CHAPTER IV

THE EMERGENCE OF TASK-SPECIFIC STRUCTURES

4.0 The Emergence of Task-Specific Structures

The induction of appropriate connectionist network architectures is a topic of current research. Most of the work by connectionists favor assuming an architecture for the network and simply learning the weights. Analytical solutions for specifying an appropriate network architecture have thus far eluded connectionist researchers.

The heuristics used in the induction of network architectures are usually overly simplistic in their determination of when and how to modify the architecture. This is because the implications of manipulating the network's architecture are not well understood. The properties of a network are sensitive to its architecture especially in recurrent networks. This level of sensitivity is not adequately expressed in a simple heuristic. As a result the associated heuristics are of poor quality and only narrowly applicable.

This chapter demonstrates that task-specific architectures for neural networks can emerge directly from interaction between an evolutionary weak method and the task being solved. An evolutionary program (Fogel, Owens and Walsh 1966; Fogel 1992) is the basis of the described algorithm. Operators specific to the manipulation of network architectures are employed. The chapter ends with several experiments that demonstrate that this method is comparable in speed to a connectionist method which only induces the weights of a network.

4.1 Connectionist Architectures

Models and simulations of networks of neurons have been investigated since the conception of the computer (McCulloch and Pitts 1943; Hebb 1949). Connectionism, the investigation of artificial networks of neurons, is a descendant of perceptron theory (Rosenblatt 1958; 1962), an early mathematical abstraction from networks of neurons. A

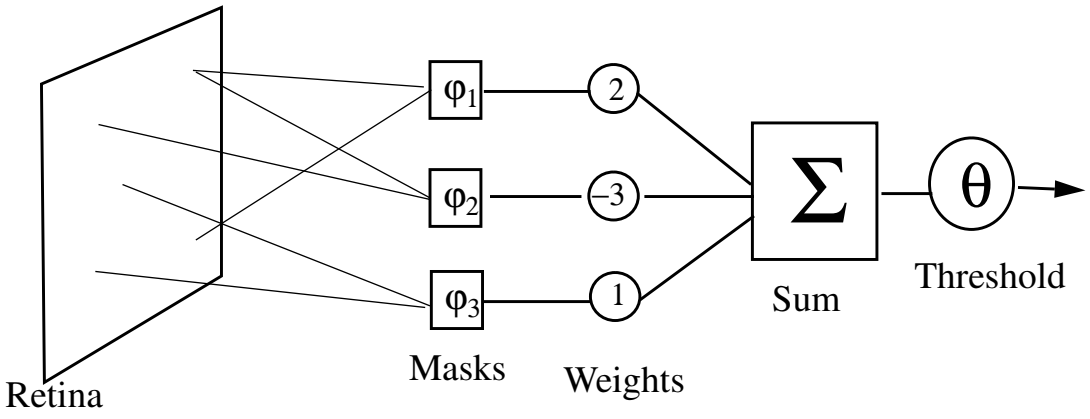


Figure 9: *Structure of a perceptron. The binary retina feeds into a set of masks that essentially multiply their inputs. These binary variables are then multiplied by the weights and summed. If the sum is larger than the stored threshold, then the concept represented by the preceptron appears in the retina.*

perceptron (Figure 9) consists of an input retina, a set of binary masks, weights on those masks, a summation device and a threshold device. A perceptron recognizes a feature of the input retina in the following manner. First, each mask takes as input a number of retinal positions and returns a one only if all of the inputs are a one. Next, each mask result is multiplied by its associated weight and summed. The feature represented by the perceptron is in the retina if this sum is greater than the given threshold θ. Rosenblatt (1958; 1962), following Hebb (1949), devised a method of training the weights in a perceptron, called the *perceptron learning rule*, to perform a particular task. Minsky and Papert (1969) showed that perceptrons were limited in the concepts they could identify. For instance, a perceptron can not perform the boolean operation exclusive-or.

Modern connectionism uses a variety of network architectures (e.g. Hopfield 1982; Ackley, Hinton and Sejnowski 1985; Rumelhart and McClelland 1986; McClelland and Rumelhart 1986a; Pollack 1990; Pollack 1991; and many others). All assume a collection of simple computational units with a set of weighted interconnections. As in perceptrons, the weights of the interconnections are modified through training to represent a concept. Modern training techniques for the weights on the interconnections takes as many forms as there are architectures: a form of simulated annealing is used in Boltzmann machines (Ackley, Hinton and Sejnowski 1985; Hinton and Sejnowski 1986); a generalization of the perceptron learning rule, called the *delta rule*, is used in standard PDP nets (Rumelhart,

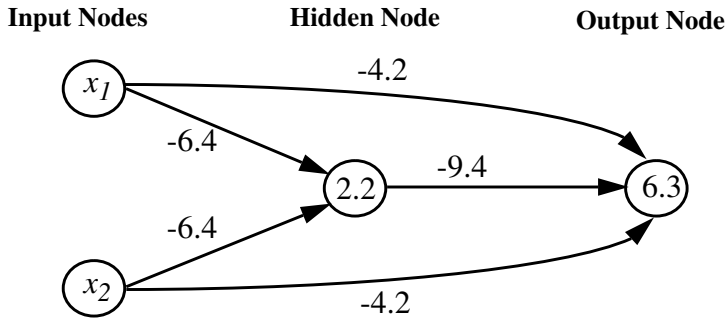


Figure 10: *Standard connectionist architecture. x_1 and x_2 are inputs to the network and are set by the environment. Their activation is multiplied by the weight on the associated connection. A hidden or output node sums all inputs, subtracts the threshold which is shown as a value inside the node, and applies a sigmoid activation function to determine output. The state of the output node is the The above network computes the exclusive-or of the inputs.*

Hinton and Williams 1986); Pollack (1990; 1991) uses a truncated version of the delta rule to train his *recursive auto-associative memories* (Pollack 1990) and *sequential cascaded networks* (Pollack 1991).

The typical connectionist architecture is shown in Figure 10. Input nodes receive information from the environment in the form of real numbers. The input nodes propagate their values along their weighted connections, also called links, to other nodes in the network. As in a perceptron, the output of a node is multiplied by the weight of the associated connection. All other nodes receive these signals as inputs and sum them together. Hidden and output nodes then apply an *activation function*, the standard activation function being the sigmoid function given by:

$$\text{activation} = \frac{1}{1 + e^{-(\sum_i x_i + \theta)}} \qquad \text{(EQ 9)}$$

where x_i is the ith input to the node and θ is the threshold associated with the node. Thus, each node of a modern connectionist architecture has a structure comparable to a single perceptron (see Figure 9). When the directed connections in the architecture form an acyclic graph, the architecture is called *feed-forward*. When a cycle is present, the architecture

is said to be *recurrent*. The rise in popularity of connectionist structures as models of cognition and computation coincides with the discovery of a simple training methods for discovering appropriate weights for feed-forward networks with fixed architectures.

4.2 The Complete Network Induction Problem

In its complete form, network induction entails both *parametric* and *structural* learning (Barto 1990), i.e., learning both weight values and an appropriate topology of nodes and links. Current connectionist methods that solve this task fall into two broad categories: constructive and destructive. *Constructive* algorithms initially assume a simple network and add nodes and links (Ash 1989; Frean 1990; Hanson 1990; Fahlman and Lebiere 1990; Fahlman 1991; Chen et al. 1993; Azimi-Sadjadi, Sheedvash and Trujillo 1993), while *destructive* methods start with a large network and prune off components (Mozer and Smolensky 1989; Le Cun, Denker and Solla 1990; Hassibi and Stork 1993; Omlin and Giles 1993). In all cases a simple heuristic, often based on statistics of the individual nodes, determines when to manipulate the architecture.

