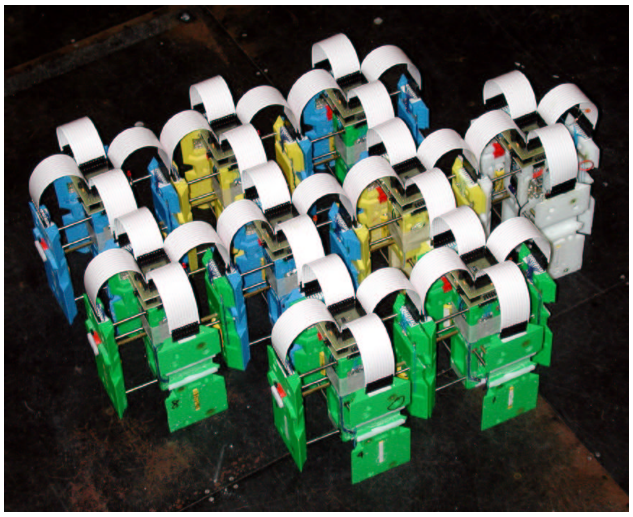


Figure 6.8: Experimental setup for goal recognition corresponding to rightmost column in Table 6.5.

## Results

We executed the 2D Goal Recognition algorithm using various configurations of the Crystal robot. Data is given in Table 6.5. Most trials were successful, but since the communication uses IR, sometimes module misalignment led to communications failure. We hope to address this issue with a new connector design. Fortunately, when this occurred the trace message continued as if the neighbor module did not exist, and the algorithm correctly recognized the subset of the goal shape.

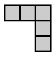
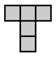
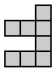
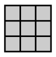
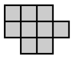
| Robot shape |  |  |  |  |  |
|---|---|---|---|---|---|
| Matches goal? | No | Yes | Yes | Yes | No |
| Trials | 50 | 50 | 50 | 50 | 50 |
| Successes | 49 | 49 | 50 | 50 | 48 |
| Total messages | 25 | 16 | 67 | 32 | 44 |

Table 6.5: Goal recognition experimental results.

#### Combining Goal Recognition and Reconfiguration

The next experiment we conducted investigated the behavior of our Goal Recognition algorithm as a subroutine of a simple reconfiguration algorithm. One column of modules was programmed to “inchworm” along a fixed group of modules (using a variation of the Attaching locomotion algorithm presented below) and automatically initiate Goal Recognition at the end of each inchworm step. Reconfiguration halts when the robot recognizes achievement of the goal shape, in this case a rectangle. Preliminary results indicate that Goal Recognition runs at least an order of magnitude faster than actuation for these robots, although we would still like to use Goal Recognition less often during reconfiguration.

### 6.2.4 Inchworm Locomotion Experiment

The ability to perform distributed locomotion depends on the modules passing their state to their immediate neighbors. The communication infrastructure enables this by allowing us to define a *state* message, which indicates whether the module is expanded in the direction of motion and whether it is connected to a fixed module. This information (along with the message type) easily fits within the two-byte limit. The other message required is the *inch* message, which tells the robot to begin locomotion, and includes the desired direction of travel. The *inch* message is sent from an external source to initiate the locomotion, and is also sent by the head module to trigger another step (when desired). Together with the soft-boot sequence described in Section 6.2.2, these algorithms therefore use only four message types to perform locomotion.

#### Results

Our locomotion experiments attempted to empirically determine the effectiveness of the Inchworm Locomotion algorithm. We tested single-row inchworms on different surfaces

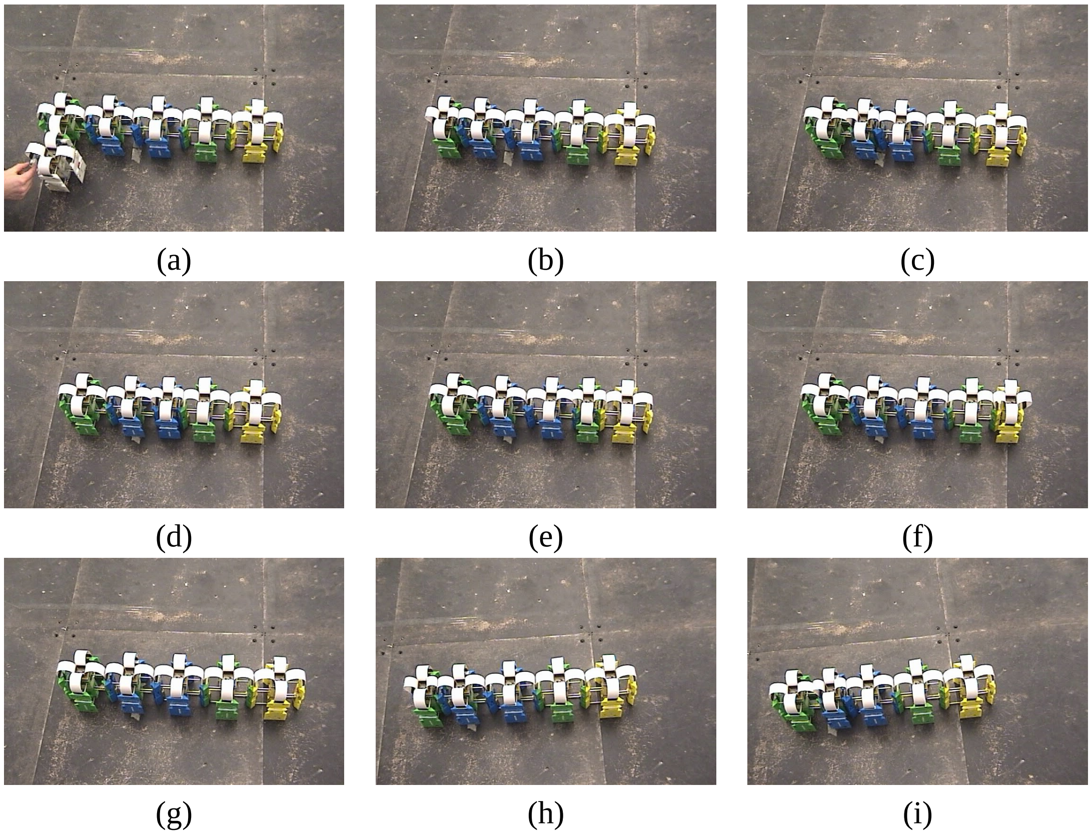
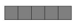
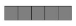
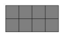
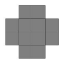


Figure 6.9: First nine steps of Inchworm Locomotion with line configuration of five modules.

as well as more complex shapes (such as the one in Figure 5.6). Snapshots from a five-module line are shown in Figure 6.9. We found the locomotion proceeds well as long as there are at least four modules in each row. The exact amount of progress is also dependent on both the surface and the individual modules in the group. That is, if there is any differential in the friction under the various modules, or if the modules actuate at different speeds (due to internal friction variance or other irregularities), the locomotion will not move as far as the theoretical distance. In addition, since groups that are not simple chains require modules to expand and contract synchronously, it was necessary to ensure that adjacent modules were reasonably well matched to avoid developing excess stress within the Crystal.

In all experimental setups, we found that along fairly smooth surfaces (plexiglass and

| Robot shape | Surface | $N_t$ | $N_s$ | Mean distance (in/step) | Min-Max distance (in/step) |
|---|---|---|---|---|---|
|  | Paint | 10 | 10 | 2.24 | 2.10-2.39 |
|  | Plexiglass | 10 | 10 | 2.32 | 2.22-2.37 |
|  | Plexiglass | 10 | 6 | 2.14 | 1.75-2.25 |
|  | Plexiglass | 10 | 6 | 1.14 | 0.9-1.3 |

$N_t$: Number of trials, $N_s$: Number of steps per trial

Table 6.6: Stand-Alone locomotion performance on painted metal floor and plexiglass. Shapes are given such that the group moves to the left or right. All distances are given as inches per step.

painted metal) the actual distance was 75-85% of the theoretical for four-module rows, and 80-90% of the theoretical for five-module rows. Increasing the group size should allow the distance per step to approach the theoretical limit, but variations in the modules and ground friction limit this effect. These results held for both single- and double-row groups, as can be seen in Table 6.6. The last row in this table is an interesting case — since the top and bottom rows are only two modules long, they work against the overall locomotion, and in fact this robot walked much more slowly than the other groups. The speed of the locomotion is effected by group size, since the time to perform one step is proportional to the length of the group. For the single-row group presented at the top of Table 6.6, the steps were made at the rate of one every 20 seconds, for an overall group speed of 6.7 in/min.

### 6.2.5 Heterogeneous Reconfiguration Experiment

In this section, we present a hardware demonstration of heterogeneous reconfiguration. The algorithm MeltSortGrow approaches the heterogeneous SR problem by reconfiguring the original shape into a regular intermediate structure (Melt) and then building the goal shape (Grow). These two steps treat the system as homogeneous. However, the middle step reconfigures the intermediate structure such that module types are correct. Since the Sort step is what handles heterogeneity, we decided to implement this in hardware using the Crystal robot. This robot was designed to be homogeneous, but by assigning modules individual colors we treat it as a heterogeneous system. Our experiment was to attempt to use self-reconfiguration to sort a row of Crystal modules according to a given color sequence. We developed an algorithm called *CrystalSort* for this task.

These experiments represent the first extensive tests of the reconfiguration ability of this version of the Crystal hardware. Our results show that our algorithms are feasible given the memory, computation, and asynchronous communication requirements of the physical system. The results also revealed mechanical limitations of the connector design that necessitate a redesign in the future.

Next we present algorithm and implementation details. Then we describe our results and comment on lessons learned. Source code from our implementation of CrystalSort both in SRSim and in hardware is listed in Appendix B.

## Sorting Modules by Color

To sort the heterogeneous Crystal robot we instantiate the generic sorting step of Melt-SortGrow to the Crystal actuation method. Given a configuration of modules with color labels, the objective is to reconfigure the structure such that modules are arranged according to a given sequence of colors.

The algorithm will sort a row of Crystal modules using a technique similar to the familiar selection sort. Because of the actuation requirements in a unit-compressible system, this operation needs surrounding structure to succeed, in the form of two additional rows. This allows any single module to retract fully from the current row. The procedure to reposition a module consists of three steps. First, the module is removed from its row by contraction of its upper neighbors. Then, the remaining modules move back and forth to align the module with its new position. Finally, the contracted modules expand to place the module back into its original row but in the new order.

A top level protocol synchronizes the sorting such that modules are placed in their correct order beginning at the left of the configuration and progressing successively to the right. The order is determined by the color sequence of the middle row. The algorithm is illustrated in Figure 6.10. Step (a) shows the initial configuration. The first operation is to move the module in the last column to the first column, which happens in steps (b) through (f). Next, the correct module for column two is located and repositioned in steps (g) and (h). The final operation exchanges modules in columns three and four and results in the sorted configuration shown in (i). Pseudocode for the decentralized implementation is listed in Algorithm 24.

Our experimental setup consisted of a 12-module robot placed on plexiglass (see Figure 6.11). The possible configurations consist of all permutations of a row of four types. We ran the algorithm from a number of initial configurations, summarized in Table 6.7.

### Results and Discussion

The robot successfully completed sorting from numerous initial configurations. In general, the longer trials requiring more motions were more prone to failure. Failures were due exclusively to mechanical limitations of the hardware; the software and algorithms performed correctly. The main hardware errors were caused by IR communication failures and by connector failures. Better hardware prototypes are needed to address the mechanical concerns, although performance can also be enhanced algorithmically by reducing the number of module motions. Friction encountered during expansion and contraction through free space leads to misalignment and hence connection failure. Manual

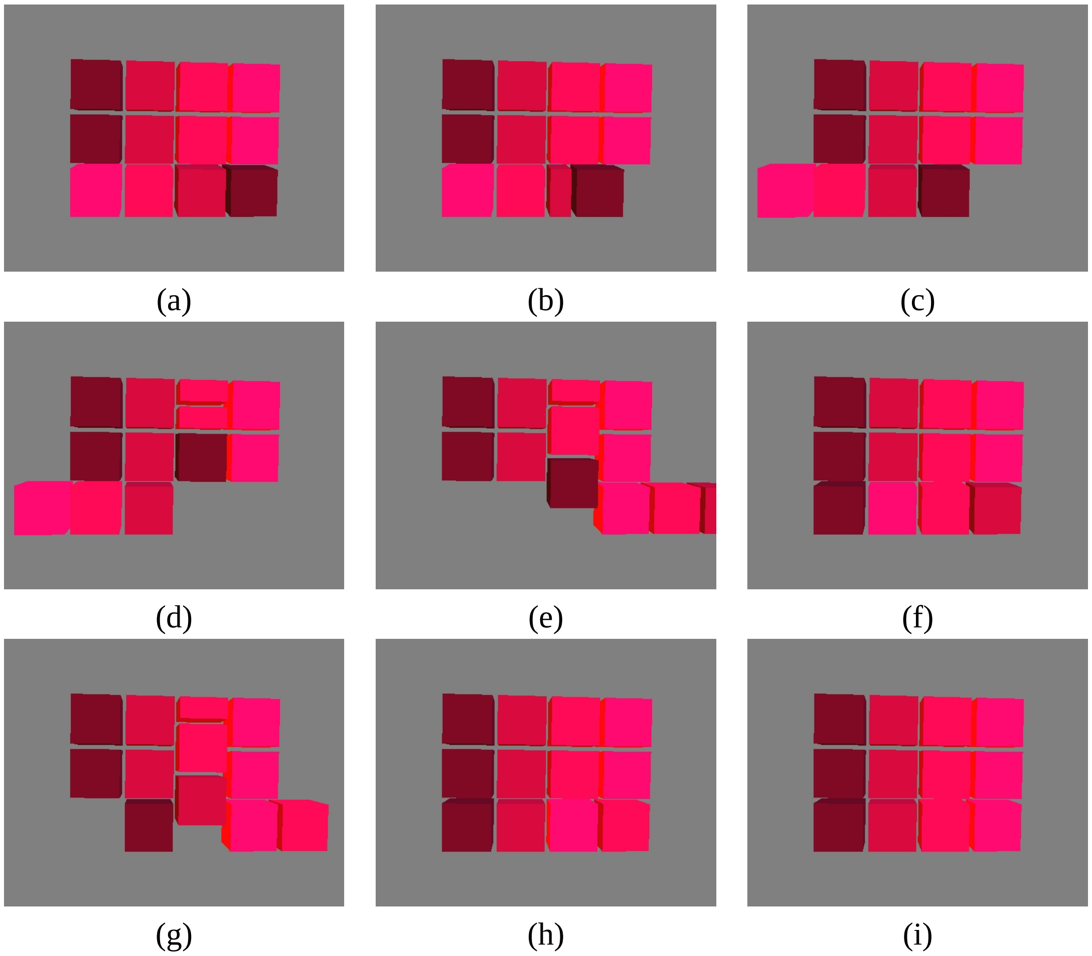


Figure 6.10: Screenshots from CrystalSort implementation in SRSim simulator. Bottom row is in reverse order in (a), and is sorted to match column type. The first swap (4-1), is completed in (f), the second (4-2) is completed in (h), and the final swap (4-3) completes the sorting in (i).

intervention was required during the experiment to ensure connections. Another way to address this issue is by spatially constraining the motion of the modules. This constraint can be accomplished in part by reducing the amount of free space used by the system, allowing stationary modules to become alignment tracks.

The robot successfully completed reconfigurations from numerous initial configurations. In general the longer trials requiring more motions were more prone to failure. Failures were due primarily to mechanical limitations of the hardware. The software

**Algorithm 24** Decentralized algorithm CrystalSort. Assumes three rows of modules, and begins with *sortToken* message sent to leftmost module (column one) in middle row. Bottom row will be sorted to match middle row.

```
State:
my_color, my color label
my_column, my current column

Messages:
sortToken, sent to synchronize sorting
    Action: Request color from bottom neighbor. If neighbor color matches my_color, send
    sortToken to right neighbor. Else search for matching color by sending sortQuery(my_color,
    my_column) to bottom neighbor.
sortQuery(requested_color, column), searches for matching color
    Action: If my_color does not match requested_color, send sortQuery message to right neigh-
    bor. Else, execute moveToPosition(column). When done, pass sortToken back to new upper
    neighbor.

Procedures:
moveToPosition(column)
    1. initiate locomotion to align myself with center of structure
    2. send command to upper neighbors to contract
    3. command bottom row to reconnect
    4. send locomotion command to lower neighbor to produce hole beneath me at desired
    column
    5. command upper neighbors to expand
    6. initiate locomotion to realign row with rest of structure
```

and algorithms performed as designed. These results represent the first extensive tests of reconfiguration in the Crystal, and also the first demonstration of heterogeneous reconfiguration.

The main source of mechanical failures was the inter-module connection mechanism. Because they require tight module alignment to successfully connect, the connectors consistently jammed and required manual intervention (jiggling) to become unstuck. Since the processor is programmed to block until the connection is made, a stuck connector is enough for complete failure of the trial. We acknowledge this as and issue, but in order to continue our evaluation of the rest of the system we circumvented the problem by manually perturbing stuck connectors.

