Evolutionary, Developmental Neural Networks for Robust Robotic Control

by

Bryan Adams

S.B. Electrical Engineering and Computer Science (1999),
M.Eng. Electrical Engineering and Computer Science (2000),
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2006

Author . . . .
Department of Electrical Engineering and Computer Science
May 24, 2006

Certified by . . . .
Rodney A. Brooks
Panasonic Professor of Robotics
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Evolutionary, Developmental Neural Networks for Robust Robotic Control

by

Bryan Adams

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The use of artificial evolution to synthesize controllers for physical robots is still in its infancy. Most applications are on very simple robots in artificial environments, and even these examples struggle to span the "reality gap," a name given to the difference between the performance of a simulated robot and the performance of a real robot using the same evolved controller.

This dissertation describes three methods for improving the use of artificial evolution as a tool for generating controllers for physical robots. First, the evolutionary process must incorporate testing on the physical robot. Second, repeated structure on the robot should be exploited. Finally, prior knowledge about the robot and task should be meaningfully incorporated. The impact of these three methods, both in simulation and on physical robots, is demonstrated, quantified, and compared to hand-designed controllers.

Thesis Supervisor: Rodney A. Brooks
Title: Panasonic Professor of Robotics

Acknowledgments

This thesis is dedicated to my parents, Mark and Patty Adams.

I have been blessed with the richest learning environment imaginable throughout my entire life, and these acknowledgements are a look back through all the people who have helped me along the way. I would not be where I am without all of their valuable contributions.

My thesis committee has been tremendously helpful. Bruce Tidor provided thoughtful feedback on this work, both after a qualifying exam and during formation of this final document. Una-May O'Reilly has been an invaluable resource over the last five years, and I appreciate her thoughtful guidance. She has also been a careful editor of this document.

Rodney Brooks has been a supervisor and mentor without parallel. Through many changes in personnel and focus, his lab has remained a place that celebrates creative thinking about robots, and I will always remember my time here fondly. Our weekly conversations are the aspect of graduate school I will miss most.

My research sponsors have been generous in allowing me to pursue ideas that didn't appear to have any near-term applicability, and I hope they find some value in this document. Rini Sherony, Danil Prokhorov, and Steve Kalik have all provided feedback and support for which I am tremendously grateful.

Julius Popp's *micro.adam* and *micro.eva* robots were inspiring and critical to the success of this work. Julius is both an inspired artist and a patient individual, and I sincerely appreciate both qualities. I hope he continues to build and distribute robots that have nothing to do with navigation. Jan Karabasz also provided extremely helpful information regarding the computational interface.

My use of the iRobot SCI was critical to my success, and I was supported by a host of helpful folks at iRobot. Clara Vu provided early support getting everything working, Tony Campbell provided some highly instructive software, Jennifer Smith generously reprogrammed both of my Roombas when they needed it. I also wish to thank Craig Hughes for very useful support on the gumstix platform.

Ken Stanley and Risto Miikkulainen's NEAT was fundamental to the work in this dissertation. Ken was particularly patient and helpful via email as I re-implemented the system for my own uses. I appreciate them letting me stand on their shoulders for my dissertation.

The humanoids group has been filled with brilliant and fun colleagues during my entire stay at MIT, and I hope that we cross paths many times in the future. Many visitors have enriched the environment over the years: Manuela Veloso, Georgio Metta, Martin Martin, Kathleen Richardson, and Lorenzo Natale all left strong positive impressions. Former labmates James McLurkin, Matt Williamson, Cynthia Brezeal, Juan Velasquez, and Artur Arsenio all contributed thoughts and ideas that stimulated and motivated me. Brian Scassellati was a fun officemate, and remains a mentor and friend. Paul Fitzpatrick was always good for a brilliant comment and sparkling conversation. Paulina Varchavskaia provoked interesting thought about the behavior-based work in this document.

My current group of labmates deserve particular mention: Myunghee Kim, Jessica

Banks, Lijin Aryananda, Aaron Edsinger, Eduardo Torres-Jara, and Varun Aggarwal have all made the last few years both enjoyable and educational. I will miss our group meetings, which surprises me. Jeff Weber made several substantive contributions to my project over its lifetime, and I am very grateful to have had access to his particular brand of genius.

Two labmates deserve special recognition. Matt "Maddog" Marjanović gave me my entry into the Cog Shop in 1997 and then whipped me into intellectual shape over the next four years. I will always appreciate the time and effort he spent mentoring me. Charles Clark Kemp wrote the Roomba tracking software that allowed me to quantify the Roomba's cleaning coverage, and provided support, suggestion, and encouragement on matters large and small over the last year. I will really miss our conversations.

Patrick Winston's tips on how to speak were invaluable during my defense, and our conversations about FSILGs were stimulating and enjoyable. Paul Gray has always been available for advice since 1996 and has been a role model for both scholarship and character. I appreciate both of their contributions during my stay at the Institute.

I also offer my sincere thanks to Rod's assistants over the years; they have helped tremendously with the day-to-day operation of the group and are all wonderful people: Annika Pfluger, Sally Persing, Christine Scott, Theresa Langston (now Tomič), Ann McNamara, who helped navigate Julius' robots into the country, and Ann Whittaker, who both helped to keep Julius' robots in the country and helped me secure a location for my wedding reception.

I have a group of close friends who have made life outside the lab interesting and fun: Matt Chun, Jeb Keiper, Dave Ting, Adam Croswell, Andy Oury, Ben Moeller, Andrew Jamieson, and Jeff Munro have been the brothers that I never quite had. It's been an honor and a pleasure to be brothers and friends.

My entire family has been instrumental to my success. My soon-to-be in-laws, James Fessenden, Sheila Ranganath, Lid Tadesse, and Shashi Ranganath have provided an oasis of fun and sanity during an incredibly stressful time, and I love them all dearly. My younger sister Emily inspired my master's thesis and brings a smile with her phone calls and IMs (but she is still forbidden from touching my computer). My youngest sister Laura has provided me with a window into the world of art while never failing to make me laugh. I love you both.

My parents, to whom this thesis is dedicated, are responsible for who I am today. My dad has spent his entire professional life representing injured workers, and his dedication and character have both shaped my thinking and filled me with pride. I appreciate his coaching little league, putting me through college, and being the most careful proofreader of this document more than I could ever express. My mother could have written her own doctoral dissertation, but instead focused on raising me and my sisters. She has always challenged me, guided me, kept my head on straight and made sure that my heart was in the right place. I love you guys and can't wait to buy you the house on golden pond that you richly deserve.

Finally, to my very-soon-to-be-wife Sonia: you make all the hard work worth it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Designing a controller for a physical robot in a complex environment is a difficult task. A human system designer, who typically has both robot experience and domain knowledge, uses some combination of ad hoc strategy, heuristic decision-making, and plain trial-and-error during the design process. The designer is only human and will make mistakes due to incomplete understanding of the robot, a failure to anticipate situations or states, or even simple implementation oversights. And, like any other human design, achieving even an imperfect result requires a significant investment in intellectual, temporal, and financial resources.

An alternative approach is to automate the synthesis of robot controllers using artificial evolution. Instead of asking a human to understand a robot, an environment, and a task and simply hoping he or she can invent a suitable controller, evolutionary algorithms mechanize the incorporation of these elements. Using some combination of a simulated and real robot, an evolutionary algorithm will evaluate many controllers, selecting the highest performers and re-evaluating new controllers based on past success. While the process can be slow and does not offer a guarantee of optimality, automated synthesis offers the hope that the messy and difficult task of programming robots, like many other messy or difficult tasks, can be turned over to machines.

1.1 Neural networks and evolutionary algorithms

This dissertation explores this idea in the context of evolved neural networks. While neural networks are more commonly used for classification tasks, neural robot controllers are a sufficiently powerful subset of all possible controllers to generate interesting behavior (and, when combined with other control strategies, can be powerful enough to compete with hand-designed algorithms, as seen in section 4). A neural network is a labeled digraph with a single, real-valued state variable at each vertex (figure 1-1). Data is input to the neural network by encoding sensor data into some real-valued range, typically [-1.0, +1.0] and setting the state variable for a set of designated "input" nodes to the sensor values. This sensor data is then passed through the nodes of network, scaled by the labels on the edges (frequently referred to as "weights"). Each node sums all of its inputs and then computes a non-linear function of the inputs (the robotic networks in this dissertation use arctangent as a transfer function). The output nodes of the neural network will provide values in the range of the transfer function, and these values are then decoded to create signals for the actuators on the robot.

Evolutionary algorithms are, in essence, a biologically inspired method of search. An evolutionary algorithm consists of an environment, a desired task within that environment, an agent capable of performing that task, and an encoding of the agent. In real evolution, the environment is the physical world, the task is self-reproduction, all living things are agents, and they are encoded with DNA. Evolutionary algorithms are similar, but typically substitute an arbitrary task for reproduction, and the performance of the arbitrary task is directly tied to an agent's level of reproduction. The evolutionary algorithm (see figure 1-2) begins with some arbitrary group of agent encodings. The agents are created, released in the environment (typically one at a time), and assigned a score (a "fitness") based on their completion of the desired task. Once each agent has been evaluated, a new population is created by choosing some combination of the highest-performing agents and making imperfect copies of their encodings. The imperfections, or "mutations," hold a slight chance of improving the

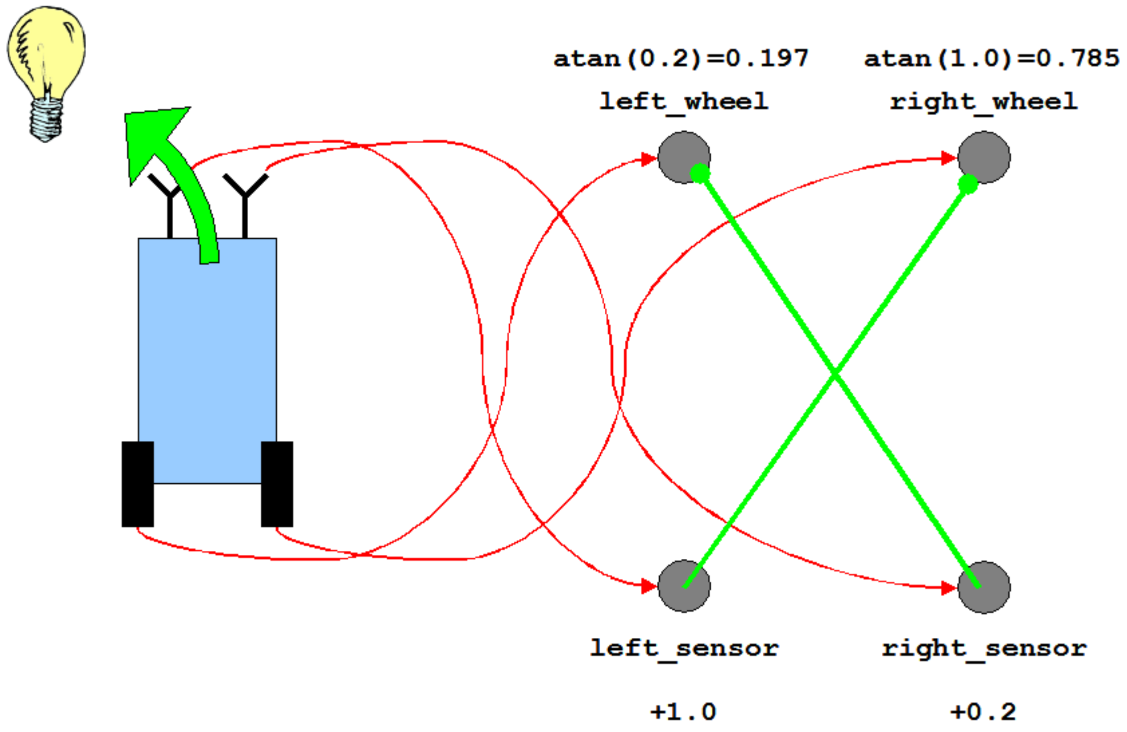


Figure 1-1: An illustration of a neural network controller on a Braitenberg-style vehicle (see [7] for details). Inputs, in this case, two light sensors, are translated into real valued numbers and input to the network on the "input" layer (bottom). The edges between the nodes (green lines, middle) transmit the values throughout the network. Hidden nodes (not shown) and output nodes (top) compute a non-linear function (in this case, arctangent) of the sum of their inputs. The values of the output nodes are translated into actuator values, in this case, wheel speeds. The network shown, with two crossed, equal, positive-weighted links, would cause the vehicle to turn and accelerate towards a light source.

performance of the agent, but these slight chances are aggregated and propagated through the populations over the course of the evolution. Evolutionary algorithms are stopped by the system designer, and typically the highest performing individual is declared the "winner" and represents the system output.

For this dissertation, the agents are evolved neural networks, encoded by simply listing the nodes and links (although a slightly more complex system will be introduced in chapter 3). The environment can either be a simulated world with a simulated robot or the physical world with a real robot. The task is typically some performance of the robot in its world, and the neural network's fitness is based on its ability to generate behaviors in the robot that satisfy the task (the systems used for evaluation are described in more detail in 2.2 and 2.3). The process for determining how to both choose the networks for reproduction and for how to vary them for subsequent evaluation are described in section 2.4.

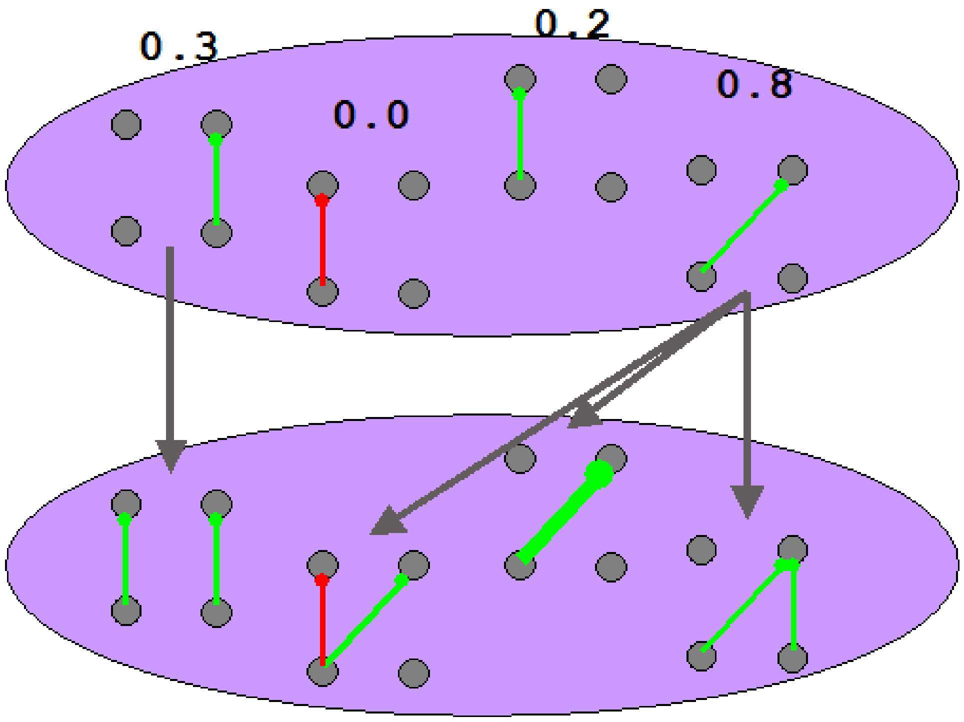


Figure 1-2: An illustration of an evolutionary algorithm for neural network. Each network in a population (purple oval) is evaluated, and offspring are rewarded on the basis of the score of the evaluation. In the above example, the most fit individual (far right) is given three offspring whereas the second most fit individual (far left) is given only one. Note that the offspring are not perfect copies, but are instead slightly changed, or *mutated*.

1.2 Overview of related work

Research combining biological and robotic systems has a nearly twenty-year history [3] [9], and evolved neural networks for robots has played a significant part. The concept has been proven on a variety of robots, from a simple six-legged walking robot [20], to small mobile robots [36], to the Sony Aibo [26], to a wide variety of simulated robots [53] [73] [4]. These neuro-evolutionary approaches generally fall into two camps: those with an evolutionary focus and those with a neural focus.

Projects with an evolutionary focus aim to improve the effectiveness of evolved controllers by improving the evolutionary mechanism. A great deal of research has gone into how the use of simulation can be more effectively included in the evolutionary process, either by carefully modeling noise [36] [60], by minimizing the simulation itself [34] [35], or by limiting the robotic domain [18]. Other interesting work has focused on providing access to a corpus of real data during the evolutionary process [44] [63]. And several creative robotic systems have been employed to make testing easier and more effective [25] [37]. (This work is reviewed in more detail in chapter 5.) These approaches are generally focused on how the mechanisms in both the evolutionary algorithm and the robotic system impact the ability to evolve a controller.

Alternatively, projects with a neural focus attempt to improve the performance of evolutionary controllers by improving the neural mechanisms available to evolution. Some have included a developmental step that allows a network to adjust to the specific task at hand [17] [2]. Related to this is the idea that neural networks should constantly adjust by using multiple signaling mechanisms [59]. Another approach is to evolve the structure of the network from a single cell, allowing evolution to form the entire structure of the solution, including the use of inputs and outputs [71] [21]. Others hope to build modularity into the system by providing mechanisms for selection among different sub-networks [11]. (This work is reviewed in more detail in chapters 3 and 4.) Generally, these approaches draw inspiration from some aspect of biology and apply it to the robotic control structure.

1.3 Contributions

This dissertation holds that both approaches have merit, and will introduce three new mechanisms for improving the use of evolved controllers on simulated and real robots. The organization of this document proceeds as follows:

Chapter 2 will describe the experimental apparatus used throughout the dissertation. A brief argument for the use of simulation is made, and then two robotic systems are introduced. The first system, Julius Popp's *micro.adam* and *micro.eva*, is a pair of underactuated ring-shaped robots that use dynamic interactions with gravity to achieve motion. The physical robots are described from both a structural and computational standpoint, and the simulation of these robots is introduced and discussed. The second robotic system, iRobot's Roomba, is introduced and its use is justified. The robot's physical and computational systems are described, and the software written to simulate these systems is introduced as well. Finally, Stanley and Miikkulainen's evolutionary algorithm NEAT is introduced and discussed. The use of NEAT is justified, the operation of the system is explained, and the parameterization of the system is documented.

Chapter 3 focuses on the use of repeated structure in evolving neural robot controllers. The chapter begins by outlining the shortcomings of typical neuro-evolutionary techniques and reviewing other approaches to overcoming them. The following section reviews the biological mechanisms for creating and regulating repeated structure, and the section after that describes a new approach that adds an evolutionary mechanism for subdividing a neural network into regions and then creating identical network structure in each region. The implementation details of this method are described, and several benchmarking experiments are used to demonstrate the system's effectiveness. The final section of this chapter describes experiments using the system to evolve controllers for the simulation of *micro.eva*.

Chapter 4 describes a process for incorporating prior knowledge of a robotic system by implementing a behavioral layer of control. The first section in the chapter outlines the deficiency of pure network controllers and the second section describes some related research efforts. The following section describes the polyclad flatworm *Notoplana acticola*, which demonstrates that a brain does not control every aspect of sensorimotor integration, but instead plays a coordinating role in activating or suppressing autonomous behaviors. The following section describes how the flatworm metaphor relates to a subsumption controller for six-legged walking, and then describes the role that a brain could play in this type of controller. The last two sections describe an implementation of a behavior-based controller for the Roomba and the results from applying neuro-evolution both directly to the Roomba and to the Roomba outfitted with a behavioral layer of control.

Chapter 5 describes a problem common to all evolutionary robotics systems: a loss of function when controllers evolved in simulation are applied to physical robots (sometimes termed the "reality gap"). The first section motivates this problem, and the second section reviews related work in solving in. The third section carefully outlines the nature of the problem on *micro.adam*, including the results of an experiment that quantifies the size of the "reality gap" when NEAT is applied to the system. The section after that describes a process for incorporating feedback from the physical robot into the evolutionary process. By automating the same fitness test from simulation on the real robot, controllers evolved in simulation can be checked for applicability in the real world, allowing the system to quickly identify and discard controllers that are effective in simulation but ineffective on the robot. The final section describes the results of applying this system to *micro.adam* and *micro.eva*.

The final chapter, 6, reviews the contributions from these experiments and suggests areas for future research.

Chapter 2

Experimental Setup and Apparatus

This chapter describes the robotic systems, simulations, and evolutionary algorithms that were used to test the claims made in chapters 5, 4, and 3. The chapter is organized as follows: the following section provides a brief argument for why the use of simulation is necessary in evolutionary robotics. The next section describes Julius Popp's *micro.adam* and *micro.eva* robotic systems, outlining both the physical and computational structures that comprise the physical robotic systems. Included in that section is a discussion of the simulation used to aid in the evolutionary experiments for evolving controllers for these robots. The next section discusses iRobot's Roomba, including both the physical and computational structures, and the Roomba simulation written specifically for this project. The final section discusses Stanley and Miikkulianen's evolutionary algorithm, NEAT, that is used throughout this dissertation, including a particular focus on the issue of parameterization.

Two robotic systems form the basis for claims about real robots in this dissertation. Both robotic platforms were designed and built without knowledge of or reference to this thesis, and both provide a non-modifiable interface. No modifications were made to either platform to make them easier to simulate, and their environments were not engineered to ease the use of, or provide better correspondence to, simulation. To the contrary, these systems have been chosen because they present real robotic issues. Dealing with the difficulties, uncertainties, and complications of the real world in the context of artificial evolution is a particular focus of this work.

2.1 The constraints of real time make simulation necessary

While the use of simulation has been criticized in the past [8], simulations are a necessary part of nearly any artificial evolutionary system. To understand why the use of simulation is critical, consider the (simulated) double-pole-balancing evolutionary experiment introduced in Stanley and Miikkulainen [61]. The task is to control a single degree of freedom mobile robot that balances two inverted pendulums of different lengths. Fitness is defined as the performance in balancing the pendulums over thirty minutes. Their system, NEAT, typically finds a solution to this problem in an average of 33,184 evaluations. Performing these tests on a real robot, working around the clock, and assuming no time wasted between tests, would require nearly two years (691 days) of evaluation time for a single run. Testing this system over various parameter settings and random seeds is, from a practical standpoint, virtually impossible. Unless the fitness test is extremely short, the number of robots enormous, the researcher impossibly (perhaps foolishly) dedicated, evolving controllers for non-trivial robots in the real world is simply not an option.[1]

The required use of simulation does not imply that just any simulation is useful. In particular, simulated environments for evolutionary algorithms can be divided into two categories: those that are written to model a pre-existing robotic system and those that are written without a corresponding robotic system. The latter type of simulation seems to be more susceptible to the type of hopeful errors that render many simulations useless when real robots are introduced. In particular, the pitfalls surrounding the modeling of sensor data can be more easily avoided when real sensor systems can provide a sense of the limitations and noise that make real systems so complicated. The simulations contained in this thesis were written to model pre-existing robotic systems, and the arguments made in defense simulations are limited to this kind of use.

[1]Unless, of course, some unforeseen breakthrough in the use of evolution dramatically reduces the number of evaluations required.

2.2 Julius Popp's *micro.adam* and *micro.eva*

Julius Popp is a German artist who works with mechanical and computational elements to create robotic art. His other works include *bit.fall*, which uses a row of small water nozzles mounted on a fifteen-foot high gantry to re-create digital images with falling drops of water, and *micro.sphere*, a ball that locomotes by shifting weight inside the sphere. For the work described in this disseratation, Popp provided the use of *micro.adam* and *micro.eva* (shown in figure 2-1). These robots are of particular interest in the context of evolutionary algorithms due to their dependence on dynamic properties. While they are artistic in function, the motion of these robots cannot be encapsulated in a purely kinematic description, making them much different from typical evolutionary robotic platforms (see section 5.2.4 for an example of a less dynamic robot).

2.2.1 The physical *micro.adam* and *micro.eva* robots

Both *micro.adam* and *micro.eva* have ring-shaped bodies, and rest on a pair of passive castors that leaving them free to rotate about their central axis. Inside the ring body, each robot has an arm or arms, contained in the plane of the ring, with enough weight to shift the robot's center of gravity, causing the robot to rotate on the castors. The *micro.adam* robot has a single arm, attached by a hinge to the ring, extending into the ring, past the center of the circle. The arm is free to sweep back and forth, constrained only by the inner diameter of the ring. The *micro.eva* robot has five arms, each attached to the ring by a pair of linkages. The linkages are of different lengths, but both the linkages are short enough to ensure that the arms do not reach to the center of the circle, and the range of motion of the arms are constrained such that the arms never collide.

These robots are underactuated, with a single active degree-of-freedom for each arm (meaning one total DOF for *micro.adam* and five total DOF for *micro.eva*), and a single passive degree of freedom that allows the robots to rotate freely on the castors. In the context of this dissertation, the goal behavior for both robots is to control the

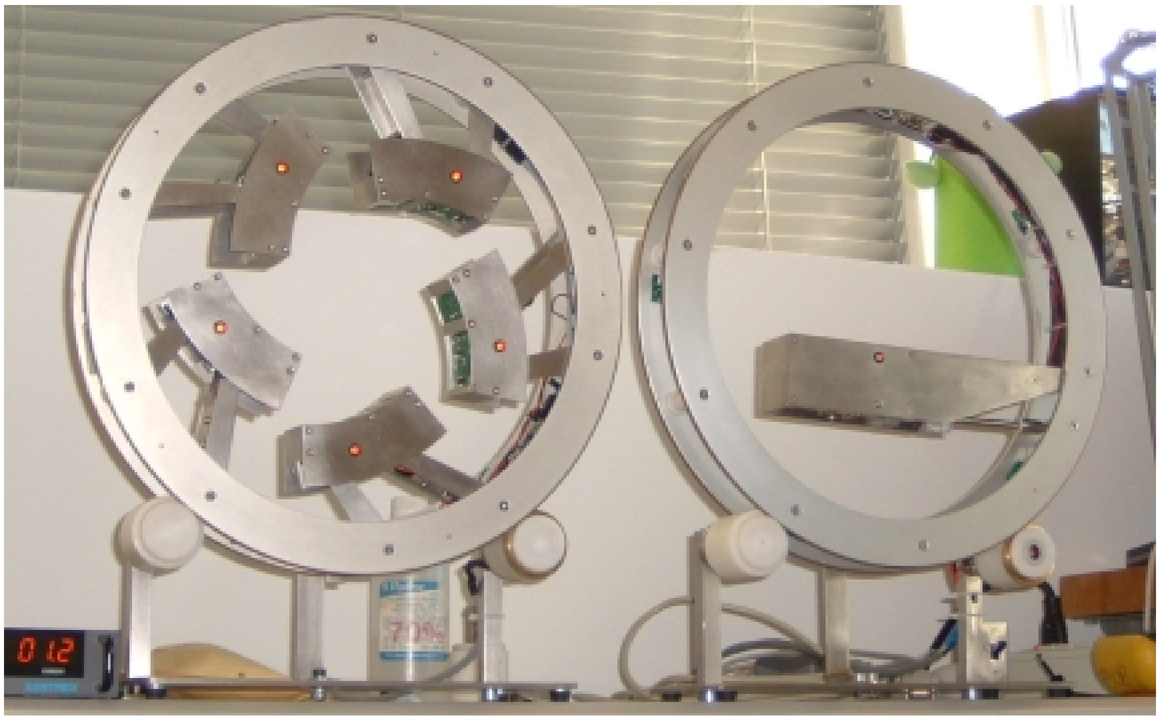


Figure 2-1: Julius Popp's robots, *micro.adam* (right) and *micro.eva* provide a unique problem domain for investigating evolved controllers.

active DOF(s) to create motion in the passive DOF. That is, like a child on a swing who pumps his legs to kick the swing higher and higher, the robots can move their arm or arms back and forth at just the right frequency to cause the ring to spin faster and faster.

The robots each communicate with a controller via a packet protocol (over a bluetooth connection). Designed as a pair, the robots are controlled by a very similar interface and provide similar sensor data. The motors that drive the arms are simple eight-bit RC servo motors, and so the command packets to the robot consist of a single eight-bit number for each arm indicating a desired position. Note that the position of the *micro.adam* arm is simply a product of the angle of the arm with respect to the ring, but the position and movement of the *micro.eva* arm includes both a translation and a rotation of the arm due to the linkage structure. The sensor information from the robot includes the position of each arm, which should only deviate from the command position due to gravity since the arms never collide with anything, as well as the motor current for each of the servos, a 16-bit reading from a gyroscope mounted on the ring, and a “ring position” sensor. The ring has five hall-effect sensors evenly spaced around the circumference of the ring, and the sensor reading is a number corresponding to the sensor that most recently passed a magnet (mounted on the base near the bottom of the ring).

In practice, these robots can be somewhat difficult to operate. Maintaining easy rotation around the passive degree of freedom requires very careful positioning and alignment of the castors, and small errors in either of these values (which inevitably accumulate over time) can both increase the friction as the robot turns and accentuate the imperfections in both the castors and the rings. Supplying power to the robots is also a non-trivial issue. Because the robots were designed to have equal distribution of weight around the circumference of the ring, mounting a battery on the body of either robot is impractical. Instead, the outer rims of the robot are metal strips through which power is delivered to the computational and motor systems of the robot. In the original design, one of the castors had two spring-loaded copper rings that are placed on the castor such that the metal strips on the robot rest directly on the metal rings, allowing the robot to be powered through this (still passive) castor. In practice, delivering power via the castor is highly unreliable; small deviations in the alignment of the robot or the metal rings cause the robot to lose power. And, even with a great deal of capacitance built into the power system of the robot, these losses of power cause the control system to reset and make extended use impractical. A new brushing system that delivers power directly to the ring (removing the metal rings on the castor entirely) has improved the reliability of the system, but at the cost of increased friction.

2.2.2 The simulated *micro.adam* and *micro.eva* system

Because these robots rely on dynamic interactions with the real world to generate movement, simulating their behavior requires a somewhat detailed model of their dynamics. For this, the open-source Open Dynamics Engine (ODE) was used to create a full simulation of the robots [58]. ODE provides the ability to predict the behavior and motion of the robots via computation, thereby allowing the user to test a control scheme entirely via computation, significantly increasing the number of evaluations able to be performed on the robot in a fixed time period. The parameters of the simulated system were set by taking measurements of the physical robots and setting the parameters of the simulation to these values. Some easily measured parameters, such as the limits on the joint angles and the top speed of the servos, were measured directly and included in the simulation. Other difficult-to-measure but very important parameters of the system, such as the friction between the ring and the castors, were measured indirectly by measuring the physical robot's behavior in response to a simple state change (e.g., moving the robot's arm to one limit and measuring the resulting oscillations) and then tuning the parameters of the simulation until that behavior is matched (the issue of properly setting the friction parameter will be revisited in the next chapter). Other hard-to-measure (and perhaps variable) parameters such as the backlash on the motors, irregularities between the arms on *micro.eva*, and small non-uniformities in mass distribution (due to computational hardware on the ring's circumference) were simply not included.

The sensors were simulated by taking readings of the sensor data on the robot and then adjusting the variables from the simulation to match the observed values. Care was taken to ensure that the units of the simulated sensors and the sensors on the robot have similar granularity; even when additional precision was available in the simulation, it was discarded. Similarly, the commands to the simulated arm were first passed through a function to limit the granularity to eight bits to mimic the effect of the RC servo. The interface to the robot and the interface to the simulation use identical conventions to prevent any "cheating" due to complete knowledge (and to simplify the process of transferring controllers from simulation to the real world).

2.3 iRobot's Roomba

The second robotic system was designed and built with a functional, not aesthetic, purpose. The Roomba [55] (figure 2-2), designed and manufactured by iRobot, drives across a floor and uses bottom-mounted brushes and vacuuming to remove dirt and dust. With over 1.5 million robots operating in normal home environments, iRobot's Roomba represents a benchmark for mass robot design. The significant market penetration of the robot implies that the value of the hand-designed navigation algorithm is sufficient to create a viable commercial product. Comparisons between this algorithm and evolved algorithms, then, give some sense of whether evolutionary navigation algorithms can achieve this level of sophistication and value.

2.3.1 The physical Roomba

Shaped like a small stack of dinner plates, the Roomba has two independently-driven wheels that allow it to travel at a straight-line speed of 500 mm/s, and three "cleaning" motors (a side brush, a main brush, and a vacuum) that sweep dirt off the floor that the Roomba travels over. The main brush, which works with the vacuum to do the bulk of the cleaning, is mounted in the middle of the robot, but does not extend the full diameter of the robot (the Roomba is approximately 330mm across, but the cleaning area is only the middle 216mm). The side brush, mounted on the Roomba's right side and towards the front, sweeps dirt on the right of the robot towards the main brush, thereby ensuring that any bit of floor that passes under the Roomba's right side is cleaned. A bumper that covers the entire front half of the Roomba provides two bits of front-collision information ("none," "left," "right," and "both") and compresses enough to allow the robot to detect a collision before stalling the motors.

