Astro Teller

There is a fundamental problem with genetic programming as it is currently practiced, the genetic recombination operators that drive the learning process act at random, without regard to how the internal components of the programs to be recombined behaved during training. This research introduces a method of program transformations that is principled, based on the program's *internal behavior*, and significantly more likely than random local sampling to improve the transformed programs' fitness values. The contribution of our research is a detailed approach by which principled credit-blame assignment can be brought to GP and that credit-blame assignment can be focused to improve that same evolutionary process. This principled credit-blame assignment is done through a new program representation called *neural programming* and applied through a set of principled processes called, collectively, *internal reinforcement in neural programming*. This *internal reinforcement* of evolving programs is presented here as a first step toward the desired gradient descent in program space.

14.1 Introduction

There is a fundamental problem with evolutionary computation, and particularly with genetic programming, as it is currently practiced. The problem is that in the space of programs, even if it has been carefully defined so that most or all examined programs are legal, the density of functions that do something "interesting" is very low. This is increasingly the case as the expressiveness of the language in which the programs are written moves up the ladder from regular languages to Turing machines. For example, in the space of Turing machines, the density of programs that act for multiple steps and then halt is conjectured to be set of measure zero [Hopcroft and Ullman, 1979].

This low density of computationally non-trivial programs, combined with the random recombination that still characterizes genetic programming, has marginalized GP, an exciting and valuable subfield of machine learning. GP needs search operators that tend to focus on good solutions and GP search operators are currently not focused, but instead altered by random transformations.

We develop a novel solution to this problem. As part of the evolutionary process, we introduce a method of program transformations that is principled, based on the program's behavior, and significantly more likely to create new programs that are worth searching than random local sampling. The contribution of our research is the identification of this problem in genetic programming and a detailed approach, both comprehensive and analytical, on how to address it. The main algorithmic innovation of this research is the process by which principled credit-blame assignment can be brought to evolution of algorithms and that credit-blame assignment can be used to improve that same evolutionary process. This principled credit-blame assignment is done through a new program representation called *neural programming* (NP) and applied through a set of principled processes called, collec-

tively, *internal reinforcement in neural programming* (IRNP). This *internal reinforcement* of evolving programs is presented in this chapter as a first step toward the desired gradient descent in program space.

Genetic programming is a successful representative of the machine learning practice of *empirical credit assignment* [Angeline, 1993]. Empirical credit assignment allows the dynamics of the system to implicitly determine credit and blame. Evolution does just this [Altenberg, 1994]. Machine learning also has successful representatives (e.g., ANNs) of the practice of *explicit credit assignment*. In explicit credit assignment machine learning techniques, the models to be learned are constructed so that why a particular model is imperfect, what part of that model needs to be changed, and how to change the model can all be described analytically with at least locally optimal (i.e., greedy) results. To be clear, this work on *internal reinforcement* is not only an attempt to approach gradient descent in program space. Internal Reinforcement is also designed to bridge this credit-blame assignment gap by finding ways in which explicit and empirical credit assignment can find mutual benefit in a single machine learning technique. In summary, the main question that our research addresses is:

> Can the evolution of algorithms be extended in a domain-independent way to incorporate accurate credit-blame assignment of each program's internal structure and behavior in such a way that focused, principled reinforcement information improves the evolutionary process?

14.2 Neural Programming

Genetic programming is a successful machine learning technique that provides powerful parameterized primitive constructs and uses evolution as its search mechanism. However, unlike some machine learning techniques, such as Artificial Neural Networks (ANNs), GP does not have a principled procedure for changing parts of a learned structure based on that structure's past performance. GP is missing a clear, locally optimal update procedure, the equivalent of gradient-descent backpropagation for ANNs. Why adapt GP instead of simply using a machine learning technique like ANNs? In general, it is not possible to give an ANN an input for every possible parameterization of each user defined primitive function that a GP program can be given. And it is far from obvious how to work complex functions into the middle of an otherwise homogeneous network of simple non-linear functions. Yet gradient-descent learning procedures, like backpropagation in ANNs, are an extremely powerful idea. Backpropagation is not only a kind of local performance guarantee, it is a kind of performance explanation. It is to achieve this kind of dual benefit that the research this chapter reports on was undertaken.

This chapter shows how to accumulate explicit credit-blame assignment information in the Neural Programming representation. These values are collectively referred to as the *Credit-Blame map*. By organizing the GP programs into a network of heterogeneous nodes and replacing program *flow of control* with *flow of data*, we can use the Credit-Blame map to propagate punishment and reward through each evolving program.

The goal of *internal reinforcement* is to provide a reasoned method to guide search in the field of program induction. Hill-climbing in a space means sampling local points and then choosing the best of those to continue from. When the gradient is available, however, it is always better (locally at least) to move in the direction of the gradient. Program evolution can work with random samplings of nearby point in program space, but can work much more effectively with *internal reinforcement*. We introduce internal reinforcement as a program evolution approximation to the gradient function in program fitness space. Said another way, it would be desirable to be able, in GP, to have reinforcement of programs be more specific (directed towards particular parts or aspects of a program) and more appropriate (telling the system *how* to change those specific parts). As will become clear later in this chapter, internal reinforcement is a partial, not complete solution to this problem.

In Section 14.3, we describe how this representation can be used to deliver explicit, useful, internal reinforcement to the evolving programs to help guide the learning search process. And in Section 14.4, we demonstrate the effectiveness of both the representation and its associated internal reinforcement strategy through an experiment on an illustrative signal classification problem.

14.2.1 The Neural Programming Representation

The essence of a programming language is a set of basic constructs and a set of legal ways of combining those constructs. A measure of the extensibility of a language is the ease with which new constructs or new construct combinations can be add to the language. It is the high degree of extensibility in GP that we want to wed to the focused update policies possible in other machine learning techniques.

The Neural Programming representation consists of a graph of nodes and arcs that support a **flow of data**, rather than the flow of control typical in programming languages. The nodes in a neural program compute arbitrary functions of the inputs. So a node can be the sum of inputs to the node and a sigmoid threshold. But it can also use other functions such as MULT, READ, WRITE, IF-THEN-ELSE, and, most importantly, potentially complex user defined functions for examining the input data. Examples NP programs are given later in this chapter. Figure 14.1 contains the important characteristics of the NP representation.

Throughout this chapter, the characteristics of NP will be used and discussed in greater detail. Let us here highlight a few aspects of NP. A parameterized signal primitive (PSP)

- An NP program is a general graph of nodes and arcs.
- Each NP node executes one of a set of functions (e.g., Read-Memory, Write-Memory, Multiply, Parameterized-Signal-Primitive-3, etc.) or zero-arity functions (e.g., constants, *Clock*, etc.).
- An arc from node x to node y (notated (x, y)) indicates that the output of x flows to y as an available input.
- On **each** time step t $(0 < t < T)$, **every** node takes some of its inputs, according to the arity of its function, computes that function, and outputs that value on *all* of its output arcs. *Data flow, not control flow.*
- One type of node function is "Output." Output nodes collect their inputs and create the program response through a function *OUT* of those values. In this chapter *OUT* is a simple weighted average. Each value is weighted by the timestep it appears on.

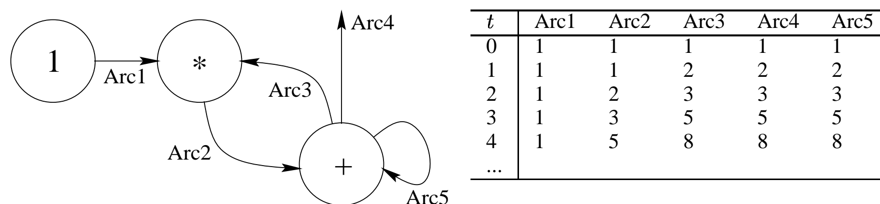
Figure 14.1
The critical characteristics of the NP representation.

is a piece of code, written by a user that expresses a way of extracting information of the input signal in a parameterized form. An example PSP might return the AVERAGE or VARIANCE of values in a range of the input data as specified by the inputs to that node. This kind of embedding of complex (often co-evolved) components as primitives in the evolving GP system has repeatedly been shown to be effective (e.g., [Koza, 1994]). Furthermore, these powerful parameterized-signal-primitives, as part of the learning process, can be used in place of brittle (fixed/static) preprocessing. As has been discussed already, the salient distinction here is the parameterization of input "features."

Each NP node may have many output arcs. See Figure 14.3 for a simple example. The multiple forked distribution of good values from any point in the program is a valuable aspect of the NP representation. Seen from a GP vantage, this is similar to a kind of highly flexible automatically defined functions (ADF) [Koza, 1994] mechanism. The idea is that once a "valuable" piece of information has been created, it can be sent to different parts of the NP program to be used further in a variety of different ways. This fan-out is an advantage of connectionist representations from which GP program representations can profit by incorporating (see [Nordin, 1997] for an alternate method).

A timestep threshold T (see Figure 14.1) is imposed on the evolving programs (in order to avoid having to solve the Halting problem). Given such a threshold, a reasonable question to ask is, "How much of a burden is this threshold?" or alternately "Can the evolving programs take advantage of additional time in which to examine an input signal?" The answers is to these questions are provided in Section 14.4.4.