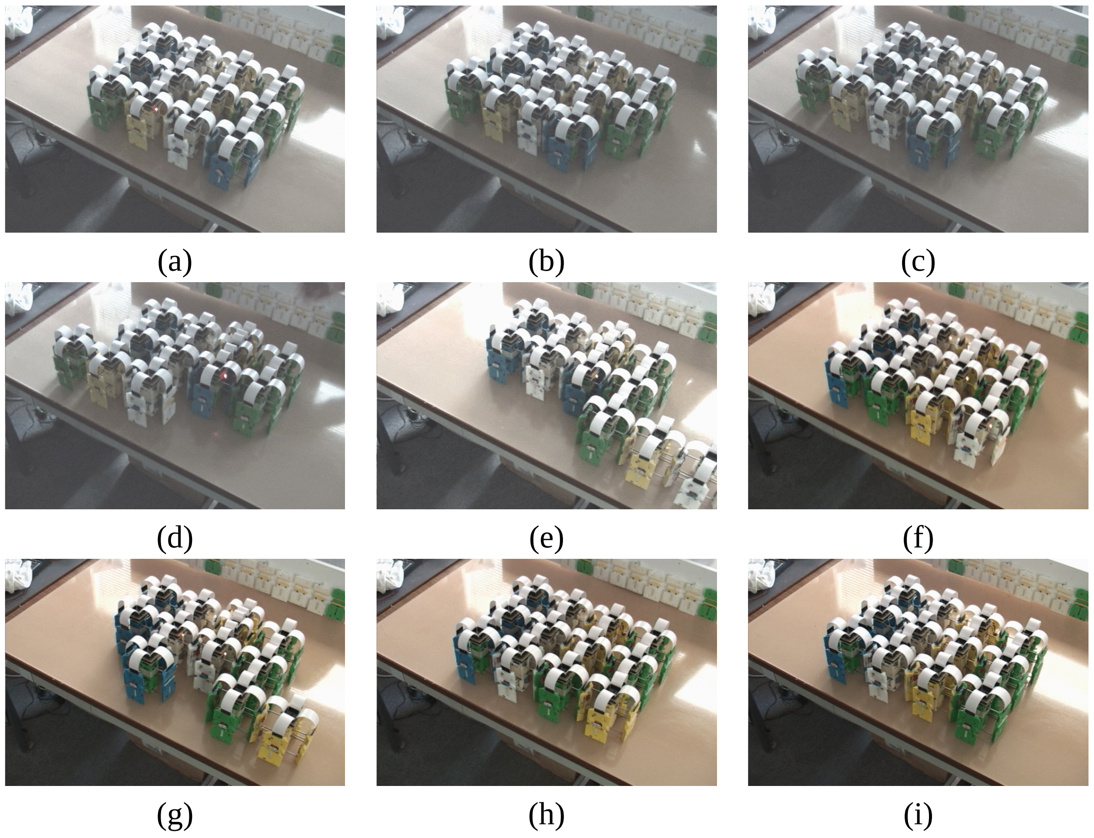


Figure 6.11: CrystalSort implementation in hardware using the Crystal robot. Steps correspond to simulation in Figure 6.10.

The other main source of failure was excessive lateral drift during expansion. Since the modules locomote in chains, this motion error compounds enough that the side faces of the modules collide with neighbors, often preventing further motion and totally disturbing the lattice structure, leading to complete failure. The cause of this problem is not enough rigidity in both the intermodule connection, and the rack mechanism that connects the faces to the core. A potential solution is to allow the modules to realign themselves by pushing against neighbors. We implemented this by partially contracting side faces during expansion. This worked well except when the deformation was too great, when the side face would slip past the neighbor face and become totally entwined. Aside from the connection problems, the modules worked reliably for a long period of time.

Table 6.7: Experimental results from CrystalSort algorithm implementation on Crystal robot from various initial configurations. Goal configuration was 1-2-3-4.

| **Initial Config** | **Number of Swaps** | **Swap Sequence** | **Successful Attempts** | **Total Attempts** | **Manual Interventions** |
|---|---|---|---|---|---|
| 2-1-3-4 | 1 | 2↔1 | 1 | 1 | 12 |
| 2-3-1-4 | 1 | 3↔1 | 1 | 1 | 12 |
| 2-4-3-1 | 1 | 4↔1 | 1 | 1 | 16 |
| 1-3-2-4 | 1 | 3↔2 | 1 | 1 | 10 |
| 1-3-4-2 | 1 | 4↔2 | 1 | 1 | 12 |
| 1-2-4-3 | 1 | 4↔3 | 2 | 4 | 11 |
| 2-1-4-3 | 2 | 2↔1; 4↔3 | 2 | 3 | 23 |
| 3-2-1-4 | 2 | 3↔1; 3↔2 | 1 | 1 | 22 |
| 3-4-1-2 | 2 | 3↔1; 4↔2 | 1 | 2 | 24 |
| 4-3-2-1 | 3 | 4↔1; 4↔2; 4↔3 | 1 | 10-15 (est.) | 35 |
| **Totals:** | | | **12** | **25-30** | |

Another source of failure came from the communications system. We use IR-based hardware and a very simple protocol with 2-byte fixed size messages and no error correction. Excessive ambient light and module misalignment could trigger an erroneous byte transmission. Once the UART entered this one-off state, it required software reboot to recover since further messages always consisted of two bytes. These problems mainly caused failures during boot-up only, but a more sophisticated protocol with error correction and variable message size would eliminate the failures and also add flexibility to software development.

Otherwise, the software architecture worked well. Further hardware prototypes are clearly necessary to address the mechanical concerns. But more importantly, we need to focus our algorithmic work on reducing the number of motions. Time spent and failures caused by expanding and contracting, and making and breaking connections make these expensive operations compared to communication and computation. Therefore it is advantageous to tradeoff computation for closer to optimal reconfiguration plans. Since optimal solutions to reconfiguration are intractable, investigating approximation algorithms is a good avenue for algorithmic work.

## 6.3 Lessons Learned

The main contributions of our experimental work can be summarized as follows:

- A useful simulation tool
- A robust software architecture for SR systems
- First major evaluation of Crystal robot prototype
- First demonstration of heterogeneous SR system in hardware
- Demonstration and evaluation of reconfiguration algorithms in simulation

In this section we discuss these results within the overall topics of hardware, algorithms. We also discuss how the results translate into recommendations for future research directions.

### 6.3.1 Hardware

In terms of hardware development, our work represents alpha testing of the Crystal prototype. We identified a number of mechanical design issues, which are detailed here. As the primary software developers for the system, we also evaluate software issues.

#### Design

Performance of the Crystal modules was generally predictable. The computational resources were adequate. We experienced no problems due to processing power or memory shortages. The mechanical subsystems were reliable in that failures were consistent.

The biggest point of failure, as discussed earlier, was the connection mechanism. In addition to jamming when attempting to connect, the glue joint holding the key assembly to the motor shaft failed frequently. The jamming problem was improved by regluing the key such that it protruded from the module face, thereby allowing greater tolerance for rotation within the neighbor channel. The downside was that this added flexibility to the connection as a whole and contributed to a greater amount of lateral drift during expansion.

Another main failure mode was broken wires connecting the core assembly to the faces. The high-gauge wire experienced considerable fatiguing at the connection with the face during expansion and contraction. Locating these failures was difficult due to the large number of wires involved. Encasing the entire connection area in hot glue added enough to rigidity to prevent further failure.

A less frequent source of failure was the communications hardware. As noted, errant transmissions were received due to ambient light and cross-talk. Communication only occurs in the connected state, so better receiver shielding could help this problem.

We also identified usability issues. Although battery life was relatively long, replacement was time-consuming since the batteries were soldered in place. With four batteries per module, this became a significant problem. Also, the lack of an on/off switch lead to premature battery replacement since it was easy to unintentionally leave the units powered on for long periods of time. To power the modules off, a ribbon cable had to be disconnected. Repeated connection and disconnect of these cables also contributed to wire fatigue and failure.

### Software

The simple software architecture we developed proved to be well-designed and easy to work with. Organizing code by message type allowed us to keep code for multiple algorithms loaded in memory without conflicts. It was troublesome to download code, however, since this involved connecting a wire to each module individually to reprogram its flash memory. As SR systems grow in size, some type of auto-programming will become necessary.

Beneath the message-passing architecture is low-level code for directly controlling motors and communications hardware. The main expansion/contraction is controlled by counting rotations of the motor shaft. Unfortunately, this meant that the width of the module had to be manually calibrated. The other type of motor in the system rotates the connector mechanism. This is controlled with the use of a hall sensor for absolute positioning, which worked well. There is no sensor to detect the presence of a neighbor, though, so we implemented neighbor detection by sending a ping message. If a timeout was exceeded without a receiving a return message, we assume there is no neighbor. Obviously this scheme fails if the neighbor module is present but unable to communicate. A touch sensor or other simple solution is desirable. The communication protocol we developed uses simple fixed-width messages with no software error correction. We were able to work around this limitation for the current algorithm implementations, but a more complex protocol will be necessary to transmit larger amounts of data.

At the level of trajectory control, we were able to compensate for some lateral drift by partially contracting lateral faces during contraction and expansion. Re-expanding the faces helped the module to self-align in some circumstances by pushing against neighbor faces.

The biggest downside of our implementation is that we were unable to recover from most mechanical failures such as stuck connectors. We could implement failure detection in software, but the failure rate in the current system was too high for this to be of use.

### 6.3.2 Algorithms

Implementing our reconfiguration algorithms in simulation aided algorithm development by making special cases apparent and by exposing incorrect assumptions. For Crystal-Sort, initial simulations sped debugging since the debugging facilities offered by modern programming languages are far superior to those found on the robot.

In terms of algorithm design, an important point that our simulation work made apparent is that asymptotic analysis is a necessary but insufficient tool for evaluation of reconfiguration algorithms. We anticipate the number of modules in SR systems to be large, but not infinite. Therefore, asymptotic analysis alone can obscure the comparison of reconfiguration techniques. Because the cost of module actuation is enormous in time and energy, more precise measures of motion counts should be used in practice. Likewise, the notion of instantiating Sliding-Cube modules with different actuation types is another source of hidden extra actuations.

### 6.3.3 Future Work

There are a number of recommendations stemming from our results. The first addresses one of the primary motivations of SR research, fault tolerance through redundancy. We feel that this goal can only be achieved with a minimum level of mechanical reliability combined with fault tolerant algorithms. Our hardware did not perform to a level sufficient to fully explore fault tolerance properties of our algorithms. The mechanical issues we identified must be mitigated in further prototypes.

Another large issue is the consideration of dynamics. Observing the behavior of BFS and DFS as search techniques for melting initial configurations into the line shape in MeltSortGrow makes it clear that arbitrary configurations place impossible demands on the strength of intermodule connections in the presence of gravity.

A final, perhaps obvious recommendation is for closer integration of hardware and algorithm designers in the research process. Tighter cycles of design, development, and testing would help to advance this research program at a much quicker pace.

# Chapter 7

# Discussion and Future Work

The versatility and adaptability of SR robots is increased by the introduction of functional specialization at the module level. In this thesis, we have laid an algorithmic groundwork for the study of heterogeneous SR systems. We showed that heterogeneous reconfiguration can be planned in polynomial time under common assumptions, and characterized the free space constraints that cause our algorithms to fail. These algorithms allow a heterogeneous robot to plan shape changes among obstacles in all realistic situations, and support the relative positioning of modules by function. We also applied the planning methods and implementation techniques developed for reconfiguration algorithms to the related problems of locomotion, self-repair, and distributed goal recognition. Finally, we constructed a novel simulation environment and performed experiments in simulation and in hardware with an existing SR robot prototype.

In addition to the discussion presented in individual chapters, we conclude the thesis with a discussion of overarching issues and suggestions for future work. Section 7.1 discusses the important issue of uncertainty in reconfiguration planning, Section 7.2 compares our overall planning approach with competing approaches, Section 7.3 discusses issues pertaining to real-world applications of SR robots, and Section 7.4 outlines a number of immediate next steps for research suggested by our results.

## 7.1 Uncertainty

All robots operating in the real world face many forms of uncertainty. SR robots are no exception. A system composed of thousands of units has the benefits of redundancy, but also suffers from thousands of opportunities for a module to experience errors and failure. Our approach to uncertainty in reconfiguration planning for SR systems is to divide the issue into a low-level, continuous layer, and a higher-level, discrete layer. The continuous layer subsumes errors in performing connections, disconnections, and other actuations by the physical mechanisms that implement these functions on a single module. An example is connection failure due to module misalignment. Total failure of low-level controllers to correct these errors is treated as a discrete failure and passed up to the higher-level. The topic of how to deal with this low-level uncertainty is critical, but is highly dependent on module implementation and is left to hardware-design researchers. The level of abstraction at which our algorithms operate thus considers only discrete module failures. The approach we present in this thesis is to treat module failure as a total loss and to replace the module with a spare. This is the self-repair approach described in Chapter 5.

There is much work remaining in this area. For uncertainty in the connection mechanism, researchers are exploring mechanical solutions that increase the misalignment tolerance for a successful connection [6, 73]. For other forms of uncertainty, solutions are still in their infancy. Assuming the availability of touch sensors, our locomotion work in Chapter 5 begins to address uncertainty in locomotion by using compliance. Algorithmically, future work could attempt to avoid the total loss of a failed module by leaving it in the structure and planning paths around it. Even so, the self-repair approach is justified based on the basic assumptions of large systems (thousand of units) and a minimum level of module reliability. In other words, if failures fall within an acceptable level, replacing modules as a solution to failure is affordable.

## 7.2 Planning versus Reacting

For all our algorithms, we started with a centralized solution and transformed it into a decentralized version. The main alternative is a pure distributed reactive approach based on gradients or local rules. This warrants comparison. The benefit of our hybrid approach is that we retain convergence and run time guarantees while still executing on a distributed system with only neighbor-to-neighbor communication. The downside is that planning requires significantly more communication and computation than reactive methods. Also, our techniques lack the natural concurrency arising from local rule-based algorithms. In some cases, such as in CA-style locomotion, the price of simple rule-based execution at run-time is paid up front with time spent developing the rules and proving them correct.

Ultimately, which approach is superior will depend on the relative costs of actuation versus computation and communication. If modules move anywhere near as fast as they do in simulation, then reactive methods are probably better. But if actuation remains as slow as it is in our observations, there is plenty of time for planning in order to reduce the total number of expensive module motions.

## 7.3 Towards Real-World Applications

There are many challenges to be met before SR robots can be effective situated in the real world. We discuss two of these challenges here. First, a critical feature of real-world situations is the presence of obstacles. Our work with reconfiguration and locomotion among obstacles is a solid contribution towards meeting this challenge. An important property of this work is that it does not require long-range sensing. Simple sensors such as touch sensors, infrared pairs, or sonar are enough. We added touch sensors to the Crystal modules during locomotion experiments, for example. More complex sensors such as a laser scanner can be added to a single module while keeping the sensing requirements low for the majority of modules in the robot.

Another important issue, outlined in Chapter 1, is how to design and implement a human-robot interface for SR systems. The basic issue is a one-to-many relationship: how can a single user interface with multiple modules? This interface must support both commands to the robot and feedback or other information returning to the user. Direct control of each module is equivalent to centralized control and is undesirable for same reasons that led us towards decentralized/distributed algorithms. It is better to enter the loop at a higher level. This idea is generally referred to as *semi-autonomous control*. For example, we were basically able to interface with the Crystal robot (see Chapter 6) by issuing single-step locomotion commands. We used a spare Crystal module with an added momentary-contact switch (a button) as the interface, and could communicate with any module in the robot.

HRI is a major open problem. Solutions for SR robots can include a number of interesting properties. The first is redundancy – the robot can include a variable number of communications modules, positioned to optimize performance. Secondly, the notion of semi-autonomous control is appealing because it hides low-level details and reduces demands on users attention. The message-based architecture we developed for the Crystal is good platform because control via an external user and an internal module is fundamentally the same. Messages could assume different priority levels to allow an external user to take full control if necessary. Finally, since the most likely physical interface device is a hand-held computer, users should be able to simply specify a desired configuration.

The compressed configuration representation discussed in this thesis (Section 4.4) and in other recent work by Stoy [91] is a good step in this direction.

## 7.4 Future Work

We discussed directions for future work as associated with individual chapters. Here, we highlight main points to conclude the thesis.

### 7.4.1 Optimal Reconfiguration Planning

Based on our experimental results, it is evident that module motion is vastly more expensive, in terms of time and errors, than computation. Therefore, it would be advantageous to reduce motions at the expense of computation. Interesting theoretical questions along these lines include approximating the optimal number of moves for a given configuration, and generating an approximate goal shape using the optimal number of moves. Given the apparent complexity of finding optimal reconfiguration plans, an intriguing question is whether reconfiguration is even polynomial-time approximable, and whether the complexity of approximation differs between homogeneous and heterogeneous systems.

### 7.4.2 Heterogeneity of Module Size and Shape

One of the most natural extensions of our reconfiguration work is to consider other types of heterogeneity. Allowing modules of varying size and shape is an important issue. This would allow interesting applications in assembling large structures. Since one of the most difficult aspects of constructing small modules is how to fit the required components in a package of limited size, large modules present an appealing alternative. The difficulty of this problem depends heavily on the definition of inter-module connection properties.

For example, consider a planar robot comprising both square and rectangular modules. The first question is, how do the two module sizes connect together? If connection is defined such that surface motion is preserved, then the problem can be addressed by a straightforward extension of existing algorithms. If the connection properties impose further constraints on module trajectory planning, then new methods would be required.

### 7.4.3 Approximate Goal Representation

Furthering the ideas initiated in our Position Constraints algorithm is another promising research direction. The two main questions are: 1) how to minimize communication volume while updating partial state estimates in the position-constrained modules, and 2) how to best use state estimates that change over time to plan trajectories for position-constrained modules. We discussed initial solutions in Chapter 5.

The notion of representing configurations other than by specifying the exact position of every module can begin to address the general problem of configuration determination, which has not yet received appreciable attention. Representations that allow variable resolution, such as non-uniform meshes in finite element methods [26], are promising.

### 7.4.4 Dynamics

We have generally ignored dynamics in our reconfiguration planning work. This is an obvious shortcoming. It is interesting to consider modifying simple algorithms such as MeltSortGrow to choose the order in which modules are moved in such a way as to preserve properties in addition to connectedness. For example, lattice-based systems are generally treated as statically stable structures and the center of mass can easily be computed. One idea is to only make moves that prevent the robot from toppling over by keeping the center of mass within the current footprint. Another idea is to favor dense structures over sparse structures so as to minimize the occurrence of long chains of unsupported modules.

### 7.4.5 Learning

Finally, we note that although our decentralized algorithms have many useful properties, they still require a significant amount of *a priori* knowledge. Reducing this requirement is desirable for operation in unknown environments. Either the module controllers were manually synthesized, as in the locomotion results, or the goal configuration was provided, as in the reconfiguration planning results. A possible progression is to use learning techniques to further this line of research. A feasible issue to attack is configuration determination, but the general approach of using learning in SR robots research remains an open question.

# Appendix A

# SRSim Source Code

SRSim is the SR robot simulation environment we built for implementing and animating our algorithms. This Appendix includes java source code samples from SRSim. Section A.1 lists full source code for the simulator's base class. Section A.2 lists the Sliding-Cube module object implementation. A code template for implementing algorithms in SRSim is listed in Section A.3.