The details of the Roomba's control architecture are proprietary, but the release of the Serial Control Interface (SCI) [32] provides basic access to the robot's sensors and motors. The motors are as described above; three cleaning motors that have binary state and two drive motors. The drive motors are not commanded by providing direct motor velocities, but instead by providing sixteen-bit "speed" and "radius" arguments that, by incorporating the geometry of the wheels, describes the global movement of the robot (instead of the local movement of the wheels). The SCI also allows the user to virtually press any of the buttons on the robot, or to control the non-functional aspects of the robot, such as the LEDs and speaker. Finally, the SCI also includes a single command that forces the Roomba to enter its "dock seeking" behavior, wherein

Figure 2-2: iRobot's Roomba, a popular floor-cleaning robot, operates in well over a million homes using a hand-designed algorithm.

it uses infra-red commands from a powered "dock" module that guides the robot to the dock where bottom-mounted metal contacts allow the robot to recharge itself.

The sensory system allows the user to poll the sensors on the robot. A variety of physical sensors indicates the current state of the robot; whether the bumper is compressed, whether the wheels are on the ground, whether the ground sensors are activated, or whether a button has been pressed. The robot has an IR sensor mounted on its front right side (above the side brush) that detects walls and other obstacles in close range on the right of the robot without the need for collision. The Roomba also performs odometry, and the results are expressed in the same terms as the commands, that is, distance traveled and angle swept (the odometry values are reset after each polling event). Two "dirt sensors" provide eight-bit measurements of the dirt being picked up by the robot, but the exact nature of these sensors is not provided. The SCI also provides access to some of the internal state of the robot, including stall sensors for the motors, the charge, capacity, temperature, and charging state of the battery, as well as the battery current and voltage.

2.3.2 The interface to the Roomba

Using the Roomba SCI has proven to be straightforward. The wheels on the Roomba have studded rubber tires, and the slip factor on carpet is minimal. This also means that the odometry is reliable over short distances. The sensor data is generally reliable, due, in part, to the fact that none of Roomba sensors measure the environment with much sophistication. Perhaps the greatest limitation of the SCI is the communication protocol itself. The Roomba has a built-in timing requirement: the bumper compresses roughly 7.5 millimeters, which means that, traveling at top speed (500 mm/s), the time between when the Roomba comes into contact with an obstacle and when the motors begin to stall is roughly 15 ms. Reading a full sensor packet from the Roomba and sending only a motor packet requires sending 31 eight-bit bytes over the serial line, which, given the RS232 protocol, requires sending 310 bits (each byte includes a "mark" bit at the beginning and "stop" bit at the end). At the 57600 baud, this will consume roughly 5 ms, leaving 10 ms for computation. This implies that the control algorithm must operate at 100 Hz in order to be able to detect a “bump” signal before the motors begin to stall (thereby activating the “motor overcurrent” signal).

This timing requirement has two implications. First, it implies that controlling the Roomba with a slow embedded processor will severely limit the amount of processing time available to the user’s control algorithm. For example, controlling the Roomba with a Motorola M68HC11 processor (a common eight-bit embedded processor that usually runs at 8 MHz), 10 ms of processing time would allow the processor to execute, for example, just over 3,000 integer load-multiply-store loops [51]. This kind of computational constraint will limit the complexity of new algorithms and the ability to integrate new sensors without stalling the motors (and wasting energy from the battery). The 10 ms constraint also means that the computation must either be local (riding on or tethered to the Roomba) or the network communication protocol must be fast enough to support the 100 Hz command requirement.

In order to test evolved controllers described in this dissertation, the Roomba was outfitted with a fast, local processor. A single board computer, the “gumstix” platform, is small enough to be mounted on the Roomba without seriously altering its behavior, and yet is fast enough (400 MHz) to allow time for significant computation during the 10 ms time slice. A small printed circuit board provides a serial port connection between the gumstix and Roomba, and a power connection and regulation mechanism allows the gumstix system to be powered by the Roomba battery. The result is a system that controls the Roomba at a reasonable command frequency, but is small enough to avoid compromising the Roomba’s behavior.[2]

2.3.3 The simulated Roomba system

The Roomba simulation focuses on the robot’s navigational abilities, and so the rigid body simulation of the Popp robots was not necessary. In this case, a basic two-dimensional model of a mobile robot was built from scratch. As with the Popp

[2]More information can be found at people.csail.mit.edu/bpadams/roomba.

robots, basic physical parameters such as the physical dimensions of the robot were included, and easily-measured elements of its motion were included, such as top speed and bumper compression, whereas difficult-to-measure elements, such as wheel slip, were not. Notably, high friction was assumed between the wheels and the ground and between the Roomba and any obstacles. In practice, the high-friction assumption is usually quite accurate, as the robot was run on carpet and the wheels do indeed provide a great deal of traction. The sensors were also simulated by observing real sensor data, and, for example, the robot's power discharge when the motors are stalled is modeled according to observed sensor readings while driving the robot into a wall. The same policies from the Popp robot simulations were applied to the Roomba simulation: the only information available to the controller from the simulation is the information available from the robot. One minor glitch is the simulation of dirt. Because the Roomba's dirt sensors are proprietary, it's not clear how dirt should be modeled, and spreading various kinds of dirt around and reading the sensor data proved to be too haphazard. The dirt sensors are not used in this thesis, either in simulation or on the real robot, and the rooms wherein the iRobot algorithms are tested were thoroughly cleaned (by the Roomba) in order to ensure that the dirt sensors did not impact the behavior of the robot.

2.4 NEAT: Stanley and Miikkulainen

Stanley and Miikkulainen's NeuroEvolution with Augmented Topology (NEAT) [61] is a general evolutionary algorithm that combines the evolution of network parameters (link weights) with network structure (hidden nodes and links). NEAT was chosen for three reasons: first, NEAT is widely-used and open source. This makes the claims about performance easy to reproduce and re-implementation feasible. Second, NEAT is carefully benchmarked against other approaches using evolved neural networks, and its performance is consistently and demonstrably faster and more efficient than other approaches. Third, NEAT's ability to search for both the structure and parameters of a solution removes some of the design-time assumptions that can make other evolved systems more difficult to evaluate.

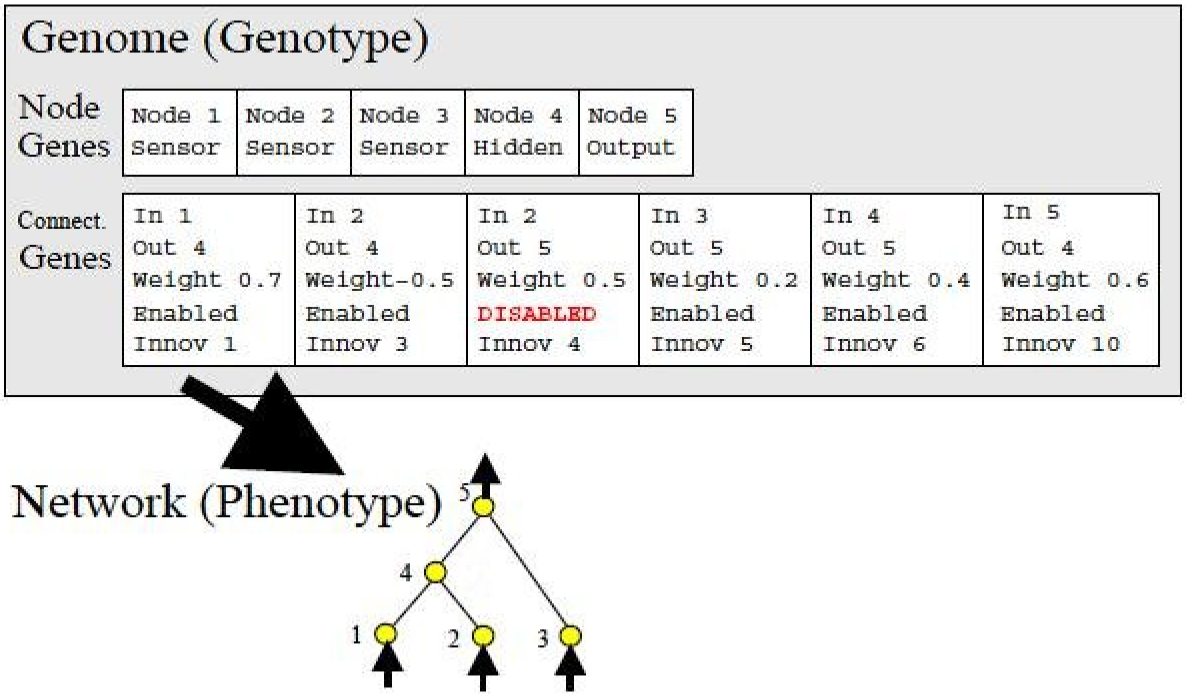
NEAT's approach is innovative and effective. Instead of assuming a typical network structure (three-layers of fully connected nodes) and adjusting only the weights via evolution, NEAT's genetic algorithm adds links and hidden nodes to an initially empty network, and then adjusts both the weights and the structure simultaneously. NEAT can therefore adjust the levels of complexity in addition to the searching for a particular solution at a given level of complexity, removing the question of problem complexity estimation from the system design domain.

2.4.1 Innovation protection in NEAT

The NEAT system keeps a global record of every link and node as it is added to a network. This record, called "innovation protection", ensures a consistent labeling scheme throughout the population (see figure 2-3 for more detail), which provides two advantages. First, when a crossover operator compares two structures, it can ensure that no gene is copied into the offspring twice. Instead, the offspring can simply contain the union of the set of links between two genomes, with the weights for intersecting links chosen at random. The result is a crossover operator that can combine solution pieces without necessarily distorting either solution individually.

The second benefit of innovation protection is the ability to sort genomes into species. Instead of assigning each genome a fitness score and then blindly allocating offspring in proportion to the fitness of each genome, the genomes share fitness with other networks in the species, and offspring are assigned according to the average performance of every network in the species. The lone exception to this rule is the use of elitism: the species with the highest performing individual in the entire population receives some bonus offspring.

Normally, similarity between two unlabeled networks would be difficult to compute, but the innovation numbers make the task trivial: lists of nodes and links are lined up according to innovation numbers, and the matching structure is easily identified by matching innovation numbers. Grouping the genomes into these clusters, or "speciating," and then sharing fitness among the groups prevents a single approach



Node 1 Sensor	Node 2 Sensor	Node 3 Sensor	Node 4 Hidden	Node 5 Output

In 1	In 2	In 2	In 3	In 4	In 5
Out 4	Out 4	Out 5	Out 5	Out 5	Out 4
Weight 0.7	Weight-0.5	Weight 0.5	Weight 0.2	Weight 0.4	Weight 0.6
Enabled	Enabled	DISABLED	Enabled	Enabled	Enabled
Innov 1	Innov 3	Innov 4	Innov 5	Innov 6	Innov 10

Figure 2-3: An illustration of a NEAT genome, from [61]. The encoding of a neural network in NEAT is extremely straightforward. The network's parameters are stored explicitly, along with the innovation numbers (described in section 2.4.1) and a boolean "enabled" bit (for more details on the use of the enabled bit, see [61]).

from dominating the population. (This approach compares with the geographic clustering suggested in [74].)

NEAT benchmarks itself by its ability to solve XOR (two inputs, one output), and it finds a solution to this problem in an average of 3600 evaluations. NEAT also does a fair job at avoiding excess structure in the XOR benchmark. Stanley and Miikkulainen report that the average solution used 2.35 hidden nodes, with a standard deviation of 1.11, meaning that the minimal structure for a solution, one hidden node, was often found. The implementation of NEAT used in this dissertation was verified by reproducing this result.

2.4.2 NEAT system parameter settings

The re-implementation of NEAT used in this dissertation includes a series of system parameter settings. Most parameters adjust the evolutionary search process (likelihood of adding structure, population size, network similarity coefficients), but others control the nature of the networks themselves (recurrent links allowed, maximum/minimum weight values). Experiments have demonstrated that changing the value of one parameter can impact results by a factor of two. Because there are 34 parameters, many of which are real-valued, searching the parameter space for optimal settings in any systematic way is impossible. Instead, a "good" set of parameters is found and used, but without any guarantees of generality. The temptation, upon using the tool in a new domain, is to "intuitively" change the parameters to better suit the problem, and a certain amount of tuning is currently inherent in the use of genetic algorithms. This dissertation will attempt to make careful note of parameter settings for each experiment. The default settings are listed in tables 2.1, 2.2 and 2.3.

Stanley's paper [61] cites the settings for 13 system parameters, and his source code formally lists another five. The NEAT implementation used in this thesis features 34 system parameters by including examples of parameters that were "hidden." For example, the 1998 publication [61] describes the function used to measure the similarity between two genomes during the speciation process as the following:

$$\delta = \frac{c_1 D}{N} + \frac{c_2 E}{N} + c_3 \bar{W} \tag{2.1}$$

where delta is the total distance, N is the number of genes, E and D are different types of mis-matched genes, $\bar{W}$ is the average weight difference of the matching genes, and $c1$, $c2$, and $c3$ are system parameters.

However, this system failed to reproduce the desired results on the XOR problem. Conversations with Stanley subsequently revealed that the results from the XOR problem had been achieved *without* normalizing the compatibility threshold on the number of links (i.e., using N in the denominator). Even though the paper did not list a binary "normalization" parameter, the impact was substantial. With normalization, the genomes were not able to differentiate from each other, and the population homogenized around local maxima. Once the normalization was deactivated, the genomes separated properly and the results were reproduced. The results were striking: the system took more than twice as many evaluations to reach a solution with normalization as without.

Other parameters, like normalization, that were not listed in the 1998 publication have been listed and their settings provided. Again, these parameter settings have been optimized for the XOR problem and, despite the potential loss of generality, held constant for other experiments.

The network parameters constrain how an individual network is constructed and mutated. Link weight limits (here, set to [-12.0, +12.0]) are not a part of standard NEAT, although they do improve performance slightly on XOR experiments. "Age protection" is a concept that is optional in standard NEAT and was used in the experiments in this dissertation. The concept is to focus weight mutations on links that have been added to the network more recently because older links are likely to have already been optimized by the evolutionary process. The use of and settings for these parameters were optimized on the XOR problem, and performance was significantly improved, especially for the multi-bit XOR problems described in section 3.

Table 2.1: NEAT Network Parameters and Settings

-1.000	Allow recurrent links? (-1=No, +1=Yes)
12.000	Maximum weight value
-12.000	Minimum weight value
2.500	Maximum value for a new weight
2.500	Maximum size of a weight change
0.250	Probability of re-enabling disabled links during crossover
10.000	Minimum number of links for "age protection"
0.200	Percentage of links considered "aged"
1.200	Scaling factor for protecting "aged" links
0.900	Probability of mutating network weights
0.500	Probability of a severe weight mutation
0.500	Normal probability of adjusting a weight
0.100	Normal probability of resetting a weight
0.700	Severe probability of adjusting a weight
0.200	Severe probability of resetting a weight

Table 2.2: NEAT Genetic Algorithm Parameters and Settings

150.000	Population size
250.000	Generations until halt
0.300	Probability of adding a link during mutation
0.030	Probability of adding a node during mutation
15.000	Generations since last improvement for species protection
0.010	Offspring penalty for a species failing to improve
0.200	Fraction of species, sorted by fitness, to use in reproduction
0.250	Probability of mutating without crossover
0.001	Probability of cross-mating between species

Table 2.3: NEAT Speciation Parameters and Settings

1.000	Disjoint link similarity parameter (see equation 2.1)
1.000	Excess link similarity parameter (see equation 2.1)
0.400	Weight difference similarity parameter (see equation 2.1)
-1.000	Normalize similarity? (-1=No, +1=Yes)
3.000	Initial species threshold
5.000	Minimum number of genomes in a species for elitism
1.000	Dynamically adjust species similarity threshold? (-1=No, +1=Yes)
0.300	Increment when adjusting species threshold dynamically
0.220	Target no. of species, as fraction of population size (high limit)
0.180	Target no. of species, as fraction of population size (low limit)

Parameters for the genetic algorithm are more easily and frequently adjusted; the number of generations is frequently increased for difficult problems and the population size is expanded as the problem becomes more difficult. NEAT also penalizes species that fail to improve fitness over fifteen generations (by scaling their share of offspring by the 0.01 "offspring penalty" in the above table). This results in species being somewhat frequently dropped from the population altogether. The original NEAT implementation included a mechanism to further concentrate offspring for the fittest individuals ("stolen babies"), but this mechanism was not implemented.

The speciation parameters are surprisingly sensitive; properly dividing the offspring into species was found anecdotally to impact the results on a given problem by as much as an order of magnitude in the speed of finding a solution. Of particular interest is the use of "dynamic speciation." The basic concept is to adjust the speciation threshold to create a "target" number of species (expressed, in the parameters, as a fraction of the total population size. In this case, the target was 20% of the population size, 150, resulting in a target of 30 species). If there are too few species after speciating generation N, then the threshold is lowered, and networks will have to be more similar to be in the same species in subsequent generations. If, on the other hand, there are too many species, the threshold is raised and the opposite result achieved for subsequent generations.

These three systems, Popp's *micro.adam* and *micro.eva*, iRobot's Roomba, and Stanley and Miikkulainen's NEAT, serve as the foundation for three sets of experiments that demonstrate how neuro-evolution can be more effectively applied to the problem of evolved robotic control. Chapter 5 will demonstrate that a system like NEAT, when evolving on a simulated robot, can effectively generate controllers for the real robot by including feedback from automated tests on the physical robot. This chapter will include descriptions of how NEAT can be adjusted to incorporate these tests and will demonstrate the results on both physical and simulated *micro.adam* and *micro.eva*. Chapter 4 will combine the NEAT system with the Roomba to illustrate the impact of a behavioral layer of control. By evolving a neural network to make connections for a subsumption architecture (instead of connections between raw sensors and motors), the performance of evolved controllers can be elevated to be competitive with the hand-designed Roomba algorithm. Chapter 3 will then make modification to the NEAT algorithm to exploit repeated structure in robots. By drawing a comparison to the embryonic development of *Drosophila*, an additional evolutionary mechanism for creating repeated structure is implemented and demonstrated on the simulated *micro.eva* robot. The final chapter will review these contributions and suggest directions for future work.

Chapter 3

Repeated Structure

Repeated structure is frequently a key element of physical robot design. Whether six legs or two arms, or a row of sonar sensors or an array of retinal cells, robots frequently feature multiple copies of a single design element. Hand-designed controllers take advantage of this repeated structure: the six-legged walking algorithm from [9] contains several elements that are copied six times, once for each leg. However, evolved neural networks for control rarely if ever reflect the morphology of the robot in this way. Even for robots with strong repeated structure, an evolved NEAT network will typically not contain any structure whatsoever. Providing a mechanism to allow evolved neural networks to better reflect the structure found in physical robots is the focus of the work described in this chapter.

This chapter is organized as follows: the next section will review some past approaches to finding and exploiting repeated structure in the domain of robot control. In particular, two sub-fields are discussed: controllers that are evolved simultaneously with morphologies and controllers that are evolved for a fixed morphology, and the argument for using evolution on the sizable base of non-modifiable robots is made. Section 3.2 will then discuss the creation of structure in *Drosophila melanogaster*, or the common fruit fly. It will be shown that repeated structure begins during embryonic development and is essentially a system for creating spatial subdivisions along the body's anterior-posterior axis. Section 3.3 will then describe how the mechanisms for creating repeated structure in *Drosophila* can be implemented as an enhancement to standard NEAT neuro-evolution. An evolutionary mechanism for repeated structure improves the scalability of evolved solutions and shortens the time to find a solution. The last two sections will demonstrate these improvements on two different problems. The first problem is a multi-bit XOR test, inspired by XOR's historical significance in neural networks, NEAT's use of an XOR benchmark, and the potential to use XOR to create a neural binary adder. The second test is a fitness task on the simulated *micro.eva* robot.

3.1 Related work

The discovery and use of modularity has been a key focus of many evolved systems. This section will focus on evolved systems with robot control (either simulated or real) as a target task. The first section will discuss the use of evolved morphologies as an alternative method, that is, evolving both a body and a brain instead of trying to evolve a brain for a fixed body. The second section will discuss more directly related approaches to creating structure in evolved neural networks.

3.1.1 Evolving structural morphologies

Biological controllers naturally exploit repeated structure because the mechanism for creating the morphology and the control structure are one and the same. Whereas controllers evolved for hand-designed robots are attempting to discover a pre-existing (and non-modifiable) structure, biological controllers are free to create structure in both the body and controllers simultaneously. Several research approaches have noted this fact and explored the concept of evolving robot bodies and brains simultaneously.

Karl Sims' virtual creatures [57] are one of the earliest examples of evolving symmetric control structures. His genomes achieve repeated structure by simply building the repeating mechanisms directly into the genotype. A virtual creature's body and control structure are created by reading a digraph, and if the path along the digraph visits a node more than once, that structure is be repeated. The nodes in the digraph include information about both the morphology and the control, and the result is simulated bodies that have both repeated structure and identically repeated control. Sims repeatedly calls his units of control “neurons,” even though (as he admits) the units compute such non-neural functions as “interpolate,” “divide,” and “memory.”

The creatures were tasked to perform a number of basic functions like locomoting and swimming, and the wide variety of unique solutions demonstrates the enormous potential of artificial evolution to innovate. But Sims' creatures were evolved in simulation without any constraints that might make the creatures plausible real robots. It is unclear if the digraph genotype would be a plausible strategy for creating a controller for a real robot.

Josh Bongard's Artificial Ontogeny system [6] also has a growth and development phase that is key to developing modularity. Recalling Sims' work, Bongard's creatures evolve a morphology and control structure simultaneously. His creatures use a genetic regulatory network to control the expression of different genes, and the result is creatures that start from a single unit, divide, grow, differentiate, and ultimately are tested in a variety of tasks, including directed locomotion and manipulating simulated objects. Like Sims, Bongard's creatures are not meant to translate to the real world, but the sophisticated results are striking.

Bongard observes in [5] that many of his best-performing evolved creatures displayed evidence of modularity between the genes that create the body and the genes that create the control system. It is unclear what aspect of the regulatory network (if, indeed, that is the source) caused these creatures to develop modules.

Lipson and Pollack have addressed one of the central concerns about evolving brains and bodies simultaneously, that is, how to bring a structure evolved in simulation into the real world. In [56], they describe a process for evolving both controllers and morphologies in simulation and then constructing them in the real world (by hand) using rapid prototyping technology. Evolved solutions to a locomotion task (similar to the simulated task introduce by Sims) demonstrate interesting and innovative solutions. Hornby [27] uses a similar technique to evolved tables using recursive grammars and then builds those tables in the real world. While these techniques require human intervention to build the structure created in simulation, the idea is fascinating and may point to an exciting future.

But, while this approach demonstrates tremendous potential, there are still important reasons to focus on evolving controllers for hand-designed robots. First, hand-designed robots represent a significant "user base," and developing systems to evolve controllers for them is a worthy goal simply due to the potential size of the impact. It further seems unlikely that hand-designed robots will be made obsolete by evolved designs in the near future. Second, evolved morphologies transferred to the real world raise serious concerns about general applicability. There are many domains where the constraints of the physical environment are difficult to fully encode in a simulated environment. Cost constraints, reliability concerns, manufacturing issues, or safety requirements could constrain the potential outcomes in such as way as to limit the ability of evolution to generate creative morphologies that are still satisfactory in the real world (see 5 for a more complete discussion of the difficulty of using simulation). In these cases, robots with fixed morphology should not preclude the creation of an evolved controller.

3.1.2 Evolving structure in neural nets for fixed morphology

Frederic Gruau [21] [71] extends the ideas from Sims by including ideas of "cellular encoding."[1] Instead of digraphs that are traversed to create a phenotype, Gruau's system uses geometrical BNF grammars that start from a single cell, divide, and connect. As one might expect, these grammars produce regular networks that have "modules." Gruau uses these grammars to assemble Boolean networks to control a simulated six-legged robot (the simulated robot was created by Beer in [3]) as well as the double pole-balancing problem introduced in chapter 2. Experiments show regular gaits on the simulated six-legged robot.

GasNets, described by Husbands et al. in [59], is an attempt to evolve non-static neural networks that are capable not only of controlling a robot, but learning as well. The name "GasNets" is derived from the biological diffusion of gaseous neuromod-

[1]A review of cellular encoding work is available in [62].

ulators in biological neural networks. The nodes in a GasNet network have spatial locations in a two-dimensional plane, and the diffused signaling channel can alter the transfer functions of neurons within the diffusion radius. Husbands et al. are particularly careful to compare their evolved controllers against similarly evolved controllers without the diffused neuromodulator model, and their results are promising. A simulated robot shape differentiation problem is more easily solved with GasNets than with networks that cannot make use of the diffusion model. Nolfi et al. have a different take, providing outputs designated as "teaching units" to help the network learn in a backpropagation-like manner. [52]. These results are verified on a small mobile robot obstacle-avoidance behavior.

Chris Adami and Alan Hampton [24] stress the developmental approach, and have a complex system, *Norgev* [2] [1], to test their ideas. Neurons in *Norgev* live in a hexagonal grid and use intercellular communication to self-organize. The system also uses logic benchmarks, but instead of XOR, evolves a double NAND gate (the use of double logic gates will be referred to later in this chapter). But instead of modularity, Adami and Hampton stress the complexity of their system, and illustrations show that, in their evolved solutions, the two NAND gates are solved with perhaps hundreds of connections and each NAND gate actually uses a separate mechanism.

Calabretta, Nolfi, Parisi, and Wagner also claim a system that demonstrates emergent modularity in [11]. Operating with a Khepera, [38] they built a fully-connected neural network that, essentially, has three output neurons for each actuator: two control neurons and a selector neuron. Evolution selects the weights on the basis of the Khepera's ability to find colored objects, pick them up, and carry them to a target location. The authors find that the networks do evolve "modules," and further find that evolving the modules *during* evolution (that is, starting with less modularity, but adding it through the use of genetic operators) provides better results. The paper finally suggests that modularity may actually be a secondary effect of duplication, as opposed to a phenomenon in its own right.

Gregory Hornby [27] cites modularity, regularity and hierarchy as the most important qualities for an evolutionary algorithm, but instead of expecting them to emerge, he builds these qualities into his system by working within a recursive grammar. His genomes are highly general, and can be applied to a wide variety of domains, in this case, the evolution of a stable table. Subsequent work with recursive grammars has been applied to morphology-control problems (similar to Bongard and Sims), antenna design, and neural net controllers for pole balancing and for robotics, specifically and Aibo (the Aibo work is reviewed in chapter 5).

Martin Hülse, Steffan Wischmann, and Frank Pasemann evolved neural controllers for Popp's *micro.eva* robot in simulation. But, instead of building modularity into their system, they impose modularity by hard-coding the problem into subdivisions. In the case of *micro.eva*, the create one sub-network for each of the five arms, and evolve them both homogeneously and heterogeneously. They also test the use of oscillating units to create additional movement. They present results in simulation that demonstrate the robot rotating at 1/4 revolutions per second in [29] and at 1/2 revolutions per second in [72].

An extension to modularize NEAT was developed by Reisinger in [49]. Like the system described below, this modular NEAT implementation attempts to subdivide the network before evolving connections, and, while developed separately and without concurrent knowledge, the Reisinger work shares some qualities with this work (such as binding inputs and outputs to patterns). Two key differences are made: Reisinger's work specifically evolves different modules as opposed to focusing on repeating modules. While a series of repeating modules would be feasible, Reisinger is more focused on problems wherein different parts of the network solve different problems, whereas this work is concerned with instances where network structure can be repeated exactly multiple times. Second, Reisinger incorporates the module use into the link and node structure, whereas this work separates the modularity component from the structural component. All links and nodes are associated with a specific module in Reisinger's work, whereas the links and nodes are created globally and without the knowledge of the particular module into which they will be incorporated. This is a design tradeoff: modularized crossover makes more sense in Resinger's work, but searching for the modularization separately from the network structure makes this work more flexible.

3.2 *Drosophila melanogaster* and repeated structure

The power of modularity comes from its ability to take a single complex problem and break it down into solvable sub-parts. Previous works have made claims about modularity taking approaches that range from hard-coding the modularity in [72] to allowing modularity to emerge from some other set of principles [5]. To better understand how biological processes perform this step, a closer look at the pattern-formation of the common fruit fly, *Drosophila melanogaster* is instructive. This section will examine the use of gap genes and the use of pair-rule genes in *Drosophila*.

3.2.1 Gap genes in *Drosophila*

Drosophila has a unique development phase wherein the embryo is technically a single cell, but the nucleus of that cell has divided several times, creating a structure called a "syncytium" [42]. The lack of cell wall means that the signaling proteins between the nuclei are not separated into inter-cellular signals and intra-cellular signals, but are instead diffused over the entire length of the embryo. The genes responsible for these signals are known as "gap genes," for reasons that will soon be clear. There are many gap genes, but this explanation will focus on just five. From anterior to posterior, these genes are *hunchback* (*hb*), *Krüppel* (*Kr*), *knirps* (*kni*), *giant* (*gt*), and *tailless* (*tll*).

The anterior-posterior axis is established by a protein gradient present from the mother, in this case, a protein from a gene called *bicoid* (*bcd*). The gene expression by the nuclei in the syncytium varies according to where in the *bcd* gradient the nuclei falls. This varying expression leads to varying expression of the gap genes, and, as the gap genes are produced, they cross-regulate as well. In most cases, the gap genes down regulate their neighbors: *hb* inhibits the production of *Kr*, *Kr* inhibits the production of *kni*, and so forth [30]. The result is a series of "protein stripes" aligned perpendicular to the *bicoid* gradient, illustrated in figure 3-1. (Recent research indicates that this process may be slightly more complex, including some feedback from genes previously thought not to be involved in the morphogenic process [33]).

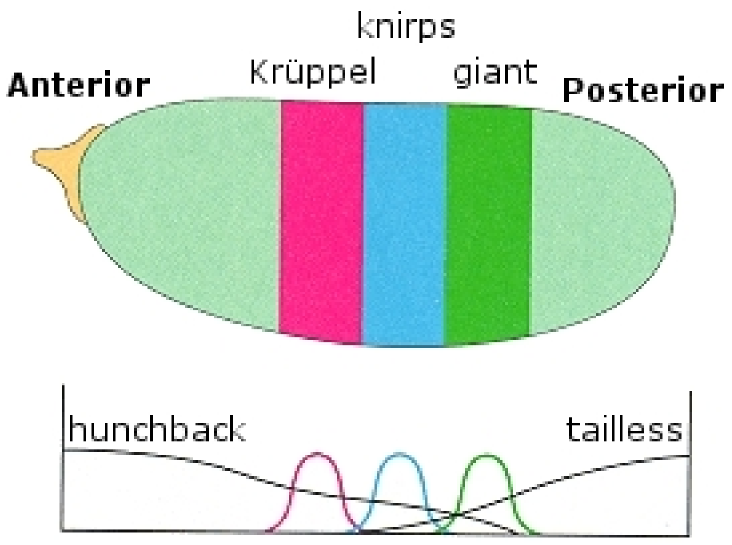


Figure 3-1: The gap genes diffuse across the length of the *Drosophila* embryo, creating a set of regions and boundaries. (Image reproduced from http://genetics.biol.ttu.edu/genetics/pictures/embryo.gif.)

3.2.2 Pair rule genes in *Drosophila*

Once these stripes are in place, a second series of genes can go to work: the pair-rule genes. These genes, such as *fushi tarazu* (*ftz*) and *even skipped* (*eve*), are expressed on a segment-by-segment basis, and the impact of their expression is limited to a given segment. The *ftz* gene, for example, is a transcription factor that promotes the creation of a strong anterior boundary in every other segment. The *eve* gene plays the same role, but in the segments not expressing *ftz*. Other pair-rule genes operate on every segment; *wingless* is a gene that orients the denticles (small hairs) on the ventral side of the larva. The denticles for each segment are oriented towards the middle, which indicates that *wingless* operates on a segment-by-segment basis. And yet, in *wingless* mutants, the denticles for the entire fly are disrupted, and the pattern is random across the entire body. Other pair-rule mutants create the same effect (their names, usually describing the knock-out mutant, provide a hint: *hedgehog*, *naked*, *disheveled*).

