2007 IEEE International Conference on
Robotics and Automation
Roma, Italy, 10-14 April 2007

Scalable Locomotion for Large Self-Reconfiguring Robots

Robert Fitch
National ICT Australia
University of New South Wales
Sydney, NSW Australia
robert.fitch@nicta.com.au

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

***Abstract*— For large self-reconfiguring robots, any algorithm that requires linear amounts of memory per module (with respect to the number of modules) or linear time for computation or communication per actuation is undesirable. While shape-forming may require linear amounts of memory, locomotion can be performed with simpler shape specifications, and therefore sublinear algorithms are possible. In this paper, we present a locomotion technique that performs both planning and actuation control in sublinear time and memory. The algorithm is inspired by reinforcement learning and uses dynamic programming to plan module paths in parallel. To ensure the physical integrity of the overall robot during motion, we have developed a novel localized cooperation scheme which may also be used with other self-reconfiguration algorithms. Our overall algorithm is able to direct locomotion over arbitrary obstacles, and the formulation of the goal used in the planning encourages dynamic stability.**

I. Introduction

Many existing algorithms for self-reconfiguring (SR) robots require linear time or memory per module (linear in the number of modules) to achieve their intended goal. However, as the size of SR systems increases, such algorithms will become impractical. For certain tasks, sublinear algorithms may not be possible — for example, to achieve arbitrary configurations, goal configurations must use an amount of memory proportional to the number of modules. However, for locomotion, sublinear algorithms are feasible. A precise goal specification is not necessary for locomotion, reducing memory requirements, and it is possible for the modules to move based on only local information, at least in certain situations [2], reducing the communication required per move. This paper describes an algorithm for SR locomotion that achieves these aims.

There are two critical aspects of any sublinear locomotion algorithm. First of all, the modules must be able to determine appropriate motions to take using only local information, where "appropriate" means both that the motions lead toward the intended location and that the system will remain connected as a result of the motion. These plans must also be robust in the face of the constantly changing topology of the robot. Secondly, the motions must be executed in parallel without global synchronization, but also without leading to deadlock or module collision.

Our solution uses simple techniques borrowed from reinforcement learning that are able to produce local coordination and use only constant memory per module to perform reliable locomotion among arbitrary obstacles. One result of executing this algorithm amongst obstacles is shown in Fig. 1. Planning is done for all modules simultaneously through distributed dynamic programming, while modules use local constant-time search and module locking to ensure physical integrity of the robot while following their paths. We assume deterministic transitions (i.e. module motion), but the method naturally extends to stochastic transitions. Continuous replanning enables the robot to follow moving goals, and the algorithm prefers locations closer to the ground to assist with the dynamics of the system. We expect this sublinear approach will enable locomotion at scales not previously possible, such as a million-module robot.

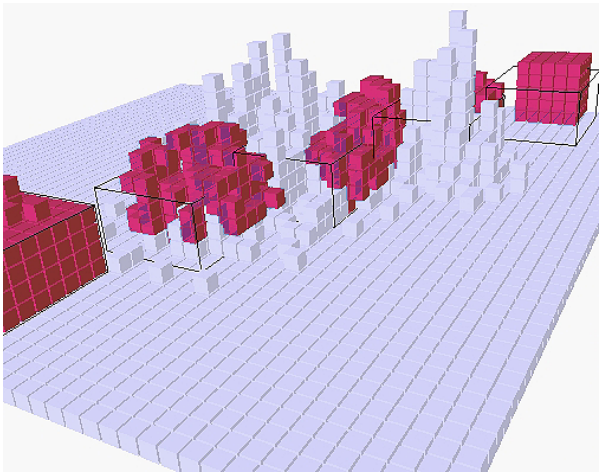


Fig. 1. An example of the locomotion algorithm operating among obstacles. The robot location and intermediate goal locations (represented by wireframe boxes) at several different points in time have been overlaid on the environment.

A. Related Work

Most algorithms for reconfiguration of lattice-based SR robots assume an exact specification of the desired shape, requiring linear amounts of memory, and many restrict actuation with explicit connectivity checks or gait modulation, requiring linear time. However, there are some exceptions. For planning, one exception is the scaffold-based work of Stoy and Nagpal [6] which uses a collection of bounding boxes to represent the goal in a way independent of the size

of the system. Our work is similar in that we currently use a simple bounding box to specify the goal of locomotion, but could easily allow the goal shape to be more detailed.