## A.1 SRSim Base Class

```
/**********************************************************************
 * SRSimBase.java
 *
 * Base class for SRSim simulations. Can be extended for implementing
 * specific algorithms, or used as a template.
 *
 *  RCF Robert Fitch
 *
 *  12/17/02    RCF     Initial version
 *
 * NOTES:
 *
**********************************************************************/

import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.SimpleUniverse;
import javax.media.j3d.*;
import com.sun.j3d.utils.behaviors.vp.*;
import javax.vecmath.*;
import com.sun.j3d.utils.image.TextureLoader;
import com.sun.j3d.utils.geometry.*;
import java.io.*;
//import ncsa.j3d.ui.record.*;


public abstract class SRSimBase extends Applet
{
    ////////////////////////////////////////////////////////////////////
    // default params
    private static float WIDTH  = 8.0f;
    private static float HEIGHT = 0.0f;
    private static float LENGTH = 8.0f;
    private static String floorImage = "floor.gif";


    ////////////////////////////////////////////////////////////////////
    // private member vars
    protected SimpleUniverse simpleUniverse     = null;
    protected PositionInterpolator pi           = null;
    protected BranchGroup removeableBehavior    = null;
    protected BranchGroup objRoot               = null;
    protected OrbitBehavior orbit               = null;
    protected Canvas3D canvas                   = null;
    //protected RecordableCanvas3D canvas           = null;
    private boolean flipYZ                      = false;

    ////////////////////////////////////////////////////////////////////
    // constructors
    public SRSimBase()
    {}

    public void init()
    {
        // set up our graphical environment
        setLayout(new BorderLayout());
        GraphicsConfiguration config =
            SimpleUniverse.getPreferredConfiguration();
        canvas = new Canvas3D(config);
        //canvas = new RecordableCanvas3D(config);
        add("Center", canvas);

        BranchGroup scene = createSceneGraph();
        scene.compile();
        orbit = new OrbitBehavior(canvas, OrbitBehavior.REVERSE_ALL
                                  | OrbitBehavior.STOP_ZOOM);
        orbit.setSchedulingBounds
            (new BoundingSphere(new Point3d(5,5,2),10));

        simpleUniverse = new SimpleUniverse(canvas);
        setViewPlatformTransform();
        simpleUniverse.getViewingPlatform().setViewPlatformBehavior
            (orbit);
        simpleUniverse.addBranchGraph(scene);
    }


    /////////////////////////////////////////////////////////////////////
    // Scene graph methods
    //
    protected BranchGroup createSceneGraph()
    {
        // create some objects
        objRoot = new BranchGroup();
        objRoot.setCapability(BranchGroup.ALLOW_CHILDREN_EXTEND);
        objRoot.setCapability(BranchGroup.ALLOW_DETACH);

        // Set up the background
        createBackground();

        // Set up the global lights
        createLights();

        // add the ground
        objRoot.addChild(createLand(new Point3f
                                    (0.0f, 0.0f, 0.0f),
                                    WIDTH,
                                    HEIGHT,
                                    LENGTH,
                                    floorImage));

        // and we’re done
        return objRoot;
    }

    protected void setViewPlatformTransform()
    {
     simpleUniverse.getViewingPlatform().setNominalViewingTransform();
    }

    protected void createBackground()
    {
        Background bgNode = new Background(0.5f, 0.5f, 0.5f);
        bgNode.setApplicationBounds(new BoundingSphere());
        objRoot.addChild(bgNode);
    }

    protected void createLights()
    {
        Color3f lColor1 = new Color3f(0.7f, 0.7f, 0.7f);
        Vector3f lDir1  = new Vector3f(-1.0f, -1.0f, -1.0f);
        Color3f alColor = new Color3f(0.2f, 0.2f, 0.2f);

        AmbientLight aLgt = new AmbientLight(alColor);
        aLgt.setInfluencingBounds
            (new BoundingSphere(new Point3d(5,5,2),10));
        DirectionalLight lgt1 = new DirectionalLight(lColor1, lDir1);
        lgt1.setInfluencingBounds
            (new BoundingSphere(new Point3d(5,5,2),10));
        objRoot.addChild(aLgt);
        objRoot.addChild(lgt1);
    }

    protected Group createLand(Point3f origin,
                               float width,
                               float height,
                               float length,
                               String textureFile)
    {
       QuadArray quadArray = new QuadArray
        (4, GeometryArray.COORDINATES |
            GeometryArray.NORMALS |
            GeometryArray.TEXTURE_COORDINATE_2);

        float[] coordArray = {
            origin.x, origin.y+height, origin.z + length,  //near left
            origin.x+width, origin.y+height, origin.z + length,//right
            origin.x+width, origin.y+height, origin.z,    // far right
            origin.x,       origin.y+height, origin.z };  // left

        quadArray.setCoordinates
            ( 0, coordArray, 0, coordArray.length/3 );


        for( int n = 0; n < coordArray.length/3; n++ )
            quadArray.setNormal( n, new Vector3f( 0,1,0 ) );

        float[] texArray = {0, 0,
                            1, 0,
                            1, 1,
                            0, 1 };

        quadArray.setTextureCoordinates
            ( 0, 0, texArray, 0, coordArray.length/3 );
        // load texture
        Texture tex = new TextureLoader
            ( textureFile, this ).getTexture( );
        Appearance app = new Appearance();
        app.setTexture( tex );

        // make the image show on both sides of the ground
        PolygonAttributes pa = new PolygonAttributes();
        pa.setCullFace(PolygonAttributes.CULL_NONE);
        app.setPolygonAttributes(pa);

        // and put it all together
        BranchGroup bg = new BranchGroup( );
        Shape3D shape = new Shape3D(quadArray, app );
        bg.addChild( shape );

        return bg;
    }

    protected Group createTerrainMappedLandFlipYZ
        (Point3f origin, float width, float height, float length,
         String textureFile, String heightMapFile)
    {
        flipYZ = true;
        Group ret = createTerrainMappedLand
          (origin, width, height, length, textureFile, heightMapFile);
        flipYZ = false;
        return ret;
    }

    protected Group createTerrainMappedLand
        (Point3f origin, float width, float height, float length,
         String textureFile, String heightMapFile)
    {
        // takes a heightmap file
        //
        boolean readTexCoords = true;
        Point2f[][] textureMap = null;
        float[][] heightMap = null;
        int xLength,zLength;

        // parse file: [col][row]
        try
        {
        FileInputStream stream = new FileInputStream(heightMapFile);
        InputStreamReader reader = new  InputStreamReader(stream);
        StreamTokenizer tokens = new StreamTokenizer(reader);

        tokens.nextToken();
        if ((int)tokens.nval==1) {readTexCoords = false;}
        tokens.nextToken();
```

```
xLength = (int)tokens.nval;
tokens.nextToken();
zLength = (int)tokens.nval;
heightMap = new float[xLength][zLength];
textureMap = new Point2f[xLength][zLength];
float xInt, zInt;
xInt = 1.0f/(xLength-1);
zInt = 1.0f/(zLength-1);

System.out.print("Reading "
                 + xLength
                 + "x"
                 + zLength
                 + " elevation map");
if (readTexCoords) {
    System.out.println(" with texture coordinates.");
}
else {
     System.out.println(" and auto-generating
                                texture coordinates.");
}
for(int i=0; i<zLength; i++)
{
    for(int j=0; j<xLength; j++)
    {
        tokens.nextToken();
        heightMap[j][i] = (float)tokens.nval;
        if (readTexCoords)
        {
        }
        else
        {
            if (flipYZ)
                textureMap[j][i] = new Point2f(j*xInt,i*zInt);
            else {
                textureMap[j][i] =
                    new Point2f(j*xInt,1.0f-i*zInt);
            }
        }
    }
}

stream.close();
}
catch (IOException e)
{
    System.out.println("error reading elevation map "
                       + heightMapFile);
    return null;
}
```

```
    return createTerrainMappedLand(origin,
                                   width,
                                   height,
                                   length,
                                   textureFile,
                                   textureMap,
                                   heightMap);
}

protected Group createTerrainMappedLandFlipYZ(Point3f origin,
                                              float width,
                                              float height,
                                              float length,
                                              String textureFile,
                                              float[][] heightMap)
{
    flipYZ = true;
    Group ret = createTerrainMappedLand(origin,
                                        width,
                                        height,
                                        length,
                                        textureFile,
                                        heightMap);
    flipYZ = false;
    return ret;
}

protected Group createTerrainMappedLand(Point3f origin,
                                        float width,
                                        float height,
                                        float length,
                                        String textureFile,
                                        float[][] heightMap)
{
    // auto-generate texture coordinates
    //
    int xLength = heightMap.length;
    int zLength = heightMap[0].length;
    Point2f[][] textureMap = new Point2f[xLength][zLength];
    float xInt = 1.0f/(xLength-1);
    float zInt = 1.0f/(zLength-1);

    for(int i=0; i<xLength; i++)
    {
        for(int j=0; j<zLength; j++)
        {
            if (flipYZ)
                textureMap[i][j] = new Point2f(i*xInt,j*zInt);
            else
                textureMap[i][j] =new Point2f(i*xInt,1.0f-j*zInt);
        }
    }

    return createTerrainMappedLand(origin, width, height,
                                   length, textureFile, textureMap,
                                   heightMap);
}

protected Group createTerrainMappedLandFlipYZ(Point3f origin,
                                           float width,
                                           float height,
                                           float length,
                                           String textureFile,
                                           Point2f[][] textureMap,
                                           float[][] heightMap)
{
    flipYZ = true;
    Group ret = createTerrainMappedLand(origin, width, height,
                                        length, textureFile,
                                        textureMap, heightMap);
    flipYZ = false;
    return ret;
}

protected Group createTerrainMappedLand(Point3f origin,
                                        float width,
                                        float height,
                                        float length,
                                        String textureFile,
                                        Point2f[][] textureMap,
                                        float[][] heightMap)
{
    // unravel coordinate data: [col][row]
    if ((textureMap.length != heightMap.length) ||
        (textureMap[0].length != heightMap[0].length))
    {
        System.out.println("error creating terrain. size of
            texture and height map data don’t match");
        return null;
    }
    int index = 0;
    Point3f[] coordArray =
        new Point3f[heightMap.length*heightMap[0].length +
            (heightMap.length-2)*heightMap[0].length];
    float[] texArray = new float[2*coordArray.length];
    int[] stripCounts = new int[heightMap.length-1];
    float xInterval = (width - origin.x)/heightMap.length;
    float zInterval = (length - origin.z)/heightMap[0].length;


    for (int i=0; i<heightMap.length-1; i++)
    {
```

```
stripCounts[i] = 2*heightMap[0].length;

if (flipYZ) {
    coordArray[index] =
        new Point3f(xInterval*i,0.0f,
                    heightMap[i][heightMap[0].length-1]);
} else {
    coordArray[index] =
        new Point3f(xInterval*i,heightMap[i][0],0.0f);
}
texArray[2*index] = textureMap[i][0].x;
texArray[2*index+1] = textureMap[i][0].y;
index++;

for(int j=0; j<heightMap[0].length-1; j++)
{
    if (flipYZ) {
        coordArray[index] =
            new Point3f(xInterval*(i+1),
               zInterval*j,
               heightMap[i+1][(heightMap[0].length-1)-j]);
    } else {
        coordArray[index] = new Point3f(xInterval*(i+1),
                                        heightMap[i+1][j],
                                        zInterval*j);
    }
    texArray[2*index] = textureMap[i+1][j].x;
    texArray[2*index+1] = textureMap[i+1][j].y;
    index++;
    if (flipYZ) {
        coordArray[index] =
        new Point3f(xInterval*i,
                 zInterval*(j+1),
                 heightMap[i][(heightMap[0].length-2)-j]);
    } else {
        coordArray[index] =
            new Point3f(xInterval*i,
                        heightMap[i][j+1],
                        zInterval*(j+1));
    }
    texArray[2*index] = textureMap[i][j+1].x;
    texArray[2*index+1] = textureMap[i][j+1].y;
    index++;
}

if (flipYZ) { coordArray[index] =
    new Point3f(xInterval*(i+1),
                zInterval*(heightMap[0].length-1),
                heightMap[i+1][0]);
} else {
    coordArray[index] =
                new Point3f(xInterval*(i+1),
                            heightMap[i+1][heightMap[0].length-1],
                            zInterval*(heightMap[0].length-1));
            }
            texArray[2*index] =
                textureMap[i+1][heightMap[0].length-1].x;
            texArray[2*index+1] =
                textureMap[i+1][heightMap[0].length-1].y;
            index++;
        }

        // load texture
        Texture tex =
            new TextureLoader( textureFile, this ).getTexture( );
        Appearance app = new Appearance();
        app.setTexture( tex );

        // build geometry
        GeometryInfo gi =
            new GeometryInfo( GeometryInfo.TRIANGLE_STRIP_ARRAY );
        gi.setTextureCoordinateParams(1,2);
        gi.setCoordinates(coordArray);
        gi.setTextureCoordinates(0,texArray);
        gi.setStripCounts( stripCounts );

        NormalGenerator normalGenerator = new NormalGenerator( );
        normalGenerator.generateNormals( gi );

        // make the image show on both sides of the ground
        PolygonAttributes pa = new PolygonAttributes();
        pa.setCullFace(PolygonAttributes.CULL_NONE);
        //pa.setPolygonMode(PolygonAttributes.POLYGON_LINE);
        app.setPolygonAttributes(pa);

        // and put it all together
        BranchGroup bg = new BranchGroup( );
        Shape3D shape = new Shape3D( gi.getGeometryArray( ), app);
        bg.addChild( shape );

        return bg;
    }
}
```

