An Evolutionary Algorithm that Constructs Recurrent Neural Networks

Peter J. Angeline, Gregory M. Saunders, and Jordan B. Pollack

Abstract—Standard methods for simultaneously inducing the structure and weights of recurrent neural networks limit every task to an assumed class of architectures. Such a simplification is necessary since the interactions between network structure and function are not well understood. Evolutionary computations, which include genetic algorithms and evolutionary programming, are population-based search methods that have shown promise in many similarly complex tasks. This paper argues that genetic algorithms are inappropriate for network acquisition and describes an evolutionary program, called GNARL, that simultaneously acquires both the structure and weights for recurrent networks. GNARL's empirical acquisition method allows for the emergence of complex behaviors and topologies that are potentially excluded by the artificial architectural constraints imposed in standard network induction methods.

I. INTRODUCTION

In its complete form, network induction entails both parametric and structural learning [1]; i.e., learning both weight values and an appropriate topology of nodes and links. Current connectionist methods to solve this task fall into two broad categories. Constructive algorithms initially assume a simple network and add nodes and links as warranted [2]–[8], while destructive methods start with a large network and prune off superfluous components [9]–[12]. Though these algorithms address the complete problem of network acquisition, they do so in a highly constrained manner. Generally, constructive and destructive methods limit the available architectures in some way. In some of these methods, once an architecture has been explored and determined to be insufficient, a new architecture is adopted and the old becomes topologically unreachable. Others use only a single predefined structural modification, such as “add a fully connected hidden unit,” to generate successive topologies. Such structural hill climbing methods are susceptible to becoming trapped at structural local optima, which places the burden of task induction mostly on the identification of suitable parametric values rather than distributing the burden evenly. In addition, constructive and destructive algorithms investigate only restricted topological subsets rather than the complete class of network architectures. For example, Ash [2] allows only feedforward networks; Fahlman [6] assumes a restricted form of recurrence, and Chen et al. [7] explore only fully connected topologies. As a consequence, these algorithms tend to force a task into an assumed architectural class rather than fitting an appropriate architecture to the task.

The deficiencies of constructive and destructive methods stem from inadequate methods for assigning credit to structural components of a network. The constrained topologies are assumed to limit the complexity of structural and parametric interactions and increase the likelihood of finding a sufficient network to solve the problem. Ideally, such limiting constraints should arise from solving the task rather than be implicit in the algorithm.

This paper presents GNARL, a network induction algorithm that simultaneously acquires both network topology and weight values while making minimal architectural restrictions and avoiding structural hill climbing. The algorithm, described in Section III, is an instance of evolutionary programming [13], [14], a class of evolutionary computations that has been shown to perform well on complex tasks. Section II argues that this class of evolutionary computation is better suited for evolving neural networks than genetic algorithms [15], [16], a more popular variety of evolutionary computations. Finally, Section IV demonstrates GNARL’s ability to create recurrent networks for a variety of problems of interest.

II. EVOLVING CONNECTIONIST NETWORKS

Evolutionary computations are a promising collection of algorithms that hold promise for structural and parametric learning of recurrent networks [17]. These algorithms are distinguished by their reliance on a population of search space positions, rather than a single position, to locate extrema of a function defined over the search space. During one search cycle, or generation, the members of the population are ranked according to a fitness function, and those with higher fitness are probabilistically selected to become parents in the next generation. New population members, called offspring, are created using specialized reproduction heuristics. Using the population, reproduction heuristics, and fitness function, evolutionary computations implement nonmonotonic search methods that perform well in complex multimodal environments. Subclasses of evolutionary computations can be distinguished by examining the specific reproduction heuristics employed.

Genetic algorithms (GA’s) [15], [16] are a popular form of evolutionary computation that rely chiefly on the reproduction heuristic of crossover. This operator forms offspring by

1 Genetic algorithms also employ other operators to manipulate the population, including a form of mutation, but their distinguishing feature is a heavy reliance on crossover.
ANGELINE, SAUNDERS, AND POLLACK: AN EVOLUTIONARY ALGORITHM THAT CONSTRUCTS RECURRENT NEURAL NETWORKS

Fig. 1. The dual representation scheme used in genetic algorithms. The interpretation function maps between the elements in recombination space on which the search is performed and the subset of structures that can be evaluated as potential task solutions.

recombining representational components from two members of the population without regard to content. Such a purely structural approach to creating novel population members assumes that components of all parent representations may be freely exchanged without inhibiting the search process.