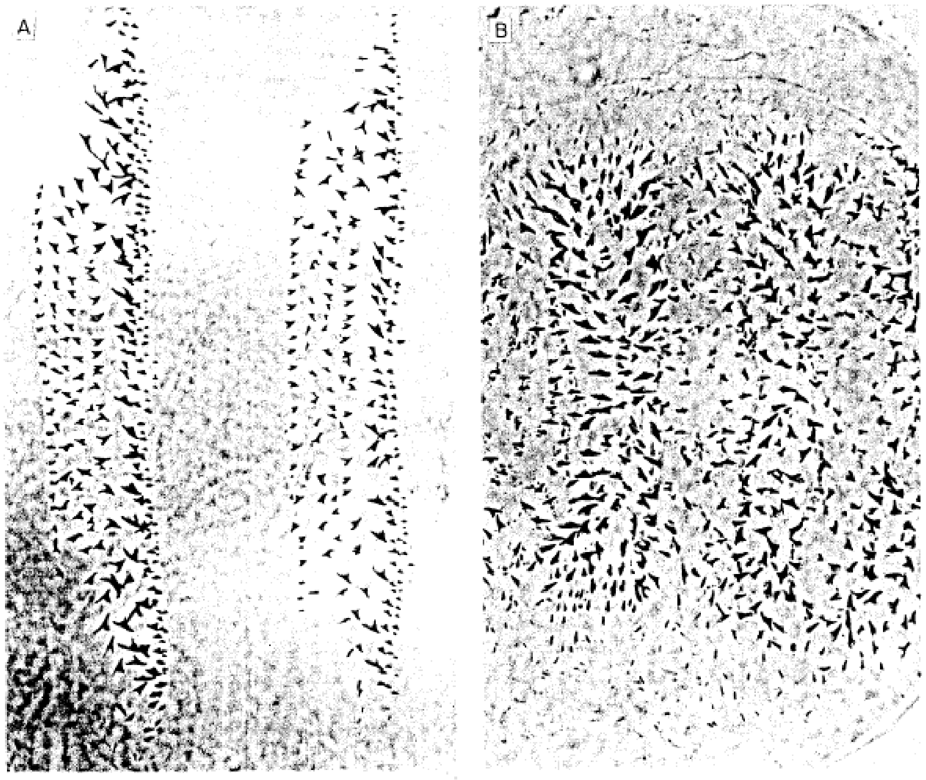


Figure 3-2: The denticles for a wildtype *Drosophila*, left, line up into even bands. However, an interruption in the gap genes results in mutants such as *wingless*, where a lack of developmental structure leads to a chaotic wildtype.

While these processes have been most clearly observed and analyzed in *Drosophila*, other research indicates that this concept — pattern formation on the basis of an original gradient, resulting in repeated structure — is at work in many different animals and at many different levels. For example, analogs to many of the genes in *Drosophila* development have been identified in the flour beetle *Tribolium* [16]. Patel finds evidence for many *Drosophila*-like processes in crustaceans [54]. Clarke identifies similar processes for the creation of brain segments, called rhombomeres, in chick brains [13].

A few important ideas can be drawn from the structural repetition in *Drosophila*. First, repeated structure is fundamentally a spatial process. The maternal *bcd* gradient defines distances from the anterior boundary, and the subsequent segments are all positioned with respect to it. Second, the individual segments, while repeated, are not periodic. Each segment is defined by a unique protein, and its boundaries are set by interactions with neighboring proteins. Repeated structure, then, arises from genes that perform the same operation on a set of unique, spatially defined segments defined by the anterior boundary of the creature.

3.3 NEAT with repeated structure

The biological processes that create repeated structure in *Drosophila* serve as the starting point for extending NEAT to exploit repeated structure on a robot by creating repeated neuro-evolutionary structure. The work in this chapter approaches repeated structure from a functional perspective; changes to NEAT should be grounded in biology enough that the benefits are fully captured by the system, but only to the extent that the enhancements add measurable improvement to the system's performance. To that end, it is important to explicitly state and justify the level of abstraction chosen to model this process. Because these enhancements are added to NEAT, which is itself a rough abstraction of a biological process, examination of NEAT's assumptions is also important.

This section will first state and explore the abstraction used by NEAT, then extend the idea to include a biological abstraction of repeated structure. The following subsection will provide technical details regarding changes to the NEAT genome, and the subsection after that will describe changes made to the NEAT genetic algorithm, including new parameters and settings. The final subsection will describe how the enhance NEAT genome is decoded to create repeated structure.

3.3.1 Abstraction description and justification

NEAT is, in part, an abstraction of the biological process of neural development. How animal brains create connections between neurons is very complex and not entirely understood. The basic mechanical features of the process are as follows: cells in the brain, both neural and non-neural, emit inter-cellular signaling proteins. As these proteins diffuse and degrade, they create overlapping gradients that can extend as far as an entire hemisphere of the brain. As a neural cell extends an axon, the tip of the axon, called the “growth cone,” senses some subset of these chemical gradients and its growth is attracted by some gradients and repelled by others. The sum of these forces guides the growth to a destination, where a set of shorter-range gradients may take over for target localization. There are also signals that can be activated once an axon is at its destination to remove improper connections [65].

NEAT models this entire process by simply focusing on a simplified version of the result: connections between neurons are represented as an interconnection matrix, listing which pairs of nodes are connected and how strong those connections are. The entire process of emitting proteins and following gradients is simplified into a direct encoding of the final state. It’s important to note that this simplification ignores many elements of the biological system, including elements are critical to the success of the biological system. For example, NEAT’s abstraction assumes that the entire connection matrix is genetically encoded. This is known to be false, the changes that occur during early development are critical to the survival of some animals. Catalano [12] demonstrates that blindfolding cats during an early development phase prevents the proper connections from being formed in the visual cortex. NEAT’s success despite this notable omission demonstrates that an abstraction need not be perfect to be effective.

This notion of imperfect but effective abstraction can be applied to the biological processes found in *Drosophila* as well. Just as the specifics of protein emission and gradient following in axon guidance were summarized by a connection matrix, the maternal gradient and subsequent segment formation can be summarized by a set of spatially defined regions. With these regions in place, the actions of the pair-rule genes in *Drosophila* can be implemented by decoding the genome independently in each region.

These ideas are implemented as a set of spatial and structural enhancements to the NEAT neuro-evolutionary system, called “NEATstruct” for convenience (see figure 3-3 for a summary of changes). NEATstruct adds spatial data to each node in the

Table 3.1: NEAT genome data

Nodes	integer	Type (input, output, bias, hidden)
	integer	Innovation number (see section 2.4.1)
	float	State variable
Links	boolean	Enabled bit
	integer	Source node
	integer	Destination node
	integer	Innovation number (see section 2.4.1)
	float	Weight

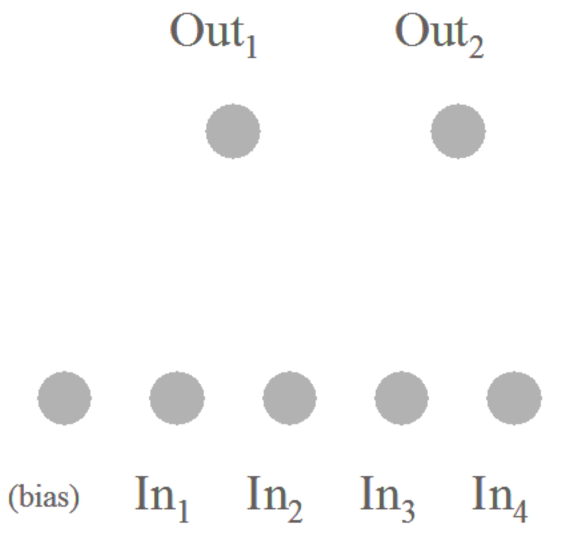
NEAT network, providing a foundation for structure to be created and repeated. The second addition is a list of subdivisions that represent regions in a spatial NEAT network. These subdivisions define regions that are analogous to *Drosophila* segments. NEATstruct also adds evolutionary mechanisms to add and mutate these subdivisions during the evolutionary process, allowing the spatial network to be subdivided in a variety of ways. Finally, NEATstruct adjusts the decoding mechanism that translates a genome into a network. Instead of a NEAT genome connecting nodes globally throughout the network, each region is treated as an independent sub-network, and the genome is decoded entirely in each region. The following three subsections will describe the details of the implementation of these ideas.

3.3.2 Changes to the NEAT genome

The regions in the fly embryo depend on the the spatial arrangement of the nuclei in the syncytium (figure 3-1). The diffusion and degrading of the proteins happens over a physical distance, and those physical distances are key for the proper development of the fly. This means that the idea of regions can only have meaning in a framework that includes a location for the elements.

The first difference between structural NEAT and plain NEAT is the inclusion of user-defined spatial information for each node. The plain NEAT algorithm begins with an empty network of input and output nodes, and their “arrangement” is merely a graphical interface convenience.[2] As the NEAT algorithm chooses links to add, its

[2]One implementation of NEAT, by Buckland [10] assigns a coordinate system to the neurons,



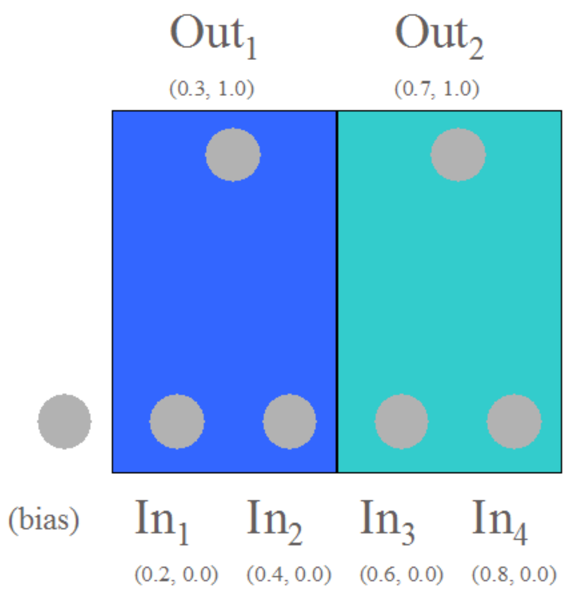


Figure 3-3: A diagram of a regular NEAT network (top) and a NEATstruct network (bottom). The NEATstruct network has two significant additions: first, the nodes have spatial data that provide an arrangement that reflect the structure of the problem (see figure 3-4 for details). Second, the nodes are divided into regions (green and blue) where the genome can be decoded multiple times, twice in this case.

Table 3.2: NEAT genome data with structural enhancements

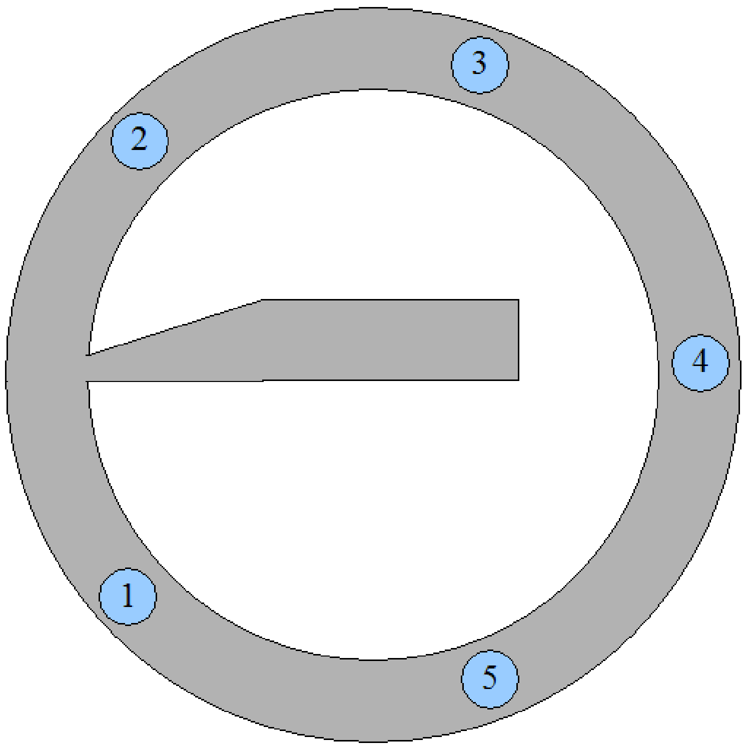
Nodes	integer	Type (input, output, bias, hidden)
	integer	Innovation number (see section 2.4.1)
	float	"X" position
	float	"Y" position
	float	State variable
Links	boolean	Enabled bit
	integer	Source node
	integer	Destination node
	integer	Innovation number (see section 2.4.1)
	float	Weight
Subdivisions	float	"X" position

only considerations are the type of node: input, output, or hidden. Because no spatial data exists in the genome, the order of the inputs and outputs could be completely scrambled without any change in function. Structural NEAT, on the other hand, arranges the input and output nodes in a configuration that reflects the structure of the problem. The nodes are given two floating-point values to determine their configuration, and convention holds that inputs have a "Y" value of 0.0 and that outputs have a "Y" value of 1.0. The "X" values for the input and output nodes are chosen by the system designer to reflect some structure of the problem (see figure 3-4 for details). For example, if ring position sensor 0 on *micro.adam* is defined to have an "X" value of 0.0, and if the entire creature is defined, for convenience, to have a length of 5.0, then ring position sensor 1 would have and "X" value of 1.0, ring position sensor 2 would have an "X" value of 2.0, and so forth.[3] The sensors for the arm data - the arm current and position - would have "X values between 1.0 and 2.0 because the arm on the physical robot is located between ring position sensor 1 and ring position sensor 2.

Once the inputs and outputs are arranged in a meaningful way, the system can then try to divide the network space into segments using subdivisions (see table 3.2). A subdivision is a single floating-point value that specifies an "X" position of a regional boundary. A structural NEAT network starts with a set of spatially

but its only functional use is to determine if a potential link would be recurrent.

[3]Recall that the ring position sensors are evenly spaced around the ring.



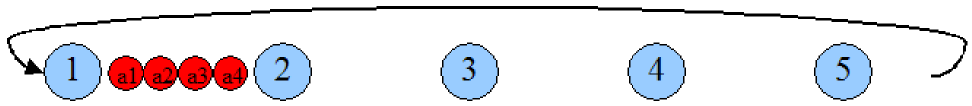


Figure 3-4: Determining the spatial layout of the input and output nodes for the *micro.adam* and *micro.eva* robots is straightforward. The ring structure provides a framework for the spatial relationship of the input and output nodes. In this example (only input nodes are shown), the nodes for the ring sensors are evenly spaced in the network because they are evenly spaced on the robot. The arm sensors are then placed between ring sensors 1 and 2 because the arm is located between ring sensors 1 and 2 on the physical robot.

parameterized input and output nodes and two subdivisions, an "anterior" subdivision and a "posterior" subdivision. The anterior subdivision will remain fixed throughout the evolutionary life of the network, but the posterior subdivision may be moved, as will be seen in the following subsection.

3.3.3 Changes to the NEAT genetic algorithm

NEAT's mutations allow the genetic algorithm to add and modify links and nodes. Structural NEAT performs these mutations, but can add a new boundary in the middle of region, effectively dividing that into two new regions. Structural NEAT can also add a new region to the posterior end of the creature, which will perhaps incorporate new input and output nodes if the previous location of the posterior end of the creature did not encompass all input and output nodes. (Mutations are summarized in table 3.3.) The procedure for adding a new link to the system is also adjusted slightly; instead of picking nodes at random from the entire network, structural NEAT first chooses a region, then chooses a source and destination node from within that region. If a region does not contain enough nodes to complete this mutation, the genetic algorithm chooses a new mutation to perform. When the link is added to the genome, any input or output nodes are recorded not by global innovation number, but by counting nodes relative to the nearest anterior boundary. That is, if a subdivision exists between input nodes 3 and 4, say, a new link connecting to input node 4 will be labeled with "1," because 4 is the first input node posterior to the boundary between input nodes 3 and 4. Note that a link added to a region that is subsequently mutated will have its referent change. The impact of this change will be reviewed in the next subsection.

These additions to the genome and genetic algorithm require new parameters. Table 3.4 lists the new parameters and the settings used in the below experiments. The parameters were chosen to make subdivision mutations roughly as likely as adding a hidden node. The addition of a new type of mutation means that the likelihood of a weight mutation falls slightly (since weight mutations are suppressed when another type of mutations happens), so the severity of weight mutations is increased (this

Table 3.3: NEAT network mutations

NEAT	Add a link Add a hidden node Mutate a link weight
Structural NEAT	Add a link Add a hidden node Mutate a link weight Add a new subdivision Change the location of a subdivision

Table 3.4: NEAT Structural Parameters and Settings

NEAT parameters adjusted	
0.800	Normal probability of adjusting a weight
0.100	Normal probability of resetting a weight
0.950	Severe probability of adjusting a weight
0.300	Severe probability of resetting a weight
New structural NEAT parameters	
0.045	Probability of adding a new subdivision
0.300	Probability of mutating a current subdivision
0.100	Given mutation, probability of reversing a current subdivision
0.900	Given mutation, probability of changing the location of a subdivision
0.500	Standard deviation of the movement of an existing subdivision
0.250	Probability of extending the posterior subdivision of the creature
0.500	The mean amount to extend the posterior subdivision of the creature
0.200	Probability of adding a new link that crosses a subdivision
0.200	Probability of adding a link that is unique to a single subdivision
4.000	Parameter for speciating by subdivision

adjustment was made after trial-and-error testing).

3.3.4 Changes to the NEAT genome decoding process

The genome is translated into a network using the inverse of the link-adding procedure outlined in 3.3.1 for genome G. An empty network is constructed and initially filled only with the input and output nodes specified by the system designer. Each region is then considered in turn. Hidden nodes are decoded with respect to the anterior-most region boundary. If a hidden node's spatial information (that is, it's "X" and "Y" values) would place it outside the current region, it is not decoded. After all hidden nodes are in place, the entire list of links is decoded in the region. If a link refers to an input, output, or hidden node that does not exist in the current region, the link is not added. When each region has been decoded in this manner, the network is complete.

Algorithm 3.3.1: Structural NEAT network decoding(G)

```
Add all input and output nodes in $G$
for each region $\in G$
    do {
        Add all hidden nodes from $G$
        if Hidden node's spatial data would place it outside the region
            then Do not add the node
        Add all links within this region
        if Link refers to node not present in region
            then Do not add link
    }
```

If a subdivision is mutated after links have been added in such a way to include an input or output node that was previously not in the region, the algorithm will decode the links with different sources and destinations than were originally intended when the link was added. If an input or output is removed from a region by a subdivision mutation, then those links will not be expressed in the mutated region. These silent changes and failures allow the system to adjust the position of region boundaries without invalidating previously evolved neural structure.

A few notable exceptions exist in this algorithm. The first deals with communication between regions. The biological analogy is unclear on this issue. *Drosophila* doesn't feature structure that "crosses" its structural boundaries, but instead shifts the definition of the boundaries many times over the course of development. So a simple, non-biological hack has been added to the NEATstruct system to allow for these types of connections. When a region boundary is decoded, the algorithm inserts two nodes into the network: one that is designated for "anterior-to-posterior" boundary-crossing links, and one that is designated for "posterior-to-anterior" boundary crossing links. Those links are added (and decoded) like any other link. Second, there is a mechanism for specifying that a link should only be decoded in a single region, allowing the symmetry to be broken. In this case, a link is added in the typical way, but only decoded into the region in which it was added.

3.4 Results on XOR

In Stanley and Miikkulainen's NEAT [61], the XOR problem is used as an evolutionary benchmark. The XOR problem is historically significant in the field of neural networks because it is provably impossible for a summing neural network to solve without a hidden node [50]. The implication of benchmarking a system on XOR is that a system can, at the very least, solve a problem requiring one hidden node. For evolutionary systems, this can be non-trivial. While a classification task like XOR is somewhat removed from the task of controlling a robot, results evolving neural networks to solve XOR problems are presented in this section both for demonstration and for benchmarking.

The XOR task that NEAT uses as a benchmark includes two input nodes and a single output node. The goal is to properly compute the XOR of the two input bits (represented by -1.0 for "0" and 1.0 for "1"). The network is said to have given the proper output if the true XOR value is 1 and the output node supplies a value of 0.5 or greater, or if the true XOR value is 0 and the output node supplies a value less than 0.5. The fitness of the network is determined by testing it on every possible input combination and rewarding a correct classification with a fitness score of 1.0 and rewarding an incorrect classification with a fitness of $1.0 - \mathit{output}$. As noted in section 2.4, NEAT typically solves this problem in roughly 3600 evaluations (with a population size of 150 and all other parameters set as listed in section 2.4.2), and often uses the minimal structure, that is, a single hidden node.

But consider XOR not as a classification problem, but instead as a part of a circuit. Solving XOR is frequently the first step to creating a binary adder. But a binary adder must solve XOR for a series of bits instead of just one. If we encode the problem of solving the XOR of two two-bit numbers (instead of two one-bit numbers), then the problem begins to take on a spatial, repeated quality (see figure 3-5). Note that this problem is very similar to the previous problem: the fundamental unit is XOR, but it needs to be solved twice. Yet, when NEAT is run on the 2XOR problem, instead of finding a solution in 100% of run (with an average of 3600 evaluations), NEAT only finds a solution within 25,000 evaluations in 2% of runs. Because NEAT does not consider the layout of the problem, this simply looks like a four-input, two-output problem, and NEAT must search among the entire space, not realizing that any link between the inputs and outputs of non-matching XOR instances (say, node In_1 and Out_2 in figure 3-5) will prevent the network from finding a solution.

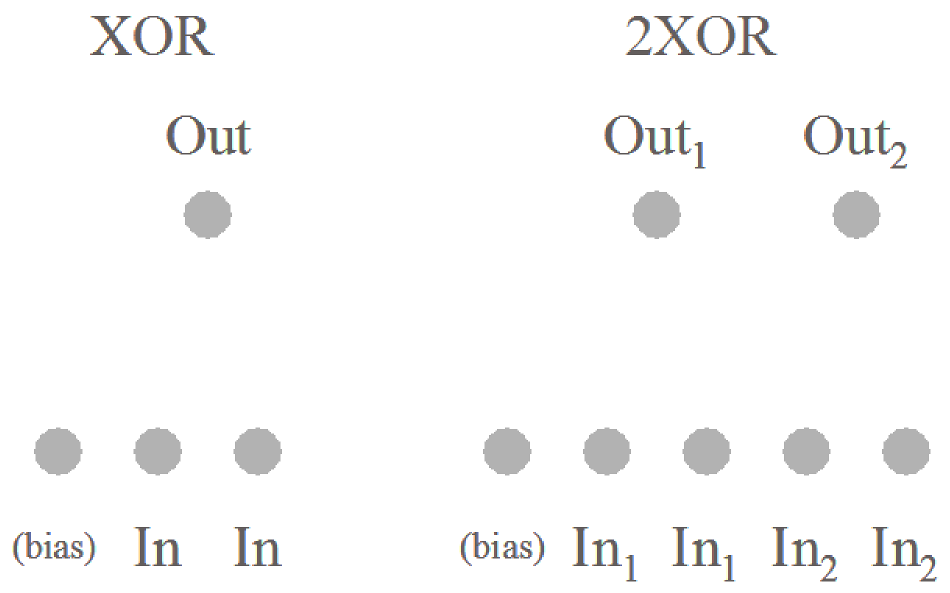


Figure 3-5: The neural representation of both XOR (left) and multi-bit XOR (right). Multi-bit XOR is a precursor to a binary neural adder, and solving that problem demonstrates the importance of an evolutionary mechanism for repeated structure.

In this case, structural NEAT's region-based approach is perfectly suited to solve this problem. As the algorithm searches for the proper links and weights, it also searches for the proper subdivision of the problem. The arrangement of the input and output nodes provides information about the structure of the problem, in this case, that there are two XOR modules that need to be evolved. As various regions are delineated, genomes that divide up the problem accurately (i.e., with the anterior end to the left of the left module, the posterior end to the right of the right module, and a subdivision between the two modules) have a tremendous fitness advantage.

The results are striking: structural NEAT solves this problem 100% of the time in an average of 53 generations (with a population size of 150). Moreover, as the problem continues to scale up, to 3XOR, 4XOR, and 5XOR, the required generations to find a solution grow linearly (solutions displayed in figures 3-7, 3-8, 3-9, and 3-10).

Sadly, the evolution of a neural binary adder appears to be beyond the capabilities of the NEATstruct system. Even with the regions and structure are both fixed, finding the appropriate weights is too difficult a task for this genetic algorithm. Finding an evolutionary mechanism that evolves a multi-bit neural binary adder is an open problem.

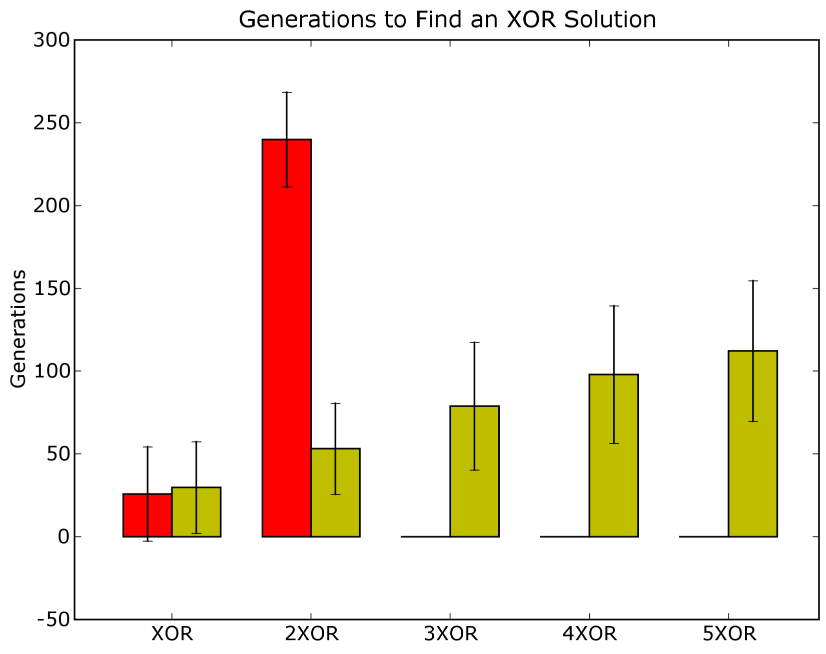


Figure 3-6: The experimental results on multi-bit XOR are striking. While standard NEAT slightly outperforms NEATstruct on the basic XOR test, NEATstruct finds solutions to 2XOR, 3XOR, and 4XOR with only linear growth in the average number of generations required to find a solution. NEAT fails to find a solution within 250 generations (and a population size of 150) 80% of the time on 2XOR and never finds a solution to either 3XOR or 4XOR. Data is averaged from 100 runs.

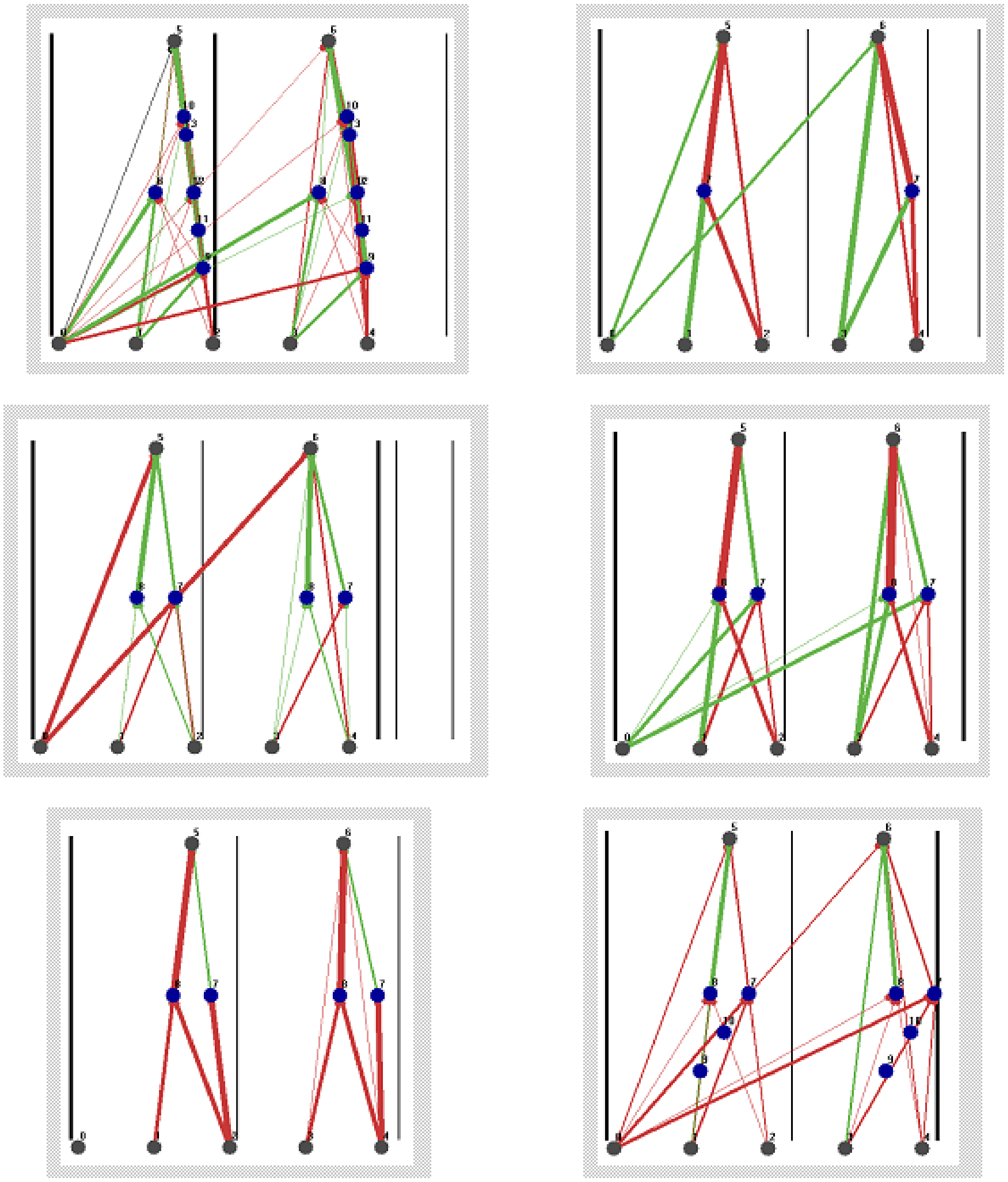


Figure 3-7: Networks evolved to solve the 2XOR problem using NEATstruct. These networks were randomly selected from the pool of successful runs.

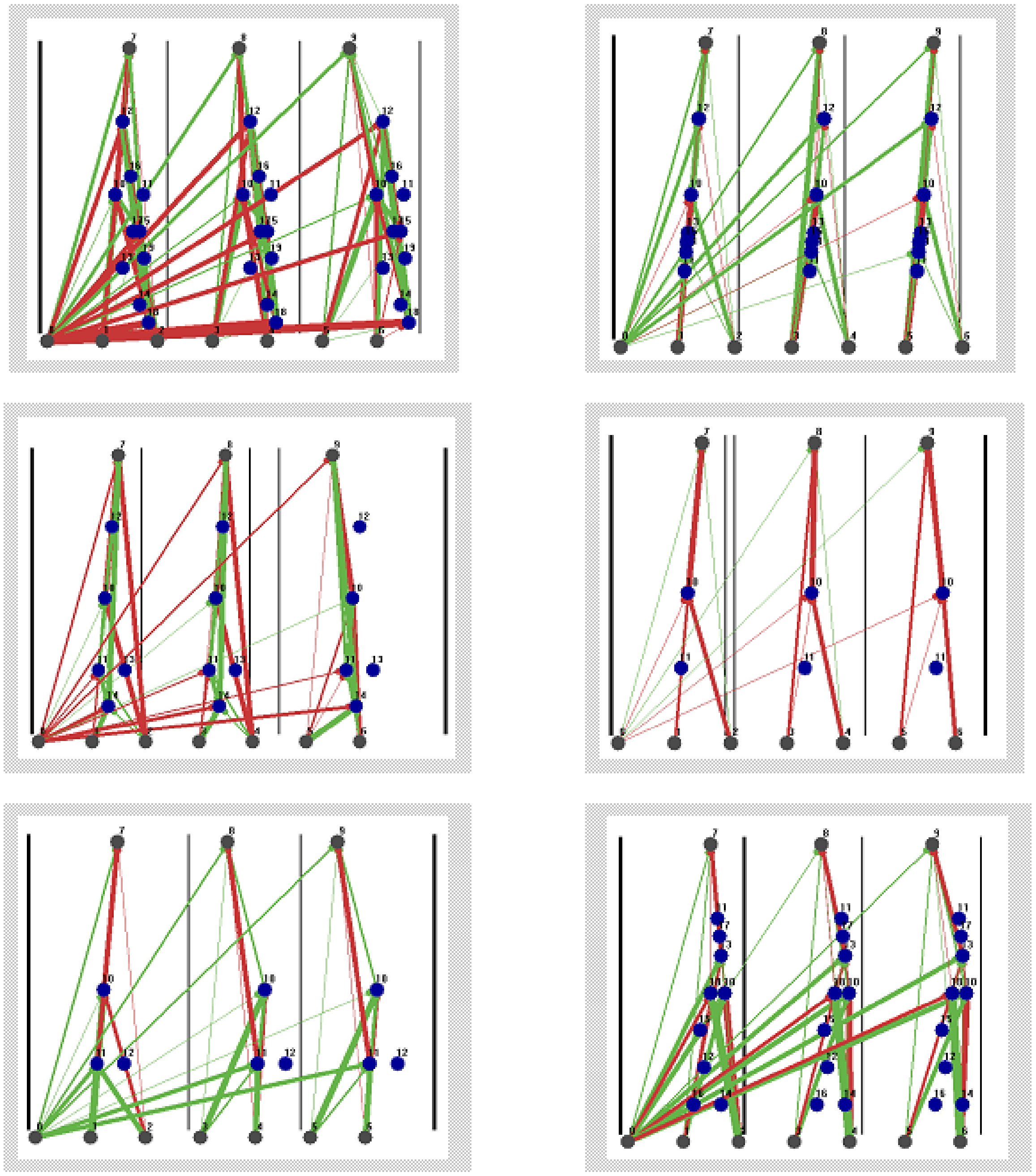


Figure 3-8: Networks evolved to solve the 3XOR problem using NEATstruct. These networks were randomly selected from the pool of successful runs.