Though these algorithms address the problem of topology acquisition, they do so in a highly constrained manner. First, 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 which is typical of such "greedy" methods. In addition, constructive and destructive algorithms make simplifying architectural assumptions to facilitate network induction. For example, Ash (1989) allows only feedforward networks; Fahlman (1991) assumes a restricted form of recurrence, and Chen et al. (1993) explores 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 connectionist methods for the induction of architectures stem from inadequate models of assigning credit to the components of a network's architecture. 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 determining an appropriate architecture should come from the task rather than be hardcoded into the bias of the algorithm.

4.3 Evolving Connectionist Networks

As with any artificial intelligence program, selecting the correct technique is imperative to a successful solution. Given the three variations of the evolutionary weak method outlined in the last chapter, a determination needs to be made as to which one is most appropriate for solving the problem at hand. Because the interest here is the acquisition of both structure and function of the recurrent neural network, evolution strategies and their reliance on fixed-length real-valued parameter strings seems an obvious mismatch. The choice then is between a genetic algorithm or an evolutionary program.

Various combinations of genetic algorithms and connectionist networks have been investigated. Much of the research concentrates only on the acquisition of parameters for a fixed network architecture (e.g., Wieland 1990; Montana and Davis 1989; Whitley, Starkweather and Davis 1990; Beer and Gallagher 1992; Collins and Jefferson 1991). 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., Miller, Todd and Hegde 1989; Belew, McInerney and Schraudolf 1992; Polani and Uthmann 1993) or simply ignores the acquisition of parametric values altogether by assuming weights of 1 and -1 for each link (Gruau 1993; Zhang and Mühlenbein 1993). 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 severely restricted (e.g., Torreele 1991; Potter 1992; Karunanithi, Das and Whitley 1992; Romaniuk 1993). In spite of this collection of studies, current theory from both genetic algorithms and connectionism suggests GAs are not appropriate for evolving networks. In the following section, the reasons for this mismatch are explored.

4.3.1 Evolving Networks with Genetic Algorithms

As stated above, genetic algorithms create new individuals by recombining the representational components of two population members. Because of this commitment to struc-

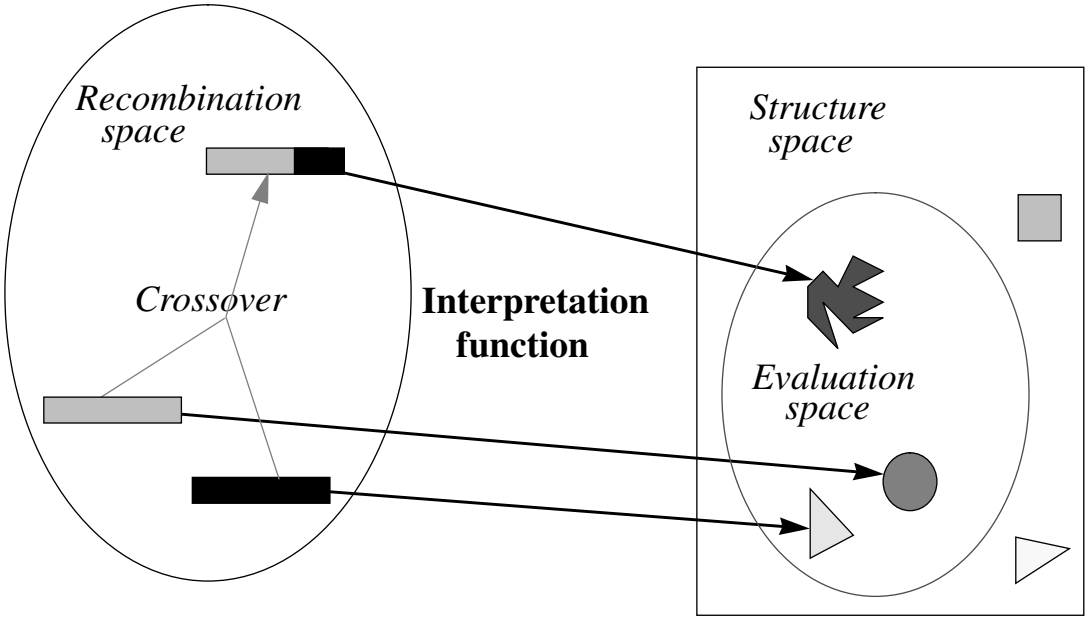


Figure 11: *The dual representation scheme used in genetic algorithms. The interpretation function maps between the elements in recombination space on which the search is performed and the subset of structures that can be evaluated as potential task solutions.*

tural recombination, genetic algorithms rely on two distinct representational spaces (Figure 11). *Recombination space*, typically 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 would be a set of networks. An *interpretation function* maps between these two representational spaces. Note that because any set of finite-length bit strings cannot represent all possible networks, the evaluation space is restricted to a predetermined set of networks. By design, this dual representation scheme allows the genetic algorithm to manipulate the bit string representations using crossover 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 genetic algorithm map to better and better networks.

The dual representation of genetic algorithms 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 a dual representation beneficial for evolving connectionist 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 dual representations is an unnecessary complication.

Characterizing tasks for which crossover is a beneficial operator is an open question. The building block hypothesis (Goldberg 1989a) suggests that crossover tends to recombine short, connected substrings, i.e. building blocks, of the bit string representation that correspond to generally good task solutions when evaluated. This hypothesis makes 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 (Goldberg 1989c). Environments where this is not true are called *deceptive* (Goldberg 1989b). However, there is some debate about the appropriateness of this concept (Grefenstette 1992).

There are three forms of deception when using crossover to evolve connectionist networks. The first involves identical networks, i.e., ones 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, as illustrated in Figure 12. In such a situation, crossover creates offspring that contain repeated components, thus losing the computational ability of some of the parents' hidden units. Consequently, the resulting networks perform worse than their parents since they are not in possession of key computational components for the task. Schaffer, Whitely and Eshelman (1992) terms this the *competing conventions* problem, and points out that the number of competing conventions grows exponentially with the number of hidden units.