Various combinations of GA’s and connectionist networks have been investigated. Much research concentrates on the acquisition of parameters for fixed network architectures [18]-[21]. Other work allows a variable topology, but disassociates structure acquisition from acquisition of weight values by interweaving a GA search for network topology with a traditional parametric training algorithm [22], [23]. Some studies attempt to coevolve both the topology and weight values within the GA framework, but as in the connectionist systems described above, the network architectures are restricted [24]-[26]. In spite of this collection of studies, current theory from both genetic algorithms and connectionism suggests that GA’s are not well-suited for evolving networks. In the following section, the reasons for this mismatch are discussed.

A. Evolving Networks With Genetic Algorithms

Genetic algorithms create new individuals by recombining the representational components of two members of the population. Because of this commitment to structural recombination, GA’s typically rely on two distinct representational spaces (Fig. 1). Recombination space, usually defined over a set of fixed-length binary strings, is the set of structures to which the genetic operators are applied. It is in this space that the search actually occurs. Evaluation space, typically involving a problem-dependent representation, is the set of task-dependent structures that are evaluated on their ability to perform the desired task. When using GA’s to evolve networks, the evaluation space is a set of networks. An interpretation function maps between these two representational spaces. Any set of finite-length bit strings cannot represent all possible networks, thus the evaluation space is restricted to a predetermined set. By design, the dual representation scheme allows the GA to crossover bit strings without any knowledge of their interpretation as networks. The implicit assumption is that the interpretation function will be defined so that bit strings created by the dynamics of the GA in recombination space will map to successively better networks in evaluation space.

The dual representation of GA’s is an important feature for searching in certain environments. For instance, when it is unclear how to search the evaluation space directly, and when there exists an interpretation function such that searching the space of bit strings by crossover leads to good points in evaluation space, then the dual representation is ideal. It is unclear, however, that there exists an interpretation function that makes GA’s beneficial for evolving neural networks. Clearly, the choice of interpretation function introduces a strong bias into the search, typically by excluding many potentially interesting and useful networks (another example of forcing the task into an architecture). Moreover, the benefits of having a dual representation hinge on crossover being an appropriate evolutionary operator for the task for some particular interpretation function; otherwise, the need to translate between the dual representations is an unnecessary complication.

Characterizing tasks for which crossover is a beneficial operator is an open question. Current theory suggests that crossover will tend to recombine short, connected substrings of the bit string representation that correspond to above-average task solutions when evaluated [15], [16]. These substrings are called building blocks, making explicit the intuition that larger structures with high fitness are built out of smaller structures with moderate fitness. Crossover tends to be most effective in environments where the fitness of a member of the population is reasonably correlated with the expected ability of its representational components [27]. Environments where this is not true are called deceptive [28].

There are three forms of deception when using crossover to evolve connectionist networks. The first involves networks that share both a common topology and common weights. Because the interpretation function may be many-to-one, identical networks need not have the same bit string representation (see Fig. 2). Given two such networks, crossover will tend to create offspring that contain repeated components rather than the full complement of either parent’s hidden units. The resulting networks will necessarily perform worse than their parents because they do not possess key computational components for the task. Schaffer et al. [29] term this the competing conventions problem, and point out that the number of competing conventions grows exponentially with the number of hidden units.

The second form of deception involves two networks with identical topologies but different weights. It is well known that for a given task, a single connectionist topology affords
multiple solutions for a task, each implemented by a unique distributed representation spread across the hidden units [30], [31]. While the removal of a small number of nodes has been shown to effect only minor alterations in the performance of a trained network [30], [31], the computational role each node plays in the overall representation of the task solution is determined purely by the presence and strengths of its interconnections. Furthermore, there need be no correlation between distinct distributed representations over a particular network architecture for a given task. This seriously reduces the chance that an arbitrary crossover operation between distinct distributed representations will construct viable offspring regardless of the interpretation function used.

Finally, deception can occur when the parents differ topologically. The types of distributed representations that can develop in a network vary widely with the number of hidden units and the network’s connectivity. Thus, the distributed representations of topologically distinct networks have a greater chance of being incompatible for recombination into viable offspring.