There are two dominant forms of change that evolving programs typically undergo: crossover and mutation.



t	Arc1	Arc2	Arc3	Arc4	Arc5
0	1	1	1	1	1
1	1	1	2	2	2
2	1	2	3	3	3
3	1	3	5	5	5
4	1	5	8	8	8
...					

Figure 14.2
A simple NP program that computes the successive elements of the Fibonacci series. All input/arc values are 1 on the first time step. On the right, progression of arc values over time.

While NP programs look more like recurrent ANNs than traditional tree-structured GP programs, NP programs are changed, not by adjusting arc weights (NP arcs have no weights), but by changing both what is inside each node as well as the topology and size of the program.

14.2.2 Illustrative Examples

NP programs are evolved and explanations using evolved examples are not practical because the evolved examples are not concise. Instead we illustrate the NP representation through a set of constructed examples. Of course, any of the following example programs and program fragments could have been the result of evolution.

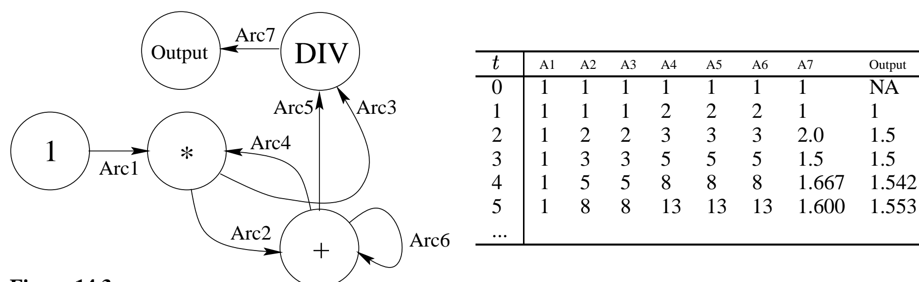
14.2.2.1 Example 1: The Fibonacci Series

Figure 14.2 shows an extremely simple NP program. This program computes the Fibonacci series, sending successive elements out on Arc4. The Fibonacci series is defined to be Fib(n) = Fib($n - 1$) + Fib($n - 2$) with Fib(0)=Fib(1)=1.

There is only one initialization necessary for the correct operation of NP programs: "what input values should all nodes use on their very first computation?" Since NP programs are data flow machines, each arc is a potential input value and so there must be some initial state to the program. For this example, let us initialize each program so that all arcs have the value 1 when a program starts up. Figure 14.2 also shows how the values of the arcs change over time.

14.2.2.2 Example 2: The Golden Mean

Let us now change slightly the computation of the simple NP program from example 1. Instead of producing a list of exponentially increasing values (as in the program shown in Figure 14.2) let us design an NP program that approximates the "Golden Mean" ($\frac{fib(i)}{fib(i-1)} = 1.618034$) through its OUTPUT node. To do this, all we need to do is to add an extra node that does Division (DIV) and pass it as its two parameters (i.e., its two input arcs) $fib(i)$ and



t	A1	A2	A3	A4	A5	A6	A7	Output
0	1	1	1	1	1	1	1	NA
1	1	1	1	2	2	2	1	1
2	1	2	2	3	3	3	2.0	1.5
3	1	3	3	5	5	5	1.5	1.5
4	1	5	5	8	8	8	1.667	1.542
5	1	8	8	13	13	13	1.600	1.553
...								

Figure 14.3
A simple Neural Program that iteratively improves an approximation to the golden mean. This program assumes that all input values are 1 on the first time step. On right, progression of arc values over time.

fib$(i-1)$ as they are computed (shown in Figure 14.3). Figure 14.3 also shows how this computation plays out through the arcs as the timesteps pass.

14.2.2.3 Example 3: Foveation

Foveation is the process changing focus of attention in response to previous perceptions. For example, this iterative process of foveation is what gives us the illusion of seeing with high-resolution across our field of vision when, in fact, our fovea (the high resolution area of the retina) fills less than 5% of our field of view.

The Fibonacci examples illustrate how the flow of data works and how the fan out of values can significantly reduce the size of a solution expression. In this example we illuminate another important feature of NP programs: the ability to foveate. NP programs have the ability to use the results of an examination of the input signal to *guide* the next part of that examination. NP programs view their inputs (called *signals* when appropriate to avoid confusion with "inputs" to a node) through *Parameterized Signal Primitives* (PSP), variable argument functions defined by the NP user.

Let us assume that this NP program is examining signals that are video images. PSP-Variance is a function that takes four arguments, a_0 through a_3, (interpreted as the rectangular region with upper-left corner (a_0, a_1) and lower-right corner (a_2, a_3)) as input and returns the variance of the pixel intensity in that region. Figure 14.4 shows what could be part of a larger NP program. The node indicated with a double circle computes the function PSP-Variance.

To simplify the explanation, this particular NP program fragment delivers static values for three of those four inputs. The fourth input indicated by a dashed circle, changes as the program proceeds. That means that PSP-Variance, at each time step, computes its function over the region (50,17,104,a_3). The simplest way to explain this mechanism is to give the pseudo-code to which it is equivalent (see Figure 14.4). Assuming again that all arcs are initialized to 1, this program finds a one-sided local minimum of PSP-Variance with respect

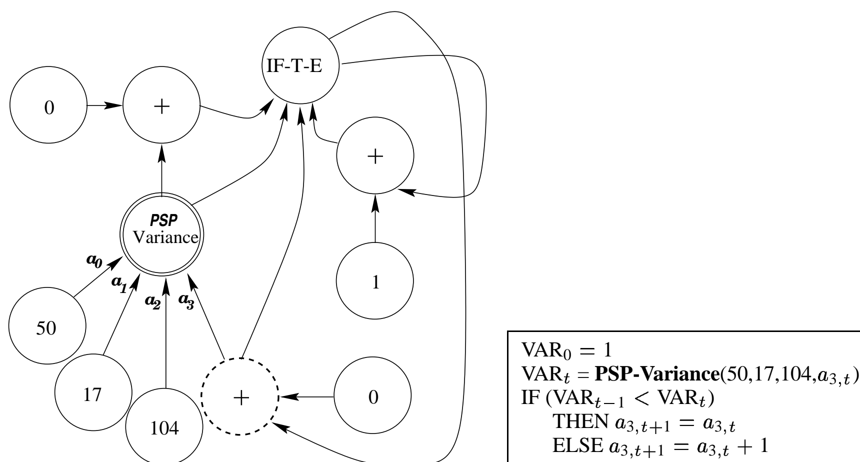


Figure 14.4
A simple NP program fragment. The output value from the dashed circle node is being iteratively refined to minimize the value returned by the PSP-Variance node. On right, pseudo-code for the behavior of this NP program fragment. VAR$_t$ denotes the value of VAR at time t and $a_{3,t}$ denotes the value of argument a_3 at time t.

to its fourth parameter. In general, the program fragment increments the fourth parameter only if (PSP-Variance(50,17,104,$a_{3,t}$) < PSP-Variance(50,17,104,$a_{3,t-1}$)) (where $a_{3,t}$ is a_3 on timestep t). This is a concise example of an NP program foveating: using the values the program receives through it's PSPs to focus further examination of the input signal.

14.3 Internal Reinforcement in NP

Evolution is a learning process. In NP (or GP for that matter) programs are tested for fitness, preferred according to those fitness tests, and then changed. These program transformations have a specific goal, to produce programs that are better, which is to say score higher on the fitness evaluations, than their ancestors. Much of the time this will not happen, but the success of evolution as a learning process is directly linked to how often a novel program is really more valuable than the parent it came from. Currently, program transformations are usually random in GP. Even when they are not random, they do not transform the programs based on how those programs have behaved in the past. If we could only look into a program and see which parts of it are "good" and which parts "bad," we could write transformation rules that were much more effective, which is to say, we could dramatically improve the action of evolution. That is the motivation for the principled update procedure at the heart of this research: *internal reinforcement*.

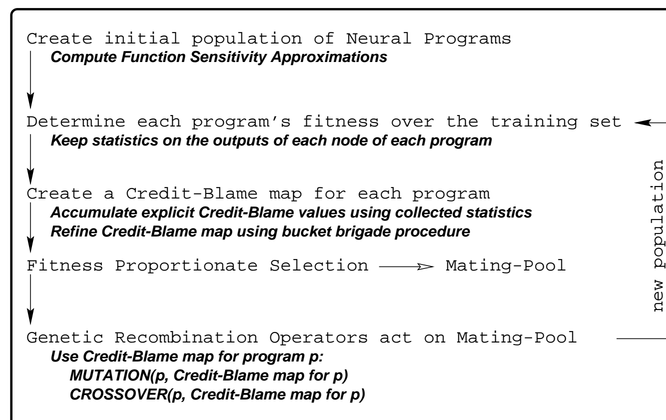


Figure 14.5
The high level flow of NP learning.

Now that we have introduced the neural programming representation, we can describe a mechanism to accomplish *internal reinforcement*. In Internal Reinforcement of Neural Programs (IRNP), there are two main stages. The first stage is to classify each node and arc of a program with its perceived contribution to the program's output. This set of labels is collectively referred to as the *Credit-Blame map* for that program. The second stage is to use this Credit-Blame map to change that program in ways that are likely to improve its performance.