## A.2 Sliding-Cube Class

```
/*********************************************************************
 * SlidingCube.java
 *
 * Implements a generic sliding cube model self-reconfiguring robot
 * module
 *
 * RCF  Robert Fitch
 *
 *  12/07/02    RCF     Initial version, extends Box.
 *  01/01/03    RCF     rewritten to extend Shape3D
 *
 * NOTES:
 *  we initially extended Primitive, but according to the selman book
 *  that’s a bad idea. so instead we’ll extend Shape3D and get the
 *  geometry by copying the source from ColorCube.java
 *
*********************************************************************/

import javax.media.j3d.*;
import javax.vecmath.Color3f;
import javax.vecmath.Point3f;
import javax.vecmath.Point3i;

public class SlidingCube extends Shape3D
{
    /////////////////////////////////////////////////////////////////
    // default params (public constants)
    public static final float DEFAULT_X = 0.045f;
    public static final float DEFAULT_Y = 0.045f;
    public static final float DEFAULT_Z = 0.045f;
    public static final Color3f COLOR_RED   =
                                  new Color3f(1.0f, 0.0f, 0.0f);
    public static final Color3f COLOR_GREEN =
                                  new Color3f(0.0f, 1.0f, 0.0f);
    public static final Color3f COLOR_BLUE  =
                                  new Color3f(0.0f, 0.0f, 1.0f);
    public static final Color3f COLOR_BLACK =
                                  new Color3f(0.0f, 0.0f, 0.0f);
    public static final Color3f COLOR_WHITE =
                                  new Color3f(1.0f, 1.0f, 1.0f);
    public static final int TRANSPARENT = 1;
    public static final int WIREFRAME   = 2;
    public static final int SOLID       = 3;
    public static final int RED         = 1;
    public static final int GREEN       = 2;
    public static final int BLUE        = 3;
    public static final int BLACK       = 4;
    public static final int WHITE       = 5;

    protected static SlidingCubeUpdater geometryUpdater = null;

    private static Appearance defaultAppearance         = null;
    private static Appearance transparentAppearance     = null;
    private static Appearance solidAppearance           = null;
    private static Appearance wireFrameAppearance       = null;
    private static Appearance flatSolidAppearanceRed    = null;
    private static Appearance flatSolidAppearanceGreen  = null;
    private static Appearance flatSolidAppearanceBlue   = null;
    private static Appearance flatSolidAppearanceBlack  = null;
    private static Appearance flatSolidAppearanceWhite  = null;
    private static int range = 2;


    private static final float[] normals = {
   // Front Face
      0.0f,  0.0f,  1.0f,     0.0f,  0.0f,  1.0f,
      0.0f,  0.0f,  1.0f,     0.0f,  0.0f,  1.0f,
   // Back Face
      0.0f,  0.0f, -1.0f,     0.0f,  0.0f, -1.0f,
      0.0f,  0.0f, -1.0f,     0.0f,  0.0f, -1.0f,
   // Right Face
      1.0f,  0.0f,  0.0f,     1.0f,  0.0f,  0.0f,
      1.0f,  0.0f,  0.0f,     1.0f,  0.0f,  0.0f,
   // Left Face
     -1.0f,  0.0f,  0.0f,    -1.0f,  0.0f,  0.0f,
     -1.0f,  0.0f,  0.0f,    -1.0f,  0.0f,  0.0f,
   // Top Face
      0.0f,  1.0f,  0.0f,     0.0f,  1.0f,  0.0f,
      0.0f,  1.0f,  0.0f,     0.0f,  1.0f,  0.0f,
   // Bottom Face
      0.0f, -1.0f,  0.0f,     0.0f, -1.0f,  0.0f,
      0.0f, -1.0f,  0.0f,     0.0f, -1.0f,  0.0f,
    };


    ///////////////////////////////////////////////////////////////////
    // private member vars
    //
    protected QuadArray cube = null;


    ///////////////////////////////////////////////////////////////////
    // private member vars
    //
    private Point3f centroid;
    private Point3f newCentroid;
    private double scale;
    private int id = 0;
    private int label = 0;
    private boolean unlabeled = true;


    /////////////////////////////////////////////////////////////////
    // constructors
    //
    public SlidingCube()
    {
        this(0.0f, 0.0f, 0.0f);
    }

    public SlidingCube(float x, float y, float z)
    {
        geometryUpdater = new SlidingCubeUpdater();
        createAppearances();

        cube = new QuadArray(24, QuadArray.COORDINATES
                             | QuadArray.NORMALS
                             | QuadArray.BY_REFERENCE);
        cube.setCapability(GeometryArray.ALLOW_REF_DATA_READ);
        cube.setCapability(GeometryArray.ALLOW_REF_DATA_WRITE);
        cube.setCapability(GeometryArray.ALLOW_COUNT_READ);

        centroid    = new Point3f(x, y, z);
        newCentroid = new Point3f(x, y, z);

        cube.setCoordRefFloat(getVerts(x, y, z));
        cube.setNormalRefFloat(normals);
        this.setGeometry(cube);

        this.setAppearance(defaultAppearance);
        this.setCapability(Shape3D.ALLOW_APPEARANCE_READ);
        this.setCapability(Shape3D.ALLOW_APPEARANCE_WRITE);
    }


    /////////////////////////////////////////////////////////////////
    // public methods
    //
    public Point3f getCentroid()
    {
        return(centroid);
    }

    public Point3f getNewCentroid()
    {
        return(newCentroid);
    }

    public void setNewCentroid(Point3f p)
```

```
{
    centroid = p;
}

public Point3i getGridCoordinates()
{
    // this returns this objects into the grid.
    // it’s just a simple transform.
    return new Point3i(Math.round(centroid.x*10.0f),
                       Math.round(centroid.y*10.0f),
                       Math.round(centroid.z*10.0f));
}

public void setAppearance(int appearanceConstant)
{
    switch(appearanceConstant)
    {
    case TRANSPARENT:
        this.setAppearance(transparentAppearance);
        break;
    case SOLID:
        this.setAppearance(solidAppearance);
        break;
    case WIREFRAME:
        this.setAppearance(wireFrameAppearance);
        break;
    }
}

public void setColor(int color)
{
    switch(color)
    {
    case RED:
        this.setAppearance(flatSolidAppearanceRed);
        break;
    case GREEN:
        this.setAppearance(flatSolidAppearanceGreen);
        break;
    case BLUE:
        this.setAppearance(flatSolidAppearanceBlue);
        break;
    case BLACK:
        this.setAppearance(flatSolidAppearanceBlack);
        break;
    case WHITE:
        this.setAppearance(flatSolidAppearanceWhite);
        break;
    }
}
```

```
public void move(float x, float y, float z)
{
    centroid.x += x;
    centroid.y += y;
    centroid.z += z;
    geometryUpdater.setOffset(x,y,z);
    cube.updateData(geometryUpdater);
}

public void move(Point3i p) {
    move(p.x*0.1f, p.y*0.1f, p.z*0.1f);
}

public void moveTo(Point3i p) {
    this.moveTo(p.x*0.1f, p.y*0.1f, p.z*0.1f);
}

public void moveTo(float x, float y, float z)
{
    centroid.x = x;
    centroid.y = y;
    centroid.z = z;
    geometryUpdater.setPosition(x,y,z);
    cube.updateData(geometryUpdater);
}

public int getID() { return id; }

public void setID(int id) {setID(id,false);}
public void setID(int id, boolean chromaCode)
{
    // set the id and generate a color coding if requested
    this.id = id;

    if (chromaCode)
    {
        float color = id*(1.0f/range);
        Appearance app = new Appearance();
        Material mat = new Material();

        /*
        if (id == 1)
            color = .4f;
        else
            color = .65f;
        */

        mat.setDiffuseColor(color, .0f, color);
        mat.setSpecularColor(.84f, .0f, .0f);
        mat.setEmissiveColor(color, .0f, .0f);
        mat.setShininess(15f);
        // ivory looking appearance
        //mat.setDiffuseColor(.2f, .2f, .7f);
        //mat.setSpecularColor(.85f, .85f, .85f);
        //mat.setEmissiveColor(.7f, .7f, .8f);
        //mat.setShininess(22f);

        app.setMaterial(mat);
        this.setAppearance(app);
    }
}

public static void setColorRange(int r) {range = r;}
public static int getColorRange()   {return range;}

public int getLabel() { return label; }
public void setLabel(int label) {
    unlabeled = false; this.label=label;
}
public boolean unlabeled() {return unlabeled;}


//////////////////////////////////////////////////////////////////
// private methods
//
protected static float[] getVerts(float x, float y, float z)
{
    float[] verts =
    {
        // front face
         DEFAULT_X + x, -DEFAULT_Y + y,  DEFAULT_Z + z,
         DEFAULT_X + x,  DEFAULT_Y + y,  DEFAULT_Z+ z,
        -DEFAULT_X + x,  DEFAULT_Y + y,  DEFAULT_Z+ z,
        -DEFAULT_X + x, -DEFAULT_Y + y,  DEFAULT_Z+ z,
        // back face
        -DEFAULT_X + x, -DEFAULT_Y + y, -DEFAULT_Z+ z,
        -DEFAULT_X + x,  DEFAULT_Y + y, -DEFAULT_Z+ z,
         DEFAULT_X + x,  DEFAULT_Y + y, -DEFAULT_Z+ z,
         DEFAULT_X + x, -DEFAULT_Y + y, -DEFAULT_Z+ z,
        // right face
         DEFAULT_X + x, -DEFAULT_Y + y, -DEFAULT_Z+ z,
         DEFAULT_X + x,  DEFAULT_Y + y, -DEFAULT_Z+ z,
         DEFAULT_X + x,  DEFAULT_Y + y,  DEFAULT_Z+ z,
         DEFAULT_X + x, -DEFAULT_Y + y,  DEFAULT_Z+ z,
        // left face
        -DEFAULT_X + x, -DEFAULT_Y + y,  DEFAULT_Z+ z,
        -DEFAULT_X + x,  DEFAULT_Y + y,  DEFAULT_Z+ z,
        -DEFAULT_X + x,  DEFAULT_Y + y, -DEFAULT_Z+ z,
        -DEFAULT_X + x, -DEFAULT_Y + y, -DEFAULT_Z+ z,
        // top face
         DEFAULT_X + x,  DEFAULT_Y + y,  DEFAULT_Z+ z,
         DEFAULT_X + x,  DEFAULT_Y + y, -DEFAULT_Z+ z,
        -DEFAULT_X + x,  DEFAULT_Y + y, -DEFAULT_Z+ z,
        -DEFAULT_X + x,  DEFAULT_Y + y,  DEFAULT_Z+ z,
        // bottom face
        -DEFAULT_X + x, -DEFAULT_Y + y,  DEFAULT_Z+ z,
        -DEFAULT_X + x, -DEFAULT_Y + y, -DEFAULT_Z+ z,
         DEFAULT_X + x, -DEFAULT_Y + y, -DEFAULT_Z+ z,
         DEFAULT_X + x, -DEFAULT_Y + y,  DEFAULT_Z+ z,
    };
    return(verts);
}

private static void createAppearances()
{
    Color3f objColor = null;
    PolygonAttributes pa = null;

    // default appearance
    pa = new PolygonAttributes();
    pa.setPolygonMode(PolygonAttributes.POLYGON_LINE);
    defaultAppearance = new Appearance();
    defaultAppearance.setPolygonAttributes(pa);

    // lit solid
    solidAppearance = new Appearance();
    Material mat = new Material();
    mat.setDiffuseColor(.34f, .0f, .34f);
    mat.setSpecularColor(.84f, .0f, .0f);
    mat.setEmissiveColor(.34f, .0f, .0f);
    mat.setShininess(15f);
    solidAppearance.setMaterial(mat);

    // transparent appearance
    transparentAppearance = new Appearance();
    TransparencyAttributes ta = new TransparencyAttributes();
    ta.setTransparencyMode(ta.BLENDED);
    ta.setTransparency(0.6f);
    transparentAppearance.setTransparencyAttributes(ta);

    pa = new PolygonAttributes();
    pa.setCullFace(pa.CULL_NONE);
    transparentAppearance.setPolygonAttributes(pa);

    objColor = new Color3f(0.7f, 0.8f, 1.0f);
    transparentAppearance.setMaterial(new Material(objColor,
                                                   COLOR_BLACK,
                                                   objColor,
                                                   COLOR_BLACK,
                                                   1.0f));

    // wireframe
    pa = new PolygonAttributes();
    pa.setPolygonMode(PolygonAttributes.POLYGON_LINE);
    wireFrameAppearance = new Appearance();
    wireFrameAppearance.setPolygonAttributes(pa);

    //colors
    flatSolidAppearanceRed = new Appearance();
    flatSolidAppearanceRed.setColoringAttributes(
        new ColoringAttributes(COLOR_RED,
                               ColoringAttributes.FASTEST));

    flatSolidAppearanceGreen = new Appearance();
    flatSolidAppearanceGreen.setColoringAttributes(
        new ColoringAttributes(COLOR_GREEN,
                               ColoringAttributes.FASTEST));

    flatSolidAppearanceBlue = new Appearance();
    flatSolidAppearanceBlue.setColoringAttributes(
        new ColoringAttributes(COLOR_BLUE,
                               ColoringAttributes.FASTEST));

    flatSolidAppearanceBlack = new Appearance();
    flatSolidAppearanceBlack.setColoringAttributes(
        new ColoringAttributes(COLOR_BLACK,
                               ColoringAttributes.FASTEST));

    flatSolidAppearanceWhite = new Appearance();
    flatSolidAppearanceWhite.setColoringAttributes(
        new ColoringAttributes(COLOR_WHITE,
                               ColoringAttributes.FASTEST));
}

//*****************************************************************
// helper classes
//
public class SlidingCubeUpdater implements GeometryUpdater
{
    GeometryArray geometryArray;
    float[] vertices;
    private float x = 0.0f;
    private float y = 0.0f;
    private float z = 0.0f;
    private boolean absolutePosition = true;

    public void setPosition(float x, float y, float z)
    {
        absolutePosition = true;
        this.x = x;
        this.y = y;
        this.z = z;
    }
```

```
        public void setOffset(float x, float y, float z)
        {
            absolutePosition = false;
            this.x = x;
            this.y = y;
            this.z = z;
        }

        public void updateData(Geometry geometry)
        {
            geometryArray = (GeometryArray)geometry;
            if (absolutePosition)
            {
                geometryArray.setCoordRefFloat(getVerts(x, y, z));
            }
            else
            {
                vertices = geometryArray.getCoordRefFloat();
                for (int i=0; i<72 ; i+=3)
                {
                    vertices[i]   += x;
                    vertices[i+1] += y;
                    vertices[i+2] += z;
                }
            }
        }
    }
}
```

## A.3 Algorithm Implementation Template

```
/*******************************************************************
 * MyNewAlgorithm.java
 *
 *  Template for using the SRSim base classes.
 *  Use this to write new algos.
 *
 *  RCF Robert Fitch
 *  --add other author initials/name pairs here---
 *
 *  01/08/03    RCF     Initial version
 *  ---add revision history here---
 *
 * NOTES:
 *  ---add whatever notes you have here---
 *
 ******************************************************************/

import java.awt.*;
import java.awt.event.*;

import com.sun.j3d.utils.applet.MainFrame;
import javax.media.j3d.*;

import javax.swing.JOptionPane;
import java.util.Enumeration;

public class MyNewAlgorithm extends SRSimBase
{
    ///////////////////////////////////////////////////////////////////
    // private member vars
    //


    ///////////////////////////////////////////////////////////////////
    // constructors and initialization
    //
    public MyNewAlgorithm()
    {
        // any data structure initilization
    }

    public void init()
    {
        // this overrides the SRSim init method, which draws
        /// everything. so make sure to call the super class
        // constructor. if there is nothing to add, this method
        // doesn’t need to be overridden.
        super.init();

        // add other swing components or whatever here

        // init your graphics objects here, or override the scene
        // graph creation methods as shown below. or encapsulate the
        // graphics stuff inside your objects (pass in a reference to
        // the scene graph and let them add themselves).
    }


    //////////////////////////////////////////////////////////////////
    // public methods
    //


    //////////////////////////////////////////////////////////////////
    // protected methods
    //
    protected BranchGroup createSceneGraph()
    {
        // overrides SRSim method and adds custom objects
        super.createSceneGraph();
        SlidingCube cube = new SlidingCube();
        objRoot.addChild(cube); // how to add an example cube

        // example of adding a custom behavior
        RobotBehavior behave = new RobotBehavior(cube);
        // won’t get scheduled without this!
        behave.setSchedulingBounds(new BoundingSphere());
        objRoot.addChild(behave);

        return(objRoot);
    }


    //////////////////////////////////////////////////////////////////
    // private methods
    //


    //////////////////////////////////////////////////////////////////
    // main entry point for the app
    //
    public static void main(String[] args)
    {
        MyNewAlgorithm me = new MyNewAlgorithm();
        new MainFrame(me, 512, 512);
    }
    //*****************************************************************
    // helper classes
    //
    class RobotBehavior extends Behavior{

        WakeupOnElapsedFrames w = null;
        //WakeupOnElapsedTime w = null;
        SlidingCube cube = null;    // or robot or whatever

        public RobotBehavior(){
            w = new WakeupOnElapsedFrames(100);
            //w = new WakeupOnElapsedTime(100);
        }

        public RobotBehavior(SlidingCube cube)
        {
            this();
            this.cube = cube;
        }

        public void initialize(){
            System.out.println("behavior init"); // debug code.
                                                // remove if not
                                                // debugging.
            wakeupOn(w);
        }

        public void processStimulus(Enumeration criteria){
            System.out.println("process stimulus");  // debug code.
                                                    // remove if not
                                                    // debugging.

            // put all data updates here.
            cube.move(0.1f, 0.0f, 0.0f);

            // and reschedule yourself
            wakeupOn(w);
        }
    }
}
```

# Appendix B

# Sorting Experiment Source Code

This Appendix lists source code for the CrystalSort algorithm implementation described in Chapter 6. CrystalSort sorts a row of modules such that their colors match the modules in the row above. Section B.1 lists java source code from simulation in SRSim, and Section B.2 lists C source code that executes on the Crystal robot hardware. The simulation implementation is very similar to the hardware implementation, since the simulation environment accurately emulates the software architecture we built for the robot.

