An Evolutionary Algorithm that Constructs Recurrent Neural Networks

Peter J. Angeline, Gregory M. Saunders and Jordan B. Pollack
Laboratory for Artificial Intelligence Research
Computer and Information Science Department
The Ohio State University
Columbus, Ohio 43210
pja@cis.ohio-state.edu
saunders@cis.ohio-state.edu
pollack@cis.ohio-state.edu

Abstract

Standard methods for inducing both the structure and weight values of recurrent neural networks fit an assumed class of architectures to every task. This simplification is necessary because the interactions between network structure and function are not well understood. Evolutionary computation, which includes genetic algorithms and evolutionary programming, is a population-based search method that has shown promise in such 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. This algorithm'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.

To Appear in:

IEEE Transactions on Neural Networks

An Evolutionary Algorithm that Constructs Recurrent Neural Networks

Peter J. Angeline, Gregory M. Saunders and Jordan B. Pollack
Laboratory for Artificial Intelligence Research
Computer and Information Science Department
The Ohio State University
Columbus, Ohio 43210
pja@cis.ohio-state.edu
saunders@cis.ohio-state.edu
pollack@cis.ohio-state.edu

Abstract

Standard methods for inducing both the structure and weight values of recurrent neural networks fit an assumed class of architectures to every task. This simplification is necessary because the interactions between network structure and function are not well understood. Evolutionary computation, which includes genetic algorithms and evolutionary programming, is a population-based search method that has shown promise in such 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. This algorithm'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.

1.0 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 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 problem of topology acquisition, they do so in a highly constrained manner. Because they monotonically modify network structure, constructive and destructive methods limit the traversal of the available architectures in that once an architecture has been explored and determined to be insufficient, a new architecture is adopted, and the old becomes topologically unreachable. Also, these methods often use only a single predefined structural modification, such as "add a fully connected hidden unit," to generate successive topologies. This is a form of *structural hill climbing*, which is susceptible to becoming trapped at structural local minima. In addition, constructive and destructive algorithms make simplifying architectural assumptions to facilitate network induction. 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. This creates a situation in which the task is forced into the architecture rather than the architecture being fit to the task.

These deficiencies of constructive and destructive methods stem from inadequate methods for assigning credit to structural components of a network. As a result, the heuristics used are overly-constrained to increase the likelihood of finding any topology to solve the problem. Ideally, the constraints for such a solution should come from 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 3, is an instance of evolutionary programming [13, 14], a class of evolutionary computation that has been shown to perform well at function optimization. Section 2 argues that this class of evolutionary computation is better suited for evolving neural networks than genetic algorithms [15, 16], a more popular class of evolutionary computation. Finally, section 4 demonstrates GNARL's ability to create recurrent networks for a variety of problems of interest.

2.0 Evolving Connectionist Networks

Evolutionary computation provides a promising collection of algorithms 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 computation implements a nonmonotonic search that performs well in complex multimodal environments. Classes of evolutionary computation can be distinguished by examining the specific reproduction heuristics employed.

Genetic algorithms (GAs) [15, 16] are a popular form of evolutionary computation that rely chiefly on the reproduction heuristic of *crossover.*[1] This operator forms offspring by recombining representational components from two members of the population without regard to content. This 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 GAs and connectionist networks have been investigated. Much research concentrates on the acquisition of parameters for a fixed network architecture (e.g., [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 (e.g., backpropagation) over weights (e.g., [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 (e.g., [24 - 26]). In spite of this collection of studies, current theory from both genetic algorithms and connectionism suggests that GAs are not well-suited for evolving networks. In the following section, the reasons for this mismatch are explored.

---

[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.

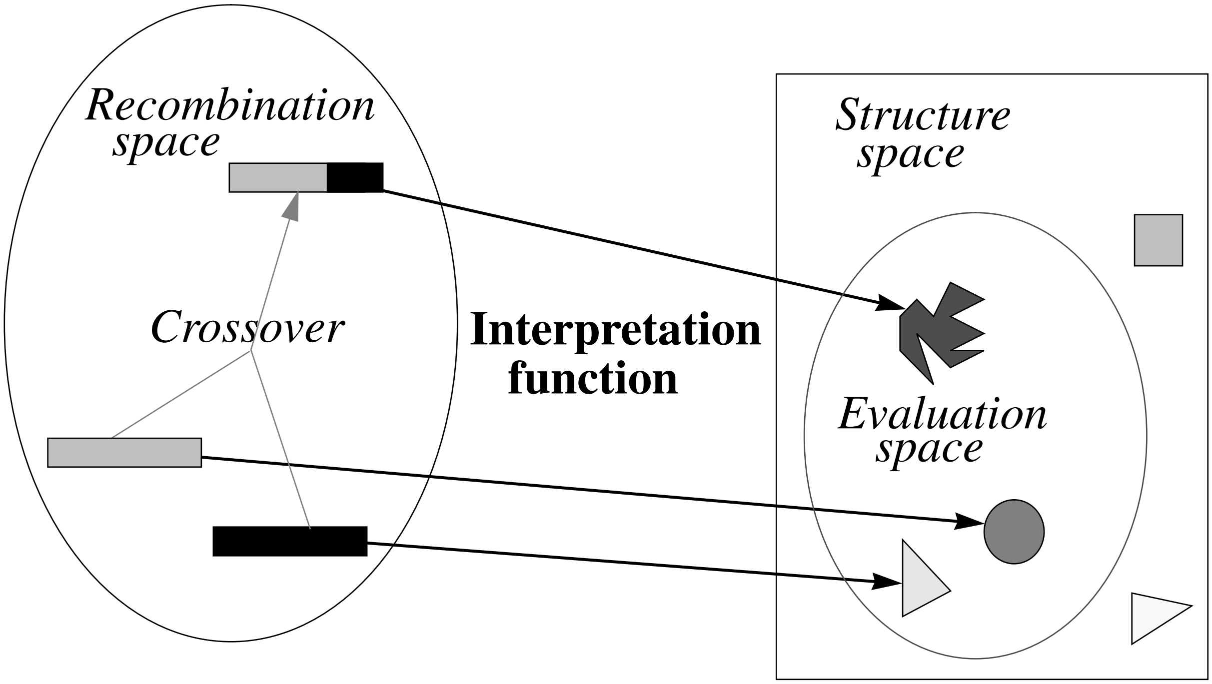


Figure 1. *The dual representation scheme used in genetic algorithms. The interpretation function maps between the elements in recombination space on which the search is performed and the subset of structures that can be evaluated as potential task solutions.*

2.1 Evolving Networks with Genetic Algorithms

Genetic algorithms create new individuals by recombining the representational components of two member of the population. Because of this commitment to structural recombination, GAs typically rely on two distinct representational spaces (Figure 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 here that the search actually occurs. *Evaluation space*, typically involving a problem-dependent representation, is the set of structures whose ability to perform a task is evaluated. In the case of using GAs to evolve networks, evaluation space is comprised of 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 of networks. By design, the dual representation scheme allows the GA to crossover the bit strings without any knowledge of their interpretation as networks. The implicit assumption is that the interpretation function is defined so that the bit strings created by the dynamics of the GA will map to successively better networks.