Our ongoing research includes investigation into which methods to use to best accomplish the goals of internal reinforcement. We have identified several methods for accomplishing each of the two stages. This chapter focuses on one technique for each of the two stages.

Figure 14.5 shows the evolutionary learning process for NP and how IRNP fits into that picture. One Credit-Blame map is created *for each population program* and when the time comes to perform genetic recombination on a particular program, the Credit-Blame map for that particular program is used.

14.3.1 Creating a Credit-Blame Map

Without loss of generality, we can assume that the evolving NP programs are trying to solve a target value prediction problem. This is so because classification problems (a non-ordered set of output symbols to be learned) can be decomposed into target value prediction problems (an ordered set of output symbols to be learned). This decomposition takes the

general form of mapping 1 C-way classifi cation problem of the form "To which of C classes does this input belong" into C different binary classifi cation problems of the form "Does this input belong to class i ($1 \leq i \leq C$) or not?" (see [Teller, 1998] for details). Therefore, let us consider an abstract input to output mapping to be learned by the neural programs.

14.3.1.1 Accumulation of Explicit Credit Scores

For each program p, for each node x in p, over all time steps on a particular training example S_i, we compress (combine) all the values node x outputs into a single value H_x^i. The compression function used in this chapter is the *mean*. Let the correct answer (the correct target value) for training instance S_i be L_i. In other words, L_i is the desired output for program p on training instance S_i.

We now have two vectors for all $|S|$ training instances: $\vec{L} = [L_1..L_i..L_{|S|}]$ and $\vec{H_x} = [H_x^1..H_x^i..H_x^{|S|}]$. We can compute the statistical correlation between them. We call the absolute value of this correlation the explicitly computed **Credit Score** for node x, notated as $\mathbf{CS}_x$. This computation is shown in Equation 14.1 (in which E is the *expected value*).

$$\text{CS}_x = \left| \frac{E(\vec{H_x} - \mu_{\vec{H_x}}) * E(\vec{L} - \mu_{\vec{L}})}{\sigma_{\vec{H_x}} \times \sigma_{\vec{L}}} \right| \tag{14.1}$$

This credit score for each node is an indication of how valuable that node is to the program. It is the case that nodes with low credit scores at this stage may still be critical to the program in question, but it is also certainly the case that nodes with high credit scores could be very valuable to the program, even if they are currently unutilized. Note that an NP program is, by defi nition, 100% correct if it has a node with a credit score of 1 and that node is the only node with an outgoing arc that terminates in an OUTPUT node. This *explicit* credit score can also be thought of as the *individual* credit score for the node. That is, the explicit credit score takes into account only how the node acts as an individual, not how it acts as part of a group of tightly coupled nodes (i.e., the program it is a part of).

The set of explicit credit scores for all nodes provides a Credit-Blame map for the program: a value associated with each node in the program that indicates its individual contribution to the program. However, we want the Credit-Blame map to capture not only a node's immediate (individual) usefulness, but also it's usefulness in the context of the program topology. The following example highlights why the explicit credit scores do not, by themselves, capture this information.

In this example, nodes x and y produce values and node z computes an XOR of these two values. In this case, even if z has a high credit score, x and y may not (e.g. $\text{CS}_z = 0.97, \text{CS}_x = 0.14, \text{CS}_y = 0.07$). There is nothing provably wrong with this situation but clearly, the topological notion of usefulness has not been captured in these explicit credit

scores. This can be seen because the nodes x and y in this example are partly responsible for node z's success (and are therefore useful) but still have low credit scores.

The Credit-Blame map can be refined to attend to this type of indebtedness relationships by passing credit and blame back through the NP programs along the arcs. The statistical correlation between $\vec{L}$ and $\vec{H}_x$ constitutes a first approximation to the credit score for node x. Because nodes are connected to each other and only a few are directly connected to the OUTPUT node, and because each node performs a specific function, the Credit-Blame map needs to be further refined. This process of refining the Credit-Blame map to take advantage of the topology of the program is described in Section 14.3.1.3.

14.3.1.2 Function Sensitivity Approximation

To pass back credit and blame through the neural program topology, we must first answer an important question: "How does each node act as a function of its inputs?" In other words, "What is the responsibility of each input parameter for the output value produced by each atomic function used in evolution?" This problem is very difficult for arbitrary functions, which is one of the main reasons why ANN backpropagation requires differentiable functions (e.g., the sigmoid or the Gaussian). Unfortunately, we can not always differentiate the functions used in NP programs as they may not always be differentiable (e.g. If-Then-Else).

In our work, we introduce *Function Sensitivity Approximation*, a method for "differentiating" an arbitrary function that can be treated as a black box. The main question that function sensitivity approximation answers about a black box function's relation to its inputs is "For argument a_i of function f, what is the likelihood that f's output will change *at all* when the value of a_i is changed to a new *random value* selected uniformly from the legal range of values?" This discovered sensitivity is a substitute to the function's derivative. This sensitivity is written as $\mathcal{S}_{f,A,i}$, the sensitivity of a particular parameter a_i for some function f that is given a parameter vector with A elements. [Teller, 1998] contains details on how such values are automatically calculated. It should also be noted that the $\mathcal{S}_{f,A,i}$ values are computed under the assumption that each node computes a function with no side effects. [Teller, 1998] also describes NP's robustness in spite of this simplification.

14.3.1.3 Refining the Credit-Blame Map

We can now combine the topology of the NP program, the explicit credit score for each node, and the sensitivity values of each primitive function in a bucket-brigade style backward propagation. This bucket-brigade refines the credit scores at each node following the procedure presented in Figure 14.6. The credit scores are refined according to the network topology and sensitivity of the node functions. To understand why a bucket-brigade backward propagation of credit is critical, refer back to the XOR example in Section 14.3.1.1.

```
Until no further changes
 For each node x in the program
  For each output arc (x, y) of that node
    y is, by definition, the destination node of (x, y)
    Let f_y be y's node function
    Let A_y be the number of inputs y has
    Let i be such that (x, y) provides a_i to y
    Let S_{f_y,A_y,i} = Sensitivity of f_y (relative to A_y and i)
   CS_x = MAX(CS_x, S_{f_y,A_y,i} * CS_y)
```

Figure 14.6
The bucket brigade refinement of Credit Scores (CS) throughout an NP program.

The high level structure of the procedure presented in Figure 14.6 is as follows. For each node, for each output arc from that node, the node's credit-score is updated to be the maximum of the credit-score it already has and the credit-score of the node pointed to by that output arc multiplied by the sensitivity of that destination node to that particular output arc. We explain this process in detail through a series of questions and answers.

A good first question for this particular method of spreading credit and blame out more appropriately over each neural program is, "does this process always converge?" The answer is that as long as the definition of "no further changes" is more specifically "no node changed its CS value by more than ϵ" ($\epsilon > 0$) then the process always halts[1] and typically in only a few passes. Because of the way $\mathcal{S}_{f,A,i}$ is defined and implemented it is, in practice, always less than 1.0, contributing to the small number of passes required for the Credit-Blame map to reach quiescence. This answer to the convergence question is also the answer to the question, "why do not you use a discount factor (γ)? Is not that usual in various forms of bucket brigade?" Using a discount factor is a common way to insure convergence, but as just noted, it is empirically unnecessary.

In this context, in which we make clear the use of a sensitivity value for each function, we can now ask "why define sensitivity in that way?" Remember that we said that the sensitivity of function f_y with arity A to input a_i is the likelihood that the output will change *at all* when the value of a_i is changed to a new *random value* selected uniformly from the legal range of values. There is no reason to believe that, in a complex system such as an evolving NP program, a node that outputs O_1 will always have a similar effect to a node that outputs O_2, no matter how close O_1 and O_2 are on the number line. For example,

[1] Proof: If the halt criteria isn't satisfied after a pass, then at least one node credit score has increased by at least ϵ and no credit score has decreased in value (by construction, see Figure 14.6). The total value in the Credit-Blame map for program p can be at most N_p (the number of nodes in p), so the total number of loops can be no more than $\frac{N_p}{\epsilon}$.

consider the function READ-MEMORY(O_1) that returns the value stored in the program's memory array index O_1. Out of context of a particular program, READ-MEMORY(5) and READ-MEMORY(6) have as much semantic similarity as READ-MEMORY(5) and READ-MEMORY(77). For this reason, sensitivity in NP is a percentage of how often the output value of a function is changed at all, not by how much that output changes.

There is also little reason to believe that in a complex system such as an evolving NP program, any particular set of numbers is more or less likely than any other to occur as inputs to a node. The sensitivity discovery process could, for example, change a_i to $(a_i \pm \Delta)$. Then $\mathcal{S}_{f,A,i}$ would measure the likelihood that the output will change when small changes are made to the input a_i. But since, unlike explicit credit-blame assignment systems (e.g., ANNs), NP cannot enforce these small changes throughout the program, it is better to have a measure of sensitivity that matches how the inputs are likely to change: to first approximation, *uniform randomness*.