## B.1 Simulation Implementation

The code listed below implements the message handlers for the CrystalSort algorithm. This class is used in conjunction with a wrapper class that extends SRSimBase, which is not listed but is based on the template given in Appendix A.

```
/*********************************************************************
 * CrystalThread.java
 *
 * Message handlers for CrystalSort algorithm
 *
 * RCF  Robert Fitch
 *
 *  06/19/03    RCF     Initial version
 *
 * NOTES:
 *
*********************************************************************/
import java.util.Vector;
import java.util.Stack;
import javax.vecmath.*;


public class CrystalThread extends Thread {
    //////////////////////////////////////////////////////////////////
    // default params (public constants)
    public static final int TOP             = 0;
    public static final int BOTTOM          = 1;
    public static final int LEFT            = 2;
    public static final int RIGHT           = 3;
    public static final int FRONT           = 4;
    public static final int BACK            = 5;
    public static final int TOP_FRONT       = 6;
    public static final int TOP_BACK        = 7;
    public static final int TOP_LEFT        = 8;
    public static final int TOP_RIGHT       = 9;
    public static final int FRONT_RIGHT     = 10;
    public static final int FRONT_LEFT      = 11;
    public static final int BACK_RIGHT      = 12;
    public static final int BACK_LEFT       = 13;
    public static final int BOTTOM_FRONT    = 14;
    public static final int BOTTOM_BACK     = 15;
    public static final int BOTTOM_LEFT     = 16;
    public static final int BOTTOM_RIGHT    = 17;

    public static final int EXPAND          = 0;
    public static final int CONTRACT        = 1;
    public static final int PULL            = 2;
    public static final int PUSH            = 3;
    public static final int INCH            = 4;
    public static final int INCH_DONE       = 5;

    public static long animationDelay       = 1;
    public static boolean debug             = true;
    public static boolean testing           = false;
    public static Physics physics           = null;


    //////////////////////////////////////////////////////////////////
    // member vars
    //
    private Crystal m;
    private CrystalThread[] neighbors;
    private boolean stop = false;
    private boolean contracted = false;
    private boolean inching = false;
    private int inchDimension;
    private int inchDirection;
    private int inchSteps;
    private int inchOriginalPosition;
    private int inchCurrentPosition;
    private int inchDestinationPosition;
    private boolean inchRow;

    // swap
    private boolean swapMaster = false;
    private int swapState = -1;

    public MessageQueue messages;


    //////////////////////////////////////////////////////////////////
    // constructors
    //
    public CrystalThread()
    {
        messages = new MessageQueue();
        neighbors = new CrystalThread[6];
    }

    public CrystalThread(Crystal m)
    {
        this();
        this.m = m;
    }

    public CrystalThread(Crystal m, String name)
    {
        this(m);
        this.setName(name);
    }


    //////////////////////////////////////////////////////////////////
    // public methods
    //
    public void run()
    {
        Message msg;

        // fire up msg loop
        while(true)
        {
            msg = (Message)messages.pop();  // pops first element
```

```
                                    // of queue
            if (stop)
        {
            if (msg.type==Message.KILL) //die
                return;
            else
                continue;
        }

        switch (msg.type)
        {
        case Message.START:
            debugPrint("got message of type: START, id " +msg.id);
            handleStart();
            break;
        case Message.STOP:
            stop = true;
            break;
        case Message.SORT_TOKEN:
            handleSort(msg);
            break;
        case Message.SORT_QUERY:
            handleSortQuery(msg);
            break;
        case Message.COMMAND:
            handleCommand(msg);
            break;
        case Message.STATE:
            handleStateChange(msg);
            break;
        case Message.SWAP:
            handleSwap(msg);
            break;
        default:
            debugPrint("unknown message type received.");
        }

        try
        {
            Thread.sleep(1);
        }
        catch (InterruptedException e) {}
    }
}

public CrystalThread getNeighbor(int n)
{
    return neighbors[n];
}

public void setNeighbor(int whichNeighbor, CrystalThread neighbor)
```

```
{
    if (neighbor != null)
        neighbors[whichNeighbor] = neighbor;
}

public void disconnect(CrystalThread neighbor)
{
    for (int i=0; i<6; i++)
    {
        if (neighbors[i] == neighbor)
            neighbors[i] = null;
    }
}

public void dumpNeighbors()
{
    debugPrint("neighbors: ");
    for (int i=0; i<6; i++)
    {
        debugPrint(neighbors[i] + ", ");
    }
    debugPrint("");
}

public SlidingCube getGeometry()
{
    return m;
}

public int getLabel()
{
    return m.getLabel();
}

public boolean isContracted()
{
    return contracted;
}


//////////////////////////////////////////////////////////////////
// private methods
//
private void refreshNeighbors()
{
    for (int i=0;i<6;i++)
        if (neighbors[i]!=null) neighbors[i].disconnect(this);
    neighbors = physics.getNewNeighbors(this);
    for (int i=0;i<6;i++)
        if (neighbors[i]!=null) {
            neighbors[i].neighbors[oppositeNeighbor(i)] = this;
        }
}

private void broadcast(Message msg)
{
    for (int i=0; i<6; i++)
    {
        if (neighbors[i] != null)
            neighbors[i].messages.send(msg);
    }
}

private void handleStart()
{
    debugPrint("starting.");

    //Message msg = new Message(Message.SWAP);
    //msg.data = new SwapData(1,4,3);

    Message msg = new Message(Message.SORT_TOKEN);
    msg.data = new SortData(-1,1,-1);

    this.messages.send(msg);
    //neighbors[TOP].messages.send(msg);
}

private void handleSort(Message msg)
{
    SortData data = (SortData)msg.data;

    //debugPrint("sort " + data.goalPosition +","+m.getID());

    // we'll have to pass type in a state msg
    if (neighbors[BOTTOM].m.getID() == m.getID())
    {
        // rock on
        refreshNeighbors();
        if (neighbors[RIGHT]!=null)
        {
            ((SortData)msg.data).goalPosition++;
            neighbors[RIGHT].messages.send(msg);
        }
         else
         {
            debugPrint("all done!");
            //////////////////////////////////
            //
            //  all done!!! woo hoo!!!
            //
            //////////////////////////////////
         }
    }
    else
    {
        // send out a query
        Message newMsg = new Message(Message.SORT_QUERY);
        newMsg.data =
            new SortData(data.goalPosition,
                         data.goalPosition,m.getID());
        neighbors[BOTTOM].messages.send(newMsg);
    }
}

private void handleSortQuery(Message msg)
{
    SortData data = (SortData)msg.data;

    //debugPrint("sort query " + data.startPosition+","
    //     +data.goalPosition+","+data.id+", "+m.getID());

    // let’s see if we match
    if (data.id == m.getID())
    {
        // we match. fire up a swap
        debugPrint("swap ("+data.startPosition+","+
                   data.goalPosition+")");
        Message newMsg = new Message(Message.SWAP);
        newMsg.data = new SwapData(1,
                                   data.startPosition,
                                   data.goalPosition);
        this.messages.send(newMsg);
    }
    else
    {
        // not us. pass it on
        data.startPosition++;
        if (neighbors[RIGHT]!=null)
            neighbors[RIGHT].messages.send(msg);
        else
        {
            // uh oh.
            System.out.println("error. sort query fails!");
            System.exit(1);
        }
    }
}

private void handleCommand(Message msg)
{
    CommandData data = (CommandData)msg.data;

    switch(data.action)
```

```
{
case CONTRACT:
    contract(data.dimension, data.direction);
    if (data.dimension==Crystal.Y)
    {
        if(neighbors[TOP]!=null)
        {
            neighbors[TOP].messages.send(msg);
        }
    }
    else if (data.dimension==Crystal.X)
    {
        if(neighbors[RIGHT]!=null)
        {
            neighbors[RIGHT].messages.send(msg);
        }
    }
    break;
case EXPAND:
    expand(data.dimension, data.direction);
    if (data.dimension==Crystal.Y)
    {
        if(neighbors[TOP]!=null)
            neighbors[TOP].messages.send(msg);
    }
    else if (data.dimension==Crystal.X)
    {
        if(neighbors[RIGHT]!=null)
            neighbors[RIGHT].messages.send(msg);
    }
    break;
case PULL:
    if (data.dimension==Crystal.Y)
    {
        if (data.direction == Crystal.POSITIVE)
        {
            m.move(0.0f,0.05f,0.0f);
            if(neighbors[BOTTOM]!=null)
                neighbors[BOTTOM].messages.send(msg);
        }
        else
        {
            m.move(0.0f,-0.05f,0.0f);
            if(neighbors[TOP]!=null)
                neighbors[TOP].messages.send(msg);
        }
    }
    else if (data.dimension==Crystal.X)
    {
        if (data.direction == Crystal.POSITIVE)
        {
            m.move(0.05f,0.0f,0.0f);
            if(neighbors[LEFT]!=null)
                neighbors[LEFT].messages.send(msg);
        }
        else
        {
            m.move(-0.05f,0.0f,0.0f);
            if(neighbors[RIGHT]!=null)
                neighbors[RIGHT].messages.send(msg);
        }
    }
    break;
case PUSH:
    if (data.dimension==Crystal.Y)
    {
        if (data.direction == Crystal.POSITIVE)
        {
            m.move(0.0f,-0.05f,0.0f);
            if (neighbors[BOTTOM]!=null)
                neighbors[BOTTOM].messages.send(msg);
        }
        else
        {
            m.move(0.0f,0.05f,0.0f);
            if (neighbors[TOP]!=null)
                neighbors[TOP].messages.send(msg);
        }
    }
    else if (data.dimension==Crystal.X)
    {
        if (data.direction == Crystal.POSITIVE)
        {
            m.move(-0.05f,0.0f,0.0f);
            if (neighbors[LEFT]!=null)
                neighbors[LEFT].messages.send(msg);
        }
        else
        {
            m.move(0.05f,0.0f,0.0f);
            if (neighbors[RIGHT]!=null)
                neighbors[RIGHT].messages.send(msg);
        }
    }
    break;
case INCH:
    inchRow = true;
    if (!inching)
    {
        inching = true;
        inchDirection = data.direction;
        inchDimension = data.dimension;
        inchSteps     = data.numberOfSteps;
        if (data.dimension==Crystal.X)
        {
            Message m;
            if ((neighbors[LEFT]!=null) && (msg.from != LEFT))
            {
                m = new Message(Message.COMMAND);
                m.data = new CommandData(INCH,
                                         inchDimension,
                                         inchDirection,
                                         inchSteps);
                m.from = RIGHT;
                neighbors[LEFT].messages.send(msg);
            }
            if ((neighbors[RIGHT]!=null) &&
                (msg.from != RIGHT))
            {
                m = new Message(Message.COMMAND);
                m.data = new CommandData(INCH,
                                         inchDimension,
                                         inchDirection,
                                         inchSteps);
                m.from = LEFT;
                neighbors[RIGHT].messages.send(msg);
            }
        }
        doInchRules(msg);
    }
    break;
case INCH_DONE:
    if (inchRow)
    {
        inchRow = false;
        refreshNeighbors();
    }

    if ((data.syncActive==true) && (swapState>0))
    {
        data.syncActive = false;
        Message newMsg = new Message(Message.SWAP);
        newMsg.data = new SwapData(swapState);
        this.messages.send(newMsg);
    }

    if (neighbors[TOP]!=null)
        neighbors[TOP].messages.send(msg);

    int d = LEFT;
    if (data.direction==Crystal.NEGATIVE)
        d = RIGHT;
        if (neighbors[d]!=null)
            neighbors[d].messages.send(msg);

        break;
    }
}

private void handleStateChange(Message msg)
{
    if (inching)    doInchRules(msg);
}

private void doInchRules(Message msg)
{
    int positiveNeighbor, negativeNeighbor;

    // rules
    if (inchDimension == Crystal.X)
    {
        if (inchDirection==Crystal.POSITIVE)
        {
            positiveNeighbor = RIGHT;
            negativeNeighbor = LEFT;
        }
        else
        {
            positiveNeighbor = LEFT;
            negativeNeighbor = RIGHT;
        }

        if (!contracted && (neighbors[negativeNeighbor]==null) ||
            (neighbors[negativeNeighbor].isContracted()))
        {
            contract(inchDimension, inchDirection);
            broadcast(new Message(Message.STATE));

            // head
            if (neighbors[positiveNeighbor]==null)
            {
                expand(inchDimension, inchDirection, false);

                if (inchSteps>1)
                {
                    inchSteps--;
                    Message newMsg = new Message(Message.COMMAND);
                    newMsg.data = new CommandData(INCH,
                                                  inchDimension,
                                                  inchDirection,
                                                  inchSteps);
                    this.messages.send(newMsg);
                }
                else
                {
                    refreshNeighbors();
                    Message newMsg = new Message(Message.COMMAND);
                    newMsg.data = new CommandData(INCH_DONE,
                                                  inchDimension,
                                                  inchDirection);
                    ((CommandData)newMsg.data).syncActive = true;
                    this.messages.send(newMsg);
                }
            }
            else
            {
                expand(inchDimension, inchDirection);
            }
            inching = false;
        }
    }
}

private void contract(int dimension, int direction)
{
    if (contracted)
    {
        System.out.println("got command to contract
                    but i’m already contracted.");
        return;
    }

    try { Thread.sleep(animationDelay); }
    catch (InterruptedException e) {}

    m.contract(dimension);
    contracted = true;

    if (dimension==Crystal.Y)
    {
        Message newMsg = new Message(Message.COMMAND);
        newMsg.data = new CommandData(PULL, Crystal.Y, direction);

        if (direction==Crystal.POSITIVE)
        {
            m.move(0.0f,0.025f,0.0f);
            if(neighbors[BOTTOM]!=null)
                neighbors[BOTTOM].messages.send(newMsg);
        }
        else
        {
            m.move(0.0f,-0.025f,0.0f);
            if(neighbors[TOP]!=null)
                neighbors[TOP].messages.send(newMsg);
        }
    }
    else if (dimension==Crystal.X)
    {
        Message newMsg = new Message(Message.COMMAND);
        newMsg.data = new CommandData(PULL, Crystal.X, direction);

        if (direction==Crystal.POSITIVE)
        {
            m.move(0.025f,0.0f,0.0f);
            if(neighbors[LEFT]!=null)
            {
                neighbors[LEFT].messages.send(newMsg);
            }
        }
        else
        {
            m.move(-0.025f,0.0f,0.0f);
            if(neighbors[RIGHT]!=null)
                neighbors[RIGHT].messages.send(newMsg);
        }
    }
}

// overload expand so we can override the push back
private void expand(int dimension, int direction)
{   expand(dimension,direction,true);
}
private void expand(int dimension, int direction, boolean push)
{
    if (!contracted)
    {
        System.out.println("got command to expand
                                but i’m already expanded.");
        return;
    }

    int positiveNeighbor, negativeNeighbor;

    try { Thread.sleep(animationDelay); }
    catch (InterruptedException e) {}
    m.expand(dimension);
    contracted = false;

    if (dimension==Crystal.Y)
    {
        if (direction==Crystal.POSITIVE)
        {
            m.move(0.0f,-0.025f,0.0f);
            negativeNeighbor = BOTTOM;
        }
        else
        {
            m.move(0.0f,0.025f,0.0f);
            negativeNeighbor = TOP;
        }

        if (push)
        {
            Message newMsg = new Message(Message.COMMAND);
            newMsg.data = new CommandData(PUSH,
                                          Crystal.Y,
                                          direction);
            if(neighbors[negativeNeighbor]!=null)
                neighbors[negativeNeighbor].messages.send(newMsg);
        }
        else
        {
            // fix this to get rid of redundant logic expression
            if (direction==Crystal.POSITIVE)
                m.move(0.0f,0.05f,0.0f);
            else
                m.move(0.0f,-0.05f,0.0f);
        }
    }
    if (dimension==Crystal.X)
    {
        if (direction==Crystal.POSITIVE)
        {
            m.move(-0.025f,0.0f,0.0f);
            negativeNeighbor = LEFT;
        }
        else
        {
            m.move(0.025f,0.0f,0.0f);
            negativeNeighbor = RIGHT;
        }

        if (push)
        {
            Message newMsg = new Message(Message.COMMAND);
            newMsg.data = new CommandData(PUSH,
                                          Crystal.X,
                                          direction);
            if(neighbors[negativeNeighbor]!=null)
                neighbors[negativeNeighbor].messages.send(newMsg);
        }
        else
        {
            // fix this to get rid of redundant logic expression
            if (direction==Crystal.POSITIVE)
                m.move(0.05f,0.0f,0.0f);
            else
                m.move(-0.05f,0.0f,0.0f);
        }
    }
}

private void handleSwap(Message msg)
{
    Message newMsg;
    SwapData data  = (SwapData)msg.data;

    //debugPrint ("swap case " + data.step);

    switch(data.step)
    {
    case 1: // drive into position
        // init data structures
        swapMaster = true;
        inchOriginalPosition = data.currentPosition;
        inchCurrentPosition = inchOriginalPosition;
        inchDestinationPosition = data.destinationPosition;

        newMsg = new Message(Message.SWAP);
        newMsg.data = new SwapData(2);
        this.messages.send(newMsg);
        break;
    case 2: // contract
        swapState = -1;
        if (swapMaster)
        {
            // we might have to drive over
            if (inchCurrentPosition == 4)
            {
                swapState = 2;
                newMsg = new Message(Message.COMMAND);
                newMsg.data = new CommandData(INCH,
                                              Crystal.X,
                                              Crystal.NEGATIVE,
                                              2);
                newMsg.from = RIGHT;
                this.messages.send(newMsg);
                inchCurrentPosition--;
                return;
            }

            newMsg = new Message(Message.SWAP);
            newMsg.data = new SwapData(2);
            neighbors[TOP].messages.send(newMsg);
        }
        else if (neighbors[TOP]==null)  // contract is done
        {
```

```
            contract(Crystal.Y,Crystal.POSITIVE);
            newMsg = new Message(Message.SWAP);
            newMsg.data = new SwapData(3);
            neighbors[BOTTOM].messages.send(newMsg);
        }
        else    // contract
        {
            contract(Crystal.Y,Crystal.POSITIVE);
            neighbors[TOP].messages.send(msg);
        }
        break;
    case 3: // contract done
        if (!swapMaster)
        {
            neighbors[BOTTOM].messages.send(msg);
            return;
        }

        newMsg = new Message(Message.SWAP);
        newMsg.data = new SwapData(4);

        CrystalThread topNeighbor = neighbors[TOP]; // refresh
                            //will blow away the real top neighbor
        refreshNeighbors();
        topNeighbor.neighbors[TOP] = neighbors[TOP];
        neighbors[TOP].neighbors[BOTTOM] = topNeighbor;
        neighbors[TOP] = topNeighbor;
        topNeighbor.neighbors[BOTTOM] = this;

        // force refresh cuz inching could have changed
        // neighbor relationships
        neighbors[RIGHT].refreshNeighbors();
        neighbors[LEFT].refreshNeighbors();

        neighbors[LEFT].messages.send(newMsg);
        if (inchOriginalPosition!=4)
            neighbors[RIGHT].messages.send(newMsg); //hack

        // cache state for when reconnect is done
        swapState = 5;
        break;
    case 4: // reconnect
        // route to bottom
        if (neighbors[BOTTOM]!=null)
        {
            neighbors[BOTTOM].messages.send(msg);
            return;
        }

        // there’ll be one singleton and one double
        if ((neighbors[LEFT]==null) && (neighbors[RIGHT]==null))
        {
            return;
        }
        else
        {
            // inch over
            int d;
            if (neighbors[LEFT]!=null)  d = Crystal.POSITIVE;
            else    d = Crystal.NEGATIVE;
            newMsg = new Message(Message.COMMAND);
            newMsg.data = new CommandData(INCH, Crystal.X, d, 2);
            this.messages.send(newMsg);
        }
        break;
    case 5: // reconnect done
        if (!swapMaster)
        {
            debugPrint("hey i got a SWAP step 5 but i’m not
                                                    swap master.");
            return;
        }

        // compute current local position. basically, there are
        // three modules below us now. are we on top of the 1st,
        // 2nd, or 3rd one?
        int currentPosition=-1;
        if ((inchOriginalPosition==2) ||(inchOriginalPosition==3))
            currentPosition = 2;
        else if (inchOriginalPosition==4)
            currentPosition = 3;
        else
        {
            debugPrint("error: invalid inchOriginalPosition");
            System.exit(1);
        }

        // now move right
        swapState = 6;  // get us set up for next step
        // *2 because it’s 2 inch steps per module step
        int stepsToMove =
            2*(currentPosition-inchDestinationPosition+1);
            newMsg = new Message(Message.COMMAND);
        newMsg.data = new CommandData(INCH,
                                      Crystal.X,
                                      Crystal.POSITIVE,
                                      stepsToMove);
        neighbors[BOTTOM].messages.send(newMsg);
        break;
    case 6: // split
        //
        // at this point, in the row below us all modules to the
        // right of us are correctly in position. directly below
        // us, there is either nothing, one, or two modules.
        // we need to tell them to split and move left if
        // there are any there.
        //
        swapState = -1;
        // if we don’t need to split, just go to next step
        if (neighbors[BOTTOM]==null)
        {
            data.step=9;        // dangerous to hardcode this!!!!!
            handleSwap(msg);
        }
        else    // tell them to split and pick us back up at the
                // expand step
        {
            swapState = 9;
            newMsg = new Message(Message.SWAP);
            newMsg.data = new SwapData(7);
            neighbors[BOTTOM].messages.send(newMsg);
        }

        break;
    case 7: // move left one
        //
        // we need to move left. if there is no one to help us,
        // we’ll have to recruit from the right and then have them
        // move back over.
        //
        if (neighbors[LEFT] != null) // just inch over and rock on
        {
            neighbors[RIGHT].disconnect(this);
            neighbors[RIGHT] = null;
            newMsg = new Message(Message.COMMAND);
            newMsg.data = new CommandData(INCH, Crystal.X,
                                          Crystal.NEGATIVE,
                                          2);
            neighbors[LEFT].messages.send(newMsg);
        }
        else                            // we need to recruit help
        {
            // this is a hack. but before we send inchDone,
            // we’ll check swap state and that will pre-empt the
            // swapState in the swapMaster from firing. it’s like
            // a second-tier swapState.
            swapState = 8;
            newMsg = new Message(Message.COMMAND);
            newMsg.data = new CommandData(INCH,
                                          Crystal.X,
                                          Crystal.NEGATIVE,
                                          2);
            this.messages.send(newMsg);
        }
        break;
    case 8: // move back right

        swapState = -1;
        neighbors[RIGHT].disconnect(this);
        newMsg = new Message(Message.COMMAND);
        newMsg.data = new CommandData(INCH,
                                      Crystal.X,
                                      Crystal.POSITIVE,
                                      2);
        newMsg.from = LEFT;
        neighbors[RIGHT].messages.send(newMsg);
        neighbors[RIGHT] = null;
        break;
    case 9: // expand
        swapState = -1;
        if (swapMaster)
        {
            newMsg = new Message(Message.SWAP);
            newMsg.data = new SwapData(9);
            neighbors[TOP].messages.send(newMsg);
        }
        else if (neighbors[TOP]==null)  // expand is done
        {
            expand(Crystal.Y,Crystal.POSITIVE);
            newMsg = new Message(Message.SWAP);
            newMsg.data = new SwapData(10);
            neighbors[BOTTOM].messages.send(newMsg);
        }
        else    // expand
        {
            expand(Crystal.Y,Crystal.POSITIVE);
            neighbors[TOP].messages.send(msg);
        }
        break;
    case 10: // expand is done, now realign
        if (!swapMaster)
        {
            neighbors[BOTTOM].messages.send(msg);
            return;
        }

        refreshNeighbors();

        // realign
        if (inchCurrentPosition!=inchDestinationPosition)
        {
            swapState = 11; // pick up after inch
            newMsg = new Message(Message.COMMAND);
            newMsg.data =
                new CommandData(INCH,
                                Crystal.X,
                                Crystal.NEGATIVE,
                                2*(inchCurrentPosition-
                                   inchDestinationPosition));
            this.messages.send(newMsg);
        }
        else
        {
            data.step=11;       // dangerous to hardcode this!!!!!
            handleSwap(msg);
        }
        break;
    case 11: // all done! signal middle row
        swapState = -1;
        newMsg = new Message(Message.SORT_TOKEN);
        newMsg.data = new SortData(-1,inchDestinationPosition,-1);
        neighbors[TOP].messages.send(newMsg);
        break;
    }
}

// utilities
private void blink(long milliseconds)
{
    m.setAppearance(SlidingCube.TRANSPARENT);
    try { Thread.sleep(milliseconds); }
    catch (InterruptedException e) {}
    m.setAppearance(SlidingCube.SOLID);
}

private int oppositeNeighbor(int n)
{
    switch(n)
    {
    case TOP:   return BOTTOM;
    case BOTTOM: return TOP;
    case FRONT: return BACK;
    case BACK: return FRONT;
    case LEFT: return RIGHT;
    case RIGHT: return LEFT;
    }
    debugPrint("bogus neighbor passed to oppositeNeighbor: " + n);
    return -1;
}

private void debugPrint(String s)
{   if (debug) System.out.println(s);   }

//////////////////////////////////////////////////////////////////
// helper classes
```

```
//
class CommandData
{
    public int action;
    public int dimension;
    public int direction;
    public int numberOfSteps=0;
    public boolean syncActive = false;

    public CommandData(){}
    public CommandData(int action, int dimension)
    {
        this.action = action;
        this.dimension = dimension;
    }
    public CommandData(int action, int dimension, int direction)
    {
        this(action, dimension);
        this.direction = direction;
    }
    public CommandData(int action, int dimension,
                       int direction, int numSteps)
    {
        this(action, dimension, direction);
        this.numberOfSteps = numSteps;
    }
}

class SwapData
{
    public int step;
    public int currentPosition;
    public int destinationPosition;

    public SwapData(){}
    public SwapData(int step)
    {
        this.step = step;
    }
    public SwapData(int step, int currentPosition,
                    int destinationPosition)
    {
        this(step);
        this.currentPosition = currentPosition;
        this.destinationPosition = destinationPosition;
    }
}
class SortData
{
    public int startPosition;
    public int goalPosition;
```

```
        public int id;

        public SortData(){}
        public SortData(int start, int goal, int id)
        {
            startPosition = start;
            goalPosition = goal;
            this.id = id;
        }
    }
}
```