In short, for crossover to be an appropriate operator for evolving networks, the interpretation function must somehow compensate for all the types of deceptiveness described above. This suggests that the complexity of an appropriate interpretation function may more than rival the complexity of the original learning problem. Thus, the prospect of evolving connectionist networks with crossover appears limited in general, and better results should be expected with representations and reproduction heuristics that respect the uniqueness of a network’s distributed representation. Part of this point has been tacitly validated in the genetic algorithm literature by the appearance of non-binary representations for evolving networks (e.g., [32], [33]). Crossover, however, is still commonplace.

B. Networks and Evolutionary Programming

Evolutionary programming (EP) [14], [34] is another form of evolutionary computation more suited to complete network induction. EP systems assume representations that are more natural for the task rather than relying on a singular, general representation as in GA’s. Once an appropriate representation is chosen, representation-dependent mutation operators are defined that create offspring within a specific behavioral locus of the parent (see Fig. 3). EP commits to mutation as the sole reproductive operator for searching over a solution space. Recombination operators are not used. Evolutionary programming is preferable to genetic algorithms when there is no sufficient calculus to guide recombination or when separating the search and evaluation spaces does not afford an advantage.

Relatively few previous EP systems have addressed the problem of evolving connectionist networks. Fogel et al. [35] investigate training feedforward networks on some classic connectionist problems. McDonnell and Waagen [36] use EP to evolve the connectivity of feedforward networks with a constant number of hidden units by evolving both a weight matrix and a connectivity matrix. Fogel [14], [37] uses EP to induce three-layer fully-connected feedforward networks with a variable number of hidden units that employ good strategies for playing tic-tac-toe.

In each of the above studies, the mutation operator alters the parameters of network \( \eta \) by the function:

\[
w = w + N(0, \alpha \varepsilon(\eta)) \quad \forall w \in \eta
\]

where \( w \) is a weight, \( \varepsilon(\eta) \) is the error of the network on the task (typically the mean squared error), \( \alpha \) is a user-defined proportionality constant, and \( N(\mu, \sigma^2) \) is a gaussian variable with mean \( \mu \) and variance \( \sigma^2 \). The implementations of structural mutations in these studies differ somewhat. McDonnell and Waagen [36] randomly select a set of positions in the connectivity matrix for feedforward networks and toggle the associated link’s presence in the network with a probability based on the variance of the incident nodes’ activation over the training set. The structural mutation used by Fogel [14], [37] adds or deletes a single fully connected hidden unit with equal probability. While these methods address complete network induction they too suffer from limiting architectural assumptions.

Evolutionary programming offers distinct advantages over genetic algorithms when evolving networks. First, EP manipulates networks directly, thus obviating the need for a dual representation and a problematic interpretation function. Second, by avoiding recombination between networks when creating offspring, the individuality of each network’s distributed representation is respected.

III. THE GNARL ALGORITHM

GNARL, which stands for Generalized Acquisition of Recurrent Links, is an evolutionary algorithm that nonmonotonically constructs recurrent networks to solve a given task. The name GNARL reflects the types of networks that arise from a generalized network induction algorithm performing both structural and parametric learning. Instead of having uniform or symmetric topologies, the resulting networks have “gnarled” interconnections of hidden units that more accurately reflect constraints inherent in the task.

The general architecture of a GNARL network is straightforward. The input and output nodes are considered to be provided by the task and are immutable by the algorithm; thus each network for a given task always has \( m_{\text{in}} \) input nodes and \( m_{\text{out}} \) output nodes. The number of hidden nodes varies from 0 to a user-supplied maximum \( h_{\text{max}} \). Bias is optional; if provided in an experiment, it is implemented as an additional input node with constant value one. All non-input nodes employ the
A.

by the user. The initial connectivity is chosen randomly (as indicated in the
connections, or full connectivity. Consequently,
of output nodes
computation. but is available as a resource for structural mutations.

Fig. 4. Sample initial network. The number of input nodes (m_in) and number
of output nodes (m_out) is fixed for a given task. The presence of a bias node
(b = 0 or 1) as well as the maximum number of hidden units (h_max) is set
by the user. The initial connectivity is chosen randomly (as indicated in the
text). The disconnected hidden node does not affect this particular network’s
computation, but is available as a resource for structural mutations.

standard sigmoid activation function. Links use real-valued
weights, and must obey three restrictions:

R1: There can be no links to an input node.
R2: There can be no links from an output node.
R3: Given two nodes x and y, there is at most one link
from x to y.