Finally, consider the equation for refining the credit scores: $\text{CS}_x = \mathbf{MAX}(\text{CS}_x, \mathcal{S}_{f_y,A_y,i} * \text{CS}_y)$. "Why should CS_x be set to the maximum of itself and $\mathcal{S}_{f_y,A_y,i} * \text{CS}_y$?" We first address the function MAX as an appropriate operator and then examine the appropriateness of the second operand. In an NP program it is the norm for a single node's output to be used in a number of different contexts. We would not want to penalize a node for creating an output that is very useful in one part of the program, but is not taken advantage of in another part of the program. If even one of the outputs of a node is "taken advantage of" (in the sense defined by the explicit credit score measure), then it is clear that the blame for not taking advantage of that output elsewhere in the program is a problem with that other part of the program, not the node in question. This means that a node's credit score should be a maximum of some function of the credit scores of the nodes to which it outputs.

Further, consider the case in which node x has an explicitly computed credit score of CS_x. Even if none of x's children (i.e., nodes that take x's output as input) has a credit score as high as CS_x, if we believe that the explicit credit score measure is a good first approximation to the usefulness of a node in a program, then we should insure that CS_x is never less than its original value. Thus, we introduce CS_x = MAX(CS_x, $F_r(\text{CS}_y)$) where F_r is some function to be determined. Now we need to pick some reasonable function F_r to apply to the credit scores of the children of node x.

The introduced sensitivity analysis of Section 14.3.1.2. can now be used. We already have a value that expresses the sensitivity of a node y to an input a_i as a function of how many inputs y has and the particular function that y happens to compute. But that's exactly what we want! The amount of reward (think CS_x) a node x that points to a node y deserves for that "reference," is exactly how good node y is, CS_y, scaled by (i.e., times) how responsive (i.e., sensitive) y is to changes in the values that x is passing it. So we have our function $F_r(\text{CS}_y)$; it is $\mathcal{S}_{f_y,A_y,i} * \text{CS}_y$.

This discussion highlighted the characteristics of our reinforcement procedure. So in summary, the refinement of credit scores in the Credit-Blame map is derived from the initial credit scores, the program's topology, and the discovered sensitivity of each possible node function.

14.3.1.4 Credit Scoring the NP arcs

NP program transformations operators (e.g., crossover and mutation) also affect NP program arcs. So far, the discussion of the Credit-Blame map has entirely focused on assigning credit and blame to the nodes. The topology of the NP programs, that is the program nodes and arcs, is used heavily in making this map, but the resulting map assigns one floating point number to each node and no number to the arcs.

The explanation for this discrepancy is that arcs are even more context dependent than the nodes that define them. For example, when considering whether to delete a particular arc (x, y), CS_y is a relevant value, but the value of CS_x is much less so. This is so because deleting one of node x's output arcs doesn't affect the other arcs from x, but deleting an input arc potentially changes what node y outputs on all it's output arcs. When, on the other hand, considering whether to reroute arc (x, y) to some other node z (i.e., $arc(x, y) \rightarrow arc(x, z)$) the current values CS_x, CS_y, and CS_z are all relevant. As is detailed in the next section, the Credit-Blame map has a great impact on the arcs during the IRNP process, but only indirectly through the credit scores of the nodes in the program to be recombined.

14.3.2 Exploration vs. Exploitation Within a Program

A tension exists between exploration (try out something new) and exploitation (stick with the best you've seen) within the recombination of a single program. IRNP could leave alone the "best" parts of the program and focus its changes on the "worse" program aspects. There are, however, two problems with this view. The first is that a "bad" part of the program must be more carefully defined. There are program nodes that have very low scores in the program's Credit-Blame map and **do** affect the values flowing into the OUTPUT nodes and there are low score nodes that **do not** affect the values flowing into the program OUTPUT nodes. This is the *node participation* problem. To be most effective, IRNP should change the first type of low score nodes, but not the second. This is so because, for example, changing what function a particular node computes is a piece of wasted search if that node's old function had no effect on any of the program's OUTPUT nodes (under the assumption that none of the functions have side-effects).

There is a second problem with seeing IRNP's job as simply focusing on the "bad" parts of a program. Occasionally, the best way to improve a program is to make the right change to an aspect of the program that is already working well. It is easy to imagine a program in

```
For each node x in the program
    Participation_x ← 0
For each node x in the program
    if (node x is an OUTPUT node)
      Participation_x ← 1
While (flags still changing)
    For each node x in the program
      if  (arc (x, y) exists and creates a_i for node y) and
          (node y has S_{f_y,A_y,i} > 0) and
          (Participation_y = 1)
        Participation_x ← 1
```

Figure 14.7
The procedure for assigning the *participation flags* to nodes in each program's Credit-Blame map.

which node y computes $a_0 + a_1$ is almost right, but the program would work even better if that node computed $a_0 * a_1$ instead.

IRNP does address both of these issues. With regards to the second problem, IRNP does occasionally change high credit-score aspects of a program. It is partly for this very reason that the mutation operators only look at a fraction of the nodes in a program before picking one to change. This means that with low probability, the "worst" program aspect seen by a particular mutation operator, will still be one of the high credit-score nodes for that program. An interesting piece of future work for IRNP is the following. Instead simply restricting how often the recombination operators change high credit-score aspects of a program, how these aspects are changed could be different. In other words, for example, mutation could be further refined so that it did "less damaging" mutations when a high credit-score node was chosen to be changed (e.g., ADD → MULT is "less damaging" than ADD → If-Then-Else).

IRNP also addresses the node participation problem. Credit-Blame map includes a *participation* flag for each program node. IRNP takes advantage of these flags by augmenting the mutation and crossover policies described in Sections 14.3.3.1 and 14.3.3.2. These participation flags are set using the process shown in Figure 14.7.

14.3.3 Using a Credit-Blame Map

The second phase of the internal reinforcement is the use of the created Credit-Blame map to increase the probability that the genetic operators lead either to a better solution or to a similar solution in less time. There are two basic ways that the Credit-Blame map can

be used to do this enhancement: through improvement of either the mutation or crossover operators.

The possibility of using internal reinforcement (explicit credit-blame assignment) not only for mutation (which has analogies to the world of ANNs) but for crossover as well is important. Traditional GP uses random crossover and relies entirely on the mechanism of empirical credit-blame assignment. Work has been done to boot-strap this mechanism by using the evolutionary process itself to evolve improved crossover procedures (e.g. [Angeline, 1996b; Teller, 1996]). This work has reaped some success, but because of the co-evolutionary nature of the work, it has not yielded deep insights into the basic mechanism of crossover. IRNP has the future potential not only to improve on the existing GP mechanism, but also to help study the central mystery of GP, namely crossover.

14.3.3.1 Mutation: Applying a Credit-Blame Map

Mutation can take a variety of forms in NP. These various mutations are: add an arc, delete an arc, swap two arcs, change a node function, add a node, delete a node. Notice that change a node function and swap two arcs are not atomic, but have been included as examples of non-atomic, but basic mutation types. In the experiments shown in the next section, each of these mutations took place with equal likelihood in both the random and internal reinforcement recombination cases. For example, to add an arc under random mutation to an NP program, we simply pick a source and destination node at random from the program to be mutated and add the arc between the nodes.

Internal reinforcement can have a positive effect on this recombination procedure. For each recombination type, we pick a node or arc (depending on the mutation type) that has maximal or minimal credit score as appropriate. For example, when deleting a program node, we can delete the node with the lowest credit score instead of just deleting a randomly selected node.

Below are the IRNP procedures for each of the six mutation types. Notice that when the terms "large" and "low" are used (as opposed to the unambiguous terms "highest" and "lowest"), this indicates that the largest or least credit score is selected from among a *sampled subset* of nodes or arcs, depending on the context.

Add an Arc : First, pick a node x with a large credit score. Then pick a node y with a low credit score **and Participation**$_y = 1$ and A inputs such that y would still be sensitive to input a_{A+1}. Finally, add an arc (x, y).

Delete an Arc : First, pick a node y with a low credit score such that y would still be sensitive to its inputs if one were removed **and Participation**$_y = 1$. Then pick a node x with a low credit score such that there exists an arc (x, y). Finally, delete arc (x, y).

Swap Two Arcs : First, let x be the node with highest CS$_x$. Then let (x, y) be the output arc of x to a node y that minimizes CS$_y$. Then, for all arcs (u, v) such that v is an OUTPUT node, pick the arc (u, v) that minimizes CS$_u$. Finally, delete arcs (x, y) and (u, v) and create arcs (x, v) and (u, y).

Change a Node Function : First, pick a node x that has a low credit score and such that (x, y) exists and creates input a_i for node y and $\mathcal{S}_{f_y,A_y,i} > 0$ **and Participation**$_y = 1$. Then change the function that x computes to another function of similar or lower arity.

Add a Node : First, create a new node z with f_z, a randomly selected function. Then let A be the arity of f_z and let O_z be the number of output arcs from z. Then find high credit score nodes x_1, ...x_A and create the arcs (x_1, z) ... (x_A, z). Then find low credit score nodes y_1, ... y_{O_z} such that $\mathcal{S}_{f_{y_i},A_{y_i}+1,A_{y_i}+1} > 0$ **and Participation**$_{y_i} = 1$ for all i in $[1..O_z]$. Finally, create the arcs (z, y_1) ... (z, y_{O_z})

Delete a Node : First, pick a low credit score node x **with Participation**$_x = 1$. Then, remove x and arcs (x, y) and (z, x) for all nodes y and all nodes z in the program.

