Algorithm Evolution with Internal Reinforcement for Signal Understanding

Astro Teller

December 5, 1998

CMU-CS-98-132

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Manuela M. Veloso, Chair
Tom Mitchell
Katsushi Ikeuchi (University of Tokyo)
Rodney Brooks (MIT)

The work has been supported through the generosity of the Fannie and John Hertz Foundation. This research is also sponsored in part by the Department of the Navy, Office of Naval Research under contract number N00014-95-1-0591. Views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of the Fannie and John Hertz Foundation, Department of the Navy, Office of Naval Research or the United States Government.

Acknowledgements

Without the support and guidance of my advisor, Manuela Veloso, this dissertation would never have happened. Financially, the Fannie and John Hertz Foundation generously supported me through most of my graduate work for which I am deeply indebted to them. Emotionally, my wife Zoe has been my lighthouse and she is certainly a necessary if not sufficient condition for the existence of this document. I have benefited from the advice and ideas of many more people than I could list on this page. Among that group however, John Koza, David Andre, and Peter Stone stand out in my mind as people who have, over a number of years, been instrumental in helping me shape the direction of my research. In addition, David Andre, Sean Luke, and Bill Langdon went out of their way to help me finish this thesis.

Contents

Abstract

Automated program evolution has existed in some form for almost forty years. Signal understanding (e.g., signal classification) has been a scientific concern for longer than that. Generating a general machine learning signal understanding system has more recently attracted considerable research interest. First, this thesis defines and creates a general machine learning approach for signal understanding independent of the signal's type and size. This is accomplished through an evolutionary strategy of signal understanding programs that is an extension of genetic programming. Second, this thesis introduces a suite of sub-mechanisms that increase the power of genetic programming and contribute to the understanding of the learning technique developed.

The central algorithmic innovation of this thesis is the process by which a novel principled credit-blame assignment is introduced and incorporated into the evolution of algorithms, thus improving the 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 collectively called *internal reinforcement in neural programming*. This thesis concentrates on these algorithmic innovations in real world signal domains where the signals are typically large and/or poorly understood.

This evolutionary learning of algorithms takes place in *PADO*, a system developed in this thesis for "parallel algorithm discovery and orchestration" and as a demonstrably effective strategy for divide-and-conquer in signal classification domains. This thesis includes an extensive empirical evaluation of the techniques developed in a rich variety of real-world signals. The results obtained demonstrate, among other things, the effectiveness of principled credit-blame assignment in algorithm evolution.

This work is unique in three aspects. No other currently existing system can learn to classify or otherwise "symbolize" signals with no space or size penalties for the signal's size or type. No other system based on genetic programming currently exists that purposefully generates and orchestrates a variety of experts along problem specific lines. And, most centrally, the thesis introduces the first analytically sound mechanism for explaining and reinforcing specific parts of an evolving program.

The goal of this thesis is to argue, explain, and demonstrate how representation and search are intimately connected in evolutionary computation and to address these dual concerns in the context of the evolution of Turing complete programs. Ideally, this thesis will inspire future research in this same area and along similar lines.

Chapter 1

Introduction

1.1 Motivation

The demand for robust AI systems with few constraints coupled with the rising cost of programmer time relative to computer cycles, has pushed the fields of signal understanding and machine learning together. Because human researchers cannot meet (i.e., program by hand) the perception demands of the continuing explosion of specialized and general applications, the signal-to-symbol problem has become central to artificial intelligence in general, and to machine learning in particular. As an example, [Ikeuchi and Veloso, 1997] highlights this problem and describes a variety of innovative research programs in the specific area of computer vision.

The signal-to-symbol problem is the general task of labelling a perceived signal with one or more symbols (e.g., identifying the primary object present in a video image). In other words, a solution to this problem takes a large number of inputs (the raw signal) and transforms it into a smaller number of outputs (the symbols). These symbols are intended to capture some useful aspects of the input. The symbols will, in turn, permit higher level reasoning based on the perceived signals.

Because machine learning strives for increasing autonomy, much of the work done in machine learning today has, as an eventual goal, a general system for producing such a signal-to-symbol mapper that works well for a wide variety of domains. Machine learning techniques exist today that attempt to accomplish this goal (see [Mitchell, 1997] for a comprehensive survey). Artificial Neural Networks (ANNs) [Rumelhart *et al.*, 1986] and Decision Trees [Quinlan, 1986] are just two examples. These techniques have advantages that include theories about convergence and expressiveness under certain assumptions. Because of these particular positive attributes, extending these systems to very general domains becomes complicated. For example, it would be hard to argue that introducing memory access and time into ANNs is a simple matter (e.g., [Hild and Waibel, 1993]). Getting decision trees to handle highly non-linear functions, especially of continuous values, requires considerable unnatural contortion of the original technique (e.g., [Squires and Sammut, 1995]). The original motivation for this thesis was the feeling that, given the slow progress of existing techniques in the general signal-to-symbol domain, perhaps a significantly different direction would be worth exploring.

1.1.1 Motivation for Using Evolutionary Computation

Evolutionary computation (EC) is any algorithm that uses the concepts of selection pressure, fitness proportionate reproduction, or sexual reproduction. EC has been an area of active research since the 1960s (e.g., [Fogel *et al.*, 1966]). The last few years have seen a tremendous surge in interest in a variety of evolutionary computation schemes including genetic algorithms (GA), genetic programming (GP), evolutionary strategies (ES), and evolutionary programming (EP). In the movement out of *good old-fashioned artificial intelligence* and towards *nouvelle* AI, evolution has taken on an increasingly central role as a source of inspiration (e.g., [Brooks, 1986, Brooks, 1990, Brooks, 1997]). In addition, the model of evolution as an appropriate model for mental learning has gained favor in philosophy and cognitive science recently (e.g., [Dennett, 1992]). Because of these trends, because of the broad range of successes GP has had in recent years (e.g., [Andre *et al.*, 1996, Koza *et al.*, 1996, Nordin, 1997a]), and because of the expressiveness and flexibility of evolving algorithms, the general paradigm of evolutionary computation was selected as the basis of research for this thesis that investigates its potential as a machine learning technique in the signal understanding arena.

1.1.2 Where Have All the Good Programs Gone?

The previous section motivated the use of evolutionary computation as the underlying mechanism for working on the signal-to-symbol problem. There is, however, a fundamental problem with evolutionary computation, and particularly with genetic programming, as it is currently practiced. The problem is that in the space of functions, even if it has been carefully defined so that most or all examined functions are legal, the density of functions that do something "interesting" is very low.[1] 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. Because PADO, the learning approach presented in this thesis, has been committed to the evolution of these more complicated algorithms, it must address this problem.

