Self-Repairing Topologies

Lauren Clement

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02114 USA
lclement@mit.edu

Abstract. Biological systems show incredible abilities for achieving robust and complex behavior from the cooperation of large numbers of identically programmed cells. Coore and Nagpal demonstrated how one can self-assemble predetermined complex and non-regular topology and geometry from identically-programmed undifferentiated agents, using a small set of biologically-inspired primitives. In this paper we show that by making the primitives for topology self-repairing, we can achieve self-repair on the level of complex patterns. Furthermore, this self-repair can be accomplished without explicitly monitoring for faults.

1 Introduction

Biological systems regularly achieve coherent, reliable and complex behavior from the cooperation of large numbers of identically-programmed agents. During development, cells with identical DNA cooperate to form incredibly complex structures, such as ourselves, in the face of large variations in cell behavior. Even after development, these systems show incredible abilities to compensate for faults and regenerate damaged or lost structures. Not only do developmental processes show resilience to variations in cell numbers, constant death and replacement of cells, and ability to fix small wounds, but many processes also have the ability to replace full grown structures. For example, limb regeneration in newts and cockroaches not only preserves geometric structure and functional constraints, but also the connectivity of nerves and sensors to pre-existing mature organs. These systems could hold important insights for new emerging fields such as reconfigurable robots, smart matter and self-assembling molecular systems, where we are trying to build global structure and function from millions of identical unreliable parts.

In this paper we investigate how one can design systems that show this property of robustness to faults, and can replace/regrow structures after larger regions are excised. This work builds on previous work by Coore and Nagpal on self-assembling topology and geometry from locally-interacting, identically-programmed agents. Coore demonstrated that a surface of identically-programmed undifferentiated agents can be made to differentiate into any topological interconnect pattern of points and lines. Nagpal presented a programming language for achieving self-assembly on an intelligent sheet composed of identically-programmed agents. In both cases the organization depends on a very small set

of primitives for local behavior. Complexity and non-regularity are achieved through repeated use of these simple local rules. The primitives are inspired by emerging ideas in developmental biology of how living cells organize in *Drosophila* and other multicellular organisms. These primitives are robust to inhomogeneous placement of agents, variations in agent numbers and random agent death during the formation process. However these systems do not adapt to faults that occur after formation.

Here we show that by making the primitive pieces self-repairing, one can automatically achieve self-repair in a much more complex pattern. Our system focuses on self-organization of topology. Coore presented a primitive for forming a line pattern between two agents designated as end-points. We present a new primitive for creating a line that adapts to faults both during and after formation. If regions of agents are "excised", then the line adapts around the fault to preserve the connectivity between the end-points. The primitive uses simple local rules in combination with time-stamps on information, which allow the system to automatically adapt over time as the distribution of functioning agents changes. The self-repair is achieved at a low computation cost and guarantees connectivity *without any explicit monitoring* for line breaks. We discuss the trade-off between between response time and frequency of information exchange, and between extent of repair and communication cost. Finally we demonstrate that a complex pattern composed of many lines, where each line is generated by the self-repairing primitive, adapts automatically around faults to preserve the topology of the original pattern.

We believe that these primitives and mechanisms are a first step towards understanding how to "engineer" adaptability into systems with myriads of homogeneous parts; or in other words how to not only engineer self-organization but self-repairing self-organizing systems. Our work addresses the constraint of maintaining topology, but eventually a structure is built from many constraints - a limb must have a particular shape and appropriate size but also must be one connected piece and contain nervous and and other systems in which topology and connectivity is key. As we build structures with complex shape and function, whether they be bridges that self-monitor or modular robotics that morph into different shapes and repair themselves, understanding how to achieve a high-level of adaptability from simple primitives will be key.

2 Motivation

This research is motivated by emerging technologies, such as MEMs[1] devices, that are making it possible to bulk-manufacture millions of tiny computing elements integrated with sensors and actuators and embed these into materials and structures. Already many novel applications that integrate computation into the environment are being envisioned and built: smart materials, sensor covered bridges, programmable assembly lines and modular self reconfiguring robots [2,

[1] Micro-electronic Mechanical Devices. Integrates mechanical sensors/actuators with silicon based integrated circuits.

4, 17, 5]. New directions in biocomputing may make it possible to harness the many sensors and actuators in cells to create programmable tissues [15].