For each of the procedures, the alternative to IRNP is the equivalent of the traditional recombination strategy in GP. This less focused strategy in NP is simply to chose randomly among all syntactically legal options (i.e., no program-behavior based bias in the recombination). Equivalently, this "vanilla" method for recombination can be thought of as IRNP with random values in the Credit-Blame map.

14.3.3.2 Crossover: Applying a Credit-Blame Map

In the random version of crossover, one simply picks a "cut" from each graph (i.e., a subset of the program nodes) at random and then exchanges and reconnects them. Figure 14.8 pictures this division of a program into two pieces. Details on how this fragment exchange can be accomplished so as to minimize the disruption to the two programs can be seen in [Teller, 1996]. In summary, sewing two fragments back together so as to minimize disruption is largely a matter of satisfying as far as possible the criteria that each "dangling" output arc is connected to an node that lost an input arc when its program was fragmented.

We keep this underlying mechanism and present an IRNP procedure that selects "good" program fragments to exchange. This means that IRNP has, as its only job to choose the fragments to be exchanged, but the way in which program fragments are exchanged and reconnected is unaffected by IRNP. There is much to be gained by taking advantage of the Credit-Blame map during this fragment exchange and reconstitution phase, but to focus the research work and contributions, this aspect of the use of credit-blame assignment has been left as future work.

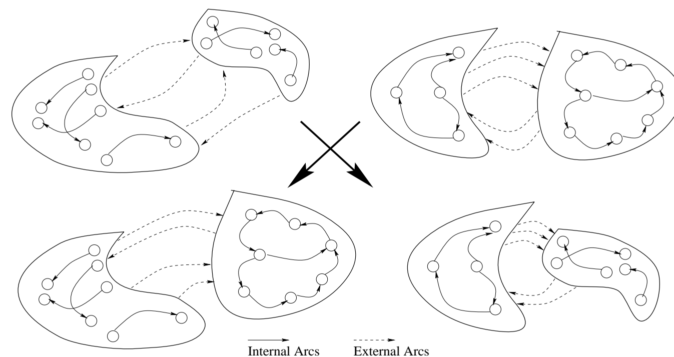


Figure 14.8
Crossover in NP: A single graph of nodes and arcs is fragmented with a cut into two fragments such that every node in the original graph is now either in Fragment_1 or Fragment_2 and every arc is either an *internal* or *external* arc.

Given that we separate a program into two fragments before crossover, let us define *CutCost* to be the sum of all credit scores of *inter*-fragment arcs, and *InternalCost* to be the sum of all credit scores of *intra*-fragment arcs in the program to be crossed-over.

NP program arcs have a shifting meaning and so their credit score must be interpreted within the context of the search operator being used. For crossover we take the credit score of an arc to be the credit score of its destination node. This is done because, as was described in Section 14.3.1.4, the disruption of an arc affects the destination node more so than the source node.

Now we say that the cost of a particular fragmentation of a program is equal to *CutCost*/*InternalCost*. If we try to minimize this value for both of the program fragments we choose, we are much less likely to disrupt a crucial part of either program during crossover. Figure 14.9 outlines this IRNP crossover procedure.

14.3.4 The Credit-Blame Map Before/After Refinement

This chapter has explained exactly how IRNP is carried out and the impact that it has on the evolution of the programs involved. It was claimed that the bucket brigade algorithm described in Section 14.3.1.3 actually does spread the credit score values out to aspects of the program that previously were not rewarded. This section illustrates this value spreading using a real snap-shot during the IRNP in a normal run. Table 14.1 shows a typical (though small) NP program (without the arcs) from generation 8 of a run learning to classify signals from a manufactured signal domain.

The first set of numbers in Table 14.1 shows the Credit Scores for each node at an inter-

```
Pick k random cuts of prog p (Fragment_1, Fragment_2)
For candidate cut i
  For each arc(x, y) in p
    Let CS_arc(x,y) = CS_y
    if (x and y are in the same Fragment_j) (j∈{1,2})
      InternalCost = InternalCost + CS_arc(x,y)
    else
      CutCost = CutCost + CS_y
  CutRanking_i = CutCost / InternalCost
Choose the cut that produced the LOWEST CutRanking with
at least one participating node on each side of the cut
```

Figure 14.9
The IRNP process for choosing a "good" fragment of a program to exchange through crossover.

mediate stage in the credit-blame assignment process as described in this chapter. Namely, the credit scores shown in Table 14.1 have undergone the process of Section 14.3.1.1, but not the process of detailed in Section 14.3.1.3. The second set of numbers in Table 14.1 shows this same NP program after the bucket brigade refi nement process has taken place.

The bold faced credit scores in Table 14.1 are those values that changed during the bucket brigade credit score refi nement process. Notice that more than half of the credit scores changed values during this process, many of them dramatically. The number of credit scores at 0.0 dropped from 52.17% to 17.39% due to the refi nement process. Notice also that even the OUTPUT nodes have their credit scores changed during this process since the output from an OUTPUT node may be very useful, even if it is not, itself, the highest correlation node in the program.

14.3.5 IRNP Discussion

It should have been made clear by this point in this chapter that NP programs are "nearly" Turing complete in that they have a suffi ciently complex function set, memory, and iteration. The topology and execution of NP programs provides iteration. Technically, a Turing complete program must have access to arbitrarily extendible memory, though in practice this is never actually provided. In NP, the form of memory that has been described, and that will be used throughout the rest of this chapter, is the data-flow memory of a program. A program with, for example, 312 arcs has a memory capacity of 312 distinct values and many billions of states even for a restricted value range. This is *implicit memory use* (i.e., memory use through the representation itself) rather than *explicit memory use*.

Table 14.1
A sample NP program (without the arcs) at the end of generation 8 **after** the Credit Scores have been assigned, shown both **before** and **after** the bucket brigade refi nement of this Credit-Blame map has taken place. # indicates the node number. *The bold values highlight Credit Score values that changed during the refinement process.*

Credit-Blame map **before** Refi nement. (Explicit Credit Scores)

#	$CS_{\#}$	Function	#	$CS_{\#}$	Function	#	$CS_{\#}$	Function	#	$CS_{\#}$	Function
0	0.0000	060	1	0.0000	Clock	2	0.0000	144	3	0.0000	Clock
4	0.0000	211	5	0.0000	Clock	6	0.0000	094	7	0.0000	182
8	0.0000	145	9	0.0000	165	10	0.0000	182	11	0.0000	045
12	0.0000	036	13	0.2069	Output	14	0.2562	PSP-Pnt	15	0.0000	Divide
16	0.6973	Output	17	0.1301	Add	18	0.1963	PSP-Max	19	0.3576	Add
20	0.0000	Multiply	21	0.3315	Output	22	0.2254	PSP-Pnt	23	0.0391	Multiply
24	0.3143	Subtract	25	0.0000	If-T-E	26	0.2254	Multiply	27	0.2380	If-T-E
28	0.0000	Split	29	0.0915	PSP-Max	30	0.7351	Split	31	0.4334	Subtract
32	0.1208	Add	33	0.0000	Subtract	34	0.4335	PSP-Pnt	35	0.0046	Add
36	0.0000	Multiply	37	0.0000	Add	38	0.2822	PSP-Max	39	0.2254	Multiply
40	0.0000	If-T-E	41	0.0162	Add	42	0.6992	Output	43	0.3315	Subtract
44	0.0000	Add	45	0.0000	Add	46	0.0000	Add			

Credit-Blame map **after** Refi nement.

#	$CS_{\#}$	Function	#	$CS_{\#}$	Function	#	$CS_{\#}$	Function	#	$CS_{\#}$	Function
0	0.0000	060	1	0.0000	Clock	2	0.0000	144	3	**0.6681**	Clock
4	0.0000	211	5	**0.1185**	Clock	6	**0.4326**	094	7	**0.0854**	182
8	**0.0011**	145	9	**0.0858**	165	10	**0.1665**	182	11	**0.6942**	045
12	**0.1629**	036	13	**0.2538**	Output	14	0.2562	PSP-Pnt	15	**0.0398**	Divide
16	0.6973	Output	17	**0.6406**	Add	18	**0.6857**	PSP-Max	19	0.3576	Add
20	0.0000	Multiply	21	**0.6811**	Output	22	**0.6675**	PSP-Pnt	23	0.0391	Multiply
24	**0.6858**	Subtract	25	0.0000	If-T-E	26	**0.2502**	Multiply	27	0.2380	If-T-E
28	**0.4254**	Split	29	0.0915	PSP-Max	30	0.7351	Split	31	0.4334	Subtract
32	0.1208	Add	33	0.0000	Subtract	34	**0.4335**	PSP-Pnt	35	0.0046	Add
36	**0.2637**	Multiply	37	**0.6854**	Add	38	0.2822	PSP-Max	39	**0.6697**	Multiply
40	0.0000	If-T-E	41	**0.6550**	Add	42	0.6992	Output	43	**0.6834**	Subtract
44	**0.6437**	Add	45	**0.6714**	Add	46	**0.0001**	Add			

Indexed Memory [Teller, 1994] is an example of *explicit memory use* in GP. In Indexed Memory, the evolving program is given access to an array of memory cells through the two functions READ(O_0), and WRITE(O_0,O_1). READ(O_0) returns the value stored in MEMORY[O_0]. WRITE(O_0,O_1) returns the value stored in MEMORY[O_0] and has the side-effect of updating MEMORY[O_0] to it's new value: O_1. Indexed Memory has been extensively studied in GP (e.g., [Teller, 1994; Andre, 1995; Langdon, 1995; Langdon, 1996; Spector and Luke, 1996]) and has demonstrated itself to be a valuable form of memory use for evolving programs. [Teller, 1998] reports on positive results on the use of indexed memory in NP learning.