The dual representation of GAs 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 dual representation 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

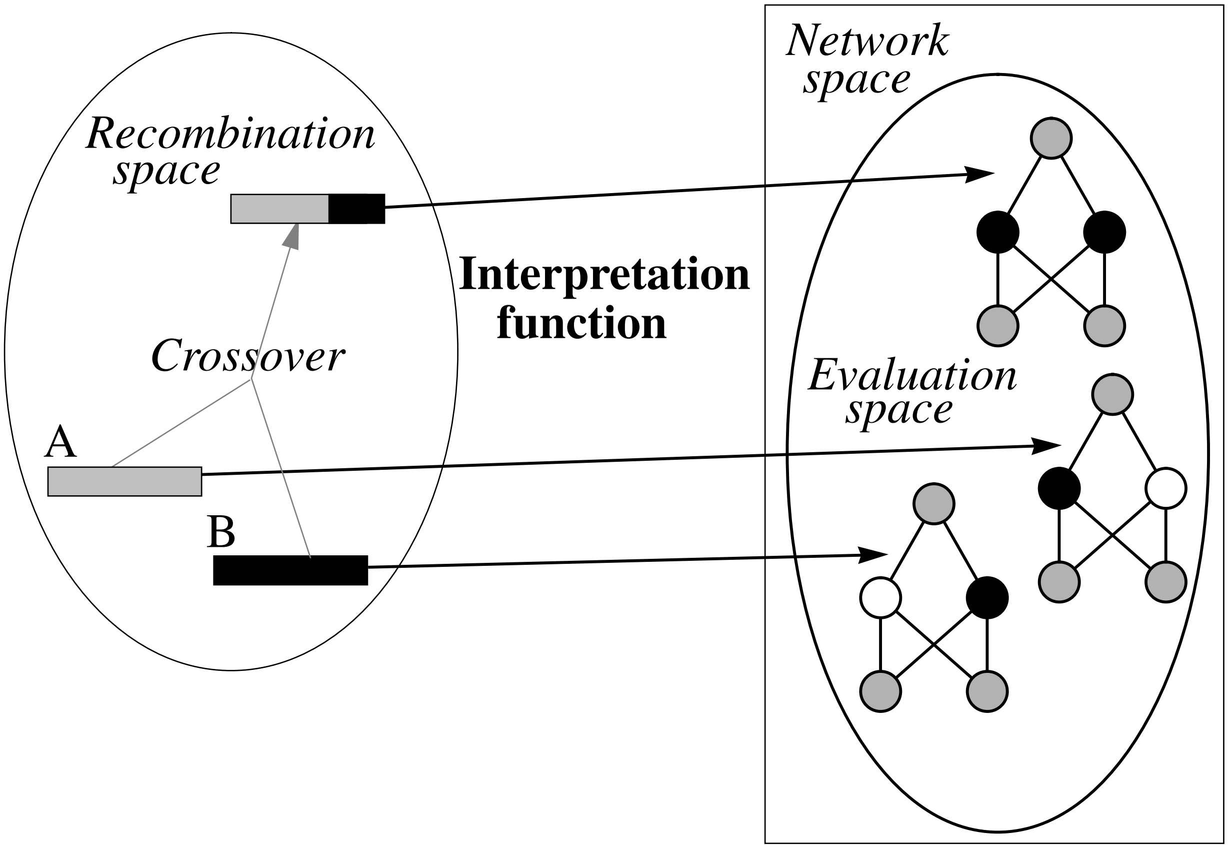


Figure 2. *The competing conventions problem [29]. Bit strings A and B map to structurally and computationally equivalent networks that assign the hidden units in different orders. Because the bit strings are distinct, crossover is likely to produce an offspring that contains multiple copies of the same hidden node, yielding a network with less computational ability than either parent.*

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 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 [16, 15]. 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, two such networks need not have the same bit string representation (see Figure 2). Crossover will then tend to create offspring that contain repeated components, and lose the computational ability of some of the parents' hidden units. The resulting networks will tend to 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 parents. This further reduces the likelihood that crossover will produce good offspring.

In short, for crossover to be a viable operator when 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 will 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 reproduction heuristics that respect the uniqueness of the distributed representations. This point has been tacitly validated in the genetic algorithm literature by a trend towards a reduced reliance on binary representations when evolving networks (e.g. [32, 33]). Crossover, however, is still commonplace.

2.2 Networks and Evolutionary Programming

Unlike genetic algorithms, evolutionary programming (EP) [14,34] defines representation-dependent mutation operators that create offspring within a specific locus of the parent (see Figure 3). EP's commitment to mutation as the sole reproductive operator for searching over a space is preferable when there is no sufficient calculus to guide recombination by crossover, 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.

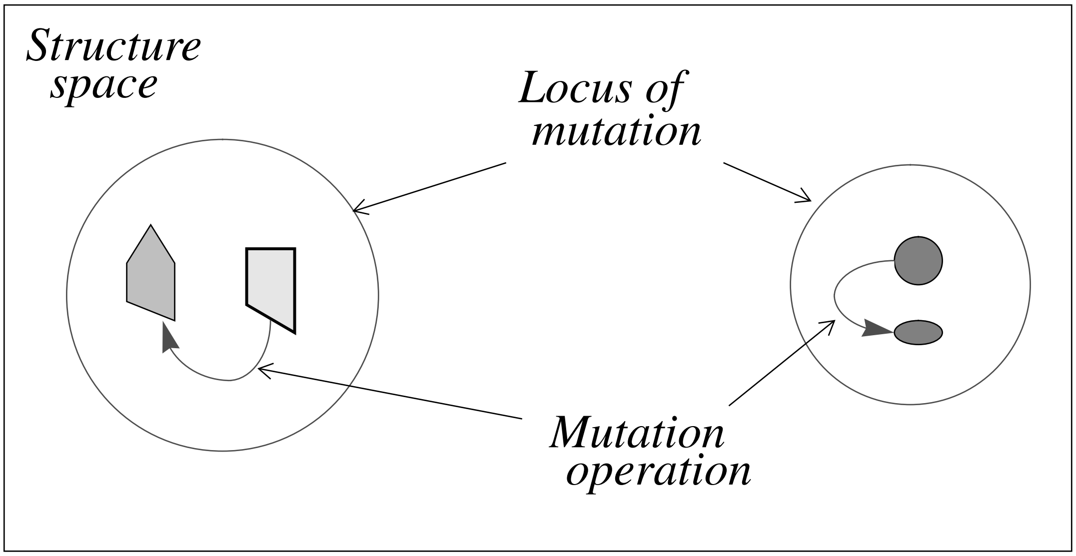


Figure 3. *The evolutionary programming approach to modeling evolution. Unlike genetic algorithms, evolutionary programs perform search in the space of networks. Offspring created by mutation remain within a locus of similarity to their parents.*

In each of the above studies, the mutation operator alters the parameters of network η by the function:

$$w = w + N(0, \alpha\varepsilon(\eta)) \qquad \forall w \in \eta \tag{EQ 1}$$

where w is a weight, $\varepsilon(\eta)$ is the error of the network on the task (typically the mean squared error), α is a user-defined proportionality constant, and $N(\mu, \sigma^2)$ is a gaussian variable with mean μ and variance σ^2. The implementations of structural mutations in these studies differ somewhat. McDonnell and Waagen [36] randomly select a set of weights and alters their values with a probability based on the variance of the incident nodes' activation over the training set; connections from nodes with a high variance having less of a chance of being altered. The structural mutation used in [14, 37] adds or deletes a single hidden unit with equal probability

Evolutionary programming provides distinct advantages over genetic algorithms when evolving networks. First, EP manipulates networks directly, thus obviating the need for a dual representation and the associated interpretation function. Second, by avoiding crossover between networks in creating offspring, the individuality of each network's distributed representation is respected. For these reasons, evolutionary programming provides a more appropriate framework for simultaneous structural and parametric learning in recurrent networks. The GNARL algorithm, presented in the next section and investigated in the remainder of this paper, describes one such approach.