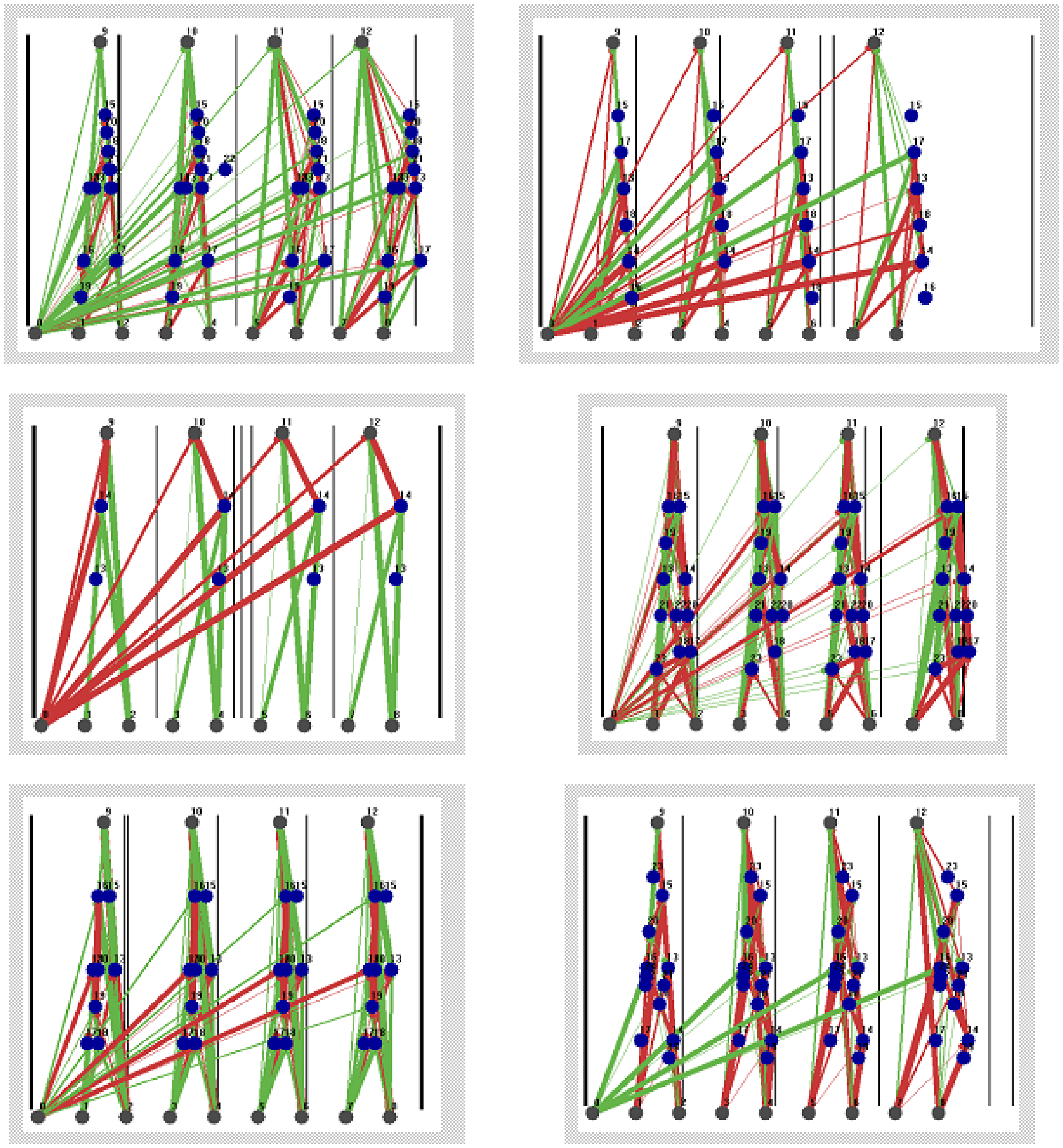


Figure 3-9: Networks evolved to solve the 4XOR problem using NEATstruct. These networks were randomly selected from the pool of successful runs.

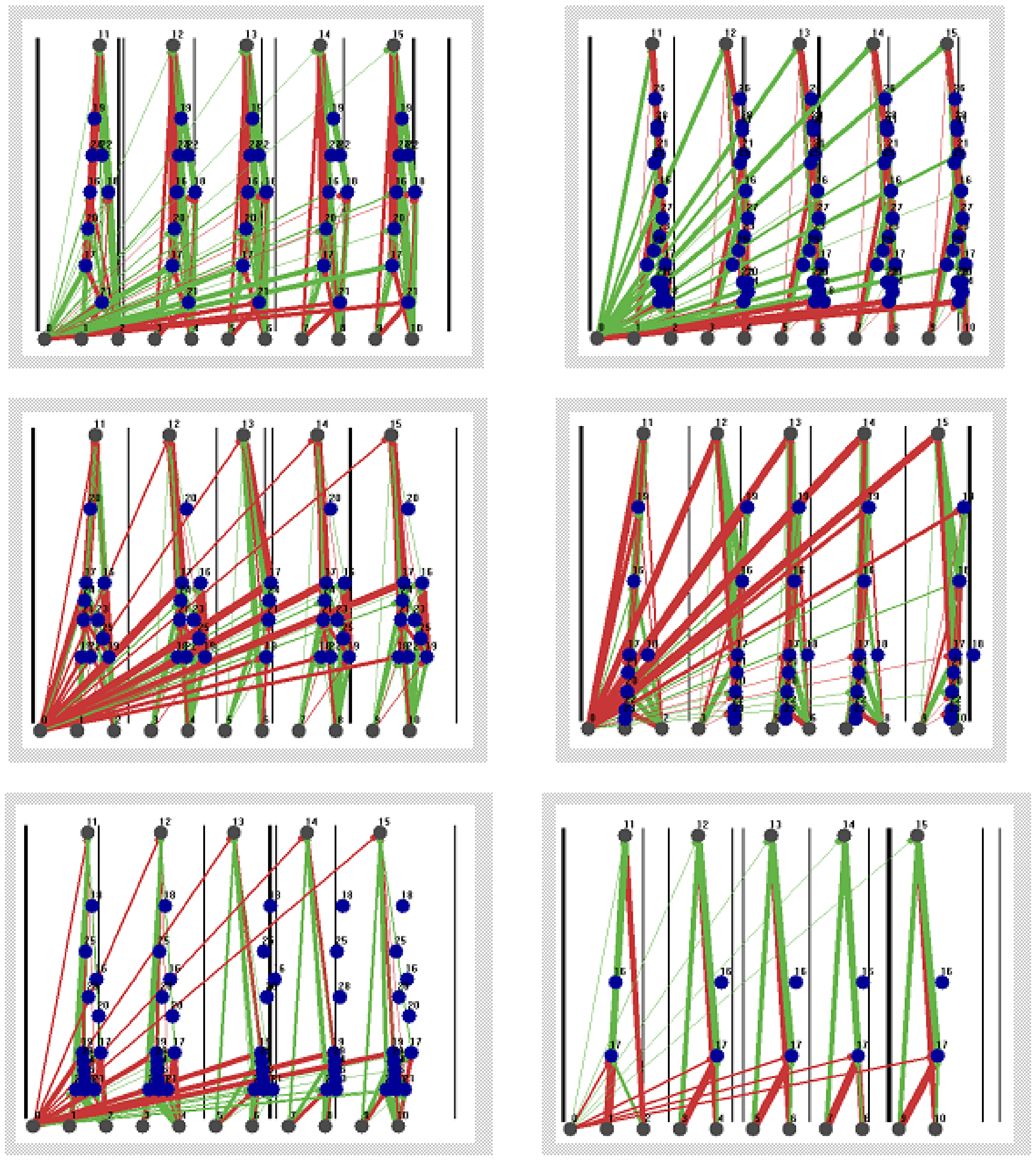


Figure 3-10: Networks evolved to solve the 5XOR problem using NEATstruct. These networks were randomly selected from the pool of successful runs.

3.5 Results on simulated *micro.eva*

The same system (with the same parameters) used to evolve multi-bit XOR solutions was then used to evolve controllers for the simulation of *micro.eva* (described in 2.2). The target task was to generate angular displacement in the robot's passive degree of freedom, that is, to rotate the robot on the castors as much as possible. The fitness was scored as

$$F = \left(\sum_{t}^{T} |\theta_t - \theta_{t-1}| \right)^2 \tag{3.1}$$

where T is the time of trial (10.0s in this chapter), θ is the angle of the passive degree of freedom on *micro.adam*. The fitness is averaged over five trials, each with a randomized starting position.

Initially, the results for *micro.eva* were disappointing. Because structural NEAT constrains the problem to a vertical segmentation, the solutions that involved segmentation were less fit than those that were free to cross vertical boundaries. However, if the output neurons were spatially shifted to the right or left by the space between two arms (that is, if the same spacing was used, but with the arm indecies incremented or decremented by one), then the results were excellent. It appears that, for simulated *micro.eva*, the repeated structure must map between sensors and actuators that are neighboring, but not immediately across from, each arm. The results from three runs are demonstrated in figure 3-11 and the networks resulting from NEAT and NEATstruct are compared in figures 3-12 and 3-13.

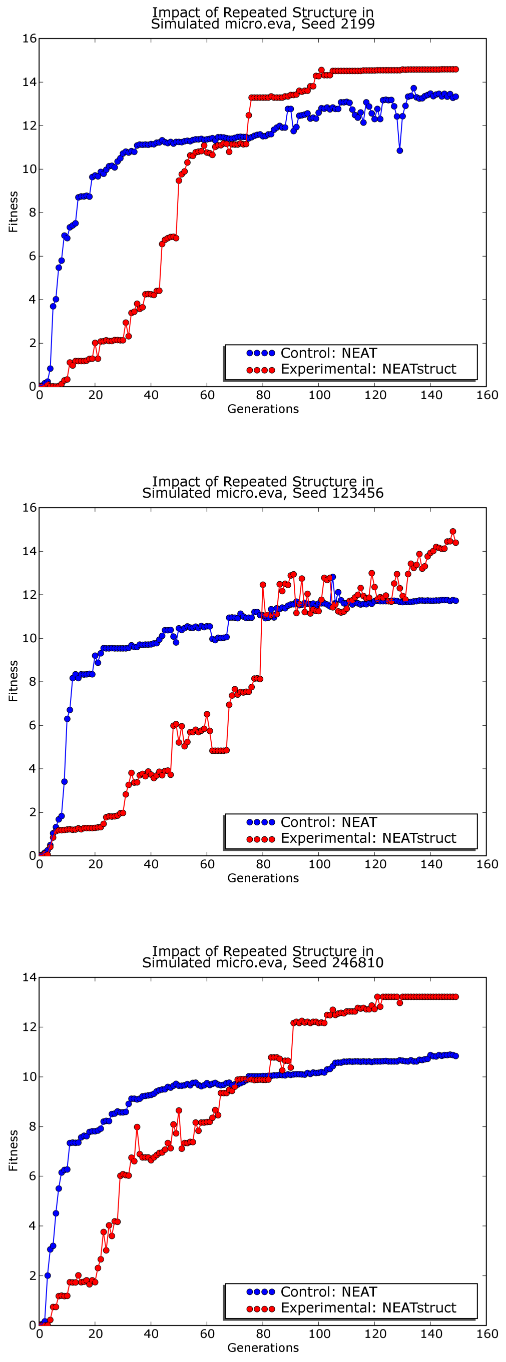


Figure 3-11: NEATstruct outperforms NEAT on simulated *micro.eva*.

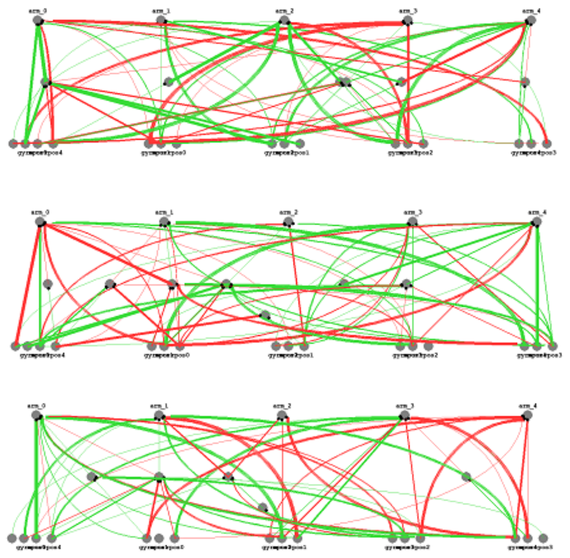


Figure 3-12: Winning networks evolved by NEAT for an angular displacement task on *micro.eva*
.

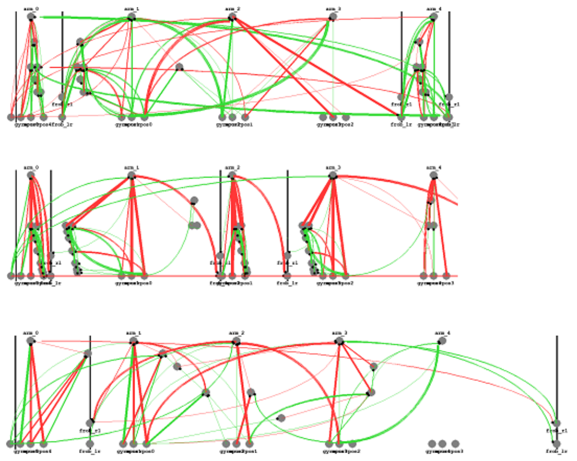


Figure 3-13: Winning networks evolved by NEAT for an angular displacement task on *micro.eva*. Although one might expect even subdivisions, as were found in the multi-bit XOR problems, the subdivisions frequently spanned several arm-sensor groups, and in only one case totaled five. The movement generated by these solutions suggests that, in fact, some movement strategies operate not by moving individual arms but by moving sets of arms. The ability to create and adjust unique segments allowed the evolutionary algorithm to employ repeated structure in a variety of ways to solve the same problem.

Chapter 4

Behavioral Primitives

Most evolved neural network robot controllers are evolved as the sole source of control for the entire robot. The network has a series of inputs corresponding to sensors and a series of outputs corresponding to actuators. All sensory data is fed into the network, processed, and then all actuators are commanded entirely by the outputs of the network. Because all sensorimotor loops pass through the network, the evolutionary algorithm must search through the entire space of all possible controllers when, in fact, the vast majority of controllers are obviously poorly suited to the task. The evolved controller starts with an inert robot and must discover, sequence, and execute appropriate behaviors simultaneously.

Biological evidence suggests that animal brains did not evolve in this way. The polyclad flatworm is believed to be among the first creatures to evolve a central nervous system (as opposed to the peripheral nervous system found in more primitive creatures). Yet, when the flatworm brain is removed entirely, the worm is still capable of basic stimulus-response behaviors. Instead of generating behaviors, the worm's central nervous system serves as a coordinator for behaviors that are generated elsewhere. These experiments suggest that an artificial central nervous system should be supplied with an artificial peripheral nervous system.

This chapter describes how such a system might be both designed and evolved. The following section will review other approaches to evolving this kind of layered controller. Section 4.2 will provide a more in-depth look at the polyclad flatworm to better understand the power of a peripheral nervous system. This section will demonstrate that the first phylogenic brains were not evolved to be sole sources of control, but rather coordinators for a well-developed distributed control network. The following section will review the use of one particular kind of distributed control in robots: subsumption architecture. The section after that will describe how subsumption architecture and evolved neural networks can be combined to control the Roomba (see section 2.3 for more information on the Roomba platform). The final section will provide results demonstrating improved performance with the use of a subsumption control layer on a short cleaning task for the Roomba. Results are also demonstrated on the physical Roomba, including a comparison with the hand-designed commercial algorithm.

4.1 Related work

Viola [69] describes one of the first systems for evolving a distributed architecture for mobile robot control. Using the subsumption architecture described in [9] (and below in section 4.3), Viola evolves a controller for a simulated sheep in a hexagonal field. The controller is comprised of modules that can access the simulated sheep sensors and drive the sheep around. Fitness is obtained by eating grass, that is, visiting new hexagons. While this work is now somewhat dated, it demonstrates that evolving distributed controllers is feasible, even with very little computational power (by today's standards).

Koza, in [41], used genetic programming to evolve a re-implementation of the subsumption architecture described in [45] on a simulated robot. By first re-implementing subsumption architecture as a series of Lisp statements, and then by further running his results on only a simulated robot, Koza makes it very difficult to compare the performance of the hand-designed subsumption architecture with the evolved one. Criticisms in [8] also point out that some of the units of the Koza evolution were used after having been validated in reality on a real robot. Nevertheless, the simulated results are compelling, and, while this early implementation was not followed with other work for some time, it remains an interesting example of how these ideas could be combined.

Lee et al. [43] approach the problem of evolving complexity by decomposing the task. Their approach uses genetic programming to evolve behavior-selection mechanisms. The fitness measure for their system is a two-stage task: a Khepera robot must first find a block and then push it towards a light. The system demonstrates that two distinct behaviors can be evolved and then properly selected to achieve this complex task. Their system evolves the lower level of behavior (as opposed to the work in this chapter, which uses hand-coded behaviors). Because only two sub-behaviors were used and the task only required a single switch between behaviors, there is some question about whether this method would scale. Moreover, concerns from evolving systems for the Khepera are discussed in section 5.2.4.

Hornby et. al. [26] evolved a neural controller to play soccer on a Sony Aibo using a hand-designed sensory and motor interface. The robot uses distance measures and a camera, and, instead of using a "neural retina" approach (such as [28]), the color of the ball is assumed to be fixed, and the inputs to the network are at the level of "ball is visible" and "number of pixels with the ball's color." Moreover, the motor commands to the Aibo are not individual motor commands, but primitives like "step forward" and "step right." This kind of higher-level sensing and actuation is closer to a behavioral model, and the network is also supplied with command-feedback sensors, such as a "last command was step forward" sensor, however, no higher-level behaviors are implemented to accompany the sensor and motor signals. The results from simulation are applied to a real Aibo, and the results are qualitatively labeled "good," although a brief comment about the differences between the simulated and real kicking of the ball suggests some serious discrepancies.

Téllez et al. [64] demonstrate that distributed, evolved neural networks can control a simulated autonomous agent. The sensors systems on a simulated mobile robot are assigned independent neural networks to control their sensing behavior. These networks are evolved separately from the central controller that can interface with the sensor sub-networks. The goal behavior is to find and orbit an obstacle in the environment, and the results are compared with the evolution of a single central controller. Using the distributed networks seems to both improve the speed of evolution and reduce the complexity of the solution. Applying this method to a more complex task (and including a physical robot) might yield interesting results.

More recently, Togelius [66] [67] demonstrated the value of what he calls "layered" evolution. The approach taken by Togelius is faithful to the original subsumption ideas. The layering, however, is achieved by altering the evolution by hand. A single individual is actually passed through four different, sequential evolutions, each designed to optimize one behavior. A simulated mobile robot (modeled after, Togelius says, the Khepera) was first evolved with a "phototaxis" fitness function, that is, moving towards light. Once this behavior was optimized to an arbitrary degree, a second evolution was begun, only the results from the first evolutionary run (the "phototaxis" run) were held constant. Evolution proceeded this way until the full creature was evolved. The simulated results from this work are interesting, although the limitation of evolving the layers separately potentially misses some interesting opportunities to interconnect different modules.

4.2 Polyclad flatworms and distributed control

The simple polyclad flatworm (figure 4-1) is an unlikely target for close examination of neurobiology. With roughly two thousand neurons in its central nervous system, (CNS), these worms are only capable of the most basic survival behaviors. But the flatworm brain is interesting not because of its complexity, but because of its simplicity [39]. Polyclads are the most primitive living creatures with a CNS; more primitive creatures typically survive with only a peripheral nervous system (PNS). By examining the role of the primitive flatworm brain, a better understanding of the purpose of simple CNS structures can be attained.

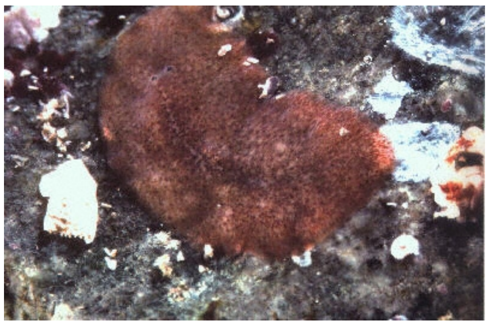


Figure 4-1: A polyclad flatworm, reproduced from http://www.rzuser.uni-heidelberg.de/ bu6/

4.2.1 Anatomy

Organisms in the phylum Platyhelminthes, commonly known as flatworms, are divided into four groups, most of which are parasitic. A small fraction of flatworms (roughly 4,500 known species) are free-living and belong to the class *turbellaria*, but the bulk of turbellarian flatworms are marine and can be found in the oceans where they are usually found in shallow water. Polycladida is a taxonomic subgroup of the Turbellaria, and the polyclads are a highly diverse group of free-living flatworms. Large, brightly colored, and typically found in tropical reefs, these free-living worms are generally no more than a few millimeters thick and range in length from less than one millimeter to over 30 centimeters [70].

Most polyclads are oval-shaped with a smooth dorsal surface and a ruffled perimeter, termed the *margin*. The outer layer of the worm is a single cell layer, called the epidermis, which features cilia that aid in motion. Just underneath the epidermis, flatworms have two layers of muscles: an outer layer of circular muscles that squeeze the body like a tube and an inner layer of muscles, parallel to the long axis of the body, that allow the worm to contract [68]. Under muscle layers is the parenchymal tissue and organs such as the highly branched gut, the reproductive system, and the nervous system.

4.2.2 Nervous system

As noted earlier, polyclads are cephalized, meaning that there is an anterior region where neural and sensory structures are spatially close. Polyclads have very primitive light-sensing cells located periodically on the epidermis, typically used to help guiding the creature away from light. There is also a more developed "eye" that senses both light and shadows and helps the worm orient towards movement or light. The worms interact with the world through "pseudotenticles," located on the anterior end, that have both tactile sensors and chemoreceptors.

This sensory information is fed into a ladder-like nervous system that, like the muscles, runs both lengthwise and across the worms body [23]. Of primary interest are the longitudinal nerves that extend from the posterior end of the worm to the anterior, cephalized end, where they join with the aforementioned two-thousand neuron brain (figure 4-2). Six pairs of nerves form the ventral nerve network, termed the ventral submuscular plexus. While the ventral plexus is more fully developed, a finer plexus exists on the dorsal side as well [40]. Of particular note are the many neural cells and neural apparatus scattered throughout the flatworm nervous system. This decentralized PNS reflects perhaps an earlier evolutionary stage wherein a fully decentralized network helped the worm's ancestors to survive.

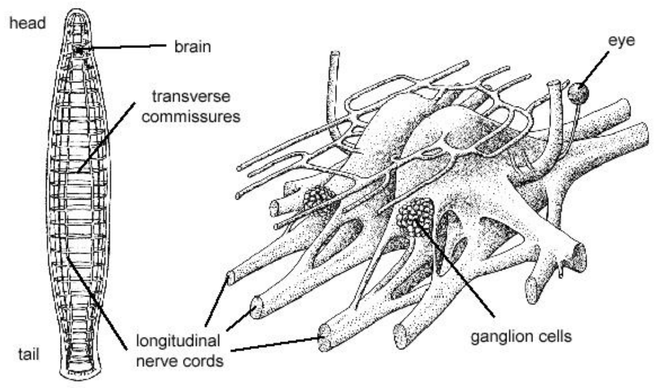


Figure 4-2: The nervous system of *Notoplana acticola*. The brain communicates with the body through a series of primary nerves (labled "longitudinal nerve cords"). Picture reproduced from http://www.rzuser.uni-heidelberg.de/ bu6/Introduction12.html

The brains of invertebrates typically consist of an outer rind of glial cells and an inner neuropile of axons and dendritic branches where integrative processes occur. In the polyclad flatworm, however, the ganglion structure is more diffuse and less connected. The rind of glial cells is indistinct, and cells are scattered throughout the brain. However, unlike most invertebrate brains, the neurons that make up the brain proper are frequently multipolar and widely branched, indicating a surprising level of sophistication in the connectivity of the flatworm brain [39].

4.2.3 Brain removal experiments

The role of the flatworm brain has been examined by observing the worm's behavior with an intact brain and then removing the brain and attempting to elicit the same behavior. In particular, four behaviors have been repeatedly observed and noted with both intact and decerebrate flatworms, in this case, *notoplana acticola* and *planocera gilchristi* [22] [14] [40].

Righting. The most basic behavior, the worm is able to flip from a dorsal-side-down orientation to a dorsal-side-up orientation by twisting its anterior end dorsally, grasping the substrate, and pulling the rest of its body over. However, decerebrate animals execute this behavior at random, thrashing wildly until a chance contact with the substrate allows the worm to make an initial grasp onto the substrate whereas the intact worm makes this move once.

Fleeing or avoidance. This experiment was carried out with a chemical stimulus for *notoplana* and a manual stimulus with *planocera*, but in both cases, the intact worm retracts the part of the body that has been stimulated. The decerebrate worm, though will react by either flailing the anterior end of the worm or by writhing its outer margins in waves that radiate out from the point of the stimulus.

Locomotion. The worm normally moves using three techniques. First, the tiny cilia on the ventral epidermis of the worm can slowly inch the worm along a substrate. Second, the worm engages in a crawling behavior wherein one lateral half of the anterior of the worm extends and grasps the substrate while the other half of the posterior end contracts, and then the behavior repeats with the opposite lateral halves. Finally, the worm can swim by rhythmically undulating the lateral margins. Decerebrate animals will locomote primarily using the cilia, although manipulating the worm can elicit some of the muscular contractions that are associated with locomotion.

Feeding. The feeding behavior is the most complex and provides perhaps the clearest insight into the role of the brain in regulating behavior. *Planocera* will, upon finding a snail, manipulate its prey with its pseudotenticles such that the hard shell is away from the animal and the flesh is closer to the ventral side and pharynx. The worm then everts its pharynx, extracts the flesh, and swallows the snail. When a decerebrated worm makes contact with a snail, it immediately everts the pharynx and gropes blindly for the flesh of the snail, only finding it and consuming the snail by chance. *Notoplana* undergoes a similar change between intact and decerebrated feeding. A worm with a brain, when presented with food somewhere along its margin, will grasp the food with the closest part of the margin, then thrusts its anterior end towards the part of the margin holding the food, regrasps it, and then inserts the food into its mouth. Decerebrate *Notoplana*, though, will simply grasp food with the closest part of the margin and stuff the food directly into its mouth.

These results suggest a role for the brain that centers on the coordination of autonomous behaviors. If the brain were a central relay unit, with all sensori-motor information traveling to the brain and all motor commands emanating from it, then the decerebrated worm would be limp and unresponsive. Instead, the worm is not only responsive, but is able to successfully (if inefficiently) execute many of the behaviors necessary for survival. The flatworm brain, viewed in this light, is less like a unique, critical organ and more like a typical evolutionary adaptation.

4.2.4 Brain transplant experiments

Further experiments with the flatworm brain demonstrate the simplicity of its role. *Notoplana* cannot regenerate its brain: when the worm is decerebrated, it will never regain the lost behaviors on its own. The worm's powerful regenerative abilities do include the connections to the brain, though, and a decerebrated worm will restore the connections to the brain if it is re-inserted and can undergo full recovery of the lost behaviors. Moreover, if two flatworms are decerebrated, one brain can be transplanted into the other worm. That is, the brain from worm A can be placed in the body of worm B, and the brain-A-body-B worm will frequently recover all of the above behaviors. This startling discovery not only underlines how simple the brain itself must be, but how simple the interface to the PNS is. The signals the brain processes cannot be complex, worm-specific encodings, but instead must communicate information that is general enough to translate between worms.

For example, one might hypothesize the various types of information that the PNS delivers to the brain during the worm's righting behavior: the lengths of some of the key muscles, tactile information from the pseudo-tenticles, or sensory information about its orientation. One might further suppose that the brain, based on this sensory information, then produces information about which muscles should contract, and in which order, to produce the desired righting behavior. However, as [23] points out, polyclad musculature is extremely sophisticated for such a primitive animal, and the particular shape and arrangement of the muscles in a particular worm will never be identical to that of another. For the transplanted brains to be as effective as they are, the level of information processing must be sufficiently abstract, and the type of processing done with that information sufficiently simple, to allow a brain that has no knowledge of any worm other than its own to successfully control a totally novel specimen.

Further examples of this type abound. In [14], brains from *notoplana* were not only transplanted between worms, but were implanted in the host worm's body in a different orientation. Removing the brain from a donor worm involves cutting through the entire dorso-ventral axis around the perimeter of the brain. This resulting "brain patch" was then implanted into the host after being rotated in three ways: reversing the dorsal and ventral sides (a D/V rotation), reversing the anterior and posterior ends (an A/P rotation) or reversing both the dorsal-ventral and anterior-posterior ends (an A/P-D/V rotation). The results were surprising in that, in each example, the worm was able to regain some of the measured behavioral functions. Ditaxic walking, for example, was regained most easily and by all three types of reversals. The A/P and A/P-D/V reversals successfully recovered a great deal of the righting and avoidance functions, although the D/V reversals did not. None of the reversals successfully regained the feeding behavior, although some worms would occasionally demonstrate the food re-grasping behavior. Moreover, further experimentation demonstrated that the brain could be placed anywhere along the four main longitudinal nerves - say, in the tail - and still regain function, including in instances with brain reversals.

The clear implication of these experiments is that the brain did not evolve to be the only control mechanism in animals. Local controllers, responsible for small but complete behaviors, are instead activated or suppressed or modified by other controllers, and the brain, at least in the flatworm, is less of a homunculus and more of a traffic cop. When considering an approach for evolving controllers for robots, this philosophy can be readily applied.

4.3 Brains and subsumption

The distributed control demonstrated in the decerebrate flatworm has parallels to the robotics world. The walking algorithm proposed by Brooks in [9] demonstrates that simple, asynchronous modules, each reading sensors and issuing commands, can coordinate a six-legged robot. Just as the flatworm maintains some of its behaviors without any coordination, many of the network elements described by Brooks are entirely functional without any interconnection. Further, the sub-networks that comprise the overall architecture have parallels to the decerebrate behaviors of the flatworm: first simple behaviors to keep the worm/robot in a suitable posture (righting/standing), then clumsy walking, then directed locomotion (avoidance/approaching). It is notable that the paper describing this network is subtitled "Emergent Behaviors from a Carefully Evolved Network," even though that particular network was arranged by hand.