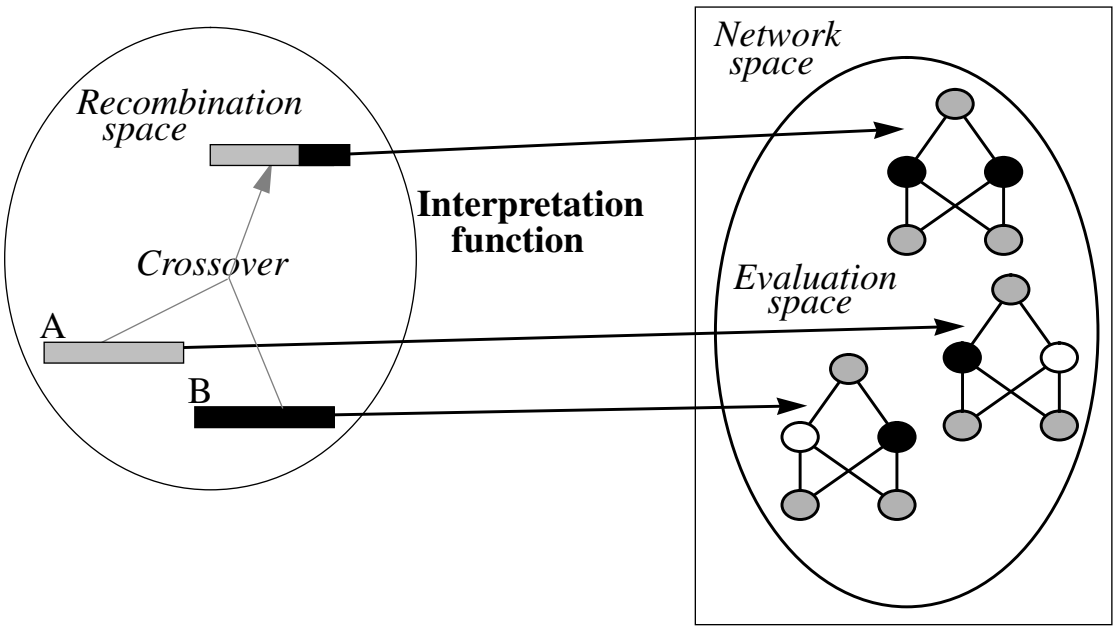


Figure 12: *The competing conventions problem (Schaffer, Whitley and Eshelman 1992). Bit strings A and B map to structurally and computationally equivalent networks that assign the hidden units in different orders. Because the bit strings are distinct, crossover is likely to produce an offspring that contains multiple copies of the same hidden node, yielding a network with less computational ability than either parent.*

A 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 (Hinton, McClelland and Rumelhart 1986; Sejnowski and Rosenberg 1987; Smolensky 1988). While the removal of a small number of nodes has been shown to effect only minor alterations in the performance of a trained network (Hinton, McClelland and Rumelhart 1986; Sejnowski and Rosenberg 1987; Smolensky 1988), the computational role each node plays in the overall representation of the 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 can construct viable offspring *regardless of the interpretation function used.*

Finally, when the architecture is variable, deception can occur when the parents differ topologically. The types of distributed representations that might develop in a network varies 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 three types of deceptiveness described above. This suggests that the complexity of an appropriate interpretation function should more than rival the complexity of the original learning problem. Thus, the prospect of evolving connectionist networks with crossover goes against current theory associated with both genetic algorithms and connectionist networks. Better results should be expected with reproduction operators that respect the uniqueness and potential incompatibility of the developing distributed representations.

4.3.2 Network Induction with Evolutionary Programming

Unlike genetic algorithms, evolutionary programming defines representation-dependent operators that create offspring within a specific behavioral locus of the parent (see Figure 13). Because EP models evolution at the level of the behavior of unmixing species, recombination of distinct population members is not considered. 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.

Some previous EP systems have addressed the problem of evolving connectionist networks. Fogel, Fogel and Porto (1990) investigates training feedforward networks on some classic connectionist problems. McDonnell and Waagen (1992) uses 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 (1992a) and Fogel (1993a) use 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. Saravanan (1992) uses an EP algorithm to evolve the weights of designed architectures.

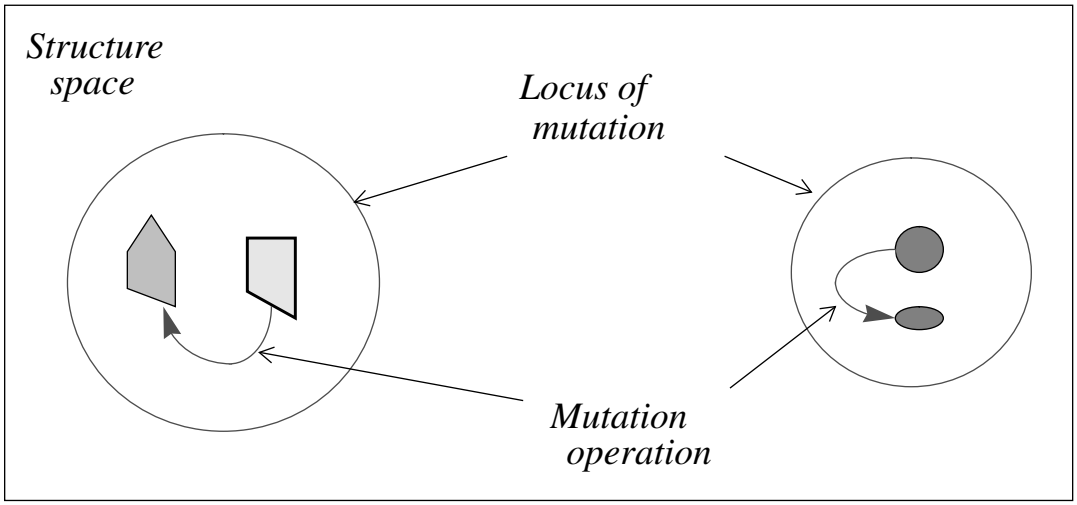


Figure 13: *The evolutionary programming approach to modeling evolution. Unlike genetic algorithm, evolutionary programs perform search in the space of networks. Offspring created by mutation remain within a locus of similarity to their parents.*

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

$$w = w + N(0, \alpha\varepsilon(\eta)) \qquad \forall w \in \eta \qquad \text{(EQ 10)}$$

where w is a weight, $\varepsilon(\eta)$ is the error of the network on the task, α is a user-defined proportionality constant, and $N(\mu, \sigma)$ is a gaussian variable with mean μ and standard deviation σ.

The implementations of structural mutations in the studies that make structural mutations differ somewhat. McDonnell and Waagen (1992) randomly selects 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 Fogel (1992a; 1992b) adds or deletes a single hidden unit with equal probability. As in connectionist and some genetic algorithm approaches to inducing network architectures, both of these EP studies employ weak heuristics of modification that limit the accessibility of certain architectures.

Evolutionary programming provides distinct advantages over genetic algorithms when evolving networks. First, EP manipulates the network representation 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. But rather than supply simplistic heuristics to signify when to manipulate the architecture, the interaction with the task environment can fully specify an appropriate architecture. The GNARL algorithm, presented in the next section and investigated in the remainder of this chapter, describes one such approach.

4.4 The GNARL Algorithm

GNARL, which stands for *GeNeralized Acquisition of Recurrent Links*, is an evolutionary algorithm that non-monotonically constructs recurrent connectionist 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 which may accurately reflect the 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 nodes are optional; if provided in an experiment, they are 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 all networks, η, that have real-valued weights

with m_{in} input nodes, m_{out} output nodes, and at most h_{max} hidden units. The maximum number of links, $\lambda(i)$, in a GNARL network with i hidden units is given by:

$$\lambda(i) = m_{in}m_{out} + m_{in}i + i^2 + im_{out} \quad \text{(EQ 11)}$$

The number of distinct architectures in the search space not counting the infinite number of parametric variations, is given by:

$$\sum_{i=0}^{h_{max}} \sum_{j=0}^{\lambda(i)} \binom{\lambda(i)}{j} \quad \text{(EQ 12)}$$

$$\sum_{i=0}^{h_{max}} 2^{\lambda(i)} \quad \text{(EQ 13)}$$

Needless to say, this is quite large even for a small number of input, output and hidden units. Note that the link restrictions, R_1-R_3, are strictly implementational constraints. Nothing in the algorithm described below hinges on the search space being pruned by these restrictions. Nor does it severely restrict the available architectures since architectures that are excluded by these restrictions are subsumed by architectures within the search space.