These novel computational environments pose significant challenges. Applications will require coherent and robust behavior from the interactions between multitudes of agents and their environment. Individual agents will be bulk-manufactured and cheap, with limited resources, limited reliability and only local communication and information. It is reasonable to assume that individual agents may malfunction, perhaps losing communications ability, losing variable values, or ceasing to function entirely. It is also quite possible that large numbers of agents may be damaged or destroyed by external perturbations. In such instances it would be ideal that the system adjust to and compensate for changes in such a way that its function is preserved. However our current engineering methodologies do not easily adapt to errors or unpredictable changes, and adding such fault-tolerance is very costly.

3 Self-Repairing Topology

Although both Coore and Nagpal's methods for pattern creation are robust to failures that occur during the formation process, they are unable to compensate for faults that may occur later on. By contrast, biological systems show great abilities to repair themselves. Humans and other mammals are capable of repairing many wounds, but are unable to regrow entire limbs or digits. However, cockroaches and other insects can regenerate entire limbs, as can many amphibians [3]. Simpler organisms, such as stentor, planarian and hydra seem to demonstrate almost unlimited regenerative ability, capable of reproducing an entire organism from a small amount of tissue [9, 16]. Organisms also effectively deal with constantly changing numbers and configurations of cells as they die and are replaced.

The problem we are interested in is, given an existing spatial pattern, is there some way to have the pattern maintain or adapt itself if individual agents malfunction, or if a patch of agents is destroyed? The key elements used by Nagpal and Coore are primitives for generating lines and endpoints. If we can make primitives for generating self-repairing lines and endpoints, then we can use the same methods of composition to achieve complex patterns that self-repair. In this paper we address the problem of creating self-repairing lines that maintain connectivity in the face of faults and demonstrate that these lines can be composed to create a complex self-repairing pattern.

3.1 Agent Model

The amorphous computing model used for the development of this work consists of a two-dimensional surface covered with randomly distributed immobile agents. All agents have the *identical* program, but execute it autonomously based on local communication and internal state. Individual agents have limited memory and instead of unique identifiers they have random number generators to

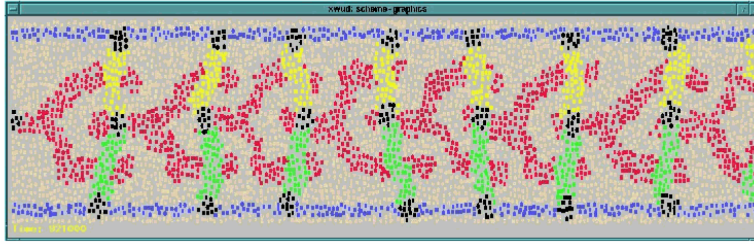

Fig. 1. Surface of identically-programmed agents differentiated into the interconnect pattern for a chain of CMOS inverters [7, 12].

break symmetry. Communication is strictly local and agents have no knowledge of global position. The agents have circular communication radii with 7 to 8 numbers neighbors on average (agents within the same communication radius are considered neighbors). Simulations were run with between 1000 and 4000 agents, using *hlsim* a Scheme based simulator [1].

3.2 Local Control for Creating a Line

Given two agents that are designated as end-points, the goal is to have a set of agents differentiate to form a line between the two end-points such that communication can travel from one end point to the other through the line of agents. For example, the pattern of an inverter chain in Figure 1 is composed of many lines that represent the flow of current. Such a graph could equally represent the movement of material on an assembly line or the flow of information between devices placed on a smart table [4].

Coore developed a primitive for creating lines based on following gradients. Given two endpoints, one endpoint creates a gradient, and the second one starts a process of choosing a successor to be in the line, based on the gradient value. Gradients are analogous to chemical gradients secreted by biological cells; the concentration provides an estimate of distance from the source of the chemical. Gradients are believed to play an important role in providing positional information in morphogenesis. For instance, in the *Drosophila*, two different proteins are emitted from opposite ends of the embryo and are used by cells to determine whether they lie in the head, thorax or abdominal regions[16].