4.3.1 Walking via subsumption

Consider the walking algorithm from [9], with an illustration reproduced in figure 4-3. Each box represents a control module, described as an "augmented finite state machine" (AFSM). AFSMs are like regular finite state machines, but also include timing elements (called "alarm clocks") and formalized input and output, in this case, eight-bit registers. The walking algorithm contains 57 of these modules; in the diagram below, each box with a solid band on the top is repeated once, a box with a striped band is repeated twice, and boxes with no band are repeated six times. Of particular note are the lines connecting the boxes, described as "wires." As their name implies these lines passively transmit messages from one AFSM to the other, asynchronously and without alteration. Each AFSM can have several input registers, and so multiple wires leading into AFSMs write to different registers. Contention between wires is settled via the nodes between AFSMs, labeled "i" for inhibitory, "d" for default and "s" for suppressive. Default or suppressive nodes allow one wire or the other to have control over the channel, and a wire tapped with an inhibitory connection goes silent when the tapping wire is active. Boxes with triangles in the lower right-hand corner indicate control over an actuator, and triangles in the upper left-hand corner indicate input from sensors.

The inputs and outputs to the AFSMs are not (necessarily) inputs and outputs to and from the motors. Communication with the robot takes place *within* the individual AFSMs while communication *between* the AFSMs is more abstract. Consider, for example, the "beta force" AFSM. This machine is responsible for reading motor current data from the servo controlling the legs, detecting when an abnormal amount of force is being applied, and sending a message to indicate that it believes that the leg has struck an obstacle. However, it cannot simply threshold the force reading;

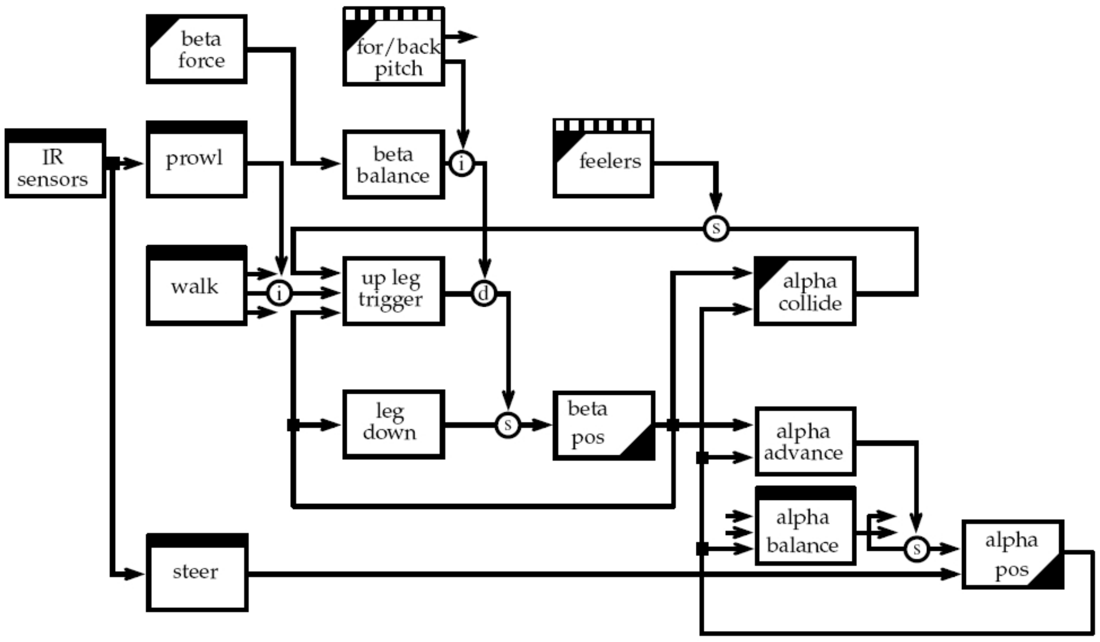


Figure 4-3: The walking algorithm for a six-legged robot, from [9].

when the leg is swinging freely, high current readings would send out false positives. The beta force AFSM discards these error signals, and the output of the beta force AFSM is not merely a noisy current signal, but a pre-filtered "obstacle detection" signal which can be more easily integrated into the rest of the controller. In this case, using the raw data may not have allowed the robot to properly utilize the signal, but providing a properly-filtered version of the signal is helpful.

But one important distinction separates the network controlling the walking robot from the flatworm: the use of centralized control. Brooks stresses the lack of a brain in his algorithm: "[r]obust walking behaviors can be produced by a distributed system with very limited central coordination," (page 11). And, even though a two-thousand neuron brain can safely be called "limited central coordination," it is clear that it plays an important role in coordinating the flatworm's ability to walk. These two approaches, fully distributed and partially distributed, seem to be in conflict.

4.3.2 Central control versus central coordination

One might argue that the flatworm brain plays the role of another layer on the subsumption architecture. The last module for the subsumption architecture ("steered prowling" on page 10) describes an algorithm for directing the robot's locomotion towards infrared activity; this module computes a "direction" of motion and writes that direction into a register controlling the backswing of each leg. A more complete layer of sophisticated modules such as this one might constitute a "brain" akin to the CNS found in the flatworm. This theory, though, fails to acknowledge that a decerebrated flatworm is not a fully-functional but less sophisticated flatworm. If the brain represented the "top layer of control," one might expect higher-level behaviors to disappear completely and lower-level behaviors to remain intact. Instead, the data reveals that behaviors at all levels of complexity are negatively impacted. The righting behavior, which one might imagine is the most basic (non-metabolic) behavior known to the flatworm, goes from a coordinated grasp-and-turn behavior to one that relies on chance contact with the substrate. The feeding behavior, though, goes from a complex ipsilateral turn with a brain to a simpler, but still extremely complex, food-stuffing behavior when decerebrated. The decerebrated feeding behavior, from purely a control perspective, is more complex than the righting behavior, and yet the removal of the brain impacts both. This makes the "layers of control" theory more questionable.

A more plausible theory combining these two ideas is that the flatworm brain is not essentially another set of modules in the subsumption architecture, but the network of connecting wires. Each module alone represents something less than a complete behavior; they each describe gestures that, when executed in the right circumstances, accomplish a broader goal. The "leg down" machine, for instance, constantly tries to command the leg to a down position. Keeping its legs down is not a complete sub-behavior of walking, but is it a *part* of walking that, when coordinated with other movements like lifting and swinging, causes the bearer of the leg to locomote. According to this line of thinking, making the proper connection with the output of the “leg down” machine is at least as important as the behavior of the machine itself. This implies that the proper connections between the elements of the subsumption architecture represent the “brain” of the controller.

This idea can be more clearly seen by rearranging figure 4-3. Instead of arranging the AFSMs in terms of “layers,” consider the layout shown in figure 4-4. No changes have been made to the connectivity of the network between this figure and figure 4-3. Instead, the AFSMs have been arranged according to their inputs and outputs. Because most AFSMs both accept input and provide output, they appear twice on the diagram. The “steer” AFSM, for example, takes input from the “IR sensors” AFSM, and thus appears on the right. However, it also supplies commands to the “alpha pos” network, and thus appears on the left. There is still only a single “steer” AFSM in the network, but it appears twice on the diagram (alternatively, imagine the diagram wrapped around a cylinder, with AFSMs on the right and left re-connected). There are some AFSMs that produce only sensory information, and they are grouped together in the upper left hand corner.

Interestingly, the AFSMs in this architecture each have only a single output (although Brooks notes that AFSMs can have multiple outputs), but many have several inputs. Also notable is that, while the network of connections is feed-forward, the resulting control architecture will not necessarily be so. In the particular case of this network, no module has its output connected directly to its input, although it is rife with other forms of feedback, such as the loop from the output of “beta pos” to the input of “leg down,” which has its output connected to the input of “beta pos.” Moreover, interaction with the sensors and motors on the robot itself are in parallel with this network; even if all the connections were broken, the robot would not be inert, but would instead display behavior that harks to the decerebrate flatworm, suggesting that the set of connections between the AFSMs are analogous to the flatworm’s brain.

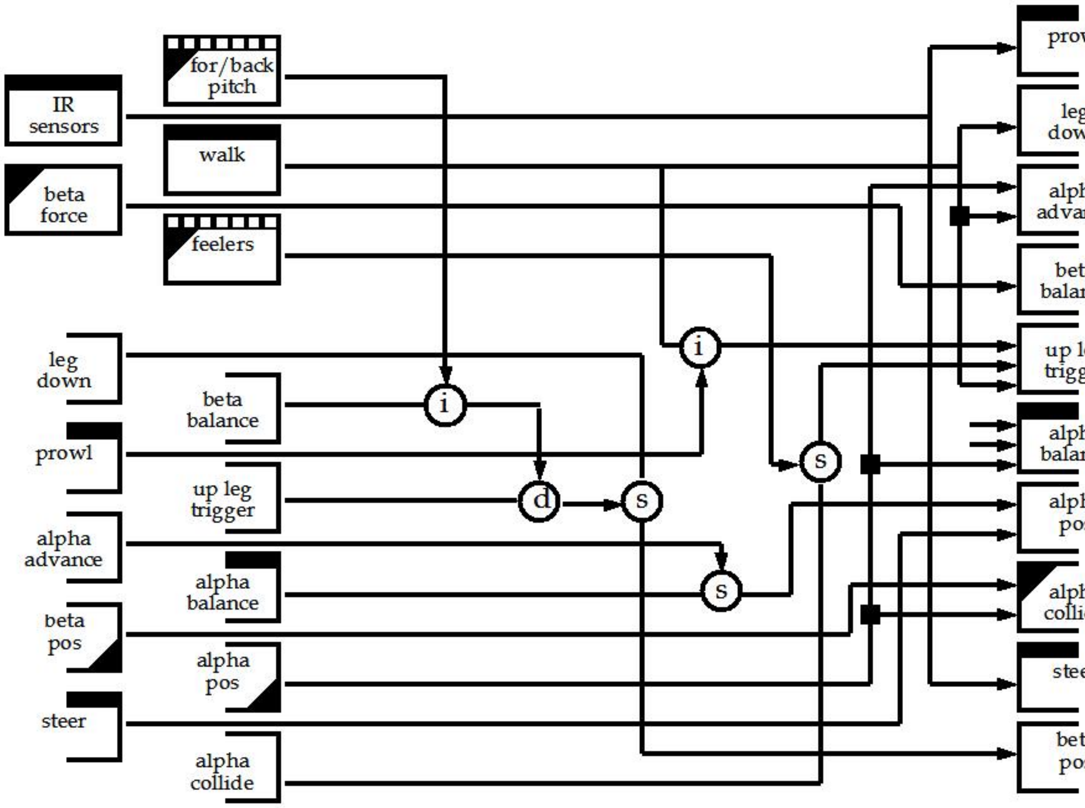


Figure 4-4: The same walking algorithm from figure 4-3, rearranged to put “inputs” on the left and “outputs” on the right.

4.4 Behavioral control on the Roomba

To test this idea of modeling a central controller as a series of connections between AFSMs, a series of AFSMs were designed for the Roomba (introduced in section 2.3), and connecting architectures were evolved. This section describes the hand-designed modules that were used, the method for evolving the connections and applying them to the system, and the results in both simulation and from the real robot.

The connecting structures in Brooks' walking algorithm were simply "wires" that passed eight-bit values between the AFSMs. In this implementation, this structure will be represented as a neural network to allow for comparability to standard neuro-evolutionary techniques. This change has three implications. The first is that the "wires" now have weights associated with them, meaning that the output signal from one AFSM will not necessarily be identical to the input signal received by the target AFSM. The second implication is that the signals will now be signed, real-valued numbers instead of eight-bit discrete values. The third implication is that signals will be not mediated with suppression/inhibition nodes, but will be summed with non-linear transfer functions (in this case, an arctangent transfer function is employed). This approach may have been problematic for the walking robot, as leg gestures tend to combine poorly. If one module wants the leg to go up to avoid an obstacle and another module wants the leg to go down to take a step, trying to combine these two commands- holding the leg straight out or twitching back and forth - fails to solve either problem and, more seriously, paralyzes the robot. With the Roomba, however, the only actuators are the two wheels that move the robot, and combining, say, a command to move forward with a command to turn left can be combined into a velocity vector somewhere between the two, allowing the robot to continue moving, collecting sensor data, and perhaps mediating the two sensor values. Of course, directly opposed values could freeze the robot in its tracks, but the likelihood of the sum of the motor commands being exactly zero for both wheels is small.

4.4.1 Justification for incorporating hand design

The hand-designed AFSMs for the Roomba experiment were created, in part by watching the hand-designed algorithm that the robot typically uses to clean floors. This approach raises an important issue: if one knows enough about a system to hand-design AFSMs, then it seems that that same designer should also know enough to hand-design the connections between the AFSMs. This point is not entirely without merit; designing and implementing distributed systems by hand is a valid way to build a robot controller. However, there are at least three good reasons to evolve, rather than hand-design, the connections between hand-designed elements for a robot.

The first reason is to improve robustness. An evolved controller has been validated by repeated testing. This validation process tests the controller under a variety of activation conditions, and as the number of hand-designed elements grows, testing the various combinations is a process that will need to be automated in some way. Using evolution to test and verify the particular connection pattern serves to combine the design and testing program into a single step.

The second reason to use evolution with hand-designed modules is to increase the likelihood of innovation. While the hand-designed AFSMs from the walking algorithm were clearly designed to be connected in a particular way (indeed, the AFSMs and connection pattern were created simultaneously), the proper coordination of behaviors is not always straightforward. When the Roomba collides with a wall, it can choose to try to follow that wall or it can choose to make a random turn and drive off. Determining the proper sensory state for when to choose one action over another is not straightforward, and the best way to design such a choice might simply be to try out different policies with a wide range of inputs in a wide variety of environments. An evolutionary approach provides this option.

The final argument in favor of evolution is to lower the requirement for human involvement. As robots become more prevalent and more complex, there will be fewer humans capable of properly designing controllers for them. Moreover, robots will be entering environments that may or may not have been anticipated by human designers. Evolution represents an automatic way for a robot to try different combinations of control strategies to optimize its behavior without requiring human intervention.

4.4.2 Roomba behavioral control layer

The hand-designed AFSMs for the Roomba are as follows:

Wheels. Two AFSMs control the wheels, one controlling the left and one controlling the right. These AFSMs drive the robot at 500 mm/s by default. If a value is written into the input register, that value, scaled by 500 mm/s, is commanded as the new speed for the robot. If the commanded speed would be higher than 500 mm/s, then the speed is capped at this maximum value, and if the value would be less than -500 mm/s, the speed is capped at this minimum value. The AFSM has a single output register that outputs the (possibly capped) speed of the robot, scaled to be the range of [-1.0, +1.0].

Backup on bump. The Roomba bump sensor, which covers the entire front half of the robot, is compressed when the robot collides with anything while traveling forward. The bump sensor provides two bits of information, a "left side" sensor and a "right side" sensor (despite the bumper itself being a single piece of plastic). This AFSM has no input register, but will command the wheels to drive at -500 mm/s for 0.15s if a bump is sensed. Two output registers reflect the state of the bump sensor, outputting 0.0 by default and outputting +1.0 if the bump sensor is currently depressed. A third output register outputs 0.0 by default, and outputs +1.0 for 0.3s starting from the *end* of the backup behavior, that is, this signal is a "done backing up" signal.

Follow wall. The Roomba has a side-mounted infrared sensor that allows the robot to detect a wall on its right side without actually bumping into it; this sensor returns a single bit indicating the presence or absence of a wall. The "follow wall" AFSM has a single input register that activates a wall following behavior when the value written to the register is 1.0. When the module is active, it has two states. A “find wall” state drives the robot to the right (a left wheel speed of 1.0 and a right wheel speed of 0.65) when no wall is sensed. If the robot is not near a wall, the robot will drive around in a circle. If, however, the robot is near a wall and the “find wall” state drives the robot such that the IR sensor is activated, then the robot will drive with a very slight leftward arc (left wheel speed of 0.95 and a right wheel speed of 1.0). If the robot is facing parallel to a wall, this will cause the robot to gently pull away from the wall, curving back to the right when the IR sensor is no longer active. The AFSM has two output registers, both defaulting to 0.0. If the AFSM is engaged in the wall following behavior, that is, if the input register is 1.0, then the first output register will output a value of 1.0. The second output register will output a value of 1.0 when the IR sensor itself is activated.

Distance and angle summation. The Roomba provides odometry figures that can be used to deduce the robot’s position. While the normal caveats for odometry-based navigation apply (slipping wheels will skew measurements), the Roomba wheels have sufficient friction with the floor that these numbers can be used productively. Both the “distance” and “angle” AFSMs take a single input argument that specifies a distance or angle. Once that distance or angle has been covered, the single output register toggles from 0.0 (the default) to 1.0 for a 1s period. The input values are scaled by 100mm, so an input value of 2.0 to the “distance” AFSM will cause the output register to read 1.0 (for 1s) every time the Roomba drives 200mm. At top speed, this would saturate the module, causing the output to remain high as long as the robot is moving forward.

Spiraling behavior. The final AFSM creates a “spiraling” behavior. The hand-designed Roomba algorithm drives the robot in circles of slowly-widening radius to allow the robot to thoroughly cover a circular area, and so this behavior has been included in the repertoire of available behaviors for the evolutionary algorithm. This AFSM has two input registers. The first is a register that establishes how long to execute the spiraling behavior. The input to this register is scaled by 60.0s, so an input value of 2.0 would indicated that the Roomba should spiral for two minutes. The input register is checked only when the Roomba is not spiraling, so holding the input register at a positive value will cause the robot to repeatedly engage in a spiraling behavior. The second input register is a "reset" register that deactivates any current spiraling behavior if its value is greater than 0.0. (If both registers are held high, the robot does not spiral.) When the spiraling behavior is activated, the right wheel speed is driven at the maximum forward speed and the left wheel speed is given by

$$V_l(t) = 1.0 - \left(\frac{t}{0.8 * t_{spiral}}\right)^2 * V_{max} \tag{4.1}$$

where t is the time from the beginning of the spiral and t_{spiral} is the total length of time to spiral.

4.4.3 Interface with neural network

The interface between a neural controller and the Roomba is straightforward: the network contains one input neuron for each sensor and one output neuron for each wheel. The network is updated by first applying the sensor values to the input nodes. The network is then "clocked" by transmitting the data along each link and updating the values at each non-input node in the network. The network must be "clocked" several times to ensure that the sensor values have fully propagated through the network. The output values are then read and and translated them into wheel speeds (this strategy is similar to the Braitenberg example from figure 1-1).

The addition of the behavioral layer means that the network now drives inputs and outputs to AFSMs, and the AFSMs are responsible for reading from the sensors and driving the motors. The method for updating these dual networks (summarized in 4.4.1 for robot R, subsumption architecture S, and neural network N) proceeds as follows: first, the sensor data is transmitted into each AFSM, and each AFSM is clocked. Then, the output registers from each AFSM are used to set corresponding input nodes in the neural network. For example, the bump sensor AFSM has three output registers and so will set three input neurons. Two neurons will be set according to the bump sensors on the robot (since the first two output registers simply hold that value), and the third neuron will be set according to the third output register, which contains the "done backing up" signal. Once each input neuron to the network is updated, the network is "clocked" several times to ensure stable output values. The output values of the neural network are then transmitted to the input registers on each AFSM. The spiral AFSM, for example, has two input registers and will thus read from two corresponding output nodes, one indicating the time to spiral and one indicating the reset state of the AFSM. The AFSMs are clocked once again, and the motor data from the AFSMs are then applied to the wheels of the robot.

Algorithm 4.4.1: NEURAL-BEHAVIORAL UPDATE(R, S, N)

```
for each sensor on $R$
  do $S \leftarrow$ sensor value
Clock $S$
for each Output register in $S$
  do Input node in $N \leftarrow$ output register in $S$
Clock $N$
for each Input register in $S$
  do Input register in $S \leftarrow$ output node in $N$
Clock $S$
for each Actuator in R
  do Actuator $\leftarrow S$
```

4.5 Results on simulated and physical Roomba

The impact of the use of a behavioral layer of control was measured by comparing the performance of evolved "pure network" controllers with the performance of evolved "behavioral-neural" controllers (the two approaches were described in section 4.4.3).

The choice of environment for the evolutions is a key factor. For the experiments below, a model of a conference room near the author's lab was used. The room is nearly eleven square meters, and is typically furnished with three chairs, a couch, and a small table in the corner. The "typical-ness" of this room is up for debate, and evolving a controller for more general solutions would require testing on a variety of rooms to ensure that the controller generalizes. However, this room has several advantages: it is small enough that it can be reasonably cleaned in a short period of time, and yet it contains enough spatial complexity that cleaning it completely is a non-trivial problem.

In an effort to keep the target task simple enough to demonstrate clear results, the fitness function is merely the average total area (measured in square meters) cleaned in five three-minute trials with a randomized starting position. Other elements were added to or removed from the fitness function in other tests. An initial set of tests included the energy spent by the Roomba during the cleaning cycle, however, the coverage costs of stalling the motors (by colliding with an object and not backing up) made this element of the fitness function unnecessary. Much longer runs were also tried: both ten-minute periods and thirty-minute periods were used for fitness trials. However, the longer periods made measuring the fitness much more difficult. Because only single-pass coverage was measured, longer cleaning periods made the problem much easier to solve. Observations from the evolved controllers made it clear that a popular solution is to simply turn left upon contact with an obstacle; as long as the robot is not perfectly lined up with one of the walls, the random effects of collisions with obstacles in the room is enough to cover the entire room. As a result, more sophisticated behaviors failed to emerge from the evolutions because simple controllers could gather the maximum fitness by simply running for a very long time. This could be solved with a more complex fitness function that considers multiple-pass cleaning and assigns a value to subsequent cleaning passes, however, a deeper understanding of cleaning would be necessary to make these choices wisely.

4.5.1 Results from simulation

With the choice of a three-minute, five-trial fitness function in the small conference room environment, the behavioral network controllers outperformed the purely networked controllers by a significant margin (data displayed in figure 4-5). Two observations indicate the nature of the results. First, the average cleaning ability of the very best network is generally higher for the behavioral controller than the pure networked controller in nearly every generation, and the performance gap is widest at the end of the fifty-generation run. Second, the variations in best performance from generation to generation are much smaller with the behavioral network controllers. While the reasons for this are unclear, observations from some of the controllers in the middle of the run suggest that the pure network controllers generalize much more poorly and depend on a favorable random starting position for their fitness. For example, starting out in the middle of the room is a tremendous benefit to a network controller that turns the robot 180 degrees upon collision with an obstacle. The behavioral controllers, though, are typically able to use either the wall following behavior, the spiraling behavior, or a combination of the distance/angle sensor to exit the robot from small areas (such as the area between the chairs in the lower right-hand corner of the room).

Figure 4-6 illustrates the networks resulting from the random seed demonstrated in figure 4-5 and figure 4-6 illustrates four examples of the coverage patterns generated by the pure network controllers (the top row of illustrations) and the behavioral network controllers (the bottom row of illustrations). Looking at the coverage patterns, it is clear that the biggest challenge for the network controllers is to clean the middle of the conference room. This network controller in particular found that entering the upper-right-hand corner of the conference room resulted in a looping behavior that passed several times over the area between the couch and the upper-right-hand corner. The behavior networks, on the other hand, illustrated the ability to both enter and exit the small spaces between the chairs, a behavior that is clearly advantageous to thoroughly cleaning the room. The experimental results were verified on runs

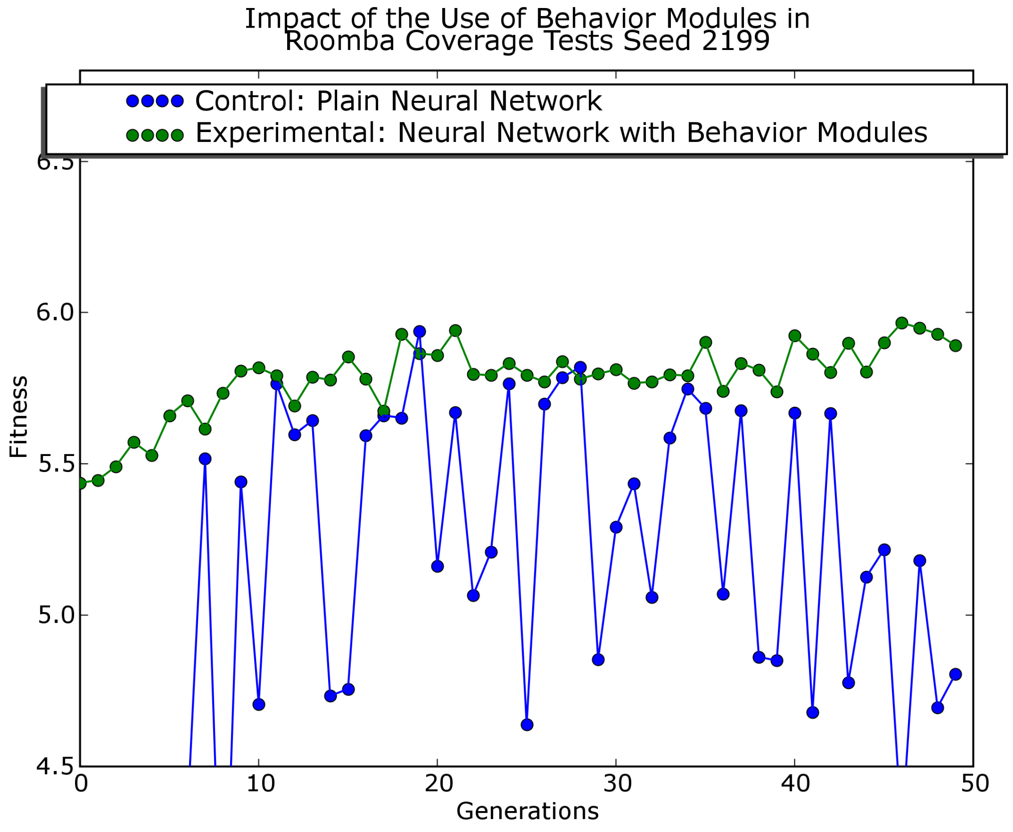


Figure 4-5: The fitness for the pure neural network controllers (blue line) was impacted by the random starting position much more than the behavior-based network controllers (green line), and almost always performed significantly worse.

generated by two other random seeds. The results are illustrated in figure 4-8.

4.5.2 Results from the physical robot

While the results on the real robot were lower than simulation, the coverage was still significant, and the behavioral controller outperformed the network controller (figure 4-9). The plain network controller from seed 2199 averaged 4.7 m^2 cleaned over its five runs in simulation, but averaged 4.13 m^2 during the real test. The behavioral network controller averaged 5.8 m^2 cleaned in simulation, but averaged only 4.81 m^2 meters cleaned on the real robot. One source of error is the measurement of the fitness in the real world. A camera with some basic motion tracking software sums the largest connected number of pixels where motion has occurred. Of course, this measurement is imprecise and motion can sometimes go undetected (although visually, this does not appear to be a large source of error). The fitness measure can also be compromised if the Roomba is occluded by a tall piece of furniture; the chair in the upper right-hand corner does obscure some of the floor surface area. But, even with those caveats, there is a clear difference in the performance of the simulated robot and the performance of the real robot. In particular, when the robot spins in place, the angle swept is different in simulation from what is observed on the real robot (this is almost certainly due to wheel slip on the robot that is not modeled). Using a system such as the one outlined in chapter 5 could also help improve the performance of these controllers on the real robot.

The results of the behavioral-neural controller on the physical robot are difficult to compare to the hand-designed algorithm (coverage patterns from the hand-designed algorithm are displayed in figure 4-10). The data are promising: the hand-designed algorithm averages 4.79 m^2 while the behavioral-neural controller averages 4.81 m^2. However, it must be noted that this particular test is not entirely fair to the hand-designed algorithm. Most notably, the evolved controller has been optimized on the size and configuration of the small room used in the evaluations. No attempts were made to generalize the cleaning algorithm, while such concerns were clearly a significant part of the hand-designed algorithm. Moreover, a deep understanding

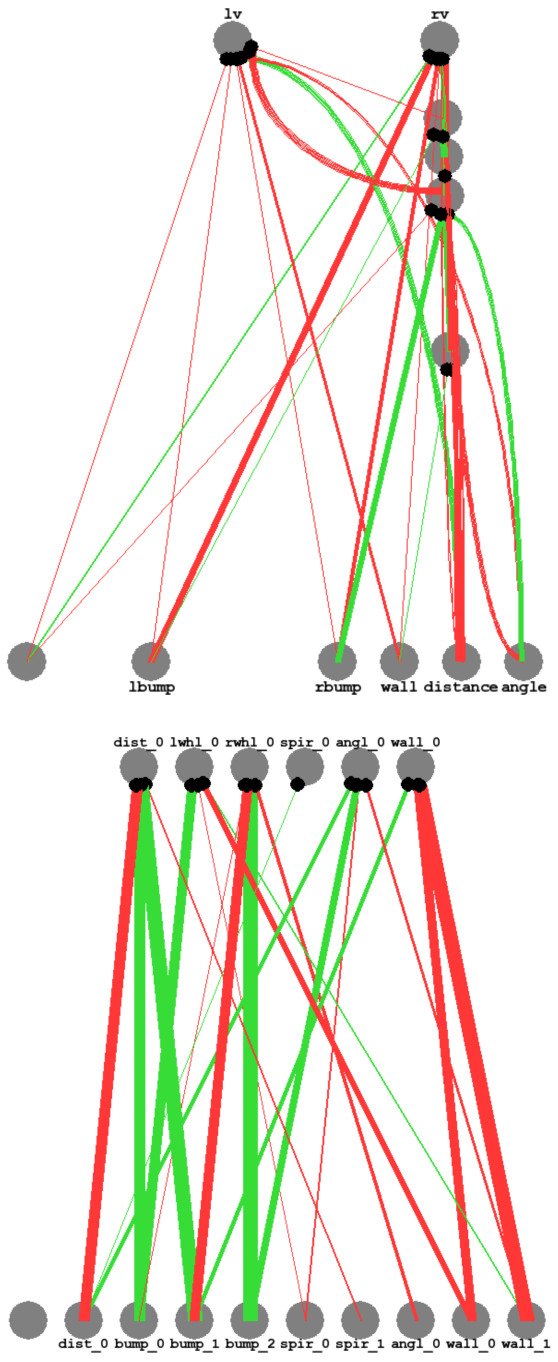


Figure 4-6: The networks for the control (top) and experimental (bottom) networks that produced the results in figure 4-7.

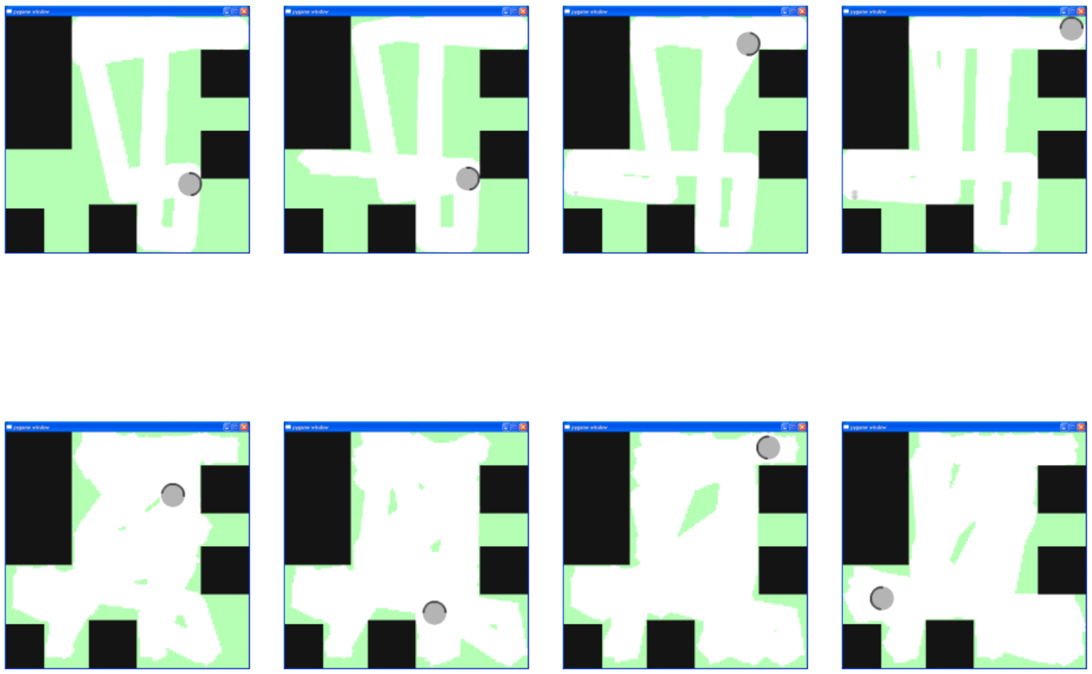
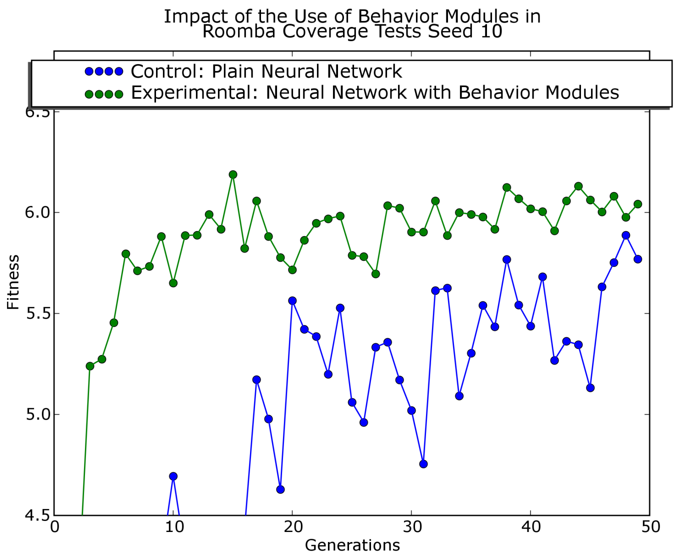


Figure 4-7: The results from plain neural network cleaning cycles (top four images) and from neural network with behavior modules (bottom four images).