3.0 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

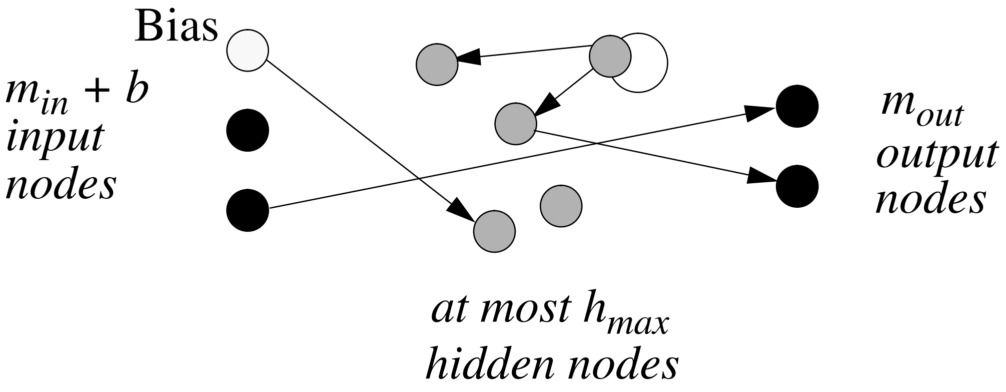


Figure 4. *Sample initial network. The number of input nodes (m_{in}) and number of output nodes (m_{out}) is fixed for a given task. The presence of a bias node (b = 0 or 1) as well as the maximum number of hidden units (h_{max}) is set by the user. The initial connectivity is chosen randomly (see text). The disconnected hidden node does not affect this particular network's computation, but is available as a resource for structural mutations.*

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 which 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_{in} input nodes and m_{out} output nodes. The number of hidden nodes varies from 0 to a user-supplied maximum h_{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 standard sigmoid activation function. Links use real-valued weights, and must obey three restrictions:

R_1: There can be no links *to* an input node.
R_2: There can be no links *from* an output node.
R_3: 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:

$\boldsymbol{S} = \{\eta$: η is a network with real-valued weights,
η satisfies R_1-R_3,
η has m_{in} + b input nodes, where b=1 if a bias node is provided, and 0 otherwise,
η has m_{out} output nodes,
η has i hidden nodes, $0 \le i \le h_{max}\}$

R_1-R_3 are strictly implementational constraints. Nothing in the algorithm described below hinges on $\boldsymbol{S}$ being pruned by these restrictions.

3.1 Selection, Reproduction and Mutation of Networks

GNARL initializes the population with randomly generated networks (see Figure 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: $\boldsymbol{S} \longrightarrow \boldsymbol{R}$, where $\boldsymbol{R}$ 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.

3.1.1 Severity of Mutations

The severity of a mutation to a given parent, η, is dictated by that network's temperature, $T(\eta)$:

$$T(\eta) = 1 - \frac{f(\eta)}{f_{\max}} \qquad \text{(EQ 2)}$$

where f_{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.

3.1.2 Parametric Mutation of Networks

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

$$w = w + N(0, \alpha T(\eta)) \qquad \forall w \in \eta \qquad \text{(EQ 3)}$$

where α 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 avoid 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 equation 3. First, the *instantaneous temperature* $\hat{T}$ of the network is computed:

$$\hat{T}(\eta) = U(0, 1)\, T(\eta) \qquad \text{(EQ 4)}$$

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

$$w = w + N(0, \alpha\hat{T}(\eta)) \qquad \forall w \in \eta \qquad \text{(EQ 5)}$$

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

3.1.3 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 R_1-R_3 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 in that a parameter identifies the likelihood that the link will originate from an input node or terminate at an output node. Once the class of incident node is determined, an actual node is chosen uniformly from the class. Biasing 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 is to become trapped at a structural local minima, 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:

$$\Delta_{\min} + \left\lfloor U[0,1]\, \hat{T}(\eta)\, (\Delta_{\max} - \Delta_{\min}) \right\rfloor \quad \text{(EQ 6)}$$

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 three while the maximum number of links added or deleted was five. The minimum number for each interval was always one.

3.2 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 $\{(x_1, y_1), (x_2, y_2), ...\}$, three possible measures of fitness for a network η are sum of square errors (equation 7), sum of absolute errors (equation 8), and sum of exponential absolute errors (equation 9):

$$\sum_i (y_i - Out(\eta, x_i))^2 \quad \text{(EQ 7)}$$

$$\sum_i |y_i - Out(\eta, x_i)| \quad \text{(EQ 8)}$$

$$\sum_i e^{|y_i - Out(\eta, x_i)|} \quad \text{(EQ 9)}$$

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.

4.0 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.

4.1 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 Figure 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;

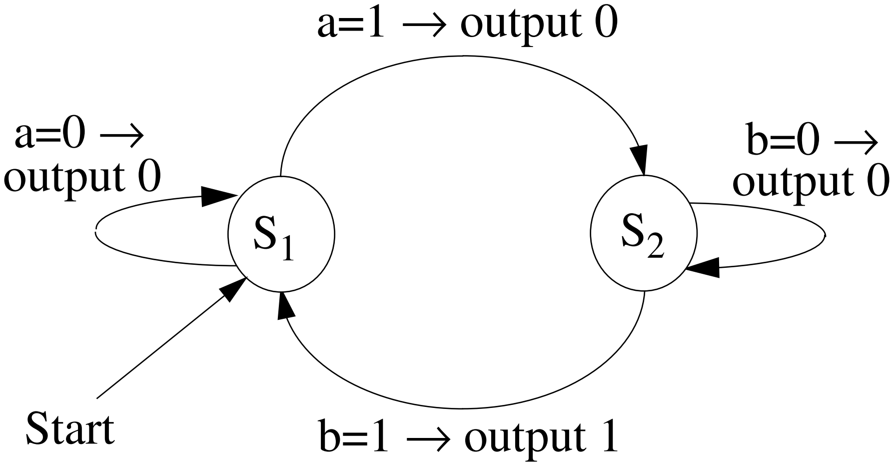


Figure 5. *An FSA that defines the enable-trigger task [39]. The system is given a data stream of bit pairs* $\{(a_1, b_1), (a_2, b_2), \ldots\}$*, and produces an output of 0's and 1's. To capture this system's input/output behavior, a connectionist network must learn to store state indefinitely.*

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 (equation 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 Table1. After 118 generations (3000 network evaluations[2]), GNARL evolved a network which solved this task for the strings in Table 1 within 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 (~20000 network evaluations total) GNARL once again found a network to solve this task, shown in Figure 6b. Note that there are two completely isolated nodes. Given the fitness function used in this experiment, the two isolated nodes do not effect 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

---

[2] Number of networks evaluated = |population| + generations * |population| * 50% of the population removed each generation, giving 50 + 118 * 50 * 0.5 = 3000 network evaluations for this trial.