Thus GNARL networks may have no connections, sparse
connections, or full connectivity. Consequently, GNARL’s
search space is:

S = \{ \eta: \eta \text{ is a network with real-valued weights,}
\eta \text{ satisfies } R1 - R3,
\eta \text{ has } m_{in} + b \text{ input nodes, where } b = 1 \text{ if a bias}
node is provided, and 0 otherwise, 
\eta \text{ has } m_{out} \text{ output nodes,}
\eta \text{ has } i \text{ hidden nodes, } 0 \leq i \leq h_{max} \}

R1 - R3 are strictly implementational constraints. Nothing
in the algorithm described below hinges on S being pruned
by these restrictions.

A. Selection, Reproduction, and Mutation of Networks

GNARL initializes the population with randomly generated
networks (see Fig. 4). The number of hidden nodes for each
network is chosen from a uniform distribution over a user-
supplied range. The number of initial links is chosen similarly
from a second user-supplied range. The incident nodes for each
link are chosen in accordance with the structural mutations
described below. Once a topology has been chosen, all links
are assigned random weights, selected uniformly from the
range [-1, 1]. There is nothing in this initialization procedure
that forces a node to have any incident links, let alone for
a path to exist between the input and output nodes. In the experiments below, the number of hidden units for a network
in the initial population was selected uniformly between one
and five, and the number of initial links varied uniformly
between one and 10.

In each generation of search, the networks are first evaluated
by a user-supplied fitness function f : S \rightarrow \mathbf{R},
where \mathbf{R}
represents the reals. Networks scoring in the top 50% are
designated as the parents of the next generation; all other
networks are discarded. This selection method is used in many
EP algorithms, although competitive methods of selection have
also been investigated [14].

Generating an offspring involves three steps: copying the
parent, determining the severity of the mutations to be performed,
and finally mutating the copy. Network mutations are
separated into two classes, corresponding with the types of
learning discussed in [1]. Parametric mutations alter the value
of parameters (link weights) currently in the network, whereas
structural mutations alter the number of hidden nodes and the
presence of links in the network, thus altering the space of
parameters.

1) Severity of Mutations: The severity of a mutation to a
given parent, \eta, is dictated by that network’s temperature,

\[ T(\eta) = 1 - \frac{f(\eta)}{f_{\text{max}}} \]

where \( f_{\text{max}} \) is the maximum fitness for a given task. Thus,
the temperature of a network is determined by how close the
network is to being a solution for the task. This measure of
the network’s performance is used to anneal the structural
and parametric similarity between parent and offspring, so that
networks with a high temperature are mutated severely, and
those with a low temperature are mutated only slightly. This
allows a coarse-grained search initially, and a progressively
finer-grained search as a network approaches a solution to the
task. \( T(\eta) \) is related to the concept of temperature in simulated
annealing [38] where a higher temperature indirectly increases
the variety of states that can be visited by the system.

2) Parametric Mutation of Networks: Parametric
mutations are accomplished by perturbing each weight \( w \) of a
network \eta with gaussian noise, a method motivated by [14],
[37]. In that body of work, weights are modified as follows:

\[ w = w + N(0, \alpha T(\eta)) \quad \forall w \in \eta \]

where \( \alpha \) is a user-defined proportionality constant, and
\( N(\mu, \sigma^2) \) is a gaussian random variable, as before. While large
parametric mutations are occasionally necessary to escape
parametric local minima during search, it is more likely they
will adversely affect the offspring’s ability to perform better
than its parent. To compensate, GNARL updates weights using
a variant of (3). First, the instantaneous temperature \( \bar{T} \) of the
network is computed:

\[ \bar{T}(\eta) = U(0, 1)T(\eta) \]

where \( U(0, 1) \) is a uniform random variable over the interval
[0, 1]. This new temperature, varying from 0 to \( \bar{T}(\eta) \), is then
substituted into (3):

\[ w = w + N(0, \alpha \bar{T}(\eta)) \quad \forall w \in \eta \]

In essence, this modification lessens the frequency of large
parametric mutations without disallowing them completely. In
the experiments described below, \( \alpha \) is one.

