Aug 2000

Topics in Evolutionary Computation: Final Report for Grant No. N00173-98-1-G010

PR Number: 55-1041-98,
Disbursing Code: N68892, AGO Code: N66020, CAGE Code: 7X764

PI: John Grefenstette, Ph.D.
School for Computational Sciences
George Mason University
MSN 4E3
10900 University Drive
Manassas, Virginia 20110

Abstract

This project contributed new principles for the development of intelligent, mobile robots performing complex tasks in unpredictable environments. In the behavior-based approach to robot design, the overall performance of the robot arises through the interaction of multiple, relatively simple, behaviors. The manual design of multiple interacting behaviors is difficult, labor-intensive and error-prone. One way to reduce the effort in the design of behavior-based robots is to develop an evolutionary approach in which the various behaviors, as well as their modes of interaction, evolve over time. Evolution may also provide a basis for the development of strategies for multiple-robot environments, for example, environments in which a robot is expected to adapt its behavior based on the current behavior of other agents or environmental conditions which themselves are changing over time. This project addressed in four complementary areas concerning the effectiveness of evolutionary algorithms for the design of autonomous robots: (1) learning multiple behaviors by asynchronous co-evolution; (2) continuous and embedded learning; (3) comparison with other reinforcement learning methods, and (4) the ability to evolve responses to changing environments. Results in each of these tasks are reported.

1 Executive Summary

This is the Final Technical Report for Grant No. N00173-98-1-G010, *Topics in Evolutionary Computation*, performed by John Grefenstette at George Mason University in response to NRL BAA 705. The general research topic addressed by this project concerned the applicability of genetic algorithms to robotic learning. The specific research objectives were as follows:

- To improve the understanding of evolution as a process that produces robust organisms that are well-adapted to complex environments;
- To improve methods for designing multi-agent systems that exhibit robust and adaptive collective behavior. Multi-agent systems include both behavior-based robots in which agents are identified with individual behaviors and multiple-robot cooperative systems.

This project contributed to ongoing work at NRL on new principles for the development of intelligent, mobile robots performing complex tasks in unpredictable environments. In the behavior-based approach to robot design, the overall performance of the robot arises through the interaction of multiple, relatively simple, behaviors. The manual design of multiple interacting behaviors is difficult, labor-intensive and error-prone. One way to reduce the effort in the design of behavior-based robots is to develop an evolutionary approach in which the various behaviors, as well as their modes of interaction, evolve over time. Evolution may also provide a basis for the development of strategies for multiple-robot environments, for example, environments in which a robot is expected to adapt its behavior based on the current behavior of other agents or environmental conditions which themselves are changing over time.

This work was performed in collaboration with the Adaptive Systems Group (Code 5514) at the Navy Center for Applied Research in Artificial Intelligence at NRL. Principles of co-evolutionary design were investigated in the context of co-evolving competitive and cooperative behaviors in mobile robots. This project addressed in four complementary areas concerning the effectiveness of evolutionary algorithms for the design of autonomous robots:

- Learning multiple behaviors by asynchronous co-evolution,
- Continuous and embedded learning,
- Comparison with other reinforcement learning methods, and
- Ability to evolve responses to changing environments.

The following section summarizes the results in each area. Detailed descriptions appear in the Appendices.

2 Summary of Results

2.1 Learning Multiple Behaviors by Asynchronous Co-evolution

This project addressed the ability of autonomous robots to learn complex behaviors by co-evolution. Using a *behavior-based* approach, the overall performance of the robot arises through the interaction of multiple, relatively simple behaviors, here called *agents*. This project advanced the state of the art in the automated learning of multiple agents. We designed, implemented and tested a number of alternative decompositions for complex tasks. In particular, we focused on the effect of *dependencies* among the learning agents, that is, what a given learning agent needs to know about the other agents in the system. These issues were examined in the context of a particular performance task in which the robot was required to track another moving agent for a prolonged period of time, without colliding with obstacles or the other agent, and without running out of fuel. If fuel becomes low, the robot must return to a docking station and re-fuel. All decisions about tracking strategy, docking strategy, and when to apply each strategy, is under control of the learning system.

In asynchronous learning, each evolving agent informs the other learning agents whenever it discovers an improved behavior, so that the improved behavior can be incorporated in the other agent's world model. Two asynchronous learning regimes were considered: an *independent* regime in which behaviors were learned as if each behavior was the only task in the system, and a *mutual* regime in which each behavior is learned using the best current agent for the other behaviors in the system. Extensive computational simulations demonstrated that independent regime consistently outperforms

the mutual regime in asynchronous learning. Details appear in Appendix A. These results were presented at the 1999 SPIE Symposium on Intelligent Systems and Advanced Manufacturing. [1].

2.2 Continuous and Embedded Learning

An important problem arising in robots that are expected to perform autonomously for extended periods is how to adapt the robot's rules of behavior in response to unexpected changes in its own capabilities. For example, suppose the robot periodically checks its sensors and its ability to perform basic actions. If it finds that some sensors or actions are no longer available, perhaps due to a problem with the robot's hardware or due to some undetected environmental cause, then it must learn new rules for performing its mission that use whatever remaining capabilities are still available.

We have developed an approach to this problem called *Continuous and Embedded Learning* (CEL). In this approach, the robot interacts both with the external environment and with an internal simulation. The robot's execution module controls the robot's interaction with the environment, and includes a monitor that dynamically modifies the robot's internal simulation model based on the monitor's observations of the actual robot and the sensed environment. The robot's learning module continuously tests new strategies for the robot against the simulation model, using a genetic algorithm to evolve improved strategies, and updates the knowledge base used by the execution module with the best available results. Whenever the simulation model is modified due to some observed change in the robot or the environment, the genetic algorithm is restarted on the modified model. The learning system operates indefinitely, and the execution system uses the results of learning as they become available.

In this project, we examined the CEL model in the face of sensor failures, specifically, how the robot can learn to adapt to failures in its sensor capabilities over time. We show that a robot adapt to the partial loss of its sensors and learn to use different sensors to continue to perform a door traversal task. Both simulation studies and experiments on an actual mobile robot showed that the approach yields effective adaptation to a variety of partial sensor failures. The robot used in these experiments is a Nomadic Technologies Nomad 200 mobile robot, a three-wheeled, synchronized-steering vehicle used in experimental studies at NRL. The internal simulation used by the robot for learning approximated the Nomad robot's sensors and effectors.

In simulation studies, the robot initially learned to improve its performance of the task from 25% to 63% with all seven sonar sensors operating. Upon the failure of three sonars (front, front right, and right), performance initially dropped to 37%, but then rebounded to over 60% as the monitor identified the failed sonars, modified the simulation, and the learning system adapted to the new simulation model. These results were verified by repeating the same rules on the Nomad robot, both with and without sensor disabled.

These result indicate that the Continuous and Embedded Learning model is a promising approach to adapting to partial sensor failures. Combined with our previous work showing adaptation to changing environments and actuator failures [2, 3, 4], this work indicates the generality of the CEL model for the design of robust autonomous robot systems. Appendix B describes the results in detail. These results were presented at the 2000 SPIE Symposium on Unmanned Ground Vehicle Technology. [5]

2.3 Comparison with Other Reinforcement Learning Methods

Reinforcement learning (RL) provides a flexible approach to the design of intelligent agents in situations for which both planning and supervised learning are impractical. RL can be applied to problems for which significant domain knowledge is either unavailable or costly to obtain. For example, designers of autonomous robots often lack sufficient knowledge of the intended operational environment to use either the planning or the supervised learning regime to design a control policy for the robot. In this case, the goal of RL would be to enable the robot to generate effective decision policies as it explores its environment.

There are two main approaches to finding the optimal policy, one involves search in *policy space* and the other involves search in *value function space*. Policy space search methods maintain explicit representation(s) of policies and modify them through a variety of search operators. Many search methods have been considered including dynamic programming, value iteration, simulated annealing, and evolutionary algorithms. In contrast, value function methods do not maintain an explicit representation of a policy. Instead, they attempt learn the value function which returns the expected cumulative reward for the optimal policy from any state. The focus of research on value function approaches to RL is to design algorithms that learn these value functions through experience. The most common approach to learning value functions is the temporal difference (TD) method. The TD learning algorithm uses observations of *prediction differences* from consecutive states to update value predictions. Both approaches assume limited knowledge of the underlying system and learn by experimenting with different policies and using reinforcement to alter those policies. Neither approach requires a precise mathematical model of the domain, and both may learn through direct interactions with the operational environment. In an article published in the *Journal of AI Research*, we reviewed the application of evolutionary algorithms for reinforcement learning (EARL), emphasizing alternative policy representations, credit assignment methods, and problem-specific genetic operators. Strengths and weaknesses of the evolutionary approach to reinforcement learning were examined, along with a survey of representative applications.

Unlike TD methods, EARL methods generally base fitness on the overall performance of a policy. In this sense, EA methods pay less attention to individual decisions than TD methods do. While this appears to make less efficient use of information, it may in fact provide a robust path toward learning good policies, especially in situations where the sensors are inadequate to observe the true state of the world. Evolutionary algorithms for RL use a distinctive set of representations for policies. First, policies may be represented either by condition-action rules or by neural networks. Second, policies may be represented by a single chromosome or the representation may be distributed through one or more populations.

We also considered the different approaches to credit assignment in the TD and EA methods. In a reinforcement learning problem, payoffs may be sparse, that is, only associated with certain states. Consequently, a payoff may reflect the quality of an extended sequence of decisions, rather than any individual decision. For example, a robot may receive a reward after a movement that places it in a "goal" position within a room. The robot's reward, however, depends on many of its previous movements leading it to that point. A difficult *credit assignment* problem therefore exists in how to apportion the rewards of a sequence of decisions to individual decisions. In general, EA and TD methods address the credit assignment problem in very different ways. In TD approaches, credit from the reward signal is explicitly propagated to each decision made by the agent. Over many iterations, payoffs are distributed across a sequence of decisions so that an appropriately discounted reward value is associated with each individual state and decision pair. In EARL systems, rewards are only associated with sequences of decisions and are not distributed to the individual decisions. Credit assignment for an individual decision is made implicitly, since policies that prescribe poor

individual decisions will have fewer offspring in future generations. By selecting against poor policies, evolution automatically selects against poor individual decisions. That is, building blocks consisting of particular state-action pairs that are highly correlated with good policies are propagated through the population, replacing state-action pairs associated with poorer policies.

The EA approach represents an interesting alternative for solving RL problems, offering several potential advantages for scaling up to realistic applications. In particular, EARL systems have been developed that address difficult challenges in RL problems, including:

- Large state spaces;
- Incomplete state information; and
- Non-stationary environments.

Areas that are especially challenging for evolutionary approaches to RL include:

- Online learning: Time and safety concerns argue in favor of using EA when a sufficiently rich simulation environment allows offline learning.
- Rare states: EA do not necessarily preserve information about rarely occurring states.
- Proofs of Optimality: Further theoretical tools are required to provide assurance of EA performance.

EARL and TD, while complementary approaches, are by no means mutually exclusive. There are examples of successful EARL systems such as SAMUEL that explicitly incorporate TD elements into their multi-level credit assignment methods. It is likely that many practical applications will depend on these kinds of multi-strategy approaches to machine learning. The complete article [6] appears in Appendix C.

2.4 Responses to Changing Environments

The environment of autonomous systems can be expected to vary over time. We examined the ability of genetic algorithms to respond to changing environments under evolutionary control. A genetic algorithm can use its population to good advantage in tracking changing objective functions. To the extent that the population remains relatively diverse, the genetic algorithm can maintain a balance between exploration (enforced by recombination and mutation) and exploitation (enforced by selection pressure). As a result, a genetic algorithm can track a slowing varying landscape without difficulty. For more rapidly changing environments, some additional mechanisms may be needed. Previous work by the PI addressed specific ways to improve the performance of genetic algorithms in dynamic environments [7, 8]. First, the use of periodic global *hypermutation*, effectively kicking the population into a temporary random exploration mode, was shown to perform well in abruptly changing landscapes. Second, a method that replaced a fixed percentage of the population each generation by *random immigrants*, i.e., population members that were uncorrelated with the members of the existing population, was shown to be especially effective on gradually shifting landscapes, and resulted in greater search efficiency than a fixed rate of hypermutation on stationary landscapes. Building on these previous results, we showed that hypermutation can be controlled genetically, resulting in a hypermutation rate that adapts to the volatility of the fitness landscape, increasing in response to abrupt shifts and decreasing when the landscape stabilizes. Such an approach may

be very well-suited for optimizing the fitness landscape that of an autonomous robot in a dynamic environment. These results were published in the 1999 Congress on Evolutionary Computation [9], and are included as Appendix D.

3 Conclusions

This project made contributions in four areas concerning the effectiveness of evolutionary algorithms for the design of autonomous robots:

- Learning multiple behaviors by asynchronous co-evolution,
- Continuous and embedded learning,
- Comparison with other reinforcement learning methods, and
- Ability to evolve responses to changing environments.

The result of this project shows that the Continuous and Embedded Learning model is a promising approach to adapting to changes such as partial sensor failures. Further work is required to investigate the ability to adapt to realistic combinations of sensor and actuator failures, and to quantify the limits of adaptability under this model.

References

[1] Robert Daley, Alan C. Schultz, and John J. Grefenstette. Co-evolution of robot behaviors. In *Proc. of SPIE Int. Symp. on Intelligent Systems and Advanced Manufacturing (ISAM '99)*, Boston, MA, 1999.

[2] J. J. Grefenstette and C. L. Ramsey. An approach to anytime learning. In *Proc. Ninth International Conference on Machine Learning*, pages 189–195, San Mateo, CA, 1992. Morgan Kaufmann.

[3] C. L. Ramsey and J. J. Grefenstette. Case-based anytime learning. In D. W. Aha, editor, *Case Based Reasoning: Papers from the 1994 Workshop*, Menlo Park, CA, 1994. Technical Report WS-94-07, AAAI Press.

[4] John J. Grefenstette. Genetic learning for adaptation in autonomous robots. In *Robotics and Manufacturing: Recent Trends in Research and Applications, Volume 6*, pages 265–270, New York, 1996. ASME Press.

[5] Alan C. Schultz and John J. Grefenstette. Continuous and embedded learning in autonomous vehicles: Adapting to sensor failures. In Grant R. Gerhart, Robert W. Gunderson, and Chuck M. Shoemaker, editors, *Unmanned Ground Vehicle Technology II, Proc. of SPIE Vol 4024*, pages 55–62, 2000.

[6] David E. Moriarty, Alan C. Schultz, and John J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:199–229, 1999.

[7] John J. Grefenstette. Genetic algorithms for changing environments. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature, 2*, pages 137–144, 1992.

[8] H. G. Cobb and John J. Grefenstette. Genetic algorithms for tracking changing environments. In *Proc. Fifth International Conference on Genetic Algorithms*, pages 523–530, 1993.

[9] John J. Grefenstette. Evolvability in dynamic fitness landscapes: A genetic algorithm approach. In *Proc. 1999 Congress on Evolutionary Computation*, pages 2031–2038, 1999.

APPENDIX A:

Co-evolution of Robot Behaviors

Robert Daley[a], Alan C. Schultz[b], and John J. Grefenstette[c]

[a]Computer Science Department, University of Pittsburgh
Pittsburgh PA, USA

[b]Adaptive Systems Group, Naval Research Laboratory,
Washington DC, USA

[c]Institute for Biosciences, Bioinformatics and Biotechnology,
George Mason University, Fairfax VA, USA

ABSTRACT

One approach to the design of intelligent autonomous robots is through evolutionary computation. In this approach, the robot's behavior is evolved through a process of simulated evolution, applying the Darwinian principles of survival-of-the-fittest and inheritance-with-variation to the development of the robot's control programs. In previous studies, we illustrated this approach on problems of learning individual behaviors for autonomous mobile robots. Our previous work has focused on tasks which were reasonably complex, but which required only a single behavior. In order to scale this approach to more realistic scenarios, we now consider methods for evolving complex **sets** of tasks. Our approach has been to extend the basic evolutionary learning method to encompasses co-evolution, that is, the simultaneously evolution of multiple behaviors. This paper addresses alternative designs within this basic paradigm. Specifically, we focus on dependencies among the learning agents, that is, what a given learning agent needs to know about other agents in the system. By using domain knowledge, it is possible to reduce or eliminate interactions among the agents, thereby reducing the effort required to co-evolve these agents as well as reducing the impediments to learning caused by these interactions.

Keywords: Robot, Learning, Genetic Algorithms, Coevolution, Evolutionary Algorithms

1. INTRODUCTION

One approach to the design of intelligent autonomous robots is through evolutionary computation. In this approach, the robot's behavior is evolved through a process of simulated evolution, applying the Darwinian principles of survival-of-the-fittest and inheritance-with-variation to the development of the robot's control programs. In previous studies, we illustrated this approach on problems of learning individual behaviors for autonomous mobile robots. Using an evolutionary learning system called SAMUEL, we have been able to evolve robot behaviors including navigation, collision avoidance, tracking and herding. Our previous work has focused on tasks which were reasonably complex, but which required only a single behavior. In order to scale this approach to more realistic scenarios, we now consider methods for evolving complex **sets** of tasks. Our approach has been to extend the basic evolutionary learning method to encompass co-evolution, that is, the simultaneously evolution of multiple behaviors. This article addresses alternative designs within this basic paradigm.

We adopt a behavior-based approach in which the overall performance of the robot arises through the interaction of multiple, relatively simple, behaviors, or *agents*. In traditional behavior-based robot design, both the individual behaviors and their modes of interaction (e.g., arbitration priorities, patterns of inhibition, etc.) have to be programmed manually. The main goal of our current research is to consider alternative automated methods for training the separate behavioral modules including their interactions in a multi-agent architecture. In particular, we are examining a set of co-evolutionary designs for alternative decompositions of a single task. Each of these designs has been implemented and tested, and the results have been analyzed to obtain a set of conditions under which various methods of co-evolutionary are likely to prove useful, as well as conditions that may lead to failure. This effort provides valuable information for future designs of co-evolutionary approaches to the design of multi-agent systems.

Send correspondence to schultz@aic.nrl.navy.mil. Proc. of the SPIE Int. Symp on Intelligent Systems and Advanced Manufacturing (ISAM '99), 19-22 Sept 1999, Boston MA.

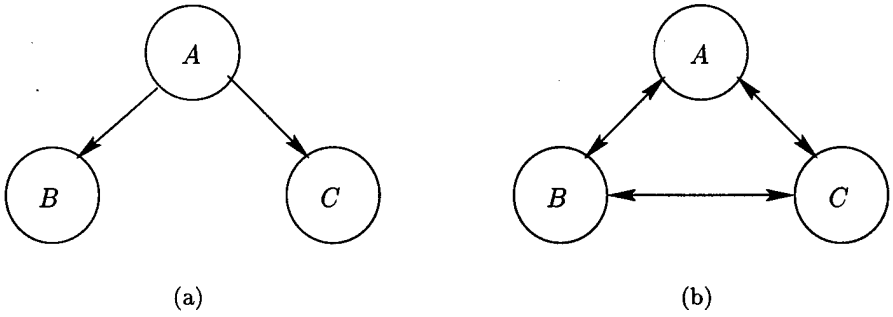


Figure 1. Some modes of agent interactions

In this paper, we focus on dependencies among the learning agents, that is, what a given learning agent needs to know about other agents in the system. By using domain knowledge, it is possible to reduce or eliminate interactions among the agents, thereby reducing the effort required to co-evolve these agents as well as reducing the impediments to learning caused by these interactions.

1.1. Behavior Engineering

Building on previous work in behavior-based robotics,[1–3] Dorrigo and Colombetti[4] outline an approach called *behavior engineering* in which machine learning techniques play a central role in increasing the behavioral quality of autonomous agents. In brief, the approach follows the following outline:

1. Complex behaviors are specified in terms of simpler component behaviors.
2. Each component behavior is mapped into a single behavioral unit comprising a learning module.
3. The agent is trained on the component behaviors according to a *shaping policy.*

How to choose the proper decomposition into component behaviors is, of course, of primary concern in this approach, but it is not the only critical concern. Even for problems (such as the ones we consider in this paper) where the "natural" or "obvious" decomposition will succeed, just how the agents interact with each other will have a tremendous impact on the success of the overall learning problem. Next, we briefly outline some of these issues arising in co-evolving autonomous systems.

1.2. Interactions Among Co-evolving Agents

In the behavior engineering paradigm, the agents perform behaviors with pre-defined roles within the system. Agents might perform tasks at a variety of levels of deliberation and abstraction, interacting with other agents as needed. For example, a multi-agent mobile robot might include one agent that performs navigation, one agent that performs collision avoidance and another agent that performs higher-level executive planning. Additionally, these agents may have vastly differing rates of decision making. The agent that performs collision avoidance would be much more frequently active than, say, the one that chooses the next destination.

In a non-learning multi-agent system, each agent can be provided with a fixed set of rules for interacting with other agents. In a multi-agent system in which the agents learn, the situation is much more complex since the capabilities of the learning agents presumably change over time. Let's briefly consider the various modes of interaction among learning agents. This discussion focuses on *dependencies among learning agents*, that is, what a given learning agent needs to know about other agents in the system. In the following figures, an arrow $A \rightarrow B$ (read, "A depends on B") indicates that learning agent A needs to include a model of agent B in its learning environment. In other words, A can't learn to perform its task without some knowledge of how B performs its own task. We will give examples from the field of autonomous mobile robots.

Two of the most common dependencies among agents are shown in Figure 1. Here, agent A is an executive decision maker that invokes subordinate agents B and C. If the executive is a learning agent, it needs a model of all subordinates in order to evaluate its own policies. What do the subordinates need to know about each other?

Figure 1(a) corresponds to the case in which the subordinates perform mutually exclusive behaviors. For example, agent B might perform the behavior *"Track_another_robot"* and agent C might perform the behavior *"Dock_at_the_refueling_station"*. The executive decides when to perform each task, perhaps based on considerations such as the urgency of knowing the other robot position and the current fuel level. In the case of such mutually exclusive agents, the two subordinate agents can learn their own tasks without a model of their sibling task(s). For example, in the training environment used by the two sibling agents, the tracking agent is always required to track and the docking agent is always required to dock. The individual agents need not model how their siblings behave, since that is irrelevant to the performance of their own subtask. In addition, each subordinate agent can be trained using a trivial model of the executive agent, in which the executive always chooses the learning agent. In this particular instance we refer to the sibling agents' independence of the remaining agents as learning *off-line*.

Consider the situation in Figure 1(b) where all three agents depend mutually on one another to solve a particular task. This is the most general relationship among three agents and is the one that would be presumed in the absence of any specific knowledge regarding agent interactions. For example, let the agents be as described above for Figure 1(a), where we "ignore" the actual interactions and use those of Figure 1(b) instead. The sibling agents are no longer independent of the other agents, and cannot use the specialized (*off-line*) executive agent. What difference can it make? Firstly, such an executive agent is desirable not only from the viewpoint of simplicity but also with regard to how much experience it provides to the learning agent. This raises the very important point that, particularly for relationships such as the executive-subordinate one, an agent's ability to learn can be affected by how often it is activated by another agent. For example, if in the course of learning its own task the executive loses confidence in the ability of agent B to dock, it may simply decide to stop docking altogether, thereby *starving* agent B of necessary learning experience.

The preceding suggests that there is a partial ordering of the interaction graphs according to the availability (or use) of domain knowledge, where additional knowledge is used to eliminate, where possible, interactions. Thus, 1(a) can be viewed as a refinement of 1(b).

We will conclude this section with a review of related work. In the following sections, we will describe the SAMUEL learning agent, method of coevolution used in this paper, the performance task and the learning methodology, and then present simulation studies and results.

1.3. Related Work

Our approach adopts a model of evolving agents based on SAMUEL, a system that learns policies expressed as sets of decision rules using genetic algorithms.[8] SAMUEL has been used successfully in many reinforcement learning applications. Schultz and Grefenstette,[9,10] used SAMUEL to learn collision avoidance and local navigation behaviors for a Nomad 200 mobile robot. Schultz and Grefenstette,[11] also used SAMUEL to evolve herding behaviors. In this task, the learning robot was required to herd a target robot to a "pasture".

The topic of co-evolutionary systems has received increased attention lately. Rosen and Belew[12] have proposed novel co-evolutionary methods for two player games. Bull, Fogerty and Snaith[13] also have studied different regimes for cooperative tasks. This study follows up on these studies, comparing a larger number of co-evolutionary regimes on a more complex multi-agent task, using a genetic algorithm that learns symbolic reactive rules (SAMUEL).