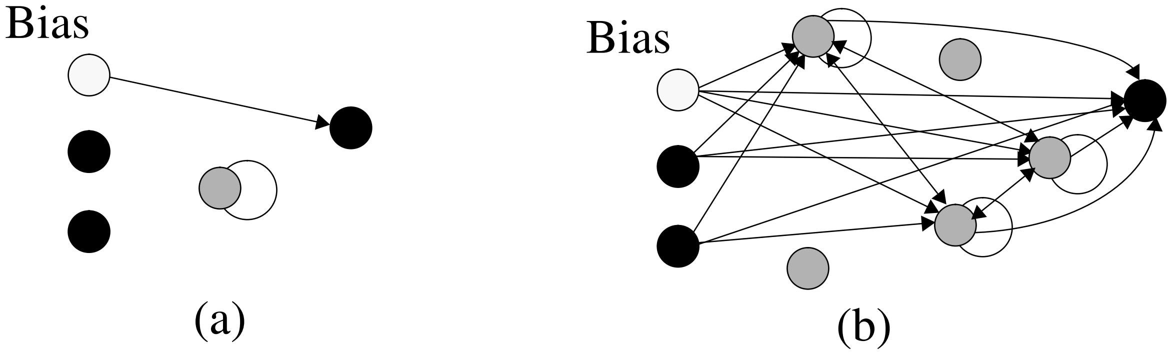


__Figure 6.__ Connectivity of two recurrent networks found in the enable-trigger experiment. (a) The best network of generation 1. (b) The best network of generation 765. This network solves the task for all strings of length eight.

increases dramatically only to be overtaken by the relatively sparse architecture shown in Figure 6b on the final generation. Apparently, this more sparsely connected network evolved more quickly than the more 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.

4.2 Inducing Regular Languages

A current topic of research in the connectionist community is the induction of finite state automata (FSAs) by networks with second-order recurrent connections. For instance, Pollack [40] trains sequential cascaded networks (SCNs) over a test set of languages, provided in [41] and

Input	Target Output
{(0, 0), (0, 0)}	{0, 0}
{(0, 0), (0, 1)}	{0, 0}
{(0, 0), (1, 0)}	{0, 0}
{(0, 0), (1, 1)}	{0, 0}
{(0, 1), (0, 0)}	{0, 0}
{(0, 1), (0, 1)}	{0, 0}
{(0, 1), (1, 0)}	{0, 0}
{(0, 1), (1, 1)}	{0, 0}

Input	Target Output
{(1, 0), (0, 0)}	{0, 0}
{(1, 0), (0, 1)}	{0, 1}
{(1, 0), (1, 0)}	{0, 0}
{(1, 0), (1, 1)}	{0, 1}
{(1, 1), (0, 0)}	{0, 0}
{(1, 1), (0, 1)}	{0, 1}
{(1, 1), (1, 0)}	{0, 0}
{(1, 1), (1, 1)}	{0, 1}

__Table 1.__ Initial training data for enable-trigger task.

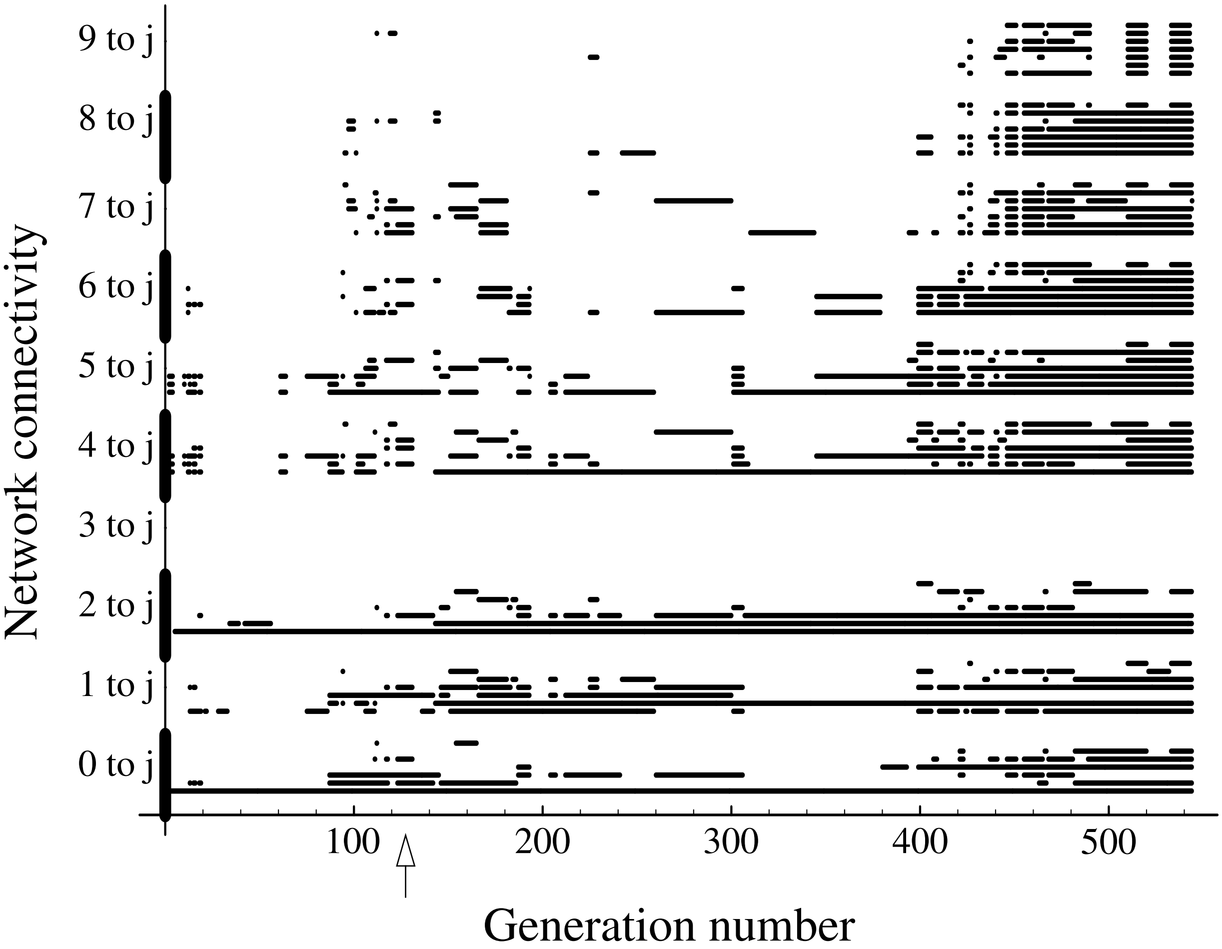


Figure 7. *Different network topologies explored by GNARL during the first 540 generations on the enable-trigger problem. The presence of a link between node i and j at generation g is indicated by a dot at position (g, 10 * i + j) in the graph. Note that because node 3 is the output node, there are no connections from it throughout the run. The arrow designates the point of transition between the first two training sets.*

shown in Table 2, 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 FSAs from the trained architectures [42 - 45].