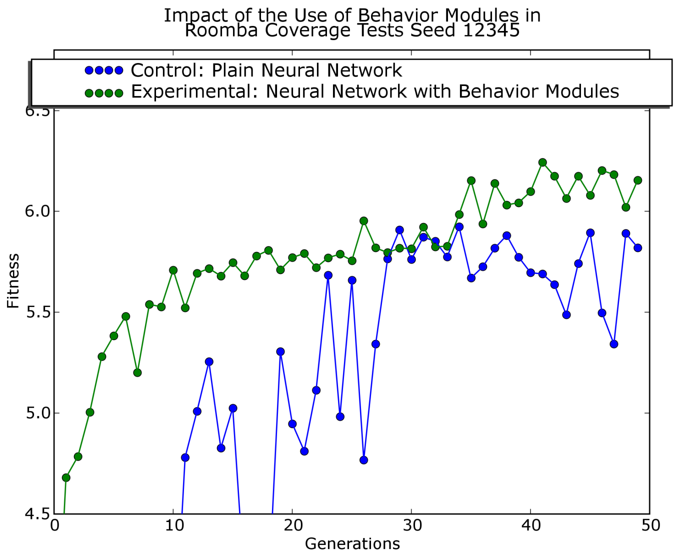


Figure 4-8: Results from two additional evolutionary runs reinforce the notion that the behavioral controllers outperform the network controllers, both in total coverage and in generation-to-generation consistency.

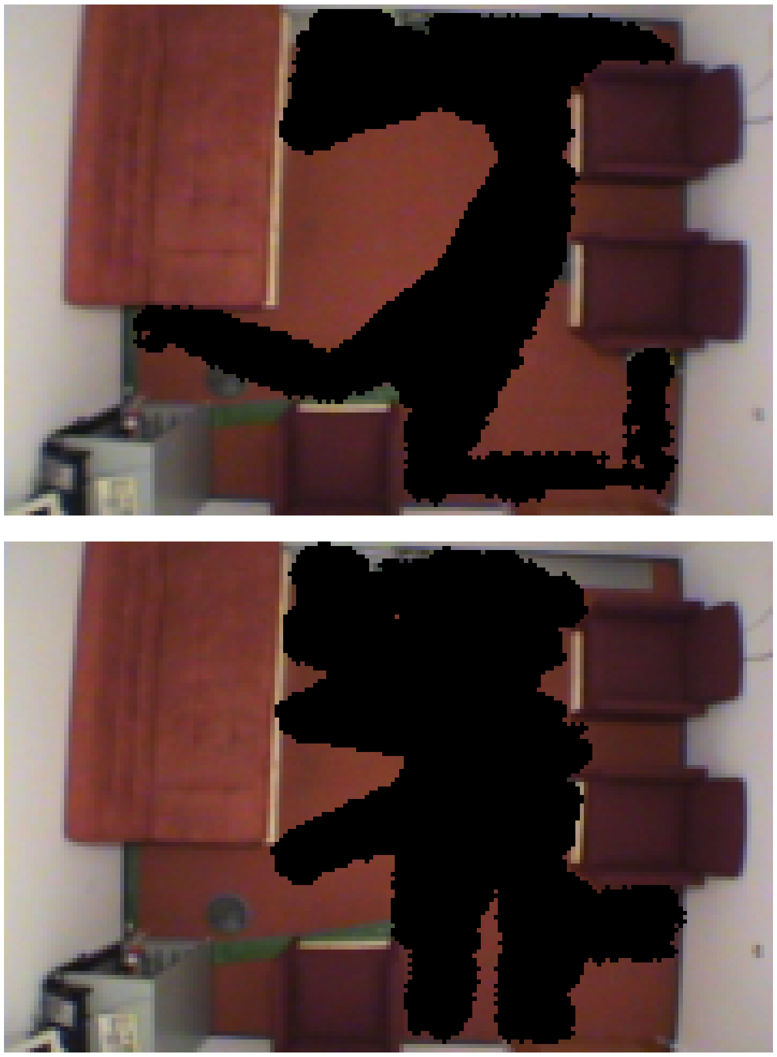


Figure 4-9: While the performance on the real Roomba was lower than the performance in simulation, the plain network controller (top) was still outperformed by the behavioral controller (bottom).

of the cleaning behavior of the robot was not included in the evolutionary fitness function. How many passes over the floor contribute to its being cleaned? What speed results in optimum cleaning? How does turning while cleaning impact the effectiveness of the side brush? Further, there may be philosophical choices made by the hand-designed algorithm that conflict with the chosen fitness function. Is it, for example, more important to clean quickly or completely? While the results from this test do not lead to the conclusion that the coverage problem can be effectively solved by the behavioral-neural evolutionary algorithm, they do suggest that the performance of a hand-designed algorithm can be approached using evolutionary methods.

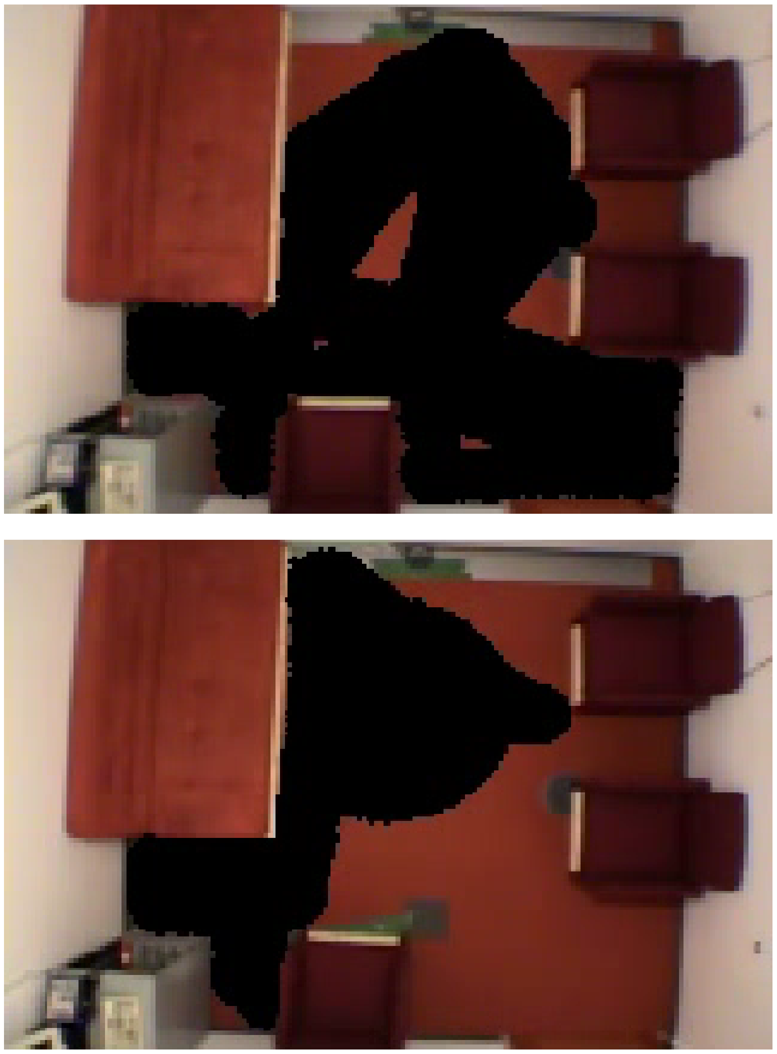


Figure 4-10: Two examples of the performance of the hand-designed Roomba coverage algorithm (compare with figure 4-9, bottom).

Chapter 5

Reality Feedback

Using a simulated robot to understand a real robot is a risky proposition. The advantages of using simulation are clear. Simulations allow a user to run experiments faster than real time, meaning that applications involving extensive testing on a robot (such as evolution) can be performed without needing to wait for days, years, or millennia to pass. A simulated robot in a simulated environment allows an experimenter complete control and observation over his experiment; the full state of every element in an environment is completely known at each instance. Simulations are cheap to build (at least, the cost for the materials are cheap as compared to real robots), do not wear out or break, and are easy to change or discard. Moreover, simulations circumvent many of the hassles presented by real robots: complex mechanical design, delicate wiring, and constant power consumption, to say nothing of broken hardware, flaky sensors, and unreliable actuators. Indeed, the advantages of simulated robots over real ones would be complete were it not for the plain fact that, as robots grow in complexity, the behavior generated by a controller in simulation will bear little relationship to the behavior generated by that same controller on a real robot.

This chapter will describe and demonstrate a method for using simulation to evolve controllers for physical, dynamic robots even in the presence of imperfect simulations. The following section will describe other approaches evolving controllers for real robots using simulation. One particular piece of related work, Matarić and Cliff's 1996 survey "Challenges in Evolving Controllers for Real Robots," will be reexamined in light of the last ten years of changes in evolutionary robotics. The body of work surrounding evolution for the Khepera will be critiqued. Section 5.2 will introduce the concept of "feedforward" evolution and argue that physical feedback is necessary to ensure that evolutionary results will transfer to physical robots. The following section will more closely examine the idea of the "reality gap" [36] and quantify this phenomena on the *micro.adam* simulation and physical robot. Section 5.4 will describe a method for incorporating feedback from the physical robot into a NEAT evolution, and provide data demonstrating that such a method dramatically improves the evolved performance of evolved neural networks on *micro.adam*.

5.1 Related Work

Gallagher, Beer, et al. provided one of the first examples of a controller evolved in simulation and transferred to a robot [20]. Using both a simulated and a real hexapod robot, walking gaits were evolved to allow the robot to locomote over a smooth surface. Many of the problems observed when transitioning from a simulated to real environment are addressed, but qualitatively

Jakobi suggests that evolved controllers will be most effective if the simulation used for evolution is as simple as possible. He terms this approach "minimal simulation," and demonstrates some success in evolving controllers in simulation for a variety of robotic systems [34]. Aside from experiments on the Khepera, Jakobi demonstrates results on an actuated vision platform, and, most interestingly, an octopod walking robot. Sadly, the difference between the performance on the simulated and real octopods is not quantified.

Martin used genetic programming to evolve a perceptual system for a mobile vision platform [44]. Instead of a simulation, Martin used real data, recorded from the robot and marked by hand, to allow the robot to evolve off-line. This solution neatly sidesteps the problem of building an accurate simulation, and it avoids introducing simulation artifacts into the real world. The results when applied to the real robot bear out the strength of this solution. However, not all evolutionary systems will be able to take advantage of the ability to pre-record enough data for the effective use of simulation, although simple systems might benefit from a smaller-scale use of the ideas in this work.

Taking a similar philosophical approach to Martin, Stilman et al. [63] use virtual models of the environment along with real sensory data to create a hybrid world in which a humanoid robot can maneuver and, potentially, interact. While this application is still preliminary, it seems to hold some promise; by constantly updating the model with data from the real world, this system has the potential to both improve the fidelity of the model and better understand the areas where strategies from the model perform poorly.

Harvey, et al., introduces a novel approach to the issue of evolving on a real robot in the Sussex Gantry Robot [25]. The Gantry Robot uses a downward-pointing camera directly above a small mirror to reproduce the effect of having a camera traveling around a real environment below the downward-pointing camera. The camera-mirror system is on a gantry, allowing the whole system to move in two dimensions to capture real data. Like Martin's system, the Gantry Robot focuses on processing camera inputs with evolutionary techniques. In particular, [37] describes a set of experiments wherein the robot learns to distinguish differently-shaped targets (triangle from rectangle) pasted to the walls of its environment.

Eggenberger, in [28], uses genetic regulatory mechanisms to evolve a 20 x 20 neural "retina" for an active vision platform. Using camera pixels as input, a neural network is evaluated on its ability to center in the visual field a single point of interest against a static background. The genetic system is particularly detailed, using a complex regulatory network and a three-dimensional spatial system for connecting the inputs to the outputs. When the results were transferred from simulation to reality, it appears that the nature of the behavior was preserved, although it is unclear whether the smooth transfer was due to a particularly accurate simulation, a simple robotic system, or an intrinsic property of the evolutionary system.

Work done at the Fraunhofer Institute is both interesting and relevant to the work done in this dissertation. Wischman, et al., [72] [29] describe a method for evolving controllers for (a different instance of) Popp's *micro.eva*. Their method focuses on decentralizing the control; that is, instead of viewing the robot as a single entity, they treat each of the five arms as individual robots that happen to be connected via the ring. They use evolution to evolve five separate neural networks, and then record the fitness as the maximum angular velocity of the robot. Their results in simulation are impressive, and, perhaps even more impressive, the controllers are still effective, albeit in a reduced manner, on the real robot. The main difference between this work and the work presented in this dissertation is the assumptions at the start. The Wischmann work assumes enough knowledge of the system to break the controller down into exactly five components. While this is an obvious choice for *micro.eva*, other robots might not present such an obvious choice of subdivision. Moreover, Wischmann builds further knowledge of the system into the components; the hall-effect sensors (denoting ring position) are encoded in each component relative to the arm represented. That is, instead of each "ring position" input labeling hall-effect sensor #1 in the same way, arm 1 will view ring sensor #1 as zero, arm 2 will view ring sensor #2 as zero, and so forth. Building in *a priori* knowledge of the robot into the sensor encodings will make further application of this method more difficult.

More recently, Boeing, et al., have applied genetic algorithms to the control of a walking biped named Andy [4]. They appear to have an excellent simulation and a robot platform that, by utilizing a gantry mechanism, would lend itself well to the methods described in this chapter. The evolved controllers do not perform well on the real robot, but, while considering future work, Boeing concludes:

> The servo model could be improved if the internal parameters of the servos PID controller were fully known. This could be achieved by additional in-depth testing of the servo motor. Alternatively, the robot hardware could be redesigned to operate with DC motors with separately designed PID controllers.

5.1.1 Revisiting Matarić and Cliff

Of particular note is the survey of problems evolving controllers for real robots by Matarić and Cliff [46]. Despite the age of the paper (written in 1996), their roundup of the options available to aspiring evolutionary roboticists still provide a decent starting point for understanding the field today. While the ensuing ten years has seen some ideas actively pursued (evolvable hardware, evolution using real vision), other ideas less actively pursued (evolution with shaping), and new important ideas emerge (evolved morphologies in simulation that can later be built in reality), the options for evolutionary robotics are similar today to what they were then.

Of particular interest to this dissertation is their discussion of the use of evolution on simulated / real robotic pairs (sections 3.1 and 3.2, pages 17 - 19). They identify five particular challenges with this approach, and a review of their points is warranted.

Real time on real hardware. The temporal issues of performing evolution on robots will always be a concern. Evolution depends on many, many evaluations for its power, and, barring a truly miraculous breakthrough, robots operate at the same timescale as the researchers who work on them. However, simulated artificial evolutionary experiments are also time-consuming, and yet are carried out frequently. If it is acceptable for a simulation to run for days or weeks, then surely a similar running time is acceptable for a real system.

Another promising alternative is to employ parallelism in the robot tests. Simply using ten robots instead of just one can reduce the evaluation time by an order of magnitude. This could make the difference between a reasonable and an unreasonable testing duration. Of course, this approach is not appropriate for every robotic system. Ten copies of a robot that is expensive or unique is obviously problematic, although such a robot is already an unlikely candidate for long, repeated testing (see the **robot lifetime** section below). Moreover, using many robots implies that the many environments are available to the robots as well.

Battery lifetime. While this is absolutely a challenge for an untethered autonomous robot, a system with constant access to power (such as *micro.adam* and *micro.eva*) can avoid the need for local power storage. But even untethered autonomous robots can participate in evolution if access to a permanent power source is available. The Roomba is capable of autonomously finding its way back to a charging dock, and other robots that can acquire their own power could very well be on the horizon. While serious, this challenge is surmountable.

Robot lifetime. Simulated robots are impervious to damage from repeated use, but for real robots, this issue represents a serious barrier to the use of robots in an evolutionary experiment. The work described in this dissertation involved the extended use of the *micro.adam* and *micro.eva* robots, and each robot experienced a motor failure after several weeks of heavy use (see figure 5-1). The limiting part, in the case of these robots, was the RC servo motors. While the wear on the motors was likely exacerbated by near-constant motion (much of it oscillatory in nature), one week of use is probably sufficient for some shorter evolutionary runs. An obvious, but non-trivial, solution to this problem is to emphasize robustness in both the design and parts selection of robots intended for evolutionary purposes.

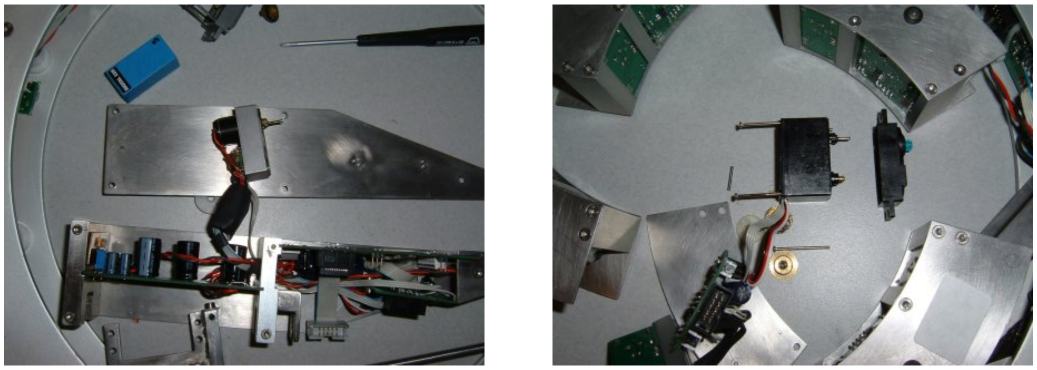


Figure 5-1: Both *micro.adam* (left) and *micro.eva* (right) suffered damage to their RC servos after roughly one week of testing. This kind of "robot lifetime" issue was anticipated by Matarić and Cliff in [46].

Noise and error models. In [36], Jakobi, et al. suggest that, in addition to properly modeling the robot and environment, a simulation must also contain an accurate model of sensor noise as well. They evolved controllers to perform obstacle avoidance on a small mobile robot (the Khepera) in simulation with three different levels of noise: no noise, a model of noise from empiric data, and "double noise," which used the empiric model but doubled the amplitude of the noise. Perhaps unsurprisingly, they found that the controllers from the simulation with the true empiric noise model transferred to the real robot most effectively. While the selection of a noise model is an interesting issue itself, it can also be viewed as an example of the many aspects of the real world that would need to be included in a perfect simulation (see section 5.2).

Generality v. usefulness of simulations. The previous challenge, stated more generally - incorporating enough detail in a simulation to successfully transfer to the real robot - works in opposition to the goal of generality. As a simulation amasses more and more detail, it becomes less and less useful when applied to any robot other than the one being simulated. One could imagine a general simulator for all two-wheeled mobile robots. However, the work on noise models by Jakobi suggests that an effective simulation must include more detail about not only the robot, but the type and amount of noise that the robot experiences in sensing the world. This suggests that the noise in actuating - that is, moving around - could also be significant. If it is the case that a simulation must be highly detailed with respect to its target robot, then the idea of a general two-wheeled simulation is unrealistic. More concerning would be if this requirement were even stronger: the specter of requiring each simulation to conform to the details of an individual robot, that is, a simulation for Roomba A that does not apply to Roomba B.

Looking at these concerns ten years after they were written, two things are clear: little progress has been made in solving the problems that are intrinsic to the use of simulation, but a great deal of progress has been made on physical robots. Power consumption issues have feasible solutions, replacement parts are cheap and available, and computational power has increased to the point that large-scale evolutions involving complex simulations can be performed at reasonable time scales.

5.2 Problems with “feedforward” simulation use

Most evolutionary robotics systems use simulation as proxy for the physical robot; the evolutionary system evolves a simulation and then applies it to the physical robot. This difference between a simulated system and its real counterpart has been described as the “reality gap” in [36]. While a surprising number of publications simply ignore the reality gap label control of a simulated system “robotic” control, [73] [53], others [8] [18] have discussed this problem in some detail.

Curiously, though, this problem has received little direct attention. Instead of measuring the error between simulation and reality and making adjustments to the evolutionary process to decrease them, the errors are simply regarded as an inevitable byproduct of the evolutionary process. In fact, most approaches that address this problem seem to focus on improving the simulation in hopes of matching the physical robot perfectly instead of acknowledging the existence of the error and accounting for it. In this way, most evolutionary robotics projects are “feedforward”; the error signal is disregarded. This section argues that the feedforward method has serious inherent limitations.

5.2.1 The perfect simulation does not exist

One fact should, at this point, be clear: no simulation, no matter how detailed, will ever fully capture the behavior of a real robotic system. Building a simulation is very difficult, and the process is fraught with pitfalls. Aside from the human errors that are all-too-common in any software system, modeling a robot inevitably involves choices about what to include and what to leave out. As [36] points out, choosing the correct type and level of sensor noise can have a serious impact on performance. But choosing this noise model is actually a series of choices. If the model is going to be based on empiric data, how many observations should be made before building a model, and under what conditions should the sensors be observed? What model should be used to generalize those observations? Or should the sensors themselves be more carefully analyzed in hopes of expanding the theoretical framework of the simulation?

And even if these questions about the robot can be satisfactorily answered, the same questions must be asked about the simulation of the robot's environment. Should a single, real environment be carefully observed and reconstructed in simulation? How does one choose an environmental model that will anticipate all the possible scenarios in which the robot will eventually find itself? Will parts of the environment provide inaccurate data to the sensors (e.g., reflections of sonar) or will the actuator performance in the world be altered by environmental factors (e.g., a slippery floor)? What practical contingencies should be included? Should motors occasionally break or sensors stop returning data? Should non-uniformities between theoretically identical sensors be modeled? Should the slow wear of the treads of tires on mobile robots be included?

Perhaps more troubling than these questions themselves is the notion that, as robots become more complex, these questions become both more important and more difficult to answer. For example, the simulation of *micro.adam* used for the experiments described in this dissertation included a very simple model of the friction between the ring and the castors supporting it. However, as the robot ran for an extended period, dirt and grime would build up on the castors (likely a result of the metal strips, used for delivering power, rubbing against the outer rails on the robot). The buildup of this substance was extremely slow; observations from one hour to the next revealed no difference to the naked eye. And yet, after a long period of operation, the buildup would be enough to cause a radical difference in the robot's behavior. With a clean set of castors, a hand algorithm for randomizing the starting position of the ring before a test would typically "get stuck" between 10 and 20 times during the first hour of testing (these "got stuck" events were recorded in a log). After eight to ten hours of operation, the robot would frequently "get stuck" multiple times during a single randomization attempt! Should either the high-friction or low-friction state simply be modeled and maintained on the robot? Should the buildup of grime on the castors over time be modeled? Should the simulation increase the ring friction parameter over the course of an evolutionary experiment?

If the imperfect correspondence between simulated and real systems is acknowledged, then incorporating feedback from tests of the real robot is the only way to ensure that evolved controllers will successfully cross the reality gap. Consider a set of controllers, C (figure 5-2). Further consider the set of controllers that successfully achieve a given task on a simulated robot in a simulated environment, $S \subset C$, and the set of controllers that achieve the same task on a real robot in the real world, $R \subset C$. If the simulated robot/environment pair and real robot/environment pair have perfect correspondence, then $S = R$. If, on the other hand, the simulation is an imperfect model of the world (as this section suggests must be the case), then S and R will not be identical. If the simulation bears any resemblance to reality (and if the set of controllers is reasonable), then one would expect that $S \cap R \neq \phi$, that is, that some controllers in C will work both in simulation and in reality. It is these controllers, controllers in $S \cap R$, that a simulated/real evolutionary system can search and exploit.

5.2.2 Trying to build the perfect simulation is a poor use of resources

Another argument for the use of real robots in evolutionary algorithms involves the utilization of the resource of the system designer. If artificial evolution is to be a truly useful tool, then the amount of time invested in constructing the evolutionary system must be less than the amount of time it would take to construct controllers of the same efficacy by hand. (For the purposes of this argument, we will assume that there are no controllers that are discoverable only by evolutionary methods). Using the terminology from the previous section, the system designer who wishes to use an evolutionary technique must be able to find the controllers that are in $S \cap R$ in less time than it would take to find a controller in R by some other method.

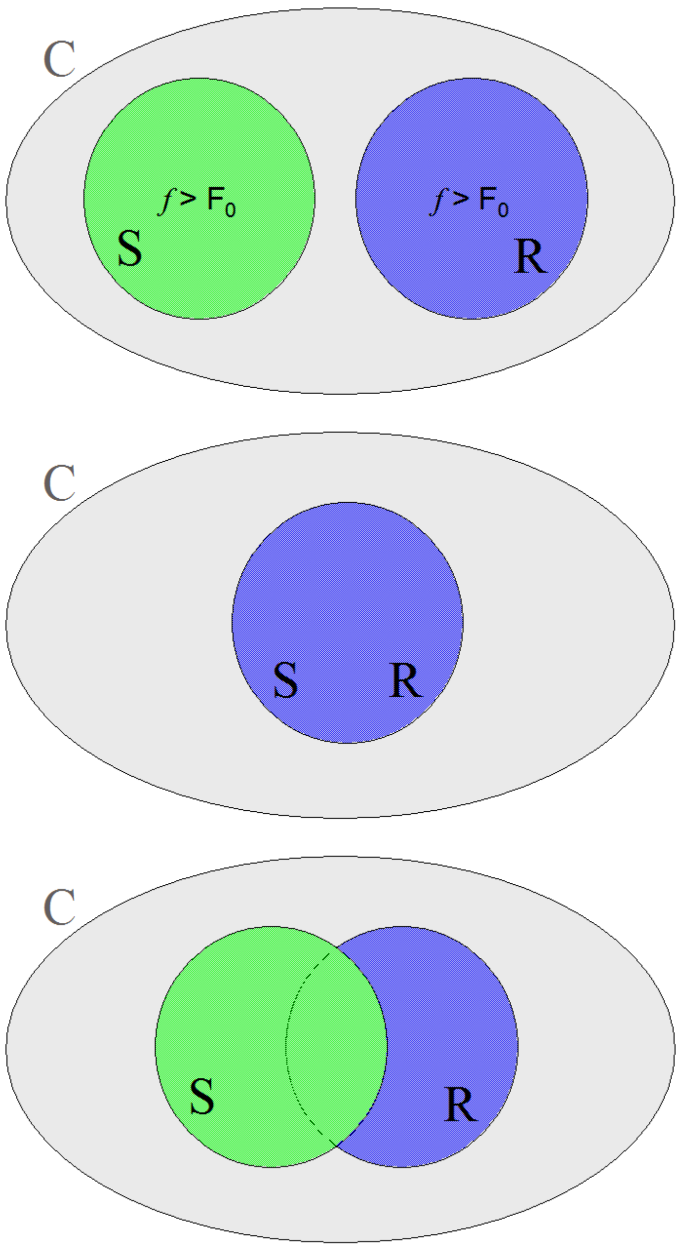


Figure 5-2: Three Venn diagrams representing the space of all neural controllers, C, controllers that achieve some fitness F_0 on a simulated robot, S, and controllers that achieve fitness F_0 on a physical robot, R. If a simulated robot and the physical robot are not related, then the controllers that achieve F_0 in simulation and the controllers that achieve F_0 on the robot will be completely different (top diagram). The assumption made by "feedforward" evolution is that the simulation is a perfect representation of the robot, and that any controller that does well in simulation will also do well on the robot (middle diagram). The case with most pairs of simulations and physical robots, especially in the case of dynamic robots, is that some controllers will work in both simulation and on the real robot, but some will work only in simulation and others will work only on the real robot (bottom diagram).

One approach to finding the controllers in $S \cap R$ is to build the best simulation possible, that is, try to enforce $S = R$, and then sample controllers from S and hope that they are in R. The previous argument asserts that this path will yield limited results, however, for this argument, we will assume that investing time and effort into a simulation can increase the number of controllers in $S \cap R$. Investing this effort will involve working with the real robot; taking measurements of the sensors in various environments, characterizing the noise, investigating the impact of actuator commands in various circumstances. As the designer acquires increasingly detailed knowledge of the robot, two countervailing forces reduce the value of this investment. First, the simulation becomes highly specific to the robotic system in question, resulting in controllers that are less likely to generalize to other robots. Second, the time invested into the validation and perfection of the simulation increases the requirements for the synthesized controller; that is, time invested into the simulation raises the bar on the type of controller that could have been designed by hand.

Another approach, advocated by Jakobi in [34], is to construct S such that it is as small and accurate as is reasonably possible in hopes that it will only include controllers that are also in R (Jakobi terms his approach "minimal simulation"). In [35], Jakobi outlines the process of creating a minimal evolution, including the identification of a "base set," defined as "real-world features and processes that are relevant to the performance of the behavior [of the robot]." Divining the elements of this "base set," it seems, requires a process that is, at least superficially, similar to the one required for building a precise simulation. Measurements of the robot must be taken and corresponding models built. But, in this case, these empiric models must then be evaluated for membership in the "base set," perhaps by building two simulations: one with the model and one without. It seems fair to say that the investment of resources in this process will be sizeable, and while the "minimal simulation" approach may result in evolved controllers that demonstrate better generalization, the investment of time, it seems, will rival the time invested in building the (mythical) perfect simulation.

5.2.3 Testing on a real robot is the only way to validate a simulation

A central problem that is completely ignored by each of the previous approaches is the problem of identifying and eliminating solutions that are in $S \cap \bar{R}$. If the reality gap is accepted as an inevitable hazard of working with simulation, then a necessary step in evolving controllers for a robot is filtering out controllers that fail to cross it. Most systems that evolve controllers in simulation simply plug the evolved controller into the physical robot with the understanding that some controllers won't work. This process makes the loop very long: the system designer learns about the robot, builds a simulation, evolves controllers for that simulation, applies the evolved controller to the robot, and then, presumably, learns something new about the robot and begins again. Formally closing this loop by providing the feedback from real tests on the robot to the evolutionary algorithm creates two major advantages.

First, automatically acquiring feedback from the real robot during the evolutionary run lowers the investment by the system designer. The tests are carried out automatically on the robot, meaning the system designer has saved time in three ways. First, the tedious and fraught process of perfecting the simulation is now unnecessary. The simulation only has to be "good enough" such that some of the controllers produced by the evolutionary machinery work on the real robot. Second, evolution in simulation is more effective, as time and resources are not be wasted searching among controllers that will not work on the real robot. Third, the controllers that the system produces at the end have been validated on the real robot, meaning that the system designer does not have to sort through a series of controllers that may or may not work.

The second major advantage of testing the real robot on the target task is that the process of matching the simulation and real robot is focused on only the relevant task at hand. In the "minimal simulation" method, the system designer must understand the task well enough to pick out the key behaviors, sensors, and environmental elements to create a successful simulation. Carrying out the fitness tests on the real robot means that only the elements that are relevant to the fitness task are fed back into the system. If, for example, a sensor is not helpful in solving a given task, then time spent measuring and modeling that sensor in either of the previous approaches would be totally wasted. In this approach, the sensor will be ignored if it is not useful *or* if its simulation has such poor correspondence with reality that its use will not transfer to reality.

5.2.4 The curse of the Khepera

It is impossible to discuss the literature for this field without first dealing with the dominant robotic platform: the Khepera [38]. The Khepera is a small mobile robot that is pervasive in the evolutionary robotics community [36] [48] [52] [25] [35] [60] [18] [47] [11] [31] [17] [19]. The size of a very short coffee mug, the reasons for Khepera's ubiquity are readily apparent. Several simulations for the robot have already been written, and the simplicity of the robot means that these simulations are both fast and accurate. Moreover, widespread use means that the parameters for these simulations have been thoroughly tested and matched to the robot. Because the Khepera is very small, the impact of dynamics on its motion is minor, so the actuation error is kept to a minimum. Finally, the limited size and mobility of the robot means that the Khepera is restricted to artificial, hand-designed, desktop environments.