In terms of efficient actuation, the work of Yim et al. [10], specifies exact shapes, but only local control is used to avoid explicit long-range path planning and allow efficient parallel actuation. In the PacMan algorithm [3], [9], which is specific to unit-compressible systems, complete paths are planned, but in parallel. Actuation is locally controlled, though this requires complex negotiations due to the nature of the modules' connectivity. Kotay specified a parallelization technique to make reconfiguration more efficient [5], and the more recent work of Abrams and Ghrist [1] showed how to optimally parallelize paths, but these are centralized techniques that do not scale well to the systems of interest here.

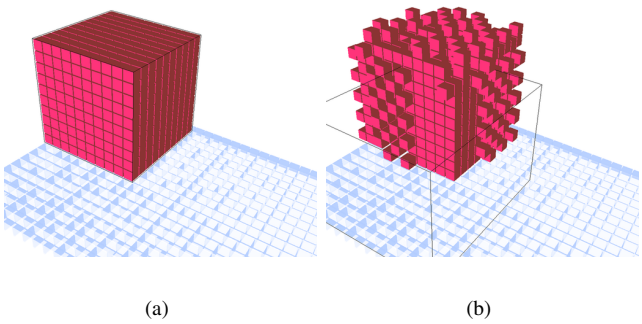
Some of our previous work uses a cellular automata framework to perform locomotion with constant information and computation requirements [2] — motion is planned and controlled using local rule sets independent of the size of the robot. However, the rule sets used tend to be complex and disconnections are prevented only by restricting the initial configuration. Here we use more explicit planning to direct a robot from any configuration to any location over obstacles, but by sharing the planning over all modules, the time required to generate the plans remains low.

Learning methods have also been used to attempt to automatically determine good rule sets to use to allow simplified controllers [8], however this has also proven difficult to achieve for complex situations such as arbitrary three-dimensional motion over obstacles. We are not attempting to produce general rules for motion, but rather using dynamic programming to produce paths on the fly for the particular geometry at hand.

II. Algorithm Formulation

Our algorithm can be divided into two components: a parallel planner which computes paths for all modules to follow toward the goal, and a parallel actuation control scheme that ensures that the motions are safely performed. There is no global synchronization assumed, however, so that various modules in the system will be performing these two functions continuously and at different times.

To address parallel path planning, we note that the search space of planning coordinated parallel paths is extremely large [1] — so instead of creating a set of global paths at once that will work in concert, we have developed a novel application of methods from reinforcement learning [7] that continually replans a large set of paths in an efficient distributed fashion as the robot moves. The efficiency is obtained by assuming that all module paths are independent, and so the plans can be generated in parallel. Since correct module motions will not be independent, the generated paths are not guaranteed to avoid conflict or be optimal in a global sense. However, the use of continuous replanning allows them to be generally efficient and will always allow the overall shape to converge. In addition, the replanning means that the goal can be arbitrarily and safely moved over the environment while the robot is moving, giving us the ability to "joystick" the robot.



(a) (b)

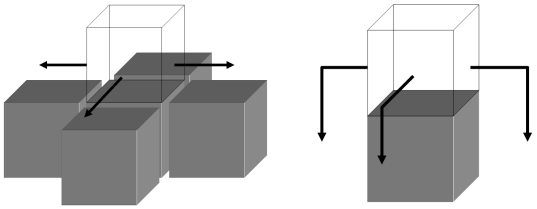
Fig. 2. Simulation of a large robot system showing the inherent parallelism of our method. (a) shows the initial cube formation and (b) after (simultaneous) initial motions of the modules.

We next consider safe path execution. In an SR robot performing locomotion, individual modules will follow some path through free space and come to rest in the goal. When using such a path, a module should only move if it does not disconnect the structure — in other words, moving modules may not be articulation points in the connectivity graph. One common simplification that can be made is to move modules one at a time, so that the mobility check can be done via simple graph operations. However, in large systems serial motion is too slow. We must exploit parallelism. To address the connectivity problem, we have developed a novel search technique that is inherently local, therefore requiring only constant time in most cases, and guarantees to preserve global connectivity. This allows a large number of modules to move simultaneously, as shown in Fig. 2.