Potter[14,15] has investigated a model of collaborative cooperative co-evolution. In this system, problem decomposition was an emergent property of the system, rather than a designed feature as in the present study. Individuals from multiple co-evolving "species" were evaluated in the context of the current best individuals from each of the other species. Individuals were rewarded based on how well they cooperated with representatives of other species. On a variety of tasks including parameter learning, string matching, neural network design and concept learning, Potter found that the cooperative co-evolution approach was able to discover important environmental niches and that subcomponents with the appropriate level of generality emerged to cover the available niches. Furthermore, the co-evolving subcomponents were able to adapt to changing fitness landscapes. Finally, the addition of dynamic species creation and extinction events resulted in the emergence of an appropriate number of subcomponents for the problem being solved. Later in this paper, we consider an approach based on cooperative co-evolution, called asynchronous co-evolution.

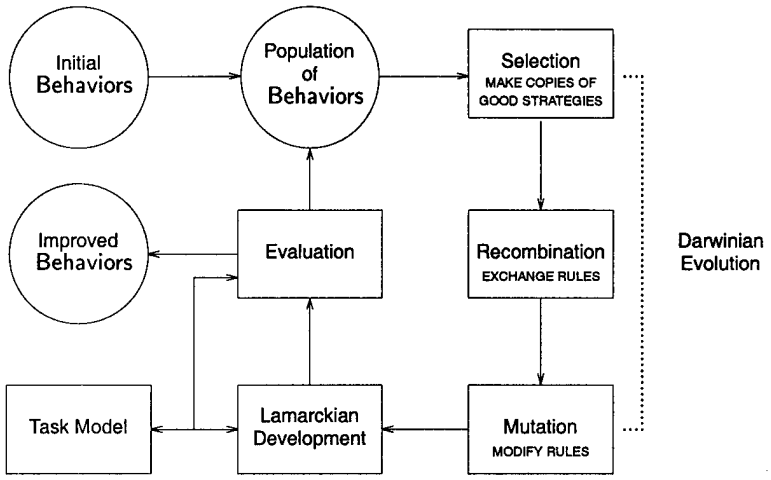


Figure 2. SAMUEL: An Evolutionary Learning System.

2. THE SAMUEL LEARNING AGENT

We adopt a model of evolving agents based on SAMUEL, a system that learns policies expressed as sets of decision rules using genetic algorithms. What follows is a brief description of SAMUEL and further details can be found in.[8] SAMUEL maintains a population of competing policies, as shown in Figure 2. Policies are evaluated by running a simulation of the task environment in which the learning agent uses the given policy to perform its task. Each attempt to perform a task is called an *episode.* After each episode, an automated *critic* provides an overall assessment that scores the performance of the learning agent on the given task. We are especially interested in complex environments in which the learning agent must deal with a variety of conditions, including noisy sensors, imprecise actuators and other independent agents. To reflect this environmental complexity, we typically average the performance of a given policy over many learning episodes, changing the environmental conditions for each episode.

Once all policies in the population have been evaluated, SAMUEL generates a new population of policies in three steps, as shown in Figure 2. First, policies are selected for reproduction based on their observed fitness. Second, the selected policies are recombined by exchanging rules between parent policies. Finally, changes (mutations) are made to generate new rules that are slight variations or combinations of existing rules. Rules that are active during high performance episodes are exchanged as a group, in order to promote the spread of effective behaviors throughout the population.

The input to SAMUEL agents are called *sensors* and their control variables are called *actions.* A policy is a set of decision rules of the form:

IF c_1 AND ... AND c_n THEN SET a_1 AND ... AND a_m

where each c_i is a condition on one of the sensors and each action a_j specifies a setting for one of the control variables. During each decision cycle, the sensors determine which rules in the policies match the current state. All matching rules *bid* for the action values specified on their right-hand-sides. Conflicts are resolved based on the *strength* of the bidding rules, where strength is a numeric measure of the past performance of the rule.

SAMUEL uses a single-chromosome, rule-based representation for policies, that is, each member of the population is a policy represented as a rule set and each gene is a rule that maps the state of the world to actions to be performed. An example rule (gene) might be:

IF $range = [35, 45]$ AND $bearing = [0, 45]$ THEN SET $turn = 16$ ($strength$ 0.8)

This rule would match the current state of the sensors if the range were between 35 and 45 inches inclusively, and the bearing were between 0 and 45 degrees inclusively, and it would recommend a turning rate of 16 degrees.

In addition to the usual genetic operators of crossover and mutation, SAMUEL uses more traditional machine learning techniques in the form of Lamarckian operators which create additional altered rules based on an agent's experiences.[16]

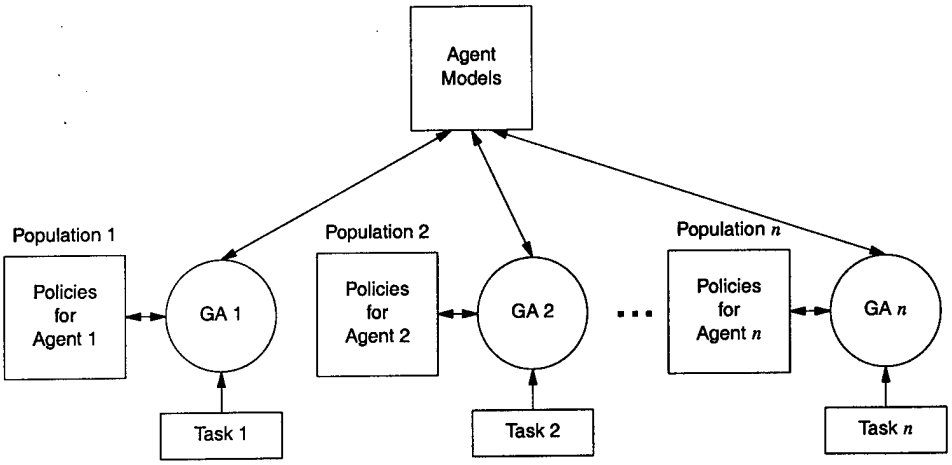


Figure 3. Asynchronous Co-evolution Model. All GAs run asynchronously. Each GA uses a model of other learning agents that can be updated asynchronously by the other GAs.

3. METHODS OF CO-EVOLUTION

There are many methodologies in training separate behavior modules in a multi-agent architecture. In this paper we will use a particular method, *asynchronous co-evolution*, in order to study the use of domain knowledge to reduce interactions between behaviors and thereby to improve the ability to co-evolve multiple behaviors.

In *Asynchronous Co-evolution,* all learning agents evolve simultaneously (Figure 3). Each learning agent has a separate genetic algorithm. Within a given agent's genetic algorithm, the learning agent is evaluated in the context of a simulation model which includes the other agents as part of the learning environment. Alternative regimes for asynchronous co-evolution are distinguished by their strategies for updating the models of external agents, i.e., what each sees of the other agents' behaviors. For example, if agent A and agent B are both evolving, agent A might use a model for agent B based on (i) B's current policy (being evaluated in B's genetic algorithm), (ii) B's best-so-far policy, (iii) a policy selected at random from a collection of B's best policies, etc. (See Section 5).

4. THE PERFORMANCE TASK: TRACKING UNDER FUEL CONSTRAINTS

Our case study focuses on a particular tracking task, one that reflects several plausible constraints likely to be encountered by real autonomous vehicles. In this task, an autonomous mobile robot, or *the Robot*, is assigned to stay within observation distance of another mobile agent, *the Target* for a prolonged period of time, as shown in Figure 4. The Target performs a random walk combined with obstacle avoidance. The Robot's performance is measured by its mean squared error from the ideal tracking distance. It is considered a mission failure if the Robot collides with the Target or with the walls surrounding the tracking area.

The Robot must also deal with fuel constraints. The Robot has a fixed maximum fuel capacity. The amount of fuel consumed increases linearly with the Robot's speed and turning rate. If the Robot runs out of fuel, it must be towed back to the dock and refilled. Since the Robot's performance is based on the time spent successfully tracking the Target, the time spent being towed significantly decreases the overall performance. Fortunately, the Robot may obtain additional fuel by docking at the fuel station. In order to dock successfully, the robot must approach the front of the dock from not more than 15 degrees to either side, and at a speed no greater than 4 inches per second (ips). If docking is successful, the Robot's fuel tank is refilled and it can continue tracking the Target.

For this task, five abstract sensors are defined for the Robot: the range and bearing from it to the nearest object, the Target's heading with respect to the Target's bearing, and the range and bearing from the Robot to the Target. These abstract sensors are derived from information available through the actual sensors on the Nomad 200 mobile robots. The abstract sensor values are all discretized. The bearing values are partitioned into intervals of five degrees. The range is partitioned into intervals of five inches from 0 to 150 inches. The heading is partitioned into 45 degree segments. The learned actions are velocity mode commands for controlling the translational rate and the steering rate. The translation is given as -4 to 16 inches/sec in 4 inch/sec intervals. The steering command is given in intervals of 4 degrees/sec from -24 to 24 degrees/sec. So, a typical rule for the learning robot might be:

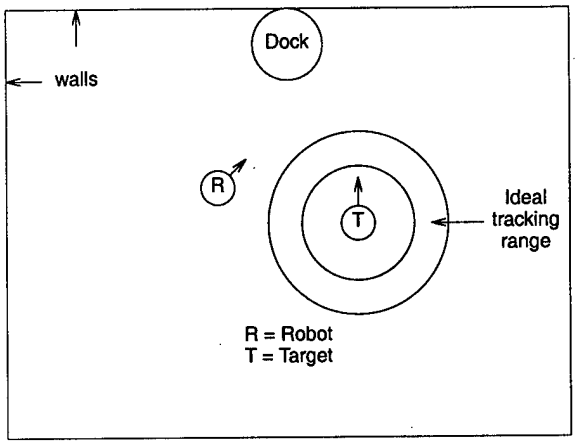


Figure 4. The Tracking Task with Fuel Constraints.

IF range = [35, 45] AND bearing = [340, 35] THEN SET turn = -24 (Strength 0.8)

To maximize the tracking time, a good strategy for the Robot would be something like:

Repeat forever:

1. Acquire tracking on the Target as quickly as possible
2. Continue tracking until gas-tank is nearly empty
3. Quickly move to dock and refuel

4.1. Task Decomposition

In order to accomplish this behavior we provide the Robot with one or more internal *agents*, each of which learns a specific part of the behavior. Although our research is considering different behavior architectures for the Robot, in this paper, we consider only the following architecture:

Two Level Behavior Architecture

The Robot has three learning agents:

`Tracking Task:`
Initiate and continue tracking on Target.

`Docking Task:`
Move to dock and refuel.

`Executive Task:`
Arbitrate between `Tracking Task` and `Docking Task`.

This is an instance of a two-level *switch* hierarchical architecture. The Executive is the high-level agent that decides which lower level subtask to perform. Figure 1(a) describes the agent interactions in this architecture (see Section 1.2 for a discussion of this).

5. LEARNING METHODOLOGY

The decomposition of the performance task in the previous section represents one dimension of the design problem for learning in autonomous robots. The second dimension is the method employed for the co-evolution of each of the resulting subtasks, which was described in Section 3. The third dimension is the model of dependencies among the behaviors.

Two experimental regimes are studied in this paper, asynchronous coevolution with complete dependencies, and asynchronous co-evolution with domain knowledge limited dependencies. For each learning regime we performed 10 simulation runs and the graphs associated with each regime represent the *average* over these 10 experiments.

5.1. Asynchronous Co-evolution

In the asynchronous co-evolution method, the `Tracking Task` and the `Docking Task` evolve independently in isolation but concurrently, and the `Executive Task` evolves using the (best) rules which have been evolved for the `Tracking Task` and the `Docking Task` so far.

Regime Mutual:

All tasks evolve asynchronously using the best rules evolved so far for the other tasks. This regime is inherently non-repeatable since the actual course of evolution for the `Executive Task` depends on CPU scheduling for the `Tracking Task` and the `Docking Task`. These experiments required three separate GAs to be running (one for each task to be learned) simultaneously. Each GA would report its best-so-far policy to the two other GAs. Thus, the `Tracking Task` would learn in an environment where the Executive and Docking policies used were the best-so-far produced by their respective GAs, and similarly for the environment of the `Docking Task` and the `Executive Task`. Figure 1(b) depicts the presumed agent interactions for this regime.

Regime Independent:

In this regime, the `Tracking Task` and `Docking Task` were independent and subordinate to the `Executive Task`. Figure 1(a) depicts the presumed agent interactions for this regime. Because the tasks were independent of each other and subordinate to the `Executive Task`, they could be first learned offline which would allow us to simulate various rates of asynchrony during the learning of the `Executive Task`. We found no appreciable differences in the rate of asynchrony.

Note that the essential difference between these two regimes is in the presumed agent interactions and dependencies. In Regime Mutual, the best-so-far Executive behavior is used during the evolution of the `Tracking Task` and the `Docking Task`, and in Regime Independent the `Tracking Task` and the `Docking Task` each evolve using an Executive agent which always selects its task, i.e., the `Tracking Task` and the `Docking Task` learn off-line (see Section 1.2).

6. SIMULATION STUDIES

We proceed now to a more detailed description of the simulation experiments including the fitness functions used, the experimental parameters, initial behaviors of the agents, and the performance measures used to evaluate the experiments. In order to understand some of these details it is necessary to briefly describe the *timescales* involved in the experimental framework. The basic time unit is a *step* (which on the real robot amounts to about .5 sec.). At each step whatever agent is controlling the robot will set its turning rate and its speed, thereby moving the robot in its environment. The basic time unit of evaluation is an *episode*, which can last a maximum of 60 steps (or approx. 30 sec.). Episodes can be terminated before this if the Robot collides with another object (the Target, a wall, or the Dock), or if it runs out of gas, or if it docks successfully. The Exec will decide at the beginning of each episode whether to Track or to Dock. Thus, we see that the `Executive Task` has a different time granularity than either the `Docking Task` or the `Tracking Task`, since it makes one decision *per episode* instead of one decision *per step*. The last time division is an *epoch*, which can last a maximum of 10 episodes, and is meant to capture the do-forever nature of the surveillance task. Epochs can be terminated prematurely by Robot collisions or out-of-gas, but not by successful docking. Finally, at the beginning of each epoch the Robot and Target are positioned randomly within the environment, and otherwise at the beginning of each episode they retain the position they had at the end of the previous episode.

6.1. Fitness Functions for Individual Tasks

The `Tracking Task` and the `Docking Task` receive a payoff at the end of every episode. The `Executive Task` receives a payoff at the end of every epoch, where if the epoch is terminated prematurely a "towing" time penalty is assessed which is added to the total time of an epoch (as explained in Section 4);

`Docking Task:`

if successful dock, then payoff $= .5 + .5 * (m/t)$, where m is the estimated minimum time to dock and t is the time Robot took to dock.

if unsuccessful dock, then payoff $= 0$.

(I.e., maximum payoff is received for minimum time docking.)

`Tracking Task:`

if Robot collides with another object, then payoff = 0, otherwise payoff = % of total time during episode spent tracking Target.

`Executive Task:`

payoff = % of total time during epoch spent tracking Target.
(I.e., payoff = $t/(e+p)$, where t is the total steps spent tracking, and e is the number of steps in a standard epoch $(10 * 60)$, and p is the assessed towing time penalty).

6.2. Simulation Parameters

All major simulation parameters were held fixed for all simulation runs.

- Generations = 50
- Population size = 50
- Individual evaluations (episodes) = 90
- Best individual evaluations (episodes) = 120, which is used for the learning curve graphs (see Section 6.3).
- Initial gas supply = 50%
- Dimension of environment = (300 inches × 300 inches)
- Tracking range = [40 inches, 60 inches]
- Towing penalty = 50.

6.3. Performance Measures

We have chosen some specific measurements and statistics to assess how well each regime (and simulation experiment) performed the overall task and to ascertain the reasons why some experiments failed to evolve an acceptable behavior.

In order to properly understand the experimental results and their interpretation it is necessary to consider the following: The Robot's energy consumption is such that if it were to remain motionless it could survive approximately eight episodes before it ran out of energy. Even while it is moving slowing it can last up to four episodes. Once the Robot has acquired the Target, it can usually maintain a track with only minimal movement (and hence energy consumption). Since an epoch (or lifetime) is 10 episodes, the Robot must dock twice each epoch to achieve good combined performance. Thus, the ideal ratio of tracking to docking is roughly 80% to 20%.

With this information we can now define the following measurements for regimes and individual experiments:

Learning Curve

This is the standard measurement of *fitness* of the best individual for each generation. Since the `Executive Task` receive no credit while docking, the maximum fitness for these should be roughly 80%.

Quality Tracking

This is the percentage of episodes in which the Robot was within tracking distance of the Target for 80% of the steps of that episode. In other words, these are episodes where there is a high level of tracking. Normally after docking, one episode is required to reacquire the Target and four more episodes of tracking can occur before the Robot must dock again, so a good tracker would score roughly 80% according to this measure.

Successful Docking

This is the percentage of episodes in which the `Executive Task` selected Docking as the objective and the Robot docked successfully.

Average Epoch Length

This is the average number of episodes in an epoch (or lifetime). It provides us with a good measure of how well the docking behavior is integrated into the Robot's overall behavior. By the remarks above we see that this ranges between a minimum of 4 episodes (no docking) and 10 episodes (good docking).

Regime	Good Docking			Poor Docking			No Docking		
	Tracking			Tracking			Tracking		
	Hi	Med	Lo	Hi	Med	Lo	Hi	Med	Lo
Mutual		1					4	3	2
Independent	5	5							

Table 1. Qualitative assessment of individual experiments for the two-level task decomposition. Table entries are the *number* of experiments with the specified criteria. Criteria are: *Good Docking* $\approx$ 20% episodes with successful docking; *Poor Docking* $\ll$ 20% episodes with successful docking; *No Docking* $<$ 5% selection of docking task; *Hi Tracking* $\geq$ 60% quality tracking; *Med Tracking* $\geq$ 30% quality tracking; *Lo Tracking* $<$ 30% quality tracking.

Finally, since the goal of this performance task is for the Robot to track (and dock) indefinitely, the real world performance gap between those regimes in which docking was intergrated into the Robot's behavior and those in which it wasn't can be expected to be much larger than what we have obtained here with an artificial lifetime of only 10 episodes.

7. RESULTS

In the asynchronous approach each evolving task informs the others when it discovers an improved behavior so that the others can use this improved behavior in their world model. We begin with the Mutual regime which involves the least amount of domain knowledge (regarding agent interactions).

Table 1 lists the qualitative assessment for all the individual experiments. Table entries are the number of experiments of the specified classification. These assessments are based on the following criteria:

Good Docking $\approx$ 20% episodes with successful docking

Poor Docking $\ll$ 20% episodes with successful docking

No Docking $<$ 5% selection of docking task (i.e., starvation of the `Docking Task`)

Hi Tracking $\geq$ 60% quality tracking

Med Tracking $\geq$ 30% quality tracking

Lo Tracking $<$ 30% quality tracking

Med Tracking should be regarded as acceptable, albeit mediocre, performance.

7.1. Regime Mutual

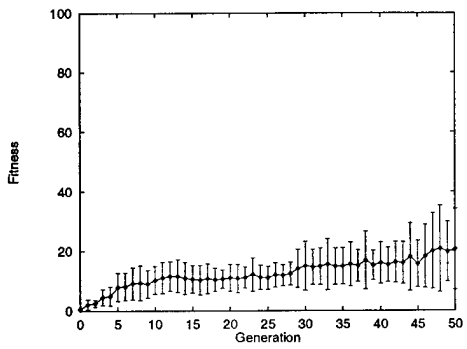
Regime Mutual presumes that each task depends on all other tasks (see Figure 1(b)) so that during its coevolution it must incorporate the best behavior discovered so far for all the other tasks. Figure 5 shows the results for this regime.

This regime proved to be unsuccessful in coevolving acceptable performance. For 9 out of the 10 experiments conducted the `Executive Task` never selected Docking as the objective, and in the one experiment where docking was integrated into the Robot's behavior the tacking quality was not high. Only 4 of the experiments showed good tracking quality (and showed no docking). Since the coevolution of the `Docking Task` relies on using the best-so-far `Executive Task`, if that task never selects Docking as the objective, then there is no possibility for the `Docking Task` to improve its performance. In essence, it is starved for experience by an `Executive Task` which ignores it. In the one experiment where it did learn to dock, the learning did not begin until half way through its evolution.

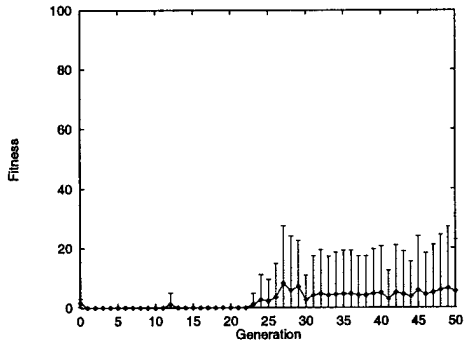
7.2. Regime Independent

The asynchronous approach represented by the Independent regime uses domain knowledge regarding the task dependencies (see Figure 1(a)), viz., the `Docking Task` and `Tracking Task` are coevolved "offline" using special executive behaviors which always select them as the objective task.

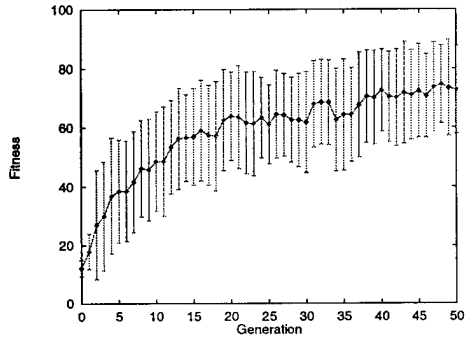
Figure 6 shows the learning curves for these experiments. The results demonstrate the importance of incorporating domain knowledge into the structure of the coevolution process. Recall, that the main difference between the Independent and Mutual regimes is the agent interactions (cf. Figure 1(a) vs Figure 1(b)).



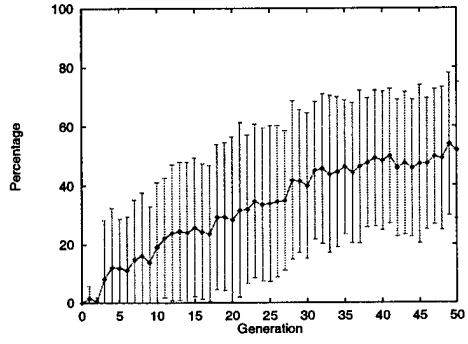
(a) Executive Task Learning Curve



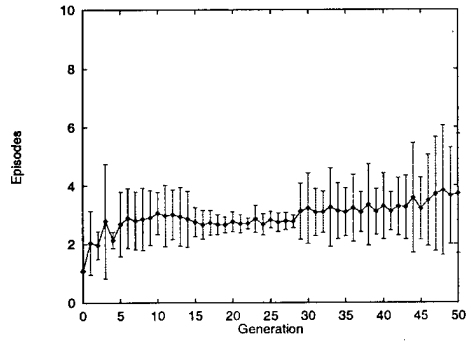
(b) Docking Task Learning Curve



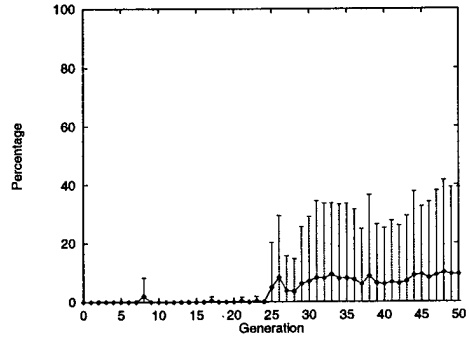
(c) Tracking Task Learning Curve



(d) Quality Tracking

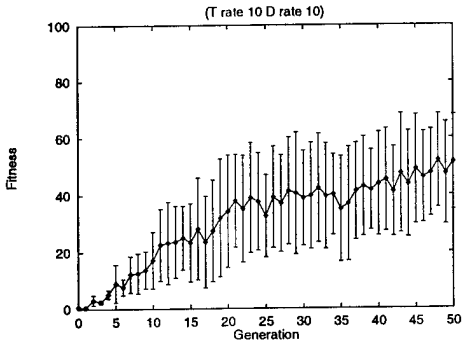


(e) Average Epoch Length

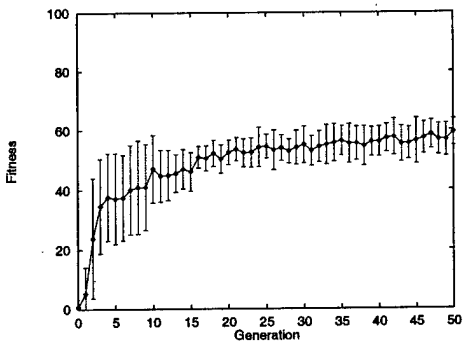


(f) Successful Docking

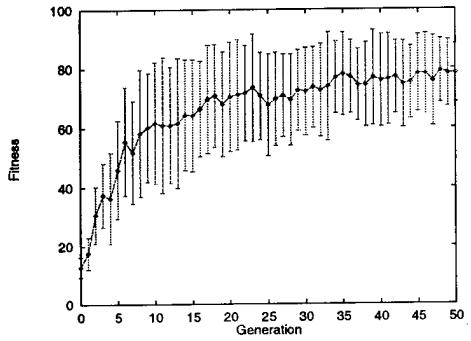
Figure 5. Mutual Regime Learning and Performance



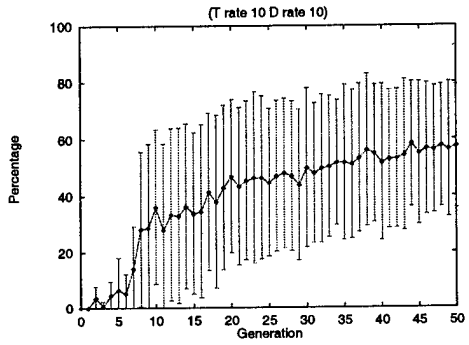
(a) **Executive Task** Learning Curve



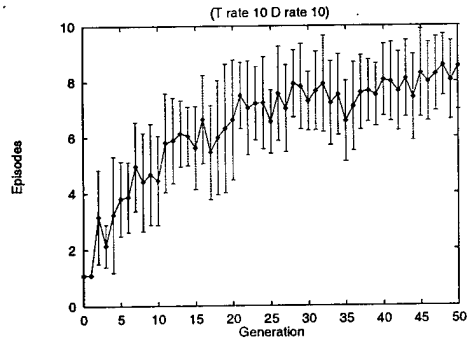
(b) **Docking Task** Learning Curve



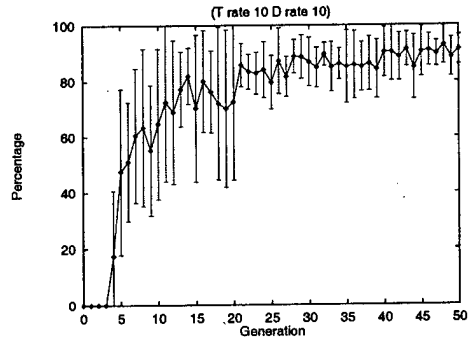
(c) **Tracking Task** Learning Curve



(d) Quality Tracking



(e) Average Epoch Length



(f) Successful Docking

Figure 6. Independent Regime Learning and Performance

8. DISCUSSION

In this paper we have explored the application of co-evolutionary learning techniques to a learning problem for mobile robots. We chose the "natural" decomposition of the problem into several learning subproblems. However, in addition to the right decomposition we showed that discerning the right relationships among the agents for these learning problems was also necessary. Specifically, we showed that the Mutual regime failed to solve the learning problem primarily because if failed to integrate the docking subtask. We conjecture/speculate that this was due to the fact that the `Executive Task` starved the `Docking Task` of the necessary learning trials, thereby preventing it from learning how to dock. The good news is that we have provided a co-evolutionary learning method (viz., the Independent regime) which solves the learning problem satisfactorily and efficiently.