## B.2 Hardware Implementation

Below is the C source code that implements the CrystalSort algorithm on the Crystal robot. Section B.2.1 lists the main message loop and message handlers, and Section B.2.2 lists utility function implementations.

### B.2.1 Message Handlers

```
/**********************************************************************
*
*   sort/version1/sort.c
*
*   Demo of heterogeneous reconfiguration algorithm.  Assumes 3x4
*   grid, sorts bottom row according to middle row.
*
*   Robert Fitch    08/11/03
*
**********************************************************************/

#include "..\..\Shared\StAlone\monbreak.h"
#include "..\..\Shared\StAlone\inlines.h"
#include "..\..\Shared\StAlone\h83644f.h"
#include "..\..\Shared\StAlone\macros.h" // for fifos, nbrs, msg types
                                        // (which need to be changed!)

// msgs defs (reuses some type numbers defined in macros.h)
#define PING        0x0101
#define SWAP        0x0102
#define INCH        0x0103
#define INCH_DONE   0x0104
```

```
#define STATE       0x0105
#define DISCONNECT  0x0106
#define CONNECT     0x0107
#define ACK         0x0108
#define SORT_TOKEN  0x0109
#define SORT_QUERY  0x010A
#define REQUEST_ID  0x010B

void error(int err);
#include "utils.c"                 // includes ..\Shared\mylib16.c

// other constants
#define TRUE 1
#define FALSE 0
#define BLUE 0  // class ids
#define WHITE 1
#define GREEN 2
#define YELLOW 3


//////////////////////////////////////////////////////////////////////
// module level vars

// pseudo constants
unsigned int TOP        =   SOUTH;
unsigned int BOTTOM     =   NORTH;
unsigned int LEFT       =   WEST;
unsigned int RIGHT      =   EAST;
unsigned int LEFT_RIGHT =   EAST_WEST;
unsigned int TOP_BOTTOM =   NORTH_SOUTH;


// app
unsigned char bSwapMaster = 0;
int nInchOriginalPosition;
int nInchCurrentPosition;
int nInchDestinationPosition;
int nSwapState=-1;
unsigned char bInching = 0;
unsigned char bInchRow = 0;
unsigned int nInchDirection;
unsigned int nInchSteps;
unsigned char bReconnectInchHead = FALSE;
unsigned int nInchDoneCount = 0;

// system
unsigned char gInitDone = 0;
unsigned int pingCount = 0;
unsigned char bError = FALSE;
int nErrorNumber = -1;
```

```
char id_label[3] = "id=";
unsigned char id = WHITE;

void doInchRules();


//////////////////////////////////////////////////////////////////////
// system message handlers
//
void preInit()
{
    // just set global variable to false to enable further SYSINIT msg
    // and propogate msg
    // reset neighbors here, they can be set later by SYSINIT msg
    int i;

    blinkLight(5000);

    for (i=0;i<4 ;i++ )
    {
        neighbors[i] = FALSE;
        connected[i] = FALSE;
    }
    gInitDone = 0;
    initFifo();

    putMsgsInFifo();
    writeWordtoNorthUART(PREINIT);
    putMsgsInFifo();
    writeWordtoEastUART(PREINIT);
    putMsgsInFifo();
    writeWordtoSouthUART(PREINIT);
    putMsgsInFifo();
    writeWordtoWestUART(PREINIT);
}

void systemInit(unsigned int msg)
{
    // do any system initialization, propogate the msg, and turn off
    // the light. but first we add the neighbor who sent this to us,
    // as long as it’s not the ghost module
    unsigned int sender, face;

    if (fromGhostModule(msg))   // eliminates need for separate ghost
                                // neighbor code
    {
        blinkLight(5000);blinkLight(5000);
    }
    else                        // normal
    {
        lightOn();
        sender = getMessageSender(msg);
        // only connect the active faces. otherwise we get deadlock.
        if ((sender==NORTH) || (sender==WEST))
        {
            connect_WithTimeout(sender,20); //timeout after
                                            // like 20 seconds
        }
        nbr_state[sender] = 0x0003;
        lightOff();
    }

    // if this is not the first SYSINIT, ignore it.
    if (gInitDone)
        return;

    // keep reading the uart’s because it seems we can’t read
    // and write at the same time. if we try to write and there’s
    // something in the uart, one of the bytes gets lost.
    for (face=0;face<4;face++)
    {
        putMsgsInFifo();
        putMsgsInFifo();
        sendMessage(SYSINIT,face);
    }
    pingCount = 0;
    gInitDone = 1;
}


//////////////////////////////////////////////////////////////////////
// Application message handlers
//
void handleSwap(unsigned int msg)
{
    unsigned int d;
    int currentPosition=-1, stepsToMove;
    int i; //debug

    //for(i=0;i<getMessageData_Step(msg);i++)
    //{
    //  blinkLight(5000);
    //}

    switch (getMessageData_Step(msg))
    {
        case 1: // init and drive into position
            //blinkLight(20000);
            bSwapMaster = TRUE;
            nInchOriginalPosition = getMessageData_StartPosition(msg);
            nInchCurrentPosition = nInchOriginalPosition;
            nInchDestinationPosition =
                getMessageData_GoalPosition(msg);

            // we might have to drive over to align
            if (nInchCurrentPosition == 4)
            {
                nSwapState = 2;
                sendMessage_Inch(LEFT, 2, ME);
                nInchCurrentPosition--;
            }
            else    // else continue
            {
                sendMessage_Swap(2,
                                 nInchOriginalPosition,
                                 nInchDestinationPosition,
                                 ME);
            }
            break;
        case 2: // contract
            if (bSwapMaster)
            {
                connect(TOP);
                disconnect(LEFT);
                disconnect(RIGHT);
                contract_short(LEFT_RIGHT);
                sendMessage_Swap(2,
                                 nInchOriginalPosition,
                                 nInchDestinationPosition,
                                 TOP);
            }
            else if (!neighbors[TOP])   // contract is done
            {
                disconnect(LEFT);
                disconnect(RIGHT);
                contract(NORTH_SOUTH);
                sendMessage_Swap(3,
                                 nInchOriginalPosition,
                                 nInchDestinationPosition,
                                 BOTTOM);
            }
            else    // contract
            {
                disconnect(LEFT);   // disconnect first just in case
                disconnect(RIGHT);
                sendMessage_Swap(2,
                                 nInchOriginalPosition,
                                 nInchDestinationPosition,
                                 TOP);
                contract(NORTH_SOUTH);
            }
            break;
        case 3: // contract done
            if (!bSwapMaster)
            {
                sendMessage_Swap(3,0,0,BOTTOM);
                return;
            }

            expand_short(LEFT_RIGHT);
            connect(LEFT);
            connect(RIGHT);

            if (nInchOriginalPosition!=4) //hack
                sendMessage_Swap(4,0,0,RIGHT);
            sendMessage_Swap(4,0,0,LEFT);

            // cache state for when reconnect is done
            nSwapState = 5;
            break;
        case 4: // reconnect
            // route to bottom
            connect(BOTTOM);  // this will time out if no one is there
            if (neighbors[BOTTOM])
            {
                sendMessage_Swap(4,0,0,BOTTOM);
                return;
            }

            // there’ll be one singleton and one double
            if (!neighbors[LEFT] && (!neighbors[RIGHT]))
            {
                return;
            }
            else
            {
                // inch over
                if (neighbors[LEFT])
                    d = RIGHT;
                else
                    d = LEFT;

                bReconnectInchHead = TRUE;  // tells us to reconnect
                                            // head when inch is done
                sendMessage_Inch(d, 2, ME);
            }
            break;
        case 5: // reconnect done
            if (!bSwapMaster)
            {
                error(-1);
                return;
            }
            // compute current local position. basically,
            // there are three modules below us now. are we on top of
            // the 1st, 2nd, or 3rd one?
            if ((nInchOriginalPosition==2) ||
                (nInchOriginalPosition==3)) {
                    currentPosition = 2;
            } else if (nInchOriginalPosition==4)
                currentPosition = 3;
            else
            {
                error(-1);
                return;;
            }

            // now move right
            nSwapState = 6; // get us set up for next step
            // *2; 2 inch steps per module step
            stepsToMove =
                2*(currentPosition-nInchDestinationPosition+1);
            sendMessage_Inch(RIGHT, stepsToMove, BOTTOM);
            break;

        case 6: // split
            //
            // at this point, in the row below us all modules to the
            // right of us are correctly in position. directly below
            // us, there is either nothing, one or two modules.
            // we need to tell them to split and move left if
            // there are any there.
            //
            // if we don’t need to split, just go to next step
            if (nInchDestinationPosition==1)    // meaning there’s
                                                // no one below us
            {
                msg &= 0xF0FF;                  // clears step
                msg |= ((9 & 0x000F) << 8);     // dangerous to
                                                // hardcode this!!!!!
                wait(15000);
                handleSwap(msg);
            }
            else    // tell them to split and pick us back up at the
                    // expand step
            {
                nSwapState = 9;
                sendMessage_Swap(7,0,0,BOTTOM);
            }

            break;
        case 7: // move left one
            //
            // we need to move left. if there is no one to help us,
            // we’ll have to recruit from the right and then have them
            // move back over.
            //
            if (neighbors[LEFT])    // just inch over and rock on
            {
                disconnect(RIGHT);
                sendMessage_Inch(LEFT, 2, ME);
            }
            else                            // we need to recruit help
            {
                // this is a hack. but before we send inchDone, we’ll
                // check swap state and that will pre-empt the
                // nSwapState in theswapMaster from firing.
                // it’s like a second-tier nSwapState.
                nSwapState = 8;
                sendMessage_Inch(LEFT, 2, ME);
            }
            break;
        case 8: // move back right (split)
            //
            //  tricky because this is basically a split.
            //
            connect(TOP);
            disconnect(RIGHT);
            sendMessage_Inch(RIGHT, 2, RIGHT);
            break;
        case 9: // expand
            if (bSwapMaster)
            {
                disconnect(LEFT);
                disconnect(RIGHT);
                contract_short(LEFT_RIGHT);
                sendMessage_Swap(9,0,0,TOP);
            }
            else if (!neighbors[TOP])   // expand is done
            {
                expand(NORTH_SOUTH);
                sendMessage_Swap(10,0,0,BOTTOM);
                connect(LEFT);
                connect(RIGHT);
            }
            else    // expand
            {
                sendMessage_Swap(9,0,0,TOP);
                expand(NORTH_SOUTH);
                connect(LEFT);
                connect(RIGHT);
            }
            break;
        case 10: // expand is done, now realign
            if (!bSwapMaster)
```

```
            {
                sendMessage_Swap(10,0,0,BOTTOM);
                return;
            }

            expand_short(LEFT_RIGHT);

            // realign
            if (nInchDestinationPosition>1)
            {
                connect(LEFT);
            }
            if (nInchDestinationPosition<4)
            {
                connect(RIGHT);
            }

            if (nInchCurrentPosition!=nInchDestinationPosition)
            {
                nSwapState = 11;    // pick up after inch
                stepsToMove =
                    2*(nInchCurrentPosition-nInchDestinationPosition);
                sendMessage_Inch(LEFT, stepsToMove, ME);
            }
            else
            {
                msg &= 0xF0FF;
                // dangerous to hardcode this!!!!!
                msg |= ((11 & 0x000F) << 8)
                handleSwap(msg);
            }
            break;
        case 11: // all done! signal middle row
            sendMessage_Sort(SORT_TOKEN,
                             0,
                             nInchDestinationPosition,
                             0,
                             TOP);  //start,goal,id
            break;
        default:    // this should really never happen
            error(-1);
            break;
    }
}

void handleInch(unsigned int msg)
{
    unsigned int nFrom = getMessageSender(msg);
    bInchRow = TRUE;

    if (!bInching)
    {
        blinkLight(5000);

        bInching        = TRUE;
        nInchDirection  = getMessageData_InchDirection(msg);
        nInchSteps      = getMessageData_InchSteps(msg);

        if((nInchDirection!=nFrom) && neighbors[nInchDirection]) {
            sendMessage_Inch(nInchDirection, nInchSteps,
                             nInchDirection);
        }
        if((oppositeNeighbor(nInchDirection)!=nFrom) &&
           neighbors[oppositeNeighbor(nInchDirection)]) {
            sendMessage_Inch(nInchDirection, nInchSteps,
                             oppositeNeighbor(nInchDirection));
        }
        doInchRules();
    }
}

void handleInchDone(unsigned int msg)
{
    int d;
    unsigned char bSyncActive = getMessageData_InchSyncActive(msg);
    int nCount = getMessageData_InchDoneCount(msg);


    if (nCount <= nInchDoneCount)
    {
        return;
    }

    blinkLight(10000);

    if (bInchRow)
        bInchRow = FALSE;

    if (bReconnectInchHead)
    {
        connect(nInchDirection);
        bReconnectInchHead = FALSE;
    }

    if (bSyncActive && (nSwapState>0))
    {
        sendMessage_Swap(nSwapState,1,1,ME);
        nSwapState = -1;
        bSyncActive = FALSE;
    }

    sendMessage_InchDone(bSyncActive, nCount, ALL);
    nInchDoneCount = nCount;
}

void handleState(unsigned int msg)
{
    blinkLight(1000);
    nbr_state[getMessageSender(msg)] = getNbrState(msg);

    if (bInching)
        doInchRules();
}

void handlePing(unsigned int msg)
{
    int i;
    unsigned int data = getMessageData(msg);

    if (data <= pingCount)
    {
        return;
    }

    blinkLight(5000);
    broadCast(msg);
    pingCount = data;
}

void handleSort(unsigned int msg)
{
    unsigned int ack, nGoalPosition;

    blinkLight(10000);

    // get id of neighbor. block because we can’t do anything else
    // in the mean time
    sendMessage(REQUEST_ID, BOTTOM);
    ack = block(ACK, 20);
    if (!ack)
    {
        // we timed out. throw an error.
        error(-1);
        return;
    }

    nGoalPosition = getMessageData_GoalPosition(msg);

    if (getMessageData_Id(ack) == id)
    {
        // rock on
        blinkLight(5000);blinkLight(5000);blinkLight(5000);
        connect(RIGHT);
        if (neighbors[RIGHT])
        {
            sendMessage_Sort(SORT_TOKEN,0,nGoalPosition+1, 0, RIGHT);
        }
        else
        {
            blinkLight(20000);blinkLight(20000);blinkLight(20000);
            //////////////////////////////////
            //
            //  all done!!! woo hoo!!!
            //
            //////////////////////////////////
        }
    }
    else
    {
        // send out a query
        sendMessage_Sort(SORT_QUERY, nGoalPosition,
                         nGoalPosition, id, BOTTOM);
    }
}

void handleSortQuery(unsigned int msg)
{
    unsigned int nStartPosition, nGoalPosition;
    nStartPosition = getMessageData_StartPosition(msg);
    nGoalPosition = getMessageData_GoalPosition(msg);

    blinkLight(5000);

    // let's see if we match
    if (getMessageData_ClassID(msg) == id)
    {
        // we match. fire up a swap
        blinkLight(5000);blinkLight(5000);blinkLight(5000);
        sendMessage_Swap(1, nStartPosition, nGoalPosition, ME);
    }
    else
    {
        // not us. pass it on
        if (neighbors[RIGHT])
            sendMessage_Sort(SORT_QUERY, nStartPosition+1,
                             nGoalPosition,
                             getMessageData_ClassID(msg), RIGHT);
        else
        {
            // uh oh.
            error(-1);  // sort query failed
            return;
        }
    }
```

```
}

void handleRequestId(unsigned int msg)
{
    // just send the id in the ack
    sendMessage_AckWithId(id, getMessageSender(msg));
}

//////////////////////////////////////////////////////////////////////
// Helper functions
//
void doInchRules()
{
    int nInchDimension = EAST_WEST;
    int nOppositeNeighbor = oppositeNeighbor(nInchDirection);
    int i;

    if ((nInchDirection==NORTH)||(nInchDirection==SOUTH))
    {
        nInchDimension = NORTH_SOUTH;
    }

    // rules
    if (expanded[nInchDimension] &&
        (!neighbors[oppositeNeighbor(nInchDirection)])
        || (!nbr_expanded(nInchDimension, nOppositeNeighbor)))
    {
        if (nInchDimension==EAST_WEST)
        {
            disconnect(NORTH);
            disconnect(SOUTH);
        }
        else if (nInchDimension==NORTH_SOUTH)
        {
            disconnect(EAST);
            disconnect(WEST);
        }

        contract(nInchDimension);
        sendMessage_State(ALL);

        // head
        if (!neighbors[nInchDirection])
        {
            expand(nInchDimension);
            sendMessage_State(ALL);

            if (nInchSteps>1)
            {
                //for(i=0;i<nInchSteps;i++) blinkLight(10000);
                nInchSteps--;
                bInching=FALSE;
                sendMessage_Inch(nInchDirection, nInchSteps, ME);
                return;
            }
            else
            {
                sendMessage_InchDone(TRUE, nInchDoneCount+1, ME);
            }
        }
        else
        {
            expand(nInchDimension);
            sendMessage_State(ALL);
        }
        bInching = FALSE;
    }
}

void error(int err)
{
    bError = TRUE;      // this will break us out of the event loop
                        // and into the error handler
    nErrorNumber = err; //  this allows us to repeat a sequence of
                        // blinks in the error handler

    // this is broken up into two functions cuz we can’t forcibly
    // break out of the msg loop
}

void handleError()
{
    switch (nErrorNumber)
    {
    default:
        lightOn();
    }
}

//////////////////////////////////////////////////////////////////////
//  Our standard message pump
//
void msgProcessLoop()
{
    unsigned int msg = 0 ;
    unsigned int fromdir;

    while(!bError)
    {
        putMsgsInFifo();
        putMsgsInFifo();
        if (getFifo(&msg) == 0)
        {
            fromdir = getMessageSender(msg);
            switch (getMessageType(msg))
            {
                case (0)            :  break;          // no op
                case (PREINIT&0x000F):  if (gInitDone) {preInit();}
                                        break;
                case (SYSINIT&0x000F):  systemInit(msg);
                                        break;
                case (PING&0x000F):     handlePing(msg);
                                        break;
                case (SWAP&0x000F):     handleSwap(msg);
                                        break;
                case (INCH&0x000F):     handleInch(msg);
                                        break;
                case (INCH_DONE&0x000F):handleInchDone(msg);
                                        break;
                case (STATE&0x000F):    handleState(msg);
                                        break;
                case (DISCONNECT&0x000F):
                            // opens (if active face) and sets state
                                        disconnect_helper(fromdir);
                                        sendMessage(ACK, fromdir);
                                        break;
                case (CONNECT&0x000F):
                                        connect_helper(fromdir);
                                        sendMessage(ACK, fromdir);
                                        break;
                case (SORT_TOKEN&0x000F):
                                        handleSort(msg);
                                        break;
                case (SORT_QUERY&0x000F):
                                        handleSortQuery(msg);
                                        break;
                case (REQUEST_ID&0x000F):
                                        handleRequestId(msg);
                                        break;
                default:
                    blinkLight(5000); blinkLight(5000);
                    blinkLight(5000); blinkLight(5000); break;
            }
        }
    }

    // error handler. this will basically crash us, but has the
    // capability to repeatedly blink an error code if we set one
    // going down
    handleError();
}
```

```
/*********************************************************************/
// Main entry point
//
int main (void)
{
    int i;
    lightOn();
    initio();
    initUARTs();
    initFifo();
    P8DR |= 0x80;   //stop WC_MOTOR
    P8DR |= 0x02;   //stop NC_MOTOR
    NC_Close();
    wait(10000);
    wait(10000);
    wait(10000);
    NC_Open();
    WC_Close();
    wait(10000);
    wait(10000);
    wait(10000);
    WC_Open();
    lightOff();

    msgProcessLoop();

    return (0);
}
```