3) Structural Mutation of Networks: The structural
mutations used by GNARL alter the number of hidden nodes and
the connectivity between all nodes, subject to restrictions
\( R1 - R3 \) discussed earlier. To avoid radical jumps in fitness
from parent to offspring, structural mutations attempt to
preserve the behavior of a network. For instance, new links
are initialized with zero weight, leaving the behavior of the
modified network unchanged. Similarly, hidden units are
added to the network without any incident connections.
Links must be added by future structural mutations to
determine how to incorporate the new computational unit. Unfortunately, achieving this behavioral continuity between parent and child is not so simple when removing a hidden node or link. Consequently, the deletion of a node involves the complete removal of the node and all incident links with no further modification to compensate for the behavioral change. Similarly, deleting a link removes that parameter from the network.

The selection of which node to remove is uniform over the collection of hidden nodes. Addition or deletion of a link is slightly more complicated. A parameter identifies the likelihood that a link to be added or deleted will be incident with an input or output node. Blasing the link selection process in this way is necessary when there is a large differential between the number of hidden nodes and the number of input or output nodes. This parameter was set to 0.2 in the experiments described in the next section.

Research in [14] and [37] uses the heuristic of adding or deleting at most a single fully connected node per structural mutation. Therefore, it is possible for this method to become trapped at a structural local minima for an indefinite time, although this is less probable than in nonevolutionary algorithms given that several topologies may be present in the population. In order to more effectively search the range of network architectures, GNARL uses a severity of mutation for each separate structural mutation. A unique user-defined interval specifying a range of modification is associated with each of the four structural mutations. Given an interval of \([\Delta_{\text{min}}, \Delta_{\text{max}}]\) for a particular structural mutation, the number of modifications of this type made to an offspring is given by:

\[
\Delta_{\text{min}} + \left[ U(0,1) \frac{\Delta_{\text{max}} - \Delta_{\text{min}}}{2} \right]
\]

Thus the number of modifications varies uniformly over a shrinking interval based on the parent network’s fitness. In the experiments below, the maximum number of nodes added or deleted was five. The minimum number for each interval was always one.

B. Fitness of a Network

In evolving networks to perform a task, GNARL does not require an explicit target vector—all that is needed is the feedback given by the fitness function \(f\). But if such a vector is present, as in supervised learning, there are many ways of transforming it into a measure of fitness. For example, given a training set \(\{(x_1, y_1), (x_2, y_2), \ldots\}\), three possible measures of fitness for a network \(\eta\) are sum of square errors (7), sum of absolute errors (8), and sum of exponential absolute errors (9):

\[
\sum_i (y_i - \text{Out}(\eta, x_i))^2
\]

\[
\sum_i |y_i - \text{Out}(\eta, x_i)|
\]

\[
\sum_i \exp(|y_i - \text{Out}(\eta, x_i)|)
\]

Furthermore, because GNARL explores the space of networks by mutation and selection, the choice of fitness function does not alter the mechanics of the algorithm. To show GNARL’s flexibility, each of these fitness functions will be demonstrated in the experiments below.

IV. EXPERIMENTS

In this section, GNARL is applied to several problems of interest. The goal in this section is to demonstrate the abilities of the algorithm on problems from language induction to search and collection. The various parameter values for the program are set as described above, unless otherwise noted.

A. Williams’ Trigger Problem

As an initial test, GNARL induced a solution for the enable-trigger task proposed in [39]. Consider the finite state generator shown in Fig. 5. At each time step the system receives two input bits, \((a, b)\), representing “enable” and “trigger” signals, respectively. This system begins in state \(S_1\), and switches to state \(S_2\) only when enabled by \(a = 1\). The system remains in \(S_2\) until it is triggered by \(b = 1\), at which point it outputs \(1\) and resets the state to \(S_1\). So, for instance, on an input stream \(\{(0,0), (0,1), (1,1), (0,1)\}\), the system will output \(\{0,0,0,1\}\) and end in \(S_1\). This simple problem allows an indefinite amount of time to pass between the enable and the trigger inputs; thus no finite length sample of the output stream will indicate the current state of the system. This forces GNARL to develop networks that can preserve state information indefinitely.

The fitness function used in this experiment was the sum of exponential absolute errors (9). Population size was 50 networks with the maximum number of hidden units restricted to six. A bias node was provided in each network in this initial experiment, ensuring that an activation value of 1 was always available. Note that this does not imply that each node had a nonzero bias; links to the bias node had to be acquired by structural mutation.