The simplicity of the Khepera system has made evaluation of simulation-to-reality systems very difficult. Because the Khepera makes little to no use of dynamics, or indeed any properties that cannot be accurately simulated, it is difficult to tell if a successful transition from simulation to reality is due to an innovative method or a very simple target system. Jakobi's work on characterizing noise in [36] is emblematic. The Khepera has both ambient light sensors and IR sensors, and light sources in the (desktop) environment triggered both sensors. While Jakobi attempted to characterize the nature of this interference, the simulation was not able to fully capture the complex phenomena, leading him to conclude the following:

"Difficulties in simulating interference between the IR and ambient light

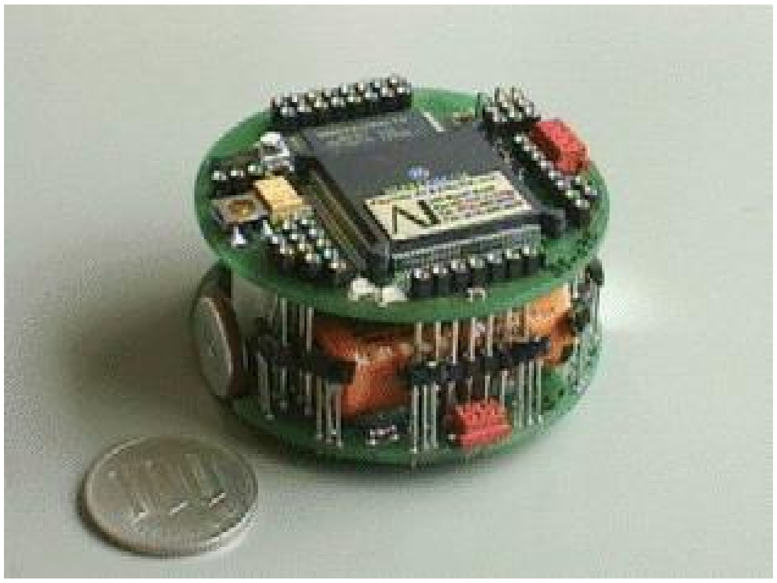


Figure 5-3: The Khepera is small enough, and constrained to such simple environments, that "reality gap" has been reduced to trivial proportions.

> modes of the Khepera's sensors suggest that the approach taken here rather quickly become less feasible as the interaction dynamics become more complex."

This dissertation concludes that the Khepera's extensive use is owed largely to the plain fact that the reality gap for this robot is very narrow. Whether intentional or not, the Khepera seems to be the product of a desire to build a robot that can be easily and successfully simulated. Is it intentional irony that the name "Khepera" derives from the Egyptian word *kheprer*, meaning *to become*? Because it seems that Khepera has proven not that robots can be successfully simulated, but instead that a simulation can be successfully instantiated. Floreano, et al., even go so far as to suggest in [18] that miniaturization is critical for any successful application of simulation in evolutionary robotics, a view this dissertation emphatically rejects. If evolutionary robotics can only be applied to systems that are easily and completely simulated, then the field has little application now and vanishing prospects for the future.

5.3 The reality gap on *micro.adam*

In order to observe the impact of feedback from a real robotic system, it is important to carefully describe and quantify the effectiveness of the feedforward method. Most evolutionary robotic systems, including and especially those using the Khepera, do not quantify either the fitness score of a hand-designed algorithm or directly compare the quantitative fitness differences between simulation and reality. This section aims to accomplish both goals.

The *micro.adam* robot was described in detail in section 2.2. The target task for the tests described in this chapter is to create a maximal amount of angular movement in a ten second period, starting from a random position. The fitness score for a single trial is defined as

$$F = \left(\sum_{t}^{T} |\theta_t - \theta_{t-1}| \right)^2 \tag{5.1}$$

where T is the time of trial (10.0s in this chapter), θ is the angle of the passive degree of freedom on *micro.adam*. The fitness is averaged over five trials, each with a randomized starting position.

While this task could be more generally described as "turn as much as possible in ten seconds," there are alternative behaviors. Displacement in either direction contributes equally towards the fitness score, so rocking back and forth is a valid way to achieve fitness (although the short time period suggests that rotating in a single direction for the entire period is likely to result in higher fitness). This difference is further exaggerated by the fitness monitoring process on the real robot: while the precise angular displacement of the ring was used in simulation, this data was not available on the real robot. Instead, a controller was credited with 0.20 turns for each change in the ring sensor value. In other words, the real robot's fitness was measured in one-fifth turns. A "rocking back and forth" strategy on the real robot would have to have an amplitude of at least 0.20 turns in order to change the ring sensor's value. While a more accurate measure using visual feedback from a camera might have improved the system's performance, such a system was not used for the results demonstrated in this chapter.

5.3.1 Results from a hand-designed controller

A hand-designed controller was designed and evaluated for *micro.adam*. The robot provides "motor direction" data that indicates which direction the arm is being pulled. The basic controller, then, is trivial: if the arm moves to its furthest position in the opposite direction that it is being pulled, the robot usually achieves movement. When, for example the arm is horizontal with the hinge on the viewer's right, it swings up and to the right. This causes most of the weight to be in the upper right-hand corner of the ring, which, when it falls, causes the ring to rotate clockwise. After slightly less than a half-turn, the arm is effectively "upside down," and the motor direction bit flips. This causes the arm to swing up again, and the momentum from the previous falling causes the ring to turn over again, and the cycle repeats. Data from 100 ten-second trials is displayed in figure 5-4.

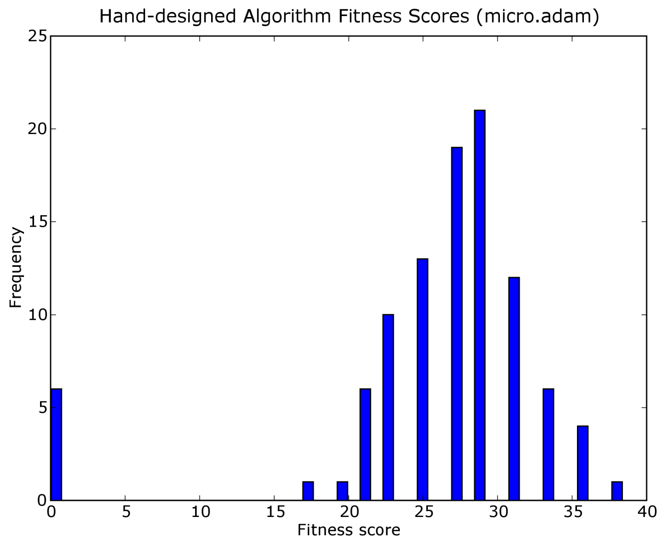


Figure 5-4: The hand-designed algorithm typically scored 26.0632 (a little over five full turns), but the "stuck" behavior resulted in several very low scores.

This hand algorithm typically scores a fitness of a little over twenty-five, meaning the ring typically achieves a little over a half-turn a second. Most of the variance was caused by the random starting position of the ring; some orientations allow the ring to build momentum more quickly than others. There is one additional observed behavior from this controller, though. If the starting position is at one of two specific starting angles, the arm will swing up, but the center of mass of the arm will be directly over the ring's center of gravity, and the ring will remain stationary. While this unstable configuration is not typical, the friction between the ring and the castors is high enough that this behavior was observed six times in the 100 trials. It is possible to re-design the hand algorithm to release from this position - a random thrashing behavior is enough to find a new orientation and allow the algorithm to work again - but this redesign must wait several seconds in case the ring is merely off to a slow start, and, in a ten-second trial, performance is not significantly improved. A precise tweaking to determine the optimal amount of time to wait before initiating the thrashing behavior was also not executed.

5.3.2 Results from feedforward evolution

Feedforward evolutionary experiments were carried out wherein neural network controllers were evolved using the NEAT algorithm and the same standard parameter set used for XOR experiments. The input neurons were as follows: five input neurons corresponding to the five ring position sensors. These neurons output 1.0 when the ring position sensor reading was equal to that neuron's index and 0.0 otherwise. The gyroscope was encoded as two antagonistic inputs; one input output 1.0 when the ring was rotating at the maximum observed velocity (roughly 1 turn per second) in a clockwise direction and -1.0 when the ring was rotating at the same rate in the opposite direction, and a second output neuron that output the same data with a reversed sign. A single neuron output the arm's position, scaled from -1.0 to 1.0, and a single neuron output the arm's motor current, scaled from 0.0 to 1.0 (1.0 indicated the maximum observed motor current reading). All hidden and output neurons used an arctangent transfer function, and the output neuron corresponding to the arm interpolated the output from the range [-1.3, +1.3] into the arm positions (any output outside that range was considered saturated and did not impact the arm's position).

Several runs were performed with a population of 150 for 250 generations, resulting in 37,500 fitness evaluations in simulation (recall that a single fitness test is five 10-second trials with randomized starting positions). The winner would then be tested on the robot for an identical fitness test. The results were surprisingly poor: all fitness was the result of a high-energy starting orientation simply falling to a lower-energy orientation. No full revolutions were observed, and most controllers would move the arm to one of the extremes and perhaps oscillate very quickly back and forth. Understanding the reasons for this requires a closer examination of one of the assumptions made by many evolutionary robotics systems.

Most systems that evolve controllers in simulation for an extended period, testing only the winner on the real robot, assume that the best performer in simulation will be the best performer in reality. A simple test can validate or refute this hypothesis: for a standard evolutionary run, test the winner after each generation on the real robot. If the assumption is true, then one would expect the real performance to monotonically increase, or, perhaps, track with the simulated performance. Running this experiment for only 100 generations (with the standard population of 150) on *micro.adam* falsifies this hypothesis, as figure 5-5 demonstrates.

The performance in reality is generally much worse than that of simulation. This is understandable, as simulating the dynamics of *micro.adam* is a difficult and inexact process. In particular, modeling the interactions between the robot and the castors is very difficult. The simulation of *micro.adam* assumes that friction is uniform over the orientation of the wheel. On the real robot, this assumption is clearly wrong: the robot is not perfectly round, and, in fact, the front and back rails are not identical. These imperfections are largely invisible at slow speeds, but when the robot builds angular momentum, the small bulges of the outer rail cause the robot to pitch forward and backward, and this rocking behavior prevents the robot from building up the high speeds that would be necessary to achieve the high speeds observed in simulation.

However, failing to achieve the performance observed in the simulation would be merely an artifact if the controllers were still effective on the real robot. But the more troubling trend in figure 5-5 is that the performance on the real robot varies wildly with the performance in reality: some of the early solutions seem to transfer

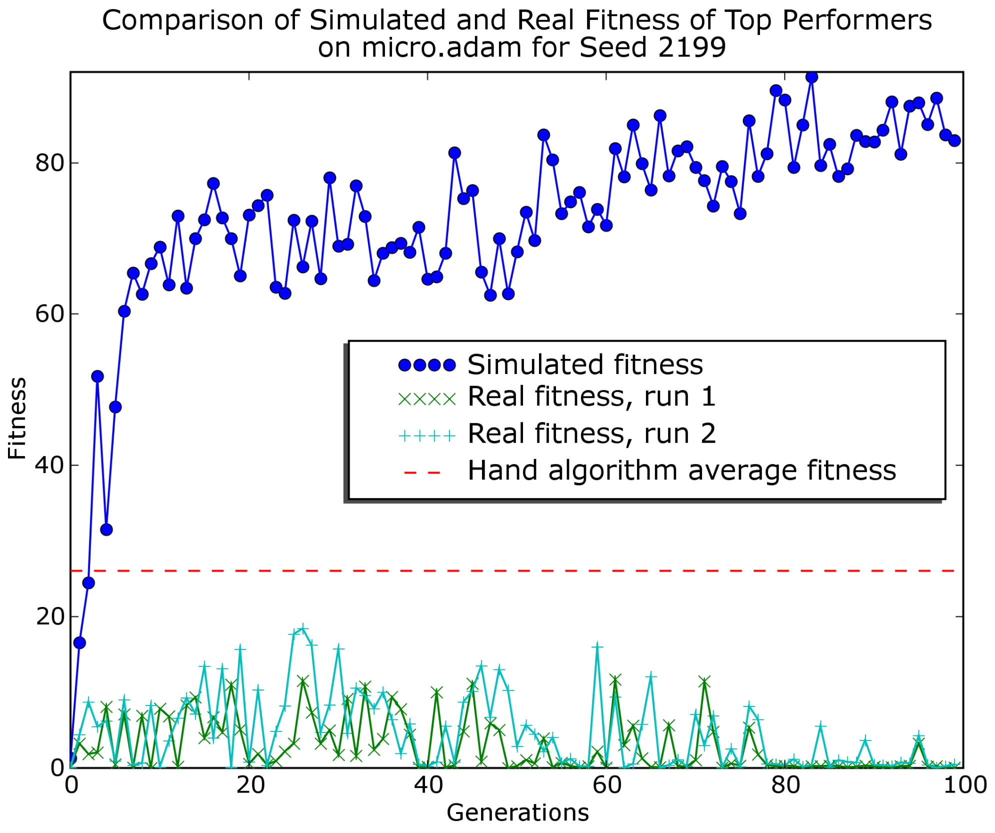


Figure 5-5: The “reality gap” between the simulation of *micro.adam* and the real robot.

effectively to the real robot whereas others do not. While many of the solutions appear to transfer effectively to reality, they are found and then lost, as generations with high real fitness will be followed by generations with very low, sometimes zero, fitness. This pattern matches the hypothesis suggested earlier, namely, that the set of controllers that perform well in simulation and the set of controllers that perform well on the real robot are overlapping, but not identical, sets. During the early generations, the evolutionary algorithm is finding solutions with higher and higher fitness (in other words, controllers in S), but, due to a lack of feedback, will only choose a controller that performs well on the robot (a controller in both S and R) at random.

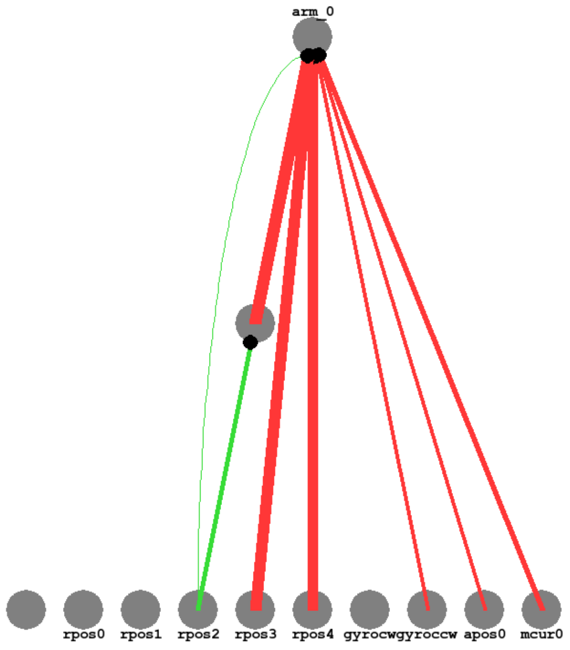
Even more troubling is the trend that seems to indicate that the solutions that are effective on the real robot become less frequent as the evolution carries on. After generation 80, no two consecutive generations achieve a fitness corresponding to more than a turn, and no solution averaged more than a few turns on the real robot, *despite the simulated fitness continuing to improve.* It's clear that these highly evolved solutions are particular to the simulation

5.3.3 Further analysis on two networks

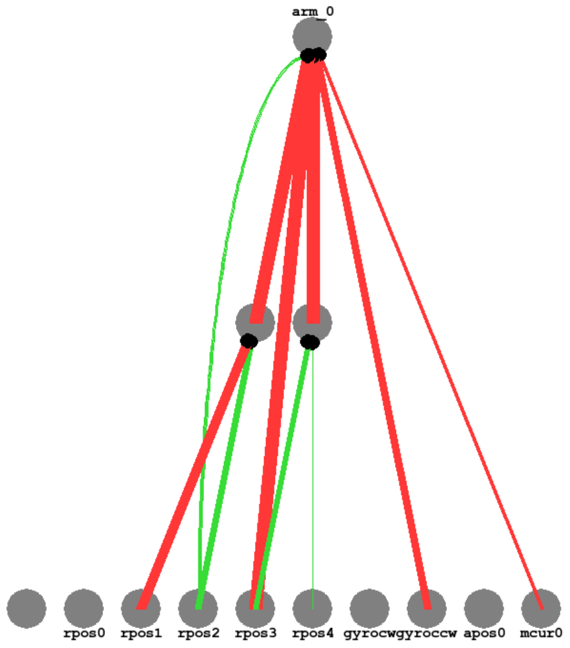
A closer look at some of the key networks from these evolutions demonstrates how these phenomena come about (see figure 5-6). Of particular interest are the networks from generations 22 and 26 of the run in figure 5-5. The fitness function used in this test is the squared number of turns in ten seconds, averaged over five tests with randomized starting positions. As the graph illustrates, the network from generation 22 (herein after "network 22") performed extremely well in simulation (a fitness of 75.7, indicating an average of 8.7 turns per test), and yet performed extremely poorly in both real tests, averaging less than a half turn per test. The network from generation 26 (herein after "network 26"), however, actually had a lower fitness in simulation (66.27, indicating just over eight full turns per test), but a much higher score on the real robot, averaging 18.4 in real one trial (4.24 turns per test) and 11.55 in the other real trial (3.39 turns per test).

Looking at the networks themselves, they appear to have some similarities. The two structural differences are a link in network 22 from the “arm position” signal to the output that isn’t present in network 26 and a link in network 26 from the “ring position 1” sensor to the output that isn’t present in network 22. This second link is of particular interest. The intuitive interpretation of its role would be to push the arm “up” when the ring position sensor reads “1,” thereby facilitating turning in the counter-clockwise direction. In network 22, this link is not present, and yet, from most orientations, the robot manages to continue rotating without the aid of that link. However, due to errors in either the friction or inertial model, the real robot does not have enough momentum to continue rotating, and falls back to a static position with ring sensor “1” at the bottom. This link makes very little difference to the simulated fitness of the robot, and yet makes a tremendous difference on the real robot.

One might argue that this is evidence that the simulation needs to be fixed; if some behavior in reality does not match the simulated behavior, then the parameters of the simulation (or perhaps the overall model of the simulation) should be adjusted to reflect the true behavior of the robot. However, identifying and fixing the source of the error in the simulation is a daunting task. Many sources of error exist between the simulation and the real robot. The ring is modeled as a uniform hoop, but the placement of sensors and computational hardware means that its mass is not truly uniform, so a more accurate mass model is one possibility. Similarly, the mass of the arm is not uniformly distributed, so the arm could be modeled with greater detail as well. No model of sensor noise has been employed, and related work suggests that carefully modeling the noise could improve performance (although it will surely be more difficult to characterize the noise on a complex, dynamic robot than it would be for the Khepera). The motor actuating the arm and the linkage that connects the actuator to the ring both have some significant backlash, so modeling these mechanical errors could help. Other subtle sources of error, such as the aforementioned non-roundness in the rings or the grime on the castors, are also candidates for improving the model. However, this process would consume significant intellectual resources,



(Network from generation 22)



(Network from generation 26)

Figure 5-6: Two winning networks from a simulated run on *micro.adam*. Red links indicate a negative weight and green links indicated a positive weight. The thickness of the link indicates the magnitude of the weight, with thicker lines indicating larger weights. The network from generation 22 performed well in simulation, but very poorly on the real robot. The network from generation 26 performed well in simulation (though not as well as the network from generation 22), but performed very well on the real robot.

increase the complexity of the model and computational power needed to simulate, and would ultimately offer no guarantee for success. The alternative - automatically testing the controllers on the real robot - is presented in the next section.

5.4 Evolution with feedback from the physical robot

This section describes a novel approach for finding controllers that perform well in both simulation and reality, that is, controllers that are in $S \cap R$. This approach is effective in finding controllers in this set by automatically testing controllers on the real robot and using these real scores as the basis for further evolution. Put another way, the use of artificial evolution with physical feedback finds controllers that work both in simulation and on the real robot (such as network 26) while discarding solutions that work in simulation without working on the real robot (such as network 22).

5.4.1 System description

The system incorporates feedback by performing a series of short evolutionary runs, testing the highest-performing networks on the physical robot, and then performing the next series of evolutionary runs with the offspring of the networks that performed best on the evolution (an illustration of the difference between feedforward evolution and evolution with feedback is given in figures 5-7 and 5-8). The short evolutionary runs generate networks that may perform well on the robot based on their performance in simulation; this can be thought of as taking a random sample of the networks in S. The tests on the physical robot, then, determine which of those samples are in $S \cap R$, and therefore suitable for propagation. The final step, continuing evolution, can be thought of as generating more networks in S, but with a bias towards networks that are topographically close to networks that are in $S \cap R$. This process can be summarized as follows for simulation R_{sim} and robot R_{phys} (assume a standard population size of 150):

Algorithm 5.4.1: PHYSICAL FEEDBACK IN NEAT(R_{sim}, R_{phys})

```
$P = \emptyset$
while $generation < N_{iterations}$
    do
        for each $seed \in seed_1, seed_2, ...seed_n$
            do
                Create species for initial population $P$
                Evolve for 3 generations using $R_{sim}$
                Test best individual $W_{seed}$ on $R_{phys}$
                Create new population $P = W_{seed_1}, W_{seed_2}, ...W_{seed_n}$
                $generation \leftarrow generation + 1$
```

Three elements from this system require closer examination. First, the system for generating candidate networks in S, second, the system for testing these networks on the physical robot, and third, the system for continuing the evolution with the physically-tested networks. The following three subsections will discuss these systems in turn.

5.4.2 Generating candidate networks in S

The comparison between simulated fitness and real fitness from figure 5-5 indicates that the real fitness of the highest-scoring individual in simulation after any given generation will be random: some winners will perform well and others will not. For these experiments, the arbitrary choice was made to select the winner after three generation with a standard population size of 150. But, to create a population of winners that were of roughly equal complexity, a set of networks was constructed by performing ten evolutionary runs with identical starting populations (i.e., a population of empty networks to start), but are then evolved using different random seeds. The different random seeds mean that different mutations are applied to the networks, although the simulation itself is deterministic. The winner is taken from each of the runs and put into a group of candidates. Note that the candidates started from the same place and evolved for the same length, but differ only by the effects of a different random seed.

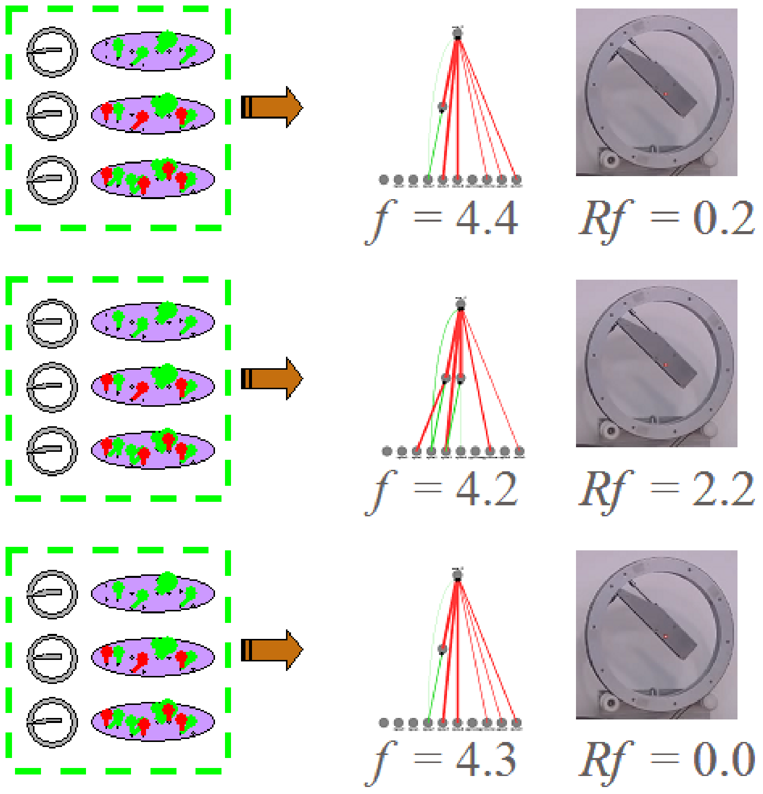


Figure 5-7: The typical method for evolving controllers on real robots involves performing several long evolutionary runs and applying the best performer from simulation to the real robot. Differences between the simulated and physical robots are disregarded, and the system user must hope that one of the simulation-based controllers will perform well on the physical robot (comare with figure 5-8).

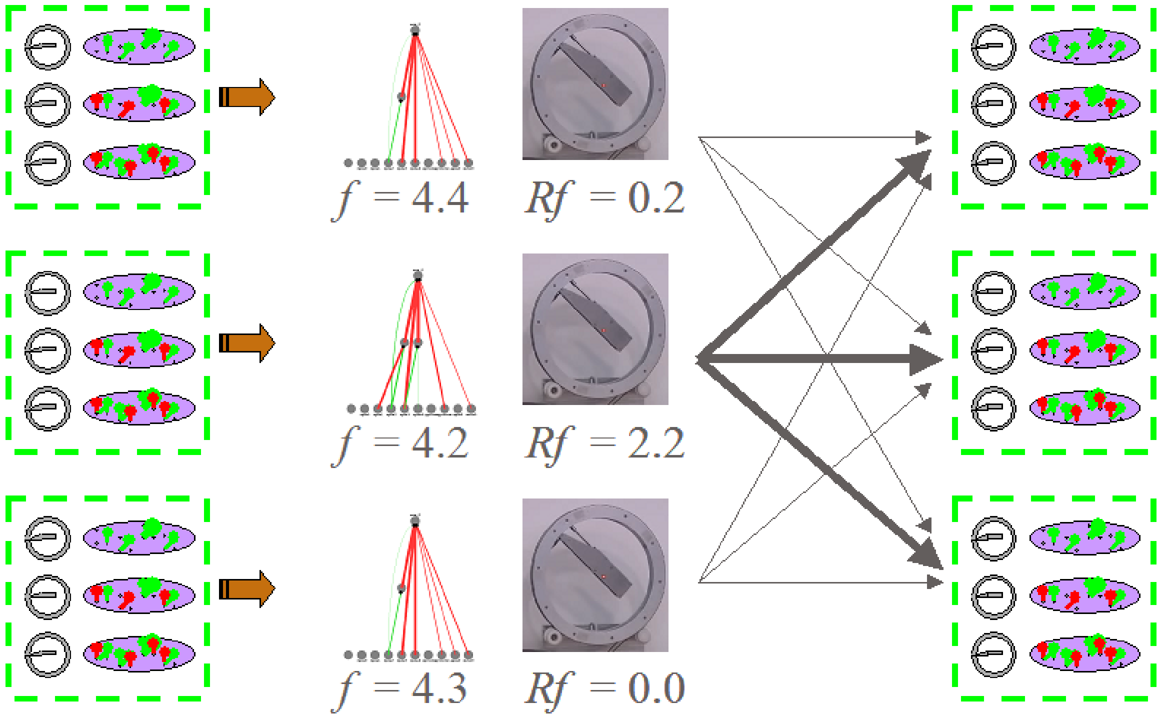


Figure 5-8: Incorporating feedback from the physical robot allows evolution to search among controllers that perform well on both the simulated and physical robots. The feedback is incorporated into the evolutionary algorithm by performing a series of short runs, taking the winner from each run (termed "candidates"), testing the candidates on the physical robot, and then creating new initial populations for the next iteration of short runs by allocating offspring to the best performers on the physical robot (compare with figure 5-7).

The process of selecting the controller evolved in simulation that is most likely to perform well in reality is an open question (despite a great deal of research). This method would be strengthened if a reliable method for doing so were found. Simply choosing the winner after a very short evolutionary period ensures that very few changes are made before they can be validated on the physical robot. In essence, this strategy assumes that any extensive testing in simulation will have a negative effect on the performance in reality. For *micro.adam* this is not necessarily the case; figure 5-5 indicates that some of the best performing solutions could be found after between fifteen and twenty generations of simulated testing. However, trial and error with this process found that number to be too high for this method.

5.4.3 Testing networks on the physical robot

This is the element that gave pause to Matarić and Cliff, but has been enabled by cheap, dependable robotics. For the Popp robotic system, constructing a real version of the simulated fitness test was probably less daunting than other systems might be. For any system to be suitable for physical feedback testing, it must meet three criteria.

Tolerant to faults. It is unreasonable to expect that any real robotic system can run for hours on end without communication errors, physical or electrical malfunction, or even "rest periods" to cool down motors or recharge batteries. Therefore, the robotic fitness testing setup must be capable of dealing gracefully with a wide variety of errors and delays. In the case of *micro.adam*, this implies that every communication with the robot should check to see if the bluetooth connection has been lost, and, if it has, a new connection must be established and any fitness tests that were interrupted must be discarded and restarted. The communication protocol between the evolutionary program and the fitness testing computational structure must also be capable of surviving a complete reboot of the testing station without discarding the data from the evolutionary run. Two strategies were tried to make the evolutionary algorithm robust: one that would assign zero fitness to networks that were unable to be tested due to errors in the base station, and a second mode that simply paused the entire evolution until the testing station regained function. The second mode proved to be more useful, as continuing an evolution without physical testing would effectively ruin any results gathered up to that point.

Initial state can be automatically randomized. For *micro.adam*, this means that the hand-designed algorithm was executed for a random period of time, the arm returned to the middle position (which allows the ring to rotate freely), and then the system would pause for a fixed period of time while any kinetic energy built up during the randomization process is dissipated. The randomization procedure also required a method for detecting the "stuck" configuration mentioned earlier, and performing an "unsticking" action would cause the randomization algorithm to reset entirely.

Another opportunity for improving the system lies in the performance of the randomization algorithm. Because the randomization is presumably a hand design, and because mechanisms must be in place to measure the fitness (see the following paragraph), the fitness score of the randomization algorithm can provide a useful benchmark for changes in the system over time. For instance, the buildup of dirt on the castors of the *micro.adam* system would degrade the fitness of networks over time,[1] but would also degrade the performance of the randomizing elements. A slow-but-steady rise in the execution of the "get unstuck" behavior tracked with the buildup of dirt, and, if run long enough, the system could become hopelessly stuck. While this was not implemented for the experiments described in this chapter, monitoring of the performance of the randomizing algorithm could help alleviate some of the concerns about controlling the experiments for robot wear.

Fitness automatically measured in real time. As mentioned earlier, the fitness is the square of the number of turns. In the case of *micro.adam*, the fitness can actually be partially recovered from the sensor values themselves. Changes to the ring position sensor were taken as an indication that the ring was moving. The fitness values were

[1]Unless a diligent graduate student cleaned the system every few hours.

scaled to be on identical scales - squared number of turns over five trials of a ten-second testing period - although using the same fitness units in both simulation and on the real robot are not strictly necessary, as we shall see in the next section.

5.4.4 Continuing evolution with the physically-tested networks

Once each candidate network has been tested and assigned a fitness score from the real robot, they are then used as the basis for a new series of evolutionary runs. The NEAT system is particularly well suited for this task. Recall from section 2.4 that offspring are assigned to a species (according to the average population of the species), not to an individual. The candidate networks, then, must be grouped into species before the offspring can be assigned. This is a critical process; if there are too few species, then the highest-performing networks are not selected with enough preference, and the crossover operators are less effective. If, on the other hand, there are too many species, very few crossovers are performed, and the subsequent generation is insufficiently innovative.

Normally, NEAT evolutions use a process called *dynamic speciation* to ensure that the total number of species is kept at a target level (see section 2.4.2 for the details of dynamic speciation). Since candidates who have been tested on the physical robot are a single generation, typical dynamic speciation cannot be used. Instead, a single-generation form of dynamic speciation was implemented, and proceeds as follows with candidate genomes G, maximum number of species N_{max} and minimum number of species N_{min}:

```
Algorithm 5.4.2: INSTANT DYNAMIC SPECIATION($G, N_{max}, N_{min}$)

$S = \emptyset$
while $|S| < N_{min}$ or $|S| > N_{max}$
    do
        if $|S| < N_{min}$
            then $T \leftarrow T + increment$
        if $|S| > N_{max}$
            then $T \leftarrow T - increment$
        Speciate $G$ into $S$ with threshold $T$
```

Once a NEAT evolutionary run has been initialized with a set of properly-speciated networks, it can proceed normally. Note that the offspring are assigned based on the performance of the species on the physical robot, not the simulated one.