Any algorithm for SR robots depends strongly on the hardware that will be used, since systems differ greatly in their formation and actuation. In this work, as in much of our previous work in the area, we use the *SlidingCube* abstraction. Under this abstraction, modules are cubes capable of performing translations across other modules and convex transitions around another module, as shown graphically in Fig. 3. We have previously shown how different physical systems can implement the SlidingCube [2]. We also note that the formulation of our planning system is not tied to this abstraction; any lattice-based system for which the motions can be enumerated and modules can move sufficiently on their own should be able to use a similar construction.

III. Parallel Path Planning

Planning for locomotion is similar to reconfiguration planning, in that modules move through the system to change the shape of the robot and reach a goal region at a distant location while avoiding all obstacles along the way. Successive overlapping goal configurations drive the robot through its environment. We define the *Parallel Path Planning* problem as finding a path from each mobile module to a position in a goal region, here defined by a bounding box. We formulate this problem as a *Markov decision process*



(a) (b)

Fig. 3. Action space with respect to a given face for a SlidingCube module. The wireframe box is the state (lattice position) in question, grey boxes are adjacent modules. In (a), there are four possible sliding transitions, indicated by arrows, with respect to the bottom face. Rear arrow not shown. Similar convex transition actions are shown in (b). Cases are symmetrical for remaining five faces.

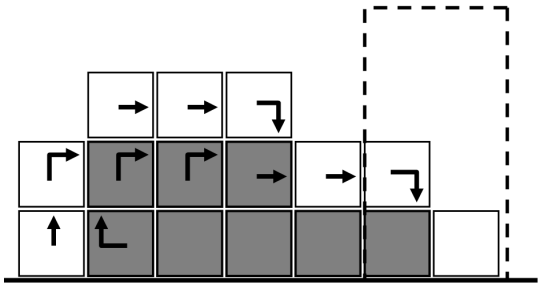


Fig. 4. Graphical representation of a policy in our MDP formulation, with the robot shown in cross-section. Each square represents a state, grey squares are occupied by modules, and the dashed-line box is the goal. Arrows indicate the optimal action to be taken by a module in that state. Straight arrows are sliding transitions, and right-angle arrows are convex transitions. Any module taking the indicated actions follows a shortest path to the goal.

(MDP) and solve it using dynamic programming (DP). This MDP will be stored in a distributed fashion; state updates are computed by adjacent module pairs. Because the DP updates (Bellman backups) execute in parallel, the policy converges in sublinear time in the size of the robot. As modules move, the underlying MDP also changes and we update the policy. Iterating this process, all modules reach the goal region. In practice, this process converges rapidly. The resulting policy yields a path from all open positions in the current configuration to a position in the goal region. It is important to note that the policy does not map modules to actions, or local neighborhood configurations to actions. Instead, the policy maps lattice positions, which are points in space, to actions. A given module may follow a path to the goal by following the optimal action associated with each lattice position it traverses, as depicted in Fig. 4.

A. MDP Formulation

To be more precise, an MDP is a $4{-}tuple < S, A, T, R >$, where S is the set of states, A is the set of actions, T is the transition function, and R is the reward function. In our MDP formulation, a state $s \in S$ is a lattice position, and an action $a \in A$ is a primitive module actuation. The transition function T maps each (s, a) pair to the resulting lattice position s'. T is deterministic and known by all modules. State-action pairs that result in collision with an obstacle or another module transition to a state with a large negative reward. Otherwise, a reward of -1 is given for each action that does not transition into the goal region. For s' in the goal region, the reward is 0 plus a small negative value determined by the height of the lattice position above the ground, decreasing from 0 towards -1. This reward function results in modules moving first into the goal region and then towards the ground as far as possible. This represents a goal ordering that avoids creating unreachable holes in the goal configuration.

The state space S is essentially that of the *gridworld* common in the reinforcement learning literature, but in three dimensions. These are world-centered coordinates – as the robot locomotes, its modules move through this state space. At the beginning of a locomotion task, a coordinate frame is attached arbitrarily to the robot. Since they know the transition function, modules can easily maintain their coordinates in this frame subsequently. Although the set of all lattice positions is infinite, the MDP only considers a small finite portion of it: lattice positions occupied by or adjacent to modules in the robot.