NP and IRNP were designed simultaneously to add computational expressiveness to algorithm evolution (NP vs. traditional tree-GP) and to solve some of the learning difficulties this new level of computational expressiveness introduced (IRNP vs. unguided genetic operators). This does not mean, however, that IRNP only applies to NP. The concept of internal reinforcement is very general and can be illustrated in other representations. The single most popular representation for algorithm evolution is the tree representation (S-expression) of traditional GP. [Teller, 1998] describes in detail how to implement IRNP in tree-based GP.

14.4 Experimental Results

Because the following experiments were run in the context of the PADO learning system, a few words must be given as explanation. PADO is a learning environment that decomposes classification problems into discrimination problems, evolves sub-solutions to these discrimination problems, and then orchestrates these sub-solutions into an overall solution to the original classification problem. The evolution that takes place inside PADO can be of any type and in the context of this chapter that evolution is the evolution of NP programs with and without IRNP for the purposes of comparison. Significant detail about PADO can be found in [Teller and Veloso, 1995; Teller and Veloso, 1997; Teller, 1998].

14.4.1 Experimental Overview

The purpose of any set of experiments is to test a set of hypotheses. The goal of the experiments shown in this chapter are to demonstrate two general attributes of the NP and IRNP approaches. The first is that the NP representation successfully applies to a wide variety of signal domains. The second is that the IRNP procedure does substantially improve learning across a variety of signal domains.

Table 14.2 gives the values for the most important parameters and Table 14.3 gives the fixed set of program primitives used in all the experiments. [Teller, 1998] addresses the issue of IRNP's sensitivity to the parameters shown in Table 14.2, and shows in an experiment that IRNP works at least as well when crossover is the dominant recombination strategy (instead of mutation as shown in Table 14.2).

Experiments are run on two dissimilar domains and in both, the PADO approach does quite well. The experiments show both that NP programs can be usefully evolved and that in all those cases PADO does noticeably better when IRNP is active as part of the learning process. Notice in particular that, empirically, the harder the problem, the greater the efficiency gain provided by IRNP.

Table 14.2
Fixed experimental values for the most important PADO parameters.

Crossover Percent Chance	36
Mutation Percent Chance	60
Population Size	(250$\times$ Number of Classes)
Maximum Number Nodes	80
Minimum Number Nodes	10
Number In Tournament	5
Number Time Steps To Run	10
Maximum Generations	80
Maximum Number Outputs	5
Maximum Number Inputs	4

Table 14.3
PADO program primitives used

Manipulation type					
Continuous	Add	Sub	Mult	Div	OUTPUT
Choice	If-Then-Else	Split			
Signal	SignalPrimitive_0	...	...	...	SignalPrimitive_i
Zero-Arity	0..MaxValue	Clock			

A word of description about the method of presentation before we launch into the experiments. Each point on each graph is the mean performance level achieved on over many independent runs, meaning that the results presented are not the best NP has ever done on a particular domain, but a report of the kind of performance you can expect from during an average run.

14.4.2 Natural Images

There are seven classes in the domain used in the following experiments. Figure 14.10 shows one randomly selected video image from each of the seven classes in both the training and testing sets. This particular domain was created as a domain for machine learning and computer vision [Thrun and Mitchell, 1994]. Each element is a 150x124 video image with 256 level of grey. Originally, these images were color images, but the color was later removed from the images to make the problem sufficiently difficult to be interesting [Teller and Veloso, 1997].

The seven classes in this domain are: Book, Bottle, Cap, Coke Can, Glasses, Hammer, and Shoe. The lighting, position and rotation of the objects varies widely. The floor and the wall behind and underneath the objects are constant. Nothing else except the object is in the image. However, the distance from the object to the camera ranges from 1.5 to 4 feet and there is often severe foreshortening and even deformation of the objects in the image.

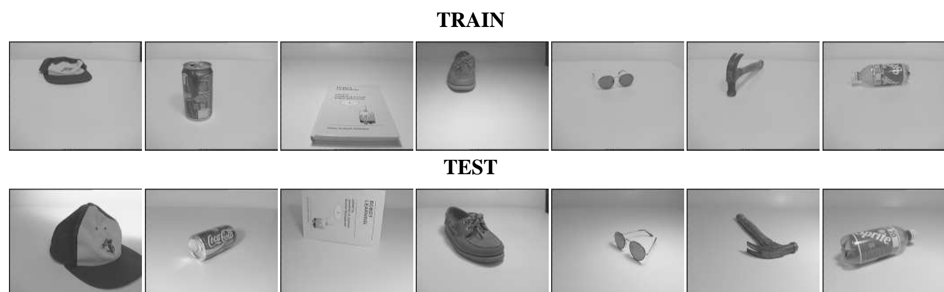


Figure 14.10
A random training and testing signal from each of the 7 classes in this classification problem.

14.4.2.1 Setting PADO up to Solve the Problem

In the experiment in this section, the total population size was 1750 (i.e., $250 * 7$). Each point on each graph is an average of at least 60 independent runs. A total of 350 (50 from each of 7 classes) images were used for training and a separate set of 350 (50 from each of 7 classes) images were withheld for testing afterwards.

The Parameterized Signal Primitives (PSPs) used in this experiment were as follows: *PSP-Point*(a_0, a_1) returns the pixel intensity at the pixel/point (a_0, a_1). *PSP-Average* (a_0, a_1, a_2, a_3) returns the *average* pixel intensity in the image region specified by the rectangle with upper left corner (a_0, a_1) and lower right corner (a_2, a_3). *PSP-Variance* (a_0, a_1, a_2, a_3) returns the *variance* of the of the pixel intensities in image region specified by the rectangle with upper left corner (a_0, a_1) and lower right corner (a_2, a_3). *PSP-Min*(a_0, a_1, a_2, a_3) returns the *lowest* pixel intensity value in the image region specified by the rectangle with upper left corner (a_0, a_1) and lower right corner (a_2, a_3). *PSP-Max*(a_0, a_1, a_2, a_3) returns the *largest* pixel intensity value in the image region specified by the rectangle with upper left corner (a_0, a_1) and lower right corner (a_2, a_3). *PSP-Diff*(a_0, a_1, a_2, a_3) returns the absolute different between the *average* pixel intensity above and below the diagonal line (a_0, a_1) to (a_2, a_3) inside the bounding rectangle with opposite corners (a_0, a_1) and (a_2, a_3). In all of these cases, if the dimension is negative (e.g., $a_2 < a_0$) the two values are interchanged.

14.4.2.2 The Results

During each run, the generalization performance on a separate set of testing images was recorded and Figure 14.11 plots the *mean* of each of these values. Figure 14.11 shows the computational effort in generations required to reach a particular level of test-set generalization performance for NP learning with and without IRNP.

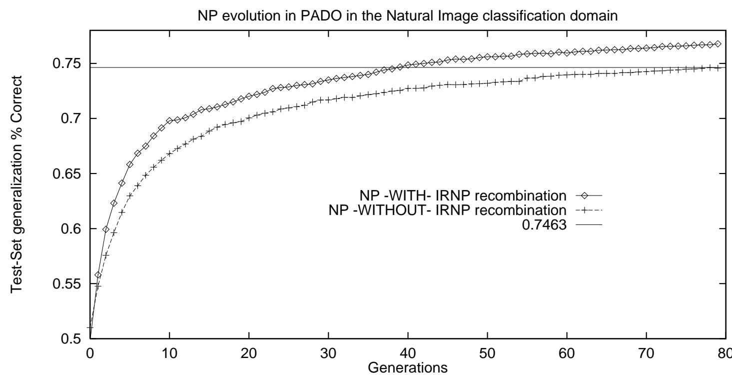


Figure 14.11
NP learning with and without IRNP (Natural Images Domain).

The most important feature of Figure 14.11 is that NP learns more than twice as fast when IRNP is applied to the recombination during evolution. That is, NP arrives at the same level of generalization performance in less than half as many generations when learning with IRNP as compared to learning without it. Also notice that NP learns quite well on this diffi cult image classifi cation problem. Random guessing in this domain would achieve only about 14.28% correct generalization performance.

It is worth noting that the performance that achieved on any domain is related to the particular orchestration strategy chosen. NP has, on this particular domain, achieved generalization performance rates as high as **86%**.

14.4.3 Acoustic Signals

The database used in this experiment contains 525 three second sound samples. These are the raw wave forms at 20K Hertz with 8 bits per sample (about 500,000 bits per sample). These sounds were taken from the SPIB ftp site at Rice University (anonymous ftp to spib.rice.edu). This database has an appealing seven way clustering (70 from each class): *the sound of a Buccanneer jet engine*, *the sound of a firing machine gun*, *the sound of an M109 tank engine*, *the sound on the floor of a car factory*, *the sound in a car production hall*, *the sound of a Volvo engine*, and *the sound of babble in an army mess hall*. There are many possible ways of subdividing this sound database; the classes chosen for these experiments are typical of the sort of distinctions that might be of use in real applications.