This low density of programs "worth searching," combined with the random recombination that characterizes genetic programming, seems to have marginalized GP, an exciting and valuable subfield of machine learning. The following quotes from the newest, most complete introduction to Genetic Programming [Banzhaf *et al.*, 1998], (written, obviously, by pro-GP researchers) highlight this inconsistency that still exists in the paradigm. Basically, the authors of this book acknowledge both that GP needs search operators that tend to focus on good solutions *and* that GP search operators are currently not focused, but instead are guided by random transformations.

> *"It should be obvious that a good machine learning system would use search operators which take a path through solution spaces that tend to encounter good solutions and to bypass bad ones."*

[1] 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]).

> *"In GP, the primary transformation operators are 'crossover' and 'mutation.' Mutation works by changing one program; crossover by changing two (or more) programs by combining them in some manner. Both are, to a large extent, controlled by pseudo-random number generators."*
>
> – From *Genetic Programming: An Introduction*, 1998.

This thesis develops a novel solution to this disparity between what GP is and what GP wants to become. As part of the evolutionary process, the thesis introduces a method of program transformations that is principled, based on the program's behavior, and significantly more likely to identify new programs that are worth searching than random local sampling. The single most notable contribution of this thesis 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 thesis is the process by which principled credit-blame assignment can be brought to evolution of algorithm 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* and applied through a set of principled processes collectively known as *internal reinforcement in neural programming.* This *internal reinforcement* of evolving programs is presented in this thesis 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-blame assignment allows the dynamics of the system to implicitly (indirectly) determine credit and blame. Evolution does just this [Altenberg, 1994]. This approach could be typified as reinforcement of the type "This program is better than 82.2% of the other programs in the evolving population."

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. An example of this type of credit assignment and reinforcement is "weight i in level j of this ANN should be 5% larger."

To be clear, this work on *internal reinforcement* is not just an attempt to enable gradient descent in program space. Internal Reinforcement is 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.*

1.1.3 The Questions This Thesis Answers

In summary, the two main questions that this thesis has used as guiding lights are:

- **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?**

- **Can the evolutionary computation paradigm be extended, and how far, to apply successfully as a machine learning technique to the general signal-to-symbol problem?**

1.2 Objectives and Approach

The objectives of this thesis arise from a combination of three factors: the desire to see a more general solution to the machine learning problem of finding mappings between signal and symbol sets; the conviction that new approaches (in particular, ones using evolutionary computation) are worth investigating; and the clear and problematic lack of explicit credit-blame assignment or credit-blame use in genetic programming.

Existing machine learning techniques have some advantages and disadvantages with respect to finding solutions to the general signal-to-symbol problem. Concretely, this thesis work has attempted to overcome some of these disadvantages without losing any of the important advantages of existing systems.

Disadvantages of existing machine learning techniques for signal understanding include a number of factors. First, the input must almost always be preprocessed. This thesis creates a machine learning technique in which signal preprocessing is unnecessary. Second, domain knowledge must be input in the form of preprocessing or of technical details of the machine learning strategy that are not obvious to a signal expert. This thesis creates a machine learning technique in which domain knowledge can be given to the learning system in a way that does not require intimate knowledge of the learning technique.

Advantages of existing machine learning techniques for signal understanding include a number of factors. First, "real-world" signals can be handled. This is a crucial aspect which the learning paradigm of this thesis also provides. Second, even when learning must be done off line, the learned function can be run in real time. Moving to the realm of learned (evolved) algorithms does add additional time to its training procedure, but this thesis maintains representations that generate algorithms that can be run in real time. Third, why the popular machine learning techniques work is well understood, thereby generating faith in those methods. This thesis, in providing a credit-blame assignment approach to the learning of models along with a more principled search mechanism using that credit-blame assignment, creates the groundwork for the same understanding demonstrating the effective use of genetic programming as a machine learning approach.

This thesis demonstrates that a learning system based on evolutionary computation can be built that takes signals as input and produces symbols as output. More specifically, this thesis demonstrates that such a system will involve the coordination of evolved programs whose basic knowledge of the signals comes, not from preprocessing, but from parameterization of domain knowledge in the form of *signal primitives*. More importantly, this learning takes places with no prior knowledge of the signal type or size and with no constraints on the symbol type to be returned by the learned algorithms. The learning, through evolution in this thesis, is driven by a principled update/search procedure in the context of a program representation that facilitates the addition of principled search heuristics into genetic programming.

A distinction is being made in these objectives between the primary work of the thesis, *efficient evolution of complex algorithms*, and the area of application on which this thesis work is focused, *signal understanding*.

There are a number of related fields that all, to some extent, can be thought of as addressing the signal understanding problem: Computer Vision, Machine Learning, Digital Signal Processing, Pattern Recognition, etc. Most generally, the problem of signal understanding is the *signal-to-symbol* mapping problem. That is:

> **Given a "signal" type and a notion of "symbols" that represent important characteristics of signals of that type, how can we create a process for automatically extracting these symbols from the signals they represent?**

For example, given a video image (e.g., 1000x1000 pixels where each pixel is a 32-bit color value), how can we extract pieces of information about the signal such as "That is a picture of a cow!" as shown in Figure 1.1. This sort of question is the motivation for the area of application of this thesis.

$\rightarrow$ "Cow"

Figure 1.1: The signal-to-symbol problem. This example is a video image to text translation.

This thesis consists of two main components. The first involves developing a machine learning mechanism with the following properties. This approach, PADO, is a supervised machine learning technique for signal understanding that employs evolutionary computation. The inputs to the system are a set of labeled example signals and a set of functions for examining those signals. The output of the system is a learned signal classifier.

The second objective and component of this thesis is the theoretical and practical exploration of *internal reinforcement*, the principled update procedure for the evolution of programs, and *neural programming*, the program language representation that enables internal reinforcement.

1.3 Thesis Contributions

This thesis contributes a machine learning mechanism that satisfies the objectives above. The several specific elements of this machine learning mechanism are technical and scientific contributions to the fields of algorithm evolution and signal understanding. The following is a list of those fields and subfields and the specific contributions to them.

- Contributions to the field of Algorithm Evolution
  - <u>Contributions to EC in Representations</u>
    Neural Programming, a general graph data-flow language for program evolution

    Representations and recombination operators are intimately linked and this representation has been created to facilitate internal reinforcement.
  - <u>Contributions to EC in Input</u>
    Parameterized Signal Primitives

    The evolving programs need some access to the signals. This access is given to PADO by the user in the form of parameterized subroutines. This means that domain knowledge can be given to the evolving programs in a simple programmatic form.
  - <u>Contributions to EC in Recombination</u>
    Internal Reinforcement using a Credit-Blame map

    Other machine learning techniques (e.g., ANNs) have the advantage that they can reinforce specific parts of the function being created because the technique allows for introspection and assignment of credit and blame to parts of the function. This is good not only because it improves learning, but also because it helps researchers to understand why and how the technique works. An *internal reinforcement* policy for algorithm evolution is shown to be possible in this thesis. This internal reinforcement policy is used to improve learning for the evolving programs.
  - <u>Contributions to EC in Results Determination</u>
    Evolution and Orchestration of multiple evolved specialist experts

    On the highest conceptual level, the PADO mechanism learns algorithms for signal discrimination using an evolutionary computation framework. These algorithms are trained by PADO to learn along "specialist" lines. Then PADO orchestrates the best of these learned programs into a complete signal understanding system.