An agent creates a gradient by sending a message to its local neighborhood with the gradient name and positive integer value. The neighboring agents forward the message to their neighbors with the value decremented by one and so on, until the gradient value reaches zero. Each agent stores the maximum value it has heard for a particular gradient name, thus the gradient value decreases away from the source. In a strictly topological sense this is equivalent to generating a breath-first-search tree and the gradient value is the shortest path to the course [10]. However because agents communicate with only neighboring agents

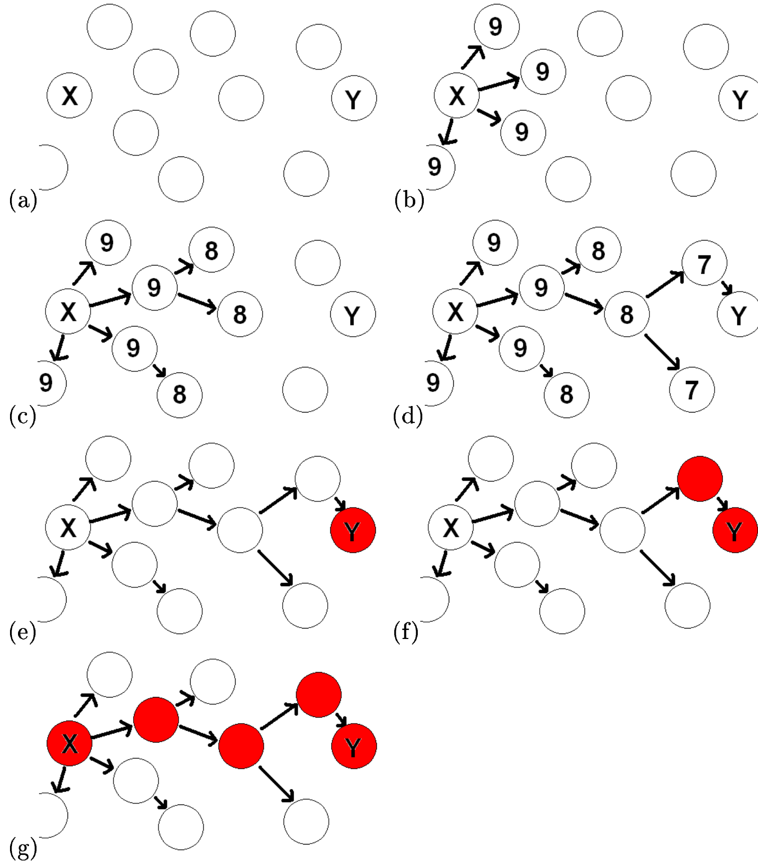

Fig. 2. Growing a Line. Figures (a)-(d) show the gradient propagating. Figures (e)-(g) show how each agent locally picks a successor with a smaller gradient value.

within a small radius, the gradient also provides a rough estimate of distance and direction from the source.

Our method for growing a line is similar to that employed by Daniel Coore, but uses a novel messaging approach. Line formation and gradients are implemented through a very simple messaging process. Agents broadcast information to local neighbors on a regular basis. These messages convey the following information.

1. *agent-id:* Every agent creates an identifier for itself by choosing a random number.
2. *gradient source and gradient value:* The gradient source is the id of an agent which initializes a gradient. The gradient value is an integer representing how many communications hops a given agent is from a given source.

3. *state:* State is a single bit used to identify the role of the agent. If the bit is 1, the agent is part of a given line; if the bit is 0 then the agent is not part of that line.
4. *successor-id:* The agent-id of a neighbor agent that is the best choice to follow in the line.

3.3 Growing the Line

Figure 2 shows how the line is formed. We begin with two endpoints, X and Y. We assume that these two agents have been chosen to be endpoints through some other process. Agents are represented by circles. Note again that agents can only communicate with other agents within a small communications radius.

X originates a gradient with a starting value of ten. The gradient propagates from X to Y. When an agent receives a message from a neighbor with a gradient value higher than it has heard before, it remembers the agent-id from that message along with the source of the gradient. If it then hears an even higher gradient value from the same source, it forgets this id and remembers the agent-id of the new highest neighbor. This id value is the successor-id and its current value is broadcast in every message. Arrows point from successor to predecessor.

When the gradient has been propagated, Y sets its state to be 1, indicating that it is now part of the line. Agents that are part of the line are in red. When the neighbor that is being identified as Y's successor receives a message from Y it recognizes that the successor-id in the message is its own agent-id. It checks the state value of the message, and since it is 1, indicating Y is part of a line, the receiving agent will set its own value to be one.

This process will continue as the successor of this agent follows the same procedure, until endpoint X is reached. The line backtracks along the path that the gradient took to go from X to Y. Note that only one such path is possible, since any agent will only ever remember and broadcast one successor-id.

The procedure above will create the backbone of a line which is only one agent thick. In the simulation images lines are approximately a communication radius thick because all the neighbors of the backbone agents are programmed to change their color.