4.4.1 Selection, Reproduction and Mutation of Networks

To begin a run, GNARL initializes the population with randomly generated networks; a sample network is shown in Figure 14. The number of hidden nodes for each network is chosen from a uniform distribution over a user-supplied range. Similarly, the number of initial links is chosen 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]. Note that 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

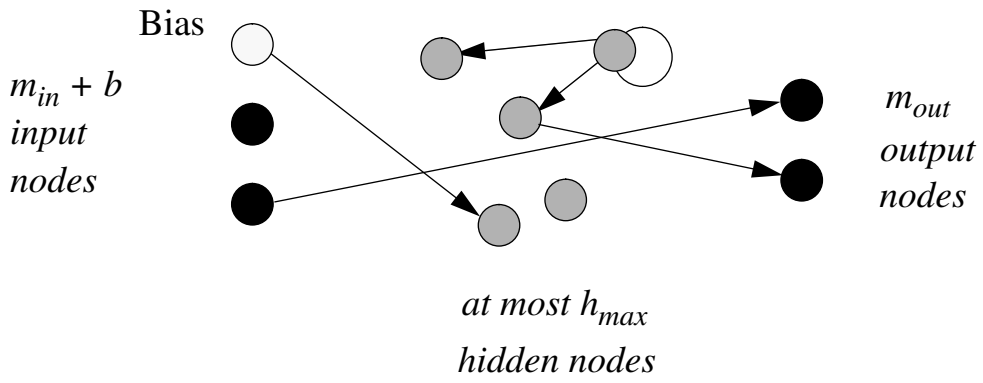


Figure 14: *Sample initial network. The number of input nodes (m_{in}) and number of output nodes (m_{out}) is fixed for a given task. The presence of a bias node (b = 0 or 1) as well as the maximum number of hidden units (h_{max}) is set by the user. The initial connectivity is chosen randomly (see text). The disconnected hidden node does not affect this particular network's computation, but is available as a resource for future structural mutations.*

experiments below, the number of hidden units for a network in the initial population was selected uniformly between 1 and 5 and the number of initial links varied uniformly between 1 and 10.

In each generation of search, the networks are first evaluated by a user-supplied fitness function $f: S \rightarrow \Re$. 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 (Fogel 1992a).

Reproduction of a network 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 network learning discussed in Barto (1990). *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.

4.4.1.1 Severity of Mutations

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

$$T(\eta) = 1 - \frac{f(\eta)}{f_{\max}} \qquad \text{(EQ 14)}$$

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 (cf. Kirkpatrick, Gelatt and Vecchi 1983). This allows a coarse-grained search initially, and a progressively finer-grained search as a network approaches a solution to the task, a process described more concretely below.

4.4.1.2 Parametric Mutation of Networks

Parametric mutations are accomplished by perturbing each weight w of a network η with gaussian noise, a method motivated by Fogel (1992a; 1992b). In that body of work, weights are modified as follows:

$$w = w + N(0, \alpha T(\eta)) \qquad \forall w \in \eta \qquad \text{(EQ 15)}$$

where α is a user-defined proportionality constant, and $N(\mu, \sigma)$ 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 15. First, the *instantaneous temperature* $\hat{T}$ of the network is computed:

$$\hat{T}(\eta) = U(0, 1)\, T(\eta) \qquad \text{(EQ 16)}$$

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 15:

$$w = w + N(0, \alpha\hat{T}(\eta)) \qquad \forall w \in \eta \qquad \text{(EQ 17)}$$

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

4.4.1.3 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, and to preserve the assumptions of evolutionary programming, structural mutations attempt to preserve the behavior of a network. For instance, new links are initialized with zero weight, thus 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.

Fogel (1992a; 1992b) 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 non-evolutionary 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 separate instantaneous temperature 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 $[\Delta_{min}, \Delta_{max}]$ for a particular structural mutation, the number of modifications of this type made to an offspring is given by:

$$\Delta_{min} + \left\lfloor U[0, 1]\, \hat{T}(\eta)\, (\Delta_{max} - \Delta_{min}) \right\rfloor \quad \text{(EQ 18)}$$

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.

4.4.2 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*. However 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 19), sum of absolute errors (Equation 20), and sum of exponential errors (Equation 21):

$$\sum_i (y_i - Out(\eta, x_i))^2 \quad \text{(EQ 19)}$$

$$\sum_i |y_i - Out(\eta, x_i)| \quad \text{(EQ 20)}$$

$$\sum_i e^{y_i - Out(\eta, x_i)} \quad \text{(EQ 21)}$$

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 are demonstrated in the experiments below.

Table 2. *Characterizations of the seven languages from Tomita (1982).*

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	$0^*1^*0^*1^*$

4.5 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 below.

4.5.1 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 (1991) trains sequential cascaded networks (SCNs) over a test set of languages, provided in Tomita (1982) and shown inTable 2. 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 Tomita (1982) have investigated various network architectures and training methods, as well as algorithms for extracting FSAs from the trained architectures (e.g. Watrous and Kuhn 1992; Giles et al. 1990; Giles et al. 1992; Zeng, Goodman and Smyth 1993).

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

Tomita (1982) supplies an explicit collection of positive and negative examples, shown in Table 3, that pose specific difficulties for inducing the intended languages. 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 Watrous and Kuhn (1992).

Following Pollack (1991), Watrous and Kuhn (1992) explored the induction of the languages of Table 2 using the Tomita (1982) data sets in a second-order recurrent neural network similar to that used in Pollack (1991). Watrous and Kuhn (1992) ran five experiments for each language and reported the number of training set presentations for each. Watrous and Kuhn (1992) measure the accuracy of a network as the percentage of strings of length 10 or less that are correctly classified by the induced network.

The results obtained in Watrous and Kuhn (1992) are summarized in Table 4. The table shows both the number of networks evaluated to learn the training set and the accuracy of generalization for each regular language of Table 2. For comparison, Table 4 lists both the average and best performance of the five runs reported by Watrous and Kuhn (1992).

Table 3. *Positive and negative examples for the seven languages investigated in Tomita (1982).* ε *signifies the null string.*

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 4. *Results from Watrous and Kuhn (1992) for the seven languages shown in Table 3.*

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

This experiment with GNARL 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 fitness (SAE) and the other using sum of square errors fitness (SSE). The error for a particular string was computed only for the final output of the network after the entire string plus three trailing *blank* symbols had been entered into the network. The concatenation of the trailing blanks was used to identify the end of the string and allow input of the null string, a method also used in Watrous and Kuhn (1992). Each network had a single input and output unit and no bias node was provided. As in Watrous and Kuhn (1992) the three possible logical inputs for this task, 0, 1, and *blank*, were represented by activations of -1, 1, and 0, respectively. The tolerance for the output value was 0.1.

Table 5 shows the number of evaluations until convergence and the accuracy of the best evolved network for both fitness functions. 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

Table 5. *Experimental results for GNARL on the seven languages shown in Table 3. Units are identical for the same measures in Table 4.*

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

uncompleted runs using SSE successfully separated the training sets but had not done so to the 0.1 tolerance within the 1000 generation limit. Figure 15 compares the number of evaluations by GNARL to the average number of evaluations reported in Watrous and Kuhn (1992) and shown in Table 4. As the graph shows, GNARL consistently evaluates more networks, but not a disproportionate number. Considering that GNARL is manipulating both the architecture and weights while Watrous and Kuhn (1992) are training an assumed second-order architecture, these numbers are well within a tolerable increase.