- Contributions to the field of Learned Signal Understanding
  - **Arbitrary signal size and type**

    The mechanism (PADO) creates a system that takes signals as input and returns one or more symbols as output, independent of the signal type and size. By "independent of the signal type and size" we mean exactly that the *number of degrees of freedom in the model (program) being learned need not change* when the signal type or size are altered. The mechanism is applicable to large, unprocessed signals. This aspect of "realism" is important for this thesis since robust learned signal understanding systems need to handle real world signals without explicit preprocessing.

- **Expert knowledge input at the symbolic (programmatic) level**
  Parameterized Signal Primitives.

The final achievements of this research are two fold. First, this thesis communicates the exciting result that through the exploration of new program representations we have captured the explanation and principled update power of explicit credit-blame assignment with the flexibility and generality of genetic programming. Second, this thesis produces a paradigm that provides real generality in the signal-to-symbol problem while remaining tractable.

1.4 A Reader's Guide to the Thesis

Chapter 2 provides an introduction to evolutionary computation for those who are unfamiliar with the basic terminology and methods. It can safely be skipped by those who already have a background in the fields of genetic programming, evolutionary strategies, genetic algorithms, and evolutionary programming.

Chapter 3 describes PADO, the general signal-to-symbol mapper that is itself one of the contributions of this thesis and is the context within which the other contributions are described. The learned programs inside PADO are each expected to learn to discriminate one signal type from all others in a set of labeled signal training examples. Detail is given to the version of PADO used as the specific context for experiments throughout this thesis.

Chapter 4 focuses on the orchestration phase of PADO.

Chapter 5 introduces and describes in detail the current program representation of PADO, *neural programming*. This representation is the basis for the evolution of programs that facilitates the creation of a *Credit-Blame map* and makes internal reinforcement possible.

Chapter 6 presents *internal reinforcement*, the thesis mechanism that provides some of the benefits and characteristics of gradient descent in program space.

Chapter 7 demonstrates a number of the claims of this thesis through an extensive empirical evaluation using a series of illustrative, real-world signal classification tasks as well as a simple, focused, manufactured generic signal domain.

Chapter 8 contains a survey of the closely related, synergistic work to the work of this thesis. In addition, it reports on a number of alternative methods that have been or are currently being examined for solving some of the general problems addressed in this thesis.

Chapter 9 concludes the thesis and discusses a number of exciting specific directions of future work in this area. There are a number of appendices that have additional information on implementation details, example programs, statistical significance of results, Turing completeness discussion, and background detail on some of the parameterized signal primitives used in the thesis experiments. Of particular important is Appendix A that contains a complete, annotated list of the mathematical notation used in this thesis.

A few additional remarks on the structure of the thesis will help to frame further discussions. This thesis builds upon the underlying assumption that understanding how to evolve Turing complete algorithms for complex tasks is an important problem.

It could be argued that *neural programming* and *internal reinforcement* are solving a problem that does not exist: "how can gradient descent in program space be accomplished, particularly in a Turing complete[2] program space?" Quite the contrary, I feel that solving this problem is important exactly because it *does not* come up in evolutionary computation with regularity. The reason Turing complete programs are almost never evolved is because a number of critical issues must be addressed before this evolution can happen effectively. Once we can begin to evolve Turing complete programs, it may quickly become clear that that is the critical aspect for EC to address. Appendix B explains the complications that evolving Turing complete algorithms brings to the field of evolutionary computation and has an important note about the actual computational expressiveness of the programs evolved in this thesis.

This thesis attempts to address this problem, not to justify this problem's existence. In particular, no detailed comparisons will be made between PADO and standard Tree-GP as supporting evidence that PADO is addressing a problem that exists in GP as it is currently practiced. Similarly, comparison with other ML techniques are not performed because it does not serve the purposes of this thesis. These comparisons would invite criticism about the fairness of the comparisons when such comparisons are beside the point; this is not a "my system outperforms your system" thesis.

The goal of this thesis is to argue, explain, and demonstrate how representation and search are intimately connected in evolutionary computation and to address these dual concerns in the context of the evolution of Turing complete programs. Ideally, this thesis will inspire future research in this same area and along similar lines.

[2] For a primer on Turing completeness, see [Hopcroft and Ullman, 1979].

Chapter 2

A Brief Overview of Algorithm Evolution

The title of this thesis is "Algorithm Evolution with Internal Reinforcement for Signal Understanding." Before entering into the details of the thesis, this chapter will give an introduction to and overview of the general field of *Algorithm Evolution*. Since this chapter has been designed as a general introduction, it is safe to skip to the next chapter if such an overview is unnecessary for the reader.

2.1 Evolutionary Computation as Machine Learning

Darwin's theory of evolution has been a point of inspiration and reference for the development of simulated evolutionary computation for learning complex models. The original and most basic form of this motivation is, "Evolution created complex behavior in the form of homo-sapiens, so perhaps evolution can be used to create other kinds of complex behavior." The details of biology and how it affects evolution have informed much of the work done in the field of Evolutionary Computation (EC).

Search is embedded in most AI algorithms. In particular, machine learning can almost always be described as a form of search in which there is a space of models to select from and the problem is to design an algorithm that finds the more desirable models while spending a minimal amount of time examining models from the space. These models take many forms in machine learning, from the weights in an ANN to the splits in a decision tree to a vector of real numbers that optimize some target function. Because search is the basic language of AI, evolution will be explained in this chapter in terms of search. Notice that simulated evolution is a form of *machine learning*.

This thesis refers to a number of terms taken from biology that are now commonly used in the field of EC. These terms are generally some mechanism in biology that inspired the EC practice, but the isomorphism between the two may not go below the surface similarity. Whenever possible, this thesis will attempt to clarify potential confusions arising from these differences.

The description of a living thing is written in a series of *chromosomes*. Most multi-celled living things have many chromosomes in their DNA (RNA in the cases of some organisms), but in EC the genetic code for a thing to be evolved is simply called its "chromosome" (singular). Each chromosome is composed of many *genes*. The different possible states of these genes, in biology, are referred to as the *alleles*. The field of EC has collapsed this distinction and simply describes the chromosome under evolution as comprised of many alleles.