3.4 Self-Repair using Active Gradients

The approach to line formation described above can be extended to be self-repairing in a very elegant way. Since the line is maintained based on gradient information, making a gradient which *adapts* to changing conditions will make the line itself adaptable. We will refer to this as an active gradient. Creating an active gradient necessitates that agents periodically broadcast state and gradient information, and involves remembering a time-stamp created using an agent's internal clock. Every time an agent hears from its successor it updates this time-stamp. Agents need only periodically check that this time-stamp has not gotten too old, which would mean that its corresponding information about state and gradient values is not recent enough to be considered reliable. Specifically,

this will occur when a successor malfunctions, or stops communicating for some reason. If this happens, the agent will decrement its gradient value by one until it has established an up-to date gradient value.

Finding this up-to-date value will also cause an agent to choose a new successor, since the successor-id is defined as the id of the neighbor who caused the agent to set its current gradient value. Unlike methods used by Coore and Nagpal where a line is formed once and considered done, here the line is "maintained" based on the gradient values. The time-dependency of the active gradient means that it will always reflect the current layout of the system. And as a result the line structure is never dependent on the functionality of agents which have stopped communicating, but will always find other neighbors on which to rely.

Figure 3 depicts how a line repairs when a region of agents is deactivated. Agents record a time-stamp when they hear from their successors. If the successor stops transmitting, as shown in the figure by the darkened region, this time-stamp will not be updated. When a certain threshold amount of time has passed, the agent will begin to decrement its gradient value. The value of this threshold should be directly related to the frequency of communication. The agent continues to decrement this value until it hears that one of its neighbors has a higher gradient value. In this way the gradient is re-established and adjusts to the new communications layout. These new gradient values will be the same as if the gradient had originally been propagated on this new layout. Agents in the line also record a time-stamp when they receive a signal to become part of the line. When this value times out, the agent de-differentiates from the line. As the gradient originating from endpoint X adjusts, the line adjusts until it finds the best state; it re-routes using the new shortest path of functioning agents, around the fault. Since endpoint Y only ever names one of its neighbors as its successor, there is gauranteed to be only one line present when the system settles. Also note that if later on the region of agents recovers, the gradient adjusts again, and thus the line will again follow the shortest communication path. Figure 4 shows a simulation of a single line repairing.

Following is some pseudocode to further clarify the algorithm.

```
The receive-loop and transmit-loop run simultaneously on each agent.
All variable values are local to an agent.

(define agent-id (random 1000000))

;; default values for variables

(define state 0)

(define gradient
  (if (endpoint1?) start-potential
      0))
```

```
(define predecessor -1)

(define successor -1)


;;time-stamps

(define predecessor-time 0)

(define successor-time 0)


;; Agents will constantly transmit and recieve
;; information (agent-id, gradient, state, id of successor)

(define (receive-loop)
  (select
   (primitive-message-event
    => (lambda (message)
 (event.clear! primitive-message-event)
 (cond
  ;;message from current successor
  ;;if the successor's gradient value has changed,
  ;;the agent will change it's own gradient
  ((= (message-id message) successor-id)
   (if (not (= (message-gradient message) (+ gradient 1)))
       (begin (set! (gradient (-1+ (message-gradient message))))
      (broadcst (make-message agent-id gradient state successor))))
           (set! successor-time (clock)))


  ;;message from new successor
  ((> (message-gradient message) (+ gradient 1))
   (begin (set! successor (message-id message))
  (set! successor-time (clock))
  (set! gradient (-1+ (message-gradient message)))
  (broadcast (make-message agent-id gradient state successor))))


  ;;message from predecessor to it's successor (which is not yet part of the line)
  ;;the agent sets it's own state to 1 and thus becomes part of the line itself
  ((and (= agent-id (message-successor message)) (= 1 (message-state message)))
   (begin (set! state 1)
  (set! predecessor (message-id message))
  (set! predecessor-time (clock))
```

```
  (broadcast (make-message agent-id gradient state successor))))

  ;;message from predecessor to agent which IS part of the line
  ;;the agent either de-diffferentiates from the line, or updates a time-stamp
  ((= (message-id message) predecessor)
   (begin
     (if (and (= state 1) (de-differentiate? message agent-id))
     ;;de-diffentiate? returns false if the agent is named as the successor of the
     ;;message and the message has a state value of 1, (that is if the agent is
     ;;still being signaled to become part of the line) and true otherwise.
 (begin (set! state 0)
(set! predecessor -1)
(set! predecessor-time (clock))
(broadcast (make-message agent-id gradient state successor))

 (set! predecessor-time (clock))))))
 (receive-loop)))))



(define (transmit-loop)
  (begin
    (wait communication-period)
;;the value of 'communication-period' determines how often agents broadcast

    (if (> (- (clock) successor-time) threshold-time)
;;The value of threshold-time specifies how often a gradient will renew
;;(how old information can be before it is considered out of date).
;;If the successor-time times out, the agent decrements its gradient
      (set! gradient (- gradient 1)))

    (if (and (> (- (clock) predecessor-time) threshold-time)
     (part-of-the-line?))
;;If the predecessor-time times out, and the agent is part of the line
;;the agent de-differentiates.
(begin (set! state 0)
       (set! predecessor -1)
       (set! predecessor-time (clock))))

    (broadcast (make-message agent-id gradient state successor))
    (transmit-loop)))
```