### B.2.2 Utility Functions

```
/**********************************************************************
*
*   General utilities for programming the Crystal modules
*   Robert Fitch, 1/29/02
*
*   Based on:
*   Distributed inchworm for crystals, 12/12/01, Zack Butler
*   Goal Recognition. Fitch, Butler, Wang, 9/01
*
*********************************************************************/

#define NORTH_SOUTH 0
#define EAST_WEST   1
#define TRUE        1
#define FALSE       0
#define ME          4
```

```
#define CONNECT_TIMEOUT 5   // timeout in seconds

#include "..\..\Shared\StAlone\mylib16.c"
#include "expand.c"         // for new expansion/contraction functions

void disconnect_helper(int face);
void connect_helper(int face);
void setStateConnected(int face);


//////////////////////////////////////////
//  global stuff
//
unsigned char expanded[2] = {1, 1}; // our state
unsigned char connected[4] = {0, 0, 1, 1}; // although only west is
                                           // used right now!
unsigned char nbr_state[4] = {0, 0, 0, 0}; // state of our nbrs:
                                           // east_west expanded,
                                           // n/s expanded
// note nbr_state is undefined at start - will be set when nbr is set.

//////////////////////////////////////////
//  LED, time utilities
//
void wait( long loop)
{
    long i,j;

    for (j=0; j<=8; j++)
    {
        for (i=0; i<= loop; i++) {;}
    }
}

void lightOn()
{
    P6DR |= 0x03;          /* set p6 0-1 high */
}

void lightOff()
{
    P6DR &= 0xFC;          /* set p6 0-1 low */
}

void blinkLight( int loop)
{
    int i;

    // turn it on for a while and check msgs
    lightOn();
    wait(loop);
    putMsgsInFifo();

    // turn it off and leave off for a bit
    lightOff();
    wait(loop);
}



//////////////////////////////////////////
//  neighbor functions
//
int oppositeNeighbor(int dir)
{
    switch(dir)
    {
        case NORTH: return SOUTH;
        case SOUTH: return NORTH;
        case EAST:  return WEST;
        case WEST:  return EAST;
        default:    return -2;  // ALL is -1
    }
}

int nbr_expanded(int dimension, int dir)
{
    switch(dimension)
    {
        case NORTH_SOUTH:
            return (nbr_state[dir] & 0x01);
        case EAST_WEST:
            return ((nbr_state[dir] & 0x02) >> 1);
        default:
            return -1;
    }
}

int nbr_CONNECT_WEST(int dir)
{
    return ((nbr_state[dir] & 0x02) >> 1);
}

int nbr_head(int dir)
{
    return ((nbr_state[dir] & 0x0C) >> 2);
}

int nbr_sig(int dir)
{
    return ((nbr_state[dir] & 0xF0) >> 4);
}
```

```
//////////////////////////////////////////
//  Messaging utilities
//
unsigned int getMessageData_Id(unsigned int wdata)
{
    return ((wdata & 0x00F0) >> 4);
}

unsigned int getMessageData_InchDirection(unsigned int wdata)
{
    return ((wdata & 0x0300) >> 8);
}

unsigned int getMessageData_InchSteps(unsigned int wdata)
{
    return ((wdata & 0x00F0) >> 4);
}

unsigned int getMessageData_InchSyncActive(unsigned int wdata)
{
    return ((wdata & 0x1000) >> 12);
}

unsigned int getMessageData_InchDoneCount(unsigned int wdata)
{
    return ((wdata & 0x0FF0) >> 4);
}

unsigned int getMessageData_StartPosition(unsigned int wdata)
{
    return ((wdata & 0x00C0) >> 6)+1;   // convert back to
                                        // one-based counting
}
unsigned int getMessageData_GoalPosition(unsigned int wdata)
{
    return ((wdata & 0x0030) >> 4);
}
unsigned int getMessageData_Step(unsigned int wdata)
{
    return ((wdata & 0x0F00) >> 8);
}
unsigned int getMessageData_ClassID(unsigned int wdata)
{
    return ((wdata & 0x0F00) >> 8);
}
unsigned int getMessageType(unsigned int wdata)
{
    return (wdata & 0x000F);
}
```

```
unsigned int getMessageSender(unsigned int wdata)
{
    return ((wdata & 0xC000) >> 14);
}

unsigned int getMessageData(unsigned int wdata)
{
    return ((wdata & 0x3F00) >> 8);
}

unsigned int getMessageID(unsigned int wdata)
{
    return ((wdata & 0x00F0) >> 4);
}

unsigned char getNbrState(unsigned int wdata)
{
    return (unsigned char)((wdata & 0x0300) >> 8);
}

void broadCast(unsigned int msg)
{
    writeWordtoEastUART(msg);
    writeWordtoSouthUART(msg);
    writeWordtoWestUART(msg);
    writeWordtoNorthUART(msg);
}

void send (unsigned int msg, int dir)
{
    switch (dir)
    {
        case ALL:
            broadCast(msg); break;
        case EAST:
            writeWordtoEastUART(msg); break;
        case SOUTH:
            writeWordtoSouthUART(msg); break;
        case WEST:
            writeWordtoWestUART(msg); break;
        case NORTH:
            writeWordtoNorthUART(msg); break;
        case ME:
            putFifo(msg|0xC000);    // hack that makes it look like
                                    // the msg is from the north in
                                    // this app, only INCH looks at
                                    // the msg sender. and, we only
                                    // inch east/west. so north is ok.
    }
}
```

```
void sendMessage_Swap ( unsigned int nStep  ,
                        unsigned int nStart ,
                        unsigned int nGoal  ,
                        int dir )
{
    unsigned int msg = 0;
    unsigned int nMessageType = SWAP;

    // convert to zero-based for transmission.
    // nGoal is ok cuz it’ll never get above 3
    nStart--;

    // build msg
    msg |= ((nStep & 0x000F) << 8);     // upper data byte
    msg |= ((nStart & 0x0003) << 6);    // start
    msg |= ((nGoal & 0x0003) << 4);     // goal
    msg |= 0x1000;                      // just so upper byte has a
                                        // 0 and a 1 in it
    msg |= (nMessageType & 0x000F);

    // and send
    send(msg,dir);
}


void sendMessage_Sort ( unsigned int nMessageType,
                        unsigned int nStart      ,
                        unsigned int nGoal       ,
                        unsigned int nClassID    ,
                        int dir )
{
    unsigned int msg = 0;

    // convert to zero-based for transmission.
    // nGoal is ok cuz it’ll never get above 3
    nStart--;

    // build msg
    msg |= ((nClassID & 0x000F) << 8);  // upper data byte
    msg |= ((nStart & 0x0003) << 6);    // start
    msg |= ((nGoal & 0x0003) << 4);     // goal
    msg |= 0x1000;                      // just so upper byte has a
                                        // 0 and a 1 in it
    msg |= (nMessageType & 0x000F);

    // and send
    send(msg,dir);
}
```

```
void sendMessage_Inch ( unsigned int nDirection ,
                        unsigned int nSteps     ,
                        int dir )
{
    unsigned int msg = 0;
    unsigned int nMessageType = INCH;

    // build msg
    msg |= ((nDirection & 0x0003) << 8);    // direction we’re inching
    msg |= ((nSteps & 0x000F) << 4);        // number of steps to inch
    msg |= 0x1000;                          // just so upper byte has
                                            // a 0 and a 1 in it
    msg |= (nMessageType & 0x000F);

    // and send
    send(msg,dir);
}

void sendMessage_InchDone (unsigned char bSyncActive,
                           unsigned int nCount,
                           int dir)
{
    unsigned int msg = 0;
    unsigned int nMessageType = INCH_DONE;

    // build msg
    msg |= ((bSyncActive & 0x0001) << 12);  // sync active
    msg |= ((nCount & 0x00FF) << 4);        // msg counter
    msg |= 0x2000;                          // just so upper byte has
                                            // a 0 and a 1 in it
    msg |= (nMessageType & 0x000F);

    // and send
    send(msg,dir);
}

void sendMessage_State(int dir)
{
    unsigned int msg = 0;
    unsigned int nMessageType = STATE;

    msg |= 0x1000;                          // just so upper byte has
                                            // a 0 and a 1 in it
    msg |= (nMessageType & 0x000F);
    msg |= ((expanded[NORTH_SOUTH] & 0x0001) << 8);  // expanded N/S?
    msg |= ((expanded[EAST_WEST] & 0x0001) << 9);  // expanded E/W?

    send(msg,dir);
}

void sendMessage_AckWithId(unsigned int id, int dir)
```

```
{
    unsigned int msg = 0;
    unsigned int nMessageType = ACK;

    msg |= 0x1000;                          // just so upper byte has
                                            /// a 0 and a 1 in it
    msg |= (nMessageType & 0x000F);
    msg |= ((id & 0x000F) << 4);            // id

    send(msg,dir);
}

void sendMessage (int type, int direction)
{
    unsigned int msg = 0;

    // build msg
    msg |= 0x1000;                          // just so upper byte has
                                            // a 0 and a 1 in it
    msg |= (type & 0x000F);

    // and send
    send(msg,direction);
}

void multiCast(unsigned int wdata)
{
    putMsgsInFifo();
    // writes msg to known neighbors only
    // order of neighbors in data structure is:
    // East, South, West, North
    if (neighbors[0])
    {
        putMsgsInFifo();
        writeWordtoEastUART(wdata);
    }
    if (neighbors[1])
    {
        putMsgsInFifo();
        writeWordtoSouthUART(wdata);
    }
    if (neighbors[2])
    {
        putMsgsInFifo();
        writeWordtoWestUART(wdata);
    }
    if (neighbors[3])
    {
        putMsgsInFifo();
        writeWordtoNorthUART(wdata);
    }
```

```
}

unsigned int fromGhostModule(unsigned int msg)
{
    // ghost module is assumed to be 16(one-based) (0xF), and we pass
    // this in the parent field of the msg
    return (0x00F0 == (msg & 0x00F0));
}

/////////////////////////////////////////
// blocks on given msg type until timeout
// for infinite (effectively) timeout, use -1
//
// this uses the hardware timer, Timer X.  even in the slowest setup,
// it’s still super fast. so what we’re actually using is the overflow
// flag, stored in bit 1 of TMRX_TCSRX. it takes about 500ms for the
// counter to overflow, so we count overflows and convert to seconds.
//
unsigned int block(unsigned int ack, int nTimeoutSeconds)
{
    int i;
    unsigned char bFound=FALSE, msg;
    long nTicks = nTimeoutSeconds*2;    // approximately 2 ticks per
                                        // second
    unsigned int ret;

    TMRX_TCRX |= 0x02;  // make sure we’re in the longest tick setting

    while ((!bFound) && (nTicks!=0))    // if nTicks is negative, it
                // won’t timeout until the long datatype rolls over.
                // that would be a long time.
    {
        TMRX_TCSRX = 0x00;  // clear timer overflow
        while (!(TMRX_TCSRX & 0x02))    // while the overflow
                                        // flag is not set
        {
            // check for incoming msgs
            putMsgsInFifo();
            putMsgsInFifo();

            // loop through queue and look for the ack
            msg = GetI;
            for (i=0;((!bFound)&&(i<Size));i++)
            {
                if (getMessageType(Fifo[msg])==(ack&0x000F))
                {
                    bFound = TRUE;
                    ret = Fifo[msg];
                    Fifo[msg] = 0;  // clear it so we don’t see
                                    // it again
                }
                else
                {
                    msg++;
                    if (msg >= FifoSize)
                    {
                        msg=0;  // wrap
                    }
                }
            }
        }
        nTicks--;
    }

    if (bFound)
        return ret;
    else
        return FALSE;
}


/////////////////////////////////////////
// motion
void contract(int dimension)
{
    int oppositeDimension = (dimension==NORTH_SOUTH) ? EAST_WEST :
                                                        NORTH_SOUTH;

    contract_helper(oppositeDimension, CON_COUNT_SHORT);
    contract_helper(dimension, CON_COUNT);
    expanded[dimension] = FALSE;
    //expand_helper(oppositeDimension, EXP_COUNT_SHORT);
}

void contract_short(int dimension)
{
    contract_helper(dimension, CON_COUNT_SHORT);
}

void expand(int dimension)
{
    int oppositeDimension = (dimension==NORTH_SOUTH) ?EAST_WEST :
                                                        NORTH_SOUTH;

    //contract_helper(oppositeDimension, CON_COUNT_SHORT);
    expand_helper(dimension, EXP_COUNT);
    expanded[dimension] = TRUE;
    expand_helper(oppositeDimension, EXP_COUNT_SHORT);
}

void expand_short(int dimension)
{
    expand_helper(dimension, EXP_COUNT_SHORT);
}

void openConnector(int face)
{
    switch (face)
    {
        case NORTH:
            NC_Open();
            break;
        case WEST:
            WC_Open();
            break;
    }
}

void closeConnector(int face)
{
    switch (face)
    {
        case NORTH:
            NC_Close();
            break;
        case WEST:
            WC_Close();
            break;
    }
}

int disconnect(int face)
{
    ////////////////////////////////////////////////////////////
    // WARNING: this function will block until it succeeds!!!!
    //
    switch (face)
    {
        case NORTH:
        case WEST:
            lightOn();
            disconnect_helper(face);    // opens and sets state
            sendMessage(DISCONNECT,face);
            if (neighbors[face])
            {
                block(ACK,-1);
            }
            lightOff();
            break;
        case SOUTH:
        case EAST:
            // send disconnect msg
            sendMessage(DISCONNECT,face);
            // block until we get the ACK
            lightOn();
            if (neighbors[face])
            {
                block(ACK,-1);
            }
            connected[face] = FALSE;
            neighbors[face] = FALSE;
            lightOff();
            break;
        default:
            break;
    }
    return TRUE;
}

void disconnect_helper(int face)
{
    switch (face)
    {
        case NORTH:
        case WEST:
            openConnector(face);
            connected[face] = FALSE;
            neighbors[face] = FALSE;
            break;
        case EAST:
        case SOUTH:
            connected[face] = FALSE;
            neighbors[face] = FALSE;
            break;
        default:
            break;
    }
}

int connect(int face)
{
    return connect_WithTimeout(face, CONNECT_TIMEOUT);
}

int connect_WithTimeout(int face, int nTimeout)
{
    ////////////////////////////////////////////////////////////
    // WARNING: this function will block until it succeeds!!!!
    //          or times out!!! (-1 for near infinite timeout)
    //
    int bRet = FALSE;

    switch (face)
    {
        case NORTH:
        case WEST:
            lightOn();
            connect_helper(face);   // close and set state
            sendMessage(CONNECT,face);
            bRet = block(ACK, nTimeout);
            if (!bRet)  // reverse
            {
                disconnect_helper(face);
            }
            lightOff();
            break;
        case SOUTH:
        case EAST:
            if (!connected[face])
            {
                // send disconnect msg
                sendMessage(CONNECT,face);

                // block until we get the ACK
                lightOn();
                bRet = block(ACK, nTimeout);
                if (!bRet)  // reverse
                {
                    disconnect_helper(face);
                }
                else
                {
                    setStateConnected(face);
                }
                lightOff();
            }
            break;
        default:
            break;
    }
    return bRet;
}

void connect_helper(int face)
{
    switch (face)
    {
        case NORTH:
        case WEST:
            closeConnector(face);
            setStateConnected(face);
            break;
        case EAST:
        case SOUTH:
            setStateConnected(face);
            break;
        default:
            break;
    }
}

void setStateConnected(int face)
{
    connected[face] = TRUE;
    neighbors[face] = TRUE;

    // we don’t really know the expansion state. let’s just
    // assume it’s expanded for now.
    nbr_state[face] = 0x0003;
}
```

# Bibliography

[1] A. Abrams and R. Ghrist. State complexes for metamorphic robot systems. *Intl. J. of Robotics Research*, to appear.