There is an important distinction between the genetic code for an organism and the instantiation of that genetic code as the organism itself. Though both are evolved simultaneously and are inexorably linked, there may (or may not) be important reasons for this distinction to exist in EC[1]. In biology, the encoding of an organism (the chromosomes) is referred to as the *genotype* of that organism (e.g., Astro's DNA). The physical realization of the organism is referred to as the *phenotype* of that organism (e.g., Astro). These same terms (*genotype* and *phenotype*) are also used in EC to refer to the description of the thing, and the thing itself. As will become clear in the following section, this can be confusing in EC because the genotype and the phenotype are identical in many cases (e.g., standard Genetic Programming).

In biology, the *fitness* of an organism is the ability of that individual to live long enough to produce genetically viable children. In EC the fitness of an evolving individual is generally the ability of the phenotype of that individual to meet specifications set by that specific EC system. Examples of such *fitness measures* will be given through this thesis.

Table 2.1 summarizes the basic process of evolutionary computation.

1. INITIALIZE: *Select an initial population of M individuals for generation 0.* Usually these individuals are generated randomly.
2. EVALUATE: *Calculate G_p, fitness of population member p.* This fitness can be a quick approximation to a more accurate fitness.
3. REPRODUCE: *M times, pick a population member and put it in the* mating pool. This selection is always biased towards high fitness individuals.
4. RECOMBINE: *Perform search operations on members of the mating pool.* These operations (*recombination operators*) are most commonly *mutation* and *crossover*.
5. Call this new mating pool the "new population" and go to step 2.

Table 2.1: Outline for the search aspect of Machine Learning using Evolutionary Computation.

The process of EC is a form of the search process called "Beam Search." This relationship is well explained in [Tackett, 1994].

[1]This distinction is actually a continuing debate in the field of EC. Appendix G describes the place of this thesis in that debate.

2.1.1 Details about the EC cycle

Step 1: INITIALIZE

There are three basic options when beginning an EC run.[2] The first is to generate the initial population members randomly. This can be done in a variety of ways and is always specific at least to the syntax in which the population members are written (i.e., the genotype syntax for the chromosomes to be evolved). This thesis will describe in detail how this process is carried out for a number of different representations that PADO has used.

The second option is to initialize the population to be evolved with non-random individuals. These individuals can either have been learned in a previous learning process or can have been designed by hand (though in that case some element of randomness in varying these hand-designed individuals is usually used to create a genetically diverse population). The third option is simply to use an initial population composed largely of random individuals with one or a few non-random individuals included. This process is called *seeding* the population in the field of EC.

Step 2: EVALUATE

The process of measuring the fitness of each individual in a population is complex and the subject for much discussion on a number of different dimensions. For the purposes of this overview, the most effective communication of this variety will simply be to give a number of examples.

Suppose that we plan to evolve a good airplane wing. To do this we design a way of encoding many of the important aspects of standard airplane wings as parameters. This list of parameters will be the genotype of the airplane wing. With a specific list of parameters, we could take this list and create a physical realization of the wing (e.g., out of plastic) or a virtual realization of the wing (e.g., a CAD model). This realization is the phenotype of the evolving structure. Imagine that we are trying to discover a wing design that has a minimal drag to lift coefficient. If we have a virtual wind-tunnel, the process for determining the fitness of chromosome p in the current population could work as follows:

- Take chromosome p and create the phenotype it encodes (a CAD model of the wing).
- Insert this virtual wing into the virtual wind-tunnel and measure the drag α and lift β created by this wing under the air conditions of interest.
- Assign the fitness to be $G_p = \frac{\alpha}{\beta}$

In this example, the airplane wings under evolution are said to be *encoded* by the bitstrings that represent them. Let's now try another example that bears more directly on the subject of this thesis: evolution of algorithms. Suppose that we are unaware of the Taylor series expansion for $f(x) = e^x$ but that we would like an approximation to e^x using the basic arithmetic operators $(+,-,*,/)$ plus the operands x and constants. We can define individuals

[2] A "run" is a full simulation that takes an initial set of population members through a series of evolutionary steps to some termination point determined by the particular system.

in the population as functions (the simplest form of algorithms). These functions can be simple, like $f(x)$ = (+ x 5), or complex, like $f(x)$ = (* (- (* (+ x 5) (/ x 2)) (* (* (+ 5 3) 2) x)) (* x (/ x 18))). Now we can define the fitness of a chromosome p in the current population as follows:

- Take chromosome p (which we will call Funct_p) and create the phenotype. Notice that, unlike the airplane wing example, this step is a NOP since the phenotype and the genotype are the same.
- Pick 200 random values for x inside the range of interest (e.g., [-100,100])
- For value x_j $(1 \leq j \leq 200)$, define the error for this x value as $E_j = (e^{x_j} - \text{Funct}_p(x_j))^2$
- Compute the fitness of chromosome p as, $G_p = \frac{1}{200}\sum_{j=1}^{200} E_j$

This particular experiment has been performed and the first several terms of the Taylor's series were successfully evolved. (See [Koza, 1992] for details).

Notice that in this example, the fitness of each chromosome is an approximation to the chromosome's "real fitness"[3] and that the fitness of chromosome p will be different if tested more than once. This is not a problem for evolution and, because approximations are usually cheaper to calculate than exact fitnesses, this is the rule rather than the exception in the field of EC.

The single most common mechanism for determining the fitness value of each program is shown in Table 2.2. In machine learning, this process of training with a set of "labeled" examples is called *supervised learning*. The referenced experiment for approximating e^x is an example of such a fitness procedure. PADO, NP, and IRNP all exist within the context of supervised learning.

For each program p in the evolving population
For each training input S_i $\quad(1 \leq i \leq \lvert S\rvert)$
Call L_i the correct/desired program response (the signal label)
Run program p on input S_i and get its response, R_p^i
$G_p = \sum_{i=1}^{\lvert S\rvert} \frac{\mathcal{F}(L_i, R_p^i)}{\lvert S\rvert}$
Function $\mathcal{F}$ differs with each task for which EC is used to create a solution.
G_p is the fitness of program p

Table 2.2: Common calculation of program fitness in the supervised machine learning form of EC.

[3]As hypothetically measured on all (often infinite in number) possible fitness cases.

Step 3: REPRODUCE

In search there is sometimes a trade-off between *exploration* and *exploitation*. Exploration means trying out options that are believed to be locally sub-optimal (in the hopes that globally these options will lead to an improved solution). Exploitation means focusing the search in the areas of the search space that have the highest known fitness values. In some kinds of search (e.g., simple hill-climbing) this trade-off does not exist, exploration never happens. However, in some machine learning paradigms (e.g., EC or reinforcement learning), the opportunity to try out multiple options makes it interesting to consider devoting some search energy to exploration. So the trade-off is between focus on the good models already found, and investigation of new models with sub-optimal values in the hope that some of those new models will lead the search to other models of even higher fitness values.