The graph of Figure 16 compares the accuracy of the GNARL networks to the *average* accuracy found in Watrous and Kuhn (1992) over five runs. As shown in the graph, the networks induced by GNARL consistently exceed the average accuracy found by Watrous and Kuhn (1992) listed in Table 4.

These results demonstrate not only GNARL's ability to simultaneously acquire the topology and weights of recurrent networks, but also that it can be done within a comparable number of network evaluations as training a network with static architecture on the same task. Furthermore, GNARL appears to generalize consistently better, possibly due to its selective inclusion and exclusion of some links.

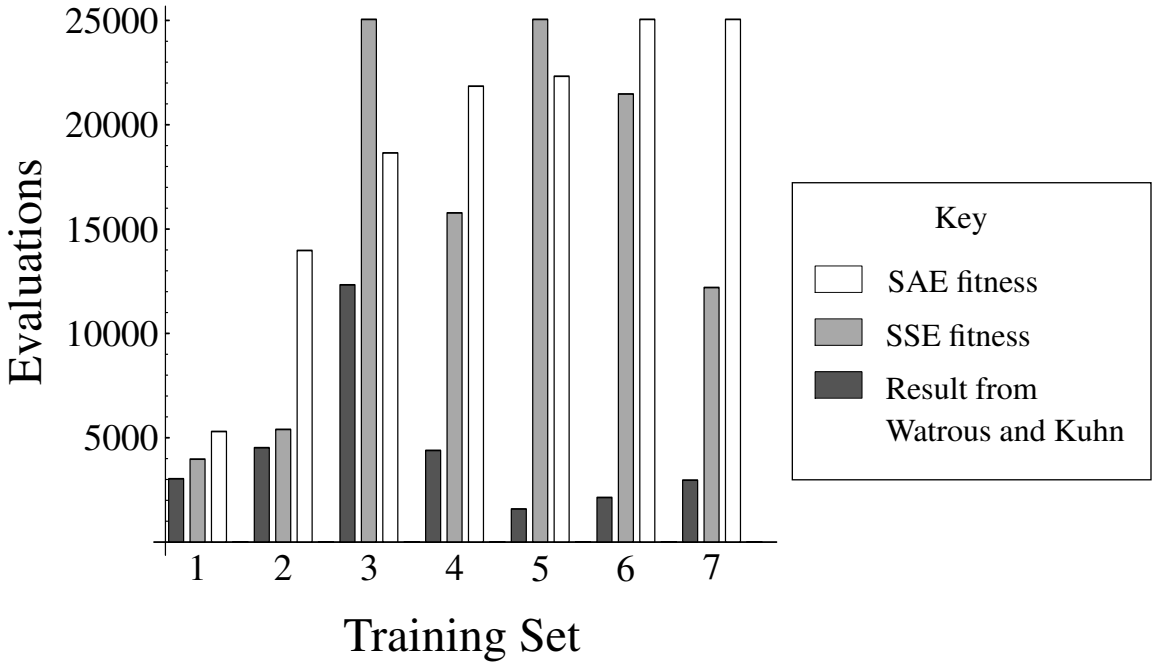


Figure 15: *The number of network evaluations required to learn the seven data sets of Table 3. GNARL (using both SAE and SSE fitness measures) compared to the average number of evaluations for the five runs described in Watrous and Kuhn (1992).*

4.5.2 The Ant Problem

GNARL was tested on a complex search and collection task – the *Tracker* task described by Jefferson et al. (1992). In this problem, a simulated ant is placed on a two-dimensional toroidal grid that contains a trail of food, shown in Figure 17. The ant traverses the grid, collecting along the way any food it contacts. The goal of the task is to discover a controller for the ant that collects the maximum number of pieces of food in a given time period.

Following Jefferson et al. (1992), each ant is controlled by a connectionist network with two input nodes and four output nodes, an example of which is shown in Figure 18. The first input node, labeled FOOD in the figure, denotes the presence of food in the square directly in front of the ant. The second input node, labeled NO FOOD in the figure, denotes the absence of food in this same square. This restricts the possible legal inputs to the network to (1, 0) or (0, 1).

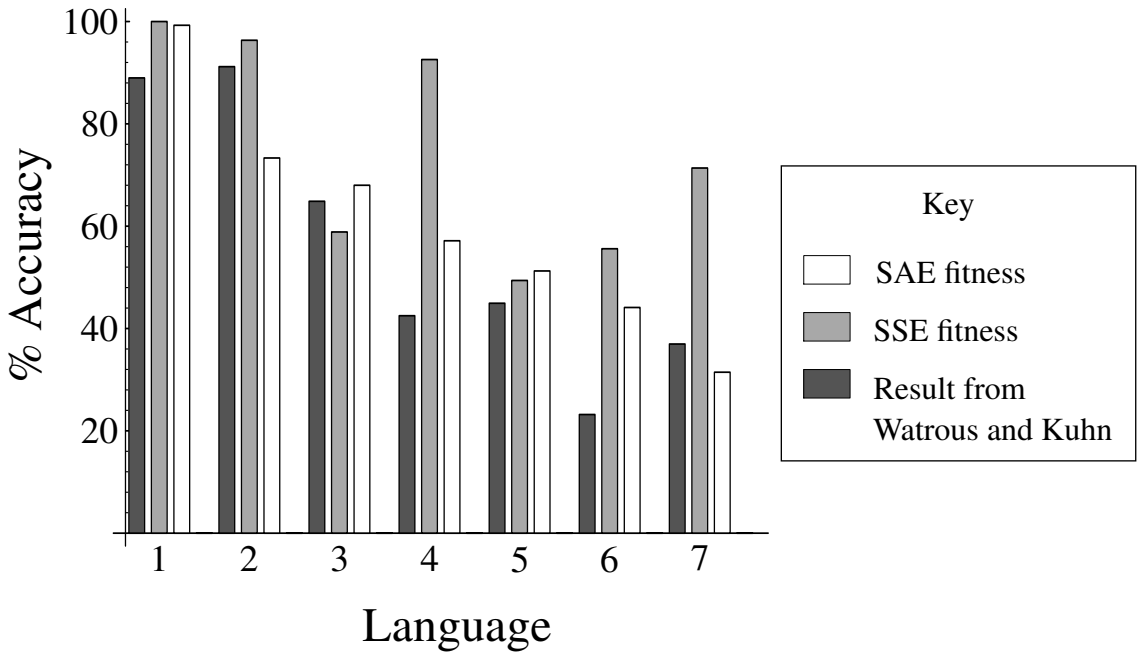


Figure 16: *Percentage accuracy of evolved networks on languages in Table 3. GNARL (using SAE and SSE fitness measures) compared to average accuracy of the five runs in Watrous and Kuhn (1992).*

Each of the four output units of the network, labeled as MOVE, LEFT, RIGHT and NO-OP in Figure 18, correspond to a unique action by the ant – move forward one step, turn left 90°, turn right 90°, and no-op respectively. At each step, the action whose corresponding output node has maximum activation is performed. As in Jefferson et al. (1992), NO-OP allows the ant to remain at a fixed position while activation flows along recurrent connections.

Fitness for an ant controller 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 18 collects 42 pieces of food before spinning endlessly at position A (in Figure 17), illustrating a very deep local minimum in the search space.