ACKNOWLEDGMENTS

The work reported here was supported in part by The Office of Naval Research.

REFERENCES

1. R. A. Brooks, "Intelligence without representation," *Artificial Intelligence* **47**, pp. 139–159, 1991.
2. A. Ram, R. Arkin, G. Boone, and M. Pearce, "Using genetic algorithms to learn reactive control parameters for autonomous robot navigation," *Adaptive Behavior* **2**(3), pp. 277–304, 1994.
3. M. Mataric, "Integration of representation into goal-driven behavior-based robots," *IEEE Transactions on Robotics and Automation* 8(3), pp. 304–312, 1992.
4. M. Dorrigo and M. Colombetti, *Robot Shaping: An Experiment in Behavior Engineering*, MIT Press, Cambridge, MA, 1998.
5. J. H. Holland and J. S. Reitman, "Cognitive systems based on adaptive algorithms," in *Pattern-directed Inference Systems*, Academic Press, New York, 1978.
6. J. H. Holland, "Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems," in *Machine Learning: An Artificial Intelligence Approach*, vol. 2, Morgan Kaufmann, Los Altos, CA, 1986.
7. S. W. Wilson, "ZCS: A zeroth level classifier system," *Evolutionary Computation* **2**(1), pp. 1–18, 1994.
8. J. J. Grefenstette, C. L. Ramsey, and A. C. Schultz, "Learning sequential decision rules using simulation models and competition," *Machine Learning* **5**(4), pp. 355–381, 1990.
9. A. C. Schultz and J. J. Grefenstette, "Using a genetic algorithm to learn behaviors for autonomous vehicles," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, (Hilton Head, SC), 1992.
10. A. C. Schultz, "Learning robot behaviors using genetic algorithms," in *Intelligent Automation and Soft Computing: Trends in Research, Development, and Applications*, pp. 607–612, TSI Press, Albuquerque, 1994.
11. A. C. Schultz and J. J. Grefenstette, "Robo-shepherd: Learning complex robotic behaviors," in *Robotics and Manufacturing: Recent Trends in Research and Applications, Volume 6*, pp. 763–768, ASME Press, New York, 1996.
12. C. Rosen and R. Belew, "Methods for competitive co-evolution: Finding opponents worth beating," in *Proc. Sixth International Conference on Genetic Algorithms*, pp. 373–380, 1995.
13. L. Bull, T. Fogarty, and M. Snaith, "Evolution in multi-agent systems: Evolving communicating classifier systems for gait in a quadrupedal robot," in *Proc. Sixth International Conference on Genetic Algorithms*, pp. 382–388, 1995.
14. M. A. Potter, K. A. DeJong, and J. J. Grefenstette, "A coevolutionary approach to learning sequential decision rules," in *Proc. Sixth International Conference on Genetic Algorithms*, pp. 366–372, 1995.
15. M. A. Potter, *Cooperative Coevolution: A Collaborative Approach to Solving Decomposable Problems.* Ph. D. Dissertation, School of Information Technology and Engineering, George Mason University, Fairfax, VA, USA, 1997.
16. J. J. Grefenstette, "Lamarckian learning in multi-agent environments," in *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 303–310, Morgan Kaufman, (San Diego, CA), 1991.

APPENDIX B:

Continuous and Embedded Learning in Autonomous Vehicles: Adapting to Sensor Failures

Alan C. Schultz[a], and John J. Grefenstette[b]

[a]Intelligent Systems Section, Naval Research Laboratory, Washington DC, USA

[b]Institute for Biosciences, Bioinformatics and Biotechnology, George Mason University, Fairfax VA, USA

ABSTRACT

This project describes an approach to creating autonomous systems that can continue to learn throughout their lives, that is, to be adaptive to changes in the environment and in their own capabilities. Evolutionary learning methods have been found to be useful in several areas in the development of autonomous vehicles. In our research, evolutionary algorithms are used to explore alternative robot behaviors within a simulation model as a way of reducing the overall knowledge engineering effort. The learned behaviors are then tested in the actual robot and the results compared. Initial research demonstrated the ability to learn reasonable complex robot behaviors such as herding, and navigation and collision avoidance using this offline learning approach. In this work, the vehicle is always exploring different strategies via an internal simulation model; the simulation, in turn, is changing over time to better match the world.

This model, which we call *Continuous and Embedded Learning* (also referred to as *Anytime Learning*), is a general approach to continuous learning in a changing environment. The agent's learning module continuously tests new strategies against a simulation model of the task environment, and dynamically updates the knowledge base used by the agent on the basis of the results. The execution module controls the agent's interaction with the environment, and includes a monitor that can dynamically modify the simulation model based on its observations of the environment. When the simulation model is modified, the learning process continues on the modified model. The learning system is assumed to operate indefinitely, and the execution system uses the results of learning as they become available. Early experimental studies demonstrate a robot that can learn to adapt to failures in its sonar sensors.

Keywords: Robotics, Learning, Genetic Algorithms, adaptation, Evolutionary Algorithms

1. INTRODUCTION

An important problem arising in robots that are expected to perform autonomously for extended periods is how to adapt the robot's rules of behavior in response to unexpected changes in its own capabilities. For example, suppose the robot periodically checks its sensors and its ability to perform basic actions. If it finds that some sensors or actions are no longer available, perhaps due to a problem with the robot's hardware or due to some undetected environmental cause, then it must learn new rules for performing its mission that use whatever remaining capabilities are still available.

We have developed an approach to this problem that we call *Anytime Learning*,[1-3] or more recently, *Continuous and Embedded Learning (CEL)*. In this approach, the robot interacts both with the external environment and with an internal simulation. The robot's execution module controls the robot's interaction with the environment, and includes a monitor that dynamically modifies the robot's internal simulation model based on the monitor's observations of the actual robot and the sensed environment. The robot's learning module continuously tests new strategies for the robot against the simulation model, using a genetic algorithm[4] to evolve improved strategies, and updates the knowledge base used by the execution module with the best available results. Whenever the simulation model is modified due to some observed change in the robot or the environment, the genetic algorithm is restarted on the modified model. The learning system operates indefinitely, and the execution system uses the results of learning as they become available.

Send correspondence to schultz@aic.nrl.navy.mil. Appeared in Unmanned Ground Vehicle Technology II, (Eds. Grant R. Gerhart, Robert W. Gunderson, Chuck M. Shoemaker), Proceedings of SPIE Vol.4024, pg. 55-62, 2000.

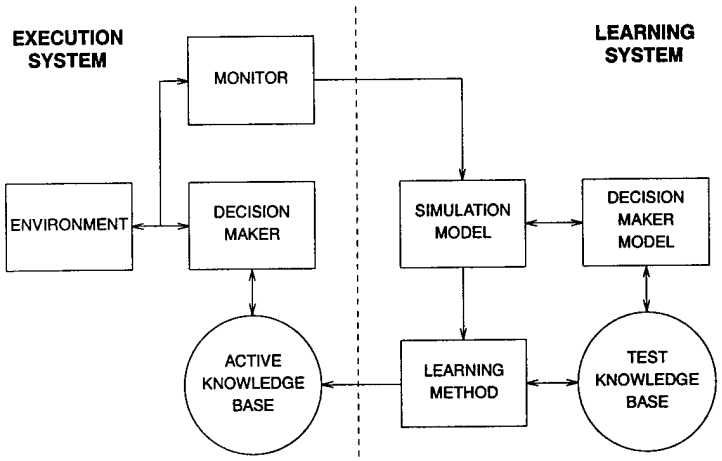


Figure 1. The Continuous and Embedded Learning Model

In this paper, we examine the CEL model with respect to sensor failures, specifically, how the robot can learn to adapt to failures in its sensor capabilities over time. We show that a robot adapt to the partial loss of its sensors and learn to use different sensors to continue to perform a door traversal task. A simulation study and execution on an actual mobile robot shows that the approach yields effective adaptation to a variety of partial sensor failures.

In the next section, we will describe the Continuous and Embedded Learning model. Section 3 discusses related work. In Section 4 we will describe the task domain. This is followed by description of the modules of the CEL model, and simulation and experimental results showing the adaptation of a robot to partial sensor failues.

2. CONTINUOUS AND EMBEDDED LEARNING

The Continuous and Embedded Learning model addresses the problem of adapting a robot's behavior in response to changes in its operating environment and its capabilities. The outline of the approach is shown in Figure 1.

There are two main modules in the CEL model. The *execution module* controls the robot's interaction with its environment. The *learning module* continuously tests new strategies for the robot against a simulation model of the environment. When the learning module discovers a new strategy that, based on simulation runs, appears to be likely to improve the robot's performance, it updates the rules used by the execution module. The execution module includes a *monitor* that measures aspects of the operational environment and the robot's own capabilities, and dynamically modifies the robot's internal simulation model based on these observations. When the monitor modifies the simulation because of an environmental change, it notifies the learning system to restart its learning process on the new simulation.

This general architecture may be implemented using a wide variety of execution modules, learning methods, and monitors. The key characteristics of the approach are:

- Learning continues indefinitely. This is unlike most machine learning methods, which employ a training phase, followed by a performance phase in which learning is disabled. This lifetime learning is what allows the system to be adaptive after being fielded.
- The learning system experiments on a simulation model. For most real-world robotic applications, experimenting with the physical robot may be time-consuming or dangerous. Using a simulation models permits the safe use of learning methods that consider strategies that may occasionally fail.
- The simulation model is updated to reflect changes in the real robot or environment. This is a secondary type of learning of the model. While this is a major research issue in its own right, we are currently not concerned with the learning at this level, and construct the monitor and simulation as appropriate. That is, we assume that it is possible to monitor the condition of the robot's sensors and actuators. For our purposes, it is not

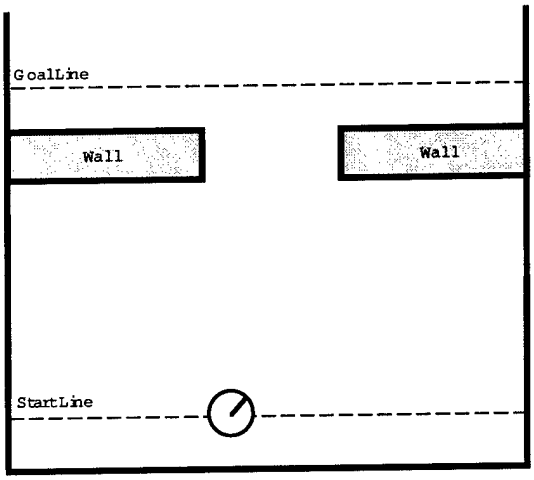


Figure 2. The door traversal task.

necessary to diagnose the cause of any detected failure, only the symptoms. We assume that the simulation has been constructed to allow the instantiation of sensor and actuator failure modes.

This final point reflects our assumption that the robot designer generally has at least partial knowledge of the robot and the environment. Knowledge that is relatively certain can be embodied in the fixed part of the simulation. Such knowledge might include certain fixed characteristics of the physical environment (e.g., gravity), as well as some aspects of the robot's design and performance. On the other hand, the robot designer should also identify those aspects of the environment and the robot's capabilities that are uncertain, and include these as changeable parts of the simulation module.

3. RELATED WORK

There has been a great deal of work done in the area of evolution of behaviors for autonomous robots. A very common approach has been in the evolution of neural controllers for robots.[5,6] An alternative approach has been the evolution of stimulus-response rules.[7]

An interesting problem has been how to add on-line adaptation to these systems in order to handle problems such as changes in the environment and to the robots capabilities. Early interesting work includes systems where both the form and the function are coevolved,[8–11] but these systems have not performed studies yet on actual robots, nor are these techniques useful for on-line performance.

Another very interesting area in the use of evolutionary algorithms on-line on real robots.[12] While this does allow a robot to be adaptive during performance, it is not clear whether this is appropriate for large robots in the real world, where trial and error experimentation can lead to damage or dangerous behavior. Our approach solves this problem by only allowing experimentation on a simulation that is internal to the robot.

4. PERFORMANCE TASK

This task requres a robot to go from one side of a room to the other, passing through an opening in a wall placed across the room, as illustrated in Figure 2.

In each trial, the robot is placed randomly along the starting line four feet in front of the back wall, facing in a randomly selected direction from -90 to 90 degrees (with 0 degrees facing the goal). The center of the front wall is located 12.5 feet from the back wall. The room is 25 feet wide. The location of the six foot opening in the front wall is also randomly selected each trial.

The robot must then reactively navigate through the opening reaching the goal line one foot beyond the wall, by learning a set of rules which map the current sensors to the actions to be performed by the robot, at a one hertz decision rate. The robot has a limited time to perform the task. Exceeding the time limit, or having a collision with any of the walls ends the current trial.

The robot used in these experiments is a Nomadic Technologies Nomad 200 Mobile Robot, a three-wheeled, synchronized-steering vehicle. The internal simulation used by the robot for learning approximated the robots sensors and effectors.

5. EXECUTION MODULE

The execution module for the robot includes a rule-based system that operates on reactive (stimulus-response) rules. A typical rule might be:

IF range = [35, 45] AND *front_sonar* < 20 AND *right_sonar* > 50 THEN SET turn = -24 (Strength 0.8)

Each decision cycle, the execution system compares the left hand side of each rule to the current sensor readings, selecting the best rule (after conflict resolution). That rule's action is then executed causing the robot to move. This is repeated until the robot succeeds or fails at the task.

In this task, the robot uses its seven front most facing sonars. Each sonar has a angular resolution of 22.5 degrees, with the front most sonar facing directly ahead, giving the robot a total sonar coverage of 157.5 degrees. We designate these sonars as *far_left*, *left*, *front_left*, *front*, *front_right*, *right*, and *far_right*. Each sonar has a maximum range of approximately 14 feet. The sonar values are all discretized; they are partitioned into intervals of 24 inches. In addition to the sonars, the robot also has a sensor that gives the range to the goal from 0 to 14 feet in 5 inch intervals, and the robots heading within the room from 0 to 359 degrees in 22.5 degree intervals.

The learned actions are velocity mode commands for controlling the translational rate and the steering rate of the robot. The translation rate is given as -4 to 10 inches/sec in 2 inch/sec intervals. The steering command is given in intervals of -10 degrees/sec from -30 to 30 degrees/sec. At each decision step, the system must choose a turning rate and steering rate based on the current sensors.

The rule *strength* is set by the learning system to estimate the quality of the rule. The execution module uses rule strengths to resolve conflicts among multiple rules that match the current sensors readings, but suggest different actions. In such cases, rules with higher strength are favored. See[13] for details.

5.1. The Monitor

In this study, the monitor periodically measures the output from the sonars, and compares them to recent readings and to the direction of motion. If the robot is moving forward, and the value of the sonar reads zero repeatedly, that particular sonar is marked as being defective. The monitor then modifies the simulation used by the learning system to replicate the failed sonar.

It is important to note that the monitor is required only to identify symptoms of problems, not the causes.

6. LEARNING MODULE

The learning module uses SAMUEL,[13] a learning program that uses genetic algorithms and other competition-based heuristics to improve its decision-making rules. Each individual in SAMUEL's genetic algorithm is an entire rule set, or strategy, for the robot. We have previously reported on using SAMUEL to learn simple robot behaviors such as navigation and collision avoidance,[14,15] robot herding,[16] and in other complex domains.

When the monitor notices a failure in any of the seven sonars, the learning module's population is re-initialized and continues with the modified simulation model. In this study, we use 50% of the population at the time of the detected failure and replace the other 50% with copies of the initial rule set (generally, go toward the goal line).

We have also used a *case-based approach* to re-initializing the population. The learning system re-initializes the population of strategies in the genetic algorithm by finding nearest neighbors from the case base consisting of previously learned strategies. Strategies in the case base are indexed by the capability list in place at the time the strategy was learned. Using previously learned strategies to initialize the population allows the system to very quickly adapt to situations that are similar to those seen before. See[2] for more details.

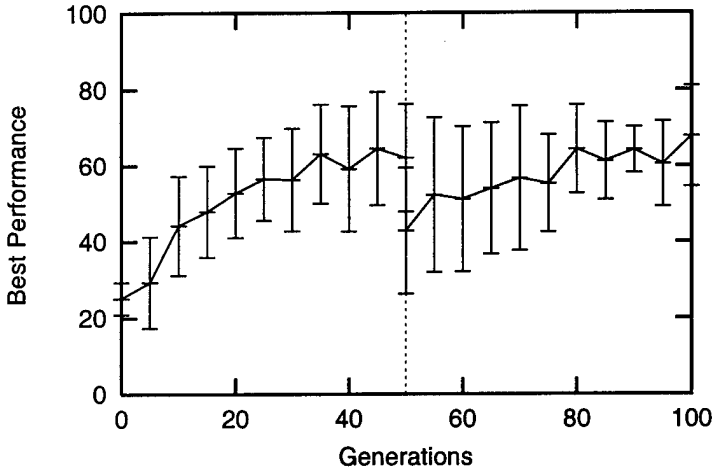


Figure 3. Learning curve showing adaptation to sensor failures occurring at generation 50. Best performance indicates the success rate for the best strategies in the current population. Results averaged over 10 runs.

7. EXPERIMENTAL METHODOLOGY

The robot begins with a set of default rules for moving toward to goal line, which give an initial success rate of 25% of getting through the doorway. These initial rules basically give the robot the direction to the goal line, but say nothing about obstacles or the walls themselves. The learning system starts with a simulation model that includeds all sonars working.

After an initial period of learning, one or more sonars are then blinded. In simulation, the failed sonar was modeled as a constant, minimum-value sonar reading; on the actual robot, one of the sonar sensors would be covered with rigid material. Once the monitor detects the failed sensor, the learning simulation is adjusted to reflect the failure, the population of competing strategies is re-initialized as described previously, and learning continues. The online robot uses the best rules discovered by the learning systems since the last change to the learning simulation model.

We initially ran an experiment with the robot adapting to a blinded *front* sonar after 50 generations, but the performance dropped a non-significant amount before continuing to improve. This is indicative of the robust rules that the SAMUEL system tends to learn. In a second experiment, we blinded the *front* sonar and the *front_right* sonar, still without a significant drop in performance. In the results reported here, three sensors were failed (*front*, *front_right*, and *right*). After the sensors were blinded, the system was allowed to continue for another 50 generations.

In these experiments we used threshold selection in the evolutionary algorithms in which the top 50% of the population reproduces (producing two offspring each). The payoff function was based on the time it took for the robot to reach the goal line. A collision resulted a a very low payoff, and exceeding the time limit resulted in a low payoff.

8. RESULTS

8.1. Quantitative Results

The experiment was repeated ten times. Figure 3 shows the average performance over time for these ten runs. The x-axis shows the generation, and the y-axis shows the external performance – the number of times out of 100 that the robot succedded in getting through the opening. The dotted vertical line at generation 50 shows where the sensor failed.

As can be seen from this learning curve, the robot initially learns to improve its performance in this task from 25% to 63% with all seven sonar sensors operating. At the beginning of generation 50, the three sonars (front, front right, and right) fail. The performance then drops to 37%, but then continues to improve as the monitor identifies the failed sonars, modifies the simulation, and learning continues with the new simulation model.

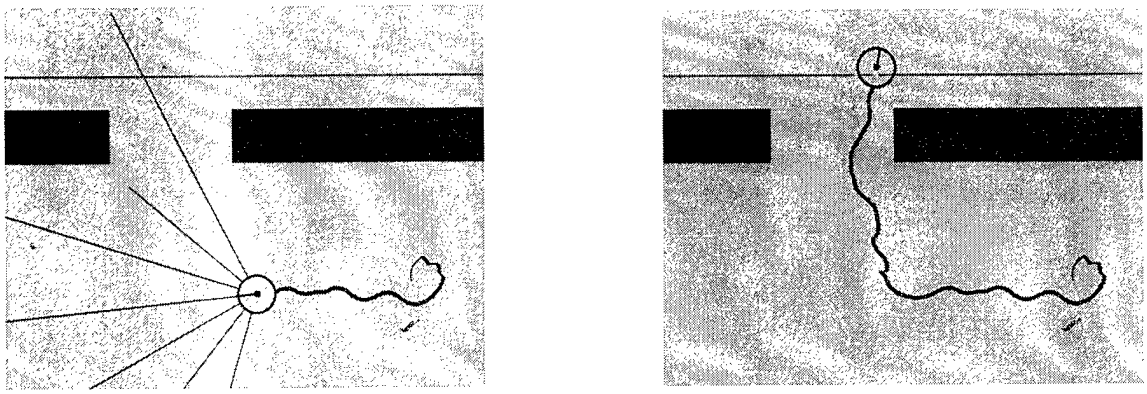


Figure 4. Robot in motion with all sensors intact, a) during run and b) at goal.

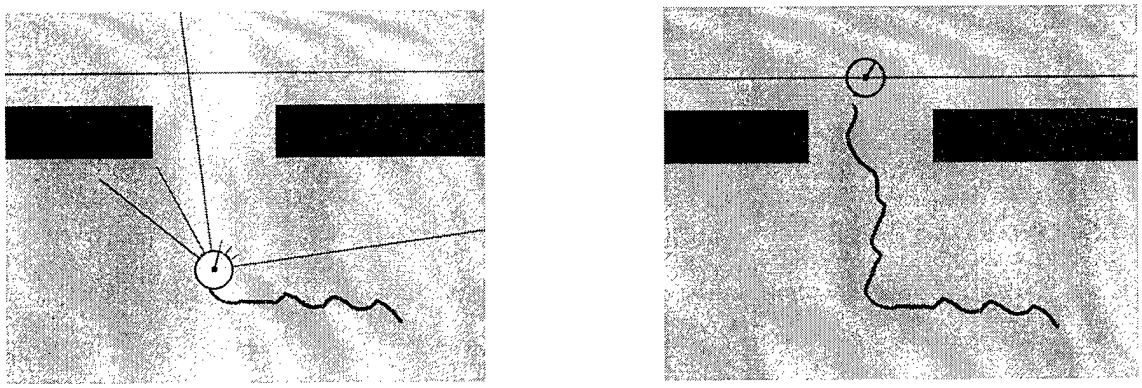


Figure 5. Robot in motion after adapting to loss of three sensors, *front*, *front_right* and *right*, a) during run, and b) at goal.