An explicit collection of positive and negative examples, shown in Table 3, 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 2 require recurrent network connections in order to induce the language completely. The type of recurrence needed for each language varies widely. 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 indi-

Language	***Description***
1	1^*
2	$(1\ 0)^*$
3	no odd length 0 strings anytime after an odd length 1 string
4	no more than two 0s in a row
5	an even sum of 10s and 01s, pairwise
6	(number of 1s - number of 0s) mod 3 = 0
7	$0^*1^*0^*1^*$

Table 2. *Regular languages to be induced.*

vidual inputs, potentially changing state from accept to reject or vice versa on each successive input.

The results obtained in [42] are summarized in Table 4. 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

Language	***Positive Instances***	***Negative Instances***
1	ε, 1, 11, 111, 1111, 11111, 111111, 1111111, 11111111	0, 10, 01, 00, 011, 110, 000, 11111110, 10111111
2	ε, 10, 1010, 101010, 10101010, 10101010101010	1, 0, 11, 00, 01, 101, 100, 1001010, 10110, 110101010
3	ε, 1, 0, 01, 11, 00, 100, 110, 111, 000, 100100, 110000011100001, 111101100010011100	10, 101, 010, 1010, 110, 1011, 10001, 111010, 1001000, 11111000, 0111001101, 11011100110
4	ε, 1, 0, 10, 01, 00, 100100, 001111110100, 0100100100, 11100, 010	000, 11000, 0001, 000000000, 00000, 0000, 11111000011, 1101010000010111, 1010010001
5	ε, 11, 00, 001, 0101, 1010, 1000111101, 1001100001111010, 111111, 0000	1, 0, 111, 010, 000000000, 1000, 01, 10, 1110010100, 010111111110, 0001, 011
6	ε, 10, 01, 1100, 101010, 111, 000000, 0111101111, 100100100	1, 0, 11, 00, 101, 011, 11001, 1111, 00000000, 010111, 10111101111, 1001001001
7	ε, 1, 0, 10, 01, 11111, 000, 00110011, 0101, 0000100001111, 00100, 011111011111, 00	1010, 00110011000, 0101010101, 1011010, 10101, 010100, 101001, 100100110101

Table 3. *Training sets for the languages of Table 2 from [41].*

Language	*Average evaluations*	*Average % accuracy*	*Fewest evaluations*	*Best % accuracy*
1	3033.8	88.98	28	100.0
2	4522.6	91.18	807	100.0
3	12326.8	64.87	442	78.31
4	4393.2	42.50	60	60.92
5	1587.2	44.94	368	66.83
6	2137.6	23.19	306	46.21
7	2969.0	36.97	373	55.74

Table 4. *Speed and generalization results reported by [42] for learning the data sets of Table 3.*

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 output of the network after the entire string plus three trailing "null" symbols had been entered, one input per time step. The 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 5 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. Figure 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 Figure 9compares 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].

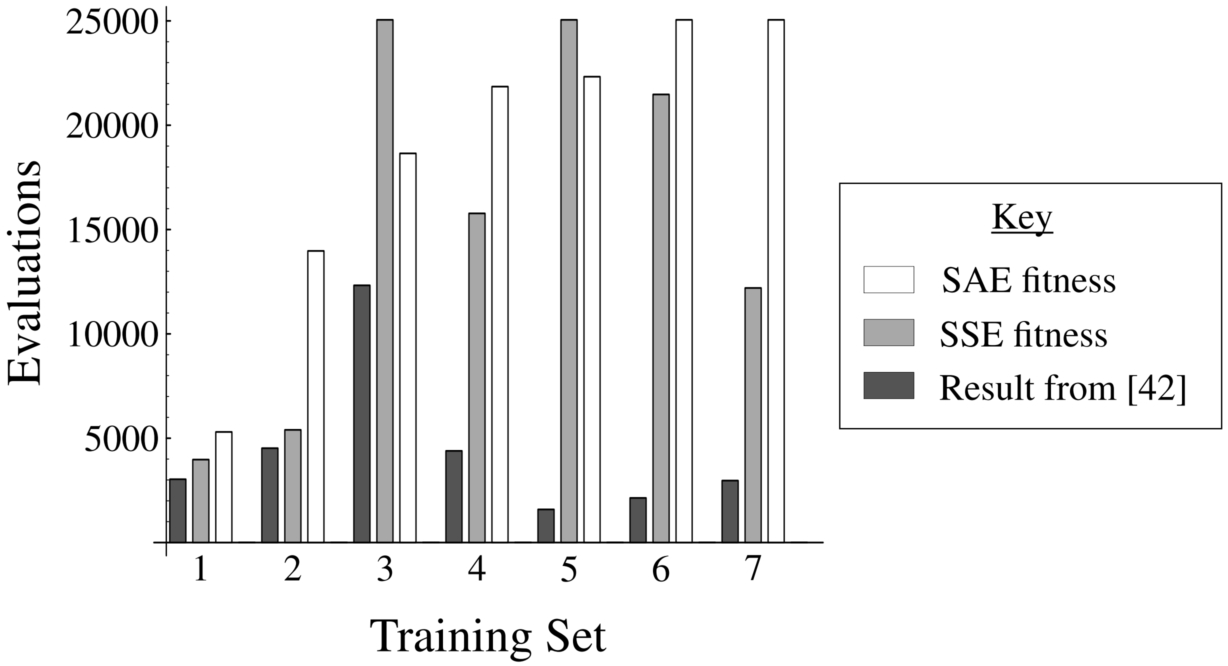


__Figure 8.__ The number of network evaluations required to learn the seven data sets of Table 3. GNARL (using both SAE and SSE fitness measures) compared to the average number of evaluations for the five runs described in [42].

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.

Language	***Evaluations (SAE)***	***% Accuracy (SAE)***	***Evaluations (SSE)***	***% Accuracy (SSE)***
1	3975	100.00	5300	99.27
2	5400	96.34	13975	73.33
3	25050^+	58.87	18650	68.00
4	15775	92.57^*	21850	57.15
5	25050^+	49.39	22325	51.25
6	21475	55.59^*	25050^+	44.11
7	12200	71.37^*	25050^+	31.46

__Table 5.__ Speed and generalization results for GNARL to train recurrent networks to recognize the data sets of Table 3.