The action space A is determined by the primitive actions available to a module in a particular state. The total number of actions for a SlidingCube module disregarding symmetry is 48. With respect to a certain neighbor, the possible moves are sliding or convex in each of the four cardinal directions, as seen in Fig. 3. The other faces are symmetric. Therefore we have $8 \times 6 = 48$ possible actions. Many of these transition to the same destination however, and only a subset are available at any given state. In particular, sliding and convex transition moves are mutually exclusive for a given neighbor and direction. This is determined by what modules are in the local neighborhood. Therefore we may reduce the action space to four actions per neighbor, or equivalently, per face. It is also possible to make no move at all, so the null action is always valid. Our set of actions then is $\{f_i a_j \,|\, 0 < i \leq 6, 0 < j \leq 4\} \cup \{do\_nothing\}$.

The MDP is formulated as if there were a single module, or *agent*. We know that in reality we have multiple agents, but a flat representation of their collective state grows exponentially in the number of agents. To overcome this problem, we use a multi-agent abstraction where all agents share the same policy and are assumed to be independent. Of course, agents are in fact dependent on each other in avoiding collisions and preserving global connectivity. We use a separate process to deal with this issue, described in the next section. Furthermore, as modules move, the structure of the MDP changes. This corresponds to barriers changing location in gridworld terms. This is why DP updates are processed continuously in response to module movements.

B. MDP Implementation

Since the action-value function is defined over lattice positions, considering only those adjacent to module faces, it is natural to store it within the modules and propagate value updates in a parallel distributed fashion. The policy therefore

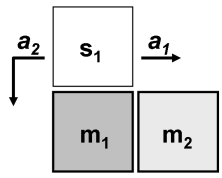
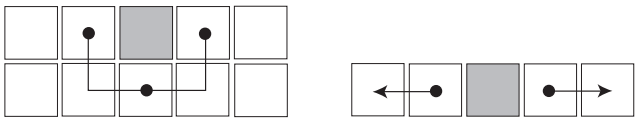


Fig. 5. Two cases for computing DP updates. Module m_1 stores action-values for state s_1. For action a_1, module m_2 must supply the value for the resulting state. However, action a_2 is simpler because the required value is already stored in m_1's memory. All other cases are symmetrical.



(a) (b)

Fig. 6. Examples of connecting cycles. The module attempting to move is shaded grey and its neighbor modules are those with dots. (a) In a dense structure, the connecting cycle joining all neighbors through other modules (shown as a dark line) is short. (b) In a chain of modules, the connecting cycle may be much longer, or not exist at all.

is represented in a distributed manner throughout the robot. This sort of distributed representation is not standard in the RL literature. Each module does not store the complete set of action-values for a particular state, but it stores a subset of action-values for a number of states. Conversely, the set of action-values for a given state is stored across multiple modules. Specifically, values for the four actions $f_1 a_j$ are stored in the module adjacent to face f_1, and so on. This can also be thought of as each module storing four values for each of its six adjacent states, or 24 in total. Values for doing nothing do not need to be stored.

DP updates can be computed by a module for the state-action pairs for which it is responsible, as explained in Fig. 5. This is possible because the values of resulting states are stored either within itself, in the case of a convex transition, or within a neighbor module in the case of a sliding transition. These values are propagated to neighbor modules via message passing. Whenever a module receives a new set of values from its neighbor, it performs DP updates for its faces as appropriate. Updates are triggered by any change in the goal region as well as by any module movement. Values propagate back from the goal, exploiting best case DP behavior, and empirically the MDP converges quite quickly. A module moves toward the goal by querying its neighbors at each step and choosing the action with maximal value if a unique one exists, or else chooses randomly from the maximal set. As indicated, this mechanism does not prevent disconnection or collisions with other moving modules. We next describe a powerful method for local motion control that solves these problems.

IV. Parallel Actuation

When moving a single module in a fixed configuration, disconnection can be avoided with simple graph analysis. The connectivity graph of the modules is a graph with a node for each module and an edge between adjacent modules. Articulation points in the connectivity graph correspond to non-mobile modules (those which are not safe to move) and can be easily detected. However, this test takes linear time, and finding a set of mobile modules that can safely move in parallel, which corresponds to finding a maximal set of nodes that can be removed simultaneously without disconnection, is significantly more difficult. Furthermore, the topology of the robot is constantly changing as modules move. It is desirable to use an approach that requires local information only. Here we describe a method that can be executed in a parallel distributed manner, prevents disconnection and collision, and allows modules to follow the specified paths.