8.2. Qualitative Results

Qualitatively, we observed the following behaviors. Figure 4 shows the simulation with all of the sensors working. The picture on the left shows the simulated robot during the run, while the picture on the right shows the robot reaching the goal. In Figure 5, we see the simulated robot with three failed sonars as described above. The picture on the left shows the robot during the run, while the picture on the right shows the robot at the goal. Note that interesting behavior in the second set of pictures, as indicated by the trail showing the robot's path. The robot uses a swaying motion in order to sweep its working sensors across the space in front of the robot. This behavior allows the robot to perform the task suucessfully.

In the experimental runs we performed, when the forward facing sensors are blocked, the robot initially responds by refusing to move forward. This is fail-safe kind of response, as it allows the offline learning system to come up with a better strategy while the online robot just sits there thinking.

Figure 6 shows shots of the actual robots. The first picture shows the robot finding the opening with its front sonar, and proccedding straight at and through the opening. The second picture shows the front sonar covered to simulate its failure. The third picture shows the robot solving the task after adapting to a sonar failure. Note that it is now using a side sonar to find the opening and then turns towards the opening.

9. DISCUSSION

This work shows that the Continuous and Embedded Learning model is a promising approach to adapting to partial sensor failures. When the monitor detects a sensor failure, it modifies the system's learning simulation. The learning system operates indefinitely, and the execution system uses the results of learning as they become available. Combined

Figure 6. a) Robot with full sensors passing directly through doorway. b) Robot with front sonar covered. c) Robot after adapting to covered sonar. It uses side sonar to find opening, and then turns into the opening.

with our previous work showing adaptation to changing environments and actuator failures,[1–3] this work indicates the generality of the CEL model for the design of robust autonomous robot systems. Future work will investigate the ability to adapt to combinations of sensor and actuator failures, and attempt to quantify the limits of adaptability under this model.

ACKNOWLEDGMENTS

The work reported here was supported by The Office of Naval Research. We would like to thank Magdalena Bugajska for her assistance in preparing some of the figures.

REFERENCES

1. J. J. Grefenstette and C. L. Ramsey, "An approach to anytime learning," in *Proc. Ninth International Conference on Machine Learning*, pp. 189–195, Morgan Kaufmann, (San Mateo, CA), 1992.
2. C. L. Ramsey and J. J. Grefenstette, "Case-based anytime learning," in *Case Based Reasoning: Papers from the 1994 Workshop*, D. W. Aha, ed., Technical Report WS-94-07, AAAI Press, (Menlo Park, CA), 1994.
3. J. J. Grefenstette, "Genetic learning for adaptation in autonomous robots," in *Robotics and Manufacturing: Recent Trends in Research and Applications, Volume 6*, pp. 265–270, ASME Press, (New York), 1996.
4. J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.
5. S. Nolfi, D. Floreano, O. Mighno, and F. Mondada, "How to evolve autonomous robots: Different approaches in evolutionary robotics," in *Proc. of the International Conference on Artificial Live IV*, R. Brooks and P. Maes, eds., pp. 190–197, MIT Press, 1994.
6. I. Harvey, P. Husbands, and D. Cliff, "Issues in evolutionary robotics," in *Proceedings of the Second International Conference on Simulation of Adaptive Behaviour*, MIT Press Bradford Books, 1993.
7. A. C. Schultz, "Using a genetic algorithm to learn strategies for collision avoidance and local navigation," in *Proc. Seventh International Symposium on Unmanned Untethered Submersible Technology*, pp. 213–215, University of New Hampshire Marine Systems Engineering Laboratory, 1991.
8. K. Sims, "Evolving 3d morphology and behavior by competition," in *Proceedings of the International Conference Artificial Live IV*, pp. 28–39, MIT Press, (Cambridge, MA), 1994.
9. W.-P. Lee, J. Hallam, and H. H. Lund, "A hybrid GP/GA approach for co-evolving controllers and robot bodies to achieve fitness-specific tasks," in *Proc. of IEEE Third International Conference on Evolutionary Computation*, IEEE Press, (NJ), 1996.
10. H. H. Lund and O. Miglino, "Evolving and breeding robots," in *Proc. of the First European Workshop on Evolutionary Robotics*, Springer-Verlag, 1998.
11. M. D. Bugajska and A. C. Schultz, "Co-evolution of form and function in the design of autonomous agents: Micro air vehicle project," in *Proc. Workshop on Evolution of Sensors GECCO 2000*, IEEE, 2000.
12. D. Floreano and F. Mondada, "Evolution of homing navigation in a real mobile robot," *IEEE Transactions on System Man and Cybernetics* **26**(3), pp. 396–407, 1996.

13. J. J. Grefenstette, C. L. Ramsey, and A. C. Schultz, "Learning sequential decision rules using simulation models and competition," *Machine Learning* **5**(4), pp. 355–381, 1990.
14. A. C. Schultz and J. J. Grefenstette, "Using a genetic algorithm to learn behaviors for autonomous vehicles," in *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, (Hilton Head, SC), 1992.
15. A. C. Schultz, "Learning robot behaviors using genetic algorithms," in *Intelligent Automation and Soft Computing: Trends in Research, Development, and Applications*, pp. 607–612, TSI Press, (Albuquerque), 1994.
16. A. C. Schultz and J. J. Grefenstette, "Robo-shepherd: Learning complex robotic behaviors," in *Robotics and Manufacturing: Recent Trends in Research and Applications, Volume 6*, pp. 763–768, ASME Press, (New York), 1996.

Journal of Artificial Intelligence Research 11 (1999) 199-229 Submitted 1/99; published 8/99

APPENDIX C:

Evolutionary Algorithms for Reinforcement Learning

David E. Moriarty MORIARTY@ISI.EDU
University of Southern California, Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292

Alan C. Schultz SCHULTZ@AIC.NRL.NAVY.MIL
Navy Center for Applied Research in Artificial Intelligence
Naval Research Laboratory, Washington DC 20375-5337

John J. Grefenstette GREF@IB3.GMU.EDU
Institute for Biosciences, Bioinformatics and Biotechnology
George Mason University, Manassas, VA 20110

Abstract

There are two distinct approaches to solving reinforcement learning problems, namely, searching in value function space and searching in policy space. Temporal difference methods and evolutionary algorithms are well-known examples of these approaches. Kaelbling, Littman and Moore recently provided an informative survey of temporal difference methods. This article focuses on the application of evolutionary algorithms to the reinforcement learning problem, emphasizing alternative policy representations, credit assignment methods, and problem-specific genetic operators. Strengths and weaknesses of the evolutionary approach to reinforcement learning are presented, along with a survey of representative applications.

1. Introduction

Kaelbling, Littman, and Moore (1996) and more recently Sutton and Barto (1998) provide informative surveys of the field of reinforcement learning (RL). They characterize two classes of methods for reinforcement learning: methods that search the space of value functions and methods that search the space of policies. The former class is exemplified by the temporal difference (TD) method and the latter by the evolutionary algorithm (EA) approach. Kaelbling et al. focus entirely on the first set of methods and they provide an excellent account of the state of the art in TD learning. This article is intended to round out the picture by addressing evolutionary methods for solving the reinforcement learning problem.

As Kaelbling et al. clearly illustrate, reinforcement learning presents a challenging array of difficulties in the process of scaling up to realistic tasks, including problems associated with very large state spaces, partially observable states, rarely occurring states, and non-stationary environments. At this point, which approach is best remains an open question, so it is sensible to pursue parallel lines of research on alternative methods. While it is beyond the scope of this article to address whether it is better in general to search value function space or policy space, we do hope to highlight some of the strengths of the evolutionary approach to the reinforcement learning problem. The reader is advised not to view this article as an EA *vs.* TD discussion. In some cases, the two methods provide complementary strengths, so hybrid approaches are advisable; in fact, our survey of implemented systems illustrates that many EA-based reinforcement learning systems include elements of TD-learning as well.

The next section spells out the reinforcement learning problem. In order to provide a specific anchor for the later discussion, Section 3 presents a particular TD method. Section 4 outlines the approach we call Evolutionary Algorithms for Reinforcement Learning (EARL), and provides a simple example of a particular EARL system. The following three sections focus on features that

distinguish EAs for RL from EAs for general function optimization, including alternative policy representations, credit assignment methods, and RL-specific genetic operators. Sections 8 and 9 highlight some strengths and weaknesses of the EA approach. Section 10 briefly surveys some successful applications of EA systems on challenging RL tasks. The final section summarizes our presentation and points out directions for further research.

2. Reinforcement Learning

All reinforcement learning methods share the same goal: to solve *sequential decision tasks* through trial and error interactions with the environment (Barto, Sutton, & Watkins, 1990; Grefenstette, Ramsey, & Schultz, 1990). In a sequential decision task, an agent interacts with a dynamic system by selecting actions that affect state transitions to optimize some reward function. More formally, at any given time step t, an agent perceives its *state* s_t and selects an *action* a_t. The system responds by giving the agent some (possibly zero) numerical *reward* $r(s_t)$ and changing into state $s_{t+1} = \delta(s_t, a_t)$. The state transition may be determined solely by the current state and the agent's action or may also involve stochastic processes.

The agent's goal is to learn a *policy*, $\pi : S \rightarrow A$, which maps states to actions. The *optimal policy*, π^*, can be defined in many ways, but is typically defined as the policy that produces the greatest cumulative reward over all states s:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} V^\pi(s), (\forall s) \tag{1}$$

where $V^\pi(s)$ is the cumulative reward received from state s using policy π. There are also many ways to compute $V^\pi(s)$. One approach uses a discount rate γ to discount rewards over time. The sum is then computed over an infinite horizon:

$$V^\pi(s_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{2}$$

where r_t is the reward received at time step t. Alternatively, $V^\pi(s)$ could be computed by summing the rewards over a finite horizon h:

$$V^\pi(s_t) = \sum_{i=0}^{h} r_{t+i} \tag{3}$$

The agent's state descriptions are usually identified with the values returned by its *sensors*, which provide a description of both the agent's current state and the state of the world. Often the sensors do not give the agent complete state information and thus the state is only *partially observable.*

Besides reinforcement learning, intelligent agents can be designed by other paradigms, notably *planning* and *supervised learning.* We briefly note some of the major differences among these approaches. In general, planning methods require an explicit model of the state transition function $\delta(s, a)$. Given such a model, a planning algorithm can search through possible action choices to find an action sequence that will guide the agent from an initial state to a goal state. Since planning algorithms operate using a model of the environment, they can backtrack or "undo" state transitions that enter undesirable states. In contrast, RL is intended to apply to situations in which a sufficiently tractable action model does not exist. Consequently, an agent in the RL paradigm must actively explore its environment in order to observe the effects of its actions. Unlike planning, RL agents cannot normally undo state transitions. Of course, in some cases it may be possible to build up an action model through experience (Sutton, 1990), enabling more planning as experience accumulates. However, RL research focuses on the behavior of an agent when it has insufficient knowledge to perform planning.

	a	b	c	d	e
1	0	2	1	-1	1
2	1	1	2	0	2
3	3	-5	4	3	1
4	1	-2	4	1	2
5	1	1	2	1	1

Figure 1: A simple grid-world sequential decision task. The agent starts in state $a1$ and receives the row and column of the current box as sensory input. The agent moves from one box to another by selecting between two moves (right or down), and the agent's score is increased by the payoff indicated in each box. The goal is to find a policy that maximizes the cumulative score.

Agents can also be trained through supervised learning. In supervised learning, the agent is presented with examples of state-action pairs, along with an indication that the action was either correct or incorrect. The goal in supervised learning is to induce a general policy from the training examples. Thus, supervised learning requires an *oracle* that can supply correctly labeled examples. In contrast, RL does not require prior knowledge of correct and incorrect decisions. RL can be applied to situations in which rewards are sparse; for example, rewards may be associated only with certain states. In such cases, it may be impossible to associate a label of "correct" or "incorrect" on particular decisions without reference to the agent's subsequent decisions, making supervised learning infeasible.

In summary, RL provides a flexible approach to the design of intelligent agents in situations for which both planning and supervised learning are impractical. RL can be applied to problems for which significant domain knowledge is either unavailable or costly to obtain. For example, a common RL task is robot control. Designers of autonomous robots often lack sufficient knowledge of the intended operational environment to use either the planning or the supervised learning regime to design a control policy for the robot. In this case, the goal of RL would be to enable the robot to generate effective decision policies as it explores its environment.

Figure 1 shows a simple sequential decision task that will be used as an example later in this paper. The task of the agent in this grid world is to move from state to state by selecting among two actions: right (R) or down (D). The sensor of the agent returns the identity of the current state. The agent always starts in state $a1$ and receives the reward indicated upon visiting each state. The task continues until the agent moves off the grid world (e.g., by taking action D from state $a5$). The goal is to learn a policy that returns the highest cumulative rewards. For example, a policy which results in the sequences of actions R, D, R, D, D, R, R, D starting from from state $a1$ gives the optimal score of 17.

2.1 Policy Space vs. Value-Function Space

Given the reinforcement learning problem as described in the previous section, we now address the main topic: how to find an optimal policy, π^*. We consider two main approaches, one involves search in *policy space* and the other involves search in *value function space.*

Policy-space search methods maintain explicit representations of policies and modify them through a variety of search operators. Many search methods have been considered, including dynamic programming, value iteration, simulated annealing, and evolutionary algorithms. This paper focuses on evolutionary algorithms that have been specialized for the reinforcement learning task.

In contrast, value function methods do not maintain an explicit representation of a policy. Instead, they attempt learn the value function V^{π^*}, which returns the expected cumulative reward for the optimal policy from any state. The focus of research on value function approaches to RL is to design algorithms that learn these value functions through experience. The most common approach to learning value functions is the temporal difference (TD) method, which is described in the next section.

3. Temporal Difference Algorithms for Reinforcement Learning

As stated in the Introduction, a comprehensive comparison of value function search and direct policy-space search is beyond the scope of this paper. Nevertheless, it will be useful to point out key conceptual differences between typical value function methods and typical evolutionary algorithms for searching policy space. The most common approach for learning a value function V for RL problems is the temporal difference (TD) method (Sutton, 1988).

The TD learning algorithm uses observations of *prediction differences* from consecutive states to update value predictions. For example, if two consecutive states i and j return payoff prediction values of 5 and 2, respectively, then the difference suggests that the payoff from state i may be overestimated and should be reduced to agree with predictions from state j. Updates to the value function V are achieved using the following update rule:

$$V(s_t) = V(s_t) + \alpha(V(s_{t+1}) - V(s_t) + r_t) \tag{4}$$

where α represents the learning rate and r_t any immediate reward. Thus, the difference in predictions $(V(s_{t+1}) - V(s_t))$ from consecutive states is used as a measure of prediction error. Consider a chain of value predictions $V(s_0)..V(s_n)$ from consecutive state transitions with the last prediction $V(s_n)$ containing the only non-zero reward from the environment. Over many iterations of this sequence, the update rule will adjust the values of each state so that they agree with their successors and eventually with the reward received in $V(s_n)$. In other words, the single reward is propagated backwards through the chain of value predictions. The net result is an accurate value function that can be used to predict the expected reward from any state of the system.

As mentioned earlier, the goal of TD methods is to learn the value function for the optimal policy, V^{π^*}. Given V^{π^*}, the optimal action, $\pi(s)$, can be computed using the following equation:

$$\pi(s) = \underset{a}{\operatorname{argmax}} V^{\pi^*}(\delta(s,a)) \tag{5}$$

Of course, we have already stated that in RL the state transition function $\delta(s,a)$ is unknown to the agent. Without this knowledge, we have no way of evaluating (5). An alternative value function that can be used to compute $\pi^*(s)$ is called a Q-function, $Q(s,a)$ (Watkins, 1989; Watkins & Dayan, 1992). The Q-function is a value function that represents the expected value of taking action a in state s and acting optimally thereafter:

$$Q(s,a) = r(s) + V^{\pi}(\delta(s,a)) \tag{6}$$

	a1	a2	a3	a4	a5	b1	b2	b3	b4	b5	c1	c2	c3	c4	c5	d1	d2	d3	d4	d5	e1	e2	e3	e4	e5
R	17	16	10	7	6	17	15	7	6	5	7	9	11	8	4	6	6	7	4	2	1	2	1	2	1
D	16	11	10	7	1	17	8	1	3	1	15	14	12	8	2	6	7	7	3	1	7	6	4	3	1

Table 1: A Q-function for the simple grid world. A value is associated with each state-action pair.

where $r(s)$ represents any immediate reward received in state s. Given the Q-function, actions from the optimal policy can be directly computed using the following equation:

$$\pi^*(s) = \operatorname*{argmax}_a Q(s,a) \tag{7}$$

Table 1 shows the Q-function for the grid world problem of Figure 1. This table-based representation of the Q-function associates cumulative future payoffs for each state-action pair in the system. (The letter-number pairs at the top represent the state given by the row and column in Figure 1, and R and D represent the actions *right* and *down*, respectively.) The TD method adjusts the Q-values after each decision. When selecting the next action, the agent considers the effect of that action by examining the expected value of the state transition caused by the action.

The Q-function is learned through the following TD update equation:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha(\max_{a_{t+1}} Q(s_{t+1},a_{t+1}) - Q(s_t,a_t) + r(s_t)) \tag{8}$$

Essentially, this equation updates $Q(s_t,a_t)$ based on the current reward and the predicted reward if all future actions are selected optimally. Watkins and Dayan (1992) proved that if updates are performed in this fashion and if every Q-value is explicitly represented, the estimates will asymptotically converge to the correct values. A reinforcement learning system can thus use the Q values to select the optimal action in any state. Because Q-learning is the most widely known implementation of temporal difference learning, we will use it in our qualitative comparisons with evolutionary approaches in later sections.

4. Evolutionary Algorithms for Reinforcement Learning (EARL)

The policy-space approach to RL searches for policies that optimize an appropriate objective function. While many search algorithms might be used, this survey focuses on evolutionary algorithms. We begin with a brief overview of a simple EA for RL, followed by a detailed discussion of features that characterize the general class of EAs for RL.

4.1 Design Considerations for Evolutionary Algorithms

Evolutionary algorithms (EAs) are global search techniques derived from Darwin's theory of evolution by natural selection. An EA iteratively updates a population of potential solutions, which are often encoded in structures called *chromosomes*. During each iteration, called a *generation*, the EA evaluates solutions and generates offspring based on the fitness of each solution in the task environment. Substructures, or *genes*, of the solutions are then modified through genetic operators such as mutation and recombination. The idea is that structures that are associated with good solutions can be mutated or combined to form even better solutions in subsequent generations. The canonical evolutionary algorithm is shown in Figure 2. There have been a wide variety of EAs developed, including genetic algorithms (Holland, 1975; Goldberg, 1989), evolutionary programming (Fogel, Owens, & Walsh, 1966), genetic programming (Koza, 1992), and evolutionary strategies (Rechenberg, 1964).

EAs are general purpose search methods and have been applied in a variety of domains including numerical function optimization, combinatorial optimization, adaptive control, adaptive testing,

```
procedure EA
begin
    t = 0;
    initialize P(t);
    evaluate structures in P(t);
    while termination condition not satisfied do
        begin
            t = t + 1;
            select P(t) from P(t-1);
            alter structures in P(t);
            evaluate structures in P(t);
        end
end.
```

Figure 2: Pseudo-code Evolutionary Algorithm.

and machine learning. One reason for the widespread success of EAs is that there are relatively few requirements for their application, namely,

1. An appropriate mapping between the search space and the space of chromosomes, and
2. An appropriate fitness function.

For example, in the case of parameter optimization, it is common to represent the list of parameters as either a vector of real numbers or a bit string that encodes the parameters. With either of these representations, the "standard" genetic operators of mutation and cut-and-splice crossover can be applied in a straightforward manner to produce the genetic variations required (see Figure 3). The user must still decide on a (rather large) number of control parameters for the EA, including population size, mutation rates, recombination rates, parent selection rules, but there is an extensive literature of studies which suggest that EAs are relatively robust over a wide range of control parameter settings (Grefenstette, 1986; Schaffer, Caruana, Eshelman, & Das, 1989). Thus, for many problems, EAs can be applied in a relatively straightforward manner.

However, for many other applications, EAs need to be specialized for the problem domain (Grefenstette, 1987). The most critical design choice facing the user is the representation, that is, the mapping between the search space of knowledge structures (or, the *phenotype* space) and the space of chromosomes (the *genotype* space). Many studies have shown that the effectiveness of EAs is sensitive to the choice of representations. It is not sufficient, for example, to choose an arbitrary mapping from the search space into the space of chromosomes, apply the standard genetic operators and hope for the best. What makes a good mapping is a subject for continuing research, but the general consensus is that candidate solutions that share important phenotypic similarities must also exhibit similar forms of "building blocks" when represented as chromosomes (Holland, 1975). It follows that the user of an EA must carefully consider the most natural way to represent the elements of the search space as chromosomes. Moreover, it is often necessary to design appropriate mutation and recombination operators that are specific to the chosen representation. The end result of this design process is that the representation and genetic operators selected for the EA comprise a form of search *bias* similar to biases in other machine learning methods. Given the proper bias, the

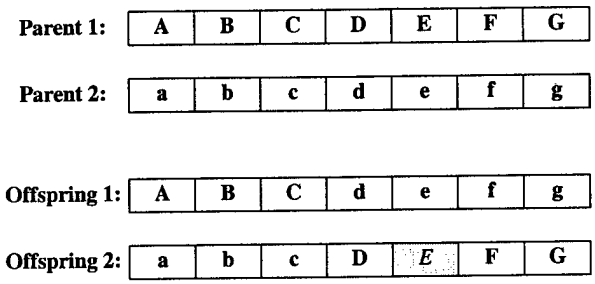


Figure 3: Genetic operators on fixed-position representation. The two offspring are generated by crossing over the selected parents. The operation shown is called *one-point crossover*. The first offspring inherits the initial segment of one parent and the final segment of the other parent. The second offspring inherits the same pattern of genes from the opposite parents. The crossover point is position 3, chosen at random. The second offspring has also incurred a mutation in the shaded gene.

EA can quickly identify useful "building blocks" within the population, and converge on the most promising areas of the search space.[1]

In the case of RL, the user needs to make two major design decisions. First, how will the space of policies be represented by chromosomes in the EA? Second, how will the fitness of population elements be assessed? The answers to these questions depend on how the user chooses to bias the EA. The next section presents a simple EARL that adopts the most straightforward set of design decisions. This example is meant only to provide a baseline for comparison with more elaborate designs.

4.2 A Simple EARL

As the remainder of this paper shows, there are many ways to use EAs to search the space of RL policies. This section provides a concrete example of a simple EARL, which we call EARL_1. The pseudo-code is shown in Figure 4. This system provides the EA counterpart to the simple table-based TD system described in Section 3.

The most straightforward way to represent a policy in an EA is to use a single chromosome per policy with a single gene associated with each observed state. In EARL_1, each gene's value (or *allele* in biological terminology) represents the action value associated with the corresponding state, as shown in Figure 5. Table 2 shows part of an EARL_1 population of policies for the sample grid world problem. The number of policies in a population is usually on the order of 100 to 1000.

The fitness of each policy in the population must reflect the expected accumulated fitness for an agent that uses the given policy. There are no fixed constraints on how the fitness of an individual policy is evaluated. If the world is deterministic, like the sample grid-world, the fitness of a policy can be evaluated during a single trial that starts with the agent in the initial state and terminates when the agent reaches a terminal state (e.g., falls off the grid in the grid-world). In non-deterministic worlds, the fitness of a policy is usually averaged over a sample of trials. Other options include measuring the total payoff achieved by the agent after a fixed number of steps, or measuring the number of steps required to achieve a fixed level of payoff.

1. Other ways to exploit problem specific knowledge in EAs include the use of heuristics to initialize the population and the hybridization with problem specific search algorithms. See (Grefenstette, 1987) for further discussions of these methods.

```
procedure EARL-1
begin
    t = 0;
    initialize a population of policies, P(t);
    evaluate policies in P(t);
    while termination condition not satisfied do
        begin
            t = t + 1;
            select high-payoff policies, P(t), from policies in P(t-1);
            update policies in P(t);
            evaluate policies in P(t);
        end
end.
```

Figure 4: Pseudo-code for Evolutionary Algorithm Reinforcement Learning system.

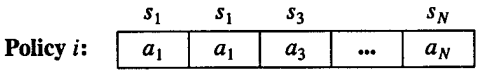


Figure 5: Table-based policy representation. Each observed state has a gene which indicates the preferred action for that state. With this representation, standard genetic operators such as mutation and crossover can be applied.

Policy	a1	a2	a3	a4	a5	b1	b2	b3	b4	b5	c1	c2	c3	c4	c5	d1	d2	d3	d4	d5	e1	e2	e3	e4	e5	Fitness
1	D	R	D	D	R	R	R	R	R	R	D	R	D	D	R	R	D	R	R	R	D	R	R	D	R	8
2	D	D	D	D	R	R	R	R	R	R	D	D	R	R	D	R	D	R	R	R	D	R	D	D	R	9
3	R	D	D	R	R	D	R	D	R	R	D	D	D	R	D	R	D	R	R	R	D	R	D	D	D	17
4	D	D	D	D	R	D	R	R	R	R	R	D	R	R	R	D	R	R	D	R	D	R	D	D	R	11
5	R	D	D	D	R	D	R	R	D	R	R	D	R	R	D	R	D	R	R	D	D	R	D	D	D	16

Table 2: An EA population of five decision policies for the sample grid world. This simple policy representation specifies an action for each state of the world. The fitness corresponds to the payoffs that are accumulated using each policy in the grid world.