This trade-off is particularly true/obvious in a highly parallel search like EC. The REPRODUCE step in the process represents the exploitation phase of the search in which we focus our search towards those areas of the search space that we have reason to believe are more promising (in EC this is judged through the fitness calculation as described above).

Given that exploitation means focusing the search on high fitness individuals, and given that each individual in an evolving population represents a unit of search, the way to focus search is to throw out some of the low fitness individuals and replace them with copies of the high fitness individuals. There are a number of popular schemes for doing this, primarily *tournament selection*, *rank selection*, and *roulette selection* (see [Goldberg, 1989] for details). Table 2.3 outlines *tournament selection*. Tournament selection is the reproduction strategy used in NP.

For a population of M individuals, **Do M times**
 Pick K individuals from the population using a uniform random probability.
 Of these K individuals, copy the individual with highest fitness into the mating pool.

Table 2.3: Outline of tournament selection in EC.

So this creates, from a population of M individuals, a mating pool of the same size in which high fitness individuals have higher representation and low fitness individuals have lower representation. The curve in Figure 2.1 shows this representation "schedule" for $K = 5$ (the value used in all the experiments in this thesis). This schedule is independent of the size of M, so Figure 2.1 shows the schedule from 0% to 100%. The important thing to understand is that K is an important parameter in EC. K can be thought of explicitly as the "greediness" of the search process. If $K = 2$ and $M = 1024$, then it will take *at least* 10 generations for a single individual to become effectively the only member represented in the population. If $K = 4$ then the floor for this phenomenon is only 5 generations. This phenomenon is called the *convergence* of the population. The number of generations until the population has reached convergence is a complicated matter. K, however, is largely responsible for a factor called the *effective take-over rate*. The effective take-over rate expresses how long it will really take (on average) for the best individual to take over the population, under the assumption that no better individuals are produced during that period. The explanation for why this rate is not equal to the lower bounds described by the parameter K will be made clear below.

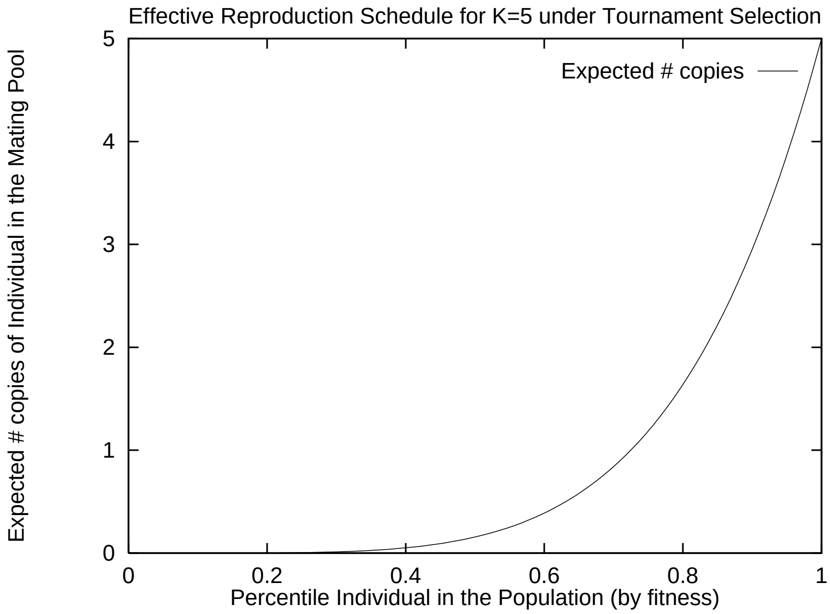


Figure 2.1: The effective schedule for exploitation created by using tournament selection with tournaments of size $K = 5$.

This general model for reproducing the population is usually called the *generation model* of reproduction. There is an equally valid model called *steady state* in which each reproduction step is the replacement not of one population with a new one, but only the replacement of a single low fitness individual with a high fitness individual. Though there is some disagreement about this, the general consensus is that these two approaches are equivalent in most important respects (see [Kinnear, Jr., 1993] for discussion). Although PADO operates on the generation model, it could be trivially modified to the steady-state model of reproduction.

Step 4: RECOMBINE

Evolutionary Computation must, by definition, include an aspect of exploration as well as an aspect of exploitation. To see why this is the case, imagine evolving a population of individuals in which fitness proportionate reproduction is done using, for example, tournament selection, but then this Mating Pool is moved to the next generation with no changes made to any of the component individuals[4]. For generation 0, there will be some individual p that has the maximum fitness for the population. After t generations of fitness measuring and fitness proportionate reproduction, there will be K^t copies of p in the population. Even for a population of 1,000,000 individuals, with $K = 5$, the entire population will be copies of

[4]Exploitation does not mean that there is no change. It means that the change is monotonic improvement, given what is known. But this is exactly the problem with EC. Right now there is no way to guarantee that a program transformation will improve the program's performance.

individual p after 9 generations. The reason evolution does not run quickly into this dead-end is that there is an opportunity for improvement; before individual p can take over the population, some new individual p' is created through the recombination process such that the fitness of p' is larger than the fitness of p.

This part of the search process, in which selected individuals are changed in an attempt to find even better parts of the search space, is called *genetic recombination*. That is, genetic recombination is the exploration aspect of search in EC. The two most popular forms of genetic recombination are crossover and mutation. What follows are one example each of the most common forms of crossover and mutation: *subtree crossover* and *point mutation*.

Crossover

In crossover, two or more individuals are chosen and some "genetic material" is exchanged between them. The hope is that high fitness individuals are made up of "building blocks" and that these building blocks can be reshuffled among high fitness individuals with positive effect at least some of the time. Crossover is often referred to in EC as *sexual recombination* to indicate that the inspiration for crossover is the apparent usefulness of the sharing of genetic material that takes place in the sexual reproductive process of most animals.

As an example of crossover, let us return to the example of evolving an arithmetic formula using only the operators $+$, $-$, $*$, $/$ and the operands x and constants to approximate e^x. Crossover is explained here in the context and representation of standard genetic programming because that paradigm and representation will be referenced most heavily later in the thesis. Figure 2.2 shows two example individuals, written in a tree structured format to make them easier to understand (as is traditional in Genetic Programming).

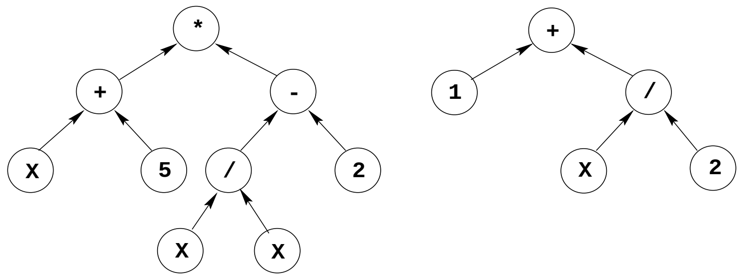


Figure 2.2: Two example functions in a hypothesized evolution example, **before** crossover.