14.4.3.1 Setting PADO up to Solve the Problem

In the experiment in this section, the total population size was 1750 (i.e., $250 * 7$). Each point on each graph is an average of at least 55 independent runs. A total of 245 (35 from each of 7 classes) images were used for training and a separate set of 245 (35 from each of 7 classes) images were withheld for testing afterwards.

The PSPs used in this experiment were as follows: *PSP-Point(*a_0, a_1*)* returns the wave height at the moment in time specifi ed by $(a_0 * 256 + a_1)$. *PSP-Average(*a_0, a_1, a_2, a_3*)* returns the *average* wave height in the sound starting at time $(a_0 * 256 + a_1)$ and ending at time $(a_2 * 256 + a_3)$. This PSP is useless for long time intervals. *PSP-Variance(*a_0, a_1, a_2, a_3*)* returns the *variance* of the wave height in the sound starting at time $(a_0 * 256 + a_1)$ and ending at time $(a_2 * 256 + a_3)$. *PSP-Min(*a_0, a_1, a_2, a_3*)* returns the *lowest* wave height in the sound starting at time $(a_0 * 256 + a_1)$ and ending at time $(a_2 * 256 + a_3)$. *PSP-Max(*a_0, a_1, a_2, a_3*)* returns the *largest* wave height in the sound starting at time $(a_0 * 256 + a_1)$ and ending at time $(a_2 * 256 + a_3)$. *PSP-Diff(*a_0, a_1, a_2, a_3*)* is equivalent to ABS(PSP-Average(a_0, a_1, $a_{0'}$, $a_{1'}$) $-$ PSP-Average($a_{0'}$, $a_{1'}$, a_2, a_3)) where ($a_{0'}$,$a_{1'}$) is the time midpoint between (a_0, a_1) and (a_2, a_3).

Notice that, other than minor adjustments necessary to reflect the change in signal type, these parameterized signal primitives are exactly the same as the PSPs used in the visual classifi cation experiment discussed in Section 14.4.2. This was not done to demonstrate the generality of these PSPs. Quite the contrary, this similarity in the experimental procedure was done to highlight how little was done to tune NP in order to achieve the reported results. NP, using IRNP, is able to make good use of these very simple PSPs that are not well focused to solving either of the domains in which they were applied.

The fi tness used for evolutionary learning (training of the NP programs) was based upon distance from returned confi dence to the correct confi dence for each training example. Given this model of one class chosen per sound, if the NP program just guessed randomly, it could achieve an generalization performance of $1/7$ (14%) correct.

14.4.3.2 The Results

Figure 14.12 shows the generalization percent correct NP reaches on average on each generation, with and without IRNP.

Notice that in these experiments, for both orchestration strategies, IRNP learning is almost three times as effi cient as learning without it. That is, NP arrives at the same level of test set generalization performance in about one third as many generations when learning with IRNP as compared to learning without it.

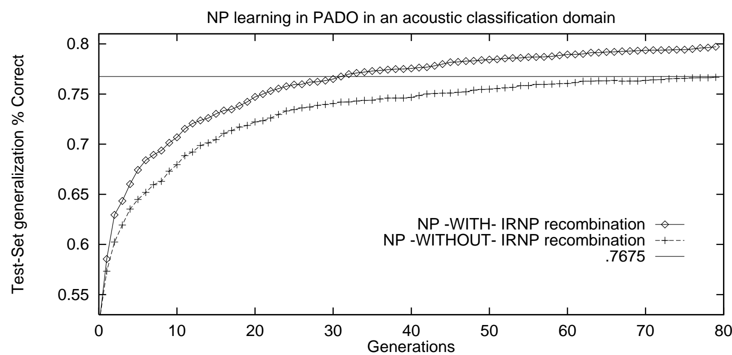


Figure 14.12
NP learning with and without IRNP (Acoustic Signals Domain).

14.4.4 Acoustic Signals Revisited

One of the most important implied aspects of this chapter is that, given more time to examine each signal the NP programs will be able to improve their evolved performance. If NP programs are really looping and foveating on the input signals, increasing the amount of time (i.e., maximum timestep threshold, T) should increase the evolved program performance. Therefore, let us revisit the acoustic signal classifi cation problem described in the previous section. As in the rest of this chapter, the experimental results in the previous section were achieved with an NP timestep threshold of 10 timesteps. In this section, we will double this value to a timestep threshold of $T = 20$ timesteps to see how that change affects both the effi ciency and, more importantly, the effectiveness of NP learning within PADO.

Since this chapter has claimed that there is an advantage to be gained from the addition of iteration and/or recursion, a demonstration that increasing the time available to each program (without increasing the number of degrees of freedom in the model being learned) will strengthen this argument. The NP programs evolving in this section have the exact same number of degrees of freedom (independently adjustable learning "parameters") as in the previous section. Programs in all ways similar to those in the previous section are simply allowed to "think longer" about the input signal. Therefore, improved performance in this experiment demonstrates that NP programs are making use of the looping/foveating aspects of the NP representation.

The domain and problem for this set of experiments is in every detail identical to the domain and problem described in the previous section.

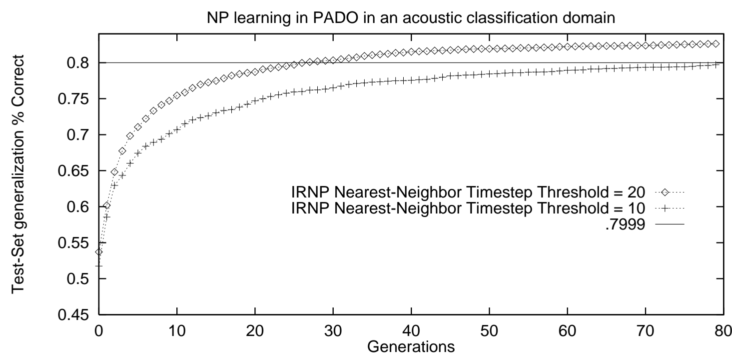


Figure 14.13
NP learning with IRNP and Timestep Threshold = 10 *and* 20 (Acoustic Signals Domain).

14.4.4.1 Setting PADO up to Solve the Problem

In setting up PADO to solve this acoustic signal classifi cation problem, every aspects was left exactly as in the previous section with a single exception. This exception was that the timestep threshold (that maximum number of timesteps after which the response is extracted from each NP program) was increased from 10 to 20.

14.4.4.2 The Results

Each point in Figure 14.13 is an average over at least 60 independent trials. Notice that it takes more time to compute the fi tness of each particular program on each particular signal, the same amount of learning is done with two different timestep thresholds. Said in another way, the additional computation time is spent because the fi tness takes twice as long to measure, not because twice as many decisions are made with the same information. This is signifi cant by itself, but more signifi cant still when we remember that learning to take advantage of this additional time available to each NP program must be done using the same amount of learning (i.e., the same number of search steps). This means, that in some sense, this test would have been more fair if more learning (and therefore more computation time) had been given for the experiments in this section, not less computation time as the computation time note in the previous paragraph seems to suggest.

Figure 14.13 shows the generalization percent correct PADO reaches on average on each generation with IRNP using this enlarged timestep threshold. The results of this experiment are quite exciting. IRNP learning with a timestep threshold of $T = 20$ is three times as effi cient as learning under the same conditions with $T = 10$. Notice that this means that IRNP learning with $T = 20$ actually accomplished the same amount of learning as do NP learning without IRNP using $T = 10$ using only about 13% of the effort.

14.5 Related Work

The main areas of work related to the topic of this chapter are: algorithm evolution applied to signal understanding, the learning of recurrent networks, and particularly, other attempts to improve the effectiveness of the search operators in GP.

In the area of evolution applied to signal understanding past work has included bitmap understanding (e.g., [Andre, 1994; Koza, 1994]), signal understanding aids (e.g., [Nguyen and Huang, 1994; Tackett, 1993; Daida, 1996; Poli, 1996b]), time series prediction (e.g, [Angeline, 1996a]), and speech discrimination (e.g., [Conrads et al., 1998; Nordin and Banzhaf, 1996].

In an important related area, considerable work has been done on the complex memory structure and use (e.g., [Langdon, 1995; Andre, 1995; Andre and Teller, 1996; Brave, 1996a; Langdon, 1996; Spector and Luke, 1996]). Some research has been done on recursion and looping in GP (e.g., [Kinnear, Jr., 1993; Brave, 1996b; Langdon, 1995]), but how to tractably evolve complex programs with these elements is still an open question.

Both because of its representational similarities and because of its computational class equivalence (i.e., both are Turing complete representations), recurrent ANNs (e.g., [Rumelhart et al., 1986]) are also of relevance to the NP and IRNP research. In ANNs, the focus on improving the power of the technique has not been on changing what is inside an "artificial neuron." Works like [Dellaert and Beer., 1994; Sharman et al., 1995] have, however, investigated the possible additional benefit of complicating and un-homogenizing artificial neurons. Though its similarity to NP is in representation, not in use or objectives (i.e., IRNP), [Poli, 1996a] is an interesting example of the evolution of graph structured programs as is [Angeline, 1997]. For a survey on data flow machine, see [Treleaven et al., 1982].