Once the fitness of all policies in the population has been determined, a new population is generated according to the steps in the usual EA (Figure 2). First, parents are selected for reproduction. A typical selection method is to probabilistically select individuals based on relative fitness:

$$\Pr(p_i) = \frac{Fitness(p_i)}{\sum_{j=1}^{n} Fitness(p_j)} \tag{9}$$

where p_i represents individual i and n is the total number of individuals. Using this selection rule, the expected number of offspring for a given policy is proportional to that policy's fitness. For example, a policy with average fitness might have a single offspring, whereas a policy with twice the average

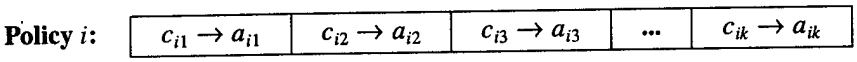


Figure 6: Rule-based policy representation. Each gene represents a condition-action rule that maps a set of states to an action. In general, such rules are independent of the position along the chromosome. Conflict resolution mechanisms may be needed if the conditions of rules are allowed to intersect.

fitness would have two offspring.[2] Offspring are formed by cloning the selected parents. Then new policies are generated by applying the standard genetic operators of crossover and mutation to the clones, as shown in Figure 3. The process of generating new populations of strategies can continue indefinitely or can be terminated after a fixed number of generations or once an acceptable level of performance is achieved.

For simple RL problems such as the grid-world, EARL$_1$ may provide an adequate approach. In later sections, we will point out some ways in which even EARL$_1$ exhibits strengths that are complementary to TD methods for RL. However, as in the case of TD methods, EARL methods have been extended to handle the many challenges inherent in more realistic RL problems. The following sections survey some of these extensions, organized around three specific biases that distinguish EAs for Reinforcement Learning (EARL) from more generic EAs: policy representations, fitness/credit-assignment models, and RL-specific genetic operators.

5. Policy Representations in EARL

Perhaps the most critical feature that distinguishes classes of EAs from one another is the representation used. For example, EAs for function optimization use a simple string or vector representation, whereas EAs for combinatorial optimization use distinctive representations for permutations, trees or other graph structures. Likewise, EAs for RL use a distinctive set of representations for policies. While the range of potential policy representations is unlimited, the representations used in most EARL systems to date can be largely categorized along two discrete dimensions. First, policies may be represented either by condition-action rules or by neural networks. Second, policies may be represented by a single chromosome or the representation may be distributed through one or more populations.

5.1 Single-Chromosome Representation of Policies

5.1.1 RULE-BASED POLICIES

For most RL problems of practical interest, the number of observable states is very large, and the simple table-based representation in EARL$_1$ is impractical. For large scale state spaces, it is more reasonable to represent a policy as a set of condition-action rules in which the condition expresses a predicate that matches a set of states, as shown in Figure 6. Early examples of this representation include the systems LS-1 (Smith, 1983) and LS-2 (Schaffer & Grefenstette, 1985), followed later by SAMUEL (Grefenstette et al., 1990).

5.1.2 NEURAL NET REPRESENTATION OF POLICIES

As in TD-based RL systems, EARL systems often employ neural net representations as function approximators. In the simplest case (see Figure 7), a neural network for the agent's decision policy is represented as a sequence of real-valued connection weights. A straightforward EA for parameter

2. Many other parent selection rules have been explored (Grefenstette, 1997a, 1997b).

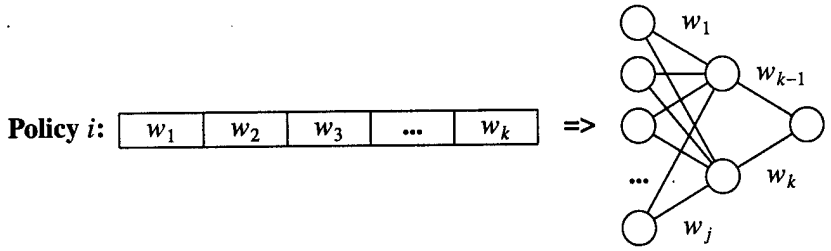


Figure 7: A simple parameter representation of weights for a neural network. The fitness of the policy is the payoff when the agent uses the corresponding neural net as its decision policy.

optimization can be used to optimize the weights of the neural network (Belew, McInerney, & Schraudolph, 1991; Whitley, Dominic, Das, & Anderson, 1993; Yamauchi & Beer, 1993). This representation thus requires the least modification of the standard EA. We now turn to distributed representations of policies in EARL systems.

5.2 Distributed Representation of Policies

In the previous section we outlined EARL approaches that treat the agent's decision policy as a single genetic structure that evolves over time. This section addresses EARL approaches that decompose a decision policy into smaller components. Such approaches have two potential advantages. First, they allow evolution to work at a more detailed level of the task, e.g., on specific subtasks. Presumably, evolving a solution to a restricted subtask should be easier than evolving a monolithic policy for a complex task. Second, decomposition permits the user to exploit background knowledge. The user might base the decomposition into subtasks on a prior analysis of the overall performance task; for example, it might be known that certain subtasks are mutually exclusive and can therefore be learned independently. The user might also decompose a complex task into subtasks such that certain components can be explicitly programmed while other components are learned.

In terms of knowledge representation in EARL, the alternative to the single chromosome representation is to distribute the policy over several population elements. By assigning a fitness to these individual elements of the policy, evolutionary selection pressure can be brought to bear on more detailed aspects of the learning task. That is, fitness is now a function of individual subpolicies or individual rules or even individual neurons. This general approach is analogous to the classic TD methods that take this approach to the extreme of learning statistics concerning each state-action pair. As in the case of single-chromosome representations, we can partition distributed EARL representations into rule-based and neural-net-based classes.

5.2.1 DISTRIBUTED RULE-BASED POLICIES

The most well-known example of a distributed rule-based approach to EARL is the Learning Classifier Systems (LCS) model (Holland & Reitman, 1978; Holland, 1987; Wilson, 1994). An LCS uses an evolutionary algorithm to evolve if-then rules called *classifiers* that map sensory input to an appropriate action. Figure 8 outlines Holland's LCS framework (Holland, 1986). When sensory input is received, it is posted on the *message list.* If the left hand side of a classifier matches a message on the message list, its right hand side is posted on the message list. These new messages may subsequently trigger other classifiers to post messages or invoke a decision from the LCS, as in the traditional forward-chaining model of rule-based systems.

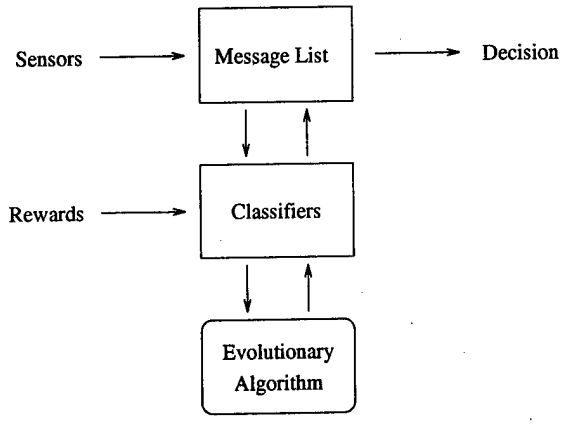


Figure 8: Holland's Learning Classifier System.

condition		action	strength
a#	→	R	0.75
#2	→	D	0.25
	...		
d3	→	D	0.50

Table 3: LCS population for grid world. The # is a *don't care* symbol which allows for generality in conditions. For example, the first rule says "Turn right in column a." The *strength* of a rule is used for conflict resolution and for parent selection in the genetic algorithm.

In an LCS, each chromosome represents a single decision rule and the entire population represents the agent's policy. In general, classifiers map a set of observed states to a set of messages, which may be interpreted as either internal state changes or actions. For example, if the learning agent for the grid world in Figure 1 has two sensors, one for the column and one for the row, then the population in an LCS might appear as shown in Table 3. The first classifier matches any state in the column a and recommends action R. Each classifier has a statistic called *strength* that estimates the utility of the rule. The strength statistics are used in both conflict resolution (when more than one action is recommended) and as fitness for the genetic algorithm. Genetic operators are applied to highly fit classifiers to generate new rules. Generally, the population size (i.e., the number of rules in the policy) is kept constant. Thus classifiers compete for space in the policy.

Another way that EARL systems distribute the representation of policies is to partition the policy into separate modules, with each module updated by its own EA. Dorigo and Colombetti (1998) describe an architecture called ALECSYS in which a complex reinforcement learning task is decomposed into subtasks, each of which is learned via a separate LCS, as shown in Figure 9. They provide a method called *behavior analysis and training* (BAT) to manage the incremental training of agents using the distributed LCS architecture.

The single-chromosome representation can also be extended by partitioning the policy across multiple co-evolving populations. For example, in the cooperative co-evolution model (Potter, 1997), the agent's policy is formed by combining chromosomes from several independently evolving populations. Each chromosome represents a set of rules, as in Figure 6, but these rules address only a subset of the performance task. For example, separate populations might evolve policies for different components of a complex task, or might address mutually exclusive sets of observed states. The fitness of each chromosome is computed based on the overall fitness of the agents that employ

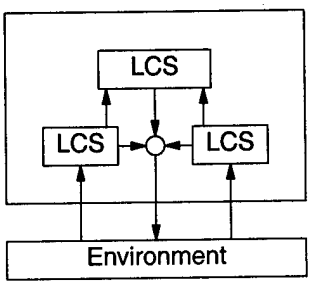


Figure 9: A two-level hierarchical ALECSYS system. Each LCS learns a specific behavior. The interactions among the rule sets are pre-programmed.

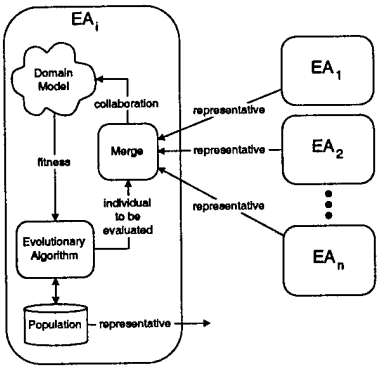


Figure 10: Cooperative coevolutionary architecture from the perspective of the i^{th} EA instance. Each EA contributes a representative, which is merged with the others' representatives to form a *collaboration*, or policy for the agent. The fitness of each representative reflects the average fitness of its collaborations.

that chromosome as part of its combined chromosomes. The combined chromosomes represent the decision policy and are called a *collaboration* (Figure 10).

5.2.2 DISTRIBUTED NETWORK-BASED POLICIES

Distributed EARL systems using neural net representations have also been designed. In (Potter & De Jong, 1995), separate populations of neurons evolve, with the evaluation of each neuron based on the fitness of a collaboration of neurons selected from each population. In SANE (Moriarty & Miikkulainen, 1996a, 1998), two separate populations are maintained and evolved: a population of neurons and a population of network blueprints. The motivation for SANE comes from our *a priori* knowledge that individual neurons are fundamental building blocks in neural networks. SANE explicitly decomposes the neural network search problem into several parallel searches for effective single neurons. The neuron-level evolution provides evaluation and recombination of the neural network building blocks, while the population of blueprints search for effective combinations of these building blocks. Figure 11 gives an overview of the interaction of the two populations.

Each individual in the blueprint population consists of a set of pointers to individuals in the neuron population. During each generation, neural networks are constructed by combining the

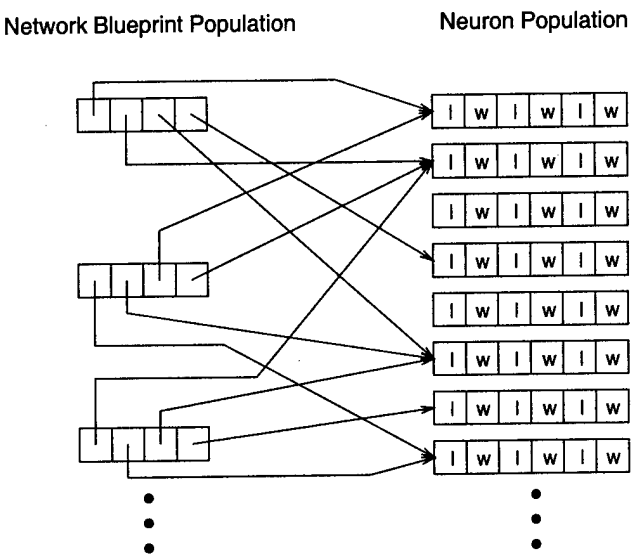


Figure 11: An overview of the two populations in SANE. Each member of the neuron population specifies a series of connections (connection labels and weights) to be made within a neural network. Each member of the network blueprint population specifies a series of pointers to specific neurons which are used to build a neural network.

hidden neurons specified in each blueprint. Each blueprint receives a fitness according to how well the corresponding network performs in the task. Each neuron receives a fitness according to how well the top networks in which it participates perform in the task. An aggressive genetic selection and recombination strategy is used to quickly build and propagate highly fit structures in both the neuron and blueprint populations.

6. Fitness and Credit Assignment in EARL

Evolutionary algorithms are all driven by the concept of natural selection: population elements that have higher fitness leave more offspring to later generations, thus influencing the direction of search in favor of high performance regions of the search space. The concept of fitness is central to any EA. In this section, we discuss features of the fitness model that are common across most EARL systems. We specifically focus on ways in which the fitness function reflects the distinctive structure of the RL problem.

6.1 The Agent Model

The first common features of all EARL fitness models is that fitness is computed with respect to an RL agent. That is, however the policy is represented in the EA, it must be converted to a decision policy for an agent operating in a RL environment. The agent is assumed to observe a description of the current state, select its next action by consulting its current policy, and collect whatever reward is provided by the environment. In EARL systems, as in TD systems, the agent is generally assumed to perform very little additional computation when selecting its next action. While neither approach limits the agent to strict stimulus-response behavior, it is usually assumed that the agent does not perform extensive planning or other reasoning before acting. This assumption reflects the

fact that RL tasks involve some sort of control activity in which the agent must respond to a dynamic environment within a limited time frame.

6.2 Policy Level Credit Assignment

As shown in the previous section, the meaning of fitness in EARL systems may vary depending on what the population elements represent. In a single-chromosome representation, fitness is associated with entire policies; in a distributed representation, fitness may be associated with individual decision rules. In any case, fitness always reflects accumulated rewards received by the agent during the course of interaction with the environment, as specified in the RL model. Fitness may also reflect effort expended, or amount of delay.

It is worthwhile considering the different approaches to credit assignment in the TD and EA methods. In a reinforcement learning problem, payoffs may be sparse, that is, associated only with certain states. Consequently, a payoff may reflect the quality of an extended sequence of decisions, rather than any individual decision. For example, a robot may receive a reward after a movement that places it in a "goal" position within a room. The robot's reward, however, depends on many of its previous movements leading it to that point. A difficult *credit assignment* problem therefore exists in how to apportion the rewards of a sequence of decisions to individual decisions.

In general, EA and TD methods address the credit assignment problem in very different ways. In TD approaches, credit from the reward signal is explicitly propagated to each decision made by the agent. Over many iterations, payoffs are distributed across a sequence of decisions so that an appropriately discounted reward value is associated with each individual state and decision pair.

In simple EARL systems such as EARL_1, rewards are associated only with sequences of decisions and are not distributed to the individual decisions. Credit assignment for an individual decision is made implicitly, since policies that prescribe poor individual decisions will have fewer offspring in future generations. By selecting against poor policies, evolution automatically selects against poor individual decisions. That is, building blocks consisting of particular state-action pairs that are highly correlated with good policies are propagated through the population, replacing state-action pairs associated with poorer policies.

Figure 12 illustrates the differences in credit assignment between TD and EARL_1 in the grid world of Figure 1. The Q-learning TD method explicitly assigns credit or blame to each individual state-action pair by passing back the immediate reward and the estimated payoff from the new state. Thus, an error term becomes associated with each action performed by the agent. The EA approach does not explicitly propagate credit to each action but rather associates an overall fitness with the entire policy. Credit is assigned implicitly, based on the fitness evaluations of entire sequences of decisions. Consequently, the EA will tend to select against policies that generate the first and third sequences because they achieve lower fitness scores. The EA thus implicitly selects against action D in state $b2$, for example, which is present in the bad sequences but not present in the good sequences.

6.3 Subpolicy Credit Assignment

Besides the implicit credit assignment performed on building blocks, EARL systems have also addressed the credit assignment problem more directly. As shown in Section 4, the individuals in an EARL system might represent either entire policies or components of a policy (e.g., component rule-sets, individual decision rules, or individual neurons). For distributed-representation EARLs, fitness is explicitly assigned to individual components. In cases in which a policy is represented by explicit components, different fitness functions can be associated with different evolving populations, allowing the implementer to "shape" the overall policy by evolving subpolicies for specific subtasks (Dorigo & Colombetti, 1998; Potter, De Jong, & Grefenstette, 1995). The most ambitious goal is to allow the system to manage the number of co-evolving species as well as the form of interactions (Potter, 1997). This exciting research is still at an early stage.

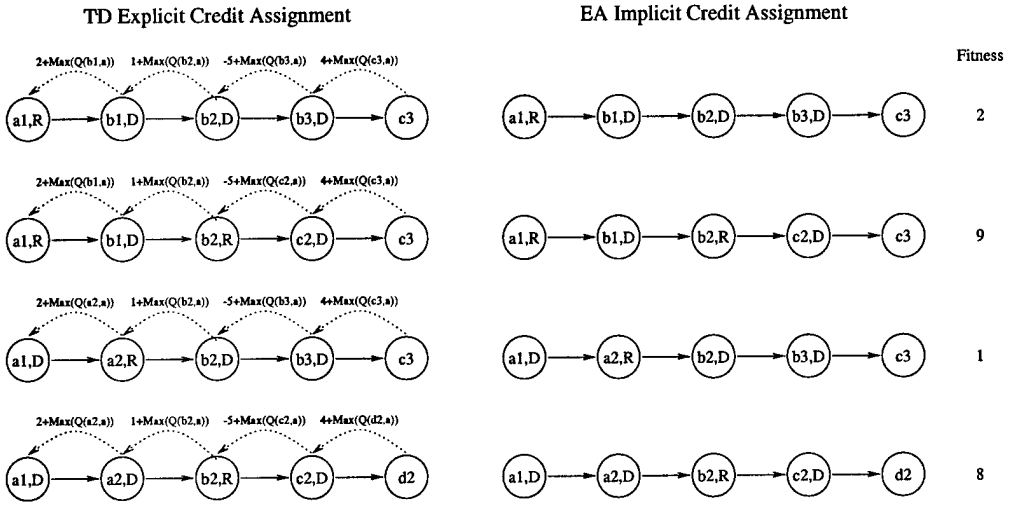


Figure 12: Explicit vs. implicit credit assignment. The Q-learning TD method assigns credit to each state-action pair based on the immediate reward and the predicted future rewards. The EA method assigns credit implicitly by associating fitness values with entire sequences of decisions.

For example, in the LCS model, each classifier (decision rule) has a *strength* which is updated using a TD-like method called the ***bucket brigade algorithm*** (Holland, 1986). In the bucket brigade algorithm, the strength of a classifier is used to bid against other classifiers for the right to post messages. Bids are subtracted from winning classifiers and passed back to the classifiers that posted the enabling message on the previous step. Classifier strengths are thus reinforced if the classifier posts a message that triggers another classifier. The classifier that invokes a decision from the LCS receives a strength reinforcement directly from the environment. The bucket brigade bid passing mechanism clearly bears a strong relation to the method of temporal differences (Sutton, 1988). The bucket brigade updates a given classifier's strength based on the strength of the classifiers that fire as a direct result of its activation. The TD methods differ slightly in this respect because they assign credit based strictly on temporal succession and do not take into account causal relations of steps. It remains unclear which is more appropriate for distributing credit.

Even for single chromosome representations, TD-like methods have been adopted in some EARL systems. In SAMUEL, each gene (decision rule) also maintains a quantity called *strength* that is used to resolve conflict when more than one rule matches the agent's current sensor readings. When payoff is obtained (thereby terminating the *trial*), the strengths of all rules that fired during the trial are updated (Grefenstette, 1988). In addition to resolving conflicts, a rule's strength also plays a role in triggering mutation operations, as described in the next section.

7. RL-Specific Genetic Operators

The creation of special genetic operators provides another avenue for imposing an RL-specific bias on EAs. Specialized operators in EARL systems first appeared in (Holland, 1986), in which so-called *triggered operators* were responsible for creating new classifiers when the learning agent found that no classifier in its existing population matched the agent's current sensor readings. In this case, a high-strength rule was explicitly generalized to cover the new set of sensor readings. A similar rule-creation operator was included in early versions of SAMUEL (Grefenstette et al., 1990). Later

versions of SAMUEL included a number of mutation operators which created altered rules based on an agent's early experiences. For example, SAMUEL's *Specialization* mutation operator is triggered when a low-strength, general rule fires during an episode that results in high payoff. In such a case, the rule's conditions are reduced in generality to more closely match the agent's sensor readings. For example, if the agent has a sensor readings (*range* = 40, *bearing* = 100) and the original rule is:

IF *range* = [25,55] AND *bearing* = [0,180] THEN SET *turn* = 24 (*strength* 0.1)

then the new rule would be:

IF *range* = [35,45] AND *bearing* = [50,140] THEN SET *turn* = 24 (*strength* 0.8)

Since the episode triggering the operator resulted in high payoff, one might suspect that the original rule was over-generalized, and that the new, more specific version might lead to better results. (The strength of the new rule is initialized to the payoff received during the triggering episode.) This is considered a Lamarckian operator because the agent's experience is causing a genetic change which is passed on to later offspring.[3]

SAMUEL also uses an RL-specific crossover operator to recombine policies. In particular, crossover in SAMUEL attempts to cluster decision rules before assigning them to offspring. For example, suppose that the traces of the most previous evaluations of the parent strategies are as follows ($R_{i,j}$ denotes the j^{th} decision rule in policy i):

Trace for parent #1:
Episode:

$\vdots$
8. $R_{1,3} \rightarrow R_{1,1} \rightarrow R_{1,7} \rightarrow R_{1,5}$ High Payoff
9. $R_{1,2} \rightarrow R_{1,8} \rightarrow R_{1,4}$ Low Payoff
$\vdots$

Trace for parent #2:

$\vdots$
4. $R_{2,7} \rightarrow R_{2,5}$ Low Payoff
5. $R_{2,6} \rightarrow R_{2,2} \rightarrow R_{2,4}$ High Payoff
$\vdots$

Then one possible offspring would be:

$$\{R_{1,8}, \ldots, R_{1,3}, R_{1,1}, R_{1,7}, R_{1,5}, \ldots, R_{2,6}, R_{2,2}, R_{2,4}, \ldots, R_{2,7}\}$$

The motivation here is that rules that fire in sequence to achieve a high payoff should be treated as a group during recombination, in order to increase the likelihood that the offspring policy will inherit some of the better behavior patterns of its parents. Rules that do not fire in successful episodes (e.g., $R_{1,8}$) are randomly assigned to one of the two offspring. This form of crossover is not only Lamarckian (since it is triggered by the experiences of the agent), but is directly related to the structure of the RL problem, since it groups components of policies according to the temporal association among the decision rules.

3. Jean Baptiste Lamarck developed an evolutionary theory that stressed the inheritance of acquired characteristics, in particular acquired characteristics that are well adapted to the surrounding environment. Of course, Lamarck's theory was superseded by Darwin's emphasis on two-stage adaptation: undirected variation followed by selection. Research has generally failed to substantiate any Lamarckian mechanisms in biological systems (Gould, 1980).

8. Strengths of EARL

The EA approach represents an interesting alternative for solving RL problems, offering several potential advantages for scaling up to realistic applications. In particular, EARL systems have been developed that address difficult challenges in RL problems, including:

- Large state spaces;
- Incomplete state information; and
- Non-stationary environments.

This section focuses on ways that EARL address these challenges.

8.1 Scaling Up to Large State Spaces

Many early papers in the RL literature analyze the efficiency of alternative learning methods on toy problems similar to the grid world shown in Figure 1. While such studies are useful as academic exercises, the number of observed states in realistic applications of RL is likely to preclude any approach that requires the explicit storage and manipulation of statistics associated with each observable state-action pair. There are two ways that EARL policy representations help address the problem of large state spaces: *generalization* and *selectivity*.