The details of the crossover mechanism vary widely through EC because the representations in which this crossover takes place also varies widely. The common thread that identifies a search operator in EC as "crossover" is the exchange of material between two or more population members. Although the following is not a defining characteristic of crossover, it is almost always the case in EC, as it is currently practiced, that when material is selected for this genetic exchange, the material is chosen **at random** (or nearly at random). The traditional GP crossover procedure is "choose one random subtree from each program and exchange the subtrees."

The two individuals in Figure 2.2 can alternately be written as (* (+ x 5)(- (/ x x) 2)) and (+ 1 (/ x 2)). Notice that one of the convenient aspects of the s-expression, functional expression, or tree representation as it is often called, is that (unless there is added typing in the functions) any two subtrees can be exchanged between programs and the resulting programs will still be viable (i.e., syntactically legal). For an example of crossover, Figure 2.3 shows the new individuals if these two bold faced pieces were exchanged: (* (+ $\boldsymbol{x}$ 5)(- (/ x x) 2)) and (+ 1 **(/ $\boldsymbol{x}$ 2)**), resulting in the two new functions: (* (+ **(/ $\boldsymbol{x}$ 2)** 5)(- (/ x x) 2)) and (+ 1 $\boldsymbol{x}$).

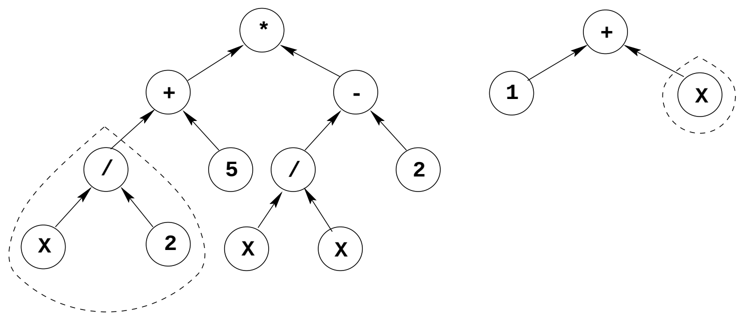


Figure 2.3: The two example functions from Figure 2.2 **after** crossover. The dashed lines indicated the exchanged genetic material (sub-functions in this case).

It should be noted that there is a disagreement about the added value (over mutation) of crossover as a search mechanism in biology as well as in EC (e.g., [Gathercole and Ross, 1996, Chellapilla, 1997, Angeline, 1997b]). In fact, two of the four main areas of EC, namely Evolutionary Strategies (ES) and Evolutionary Programming (EP), use no crossover at all. The author of this thesis is not convinced that crossover provides no additional benefit and as such, this thesis will address both crossover and mutation when the issue of recombination surfaces (as it does repeatedly in later chapters).

Mutation

In the recombination search operator called *mutation*, a single individual is taken and changed in some way that is independent of the other members of the population. This style of simulated evolution search operator is inspired by the biological genetic recombination process of mutation. Mutation in chromosomes can happen either because a gene is changed through the proverbial "cosmic ray" or through a mis-copying that takes place while a cell is being duplicated.

As an example of mutation, let us continue with the example of evolving an arithmetic formula using only the operators $+$, $-$, $*$, $/$ and the operands x and constants to approximate (fit) a set of data points. This time the two functions shown in Figure 2.2 will be changed through mutation. This means that some part of the individual (function) will be selected and changed to a new value (a new subtree in this particular genotype representation). How these subtrees to be changed are selected and how the subtrees to replace them are created will be addressed in detail later in this thesis. Figure 2.4 illustrates the mutation (subtree replacement) process in GP.

The details of the mutation mechanism vary widely through EC because the representations in which mutation takes place vary widely. The common thread that identifies a search

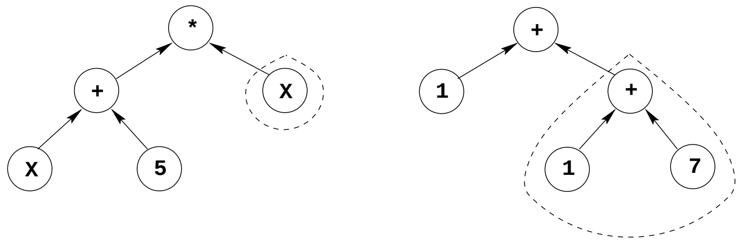


Figure 2.4: The two example functions from Figure 2.2 **after** mutation.

operator in EC as "mutation" is the creation of new genetic material and the replacement of existing genetic material in an individual with this created genetic material. Although the following is not a defining characteristic of mutation, it is almost always the case in EC, as it is currently practiced, that when material is selected for this genetic replacement, the material is chosen **at random** (or nearly at random).

2.1.2 Implications for this Thesis

In summary, there are three main phases of the EC learning loop and this thesis contributes novel aspects to one of them:

- *Evaluation of the fitnesses of each individual*
  It is **not** with this aspect of EC that this thesis takes issue.

- *Fitness-proportionate reproduction into a mating pool*
  It is **not** with this aspect of EC that this thesis takes issue.

- *Genetic recombination of the mating pool*
  It **is** with this aspect of EC that this thesis takes issue. There is no evidence that the aimless recombination that is so common in EC is better than focused recombination. On the other hand, this thesis provides specific evidence that there is something useful about carefully and purposefully choosing pieces of material to change or exchange during program transformations.

2.2 Evolution of Algorithms

The four main areas of Evolutionary Computation are, roughly, Evolutionary Strategies (ES), Genetic Algorithms (GA), Evolutionary Programming (EP), and Genetic Programming. Two additional fields, ALIFE and theoretical biological modelling, are closely related to EC but will not be described in this chapter.

Of these four disciplines, the work of this thesis is mostly closely related to genetic programming. As such, this section will first describe the standard genetic programming paradigm followed by the ways in which the other three areas differ from GP. It is important to note that historically, this is backwards. The correct historical order for the appearance of these disciplines is Evolutionary Strategies [Rechenberg, 1965], Evolutionary Programming [Fogel *et al.*, 1966], Genetic Algorithms [Holland, 1975], and then most recently, Genetic Programming [Koza, 1992].

One of the main attractions of evolutionary computation (in all of the flavors just mentioned) is that EC is trivially parallelizable. In fact, there is ample evidence that parallelizing EC often provides a super-linear speed-up.[5]

2.2.1 Genetic Programming

This subsection will give only the highest level impression of GP as an instantiation of EC. The best sources for learning more about GP are [Koza, 1992, Koza, 1994, Kinnear, Jr., 1994, Angeline and Kinnear, Jr., 1996].