Training began with all two input strings of length two, shown in Table I. After 118 generations (3000 network evaluations.), GNARL evolved a network that solved this task for the strings in Table I within a tolerance of 0.3 on the output units. The training set was then increased to include all 64 input strings of length three and evolution of the networks was allowed to continue. After an additional 422 generations, GNARL once again found a suitable network. At this point, the difficulty of the task was increased a final time by training on all 256 strings of length four. After another 225 generations

2 Number of networks evaluated in a run is popsize * 0.5 * generations * popsize), giving 3000 network evaluations for this trial.
TABLE I
INITIAL TRAINING DATA FOR ENABLE-TRIGGER TASK

<table>
<thead>
<tr>
<th>Input</th>
<th>Target output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0), (0, 1)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>(0, 0), (1, 0)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>(0, 0), (1, 1)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>(0, 1), (0, 0)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>(0, 1), (0, 1)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>(0, 1), (1, 0)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>(0, 1), (1, 1)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>(0, 1), (1, 0)</td>
<td>(0, 0)</td>
</tr>
</tbody>
</table>

Fig. 6. Connectivity of two recurrent networks found in the enable-trigger experiment. (a) The best network of generation 1. (b) The best network of generation 765. This network solves the task for all strings of length eight.

(~ 20000 network evaluations in total) GNARL once again found a network to solve this task, shown in Fig. 6b. Note that there are two completely isolated nodes. Given the fitness function used in this experiment, the two isolated nodes do not affect the network’s viability. To investigate the generalization of this network, it was tested over all 4096 unique strings of length six. The outputs were rounded off to the nearest integer, testing only the network’s separation of the strings. The network performed correctly on 99.5% of this novel set, generating incorrect responses for only 20 strings.

Figure 7 shows the connectivity of the population member with the best fitness for each generation over the course of the run. Initially, the best network is sparsely-connected and remains sparsely-connected throughout most of the run. At about generation 400, the size and connectivity increases dramatically only to be overtaken by the relatively sparse architecture shown in Fig. 6(b) on the final generation. Apparently, this sparsely connected network evolved more quickly than the full architectures that were best in earlier generations. The oscillations between different network architectures throughout the run reflects the development of such competing architectures in the population.

B. Inducing Regular Languages

A current topic of research in the connectionist community is the induction of finite state automata (FSA’s) by networks with second-order recurrent connections. For instance, Pollack [40] trains sequential cascaded networks (SCN’s) over a test set of languages, provided in [41] and shown in Table II, using a variation of backpropagation. An interesting result of this work is that the number of states used by the network to implement finite state behavior is potentially infinite. Other studies using the training sets in [41] have investigated various network architectures and training methods, as well as algorithms for extracting FSA’s from the trained architectures [42]-[45].

An explicit collection of positive and negative examples, shown in Table III, that pose specific difficulties for inducing the intended languages is offered in [41]. Notice that the training sets are unbalanced, incomplete, and vary widely in their ability to strictly define the intended regular language. GNARL’s ability to learn and generalize from these training sets was compared against the training results reported for the second-order architecture used in [42]. Notice that all the languages in Table II require recurrent connections in order to induce the language completely. The type of recurrence needed for each language varies. For instance, languages 1 through 4 require an incorrect input be remembered indefinitely, forcing the network to develop an analog version of a trap state. Networks for language 6, however, must parse and count individual inputs, potentially changing state from accept to reject or vice versa on each successive input.

The results obtained in [42] are summarized in Table IV. The table shows the number of networks evaluated to learn the training set and the accuracy of generalization for the learned network to the intended regular language. Accuracy is measured as the percentage of strings of length 10 or less that are correctly classified by the network. For comparison,
the table lists both the average and best performance of the five runs reported in [42].

This experiment used a population of 50 networks, each limited to at most eight hidden units. Each run lasted at most 1000 generations, allowing a maximum of 25050 networks to be evaluated for a single data set. Two experiments were run for each data set, one using the sum of absolute errors (SAE) and the other using sum of square errors (SSE). The error for a particular string was computed only for the final concatenation of the trailing null symbols was used to identify the end of the string and allow input of the null string, a method also used in [42]. Each network had a single input and output and no bias node was provided. The three possible logical inputs for this task, 0, 1, and null, were represented by activations of -1, 1, and 0, respectively. The tolerance for the output value was 0.1, as in [42].