By definition, the connectivity graph is connected if there exists a path between all nodes. If we remove a module from the connectivity graph and the graph remains connected, then the removed module is mobile. Equivalently, it can be removed if all its neighbors are connected by a path not passing through it (if it has only one neighbor, then it is immediately known to be mobile). We call these local connecting paths *connecting cycles*. A simple example of a connecting cycle (and a situation in which one might not exist) is given in Fig. 6. In sparse graphs, the connecting cycles can be long (up to length n in a ring of modules), but in dense graphs where modules are tightly packed, they are likely to be short. We can therefore identify a module as mobile by using a local search that attempts to find such a cycle. The search begins at each neighbor, using iterative deepening search implemented with message passing. Depth is initially limited to a constant so that short cycles will be found, but is allowed to increase to guarantee finding paths in cases where they exist but the local density is low.

Since actuation is not instantaneous, the modules along the search path are locked to preserve connectivity. We must also lock the immediate destination position to prevent collision with another moving module. This is implemented using message passing to simulate a test-and-set operation. After actuation, the locked modules are free to attempt their own motions. All modules execute this complete process in parallel, allowing parallel actuation. This is a conservative test in that the depth limit means we do not find *all* mobile modules, but we expect to find many mobile modules in dense configurations. Also, the depth limit gives a tight constant bound on the time it takes to do mobility checking.

Modules continuously attempt to move, guided by the value function. Motion control and path planning are executed until the robot achieves the goal configuration. Then, the bounding box is shifted either by a human operator or by some higher-level process and locomotion continues.

V. Experiments

Our goal is to implement this algorithm in a system with one million modules. We have implemented our approach in simulation using the SRSim simulator [4]. To experiment with large robots simulated on a serial computer, we used a

TABLE I

SIMULATION RESULTS FOR CUBE-SHAPED ROBOTS OVER COMPARABLE DISTANCES ON FLAT GROUND. ONE TIME STEP IS THE TIME REQUIRED FOR A SINGLE ACTUATION.

Robot size	Total number of actuations	Time steps	Average parallel actuations	Average surface modules	CPU time per motion
125 (5^3)	1012	53	18.2	46.0	4.6 ms
1000 (10^3)	23327	243	96.0	345.3	8 ms
8000 (20^3)	616493	930	663	2928	16 ms

TABLE II

SIMULATION RESULTS FOR 10 TIME STEPS OF MOTION OF VARIOUS LARGE CUBE-SHAPED ROBOTS.

Robot size	Average parallel actuations	Average surface modules	CPU time per motion
15625 (25^3)	1090	4534	1.1 ms
125000 (50^3)	4520	19254	1.4 ms
421875 (75^3)	10213	43674	1.9 ms

centralized implementation. We are also currently working on extending this to a cluster-based implementation in the near future, and incorporating higher-fidelity simulation of message passing between modules through the use of multiple threads of execution.

We instrumented the simulation to measure total elapsed time for locomotion as well as the amount of parallelism in the actuation. Results presented here are for robots of various sizes moving the same relative distance over flat ground. That is, for a cube of $n \times n \times n$ modules, we measured the time and number of moves required to move the length of $n - 1$ modules. This experiment thus explores the dependence on module size (using smaller modules to create an overall robot of the same size to go a given distance). Table I shows results for robots of increasing size n.

Table I also presents the average number of simultaneous motions made. This is compared to the average surface area of the robot as it moves, which represents an upper bound on the possible amount of parallelism. We note that for simple cubic shapes, the surface area of the robot goes up as $n^{2/3}$, so in fact we should expect parallelism relative to n to decrease as the robot gets bigger. For less compact shapes, such as flattened rectangles, parallelism should be closer to linear as the robot gets larger, since a larger fraction of the modules will be on the surface, and locomotion faster.