The most important aspect of an instantiation of EC is the representation of the individuals to be evolved. In genetic programming, individuals are called *functions* or a *program* and are usually written in a functional, lisp-like structure called an s-expression. A simple example of such a function would be (* (+ (/ x 2) 5)(- (/ x x) 2)) and can be written as a tree (as shown in the previous section). The next chapter will make a case for the fact that functions are not the same as programs (even practically) and that the distinction is an important research question.

The atomic units with which these functions are constructed (i.e., the alleles out of which the chromosome is built) are called *non-terminals* for the positive-arity actions (e.g., $+$, $-$, $*$, $/$, cos, IF-THEN-ELSE) and *terminals* for the zero-arity actions (e.g., X, input-18, 17.98876). Notice that this representation just described is the genotype for evolution, but the phenotype (the actual thing to be tested that is created using a particular genotype as a blue-print) is the same. A GP function can be run "exactly as it is."[6] That is to say, GP functions can usually be interpreted directly, rather than going through a translation or compilation phase.

Typically, GP functions are evolved through the generational model rather than the steady-state model. It is traditional to denote the size of the population by M and the size of the tournaments (when using tournament selection) K.

After creating a mating pool through fitness proportionate selection, the standard GP paradigm usually divides the genetic recombination stage into two aspects. First, P_c% of the functions in the mating pool are subjected to crossover. Second, P_m% of the functions in the mating pool are subjected to mutation. Typically $90\% < P_c + P_m < 99\%$

Although crossover in GP has a number of different styles, all have the same important foundation. The process is to choose two subtrees **at random** and exchange them. In Mutation, the function is taken, a **random** subtree is chosen, a new **random** subtree is created, and this newly created subtree replaces the subtree just removed. One of the major contributions of this thesis is the answer to the question, "What can we do in the evolution of algorithms that is better than recombination **at random**?"[7]

---

[5]This phenomena, "the island model," has been discussed in works such as [Collins, 1992].

[6]In fact, GP was first implemented in lisp for this exact reason. In lisp, since data can be executed, the GP functions were stored as data in the form of pieces of lisp code that could be directly executed.

[7]The use of "random" in this context does not necessarily imply uniform distribution, but refers to the

2.2.2 Genetic Algorithms, Evolutionary Strategies, and Evolutionary Programming

Genetic Algorithms differ from Genetic Programming mainly in the representation of the evolving individuals. While in GP those individuals were represented as S-expression functions, in GA an individual is represented as a bitstring. This is a case in EC where genotype is not the same as the phenotype. The genotype in the case of GA is the bitstring. This bitstring can encode almost anything. For example, imagine that each chromosome is exactly 100 bits long and that each series of 10 consecutive bits encodes a value in the range [0,4095]. These 10 values might be parameters that, when properly optimized, describe an airplane wing with particular, desirable properties.

There are three popular forms of crossover in GA. *One point* crossover simply divides each bitstring into two pieces and exchanges the first section of each bitstring. This crossover is shown in Figure 2.5. In *two point* crossover, a consecutive series of bits in each bitstring is selected and exchanged. In *uniform* crossover, each bit is independently and with low probability, swapped for the bit in its equivalent position in the other string.

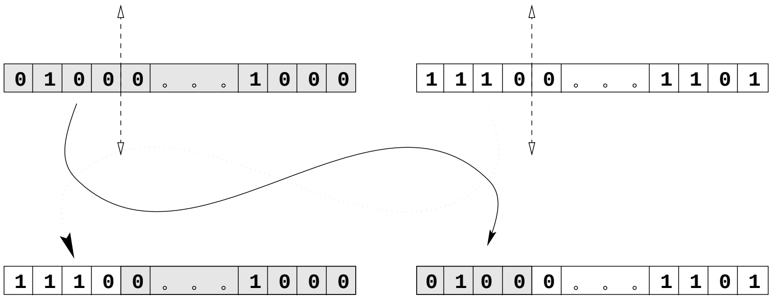


Figure 2.5: Two example functions in this hypothesized evolution example, before and after crossover.

A common GA mutation strategy is that each bit is, with some low probability, flipped to the opposite value.

The bitstrings for the individuals in a GA do not have to be of a fixed length (though they often are for a variety of reasons). GP can be seen as a subset of GA in the following sense. Imagine a GA that uses a variable length bitstring. Each GA chromosome is a bitstring and each series of 5 consecutive bits encodes 1 of 32 different terminals or non-terminals. If we now constrain the definition of crossover and mutation to match those of the GP paradigm, we have effectively embedded GP in a GA. In fact, GP is sometimes referred to as variable length, tree-based genetic algorithms.

In most other respects GA and GP are similar in tradition (e.g., the use of tournament selection or the generational model for simulated evolution). This is no surprise since, by most accounts, GP is one of the most promising progeny of the GA field. It is worth noting that while GP was an attempt to add to the GA paradigm a representation that worked better with the crossover operator, work in Messy GA (e.g., [Goldberg *et al.*, 1989]) has improved the GA field in some of the same ways.

independence from syntactic and semantic factors of the evolving programs that this thesis work capitalizes on.

Evolutionary Strategies differs from GA in three main ways. The first is generally that chromosomes in ES are floating point numbers rather than bits. The second is that mutation changes many or all of the values in a chromosome by convolving them with a Gaussian. The third is that ES typically uses a version of the steady-state evolutionary model. ES can be characterized as a form of parallel hill-climbing.

Evolutionary programming used to be the study of how to evolve finite state machines to recognize or reject strings correctly according to some target regular language. The EP field has changed dramatically since its introduction and now the two main differences between EP and GA are that EP uses only mutation (while GA also uses crossover) and that in EP there is no restriction on the genome type (as opposed to the bitstring representation so common in GA). However, because it has more bearing on the work in this thesis, EP will be addressed here in terms of mutation search through FSM space. Figure 2.6 shows an example of the phenotype of a simple finite state machine (FSM). The genotype is generally in a different representation whose details are not important for this overview.

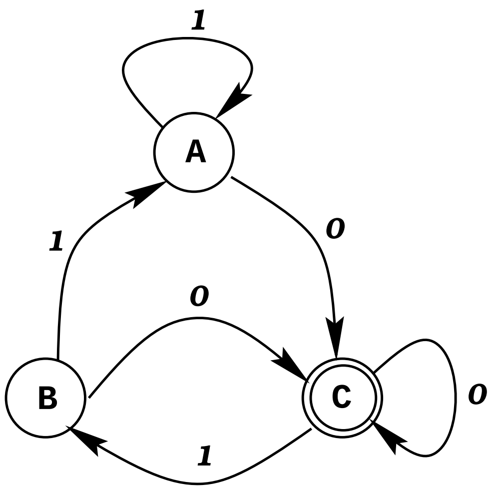


Figure 2.6: An example Finite State Machine (FSM) under evolution in an EP system

EP's recombination is entirely mutation, there is typically no crossover among the FSMs [Fogel *et al.*, 1966]. EP, like ES, has traditionally favored the steady-state evolutionary model over the generational evolutionary model.