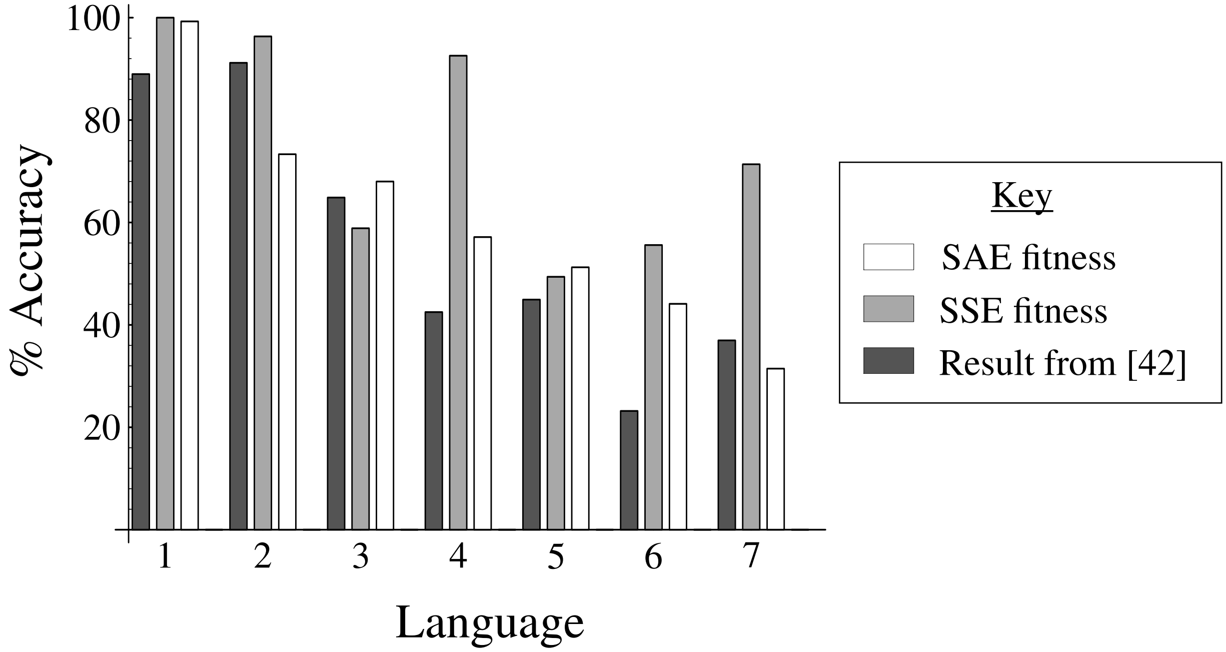


Figure 9. *Percentage accuracy of evolved networks on languages in Table 2. GNARL (using SAE and SSE fitness measures) compared to average accuracy of the five runs in [42].*

4.3 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 which collects the maximum number of pieces of food in a given time period. (Figure 10).

Following [46], each ant is controlled by a network with two input nodes and four output nodes (Figure 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 Figure 11 collects 42 pieces of food before spinning endlessly at position A (in Figure 10), 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 (Figure 12b) 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 Figure 13, hand-crafted in [46] as an approximate solution to the problem. This simple machine receives a score of 81 in the allotted

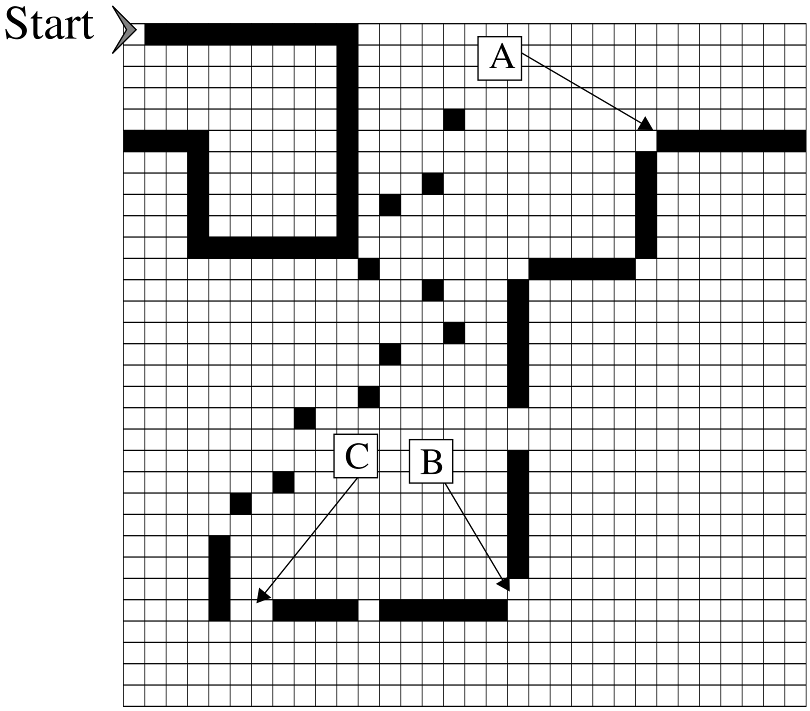


Figure 10. *The ant problem. The trail is connected initially, but becomes progressively more difficult to follow. The underlying 2-d grid is toroidal, so that position "A" is the first break in the trail – it is simple to reach this point. Positions "B" and "C" indicate the only two positions along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

200 time steps, and clears the entire trail only five time steps faster than the network in Figure 12b. 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 Figure 10. Here the evolved network makes a few additional moves, accounting for the slightly longer completion time.