This experiment with GNARL used a population of 100 networks, each limited to at most 9 hidden units, and did not provide a bias node. In the first run (2090 generations), GNARL found a network (Figure 19b) that cleared 81 grid positions within the 200 time

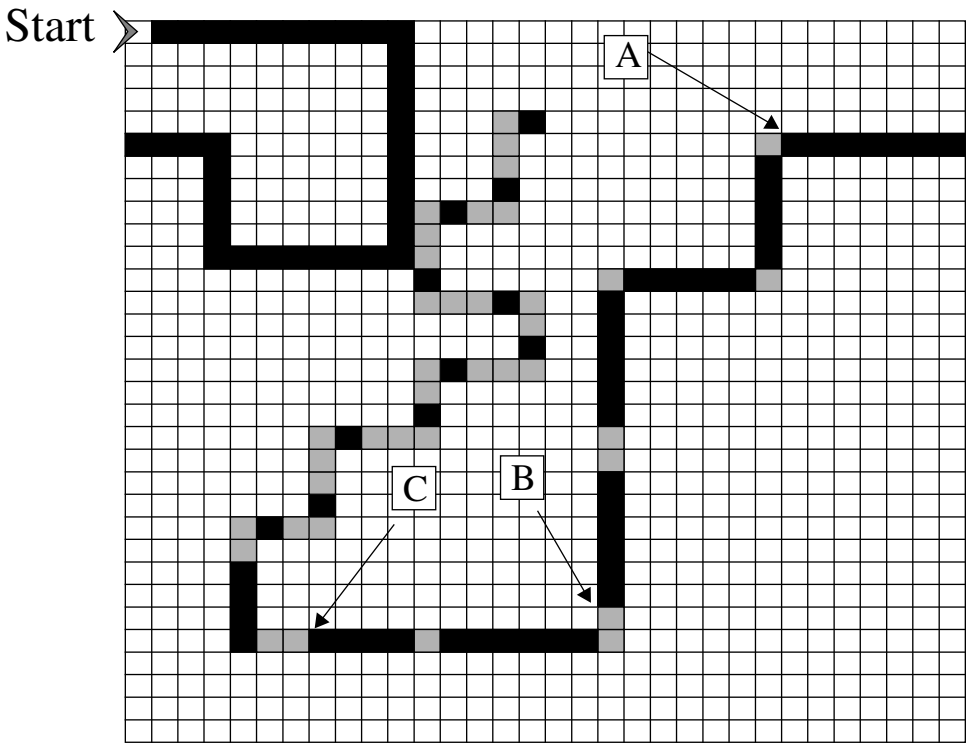


Figure 17: *The ant problem. Black squares indicate food while grey squares show the shortest path. The trail is connected initially, but becomes progressively more difficult to follow. The underlying 2-d grid is toroidal, so that position "A" is the first break in the trail – it is simple to reach this 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 Jefferson et al. (1992) (see Figure 20).*

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 20, hand-crafted by Jefferson et al. (1992) as an approximate solution to the problem. This simple machine also receives a score of 81 in the allotted 200 time steps, and clears the entire trail only 5 time steps faster than the network of Figure 19b. A step by step comparison showed there is only a slight difference between this FSA and the network of Figure 19b. GNARL's evolved network follows the same general strategy embodied by this FSA at all but two places, marked as positions B and C in Figure 17. Here the evolved network makes a few additional moves, accounting for the slightly longer completion time.

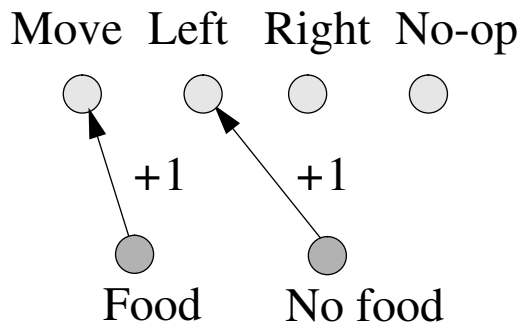
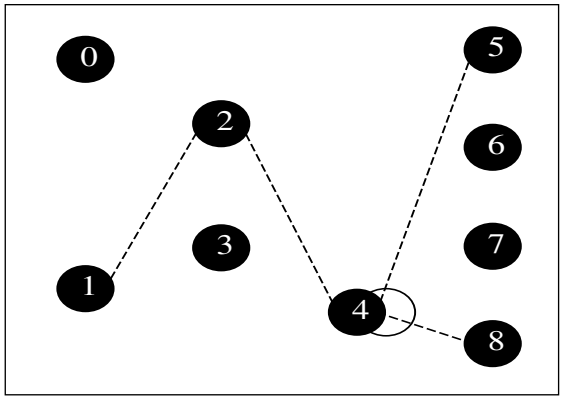


Figure 18: *The semantics of the I/O units for the ant network. The first input node denotes the presence of food in the square directly in front of the ant; the second denotes the absence of food in this same square. This particular network finds 42 pieces of food before spinning endlessly in place at position P, illustrating a very deep local minimum in the search space.*

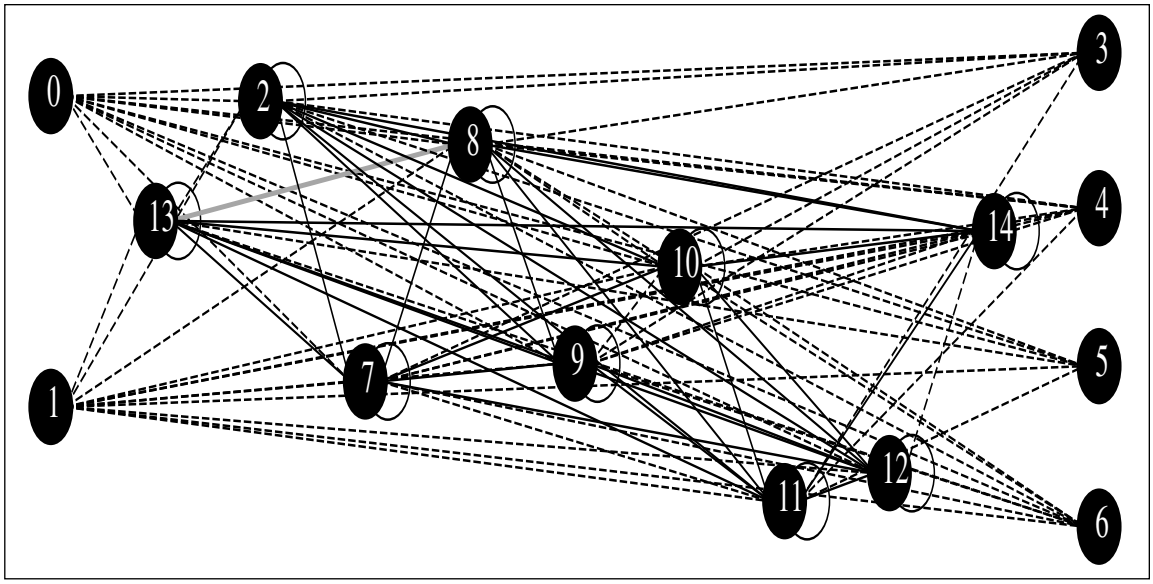
Figure 21 illustrates the strategy the network uses to implement the FSA by showing the state of the output units of the network over three different sets. Each point is a triple of the form (*move*, *right*, *left*).[1] Figure 21a shows that the result of supplying network 200 FOOD inputs to the network is a fixed point that executes MOVE. Figure 21b shows the sequence of states reached when 200 NO FOOD are given to the network. The result is a collection of points describing a limit cycle of length 5 that repeatedly executes the sequence "Right, Right, Right, Right, Move." These two attractors determine the response of the network to the task environment, shown in Figure 21c and d. The additional points in Figure 21c 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 20 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 of GNARL's evolved network behaviors are so simple as to approximate an FSA (cf. Pollack 1991). In a second run (1595 generations) GNARL induced a network that cleared 82 grid points within the 200 time steps. Figure 22 demonstrates the behavior of this network using the same graphs as Figure 21. Once again, the FOOD