8.1.1 POLICY GENERALIZATION

Most EARL policy representations specify the policy at a level of abstraction higher than an explicit mapping from observed states to actions. In the case of rule-based representations, the rule language allows conditions to match sets of states, thus greatly reducing the storage required to specify a policy. It should be noted, however, that the generality of the rules within a policy may vary considerably, from the level of rules that specify an action for a single observed state all the way to completely general rules that recommend an action regardless of the current state. Likewise, in neural net representations, the mapping function is stored implicitly in the weights on the connections of the neural net. In either case, a generalized policy representation facilitates the search for good policies by grouping together states for which the same action is required.

8.1.2 POLICY SELECTIVITY

Most EARL systems have *selective* representations of policies. That is, the EA learns mappings from observed states to recommended actions, usually eliminating explicit information concerning less desirable actions. Knowledge about bad decisions is not explicitly preserved, since policies that make such decisions are selected against by the evolutionary algorithm and are eventually eliminated from the population. The advantage of selective representations is that attention is focused on profitable actions only, reducing space requirements for policies.

Consider our example of the simple EARL operating on the grid world. As the population evolves, policies normally converge to the best actions from a specific state, because of the selective pressure to achieve high fitness levels. For example, the population shown in Table 2 has converged alleles (actions) in states $a3, a5, b2, b5, d3, e1$, and $e2$. Each of these converged state-action pairs is highly correlated with fitness. For example, all policies have converged to action R in state $b2$. Taking action R in state $b2$ achieves a much higher expected return than action D (15 *vs.* 8 from Table 1). Policies that select action D from state $b2$ achieve lower fitness scores and are selected against. For this simple EARL, a snapshot of the population (Table 2) provides an implicit estimate of a corresponding TD value function (Table 4), but the distribution is biased toward the more profitable state-actions pairs.

	a1	a2	a3	a4	a5	b1	b2	b3	b4	b5	c1	c2	c3	c4	c5	d1	d2	d3	d4	d5	e1	e2	e3	e4	e5
R	16	7	?	17	12	8	12	11	11	12	14	7	12	13	9	12	11	12	12	11	?	12	7	?	9
L	9	13	12	11	?	15	?	17	16	?	11	13	12	7	14	11	12	?	11	16	12	?	13	12	16

Table 4: An approximated value function from the population in Table 2. The table displays the average fitness for policies that select each state-action pair and reflects the estimated impact each action has on overall fitness. Given the tiny population size in this example, the estimates are not particularly accurate. Note the question marks in states where actions have converged. Since no policies select the alternative action, the population has no statistics on the impact of these actions on fitness. This is different from simple TD methods, where statistics on all actions are maintained.

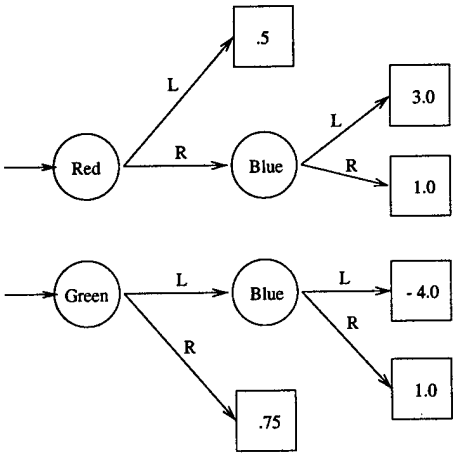


Figure 13: An environment with incomplete state information. The circles represent the states of the world and the colors represent the agent's sensory input. The agent is equally likely to start in the *red* state or the *green* state

8.2 Dealing with Incomplete State Information

Clearly, the most favorable condition for reinforcement learning occurs when the agent can observe the true state of the dynamic system with which it interacts. When complete state information is available, TD methods make efficient use of available feedback by associating reward directly with individual decisions. In real world situations, however, the agent's sensors are more likely to provide only a partial view that may fail to disambiguate many states. Consequently, the agent will often be unable to completely distinguish its current state. This problem has been termed *perceptual aliasing* or the *hidden state* problem. In the case of limited sensory information, it may be more useful to associate rewards with larger blocks of decisions. Consider the situation in Figure 13, in which the agent must act without complete state information. Circles represent the specific states of the world, and the colors represent the sensor information the agent receives within the state. Square nodes represent goal states with the corresponding reward shown inside. In each state, the agent has a choice of two actions (L or R). We further assume that the state transitions are deterministic and that the agent is equally likely to start in either the state with the red or green sensor readings.

In this example, there are two different states that return a sensor reading of *blue*, and the agent is unable to distinguish between them. Moreover, the actions for each *blue* state return very different rewards. A Q function applied to this problem treats the sensor reading of *blue* as one observable

	Value Function Policy	Optimal Policy
Red	R	R
Green	L	R
Blue	R	L
Expected Reward	1.0	1.875

Table 5: The policy and expected reward returned by a converged Q function compared to the optimal policy given the same sensory information.

state, and the rewards for each action are averaged over both *blue* states. Thus, $Q(blue, L)$ and $Q(blue, R)$ will converge to -0.5 and 1, respectively. Since the reward from $Q(blue, R)$ is higher than the alternatives from observable states *red* and *green*, the agent's policy under Q-learning will choose to enter observable state *blue* each time. The final decision policy under Q-learning is shown in Table 5. This table also shows the optimal policy with respect to the agent's limited view of its world. In other words, the policy reflects the optimal choices if the agent cannot distinguish the two blue states.

By associating values with individual observable states, the simple TD methods are vulnerable to hidden state problems. In this example, the ambiguous state information misleads the TD method, and it mistakenly combines the rewards from two different states of the system. By confounding information from multiple states, TD cannot recognize that advantages might be associated with specific actions from specific states, for example, that action L from the top *blue* state achieves a very high reward.

In contrast, since EA methods associate credit with entire policies, they rely more on the net results of decision sequences than on sensor information, that may, after all, be ambiguous. In this example, the evolutionary algorithm exploits the disparity in rewards from the different *blue* states and evolves policies that enter the good *blue* state and avoid the bad one. The agent itself remains unable to distinguish the two *blue* states, but the evolutionary algorithm implicitly distinguishes among ambiguous states by rewarding policies that avoid the bad states.

For example, an EA method can be expected to evolve an optimal policy in the current example given the existing, ambiguous state information. Policies that choose the action sequence R,L when starting in the *red* state will achieve the highest levels of fitness, and will therefore be selected for reproduction by the EA. If agents using these policies are placed in the *green* state and select action L, they receive the lowest fitness score, since their subsequent action, L from the *blue* sensors, returns a negative reward. Thus, many of the policies that achieve high fitness when started in the *red* state will be selected against if they choose L from the *green* state. Over the course of many generations, the policies must choose action R from the *green* state to maximize their fitness and ensure their survival.

We confirmed these hypotheses in empirical tests. A Q-learner using single-step updates and a table-based representation converged to the values in Table 5 in every run. An evolutionary algorithm[4] consistently converged 80% of its population on the optimal policy. Figure 14 shows the average percentage of the optimal policy in the population as a function of time, averaged over 100 independent runs.

Thus even simple EA methods such as EARL_1 appear to be more robust in the presence of hidden states than simple TD methods. However, more refined sensor information could still be helpful. In the previous example, although the EA policies achieve a better average reward than the TD policy, the evolved policy remains unable to procure both the 3.0 and 1.0 rewards from the two *blue* states. These rewards could be realized, however, if the agent could separate the two *blue* states. Thus, any method that generates additional features to disambiguate states presents an important asset to EA

4. We used a binary tournament selection, a 50 policy population, 0.8 crossover probability, and 0.01 mutation rate.

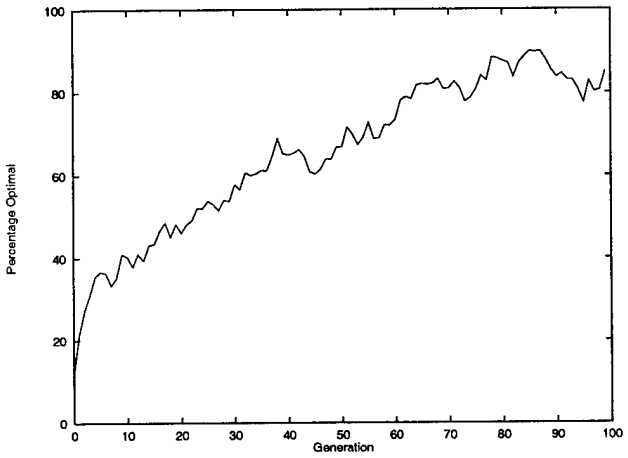


Figure 14: The optimal policy distribution in the hidden state problem for an evolutionary algorithm. The graph plots the percentage of optimal policies in the population, averaged over 100 runs.

methods. Kaelbling et al. (1996) describe several promising solutions to the hidden state problem, in which additional features such as the agent's previous decisions and observations are automatically generated and included in the agent's sensory information (Chrisman, 1992; Lin & Mitchell, 1992; McCallum, 1995; Ring, 1994). These methods have been effective at disambiguating states for TD methods in initial studies, but further research is required to determine the extent to which similar methods can resolve significant hidden state information in realistic applications. It would be useful to develop ways to use such methods to augment the sensory data available in EA methods as well.

8.3 Non-Stationary Environments

If the agent's environment changes over time, the RL problem becomes even more difficult, since the optimal policy becomes a moving target. The classic trade-off between exploration and exploitation becomes even more pronounced. Techniques for encouraging exploration in TD-based RL include adding an *exploration bonus* to the estimated value of state-action pairs that reflects how long it has been since the agent has tried that action (Sutton, 1990), and building a statistical model of the agent's uncertainty (Dayan & Sejnowski, 1996). Simple modifications of standard evolutionary algorithms offer an ability to track non-stationary environments, and thus provide a promising approach to RL for these difficult cases.

The fact that evolutionary search is based on competition within a population of policies suggest some immediate benefits for tracking non-stationary environments. To the extent that the population maintains a diverse set of policies, changes in the environment will bias selective pressure in favor of the policies that are most fit for the current environment. As long as the environment changes slowly with respect to the time required to evaluate a population of policies, the population should be able to track a changing fitness landscape without any alteration of the algorithm. Empirical studies show that maintaining the diversity within the population may require a higher mutation rate than those usually adopted for stationary environments (Cobb & Grefenstette, 1993).

In addition, special mechanisms have been explored in order to make EAs more responsive to rapidly changing environments. For example, (Grefenstette, 1992) suggests maintaining a random search within a restricted portion of the population. The random population elements are analogous to immigrants from other populations with uncorrelated fitness landscapes. Maintaining this source

of diversity permits the EA to respond rapidly to large, sudden changes in the fitness landscape. By keeping the randomized portion of the population to less than about 30% of the population, the impact on search efficiency in stationary environments is minimized. This is a general approach that can easily be applied in EARL systems.

Other useful algorithms that have been developed to ensure diversity in evolving popultions include fitness sharing (Goldberg & Richardson, 1987), crowding (De Jong, 1975), and local mating (Collins & Jefferson, 1991). In Goldberg's fitness sharing model, for example, similar individuals are forced to share a large portion of a single fitness value from the shared solution point. Sharing decreases the fitness of similar individuals and causes evolution to select against individuals in overpopulated niches.

EARL methods that employ distributed policy representations achieve diversity automatically and are well-suited for adaptation in dynamic environments. In a distributed representation, each individual represents only a partial solution. Complete solutions are built by combining individuals. Because no individual can solve the task on its own, the evolutionary algorithm will search for several complementary individuals that together can solve the task. Evolutionary pressures are therefore present to prevent convergence of the population. Moriarty and Miikkulainen (1998) showed how the inherent diversity and specialization in SANE allow it to adapt much more quickly to changes in the environment than standard, convergent evolutionary algorithms.

Finally, if the learning system can detect changes in the environment, even more direct response is possible. In the *anytime learning* model (Grefenstette & Ramsey, 1992), an EARL system maintains a case-base of policies, indexed by the values of the environmental detectors corresponding to the environment in which a given policy was evolved. When an environmental change is detected, the population of policies is partially reinitialized, using previously learned policies selected on the basis of similarity between the previously encountered environment and the current environment. As a result, if the environment changes are cyclic, then the population can be immediately seeded with those policies in effect during the last occurrence of the current environment. By having a population of policies, this approach is protected against some kinds of errors in detecting environmental changes. For example, even if a spurious environmental change is mistakenly detected, learning is not unduly affected, since only a part of the current population of policies is replaced by previously learned policies. Zhou (1990) explored a similar approach based on LCS.

In summary, EARL systems can respond to non-stationary environments, both by techniques that are generic to evolutionary algorithms and by techniques that have been specifically designed with RL in mind.

9. Limitations of EARL

Although the EA approach to RL is promising and has a growing list of successful applications (as outlined in the following section), a number of challenges remain.

9.1 Online Learning

We can distinguish two broad approaches to reinforcement learning —*online learning* and *offline learning*. In online learning, an agent learns directly from its experiences in its operational environment. For example, a robot might learn to navigate in a warehouse by actually moving about its physical environment. There are two problems with using EARL in this situation. First, it is likely to require a large number of experiences in order to evaluate a large population of policies. Depending on how quickly the agent performs tasks that result in some environmental feedback, it may take an unacceptable amount of time to run hundreds of generations of an EA that evaluates hundreds or thousands of policies. Second, it may be dangerous or expensive to permit an agent to perform some actions in its actual operational environment that might cause harm to itself or its environment. Yet it is very likely that at least some policies that the EA generates will be very bad

policies. Both of these objections apply to TD methods as well. For example, the theoretical results that prove the optimality of Q-learning require that every state be visited infinitely often, which is obviously impossible in practice. Likewise, TD methods may explore some very undesirable states before an acceptable value-function is found.

For both TD and EARL, practical considerations point toward the use of offline learning, in which the RL system performs its exploration on simulation models of the environment. Simulation models provide a number of advantages for EARL, including the ability to perform parallel evaluations of all the policies in a population simultaneously (Grefenstette, 1995).

9.2 Rare States

The memory or record of observed states and rewards differs greatly between EA and TD methods. Temporal difference methods normally maintain statistics concerning every state-action pair. As states are revisited, the new reinforcement is combined with the previous value. New information thus supplements previous information, and the information content of the agent's reinforcement model increases during exploration. In this manner, TD methods sustain knowledge of both good and bad state-action pairs.

As pointed out previously, EA methods normally maintain information only about good policies or policy components. Knowledge of bad decisions is not explicitly preserved, since policies that make such decisions are selected against by the evolutionary algorithm and are eventually eliminated from the population. For example, refer once again to Table 4, which shows the implicit statistics of the population from Table 2. Note the question marks in states where actions have converged. Since no policies in the population select the alternative action, the EA has no statistics on the impact of these actions on fitness.

This reduction in information content within the evolving population can be a disadvantage with respect to states that are rarely visited. In any evolutionary algorithm, the value of genes that have no real impact on the fitness of the individual tends to drift to random values, since mutations tend to accumulate in these genes. If a state is rarely encountered, mutations may freely accumulate in the gene that describes the best action for that state. As a result, even if the evolutionary algorithm learns the correct action for a rare state, that information may eventually be lost due to mutations. In contrast, since table-based TD methods permanently record information about all state-action pairs, they may be more robust when the learning agent does encounter a rare state. Of course, if a TD method uses a function approximator such as a neural network as its value function, then it too can suffer from memory loss concerning rare states, since many updates from frequently occurring states can dominate the few updates from the rare states.

9.3 Proofs of Optimality

One of the attractive features of TD methods is that the Q-learning algorithm has a proof of optimality (Watkins & Dayan, 1992). However, the practical importance of this result is limited, since the assumptions underlying the proof (e.g., no hidden states, all state visited infinitely often) are not satisfied in realistic applications. The current theory of evolutionary algorithms provide a similar level of optimality proofs for restricted classes of search spaces (Vose & Wright, 1995). However, no general theoretical tools are available that can be applied to realistic RL problems. In any case, ultimate convergence to an optimal policy may be less important in practice than efficiently finding a reasonable approximation.

A more pragmatic approach may be to ask how efficient alternative RL algorithms are, in terms of the number of reinforcements received before developing a policy that is within some tolerance level of an optimal policy. In the model of *probably approximately correct* (PAC) learning (Valiant, 1984), the performance of a learner is measured by how many learning experiences (e.g., samples in supervised learning) are required before converging to a correct hypothesis within specified error

bounds. Although developed initially for supervised learning, the PAC approach has been extended recently to both TD methods (Fiechter, 1994) and to general EA methods (Ros, 1997). These analytic methods are still in an early stage of development, but further research along these lines may one day provide useful tools for understanding the theoretical and practical advantages of alternative approaches to RL. Until that time, experimental studies will provide valuable evidence for the utility of an approach.

10. Examples of EARL Methods

Finally, we take a look at a few significant examples of the EARL approach and results on RL problems. Rather than attempt an exhaustive survey, we have selected four EARL systems that are representative of the diverse policies representations outlined in Section 5. SAMUEL represents the class of single-chromosome rule-based EARL systems. ALECSYS is an example of a distributed rule-based EARL method. GENITOR is a single chromosome neural-net system, and SANE is a distributed neural net system. This brief survey should provide a starting point for those interested in investigating the evolutionary approach to reinforcement learning.

10.1 SAMUEL

SAMUEL (Grefenstette et al., 1990) is an EARL system that combines Darwinian and Lamarckian evolution with aspects of temporal difference reinforcement learning. SAMUEL has been used to learn behaviors such as navigation and collision avoidance, tracking, and herding, for robots and other autonomous vehicles.

SAMUEL uses a single-chromosome, rule-based representation for policies, that is, each member of the population is a policy represented as a rule set and each gene is a rule that maps the state of the world to actions to be performed. An example rule might be:

IF *range* = [35,45] AND *bearing* = [0,45] THEN SET *turn* = 16 (*strength* 0.8)

The use of a high-level language for rules offers several advantages over low-level binary pattern languages typically adopted in genetic learning systems. First, it makes it easier to incorporate existing knowledge, whether acquired from experts or by symbolic learning programs. Second, it is easier to transfer the knowledge learned to human operators. SAMUEL also includes mechanisms to allow coevolution of multiple behaviors simultaneously. In addition to the usual genetic operators of crossover and mutation, SAMUEL uses more traditional machine learning techniques in the form of Lamarckian operators. SAMUEL keeps a record of recent experiences and will allow operators such as generalization, specialization, covering, and deletion to make informed changes to the individual genes (rules) based on these experiences.

SAMUEL has been used successfully in many reinforcement learning applications. Here we will briefly describe three examples of learning complex behaviors for real robots. In these applications of SAMUEL, learning is performed under simulation, reflecting the fact that during the initial phases of learning, controlling a real system can be expensive or dangerous. Learned behaviors are then tested on the on-line system.

In (Schultz & Grefenstette, 1992; Schultz, 1994; Schultz & Grefenstette, 1996), SAMUEL is used to learn collision avoidance and local navigation behaviors for a Nomad 200 mobile robot. The sensors available to the learning task were five sonars, five infrared sensors, and the range and bearing to the goal, and the current speed of the vehicle. SAMUEL learned a mapping from those sensors to the controllable actions – a turning rate and a translation rate for the wheels. SAMUEL took a human-written rule set that could reach the goal within a limited time without hitting an obstacle only 70 percent of the time, and after 50 generations was able to obtain a 93.5 percent success rate.

In (Schultz & Grefenstette, 1996), the robot learned to herd a second robot to a "pasture". In this task, the learning system used the range and bearing to the second robot, the heading of the

second robot, and the range and bearing to the goal, as its input sensors. The system learned a mapping from these sensors to a turning rate and steering rate. In these experiments, success was measured as the percentage of times that the robot could maneuver the second robot to the goal within a limited amount of time. The second robot implemented a random walk, plus a behavior that made it avoid any nearby obstacles. The first robot learned to exploit this to achieve its goal of moving the second robot to the goal. SAMUEL was given an initial, human-designed rule set with a performance of 27 percent, and after 250 generations was able to move the second robot to the goal 86 percent of the time.

In (Grefenstette, 1996) the SAMUEL EA system is combined with case-based learning to address the adaptation problem. In this approach, called *anytime learning* (Grefenstette & Ramsey, 1992), the learning agent interacts both with the external environment and with an internal simulation. The anytime learning approach involves two continuously running and interacting modules: an execution module and a learning module. The execution module controls the agent's interaction with the environment and includes a monitor that dynamically modifies the internal simulation model based on observations of the actual agent and the environment. The learning module continuously tests new strategies for the agent against the simulation model, using a genetic algorithm to evolve improved strategies, and updates the knowledge base used by the execution module with the best available results. Whenever the simulation model is modified due to some observed change in the agent or the environment, the genetic algorithm is restarted on the modified model. The learning system operates indefinitely, and the execution system uses the results of learning as they become available. The work with SAMUEL shows that the EA method is particularly well-suited for anytime learning. Previously learned strategies can be treated as cases, indexed by the set of conditions under which they were learned. When a new situation is encountered, a nearest neighbor algorithm is used to find the most similar previously learned cases. These nearest neighbors are used to re-initialize the genetic population of policies for the new case. Grefenstette (1996) reports on experiments in which a mobile robot learns to track another robot, and dynamically adapts its policies using anytime learning as its encounters a series of partial system failures. This approach blurs the line between online and offline learning, since the online system is being updated whenever the offline learning system develops an improved policy. In fact, the offline learning system can even be executed on-board the operating mobile robot.

10.2 ALECSYS

As described previously, ALECSYS (Dorigo & Colombetti, 1998) is a distributed rule-based EA that supports an approach to the design of autonomous systems called *behavioral engineering*. In this approach, the tasks to be performed by a complex autonomous systems are decomposed into individual behaviors, each of which is learned via a learning classifier systems module, as shown in Figure 9. The decomposition is performed by the human designer, so the fitness function associated with each LCS can be carefully designed to reflect the role of the associated component behavior within the overall autonomous system. Furthermore, the interactions among the modules is also preprogrammed. For example, the designer may decide that the robot should learn to approach a goal except when a threatening predator is near, in which case the robot should evade the predator. The overall architecture of the set of behaviors can then be set such that the evasion behavior has higher priority than the goal-seeking behavior, but the individual LCS modules can evolve decision rules for optimally performing the subtasks.

ALECSYS has been used to develop behavioral rules for a number of behaviors for autonomous robots, including complex behavior groups such as CHASE/FEED/ESCAPE (Dorigo & Colombetti, 1998). The approach has been implemented and tested on both simulated robots and on real robots. Because it exploits both human design and EARL methods to optimize system performance, this method shows much promise for scaling up to realistic tasks.

10.3 GENITOR

GENITOR (Whitley & Kauth, 1988; Whitley, 1989) is an aggressive, general purpose genetic algorithm that has been shown effective when specialized for use on reinforcement-learning problems. Whitley et al. (1993) demonstrated how GENITOR can efficiently evolve decision policies represented as neural networks using only limited reinforcement from the domain.

GENITOR relies solely on its evolutionary algorithm to adjust the weights in neural networks. In solving RL problems, each member of the population in GENITOR represents a neural network as a sequence of connection weights. The weights are concatenated in a real-valued chromosome along with a gene that represents a crossover probability. The crossover gene determines whether the network is to be mutated (randomly perturbed) or whether a crossover operation (recombination with another network) is to be performed. The crossover gene is modified and passed to the offspring based on the offspring's performance compared to the parent. If the offspring outperforms the parent, the crossover probability is decreased. Otherwise, it is increased. Whitley et al. refer to this technique as *adaptive mutation*, which tends to increase the mutation rate as populations converge. Essentially, this method promotes diversity within the population to encourage continual exploration of the solution space.

GENITOR also uses a so-called "steady-state" genetic algorithm in which new parents are selected and genetic operators are applied after each individual is evaluated. This approach contrasts with "generational" GAs in which the entire population is evaluated and replaced during each generation. In a steady-state GA, each policy is evaluated just once and retains this same fitness value indefinitely. Since policies with lower fitness are more likely to be replaced, it is possible that a fitness based on a noisy evaluation function may have an undesirable influence on the direction of the search. In the case of the pole-balancing RL application, the fitness value depends on the length of time that the policy can maintain a good balance, given a randomly chosen initial state. The fitness is therefore a random variable that depends on the initial state. The authors believe that noise in the fitness function had little negative impact on learning good policies, perhaps because it was more difficult for poor networks to obtain a good fitness than for good networks (of which there were many copies in the population) to survive an occasional bad fitness evaluation. This is an interesting general issue in EARL that needs further analysis.

GENITOR adopts some specific modification for its RL applications. First, the representation uses a real-valued chromosome rather than a bit-string representation for the weights. Consequently, GENITOR always recombines policies between weight definitions, thus reducing potentially random disruption of neural network weights that might result if crossover operations occurred in the middle of a weight definition. The second modification is a very high mutation rate which helps to maintain diversity and promote rapid exploration of the policy space. Finally, GENITOR uses unusually small populations in order to discourage different, competing neural network "species" from forming within the population. Whitley et al. (1993) argue that speciation leads to competing conventions and produces poor offspring when two dissimilar networks are recombined.