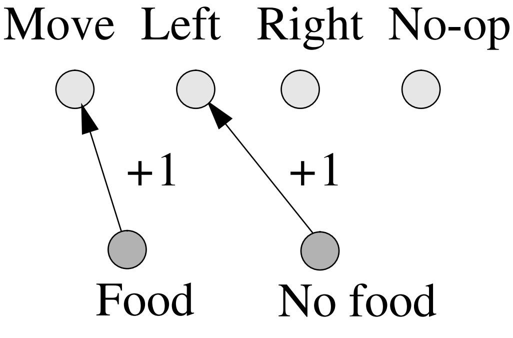
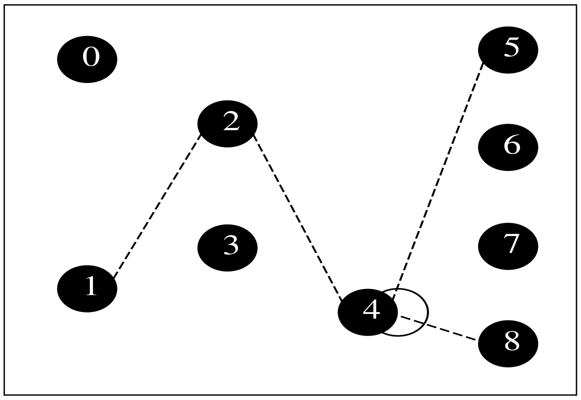
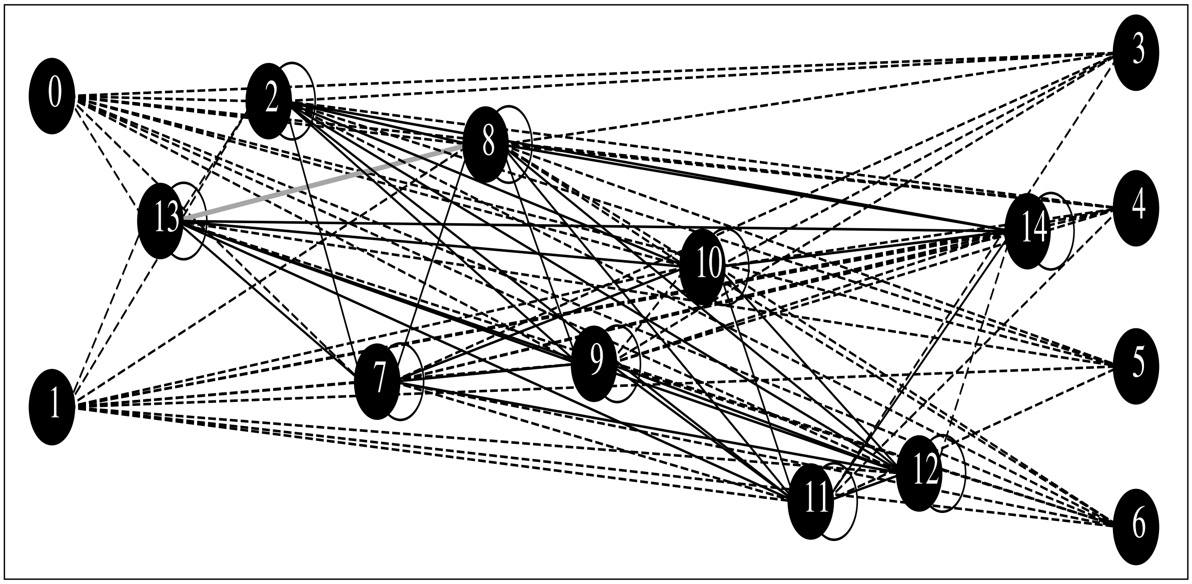


Figure 11. *The semantics of the I/O units for the ant network. The first input node denotes the presence of food in the square directly in front of the ant; the second denotes the absence of food in this same square. This particular network finds 42 pieces of food before spinning endlessly in place at position P, illustrating a very deep local minimum in the search space.*



(a)



(b)

Figure 12. *The Tracker Task, first run. (a) The best network in the initial population. Nodes 0 and 1 are input, nodes 5-8 are output, and nodes 2-4 are hidden nodes. (b) Network induced by GNARL after 2090 generations. Forward links are dashed; bidirectional links and loops are solid. The light gray connection between nodes 8 and 13 is the sole backlink. This network clears the trail in 319 epochs.*

Figure 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 14a shows the result of supplying to the network 200 “food” inputs – a fixed point that executes “Move.” Figure 14b 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 (Figure 14c, d); the additional points in Fig-

---

[3] No-op is not shown because it was never used in the final network.

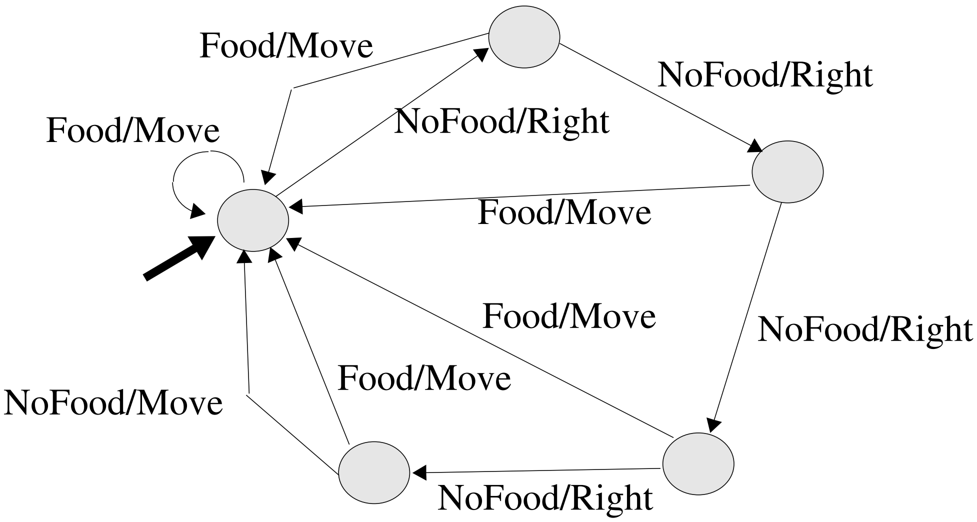


Figure 13. *FSA hand-crafted for the Tracker task in [46]. The large arrow indicates the initial state. This simple system implements the strategy "move forward if there is food in front of you, otherwise turn right four times, looking for food. If food is found while turning, pursue it, otherwise, move 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.*

ure 14c 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 Figure 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. Figure 15 demonstrates the behavior of this network. Once again, the "food" attractor, shown in Figure 15a, 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 (Figure 15b). The placement of the "D" is in the "Move / Right" corner of the space and encodes a complex alternation between these two operations (see Figure 15d).

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 1,123,942. 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.