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



(a)



(b)

Figure 19: *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.*

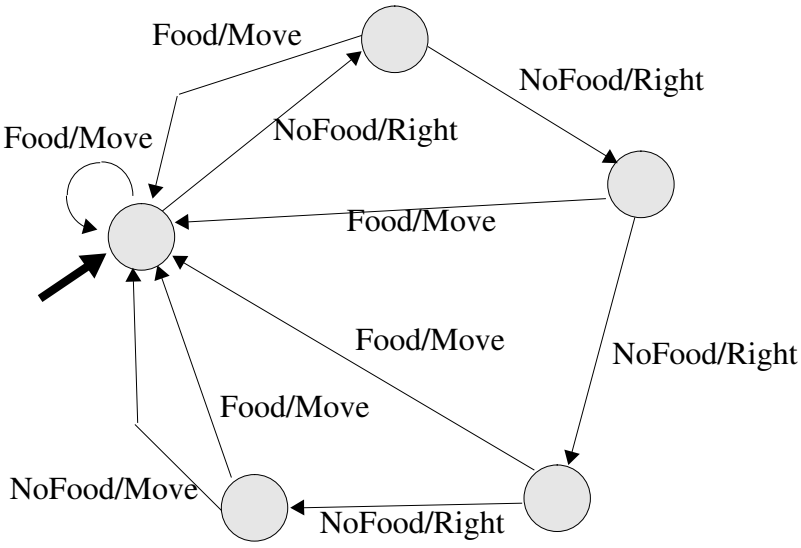


Figure 20: *FSA hand-crafted for the Tracker task by Jefferson et al. (1992). Food is indicated by 1, no food is indicated by 0. This simple system implements the strategy "move forward if there is food in front of you, otherwise turn right four times, looking for food. If food is found while turning, pursue it, otherwise, move forward one step and repeat." This simple FSA traverses the entire trail in 314 steps, and gets a score of 81 in the allotted 200 time steps.*

attractor, shown in Figure 22a, 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 22b). 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 22d).

In contrast to this work, Jefferson et al. (1992) 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 5 hidden units to solve this task. The particular network architecture in Jefferson et al. (1992) 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 Jefferson et al. (1992) discovered a comparable network after about 17 generations. Given Jefferson et al. (1992) used a population size of 65,536 and replaced 95% of the population each generation, the total number of network

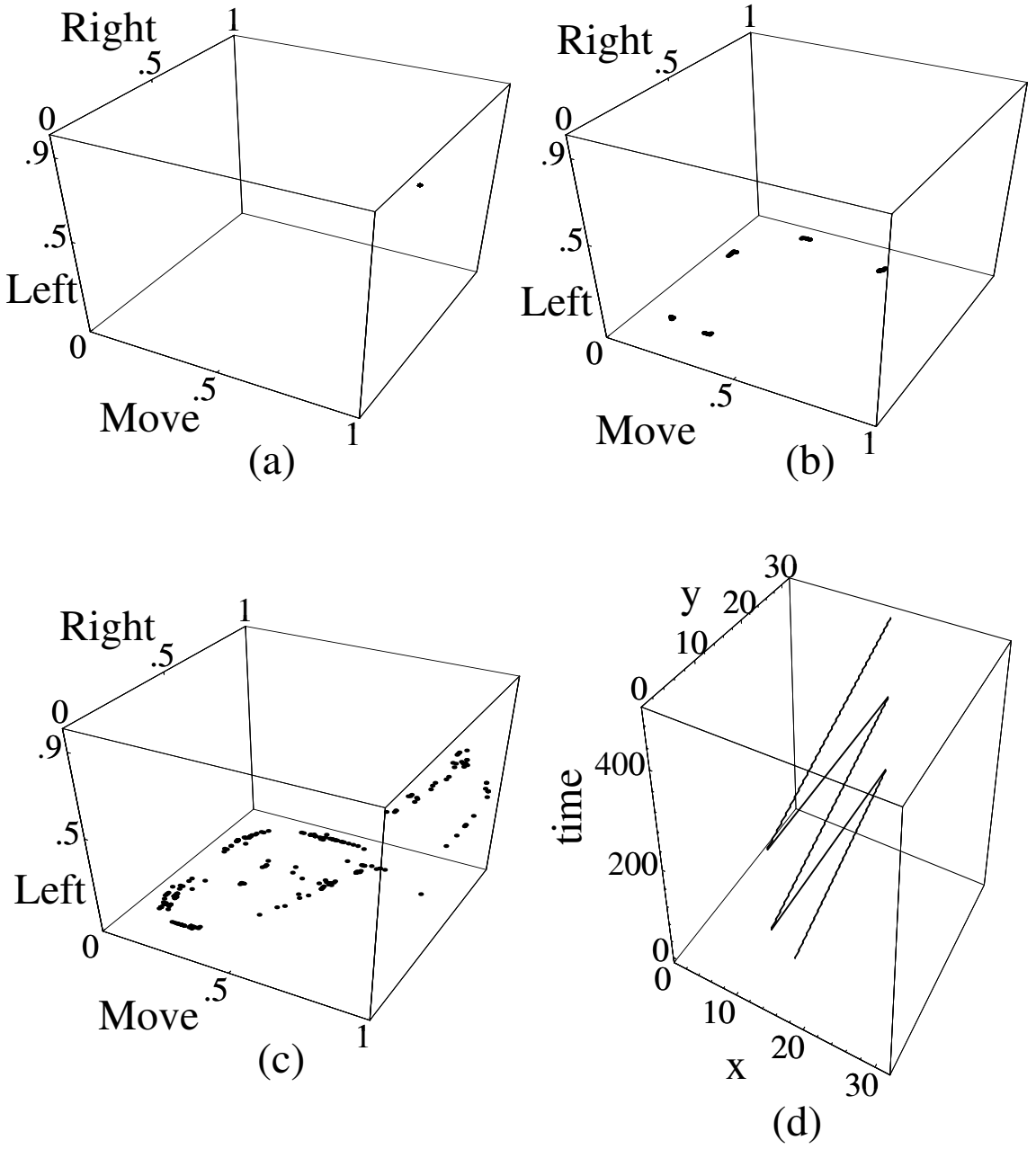


Figure 21: *Limit behavior of the network that clears the trail in 319 steps. Graphs show the state of the output units Move, Right, Left. (a) Fixed point attractor that results for sequence of 500 "food" signals; (b) Limit cycle attractor that results when a sequence of 500 "no food" signals is given 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.*