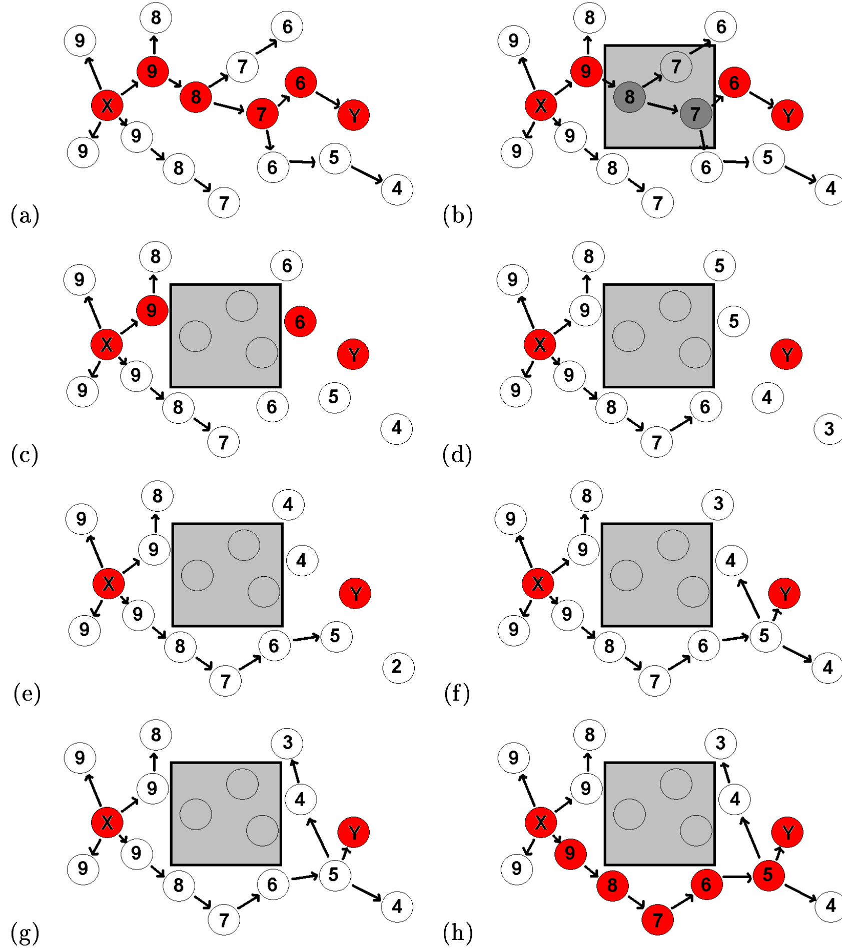

Fig. 3. A line self-repairing. Figures (a)-(c) show an area (shaded box) of agents being deactivated. These agents stop communicating, causing the time-stamps of the agents downstream of them to time-out. When the time-out occurs, agents begin to decrement their gradient values. This continues in (d)-(g) until the agent hears from a neighbor with a gradient value which is higher than its own current value. The signal to remain part of the line also times out and agents who were part of the old line de-differentiate. The new line follows the adapting gradient as a new communication path is established between the endpoints (h).