Two technical notes are worthy of mention. First, because the results of the real fitness test from candidate N do not impact the evolution of candidate $N + 1$, the evolution of networks in simulation can happen simultaneously with the physical test on the robot. While the robot is idle during the generation of the first candidate and the evolutionary algorithm must wait for the results of the last physical test on the robot before creating the next population, during the majority of the time, the system is both evolving a candidate network in simulation and testing a candidate network on the physical robot simultaneously. This prevents the use of the real robot from slowing the overall testing cycle down to unacceptable speeds. Moreover, the system can actually monitor the time taken for both the simulated runs and for the real tests and adjust the population size parameters in an attempt to maximize the time usage of both resources. This "dynamic population sizing" strategy was implemented and tested, but not used in the experiments described in this chapter (in part to allow for comparison between the no-feedback and feedback cases on the basis of total number of simulated evaluations).

Second, it is important to point out that, by testing on a single robot, the evolution is likely to be biased towards the idiosyncrasies of that particular robot. For example, mistakes in the alignment of *micro.adam*'s arm's "center" position can alter the performance of the robot during real fitness testing. If the “center” position of the arm does not correspond to the true center of mass of the robot, then the ring cannot effectively randomize; instead, it will fall to one of two lowest-potential-energy positions (the arm below the true center of mass and the hinge on either the right or the left). In these cases, the real fitness of the networks will not correspond to their ability to cause the robot to rotate when starting from a random position, but their ability to cause the robot to rotate when starting from one of the two lowest-energy positions. In some cases, this effect may be desirable: if the robot target is one-of-a-kind, then discovering and optimizing around these perhaps unrealized (and probably not simulated) qualities will help bias the evolutionary process towards controllers that exploit them. If, on the other hand, the target robot is merely examples of a broader class of robots, then some method for ensuring generalization to the class of robots must be used. In particular, having a cluster of robots on which the testing can be performed would both speed up the evolutionary process and meet this need nicely.

5.4.5 Results on simulated and real *micro.adam*

When this system was used to evolve controllers for *micro.adam* in both simulation and on the real robot, results were positive. Over three different runs, the controllers evolved very quickly. Figure 5-9 illustrates the fitness performance. The “generations” axis indicates generations in the typical NEAT sense; that is, a “generation” is N evaluations, where N is the population size (in all of these experiments, a population size of 150 was used).

In order to ensure that the results from the 2199 random seed were not simply a product of luck, the experiment was repeated with two different random seeds. The results are illustrated in figure 5-10. Note in particular that the run resulting from random seed 1234567 achieved a fitness on the real robot that nearly matched the results from the hand-designed algorithm. Of further interest is the note that, of the three evolutionary runs with real feedback, the simulated results for this run never achieved a fitness greater than 70; this run was the best performer in the real world

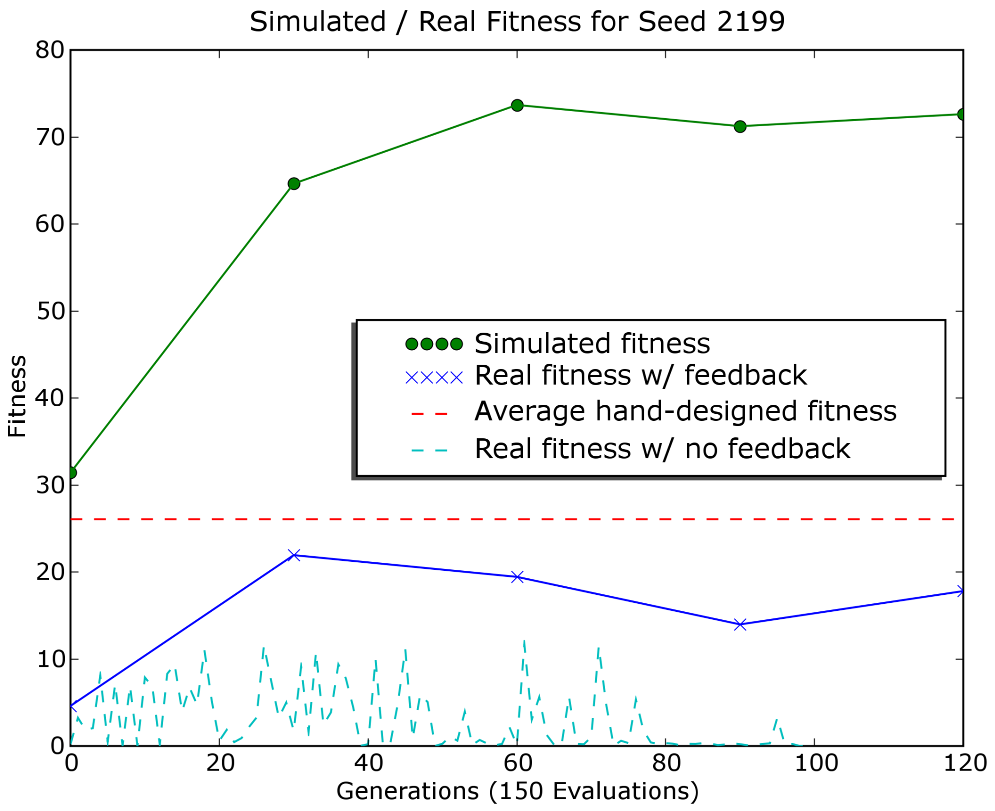
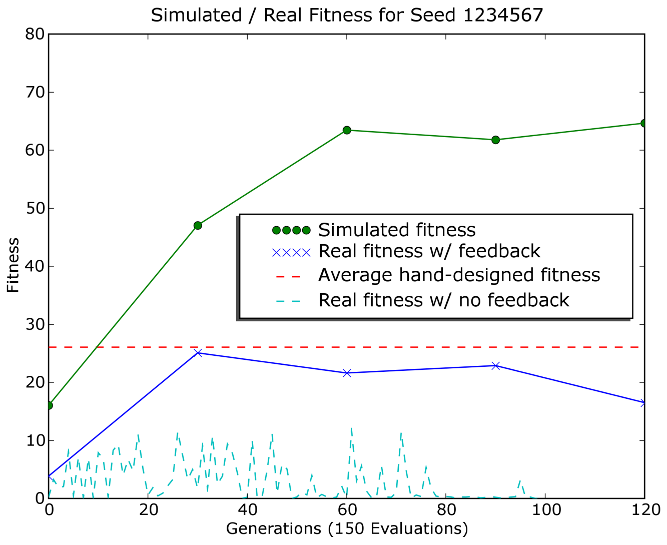


Figure 5-9: When feedback is incorporated into the evolutionary system, the resulting evolved controllers improve over time, approaching the average performance of the hand-designed algorithm. The bottom graph is the data from figure 5-5.

and the worst performer in simulation.



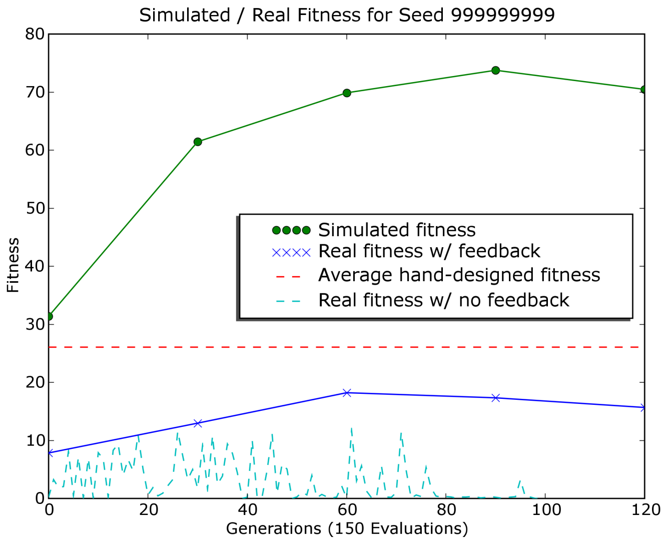


Figure 5-10: Results from using the real-feedback evolutionary system with two different random seeds. In both cases, the fitness of the performance on the real robot was much higher than when the feedback was not used.

Chapter 6

Evolution, Simulation, and Reality

The use of evolutionary algorithms to synthesize controllers for physical robots is not yet a mature technology. If the eventual goal is to use evolutionary algorithms as a tool for creating controllers on robots that compete with or surpass hand designs, much work remains to be done. While research is ongoing in several university labs, no known military technologies or commercial ventures make use of evolved neural controllers for robots. The work in this dissertation is focused on extending the understanding of evolutionary algorithms for controlling physical robots, with particular focus on how this understanding can be applied to difficult problems in the real world.

Section 6.1 will propose a scenario in which evolutionary robotics could augment or replace hand-designed control algorithms. Section 6.2 will then examine the progress towards the goal of a robot incubator from the standpoint of Technology Readiness Levels [15], with a specific focus on how the work in this dissertation has improved the readiness of evolutionary algorithms on physical robots. The last three sections will examine this dissertation's contributions toward this goal, suggesting directions for future work. Section 6.3 will discuss the contributions from the repeated structure chapter of this dissertation, including both high-level questions about biological modeling and more detailed discussion of open problems. Section 6.4 will discuss the contributions from the work on combining subsumption and neuro-evolution in this dissertation, with a focus on how evolution might be deployed to create more robust distributed systems. Finally, section 6.5 will review the contributions of the work incorporating feedback from physical robotic systems, including how this new step impacts the future of synthetic robot controller design.

6.1 The robot incubator

Programming a robot is a task that requires knowledge not just about the robot, but about the domain in which the robot operates. The system designer must understand how the robot operates, how those operations will change the environment, and which changes will most effectively lead to the final goal state. In the case of the Roomba, for example, the system designer must be able to understand how to use odometry for navigation, but must also understand how slipping wheels can make odometry inaccurate, and further, how inaccuracies in the odometry might impact a chosen algorithm for cleaning the floor. While these problems are all tractable (and the hand-designed algorithm for the Roomba is effective), as robot become more complex, hand-designs will be more difficult to invent and implement.

Consider, on the other hand, the model for evolutionary robotics. The system designer can focus his energy on building a robust physical robot, a hand-designed control layer, and a reasonable simulation of the robot and environment. These elements are combined into a single system that one might term a “robot incubator.” The incubator is seeded with these raw materials and then left to develop slowly, but autonomously, over an extended period. As the incubator tests controllers in both simulation and on the physical robot, the details of the interactions between the controller and the robot and the environment are used to filter the controllers. At the end of the development period, the system designer returns to find a controller that has been synthesized and validated both in simulation and on the physical robot in the target environment.

Like any evolutionary system, the main drawback to the robot incubator would be uncertainty about the optimality of the result. A system designer would have to evaluate the performance of the evolved controller without any concrete reference for performance. However, very few hand-designed controllers guarantee optimal behavior, and, for many complex robotic systems and environments, "optimal" behavior would be difficult to quantify. A more likely benchmark is a hand-designed controller, and this suggests that the robot incubator, at first, might be best employed in parallel with the development of a hand-designed solution.

6.2 Technology Readiness Levels

While the robot incubator seems to be an interesting and potentially powerful tool, evolutionary algorithms are not yet well-understood enough to support it as an application. Because this dissertation aims to improve the use of evolutionary algorithms towards that goal, it is helpful to understand what the state of the technology is currently, and how the work in this dissertation improves that state.

A standard method for judging the state of evolutionary robotics would be to use the Technology Readiness Levels described in [15]. The system of Technology Readiness Levels provides a template by which to assess the progress of an emerging technology, and, while imperfect, represents a widely-used standard for measurement. The scale starts at TRL 1 ("Basic principles observed and reported") and runs through TRL 9 ("Actual system proven through successful mission operation"). Table 3-3 of [15] provides lengthier descriptions of the TRL levels for software, and evolved neural network controllers for physical robots are between TRL 3:

> Algorithms run on a surrogate processor in a laboratory environment, instrumented components operating in laboratory environment, laboratory results showing validation of critical properties.

and TRL 4:

> Advanced technology development, stand-alone prototype solving a synthetic full-scale problem, or standalone prototype processing fully representative data sets.

Two key phrases from TRL 4 stand out: "fully representative data sets" and "full-scale problem." This dissertation applied evolutionary algorithms to two new robot platforms that are described by those phrases. While many (if not most) applications of evolutionary algorithms take place on small, stable mobile robots like the Khepera, Julius Popp's *micro.adam* and *micro.eva* depend on the dynamics of the real world for angular motion. Applying evolutionary algorithms, both in the context of repeated structure and in the context of physical feedback, to these highly dynamic robots and measuring the resulting behavior represents a step toward more "fully representative data sets." Further, the Roomba is a robot that has been carefully programmed by hand to clean dirt from floors (and is widely used by consumers). Creating a controller that competes with a truly powerful hand-designed algorithm through the use of a behavioral layer of control is a small step towards using evolutionary algorithms on the "full-scale problem." The following three sections will review the contributions to the field of evolutionary algorithms that enabled these advances and suggest additional research that would further progress towards this goal.

6.3 Repeated structure

Because robots designed by hand (and many evolved ones as well) are likely to have repeated structure, an evolutionary mechanism for repeating structure in evolved controllers is highly advantageous. Biological solutions include this mechanism by default: the same evolutionary machinery that creates repeated structure in the body can create it in the control. Chapter 3 demonstrated a set of enhancements to NEAT that added a spatial representation to evolved networks and then used that spatial representation to discover regions of repeated structure in the problem space. The implementation was described in detail, and the system was demonstrated on both a multi-bit XOR benchmark task and on a simulated robot task. The enhancements made the multi-bit XOR task evolvable and created a significant improvement in the simulated *micro.eva* task.

The precise implementation of the repeated structure mechanism raises several important questions about the use of biological metaphors in artificial evolution. The entire evolutionary enterprise is based on the assumption that biological machinery can be abstracted while preserving the essential phenomena.[1] However, the choice of which details to include and which to leave out are difficult to compare when the choices typically exist in very different systems. For example, the GasNets system [59] and the Norgev system [2] have significant overlap in biological inspiration: both are attempting to extract function from the notion of spatial distribution and diffusive signaling. However, GasNets allows the neurons to exist anywhere in a real-valued plane, whereas Norgev divides the space of both neurons and diffusion into a discrete hexagonal plane. However, comparing these systems does not reveal if the use of a real-valued space provides an advantage over a discrete space because the systems are also different in many other respects, and are tested on different tasks. Solving this problem is tricky: researchers cannot be asked to vary each element of their abstraction to better understand the impact of the choice. And yet, someone building an evolutionary system *de novo* still has no sense of the tradeoffs involved in choosing a real-valued versus a discrete neural space.

NEAT avoids this problem by focusing on the function of the final system; by benchmarking its performance on the XOR problem, it demonstrates that the particular choices made by NEAT, when optimized, result in a given level of performance. By comparing that level to other systems on the same problem, they provide a measure, albeit limited, of how innovation numbers (which are also motivated by biological mechanisms) improve the performance of evolved neural networks. The mechanism for repeated structure outlined in chapter 3 was similarly measured on the multi-bit XOR test, which might serve as a useful benchmark for repeated structure mechanisms. Perhaps even more useful would be the benchmark of the multi-bit ADD test, wherein the goal is to take two interleaved binary numbers and output their binary sum. Evolving such a mechanism was a goal of the work in this dissertation, but very modest progress was made (a solution was evolved to the two-bit addition problem on less than 2% of runs). A more thorough understanding of this problem, which

[1]This statement is limited to the functional use of evolutionary algorithms, which includes most if not all of evolutionary robotics. Those using evolutionary methods to reproduce biological phenomena for its own sake have different goals and would make different decisions about abstraction.

requires interleaved repeated structure (that is, links that connect one module to the next) might further progress towards the goal of discovering mechanisms for evolved modularity.

6.4 Behavioral control

To the extent that artificial evolution mimics biological evolution, the distributed control system of the polyclad flatworm should serve as a model for how neuroevolutionary techniques might start to build complexity. Instead of starting with a simple central controller that is augmented through evolution, biological control structures seem to have evolved independently (even when controlling the same creature), and are later coordinated and combined using a central control structure. Chapter 4 demonstrated an implementation of this idea on both the simulated and the physical Roomba. A simple distributed control structure was built by hand and its implementation was described. Controllers were then evolved for the Roomba on a floor coverage task, both directly and using this hand-designed layer. The use of the hand-designed layer demonstrated a significant improvement over the direct evolutionary approach, both in simulation and on the physical robot. Moreover, the improvement on the physical robot elevated the performance to be competitive with the hand-designed controller, albeit in a somewhat limited context.

The approach taken in chapter 4 could be extended through further research into the interface between the hand-designed control layer and the evolved control layer. The hand-designed AFSMs described in section 4.4.2 are fairly primitive and not necessarily optimized for use in evolution. Further thought and experimentation might suggest better rules or heuristics for how to design a distributed control mechanism to be "evolvable."

Another, perhaps even more powerful extension would be to evolve both the small, distributed controllers that drive the sensorimotor reflexes on the robot as well as the coordinating central control. By removing the hand-designed elements but retaining the independence of different control structures, the biological analog could be more closely followed and the resulting controller might find additional robustness. A first step in this direction might be simply combining some versions of current work; the use of NEAT in combination with work such as [64] applied to a real robot might yield interesting results.

6.5 Feedback from the physical robot

The use of mobile robots, simulated and real, is ubiquitous in evolutionary robotics. Indeed, a survey of evolutionary robotics that excluded locomotion and navigation as task domains would be vanishingly small. This preponderance on two problems reveals an assumption that the use of evolutionary robotics is limited to simple robots with trivial dynamics. Unfortunately, simple systems with trivial dynamics are precisely the systems for which hand-designed solutions are readily available. In many cases, the controllers evolved by evolutionary approaches represent significant accomplishments in evolutionary robotics, and yet suffer in comparison to hand designs due to the simplicity of the robot and the environment.

The physical feedback system presented in chapter 5 allows more complex robotic systems with less accurate simulations to take advantage of an evolutionary approach. The problem of the "reality gap," controllers that evolve to perform well in simulation but perform poorly on a physical robot, was introduced and quantified on *micro.adam*. A system for periodically evaluating evolved controllers on the physical robot and feeding the results of those evaluations back into the evolutionary algorithm was introduced and some implementation details were reviewed. The system was then applied to *micro.adam*, and the performance of the evolved controllers was demonstrated to improve dramatically.

An obvious approach for expanding on the use of physical feedback would be to combine it with the previous two ideas in this dissertation, repeated structure and behavioral control. The use of feedback from a physical robot seems to be a natural extension to both the repeated structure experiments and the behavioral primitive experiments. The repeated structure experiments demonstrated a performance improvement in simulation, but further experimentation is required to demonstrate that those improvements would apply to the physical robot. The combination of subsumption and physical feedback creates two opportunities for improvements. Testing the evolved networks will on the physical robot will ensure that no “reality gap” solutions have been discovered, but the physical feedback could also be used to verify the hand-designed AFSMs. By executing the same AFSM in simulation and on the physical robot, either the AFSM or the simulation itself could be adjusted to narrow the “reality gap” and create better solutions.

Finally, the use of physical feedback in this dissertation is limited by the real-time performance of the robot. While the artificial evolution strategy can be adjusted to best use the time spent testing on the physical robot, many long evaluations are still impractical for the reasons listed in [46]. However, if many physical robots were available, this bottleneck could be partially alleviated. Instead of running ten short evolutions in sequence, testing each on an individual robot in turn, ten short evolutions could be run in parallel, and the tests could be run simultaneously, drastically shortening the evaluation time. The use of a “farm” of testing robots would also address another concern: overfitting to a single robot. Having many copies of a physical robot creates the possibility for testing the generality of an evolved solution despite changes in the specific parameters of the physical robot. In this way, artificial evolution takes a significant step away from being an academic exercise for trivial robots and towards being a serious tool for use on complex robots in the real world.

Bibliography

[1] Artificial Life. *A Developmental Model for the Evolution of Artificial Neural Networks*, volume 6, 2000.

[2] Jens C. Astor and Christoph Adami. Development and evolution of neural networks in an artificial chemistry. In *Proceedings of the Third German Workshop on Artificial Life*, pages 15–30, 1998.

[3] R.D. Beer, H.J. Chiel, and L.S. Sterling. A biological perspective on autonomous agent design. *Journal of Robotics and Autonomous Systems*, 6:163–178, 1990.

[4] Adrian Boeing, Stephen Hanham, and Thomas Braunl. Evolving autonomous biped control from simulation to reality. In *Proceedings of 2nd International Conference on Autonomous Robots and Agents (ICARA2004)*, pages 440–446, Palmerston North, New Zealand, December 2004.

[5] J. Bongard. Evolving modular genetic regulatory networks. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC2002)*, pages 1872–1877, 2002.

[6] J. Bongard and R. Pfeifer. Evolving complete agents using artificial ontogeny. *Morpho-functional Machines: The New Species (Designing Embodied Intelligence)*, pages 237–258, 2003.

[7] Valentio Braitenberg. *Experiments in Synthetic Biology.* MIT Press, Cambridge, MA, 1984.

[8] R. Brooks. Artificial life and real robots. In *European Conference on Artificial Life*, pages 3–10, 1992.

[9] Rodney Brooks. A robot that walks: Emergent behavior from a carefully evolved network. *Neural Computation*, 1(2):253–262, 1989.

[10] Mat Buckland. *AI Techniquies for Game Programming*. Course Technology PTR, 2645 Erie Ave. Cincinnati, OH 45208, 1 edition, October 2002.

[11] R. Calabretta, S. Nolfi, D. Parisi, and G. Wagner. Emergence of functional modularity in robots. In R. Pfeifer, B. Blumberg, J.-A. Meyer, and S.W. Wilson, editors, *From Animals to Animats 5*, pages 497–504, Cambridge, MA, 1998. MIT Press.

[12] S.M. Catalano and C. J. Shatz. Activity-dependent cortical target selection by thalamic axons. *Science*, 281(5376):559–62, July 1998.

[13] J.D.W. Clarke and Andrew Lumsden. Segmental repetition of neuronal phenotype sets in the chick embryo hindbrain. *Development*, 118:151–162, 1993.

[14] Lynnae Davies, Larry Keenan, and Harold Koopowitz. Nerve repair and behavioral recovery following brain transplantation in notoplana acticola, a polyclad flatworm. *The Journal of Experimental Zoology*, 235:157–173, 1985.

[15] Department of Defense. *Technology Readiness Assessment (TRA) Deskbook*, May 2005.

[16] Christoph Eckert, Manuel Aranda, Christian Wolff, and Diethard Tautz. Separable stripe enhancer elements for the pair–rule gene hairy in the beetle tribolium. *EMBO reports*, pages 638–642, May 2004.

[17] Peter Eggenberger, Akio Ishiguro, Seiji Tokura, Toshiyuki Kondo, and Yoshiki Uchikawa. Toward seamless transfer from simulated to real worlds: A dynamically–rearranging neural network approach. In *EWLR–8: Proceedings*

of the 8th European Workshop on Learning Robots, pages 44–60, London, UK, 2000. Springer–Verlag.

[18] Dario Floreano and Francesco Mondada. Hardware solutions for evolutionary robotics. In Phil Husbands and Jean-Arcady Meyer, editors, *Proceedings of the First European Workshop on Evolutionary Robotics*, pages 137–151, 1998.

[19] Dario Floreano and Mototaka Suzuki. Active Vision and Neural Development in Animals and Robots. In *The Seventh International Conference on Cognitive Modeling (ICCM'06)*, pages 10–11, 2006.

[20] John C. Gallagher, Randall D. Beer, Kenneth S. Espenschied, and Roger D. Quinn. Application of evolved locomotion controllers to a hexapod robot. *Robotics and Autonomous Systems*, 19:95–103, 1996.

[21] Frederic Gruau. Automatic definition of modular neural networks. *Adaptive Behaviour*, 3(2):151–183, 1995.

[22] Sheila Gruber and D. W. Ewer. Observation on the myo-neural physiology of the polyclad, planocera gilchristi. *Journal of Experimental Biology*, 39(3):459–477, 1962.

[23] D. W. Halton and A. G. Maule. Flatworm nerve-muscle: structural and functional analysis. *Canadian Journal of Zoology*, 82:316–333, 2004.

[24] Alan N Hampton and Chris Adami. Evolution of robust developmental neural networks. In *Artificial Life*, volume 9, 2004.

[25] I. Harvey, P. Husbands, D. Cliff, A. Thompson, and N. Jakobi. Evolutionary robotics: the sussex approach. *Robotics and Autonomous Systems*, 1996.

[26] Greg S. Hornby, S. Takamura, J. Yokono, O. Hanagata, M. Fujita, and J. Pollack. Evolution of controllers from a high–level simulator to a high dof robot. In J. Miller, editor, *Evolvable Systems: from biology to hardware; proceedings of the third international conference (ICES 2000)*, pages 80–89, 2000.

[27] Gregory S. Hornby. Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1729–1736, New York, NY, USA, 2005. ACM Press.

[28] Peter Eggenberger Hotz, Gabriel Gomez, and Rolf Pfeifer. Evolving the morphology of a neural network for controlling a foveating retina – and its test on a real robot. In *Artificial Life VIII*, 2002.

[29] M. Hülse, S. Wischmann, and F. Pasemann. Structure and function of evolved neuro-controllers for autonomous robots. *Connection Science*, 16(4):249–266, 2004.

[30] Martin Hülskamp and Diethard Tautz. Gap genes and gradients – the logic behind the gaps. *Bioessays*, 13(6):261–268, June 1991.

[31] Phil Husbands. Evolving robot behaviours with diffusing gas networks. In *EvoRobots*, pages 71–86, 1998.

[32] iRobot. *The Roomba Serial Control Interface*, 2005.

[33] Johannes Jaeger, Svetlana Surkova, Maxim Blagov, Hilde Janssens, David Kosman, Konstantin N. Kozlov, Manu, Ekaterina Myasnikova, Carlos E. Vanario-Alonso, Maria Samsonova, David H. Sharp, , and John Reinitz. Dynamic control of positional information in the early drosophila embryo. *Nature*, 430:368–371, July 2004.

[34] N. Jakobi. *Minimal Simulations for Evolutionary Robotics.* PhD thesis, University of Sussex, 1998.

[35] N. Jakobi. Running across the reality gap: Octopod locomotion evolved in a minimal simulation. In Phil Husbands and Jean-Arcady Meyer, editors, *Proceedings of the First European Workshop on Evolutionary Robotics*, pages 39–58, 1998.

[36] N. Jakobi, P. Husbands, and I. Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In F. Moran, A. Moreno, J. J. Merelo, and P. Chacon, editors, *Advances in Artificial Life: Proceedings of the Third International Conference on Artificial Life*, pages 704–720. Springer-Verlag, 1995.

[37] Nick Jakobi, Phil Husbands, and Tom Smith. Robot space exploration by trial and error. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 807–815, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[38] K–Team. *The Khepera Users Manual.*

[39] Harold Koopowitz. Polyclad neurobiology and the evolution of central nervous systems. In Peter A.V. Anderson, editor, *Evolution of the First Nervous systems*, pages 315–327, New York, 1989. Plenum Press.

[40] Harold Koopowitz and Larry Keenan. The primitive brains of platyhelminthes. *Trends in Neurosciences*, 5(3):77–79, 1982.

[41] John R. Koza. Evolution of a subsumption architecture that performs a wall following task for an autonomous mobile robot via genetic programming. In Thomas Petsche, editor, *Computational Learning Theory and Natural Learning Systems*, volume 2, pages 321–346. MIT Press, Cambridge, MA, USA, 1994.

[42] Peter A. Lawrence. *The Making of a Fly.* Blackwell Publishers, Cambridge, MA, 1992.

[43] Wei-Po Lee, John Hallam, and Henrik Hautop Lund. Learning complex robot behaviours by evolutionary computing with task decomposition. In *EWLR-6: Proceedings of the 6th European Workshop on Learning Robots*, pages 155–172, London, UK, 1998. Springer-Verlag.

[44] Martin C. Martin. *The Simulated Evolution of Robot Perception.* PhD thesis, Carnegie Mellon University, 2001.

[45] Maja Matarić. A distributed model for mobile robot environment-learning and navigation. Master's thesis, Massachusetts Institute of Technology, 1990.

[46] Maja Matarić and David Cliff. Challenges in evolving controllers for physical robots. *Robotics and Autonomous Systems*, 19:67–83, 1996.

[47] Olivier Michel. Webots: Symbiosis between virtual and real mobile robots. *Lecture Notes in Computer Science*, 1434, January 1998.

[48] Orazio Miglino, Henrik Hautop Lund, and Stefano Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2(4):417–434, 1995.

[49] Risto Miikkulainen, Joseph Reisinger, and Kenneth Stanley. Evolving reusable neural modules. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004)*, New York, NY, 2004. Springer-Verlag.

[50] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, 1988.

[51] Motorola. *M68HC11 Reference Manual*, 1991.

[52] S. Nolfi and D. Parisi. Learning to adapt to changing environments in evolving neural networks. *Adaptive Behavior*, 5(1):75–98, 1997.

[53] Choong K. Oh and Gregory J. Barlow. Autonomous controller design for unmanned aerial vehicles using multi-objective genetic programming. In *Proceedings of the 2004 Congress on Evolutionary Computation*, pages 1538–1545, Portland, OR, June 2004.

[54] N.H. Patel. Developmental evolution: insights from studies of insect segmentation. *Science*, 266:581–90, October 1994.

[55] David Pogue. The robot returns, cleaning up. The New York Times, August 28 2003.

[56] Jordan B. Pollack, Hod Lipson, Sevan Ficci, Pablo Funes, Greg Hornby, and Richard A. Watson. Evolutionary techniques in physical robotics. In J. Miller, editor, *Evolvable Systems: from biology to hardware; proceedings of the third international conference (ICES 2000)*, pages 175–186, 2000.

[57] K. Sims. Evolving 3d morphology and behavior by competition. In R. Brooks and P. Maes, editors, *Artificial Life IV*, pages 28–39, Cambridge, MA, 1994. MIT Press.

[58] Russel Smith. *The Open Dynamics Engine v0.5 User Guide*. The ODE Project, 2006.

[59] Tom Smith, Phil Husbands, and Michael O'Shea. Temporally adaptive networks: Analysis of gasnet robot controllers. In *Artificial Life VIII*, pages 274–282, 2002.

[60] Tom M. C. Smith. Blurred vision: Simulation–reality transfer of a visually guided robot. In Phil Husbands and Jean-Arcady Meyer, editors, *Proceedings of the First European Workshop on Evolutionary Robotics*, pages 253–164, 1998.

[61] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[62] Kenneth O. Stanley and Risto Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130, 2003.

[63] Michael Stilman, Philipp Michel, Joel Chestnutt, Koichi Nishiwaki, Satoshi Kagami, and James Kuffner. Augmented reality for robot development and experimentation. Technical Report CMU-RI-TR-05-55, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, November 2005.

[64] Ricardo A. Téllez and Cecilio Angulo. Evolving cooperation of simple agents for the control of an autonomous robot. In *5th IFAC Symposium on Intelligent Autonomous Vehicles (IAV04)*, 2005.

[65] Marc Tessier-Lavigne and Corey S. Goodman. The molecular biology of axon guidance. *Science*, 274:1123–1133, 1996.

[66] Julian Togelius. Evolution of the layers in a subsumption architecture robot controller. Master's thesis, University of Sussex, 2003.

[67] Julian Togelius. Evolution of a subsumption architecture neurocontroller. *Journal of Intelligent and Fuzzy Systems*, 15(1), 2004.

[68] Seth Tyler and Matthew Hooge. Comparitive morphology of the body wall in flatworms (platyhelminthes). *Canadian Journal of Zoology*, 82:194–210, 2004.

[69] Paul Viola. Mobile robot evolution. Master's thesis, Massachusetts Institute of Technology, May 1988.

[70] Rodney A. Webb. Studies on platyhelminthes: yesterday, today, and tomorrow. *Canadian Journal of Zoology*, 82:161–167, 2004.

[71] Darrell Whitley, Frederic Gruau, and Larry Pyeatt. Cellular encoding applied to neurocontrol. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 460–467, Pittsburgh, PA, USA, 15-19 1995. Morgan Kaufmann.

[72] S. Wischmann, M. Hülse, and F. Pasemann. (coevolution of (de)centralized neural control for a gravitationally driven machine. In M. Capcarrere, A. A. Freitas, P. J. Bentley, C.G. Johnson, and J. Timmis, editors, *Proc. of the 8th European Conference on Artificial Life (ECAL 2005)*, LNAI 3630, pages 179–188, 2005.

[73] Krister Wolff and Peter Nordin. An evolutionary based approach for control programming of humanoids. In *Proceedings of the Third IEEE–RAS International Conference on Humanoid Robots*, 2003.

[74] Z.H. Zhou and X.H. Shen. Virtual creatures controlled by developmental and evolutionary cpm neural networks. *Intelligent Automation and Soft Computing, 2003, 9(1): 23–30*, 9(1):23–30, 2003.