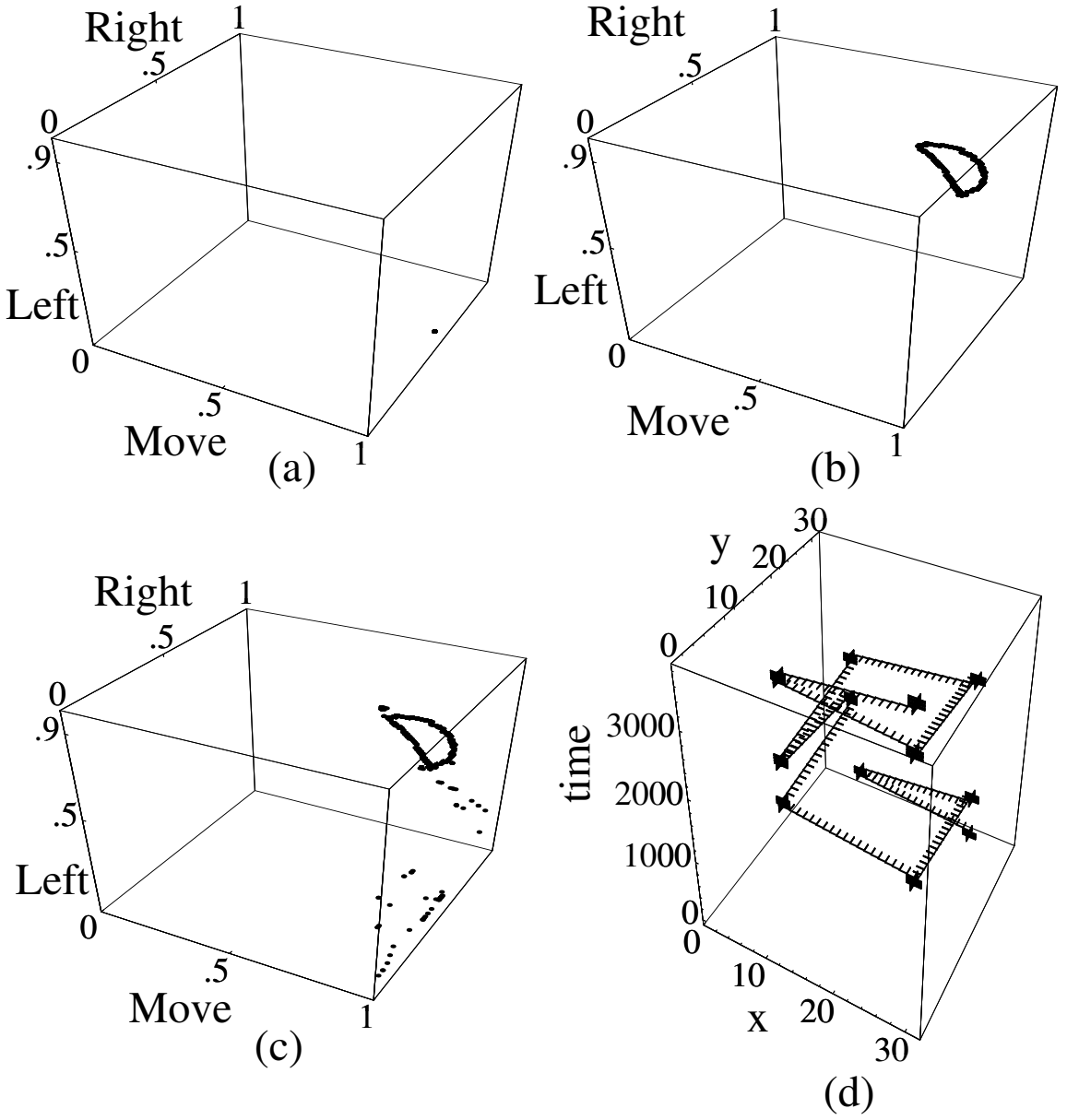


Figure 22: *Limit behavior of the network of the second run. Graphs show the state of the output units Move, Right, Left. (a) Fixed point attractor that results for sequence of 3500 "food" signals; (b) Limit cycle attractor that results when a sequence of 3500 "no food" signals is given to network; (c) All states visited while traversing the trail; (d) The path of the ant on an empty grid. The z axis represents time. The ant's path is comprised of a set of "railroad tracks." Along each track, tick marks represent back and forth movement. At the junctures between tracks, a more complicated movement occurs. There are no artifacts of the toroidal grid in this plot, all are actual movements (cf. Figure 21d).*

evaluations was 1,123,942 to acquire the equivalent network. 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 is a strong empirical validation that crossover is detrimental to the evolution of networks.

4.6 Discussion

In each of the reviewed methods for the complete network induction problem, both by connectionist and evolutionary algorithms researchers, a heuristic for architectural manipulation was designed into the problem solver to compensate for the lack of a stronger method for determining an appropriate architecture for the task. Each of these heuristics resulted in a system that forced a task into an architecture rather than fit an architecture to the task. Thus these methods only partially addressed the complete network induction problem.

GNARL, on the other hand, allowed the interaction with the task environment to constrain the acquisition of appropriate architectures rather than provide a over-simple heuristic. The dynamics reviewed in Chapter 3 for evolutionary weak methods set up race conditions in the population. In effect, the dynamics of the GNARL algorithm select for networks that can propagate within the average perturbations imposed by the mutation operators and can continue to increase their ability at a better than average rate. Consider that as a particularly good network is proliferating through the population, there are others competing for the same resources, i.e. vying for more allotment of credit. If one type of network is in a solution dense area of the search space, relative to the defined reproduction operators, then it has a better chance to create networks that are even better than itself. This sets up an implied "arms race" where the networks that lead to better offspring more quickly, relative to the task environment, are favored. Any network whose progeny can not evolve at a similar rate are discarded.

At a high level, the task environment, here the training sets and the error functions that assign an error to the network's performance, the population appears to acquire networks that are more and more suited to the task. Architectures that are too brittle, i.e. can not be manipulated by GNARL's operators, can not progress through the population. But architectures that are mutable within reasonable limits and still perform adequately on the task

can proliferate through the population. Thus, the knowledge of what architectures are appropriate for the task emerges from interaction of GNARL with the task environment.

The results of GNARL for the complete induction of networks for regular languages are comparable to the results Watrous and Kuhn (1992) which assume a single architecture. Because GNARL makes only minimal architectural assumptions, GNARL may be competitive against all such training algorithms when an inappropriate architecture is assumed. GNARL's ability to generalize to the languages consistently better than Watrous and Kuhn (1992) follows from a similar argument.

The results of the ant experiment also demonstrate the ability of GNARL to acquire a variety of solutions with competitive speed while reinforcing the arguments of this chapter on the selective ability of crossover as a reproductive operator. GNARL's acquisition of two recurrent networks that differ greatly in their approaches to solving the Tracker task is indicative of both the generality of the recurrent neural network and the lack of bias in the GNARL algorithm. The comparison between the first network induced and shown in Figure 19b and the handcrafted FSA of Figure 20 illustrate that while the behavior of these ants are nearly identical, the complexity of the underlying machine that provides the behavior can be arbitrary. This supports the observations of Pollack (1991) and reinforces the need to distinguish between observed behavior and the method of implementation from which it emerged.

4.7 Conclusion

The type of emergent intelligence demonstrated in this chapter is a natural consequence of using an evolutionary weak method for solving a problem. The acquisition of an appropriate recurrent network is a straight-forward usage of empirical credit assignment and the emergent knowledge is of the same variety that could be ascribed to problem solvers using situated models, like Pengi (Agre and Chapman 1987) and Blockhead (Chapman 1989) and Brook's (1986; 1991) embedded systems. The following chapters demonstrate the emergence of knowledge that improves on this process.