Table V shows, for both fitness functions, the number of evaluations until convergence and the accuracy of the best evolved network. Only four of the runs, each of those denoted by a ‘+’ in the table, failed to produce a network with the specified tolerance in the allotted 1000 generations. In the runs using SAE, the two runs that did not converge had not separated a few elements of the associated training set and appeared to be far from discovering a network that could correctly classify the complete training set. Both of the uncompleted runs using SSE successfully separated the data sets but had not done so to the 0.1 tolerance within the 1000 generation limit. Fig. 8 compares the number of evaluations by GNARL to the average number of evaluations reported in [42]. As the graph shows, GNARL consistently evaluates more networks, but not a disproportionate number. Considering that the space of networks being searched by GNARL is much larger than the space being searched by [42], these numbers appear to be within a tolerable increase.

The graph of Fig. 9 compares the accuracy of the GNARL networks to the average accuracy found in [42] over five runs. The GNARL networks consistently exceeded the average accuracy found in [42]. These results demonstrate GNARL’s ability to simultaneously acquire the topology and weights of recurrent networks, and that this can be done within a comparable number of network evaluations as training a network with static architecture on the same task. GNARL also appears to generalize better consistently, possibly due to its selective inclusion and exclusion of some links.
Fig. 9. Percentage accuracy of evolved networks on languages in Table II. GNARL (using SAE and SSE fitness measures) compared to average accuracy of the five runs in [42].

Fig. 10. The ant problem. The trail is connected initially, but becomes progressively more difficult to follow. The underlying 2-D grid is toroidal, so that position "A" is the first break in the trail—it is simple to reach this point. Positions "B" and "C" indicate the only two positions along the trail where the ant discovered in run 1 behaves differently from the 5-state FSA of [46] (see Fig. 13).

C. The Ant Problem

GNARL was tested on a complex search and collection task—the Tracker task described in [46], and further investigated in [47]. In this problem, a simulated ant is placed on a two-dimensional toroidal grid that contains a trail of food. The ant traverses the grid, collecting any food it contacts along the way. The goal of the task is to discover an ant that collects the maximum number of pieces of food in a given time period (see Fig. 10).

Following [46], each ant is controlled by a network with two input nodes and four output nodes (Fig. 11). The first input node denotes the presence of food in the square directly in front of the ant; the second denotes the absence of food in this same square, restricting the possible legal inputs to the network to (1, 0) or (0, 1). Each of the four output units corresponds to a unique action: move forward one step, turn left 90°, turn right 90°, or no-op. At each step, the action whose corresponding output node has maximum activation is performed. As in the original study [46], no-op allows the ant to remain at a fixed position while activation flows along recurrent connections. Fitness is defined as the number of grid positions cleared within 200 time steps. The task is difficult because simple networks can perform surprisingly well; the network shown in Fig. 11 collects 42 pieces of food before spinning endlessly in place at position P, illustrating a very high local maximum in the search space.

The experiment used a population of 100 networks, each limited to at most nine hidden units, and did not provide a bias node. In the first run (2090 generations), GNARL found a network [see Fig. 12(b)] that clears 81 grid positions within the 200 time steps. When this ant is run for an additional 119 time steps, it successfully clears the entire trail. To understand how the network traverses the path of food, consider the simple FSA shown in Fig. 13, hand-crafted in [46] as an approximate solution to the problem. This simple machine receives a score of 81 in the allotted 200 time steps, and clears the entire trail only five time steps faster than the network in Fig. 12(b). A step by step comparison indicates there is only a slight difference between the two. GNARL's evolved network follows the general strategy embodied by this FSA at all but two places, marked as positions B and C in Fig. 10. Here the evolved network makes a few additional moves, accounting for the slightly longer completion time.

Fig. 14 illustrates the strategy the network uses to implement the FSA by showing the state of the output units of the network over three different sets. Each point is a triple of the form (move, right, left).³ Figure 14(a) shows the result of supplying to the network 200 "food" inputs—a fixed point that executes

³No-op is not shown because it was never used in the final network.
Fig. 13. FSA hand-crafted for the Tracker task in [46]. The large arrow indicates the initial state. This simple system implements the strategy "move forward if there is food in front of you, otherwise turn right four times, looking for food. If food is found while turning, pursue it, otherwise, move forward one step and repeat." This FSA traverses the entire trail in 314 steps, and gets a score of 81 in the allotted 200 time steps.