Since we are running the parallel algorithm on a serial computer, we also present running time normalized by the number of module motions. This is more conservative than simply dividing the total computation time by the number of modules, and we believe this is a good measure of the computation required for the algorithm. Note that these numbers are for a simulation run on a high-end workstation and that we would expect modules to use somewhat slower processors. Under this metric, the experimental data confirms the running time grows approximately with $\sqrt[3]{n}$ for cubic robots with n modules, as predicted. For larger robots, we were not able to run full-length experiments due to time and memory limitations, however we did obtain results for the initial steps of motion for cubic robots up to 75 modules on a side, presented in Table II. We believe these to be generally indicative of the overall time and parallelism of the system in a relative sense. The times per move in Table II are lower than in Table I because the motion is at its most efficient in the beginning, when a small number of modules are already in the goal region and the robot's surface area is large. We expect that a multiprocessor implementation will allow us to extend these results on very large systems.

A. Rough terrain

From a qualitative standpoint, we also wish to verify that the algorithm correctly produces paths and motion in the presence of obstacles. We chose two types of obstacles to representative of potential challenging situations. In the first, the robot is driven under a concave obstacle and then asked to move beyond it, which may be difficult for a greedy algorithm to achieve. Figure 7 shows successful locomotion in this case. The second was a comb-shaped obstacle, to test the ability to locally break apart to use multiple thin gaps in the obstacle without globally disconnecting the robot. This was also successful, as shown in Fig. 8.

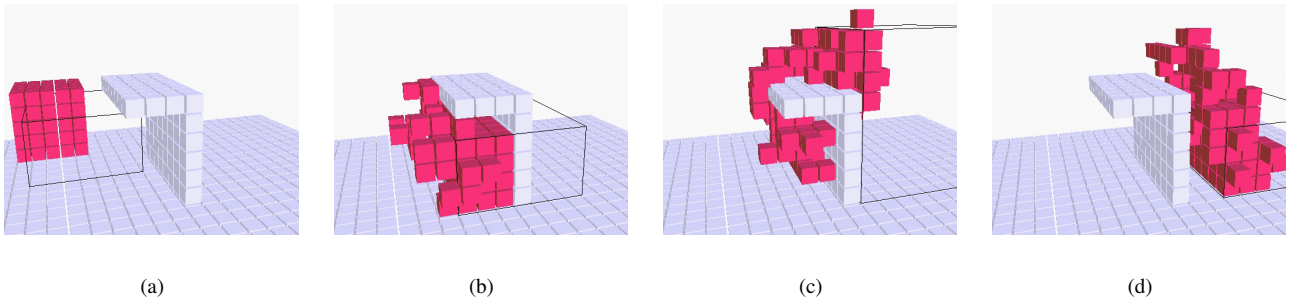
We also ran a number of experiments with robots of various sizes over randomized "bumps" such as those shown in Fig. 1, and found that as long as the bounding box was allowed to be sufficiently large when intersecting obstacles, the robot would correctly perform the locomotion task.

VI. ANALYSIS

Since our approach comes from a well-studied area, we can borrow much of that analysis to prove properties of our algorithm. For instance, the convergence properties of MDPs are generally well understood. The formulation we are using assumes a known (deterministic) state transition model, so the dynamic program will converge to the optimal result for an individual module. We do note that it does not assign modules to goals, so that many modules may initially try to reach the same goal point if that is optimal for all of them.

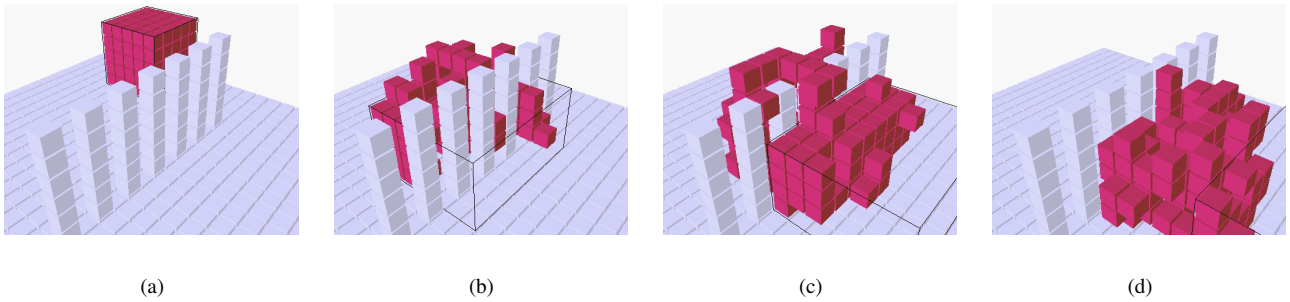
Deterministic state transition models are rare in RL applications; a main strength of RL is its ability to handle stochasticity. If transition probabilities are stochastic but known, our algorithm works as is. Otherwise, the transition model can be learned from experience using Monte Carlo or other standard RL methods. This is important since real robots always exhibit uncertainty.