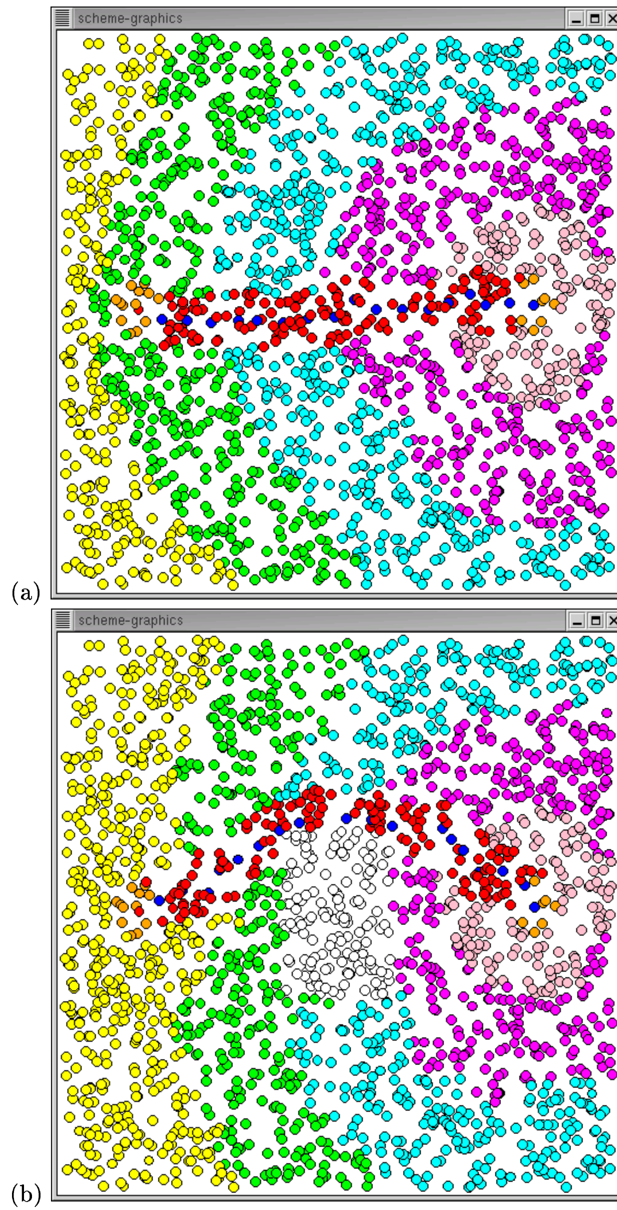

Fig. 4. Simulation of a single line repairing after being disconnected. Figure (a) shows a line formed between two end-points using an active gradient. In figure (b) a region of agents is deactivated (shown in gray). The active gradient adjusts its value and the line adjusts accordingly to go around the region of dead agents. Eventually the gradient values stabilize and the line forms along the shortest path between the end point. Note that if the agents get reactivated, the line would automatically straighten to it original position.

3.5 Multiple Self-Repairing Lines

The methods described above for a single line can be applied to more complex topologies. In order to have several self-repairing lines, each line (pair of end-points) must have an associated active gradient which it uses both for formation and self-repair. Agents must have the ability to remember gradient values, and corresponding successor-ids, from multiple sources. In the next section we discuss ways of making this efficient in spite of the large number of gradients sources. Because each connection is based on active gradients, it will be robust to breaks. Figure 5 shows and example of a multi-line topology that resembles the neural system of a newt.

3.6 Results

In this section we present an analysis of the robustness and efficiency of this algorithm for self-repair.

There is always guaranteed to be a single line between two end-points. Since which agents are part of the line is dictated by gradient values, the line will adjust along with the gradient. As a better path along the gradient develops, the agents in this path will be signaled to become part of the line. Since every agent can only have one successor, this means that old members of the line will no longer be receiving such a signal. Because they periodically check time-stamp values, these agents know that after a certain period having not received this signal they should de-differentiate (that is, that they should no longer be part of the line). This ensures that there will only ever be one line, as each agent in the backbone will always have exactly one successor and will only remain a part of the line if it is being named as a successor by another agent in the line. As long as it is possible for a message to travel between endpoints (for a gradient to propagate from one to the other), connectivity will be re-established.

The response time, or time taken for connectivity to be re-established, is determined by how long a time-stamped data is considered valid, which is directly correlated to how often agents broadcast state and gradient information. If agents are broadcasting frequently, then the system will be able to adjust quickly because information about changes in the system will propagate faster. Therefore one can tune the desired response time by picking the communication frequency (and hence cost) that is acceptable.