"Move." Figure 14(b) shows the sequence of states reached when 200 "no food" signals are supplied to the network—a collection of points describing a limit cycle of length five that repeatedly executes the sequence "Right, Right, Right, Right, Move." These two attractors determine the response of the network to the task [Fig. 14(c), (d)]; the additional points in Fig. 14(c) are transients encountered as the network alternates between these attractors. The differences in the number of steps required to clear the trail between the FSA of Fig. 13 and GNARL's network arise due to the state of the hidden units when transferring from the "food" attractor to the "no food" attractor.

However, not all evolved network behaviors are so simple as to approximate an FSA [40]. In a second run (1595 generations) GNARL induced a network that cleared 82 grid points within the 200 time steps. Fig. 15 demonstrates the behavior of this network. Once again, the "food" attractor, shown in Fig. 15(a), is a single point in the space that always executes "Move." The "no food" behavior, however, is not an FSA; instead, it is a quasiperiodic trajectory of points shaped like a "D" in output space [see Fig. 15(b)]. The placement of the "D" is in the "Move / Right" corner of the space and encodes a complex alternation between these two operations [see Fig. 15(d)].

In contrast, research in [46] uses a genetic algorithm on a population of 65 536 bit strings with a direct encoding to evolve only the weights of a neural network with five hidden units to solve this task. The particular network architecture in [46] uses Boolean threshold logic for the hidden units and an identity activation function for the output units. The first GNARL network was discovered after evaluating a total of 104 600 networks, while the second was found after evaluating 79 850. The experiment reported in [46] discovered a comparable network after about 17 generations. Given [46] used a population size of 65 536 and replaced 95% of the population each generation, the total number of network evaluations to acquire the equivalent network was 2 123 942. This is 10.74 and 14.07 times the number of networks evaluated by GNARL in the two runs. In spite of the differences between the two studies, this significant reduction in the number of evaluations provides empirical evidence that crossover may not be best suited to the evolution of networks.

V. CONCLUSION

Allowing the task to specify an appropriate architecture for its solution should, in principle, be the defining aspect of the complete network induction problem. By restricting the space of networks explored, constructive, destructive, and genetic algorithms only partially address the problem of topology acquisition. GNARL's architectural constraints
The dynamics of the algorithms guided by the task constraints represented in the fitness function allow GNARL to empirically determine an appropriate architecture. Over time, the continual cycle of test-prune-reproduce will constrain the population to only those architectures that have acquired the task most rapidly. Inappropriate networks will not be indefinitely competitive and will be removed from the population eventually.

Complete network induction must be approached with respect to the complex interaction between network topology, parametric values, and task performance. By fixing topology, gradient descent methods can be used to discover appropriate solutions. But the relationship between network structure and task performance is not well understood, and there is no "backpropagation" through the space of network architectures. Instead, the network induction problem is approached with heuristics that, as described above, often restrict the available architectures, the dynamics of the search mechanism, or both. Artificial architectural constraints (such as "feedforwardness") or overly constrained search mechanisms can impede the induction of entire classes of behaviors, while forced structural liberties (such as assumed full recurrence) may unnecessarily increase structural complexity or learning time. By relying on a simple stochastic process, GNARL strikes a middle ground between these two extremes, allowing the network's complexity and behavior to emerge in response to the requirements of the task.

ACKNOWLEDGMENT

This research has been partially supported by ONR grants N00014-92-J-1195 and N00014-93-1-0059. We are indebted to Ed Large, Dave Stucki, and especially John Kolen for proofreading help and discussions during the development of this research. We would also like to thank our anonymous reviewers, David Fogel, and the attendees of Connectfest '92 for feedback on preliminary versions of this work.

REFERENCES

Gregory M. Saunders received the B.S. degree in mathematics in 1988 from Ohio State University, Columbus, Ohio. From 1988 to 1991 he held a National Science Foundation fellowship, and in 1989 received the M.S. degree in computer science from Ohio State University. He is currently a Ph.D. candidate in computer science at Ohio State University, Columbus, Ohio. His research interests include evolutionary computation, neural networks, and connectionist methods of behavior-based control.

Jordan B. Pollack received the Ph.D. degree in 1987 from the University of Illinois and is now an assistant professor in the Computer and Information Sciences Department of The Ohio State University, where his interests span cognitive science, artificial intelligence, connectionist and neural networks, dynamical systems, and artificial life.