Whitley et al. (1993) compare GENITOR to the Adaptive Heuristic Critic (Anderson, 1989, AHC), which uses the TD method of reinforcement learning. In several different versions of the common pole-balancing benchmark task, GENITOR was found to be comparable to the AHC in both learning rate and generalization. One interesting difference Whitley et al. found was that GENITOR was more consistent than the AHC in solving the pole-balancing problem when the failure signals occurs at wider pole bounds (make the problem much harder). For AHC, the preponderance of failures appears to cause all states to overpredict failure. In contrast, the EA method appears more effective in finding policies that obtain better overall performance, even if success is uncommon. The difference seems to be that the EA tends to ignore those cases where the pole cannot be balanced, and concentrate on successful cases. This serves as another example of the advantages associated with search in policy space, based on overall policy performance, rather than paying too much attention to the value associated with individual states.

10.4 SANE

The SANE (Symbiotic, Adaptive Neuro-Evolution) system was designed as a efficient method for building artificial neural networks in RL domains where it is not possible to generate training data for normal supervised learning (Moriarty & Miikkulainen, 1996a, 1998). The SANE system uses an evolutionary algorithm to form the hidden layer connections and weights in a neural network. The neural network forms a direct mapping from sensors to actions and provides effective generalization over the state space. SANE's only method of credit assignment is through the EA, which allows it to apply to many problems where reinforcement is sparse and covers a sequence of decisions. As described previously, SANE uses a distributed representation for policies.

SANE offers two important advantages for reinforcement learning that are normally not present in other implementations of neuro-evolution. First, it maintains diverse populations. Unlike the canonical function optimization EA that converge the population on a single solution, SANE forms solutions in an *unconverged* population. Because several different types of neurons are necessary to build an effective neural network, there is inherent evolutionary pressure to develop neurons that perform different functions and thus maintain several different types of individuals within the population. Diversity allows recombination operators such as crossover to continue to generate new neural structures even in prolonged evolution. This feature helps ensure that the solution space will be explored efficiently throughout the learning process. SANE is therefore more resilient to suboptimal convergence and more adaptive to changes in the domain.

The second feature of SANE is that it explicitly decomposes the search for complete solutions into a search for partial solutions. Instead of searching for complete neural networks all at once, solutions to smaller problems (good neurons) are evolved, which can be combined to form an effective full solution (a neural network). In other words, SANE effectively performs a problem reduction search on the space of neural networks.

SANE has been shown effective in several different large scale problems. In one problem, SANE evolved neural networks to direct or focus a minimax game-tree search (Moriarty & Miikkulainen, 1994). By selecting which moves should be evaluated from a given game situation, SANE guides the search away from misinformation in the search tree and towards the most effective moves. SANE was tested in a game tree search in Othello using the evaluation function from the former world champion program Bill (Lee & Mahajan, 1990). Tested against a full-width minimax search, SANE significantly improved the play of Bill, while examining only a subset of the board positions.

In a second application, SANE was used to learn obstacle avoidance behaviors in a robot arm (Moriarty & Miikkulainen, 1996b). Most approaches for learning robot arm control learn hand-eye coordination through supervised training methods where examples of correct behavior are explicitly given. Unfortunately in domains with obstacles where the arm must make several intermediate joint rotations before reaching the target, generating training examples is extremely difficult. A reinforcement learning approach, however, does not require examples of correct behavior and can learn the intermediate movements from general reinforcements. SANE was implemented to form neuro-control networks capable of maneuvering the OSCAR-6 robot arm among obstacles to reach random target locations. Given both camera-based visual and infrared sensory input, the neural networks learned to effectively combine both target reaching and obstacle avoidance strategies.

For further related examples of evolutionary methods for learning neural-net control systems for robotics, the reader should see (Cliff, Harvey, & Husbands, 1993; Husbands, Harvey, & Cliff, 1995; Yamauchi & Beer, 1993).

11. Summary

This article began by suggesting two distinct approaches to solving reinforcement learning problems; one can search in value function space or one can search in policy space. TD and EARL are examples of these two complementary approaches. Both approaches assume limited knowledge of

the underlying system and learn by experimenting with different policies and using reinforcement to alter those policies. Neither approach requires a precise mathematical model of the domain, and both may learn through direct interactions with the operational environment.

Unlike TD methods, EARL methods generally base fitness on the overall performance of a policy. In this sense, EA methods pay less attention to individual decisions than TD methods do. While at first glance, this approach appears to make less efficient use of information, it may in fact provide a robust path toward learning good policies, especially in situations where the sensors are inadequate to observe the true state of the world.

It is not useful to view the path toward practical RL systems as a choice between EA and TD methods. We have tried to highlight some of the strengths of the evolutionary approach, but we have also shown that EARL and TD, while complementary approaches, are by no means mutually exclusive. We have cited examples of successful EARL systems such as SAMUEL and ALECSYS that explicitly incorporate TD elements into their multi-level credit assignment methods. It is likely that many practical applications will depend on these kinds of multi-strategy approaches to machine learning.

We have also listed a number of areas that need further work, particularly on the theoretical side. In RL, it would be highly desirable to have a better tools for predicting the amount of experience needed by a learning agent before reaching a specified level of performance. The existing proofs of optimality for both Q-learning and EA are of extremely limited practical use in predicting how well either approach will perform on realistic problems. Preliminary results have shown that the tools of PAC analysis can be applied to both EA an TD methods, but much more effort is needed in this direction.

Many serious challenges remain in scaling up reinforcement learning methods to realistic applications. By pointing out the shared goals and concerns of two complementary approaches, we hope to motivate further collaboration and progress in this field.

References

Anderson, C. W. (1989). Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine, 9*, 31–37.

Barto, A. G., Sutton, R. S., & Watkins, C. J. C. H. (1990). Learning and sequential decision making. In Gabriel, M., & Moore, J. W. (Eds.), *Learning and Computational Neuroscience.* MIT Press, Cambridge, MA.

Belew, R. K., McInerney, J., & Schraudolph, N. N. (1991). Evolving networks: Using the genetic algorithm with connectionist learning. In Farmer, J. D., Langton, C., Rasmussen, S., & Taylor, C. (Eds.), *Artificial Life II* Reading, MA. Addison-Wesley.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 183–188 San Jose, CA.

Cliff, D., Harvey, I., & Husbands, P. (1993). Explorations in evolutionary robotics. *Adaptive Behavior, 2*, 73–110.

Cobb, H. G., & Grefenstette, J. J. (1993). Genetic algorithms for tracking changing environments. In *Proc. Fifth International Conference on Genetic Algorithms*, pp. 523–530.

Collins, R. J., & Jefferson, D. R. (1991). Selection in massively parallel genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 249–256 San Mateo, CA. Morgan Kaufmann.

Dayan, P., & Sejnowski, T. J. (1996). Exploration bonuses and dual control. *Machine Learning, 25*(1), 5–22.

De Jong, K. A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Ph.D. thesis, The University of Michigan, Ann Arbor, MI.

Dorigo, M., & Colombetti, M. (1998). *Robot Shaping: An Experiment in Behavioral Engineering.* MIT Press, Cambridge, MA.

Fiechter, C.-N. (1994). Efficient reinforcement learning. In *Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory*, pp. 88–97. Association for Computing Machinery.

Fogel, L. J., Owens, A. J., & Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution.* Wiley Publishing, New York.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA.

Goldberg, D. E., & Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms*, pp. 148–154 San Mateo, CA. Morgan Kaufmann.

Grefenstette, J. J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man & Cybernetics, SMC-16*(1), 122–128.

Grefenstette, J. J. (1987). Incorporating problem specific knowledge into genetic algorithms. In Davis, L. (Ed.), *Genetic Algorithms and Simulated Annealing*, pp. 42–60 San Mateo, CA. Morgan Kaufmann.

Grefenstette, J. J. (1988). Credit assignment in rule discovery system based on genetic algorithms. *Machine Learning, 3*(2/3), 225–245.

Grefenstette, J. J. (1992). Genetic algorithms for changing environments. In Männer, R., & Manderick, B. (Eds.), *Parallel Problem Solving from Nature, 2*, pp. 137–144.

Grefenstette, J. J. (1995). Robot learning with parallel genetic algorithms on networked computers. In *Proceedings of the 1995 Summer Computer Simulation Conference (SCSC '95)*, pp. 352–257.

Grefenstette, J. J. (1996). Genetic learning for adaptation in autonomous robots. In *Robotics and Manufacturing: Recent Trends in Research and Applications, Volume 6*, pp. 265–270. ASME Press, New York.

Grefenstette, J. J. (1997a). Proportional selection and sampling algorithms. In *Handbook of Evolutionary Computation*, chap. C2.2. IOP Publishing and Oxford University Press.

Grefenstette, J. J. (1997b). Rank-based selection. In *Handbook of Evolutionary Computation*, chap. C2.4. IOP Publishing and Oxford University Press.

Grefenstette, J. J., & Ramsey, C. L. (1992). An approach to anytime learning. In *Proc. Ninth International Conference on Machine Learning*, pp. 189–195 San Mateo, CA. Morgan Kaufmann.

Grefenstette, J. J., Ramsey, C. L., & Schultz, A. C. (1990). Learning sequential decision rules using simulation models and competition. *Machine Learning, 5*, 355–381.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* University of Michigan Press, Ann Arbor, MI.

Holland, J. H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In *Machine Learning: An Artificial Intelligence Approach*, Vol. 2. Morgan Kaufmann, Los Altos, CA.

Holland, J. H. (1987). Genetic algorithms and classifier systems: Foundations and future directions. In *Proceedings of the Second International Conference on Genetic Algorithms*, pp. 82–89 Hillsdale, New Jersey.

Holland, J. H., & Reitman, J. S. (1978). Cognitive systems based on adaptive algorithms. In *Pattern-Directed Inference Systems*. Academic Press, New York.

Husbands, P., Harvey, I., & Cliff, D. (1995). Circle in the round: state space attractors for evolved sighted robots. *Robot. Autonomous Systems, 15*, 83–106.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research, 4*, 237–285.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Lee, K.-F., & Mahajan, S. (1990). The development of a world class Othello program. *Artificial Intelligence, 43*, 21–36.

Lin, L.-J., & Mitchell, T. M. (1992). Memory approaches to reinforcement learning in non-Markovian domains. Tech. rep. CMU-CS-92-138, Carnegie Mellon University, School of Computer Science.

McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D. thesis, The University of Rochester.

Moriarty, D. E., & Miikkulainen, R. (1994). Evolving neural networks to focus minimax search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pp. 1371–1377 Seattle, WA. MIT Press.

Moriarty, D. E., & Miikkulainen, R. (1996a). Efficient reinforcement learning through symbiotic evolution. *Machine Learning, 22*, 11–32.

Moriarty, D. E., & Miikkulainen, R. (1996b). Evolving obstacle avoidance behavior in a robot arm. In *From Animals to Animats: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB-96)*, pp. 468–475 Cape Cod, MA.

Moriarty, D. E., & Miikkulainen, R. (1998). Forming neural networks through efficient and adaptive co-evolution. *Evolutionary Computation, 5*(4), 373–399.

Potter, M. A. (1997). *The Design and Analysis of a Computational Model of Cooperative Coevolution*. Ph.D. thesis, George Mason University.

Potter, M. A., & De Jong, K. A. (1995). Evolving neural networks with collaborative species. In *Proceedings of the 1995 Summer Computer Simulation Conference* Ottawa, Canada.

Potter, M. A., De Jong, K. A., & Grefenstette, J. (1995). A coevolutionary approach to learning sequential decision rules. In Eshelman, L. (Ed.), *Proceedings of the Sixth International Conference on Genetic Algorithms* Pittsburgh, PA.

Rechenberg, I. (1964). Cybernetic solution path of an experimental problem. In *Library Translation 1122*. Royal Aircraft Establishment, Farnborough, Hants, Aug. 1965.

Ring, M. B. (1994). *Continual Learning in Reinforcement Environments.* Ph.D. thesis, The University of Texas at Austin.

Ros, J. P. (1997). Probably approximately correct (PAC) learning analysis. In *Handbook of Evolutionary Computation*, chap. B2.8. IOP Publishing and Oxford University Press.

Schaffer, J. D., Caruana, R. A., Eshelman, L. J., & Das, R. (1989). A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 51–60. Morgan Kaufmann.

Schaffer, J. D., & Grefenstette, J. J. (1985). Multi-objective learning via genetic algorithms. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 593–595. Morgan Kaufmann.

Schultz, A. C. (1994). Learning robot behaviors using genetic algorithms. In *Intelligent Automation and Soft Computing: Trends in Research, Development, and Applications*, pp. 607–612. TSI Press, Albuquerque.

Schultz, A. C., & Grefenstette, J. J. (1992). Using a genetic algorithm to learn behaviors for autonomous vehicles. In *Proceedings of the AiAA Guidance, Navigation, and Control Conference* Hilton Head, SC.

Schultz, A. C., & Grefenstette, J. J. (1996). Robo-shepherd: Learning complex robotic behaviors. In *Robotics and Manufacturing: Recent Trends in Research and Applications, Volume 6*, pp. 763–768. ASME Press, New York.

Smith, S. F. (1983). Flexible learning of problem solving heuristics through adaptive search. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pp. 422–425. Morgan Kaufmann.

Sutton, R. (1990). Integrated architectures for learning, planning, and reacting based on approximate dynamic programming. In *Machine Learning: Proceedings of the Seventh International Conference*, pp. 216–224.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning, 3*, 9–44.

Sutton, R. S., & Barto, A. (1998). *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM, 27*, 1134–1142.

Vose, M. D., & Wright, A. H. (1995). Simple genetic algorithms with linear fitness. *Evolutionary Computation, 2*, 347–368.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* Ph.D. thesis, University of Cambridge, England.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning, 8*(3), 279–292.

Whitley, D. (1989). The GENITOR algorithm and selective pressure. In *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 116–121 San Mateo, CA. Morgan Kaufman.

Whitley, D., & Kauth, J. (1988). GENITOR: A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pp. 118–130 Denver, CO.

Whitley, D., Dominic, S., Das, R., & Anderson, C. W. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning, 13*, 259–284.

Wilson, S. W. (1994). ZCS: A zeroth level classifier system. *Evolutionary Computation, 2*(1), 1–18.

Yamauchi, B. M., & Beer, R. D. (1993). Sequential behavior and learning in evolved dynamical neural networks. *Adaptive Behavior, 2*, 219–246.

Zhou, H. (1990). CSM: A computational model of cumulative learning. *Machine Learning, 5*(4), 383–406.

APPENDIX D:

Evolvability in Dynamic Fitness Landscapes: A Genetic Algorithm Approach

John J. Grefenstette
Institute for Biosciences, Bioinformatics and Biotechnology
George Mason University
Manassas, VA 20110
gref@ib3.gmu.edu

Abstract- Evolvability refers to the adaptation of a population's genetic operator set over time. In traditional genetic algorithms, the genetic operator set, consisting of mutation operators, crossover operators, and their associated rates, is usually fixed. Here, we explore the effects of allowing these operators and rates to vary under the influence of selection. This paper focuses on the suitability of alternative mutation models in dynamic landscapes. The mutation models include both traditional models in which all members of the population are subject the same level of mutation and models in which mutation rates are genetically controlled.

1 Introduction

This project takes a genetic algorithm approach to the study of evolvability, meaning the adaptation of a population's genetic operator set over time (Liberman et. al, 1986). In traditional genetic algorithms, the genetic operator set, consisting of mutation operators, crossover operators, and their associated rates, are fixed. Here, we explore the effects of allowing some of these operators and rates to vary under the influence of selection (Bäck, 1997). This study focuses on dynamic fitness landscapes. Genetic algorithms are expected to be well-suited to problems in which the objective function, or fitness landscape, changes over time. In nature, of course, the fitness landscape is constantly changing due to physical changes as well as the co-evolution of competing species.

The study of evolvability may lead to genetic algorithms that are more efficient at searching in dynamic environments. The study of evolvability is also motivated in part by its importance in several research areas, including emerging infectious diseases, environmentally-induced mutagenesis, human genetic disease, cancer and aging. It is hoped that understanding the behavior of several models of evolvability in artificial evolving systems will point the way toward a better understanding of natural mechanisms in evolving systems (Liberman and Feldman, 1986; Wolfe et. al, 1989).

1.1 Previous Work

Research on adaptive operator probabilities in genetic algorithms have usually addressed the optimization of fixed fitness landscapes, and have explored adapting the operator through externally imposed heuristics (Davis, 1989; Bäck, 1992). In contrast, this work continues a line of research concerning self-adaptation of genetic operators (Bäck, 1997). Of course, self-adapting mutation rates are widely used in evolution strategies (Bäck and Schwefel, 1993). This work differs from much of the previous work by focusing on self-adaptation in a genetic algorithm model for searching dynamic fitness landscapes.

In previous studies (Grefenstette, 1992; Cobb and Grefenstette, 1993), we explored modifications to a standard genetic algorithm that would permit the tracking of optima in non-stationary environments. These papers investigated two mechanisms. First, a global hypermutation could occur at intervals, effectively kicking the population into a temporary random exploration mode. Second, a fixed percentage of the population could be replaced by *random immigrants*. The latter mechanism was shown to be effective on some dynamic landscapes, and resulted in little disruption in the case of stationary landscapes. Based on this previous work, we suspect that hypermutation (or, equivalently, immigration) may be an especially useful mutation strategy for adapting to dynamic fitness landscapes. Our previous work was limited to a fixed form of hypermutation and considered a very limited set of test problems. This work aims to extend our previous work by considering a wider range of mutation models as well as a larger set of dynamic landscapes.

2 Methods

2.1 Genetic Algorithms

The genetic algorithm used in these studies is a standard generational model operating on binary genomes. The GA uses proportional selection and 2-point crossover, but with a possibly varying mutation rate, explained below. Since we are considering dynamic environments, all individuals are evaluated in every generation.

In genetic algorithms on fixed fitness landscapes, the usual practice is to scale the fitness by, say, increasing the baseline value of the objective function against which fitness is measured. This is necessary to maintain selective pressure as the population converges toward high fitness regions (Grefenstette, 1986). In a dynamic fitness landscape, such baseline scaling may lead to instabilities since the mean fitness of the population may vary dramatically as the landscape shifts. Therefore, baseline fitness scaling is disabled.

Proc. 1999 Congress on Evolutionary Computation. Washington, DC. July 1999, 2031-2038.

2.2 Models of Evolvability

Evolvability refers to the adaptation of a population's genetic operator set over time. This work explores evolvability by comparing methods that vary operator rates under the influence of selection, mutation and crossover. In particular we focus on the suitability of alternative mutation models in dynamic landscapes. Evolvable crossover operators will be considered in a future study. The mutation models include both traditional models in which all members of the population are subject to the same level of mutation and models in which mutation rates are genetically controlled. Mutation control genes (if any) do not contribute directly to the fitness calculation, but are used to determine the individual's mutation rate. The following mutation models are compared (model parameters are shown in parentheses):

Fixed Mutation Model, FM(m_r): This is the baseline case in which all individuals are subject to a fixed base mutation rate m_r, where mutation rate is defined in terms of the probability of randomly resetting each bit in the genome. That is, a mutation rate of $m_r = 0.5$ means that approximately half of the bits will be reset to random values (not flipped).

Genetic Mutation Model, GM(α_M): The mutation rate is under genetic control. In this work, a 3-bit mutation control gene is interpreted as an integer value k between 0 and 7. The mutation rate for the individual is set to $m_r = \alpha_M + (k/8))$, where α_M is the minimum mutation rate. Preliminary experiments showed that unless a minimum mutation rate was specified, the genetically controlled mutation rate quickly converges to 0. The likely explanation is that the mutation rate levels form a Markov chain, with the mutation rate of zero forming an absorbing state. That is, low mutation levels, once established, are very difficult to overcome, since mutation itself is the primary means for changing the mutation rate. In this study, we set $\alpha_M = 0.03$, so we expect a minimum mutation rate of slightly over one mutation per genome per generation (on a genome of 43 bits).

Fixed Hypermutation Model, FH(h_r, m_r): During each generation, a fixed fraction h_r of the population is subject to hypermutation (e.g. a random reset of the genotype). Based on previous work (Grefenstette, 1992), this study adopts a fixed hypermutation rate of $h_r = 0.2$. That is, 20% of the population is replaced by random individual in each generation. The remaining individuals are subject to the baseline mutation rate m_r. The effect of this mutation model is the same as an influx of immigrants from some other uncorrelated fitness landscapes (Grefenstette, 1992).

Genetic Hypermutation Model, GH(α_H, m_r): In this model the hypermutation rate is under genetic control. As in the GM model, a 3-bit mutation control gene is interpreted as an integer value k between 0 and 7. The probability that the individual undergoes hypermutation is set to $h_r = \alpha_H + (k/8)$. If the individual does not hypermutate, then it is subjected to the base mutation rate m_r. As with GM, preliminary experiments showed that, in the absence of a minimum hypermutation rate, the genetically controlled hypermutation rate consistently converges to 0. Therefore, the minimum hypermutation probability was set to $\alpha_H = 0.05$.

As indicated above, the mutation models are all dependent on the underlying base mutation rate m_r, except GM, in which the mutation rate is entirely governed by the mutation control gene (subject to the minimum mutation rate α_M). These alternative mutation models are compared on a specific class of dynamic landscapes, described in the following section.

2.3 Models of Dynamic Fitness Landscapes

The "no-free-lunch" theorems of Wolpert and Macready (1997) show that any comparison of competing algorithms must be qualified by a specification of the class of problems (i.e., fitness landscapes) under consideration. This insight has led to the development of problem generators that can produce a large and well-characterized set of test landscapes (De Jong, Potter and Spears, 1997; Morrison and De Jong, 1999). For this study, we have developed a problem generator for dynamic landscapes. Each landscape is composed of numerous component landscapes, each of which can change independently. By setting a few runtime parameters that specify how the component landscapes change, the user can generate an indefinite number of distinct dynamic landscapes with controllable characteristics along several dimensions of difficulty.

In this problem generator, individuals are interpreted as points in n-dimensional real space, i.e.,

$$\mathbf{x} = (x_1, x_2, \cdots, x_n)$$

A dynamic landscape is specified as a set $\{g_i\}$ where each g_i is a component landscape consisting of a single, time-varying n-dimensional Gaussian peak. The overall fitness landscape is defined by:

$$f(\mathbf{x}) = \max g_i(\mathbf{x})$$

That is, the fitness of $\mathbf{x}$ is defined as the maximum contribution of any of the component peaks. Each peak g_i is characterized by three time-varying features: (1) its center c_i, (2) its amplitude A_i, and (3) its width σ_i. These features are defined as follows:

- The *center* $c_i(t)$ specifies the coordinates associated with the maximum value of the peak at time t.
- The *amplitude* $A_i(t)$ is the fitness contribution obtained by an individual located at the center of the peak.
- The *width* $\sigma_i(t)$ specifies how the fitness contribution from this peak decreases as a function of the distance from the center of the peak.

Given these parameters, the fitness contribution to point x from peak g_i is defined by the formula:

$$g_i(\mathbf{x}) = A_i(t) \exp(-d(\mathbf{x}, c_i(t))^2/(2\,\sigma_i^2(t)))$$

where $d(\mathbf{x}, c_i(t))$ is the Euclidean distance between x and the peak's center $c_i(t)$.

This formulation for the dynamic landscape offers several useful features. First, the user can easily control the rate of overall change by setting the speed at which the peaks move. Second, the continuous nature of the landscapes makes it easy to visualize (in low dimensions, at least). Third, the fact that each local peak is Gaussian may be advantageous to some local search algorithms, so the landscapes are not specially constructed to favor genetic algorithms. Fourth, although each peak is a Gaussian, the overall landscape may be fairly "rugged" since the fitness is defined as the maximum contribution from all the peaks. This definition produces discontinuities in the derivatives wherever two peaks overlap (see the Figure 1). Finally, this model scales up to provide a wide range of levels of difficulty.

It is planned to make this problem generator available as a stand-alone package for use by other genetic algorithm researchers.

2.4 Computational Experiments

2.4.1 Dynamic Landscape Parameters

Given the flexibility of the problem generator, any empirical study necessarily explores only a small fraction of the space of dynamic fitness landscapes. In this study we generated dynamic landscapes using the following parameters for the problem generator:

- Domain: the 2-dimensional region bounded by (0,0) and (100,100).
- Number of peaks = 256
- Number of optimal peaks = 1
- Initial location of peaks: uniformly distributed over the domain.
- All peaks move in randomly selected directions over time.
- Peak amplitudes are initialized to values chosen uniformly from the interval [10, 50], except that the unique optimal peak has amplitude 100. Amplitudes do not vary over time.
- Peak width = 4. That is, a peak's fitness contribution drops to about 50% of its amplitude at a distance of 4 from its center.

The motion of the individual peaks are controlled by two additional parameters. The distance that a peak moves per generation is selected uniformly from the interval [0.0, max_drift], where max_drift is a run-time parameter. The *punctuation* is the number of generations between changes in the landscape.