In addition, the dynamics of any SR system are important, and the reward structure tends to enforce good dynamic structure, as modules will preferentially move toward goal locations lower to the ground. While this does not guarantee anything about the overall shape during motion, we can trivially show that the rewards will allow the modules to not leave any holes in the goal area. For any given location to be a hole in the structure, there must be a module directly above



(a) (b) (c) (d)

Fig. 7. Simulation of a small robot escaping from under an overhanging obstacle. (a) The robot's initial position. (b) The robot has been directed under the obstacle. (c) The goal is moved forward by the human operator such that it still overlaps the current robot location but is also taller than the obstacle; the robot starts to escape. (d) Successful traversal of the obstacle.



(a) (b) (c) (d)

Fig. 8. Simulation of a small robot through a comb-shaped obstacle with single-module width gaps. (a) The robot's initial position. (b) First encounter of the obstacle. (c) The modules use several of the gaps to move through the obstacle. (d) Successful traversal of the obstacle.

it. However, this module will immediately move into the hole, since the reward at the hole location will be better than the module would get by remaining in its current location. Thus, eventually all holes will be filled by the module(s) above it. This holds for systems over flat ground or over any obstacles that do not contain overhangs.

VII. Discussion

The development of this algorithm has pointed out interesting issues with large, dense systems relative to smaller, sparse ones. For example, the mobile module detection process is ideally suited to dense configurations, and may work reasonably for regular structures like scaffolds, but gracefully degrades in sparse structures, such as a single ring of n modules, where moving the first module requires $O(n)$ time. Similarly, the DP propagation takes time proportional to the diameter of the shape, which can be anywhere from $\sqrt[3]{n}$ for a cube or similar shape to n for a single line of modules. On the other hand, moving a large cube of modules by having each module move in turn from the back to the front while maintaining a generally cubic shape turns out to take a very large number of individual motions which are not as parallelizable as they would be in stringier shapes. We believe it is critical for any algorithm for SR robots to consider its configuration dependence, that is, under what circumstances it performs well, and in general to not only shape the robot to the task, but choose an algorithm appropriate to the particular configurations and vice versa.

Acknowledgment

National ICT Australia is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council. Special thanks go to fellow SMLKA Program members at NICTA for general discussion of the MDP formulation.

References

[1] A. Abrams and R. Ghrist. State complexes for metamorphic robot systems. *Intl. J. of Robotics Research*, 23(7-8):809–824, 2004.

[2] Z. Butler, K. Kotay, D. Rus, and K. Tomita. Generic decentralized locomotion control for lattice- based self-reconfigurable robots. *Int'l Journal of Robotics Research*, 23(9):919–38, 2004.

[3] Z. Butler and D. Rus. Distributed motion planning for modular robots with unit-compressible modules. *Int'l Journal of Robotics Research*, 22(9):699–716, 2003.

[4] R. Fitch. *Heterogeneous Self-Reconfiguring Robotics*. PhD thesis, Dartmouth College, 2004.

[5] K. Kotay. *Self-Reconfiguring Robots: Designs, Algorithms, and Applications*. PhD thesis, Dartmouth College, Computer Science Department, 2003.

[6] K. Stoy and R. Nagpal. Self-reconfiguration using directed growth. In *7th International Symposium on Distributed Autonomous Robotic Systems (DARS'04)*, 2004.

[7] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[8] P. Varshavskaya, L. Kaelbling, and D. Rus. Learning distributed control for modular robots. In *Proc. of IROS*, pages 2648–53, 2004.

[9] S. Vassilvitskii, M. Yim, and J. Suh. A complete, local and parallel reconfiguration algorithm for cube style modular robots. In *Proc. of IEEE ICRA*, pages 117–22, 2002.

[10] M. Yim, Y. Zhang, J. Lamping, and E. Mao. Distributed control for 3D shape metamorphosis. *Autonomous Robots*, 10(1):41–56, 2001.