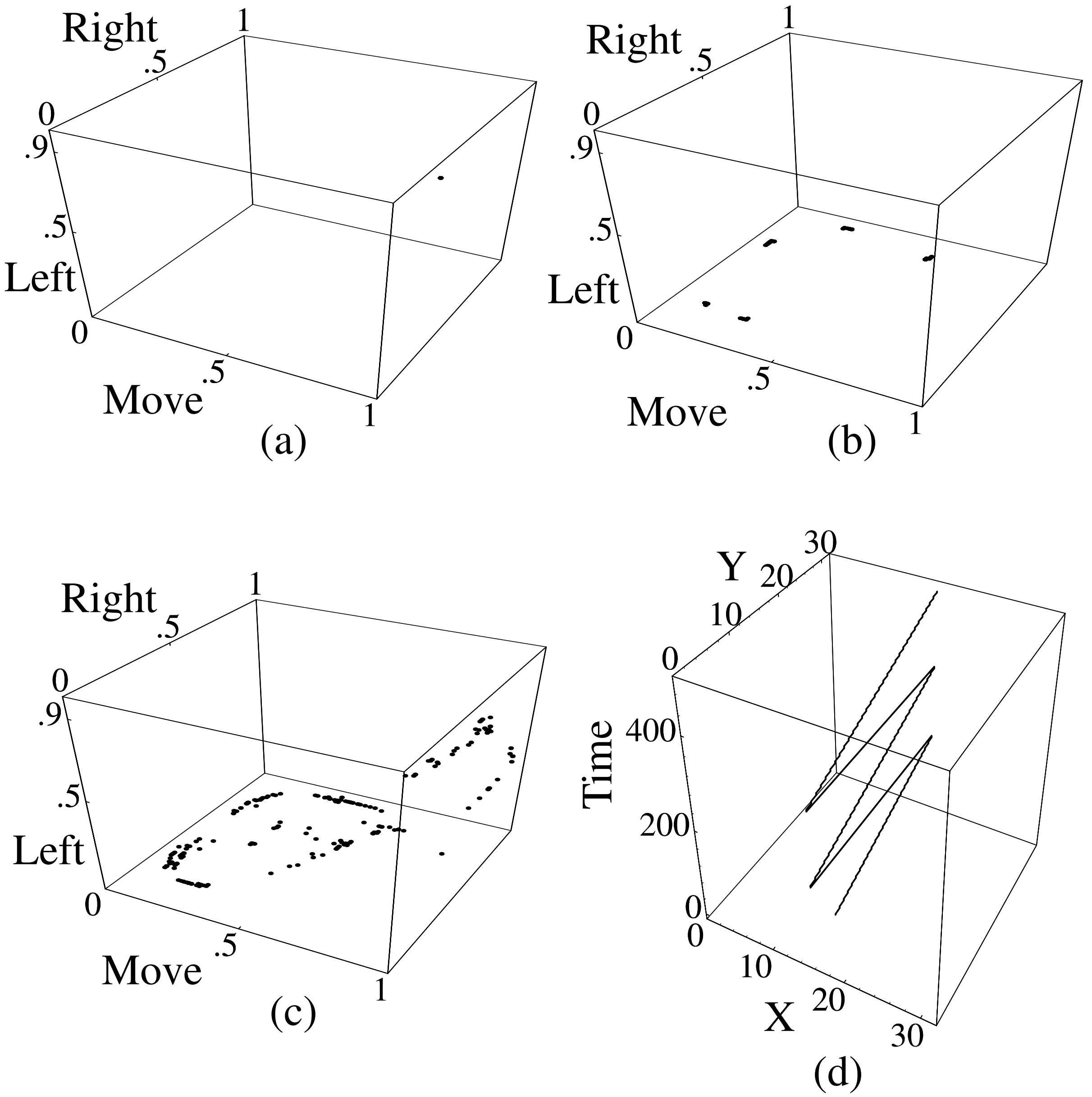
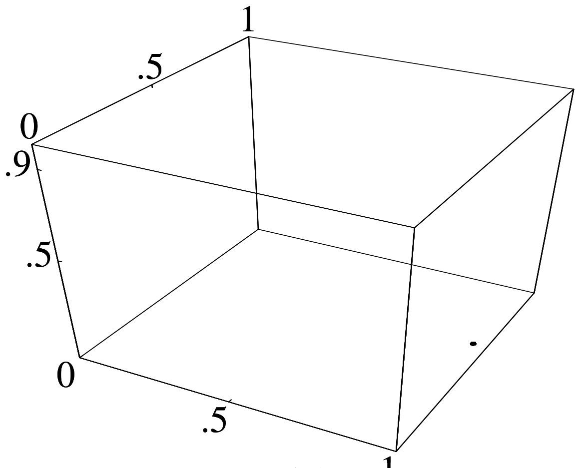


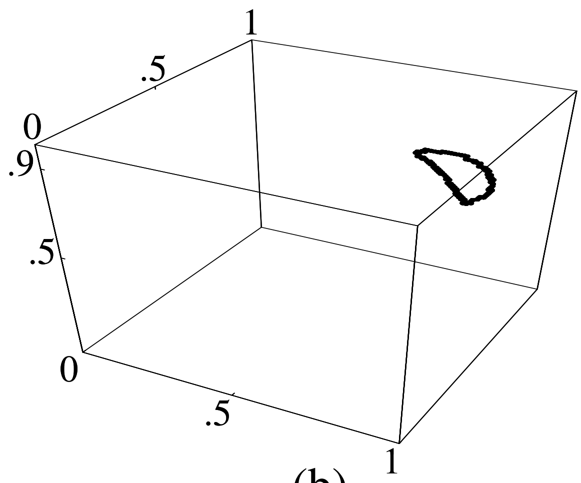
Figure 14. Limit behavior of the network that clears the trail in 319 steps. Graphs show the state of the output units Move, Right, Left. (a) Fixed point attractor that results for sequence of 500 "food" signals; (b) Limit cycle attractor that results when a sequence of 500 "no food" signals is given to network; (c) All states visited while traversing the trail; (d) The path of the ant on an empty grid. The Z axis represents time. Note that x is fixed, and y increases monotonically at a fixed rate. The large jumps in y position are artifacts of the toroidal grid.

5.0 Conclusions

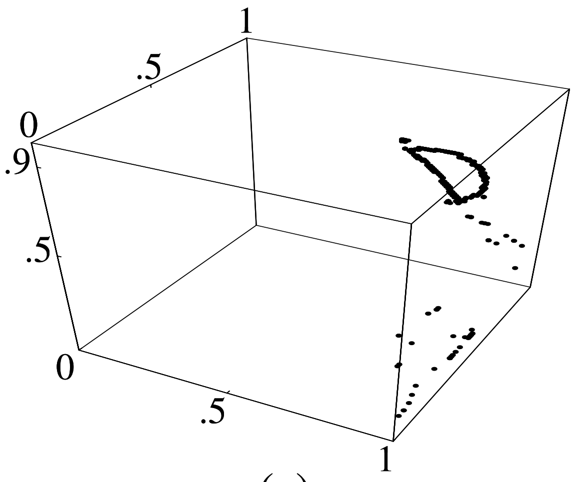
Allowing the task to specify an appropriate architecture for its solution should, in principle, be the defining aspect of the complete network induction problem. By restricting the space of networks explored, constructive, destructive, and genetic algorithms only partially address the problem of topology acquisition. GNARL's architectural constraints R_1-R_3 similarly reduce the search space, but to a far less degree. Furthermore, none of these constraints is necessary, and their



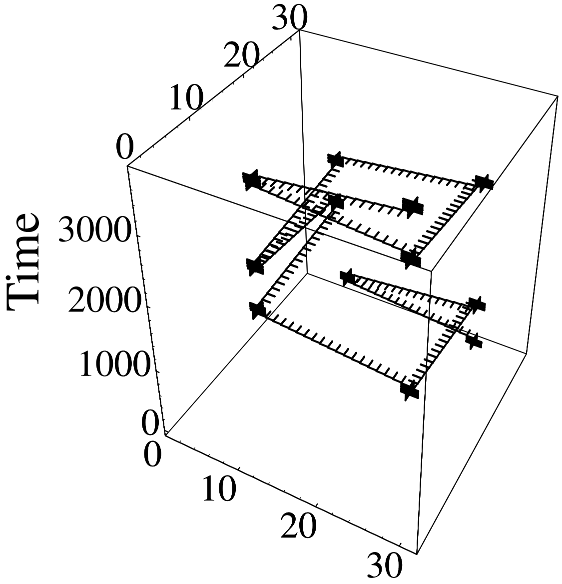
(a)



(b)



(c)



(d)

removal would affect only ease of implementation. In fact, no assumed features of GNARL's networks are essential for the algorithm's operation. GNARL could even use nondifferentiable activation functions, a constraint for backpropagation.

GNARL's minimal representational constraints would be meaningless if not complemented by appropriate search dynamics to traverse the space of networks. First, unlike constructive and destructive algorithms, GNARL permits a nonmonotonic search over the space of network topol-