One of the best descriptions of and attacks on the lack of a clear, locally optimal update procedure is [O'Reilly, 1995]. In her thesis, O'Reilly gives good evidence for this as an important flaw in the GP paradigm and introduces a locally optimal hill-climbing variant as a recombination element within GP. [Olsson, 1995] uses a form of iterative deepening done on minimum description length codes with ML-specific program transformations. [Angeline, 1996b] and [Fogel et al., 1995] describe possible approaches for allowing the mechanism of evolution to provide self-adaptation all the way down to the single node level. Another take on guided crossover can be seen in [Langdon, 1996].

The bucket-brigade algorithm is one of the oldest versions of credit assignment discussed as an explicit mechanism by Holland [Holland, 1975] or as an implicit mechanism in works such as [Wilson, 1987]. The variant of a *profit-sharing plan* was introduced in [Holland and Reitman, 1978]. The bucket-brigade algorithm is just a special case of the general temporal difference methods (TDM) [Sutton, 1988] like Q-learning [Watkins, 1989].

14.6 Conclusions

This research has contributed a new representation for learning complex programs. This new connectionist program language, *Neural Programming*, has been developed with the goal of enabling a principled update policy for algorithm evolution called *Internal Reinforcement*. This is the first such principled update policy created for the field of genetic programming. Neural Programming enables the construction of a *Credit-Blame map* for each evolving program. Sensitivity-based bucket-brigade for refining each program's Credit-Blame map leads to a credit assignment of sufficient detail to allow internal reinforcement to perform focused, beneficial search operations during the algorithm evolution. We illustrated these techniques with experiments that showed that internal reinforcement improves the speed and accuracy of Neural Programming learning. These same experiments also demonstrated that Neural Programming can successfully learn to correctly classify large signals from many classes in real world domains.

The goal of this chapter has been to communicate the exciting result that, through the exploration of new program representations, we have captured the explanation and principled update power of explicit credit-assignment with the flexibility and generality of genetic programming.

Acknowledgements

This work has been funded through the generosity of the Fannie and John Hertz Foundation.

Bibliography

Altenberg, L. (1994), "The evolution of evolvability in genetic programming," in *Advances In Genetic Programming*, K. E. Kinnear, Jr. (Ed.), pp 47–74, MIT Press.

Andre, D. (1994), "Automatically defined features: The simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them," in *Advances In Genetic Programming*, K. E. Kinnear, Jr. (Ed.), pp 477–494, MIT Press.

Andre, D. (1995), "The evolution of agents that build mental models and create simple plans using genetic programming," in *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, L. Eshelman (Ed.), pp 248–255, Pittsburgh, PA, USA: Morgan Kaufmann.

Andre, D. and Teller, A. (1996), "A study in program response and the negative effects of introns in genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 12–19, Stanford University, CA, USA: MIT Press.

Angeline, P. J. (1993), *Evolutionary Algorithms and Emergent Intelligence*, PhD thesis, Ohio State University, Computer Science Department.

Angeline, P. J. (1996a), "An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 21–29, Stanford University, CA, USA: MIT Press.

Angeline, P. J. (1996b), "Two self-adaptive crossover operators for genetic programming," in *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear, Jr. (Eds.), MIT Press.

Angeline, P. J. (1997), "An alternative to indexed memory for evolving programs with explicit state representations," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), pp 423–430, Stanford University, CA, USA: Morgan Kaufmann.

Brave, S. (1996a), "The evolution of memory and mental models using genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 261–266, Stanford University, CA, USA: MIT Press.

Brave, S. (1996b), "Using genetic programming to evolve recursive programs for tree search," in *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear, Jr. (Eds.), Chapter 10, Cambridge, MA, USA: MIT Press.

Conrads, M., Nordin, P., and Banzhaf, W. (1998), "Speech sound discrimination with genetic programming," in *Proceedings of the First European Workshop on Genetic Programming*, W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty (Eds.), LNCS 1391, Paris: Springer-Verlag.

Daida, J. (1996), "Algorithm discovery using the genetic programming paradigm: Extracting low-contrast curvilinear features from SAR images of arctic ice," in *Advances in Genetic Programming 2* P. J. Angeline and K. E. Kinnear, Jr. (Eds.), Chapter 21, Cambridge, MA, USA: MIT Press.

Dellaert, F. and Beer., R. (1994), "Co-evolving body and brain in autonomous agents using a developmental model," in *Technical Report CES-94-16, Department of Computer Engineering and Science*, Case Western Reserve University, Cleveland, OH 44106.

Fogel, L., Angeline, P. J., and Fogel, D. (1995), "An evolutionary programming approach to self-adaptation on finite state machines," in *Proceedings of the 4th Annual Conference on Evolutionary Programming*, J. McDonnell, R. Reynolds, and D. Fogel (Eds.), MIT Press.

Holland, J. (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press.

Holland, J. and Reitman, J. S. (1978), "Cognitive systems based on adaptive algorithms," in *Pattern Directed Inference Systems*, Academic Press.

Hopcroft, J. and Ullman, J. (1979), *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley.

Kinnear, Jr., K. E. (1993), "Generality and difficulty in genetic programming: Evolving a sort," in *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, S. Forrest (Ed.), pp 287–294, Morgan Kaufmann.

Koza, J. (1994), *Genetic Programming 2*, MIT Press.

Langdon, W. B. (1995), "Evolving data structures with genetic programming," in *Proceedings of the Sixth International Conference on Genetic Algorithms*, S. Forrest (Ed.), Pittsburgh, PA, USA: Morgan Kauffman.

Langdon, W. B. (1996), "Data structures and genetic programming," in *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear, Jr. (Eds.), MIT Press.

Nguyen, T. and Huang, T. (1994), "Evolvable 3d modeling for model-based object recognition systems," in *Advances In Genetic Programming*, K. E. Kinnear, Jr. (Ed.), MIT Press.

Nordin, P. (1997), *Evolutionary Program Induction of Binary Machine Code and its Applications*, PhD thesis, der Universitat Dortmund am Fachereich Informatik.

Nordin, P. and Banzhaf, W. (1996), "Programmatic compression of images and sound," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 345–350, Stanford University, CA, USA: MIT Press.

Olsson, R. (1995), *Inductive Functional Programming using Incremental Program Transformation*, PhD thesis, University of Oslo.

O'Reilly, U.-M. (1995), *An Analysis of Genetic Programming*, PhD thesis, Carelton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada.

Poli, R. (1996a), "Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming," in *Technical Report CSRP-96-14, School of Computer Science, University of Birmingham*, University of Birmingham.

Poli, R. (1996b), "Genetic programming for image analysis," in *Genetic Programming 1996*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 363–368, Stanford University, CA, USA: MIT Press.

Rumelhart, D., Hinton, G., and Williams, R. (1986), "Learning internal represenations by error propogation," in *Parallel Distributed Processing*, Cambridge, MA, USA: MIT Press.

Sharman, K., Alcazar, A., and Li, Y. (1995), "Evolving signal processing algorithms by genetic programming," in *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*.

Spector, L. and Luke, S. (1996), "Culture enhances the evolvability of cognition," in *Cognitive Science (CogSci) 1996 Conference Proceedings*.

Sutton, R. (1988), "Learning to predict by the methods of temporal differences," in *Proceedings of the International Conference on Machine Learning*, AAAI Press.

Tackett, W. A. (1993), "Genetic programming for feature discovery and image discrimination," in *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest (Ed.), Morgan Kauffman.

Teller, A. (1994), "The evolution of mental models," in *Advances In Genetic Programming*, K. E. Kinnear, Jr. (Ed.), pp 199–220, MIT Press.

Teller, A. (1996), "Evolving programmers: The co-evolution of intelligent recombination operators," in *Advances in Genetic Programming 2*, K. E. Kinnear, Jr. and P. J. Angeline (Eds.), MIT Press.

Teller, A. (1998), *Algorithm Evolution with Internal Reinforcement for Signal Understanding*, PhD thesis, Computer Science Department, Carnegie Mellon University.

Teller, A. and Veloso, M. (1995), "Program evolution for data mining," in *The International Journal of Expert Systems. Third Quarter. Special Issue on Genetic Algorithms and Knowledge Bases.*, S. Louis (Ed.), pp 216–236, JAI Press.

Teller, A. and Veloso, M. (1997), "PADO: A new learning architecture for object recognition," in *Symbolic Visual Learning*, K. Ikeuchi and M. Veloso (Eds.), Oxford University Press.

Thrun, S. and Mitchell, T. (1994), "Learning one more thing," Technical Report CMU-CS-94-184, Computer Science Department, Carnegie Mellon Unversity.

Treleaven, P. C., Brownbridge, D. R., and Hopkins, R. P. (1982), "Data-driven and demand-driven computer architecture," *ACM Computing Surveys*, 14(1):93–143.

Watkins, C. J. (1989), *Learning from Delayed Rewards.*, PhD thesis, King's College.

Wilson, S. (1987), "Hierarchical credit allocation in a classifi er system," in *Genetic Algorithms and Simulated Annealing*, Morgan Kaufman Publishers.