[2] L. Adleman. Towards a mathematical theory of self-assembly. Technical Report 00-722, University of Southern California, 2000.

[3] G. Beni. The concept of cellular robotic system. In *Proc. of IEEE International Symposium on Intelligent Control*, pages 57–62, 1988.

[4] G. Beni and J. Wang. Theoretical problems for the realization of distributed robotic systems. In *Proc. of IEEE International Symposium on Intelligent Control*, pages 1914–1920, 1991.

[5] Z. Butler, S. Byrnes, and D. Rus. Distributed motion planning for modular robots with unit-compressible modules. In *Proc. of the Int'l Conf. on Intelligent Robots and Systems*, 2001.

[6] Z. Butler, R. Fitch, and D. Rus. Distributed control for unit-compressible robots: Goal-recognition, locomotion and splitting. *IEEE/ASME Trans. on Mechatronics*, 7(4):418–30, Dec. 2002.

[7] Z. Butler, K. Kotay, D. Rus, and K. Tomita. Cellular automata for decentralized control of self-reconfigurable robots. In *ICRA 2001 Workshop on Modular Self-Reconfigurable Robots*, 2001.

[8] Z. Butler, K. Kotay, D. Rus, and K. Tomita. Generic decentralized control for a class of self-reconfigurable robots. In *Proc of IEEE ICRA*, 2002.

[9] 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.

[10] Z. Butler, S. Murata, and D. Rus. Distributed replication algorithms for self-reconfiguring modular robots. In *Distributed Autonomous Robotic Systems 5*, 2002.

[11] 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.

[12] Edited by T. Balch and L. Parker. *Robot Teams: From Diversity to Polymorphism*. A K Peters, Ltd., 2002.

[13] A. Cai, T. Fukuda, F. Arai, T. Ueyama, and A. Sakai. Hierarchical control architecture for cellular robotic system - simulations and experiments. In *Proc. of IEEE ICRA*, pages 1191–1196, 1995.

[14] Y. Cao, A. Fukunaga, and A. Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4:1–23, 1997.

[15] A. Castano and P. Will. Mechanical design of a module for reconfigurable robots. In *Proc. of the Int'l Conf. on Intelligent Robots and Systems*, pages 2203–2209, 2000.

[16] A. Castano and P. Will. A polymorphic robot team. In T. Balch and L. Parker, editors, *Robot Teams: From Diversity to Polymorphism*. A K Peters, Ltd., 2002.

[17] C.D.Yang, D.T. Lee, and C.K. Wong. On bends and lengths of rectilinear paths: a graph-theoretic approach. *Internat. J. Comput. Geom. Appl.*, 2:61–74, 1992.

[18] I. Chen and J. Burdick. Determining task optimal modular robot assembly configurations. In *Proc. of IEEE ICRA*, volume 1, pages 132 –137, 1995.

[19] C.-H. Chiang and G. Chirikjian. Modular robot motion planning using similarity metrics. *Autonomous Robots*, 10(1):91–106, 2001.

[20] G. Chirikjian. Kinematics of a metamorphic robotic system. In *Proc. of IEEE ICRA*, pages 449–455, 1994.

[21] G. Chirikjian and A. Pamecha. Bounds for self-reconfiguration of metamorphic robots. In *Proc. of IEEE ICRA*, pages 1452–1457, 1996.

[22] S. Chitta and J. Ostrowski. Motion planning for hetrogeneous modular mobile systems. In *Proc. of IEEE ICRA*, pages 4077–4082, 2002.

[23] J. Choi and C.K. Yap. Rectilinear geodesics in 3-space (extended abstract). In *11th Symp. Computational Geometry*, pages 380–389, 1995.

[24] 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.

[25] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, second edition, 2001.

[26] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, 2000.

[27] P.J. de Rezende, D.T. Lee, and W.F. Wu. Rectilinear shortest paths in the presnece of rectangular barriers. *Discrete and Computational Geometry*, 4:41–53, 1989.

[28] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 2nd edition, 2000.

[29] HYDRA EU. `http://www.hydra-robot.com`. WebSite, 2004.

[30] S. Farritor and S. Dubowski. On modular design of field robotic systems. *Autonomous Robots*, 10(1):57–65, 2001.

[31] J. Fax and R. Murray. Graph laplacians and stabilization of vehicle formations. In *Proc. of IFAC World Congress*, 2002.

[32] J. Fax and R. Murray. Information flow and cooperative control of vehicle formations. In *Proc. of IFAC World Congress*, 2002.

[33] 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.

[34] National Center for Supercomputing Applications. `http://www.ncsa.uiuc.edu/`. WebSite.

[35] T. Fukuda, M. Buss, and Y. Kawauchi. Communication system of cellular robot: Cebot. In *Proceedings of IECON '89: 1989 International Conference on Industrial Electronics, Control, and Instrumentation*, pages 695–700, 1989.

[36] T. Fukuda and T. Kaga. Distributed decision making of dynamically reconfigurable robotic system. In *Proc. of IEEE IROS*, pages 1604–1609, 1997.

[37] T. Fukuda and Y. Kawauchi. Cellular robotic system (CEBOT) as one of the realization of self-organizing intelligent universal manipulator. In *Proc. of IEEE ICRA*, pages 662–7, 1990.

[38] T. Fukuda and S. Nakagawa. A dynamically reconfigurable robotic system (concept of a system and optimal configurations). In *Proceedings of IECON '87: 1987 International Conference on Industrial Electronics, Control, and Instrumentation*, 1987.

[39] T. Fukuda, S. Nakagawa, Y. Kawauchi, and M. Buss. Self organising robots based on cell structures - cebot. In *Proc. 1988 IEEE International Workshop on Intelligent Robots and Systems*, pages 145–150, 1988.

[40] T. Fukuda, T. Ueyama, and F. Arai. Control strategy for a network of cellular robots - determination of a master cell for cellular robotic network based on a potential energy. In *Proc. of IEEE ICRA*, pages 1616–1621, 1991.

[41] C. Guéganno and D. Duhaut. A hardware/software architecture for the control of self-reconfigurable robots. In *7th International Symposium on Distributed Autonomous Robotic Systems (DARS'04)*, 2004.

[42] S. Hackwood and J. Wang. The engineering of cellular robotic systems. In *Proc. of IEEE International Symposium on Intelligent Control*, pages 70–75, 1988.

[43] 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".

[44] J. Hopcroft, J. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects; $PSPACE-$hardness of the "warehouseman's problem". *The International Journal of Robotics Research*, 3(4):76–88, 1984.

[45] 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.

[46] R. Hui, N. Kircanski, A. Goldenberg, C. Zhou, P. Kuzan, J. Wiercienski, D. Gershon, and P. Sinha. Design of the iris facility-a modular, reconfigurable and expandable robot test bed. In *Proc. of IEEE ICRA*, volume 3, pages 155 –160, 1993.

[47] N. Inou, M. Koseki, and H. Kobayashi. Pneumatic cellular robots forming a mechanical structure. In *TITech COE/Super Mechano-System Symposium 2001*, page 67, 2001.

[48] N. Inou, K. Minami, and M. Koseki. Group robots forming a mechanical structure (development of slide motion mechanism and estimation of energy consumption of the structural formation). In *Proc. IEEE Int. Symp. on Computational Intelligence in Robotics and Automation*, pages 874–879, 2003.

[49] T. Kawauchi, M. Inaba, and T. Fukuda. A relation between resource amount and system performance of the cellular robotic system (cebot). In *Proc. of IEEE IROS*, pages 454–459, 1993.

[50] E. Klavins. Automatic synthesis of controllers for distributed assembly and formation forming. In *Proc. of IEEE ICRA*, pages 3296 –3302, 2002.

[51] E. Klavins, R. Ghrist, and D. Lipsky. Graph grammars for self-assembling robotic systems. In *Proc. of IEEE ICRA*, 2004.

[52] G. Konidaris, T. Taylor, and J. Hallam. Hydrogen: Automatically generating self-assembly code for hydron units. In *7th International Symposium on Distributed Autonomous Robotic Systems (DARS'04)*, 2004.

[53] M. Koseki, K. Minami, and N. Inou. Cellular robots forming a mechanical structure (evaluation of structural formation and hardware design of "CHOBIE II"). In *7th International Symposium on Distributed Autonomous Robotic Systems (DARS'04)*, 2004.

[54] K. Kotay. *Self-Reconfiguring Robots: Designs, Algorithms, and Applications*. PhD thesis, Dartmouth College, Computer Science Department, 2003.

[55] K. Kotay and D. Rus. Locomotion versatility through self-reconfiguration. *Robotics and Autonomous Systems*, 26:217–32, 1999.

[56] K. Kotay and D. Rus. Scalable parallel algorithm for configuration planning for self-reconfiguring robots. In *Proceedings of the Society of Photo-Optical Instrumentation Engineers*, Boston, 2000.

[57] J. Kubica, A. Casal, and T. Hogg. Complex behaviors from local rules in modular self-reconfigurable robots. In *Proc. of IEEE ICRA*, pages 360–7, 2001.

[58] H. Kurokawa, A. Kamimura, S. Murata, E. Yoshida, K. Tomita, and S. Kokaji. M-TRAN II: Metamorphosis from a four-legged walker to a caterpillar. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, volume 3, pages 2454 – 2459, 2003.

[59] D.T. Lee, C.D. Yang, and C.K. Wong. On bends and distances of paths among obstacles in two-layer interconnection model. *IEEE Transactions on Computers*, 43(6):711–724, 1994.

[60] D.T. Lee, C.D. Yang, and C.K. Wong. Rectilinear paths among rectilinear obstacles. *Discrete Applied Mathematics*, 70:185–215, 1996.

[61] W. H. Lee and A. Sanderson. Dynamic analysis and distributed control of the tetrabot modular reconfigurable robot system. *Autonomous Robots*, 10(1):67–82, 2001.

[62] H. Lipson and J.B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406:974–978, 2000.

[63] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, 1996.

[64] A. Maheshwari, J.R. Sack, and H.N. Djidjev. Link distance problems. In J.R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 519–558. Elsevier Science Publishers B.V., NorthHolland, Amsterdam, 2000.

[65] G. Majno and I. Joris. *Cells, Tissues, and Disease: Principles of General Pathology*. Blackwell Science, 1996.

[66] J.S.B. Mitchell. $L_1$ shortest paths among polygonal obstacles in the plane. *Algorithmica*, 8:55–88, 1992.

[67] J.S.B. Mitchell. Geometric shortest paths and network optimization. In J.R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 633–701. Elsevier Science Publishers B.V., NorthHolland, Amsterdam, 2000.

[68] S. Murata, H. Kurokawa, and S. Kokaji. Self-assembling machine. In *Proc. of IEEE ICRA*, pages 442–8, 1994.

[69] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji. A 3-D self-reconfigurable structure. In *Proc. of IEEE ICRA*, pages 432–9, May 1998.

[70] 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.

[71] R. Nagpal. *Programmable Self-Assembly: Constructing Global shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, Massachusetts Institute of Technology, 2001. AI Technical Report 2001-008.

[72] M. Nilsson. Why snake robots need torsion-free joints and how to build them. In *Proc. of IEEE ICRA*, pages 412–417, 1998.

[73] M. Nilsson. Heavy-duty connectors for self-reconfiguring robots. In *Proc. of IEEE ICRA*, pages 4071–4076, 2002.

[74] R. Olfati-Saber and R. Murray. Distributed cooperative control of multiple vehicle formations using structural potential functions. In *Proc. of IFAC World Congress*, 2002.

[75] E. Østergaard and H. Lund. Evolving control for modular robotic units. In *Proceedings of IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA'03)*, pages 886–892, 2003.

[76] E. Østergaard and H. Lund. Distributed cluster walk for the atron self-reconfigurable robot. In *Proceedings of the The 8th Conference on Intelligent Autonomous Systems (IAS-8)*, pages 291–298, 2004.

[77] 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.

[78] 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.

[79] C. Paredis, H.B. Brown, and P. Khosla. A rapidly deployable manipulator system. In *Proc. of IEEE ICRA*, pages 1434–1439, 1996.

[80] C.D. Piatko. *Geometric Bicriteria Optimal Path Problems*. PhD thesis, Cornell University, 1993.

[81] P. Rothemund and E. Winfree. The program-size complexity of self-assembled squares. In *STOC 2000*, 2000.

[82] D. Rus and M. Vona. Self-reconfiguration planning with unit compressible modules. In *Proc. of IEEE ICRA*, pages 2513–20, 1999.

[83] D. Rus and M. Vona. Crystalline robots: Self-reconfiguration with unit-compressible modules. *Autonomous Robots*, 10(1):107–24, 2001.

[84] K. Saitou and M. Jakiela. Subassembly generation via mechanical conformational switches. *Artificial Life*, 2:377–416, 1995.

[85] B. Salemi, W.-M. Shen, and P. Will. Hormone-controlled metamorphic robots. In *Proc. of IEEE ICRA*, 2001.

[86] G. Sandini. Cellular robotics - annotated bibliography. Technical Report TR 1/92, LIRA-Lab - DIST University of Genova, Via Opera Pia 13 - 16145 Genova, Italy, July 1992.

[87] R. Sharma and Y. Aloimonos. Coordinated motion planning: The warehousman's problem with constraints on free space. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(1):130–141, 1992.

[88] W.-M. Shen, B. Salemi, and P. Will. Hormones for self-reconfigurable robots. In *Proc. of IAS-6*, 2000.

[89] W.-M. Shen, P. Will, and A. Castano. Robot modularity for self-reconfiguration. In *SPIE Conf. on Sensor Fusion and Decentralized Control in Robotic Systems 2*, 1999.

[90] K. Stoy. Controlling self-reconfiguration using cellular automata and gradients. In *Proceedings of the The 8th Conference on Intelligent Autonomous Systems (IAS-8)*, 2004.

[91] K. Stoy and R. Nagpal. Self-reconfiguration using directed growth. In *7th International Symposium on Distributed Autonomous Robotic Systems (DARS'04)*, 2004.

[92] K. Stoy, W.-M. Shen, and P. Will. Global locomotion from local interaction in self-reconfigurable robots. In *Proc. of IAS-7*, 2002.

[93] S. Strogatz. *Sync: the emerging science of spontaneous order*. Hyperion, New York, 1st edition, 2003.

[94] J. Suh, S. Homans, and M. Yim. *Telecubes*: Mechanical design of a module for self-reconfiguring robotics. In *Proc of IEEE ICRA*, 2002.

[95] K. Tanie and H. Maekawa. Self-reconfigurable cellular robotic system. US Patent 5361186, 1993.

[96] K. Tomita, S. Murata, H. Kurokawa, E. Yoshida, and S. Kokaji. Self-assembly and self-repair method for a distributed mechanical system. *IEEE Trans. on Robotics and Automation*, 15(6):1035–45, Dec. 1999.

[97] C. Ünsal and P. Khosla. I(ces)-cubes: a modular self-reconfigurable bipartite robotic system. In *Proc. of SPIE,* volume 3839: Sensor Fusion and Decentralized Control in Robotic Systems II, pages 258–269, 1999.

[98] Cem Ünsal and Pradeep Khosla. Mechatronic design of a modular self-reconfiguring robotic system. In *Proc. of IEEE ICRA*, pages 1742–7, 2000.

[99] 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.

[100] M. Vona. A two dimensional crystalline atomic unit modular self-reconfigurable robot. Undergraduate Honors Thesis, Dartmouth College, 1999.

[101] 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.

[102] P.J. White, K. Kopanski, and H. Lipson. Stochastic self-reconfigurable cellular robotics. In *IEEE International Conference on Robotics and Automation (ICRA04)*, 2004.

[103] M. Yim. A reconfigurable modular robot with multiple modes of locomotion. In *Proc. of JSME Conf. on Advanced Mechatronics*, Tokyo, 1993.

[104] M. Yim. New locomotion gaits. In *Proc. of IEEE ICRA*, pages 2508–2514, 1994.

[105] M. Yim, J. Lamping, E. Mao, and J.G. Chase. Rhombic dodecahedron shape for self-assembling robots. SPL TechReport P9710777, Xerox Palo Alto Research Center, 1997.

[106] M. Yim, Y. Zhang, J. Lamping, and E. Mao. Distributed control for 3D shape metamorphosis. *Autonomous Robots*, 10(1):41–56, 2001.

[107] E. Yoshida, S. Kokaji, S. Murata, H. Kurokawa, and K. Tomita. Miniaturized self-reconfigurable system using shape memory alloy. In *Proc. of the Int'l Conf. on Intelligent Robots and Systems*, pages 1579–1585, 1999.

[108] E. Yoshida, S. Murata, A. Kaminura, K. Tomita, H. Kurokawa, and S. Kokaji. Motion planning of self-reconfigurable modular robot. In *Proc. of Int'l Symposium on Experimental Robotics*, 2000.

[109] E. Yoshida, S. Murata, K. Tomita, H. Kurokawa, and S. Kokaji. An experimental study on a self-repairing modular machine. *Robotics and Autonomous Systems*, 29:79–89, 1999.