In this paper, *gradual landscapes* are defined by setting $max\_drift = 1$ and $punctuation = 1$; that is, all peaks

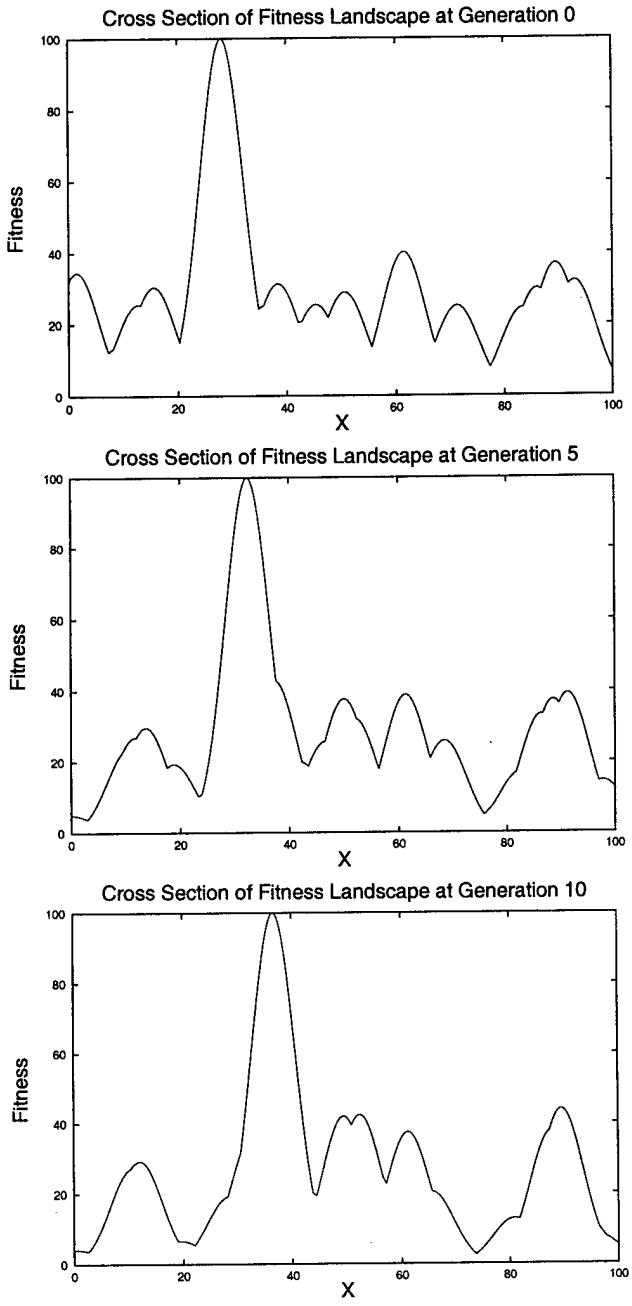


Figure 1: Cross section of the fitness landscape through the maximum height peak at generation 0, generation 5 and generation 10. All peaks move independently in random directions. The number of generations between changes and the size of the average displacement are controllable parameters.

move a small distance each generation. It is difficult to display the movement in a static presentation like this paper, but figure 1 attempts to convey the sort of changes that occur in a gradual landscape. This figure shows a 1-dimensional cross section of the two dimensional fitness landscape, with snapshots at generations 0, 5 and 10. The prominent global peak is moving to the right, while the surrounding peaks each move in random directions. Notice that the overall landscape can have sharp valleys and irregular peaks caused by the intersection of two or more individual peaks.

To define *abrupt landscapes*, we set $max\_drift = 50$ and $punctuation = 20$. Thus, abrupt landscapes are quiescent for 20 generations and then change an amount equivalent to 50 generations of change in gradual landscapes, all at once.

2.4.2 Genetic Representation

In this study, genomes are represented by 43-bit strings. Individuals are interpreted as two-dimensional points $\mathbf{x} = (x_1, x_2)$ in the region bounded by (0,0) and (100,100). The first 20 bits are interpreted as the value for x_1, linearly scaled to the interval [0.0, 100.0). Bits 21-40 are similarly interpreted as the value for x_2. Bits 41-43 are interpreted as a mutation control gene, as described in the definition of the various mutation models above. The mutation control gene does not directly affect the fitness of the individual.

2.4.3 Performance Metrics

There are many possible ways to measure how well a genetic algorithm adapts in a dynamic environment. We report two primary statistics: *online performance* and *current-best performance*.

Online performance is simply the mean raw fitness of all individuals generated during a given run (De Jong, 1975). Online performance is an appropriate measure of how well the GA tracks the changing fitness function, since a perfectly tracking algorithm would quickly identify the optimal peak and keep the population largely converged near the optimum. The problem generator was designed such that the maximum fitness is 100 at all times, so online values closer to 100 indicate successfully convergence to the dynamically drifting optimum.

On the other hand, the ultimate goal in most applications of genetic algorithms is to get as close as possible to the optimal fitness value. Traditionally, *offline performance*, computed as a running average of the best-so-far values, indicates how quickly a genetic algorithm approaches the optimum (De Jong, 1975). However, in a dynamic environment, the value of previously found solutions is irrelevant. Hence, a better measure of optimization is the *current-best* metric, computed as the average value of the best fitness value in the current population.

In practice, trade-offs are likely between these two performance metrics. For example, we would expect that a genetic algorithm with a very high mutation rate may exhibit good current-best performance in some cases (if it can frequently discover the optimum peak by chance), but the online measure should suffer accordingly. On the other hand, a local search algorithm that climbs the nearest peak and tracks it may have a fairly high online performance score, but its current-best score should not be much higher than its online performance. Clearly, an ideal algorithm would maximize both online and current-best performance.

2.4.4 Experimental Design

Several computational experiments are reported below. One set of experiments concerned gradually changing landscapes; the landscape changed slightly after each generation, so the GA saw 200 slightly different (static) fitness functions in each 200 generation run. The second set of experiments concerned abruptly changing landscapes; the landscape changed drastically every 20 generations. In this case, the GA saw 10 different (static) fitness functions in each 200 generation run.

In a given experiment, 100 complete runs of the GA were performed for each mutation model. The 100 runs consisted of 10 runs on each of 10 different dynamically changing landscapes (that is, 10 different gradually changing landscapes or 10 different abruptly changing landscapes, as described above). Each mutation model was tested on the same set of 10 dynamic landscapes. Each run lasted for 200 generations; the population size was 100.

Finally, since all of the four mutation models except GM depend in part on the underlying mutation rate, experiments were repeated for several values of the base mutation rate. In summary, the experiments were performed over all combinations of the following conditions:

- Dynamic landscape class: gradual or abrupt.
- Base mutation rate:

  $m_r = 0.001, 0.003, 0.01, 0.03, 0.06, 0.1, 0.2$ or 0.3.

- Mutation model:
  FM(m_r), GM(0.03), FH(0.2, m_r) or GH(0.05, m_r).

For each experimental condition, statistics were gathered from 100 runs of the genetic algorithm. Results are presented in the next section.

3 Results

A natural first question is: how well do the various evolvability models track the moving optima? Figures 2 and 3 show the time-series of the best-of-generation values averaged[1] over all 100 runs of the genetic algorithms on gradually changing environments and dynamically changing environments, respectively, when the baseline mutation rate is set to $m_r = 0.03$. This baseline mutation value is close to the

[1]Error bars are omitted from all plots to improve readability.

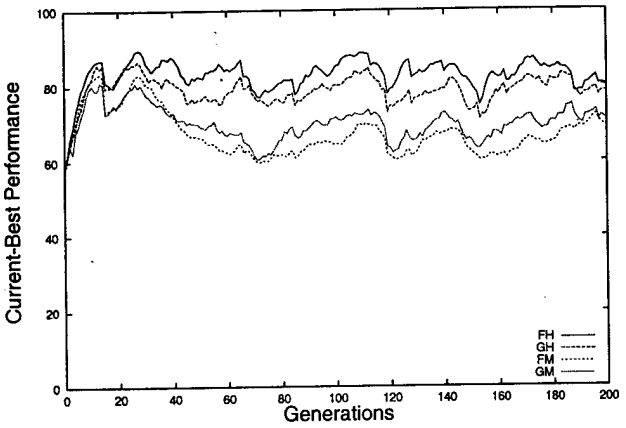


Figure 2: Tracking performance on gradual dynamic landscapes.

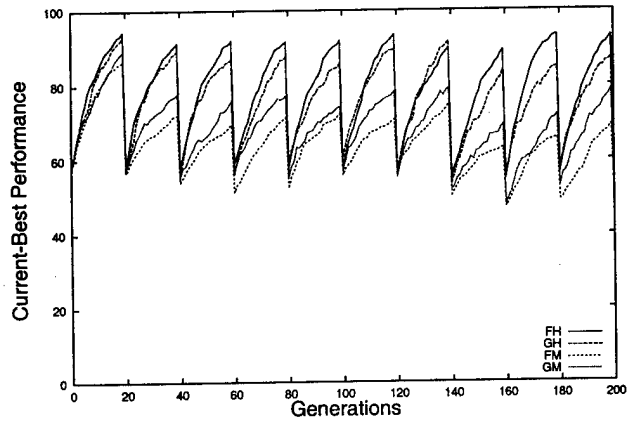


Figure 3: Tracking performance on abrupt dynamic landscapes.

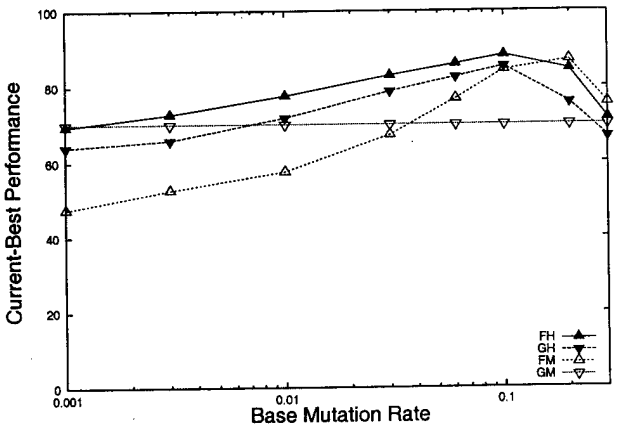


Figure 4: Current-best performance on gradual dynamic landscapes.

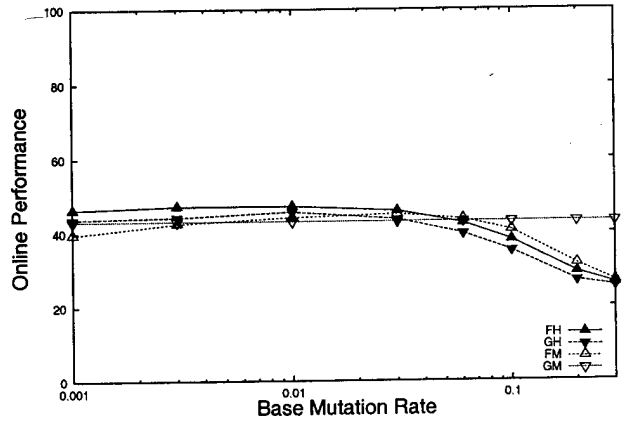


Figure 5: Online performance on gradual dynamic landscapes.

commonly used mutation rate of $1/(\text{genome length})$. These figures suggest no significant difference between the fixed hypermutation model FH and the genetically controlled hypermutation model GH. Likewise, there appears to be no significant difference between the fixed mutation model FM and the genetically controlled mutation model GM. However, there does appear to be significant gap between the hypermutation models and the ordinary mutation models.

In both figures, all mutation models successfully converge toward the optimal peak during the first 20 generations. The hypermutation models FM and GM are able to adapt consistently to both gradual and abrupt changes in the fitness landscape. However, the ordinary mutation models FM and GM appear to lose the ability to track gradually changing landscapes after about 50 generations. Furthermore, these mutation models seem to be unable to explore adequately in the case of abruptly changing landscapes, as shown by their behavior after generation 20 in figure 3.

Given the definition of the four mutation models, we expect that their relative performance will depend on the underlying base mutation rate m_r. Space does not permit the display of similar graphs for all values of the underlying mutation rate, so in the next sections we reduce each such graph to a single value, averaged over all 200 generations.

3.1 Gradually Changing Landscapes

Figures 4 and 5 shows current-best and online performance of the various mutation models on gradually changing landscapes, as a function of the underlying base mutation rate. As expected, the GM curve is flat because GM is independent of the underlying base mutation rate.

Figure 4 shows that the fixed hypermutation model FH gives consistently higher current-best performance than the other mutation models for $m_r \leq 0.1$. FH also gives slightly better online performance than the other mutation models when $m_r \leq 0.03$. The FM model does worse than the

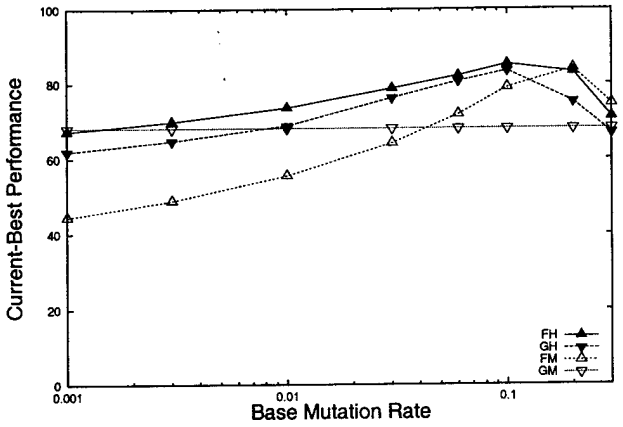


Figure 6: Current-best performance on abrupt dynamic landscapes.

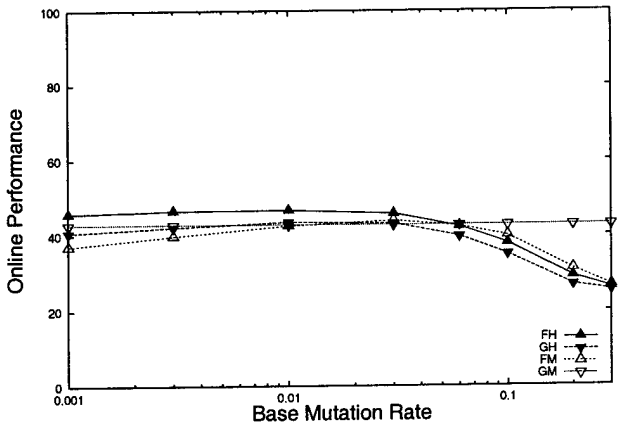


Figure 7: Online performance on abrupt dynamic landscapes.

hypermutation models until the mutation rate is rather high ($m_r \geq 0.3$). The genetically controlled mutation rate model GM shows essentially identical performance as the model FM(0.03). In contrast, the GH model shows a level current-best performance nearly as high as the FH model, and significantly higher than the fixed mutation rate model, perhaps thanks to the effects of the lower bound probability of hypermutation ($\alpha_H = 0.05$).

As might be expected, the online performance for all models drops of as the base mutation rate increases beyond 0.1. The current best performance also declines in this region of parameter space, as all models approach the performance of random search. We now examine how these observations compare with more rapidly changing environments.

3.2 Abruptly Changing Landscapes

Our model of abruptly changing landscapes is that the landscape is fixed for 20 generations, at which time each peak shifts immediately in a randomly selected direction, to a distance approximately 50 times as far as in the gradually changing case. Figures 6 and 7 show performance of the various mutation models on abruptly changing landscapes. These graphs are slightly lower than their counterparts in figures 4 and 5, but the relative strengths of the various mutation models are the same as in the gradually changing case. The FH model again gives the highest current-best performance for base mutation rates up to 0.1. And once again, the genetically controlled hypermutation model performs much more like the fixed hypermutation model than the fixed mutation model, indicating a significant level of genetically controlled hypermutation. Online performance begins to drop off for all models with $m_r \geq 0.03$.

3.3 Genetically Controlled Mutation Models

Finally, we focus in more detail on the models with genetically controlled mutation rates. In particular, we want to know the extent to which the mutation rate can adapt to changes in the environment when the mutation rate is under genetic control.

One hypothesis is that individuals with a high mutation rate have a selective advantage whenever the landscape changing abruptly. This is due to the improved chances of landing on a relatively favorable area by hypermutation, compared to the many individuals clustered near the previous optimal peak and who must "move" away by means of the baseline mutation rate. Under this interpretation, having a high hypermutation rate would be most favorable immediately after an abrupt change in the fitness landscape, and become progressively less favorable as the population converges to the optimal peak. To examine this hypothesis, we collected mutation rate statistics from the runs with abrupt changes in the landscape.

Figure 8 shows a time-series of both the best fitness values in the population and the level of hypermutation in the population, averaged over the 100 runs of the model GH(0.05,0.03) on abruptly changing landscapes. As the population converges to the region near the optimal peak, the level of hypermutation decreases. When the landscape shifts, the level of hypermutation increases.

To examine this phenomenon as a function of the base mutation rates, see figure 9. In this figure, the *pre-adaptive mutation rate* refers to the average genetically controlled hypermutation rate in the population during the first 5 generations after an abrupt shift in the fitness landscape. The *post-adaptive mutation rate* is the average hypermutation rate in the final five generations before the landscape shifts.

There is a consistent drop in hypermutation rate between the pre-adaptive stage and the post-adaptive stage. The difference is most pronounced when the underlying base mutation rate is 0.03, but vanishes as the base rate increases beyond

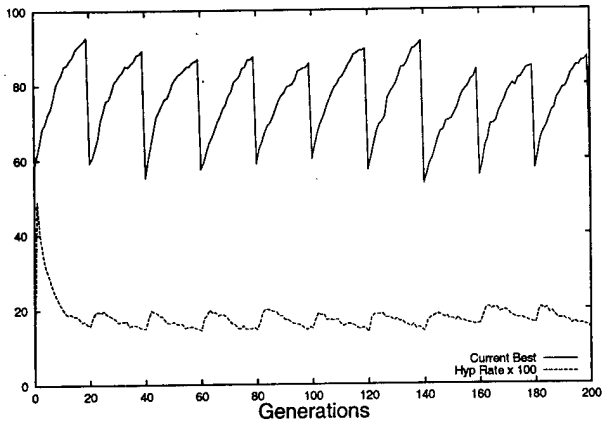


Figure 8: The upper curve show the current-best fitness values. The lower curve shows the percentage of the population undergoing hypermutation, as controlled by the hypermutation control genes. As the population converges to the region near the optimal peak, the level of hypermutation decreases. When the landscape shifts, the level of hypermutation increases.

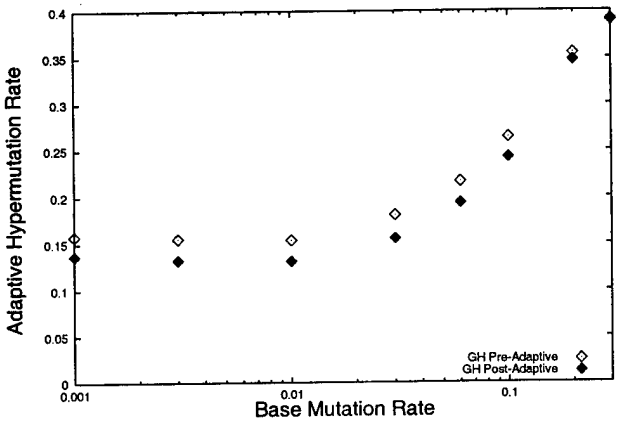


Figure 9: Adaptive mutation rates in genetically controlled hypermutation model on abrupt landscapes.

0.1 (probably because the genetically controlled hypermutation offers less fitness advantage when the base mutation rate is very high). This evidence supports the hypothesis that the hypermutation rate provides a selective advantage just after a major shift in the landscape and that the advantage declines over a period of stability in the fitness landscape.

4 Discussion

This paper focuses on the suitability of alternative mutation models in dynamic landscapes. The mutation models include both traditional models in which all members of the population are subject the same level of mutation and models in which mutation rates are genetically controlled. The alternative mutation models were compared by empirical tests on an artificially generated set of dynamic fitness landscapes. The main positive results are:

- Hypermutation strategies (in which some individuals are randomly reset while the majority are subjected to a base mutation rate) perform well in both gradually changing and abruptly changing landscape.
- The hypermutation rate can be successfully controlled genetically.
- When the hypermutation rate is controlled genetically, it climbs just after an abrupt shift in the fitness landscape, and declines when the landscape remains stable for a period.

Now that the problem generator for dynamic fitness landscapes has been developed, it will be possible to perform much more extensive tests of alternate mutation models. Future work will address the generality of these results on other classes of landscapes. It would be of interest to know if similar phenomena are observed on higher-dimensional landscapes, as well as landscapes with more rugged surfaces. Further studies will also include genetically controlled crossover operators. It is hoped that this line of research will eventually yield useful insights into the application of genetic algorithms to problems that involve dynamic fitness functions, as well as insights into the phenomena of genetically controlled evolvability in biological systems.

Acknowledgments

This work is supported in part by the Office of Naval Research.

Bibliography

Bäck, T. (1992). The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. In *Parallel Problem Solving from Nature, 2*, R. Männer and B. Manderick (Eds.), 85-94, 1992.

Bäck, T. (1997). Self-adaptation. In *The Handbook of Evolutionary Computation*, IOP Publishing and Oxford University Press, 1997.

Bäck, T. and H.-P. Schwefel (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation* **1**(1):1-23.

Cobb H.G. and J.J. Grefenstette (1993). Genetic algorithms for tracking changing environments. *Proc. Fifth International Conference on Genetic Algorithms*, 523-530, Morgan Kaufmann.

Davis, L. D. (1989). Adapting operator probabilities in genetic algorithms. *Proc. Third International Conference on Genetic Algorithms*, J. D. Schaffer (Ed.), 61-69. Morgan Kaufmann.

De Jong, K.A. (1975). *Analysis of the behavior of a class of genetic adaptive systems.* Doctoral dissertation, Department of Computer and Communications Sciences, University of Michigan, Ann Arbor, 1995.

De Jong, K.A, M.A. Potter and W.M. Spears (1997). Using problem generators to explore the effects of epistasis. *Proc. Seventh International Conference on Genetic Algorithms*, 338-345. Morgan Kaufmann.

Grefenstette, J.J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics* **SMC-16**(1), 122-128.

Grefenstette, J.J. (1992). Genetic algorithms for changing environments. *Proceedings of Parallel Problem Solving from Nature-2*, R. Männer and B. Manderick (Eds.), North-Holland, 137-144.

Liberman, U. and M.W. Feldman (1986). Modifiers of mutation rate: A general reduction principle. *Theoretical Population Biology* **30**: 125-142.

Morrison, R.W. and K.A. De Jong (1999). A test problem generator for non-stationary environments. *Proc. 1999 Congress on Evolutionary Computation*, Washington, DC.

Wolfe, K.H., P.M. Sharp and W-H. Li (1989). Mutation rates differ among regions of the mammalian genome. *Nature* **337**: 283-285.

Wolpert, D.H. and W.G. Macready (1997). No free-lunch theorems for optimization. *IEEE Trans. on Evolutionary Computation* **1**(1), 67-82.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT DATE	3. DATES COVERED *(From - To)*
02-08-2002	Final	May 1998 - Aug 2000

4. TITLE AND SUBTITLE
Topics in Evolutionary Computation

5a. CONTRACT NUMBER

5b. GRANT NUMBER
N00173-98-1-G010

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
John J. Grefenstette

5d. PROJECT NUMBER
55-1041-98

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
George Mason University
4400 University Drive
Fairfax, VA 22030-4444

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NRL Code 5510
Naval Research Laboratory
4555 Overlook Avenue, S.W.
Washington, DC 20375-5320

10. SPONSOR/MONITOR'S ACRONYM(S)
NRL Code 5510

11. SPONSORING/MONITORING AGENCY REPORT NUMBER

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This project contributed new principles for the development of intelligent, mobile robots performing complex tasks in unpredictable environments. In the behavior-based approach to robot design, the overall performance of the robot arises through the interaction of multiple, relatively simple, behaviors. The manual design of multiple interacting behaviors is difficult, labor-intensive and error-prone. One way to reduce the effort in the design of behavior-based robots is to develop an evolutionary approach in which the various behaviors, as well as their modes of interaction, evolve over time. Evolution may also provide a basis for the development of strategies for multiple-robot environments, for example, environments in which a robot is expected to adapt its behavior based on the current behavior of other agents or environmental conditions which themselves are changing over time. This project addressed in four complementary areas concerning the effectiveness of evolutionary algorithms for the design of autonomous robots: (1) learning multiple behaviors by asynchronous co-evolution; (2) continuous and embedded learning; (3) comparison with other reinforcement learning methods, and (4) the ability to evolve responses to changing environments. Results in each of these tasks are reported.

15. SUBJECT TERMS
Adaptive systems, evolutionary algorithms, mobile robots

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPONE NUMBER *(Include area code)*
U	U	U	UU	66	