In a multiple line system, an agent would need to store one state bit and one gradient value/timestamp per line and a naive approach would be to broadcast information on all lines at every round. In this case both the communication cost and state stored by an agent would be proportional to the complexity of the entire pattern. However there are many ways in which to make this efficient. First, an agent can transmit only the gradient values that have changed, so in the steady state all that is transmitted is a small message that serves as a "I'm alive" message to the neighbors. Heavy communication occurs only during initial line growing and when the line needs to repair. The second method is to limit the span of the active gradient for each line. Since gradient values count down,

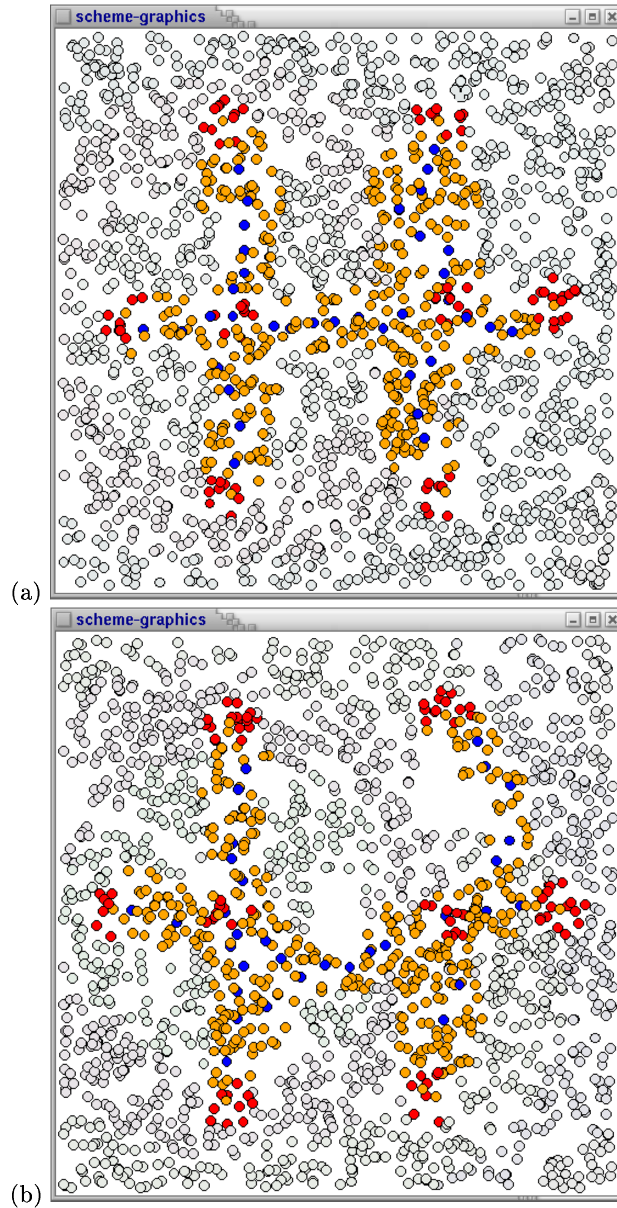

Fig. 5. Simulation of a complex pattern composed of many lines reapiring after regions of agents are excised. (a) The topology is a caricature of a neural system connecting various sensors. The end-points are shown in red. (b) Two regions of agents are removed. The lines adjust to maintain connectivity between the end-points.

the initial value of the gradient determines the maximum distance it can reach. This value automatically limits the extent of self-repair because if the gradient cannot reach the other end-point then a line cannot be formed. This approach not only decreases the amount of communication but also the amount of state an agent must store, since an agent will only hear and store gradient values for end-points within a certain distance. This can be beneficial in other ways too, since it may not be desirable for repair to occur if the length of the new line is too long.

One of the interesting aspects of this algorithm is that the self-repairing ability of this method is almost a side effect of the way in which the line is grown. Breaks are never explicitly detected; instead line are dynamically maintained as dictated by the current gradient. As a result the system automatically adjusts to multiple breaks in the line and adjusts if regions are replaced or recover, without any modification to the algorithm. It is similar to having the line always in a state of equilibrium; if perturbed it settles into a new state. We have also experimented with algorithms that explicitly detect breaks and then create new gradient to repair around the fault, however these algorithms all require extra code to handle multiple breaks and are unable to take advantage of recovering regions.

3.7 Future Work

We have demonstrated that self-repair of topologies is possible and can be accomplished in a straightforward way while adhering to the ideas of amorphous computing. We first showed that connectivity between two fixed agents can be accomplished using time-stamps and active gradients. These methods can be applied to establish self-repair in systems with many endpoints and complex topologies.

In the near future, we hope to automatically incorporate self-repair into pattern formation. Achieving self-repair in a multiple line pattern requires that large portions of similar code be generated for each gradient in the system. For this reason it would be highly desireable to implement a compiler which would generate this code automatically. We hope to create a simple language in which a user provides a list of endpoints and what they should be connected to, and a robust structure with the specified connections is produced. Connectivity in such a structure will always be preserved. We are currently working on creating a compiler which translates statements of the form:

```
(make-self-repairing-line (endpoint1 endpoint2)
                          (endpoint4 endpoint2)
                          (endpoint3 endpoint1))
```

where each of the pairs given as inputs represents a connection between endpoints, and produces code to create a self-repairing structure with the specified connections. The translation thus far is somewhat inefficient in that it will produce a new gradient for each of the listed pairs. However, this can be fixed in the

future by writing a function which takes such an input and determines which endpoints should produce gradients in order to minimize the number of gradients needed. We have had some success with the compiler so far, what remains to be done is just a matter of finishing the translation through string manipulation, and checking for errors. Given the progress thus far, we do not anticipate any major difficulties in completing the compiler. The important thing that the work on the compiler is showing, is that it will indeed be possible to automatically create amorphous patterns which will maintain topological constraints simply by specifying what those constraints are. This means that the user does not explicily have to deal with how the system reacts to faults because self-repair is being handled on a lower level.

Up to this point we have not dealt with the possibility of losing an endpoint. Since line maintenance is dependent on the existence of these agents, the loss of an endpoint results in the loss of any line which is defined by that endpoint. We plan to explore methods for making endpoints self-repairing, and thus make our structures robust to any faults. Preserving connectivity between endpoints, as described in this paper, is already possible. Combining this with a mechanism which would replace lost endpoints would constitute a comprehensive strategy for self-repair.

4 Acknowledgments

This research is supported by a National Science Foundation grant on Quantum and Biologically Inspired Computing (QuBIC) from the Division of Experimental and Integrative Activities, contract EIA-0130391.

References

1. H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. Amorphous computing. *Communications of the ACM*, 43(5), May 2000.
2. A. Berlin. *Towards Intelligent Structures: Active Control of Buckling.* PhD thesis, MIT, Dept of Electrical Eng. and Computer Science, May 1994.
3. H. Bohn. Extent and properties of the regeneration field in the larval legs of cockroaches (leucophaea maderae).i. extirpation experiments. *J. Embryol. exp. Morph*, 31:557–572, 1974.
4. W. Butera and V. Bove. Literally embedded processors. In *Proc. of SPIE Media Processors*, 2001.
5. Z. Butler, S. Byrnes, and D. Rus. Distributed motion planning for modular robots with unit-compressible modules. *Proceedings of the Intl Conf. on Intelligent Robots and Systems*, 2001.
6. Z. Butler, K. Kotay, D. Rus, and K. Tomita. Generic decentralized control for a class of self-reconfigurable robots. *Proceedings of the IEEE Intl Conf on Robotics and Automation (ICRA)*, 2002.
7. D. Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer.* PhD thesis, MIT, Dept of Electrical Eng. and Computer Science, Feb. 1999.

8. S. Forrest and M. Mitchell. What makes a problem hard for a genetic algorithm? *Machine Learning*, 13:285–319, 1993.
9. R. Goss. *Principles of Regeneration.* Brown University, Academic Press Inc., 1969.
10. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Wonderland, 1996.
11. M. Mataric. Issues and approaches in the design of collective autonomous agents. *Robotics and Autonomous Systems*, 16((2-4)):321–331, Dec. 1995.
12. R. Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics.* PhD thesis, MIT, Dept of Electrical Engineering and Computer Science, June 2001.
13. R. Nagpal. Programmable self-assembly using biologically-inspired multiagent control. In *Autonomous Agents and Multiagent Systems (AAMAS)*, July 2002.
14. Pamecha, Ebert-Uphoff, and Chirikjian. Useful metrics for modular robot planning. *IEEE Trans. on Robotics and Automation*, 13(4), Aug. 1997.
15. R. Weiss, G. Homsy, and T. Knight. Toward in vivo digital circuits. In *Dimacs Workshop on Evolution as Computation*, Jan. 1999.
16. L. Wolpert. *Principles of Development.* Oxford University Press, U.K., 1998.
17. M. Yim, J. Lamping, E. Mao, and J. G. Chase. Rhombic dodecahedron shape for self-assembling robots. Spl techreport p9710777, Xerox PARC, 2000.