Chapter 3

The PADO Approach

PADO (**P***arallel* **A***lgorithm* **D***iscovery and* **O***rchestration*) is a machine learning tool specifically designed for tackling signal classification problems. PADO is one of the contributions of this thesis and is the context in which the other contributions will be explained and evaluated. This chapter describes the essential aspects of the PADO approach to signal classification and its associated benefits and trade-offs.

3.1 A PADO Overview

This section is set up to provide the reader with a clear picture of where PADO fits in the field of GP and the terminology that will later be used to refer to aspects of the PADO approach.

At the highest level, PADO is a supervised learning algorithm for automatically creating an executable program[1] that can do a variety of signal understanding tasks on complex signals it has never seen before. The input required to the PADO system is a set of signals, a set of accompanying labels such that each signal has a corresponding label that describes the feature PADO should learn to associate with that sort of signal, and a set of parameterized signal primitives (PSPs). This high level process is shown in Figure 3.1.

The first of the three inputs (labeled 1 in Figure 3.1) to the PADO system is a set of signals. For PADO, a *signal* can be *any set of information*. The distinction between signals and *symbols* is the type distinction between the signals and their associated labels (as specified by some supervisor). In practice, signals take on a number of common conceptual forms and many instantiations of each of these conceptual forms. For example, acoustic, visual, and text "signals" are certainly three prevalent types of information blocks available to anyone and any machine on-line. For a particular signal conceptual type (e.g., visual signals) there are many instantiations (e.g., jpeg images, gif images, pict images). PADO can work with any conceptual signal type and any instantiation of those types.[2] In Chapter 7, PADO will be shown working on signals as diverse as sounds, images, and amino acid sequences.

[1]This executable program is made up of many smaller, individually-learned programs as described later in this chapter.

[2]That is, nothing in PADO has made a commitment to any of these issues.

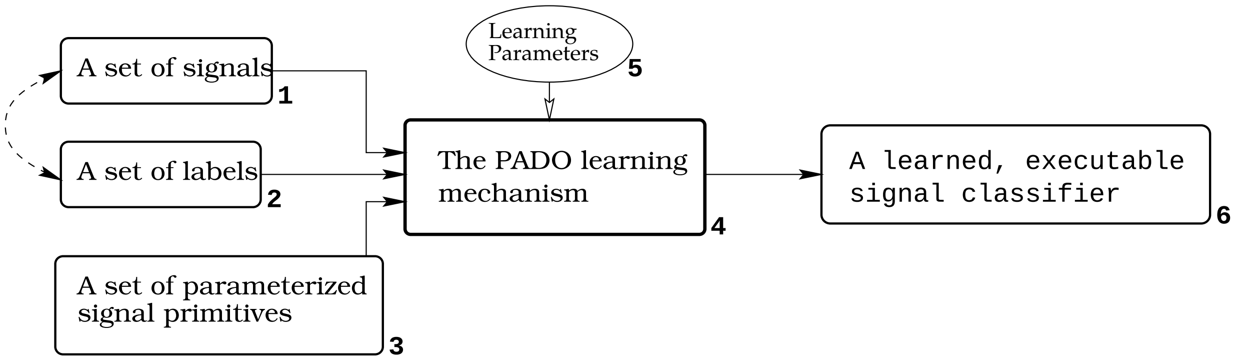


Figure 3.1: PADO as a learning method.

The second of the three inputs to the PADO system is a set of labels (symbols). There must be one label for each signal given to the PADO system. These labels can, like the signals, be of any form. PADO requires only that the labels be ordered values (e.g., 5.5, 19.10, 0.004), in which case the problem is a target value approximation problem, or that they be unordered values (e.g., "cloudy day photo", "sunny day photo", "rainy day photo") in which case the problem is considered by PADO to be a classification problem (i.e., "for each signal you see, learn to identify what sort of a weather-day the photo was taken on"). In this sense PADO is a *supervised machine learning* system.

The third input that PADO must have is a set of parameterized signal primitives (PSPs). As will be described at length in the rest of this chapter, PADO learns the desired mapping from signals to symbols by evolving programs in a supervised framework. Since PADO makes no assumptions about what form the signals will take, the user of the PADO system must provide the ways for evolving programs to examine the signals in question. For example, in Section 7.6, PADO is shown learning to distinguish classes of proteins. The signals in this case are strings of characters that represent the strings of amino acids that make up each protein. The labels in this case are one of five protein classes that define the problem. One of the parameterized signal primitives, PSP-Hydro, takes two parameters (*start* and *stop*). This PSP returns the amount of hydrophobicity/aliphaticity[3] of the amino acids in the amino acid sequence interval [start,stop]. What hydrophobicity/aliphaticity is and whether it is a useful view of the signals (with respect to solving this particular classification problem) is the responsibility of the user.

The parameterized signal primitives can be viewed as *replacing* brittle and equally design-time consuming pre-processing of the signals. A simple example will help to explain the sort of functionality that can be easily placed in the parameterized signal primitives. *AVERAGE* is a signal primitive that has been used in much past PADO work. *AVERAGE* is a function (written in C) that takes four values (x_1, y_1, x_2, y_2), and returns the average intensity value in the rectangle of the image signal with upper left corner (x_1, y_1) and lower right corner (x_2, y_2).

It is important to clearly draw the distinction between pre-processing features and parameterized signal primitives. PSPs allow the evolving programs to foveate on the signals.

[3]These are chemical/biological terms explained briefly in Appendix D.

This can only happen through parameterization or through selection of existing options. When there are very few features, providing them all is both simple and cheap while PSPs are unnecessarily time consuming. However, in the PSP-Hydro primitive for example (see section 7.6.2), there are about 500,000 possible features for each signal. And there are PSPs used in experiments reported in this thesis that give the programs access to over one billion features through the use of four parameters. Clearly providing all one billion features to each evolving program is not the right solution.

Now let us look down into boxes 4 and 6 from Figure 3.1. PADO starts by breaking down the classification problem to be solved into as many different smaller problems as there are classes to be distinguished between. These sub-problems are all learned in parallel by PADO using evolution as the driving learning force. Periodically, good programs for the solution of each of these sub-problems are chosen and grouped in order to produce a complete executable system (box 6 from Figure 3.1). This executable, *orchestrated* set of evolved (learned) programs is then tested against a set of signals that are not part of the training signal set. Figure 3.2 shows this basic process. Section 3.2 contains details on each of the steps in Figure 3.2.

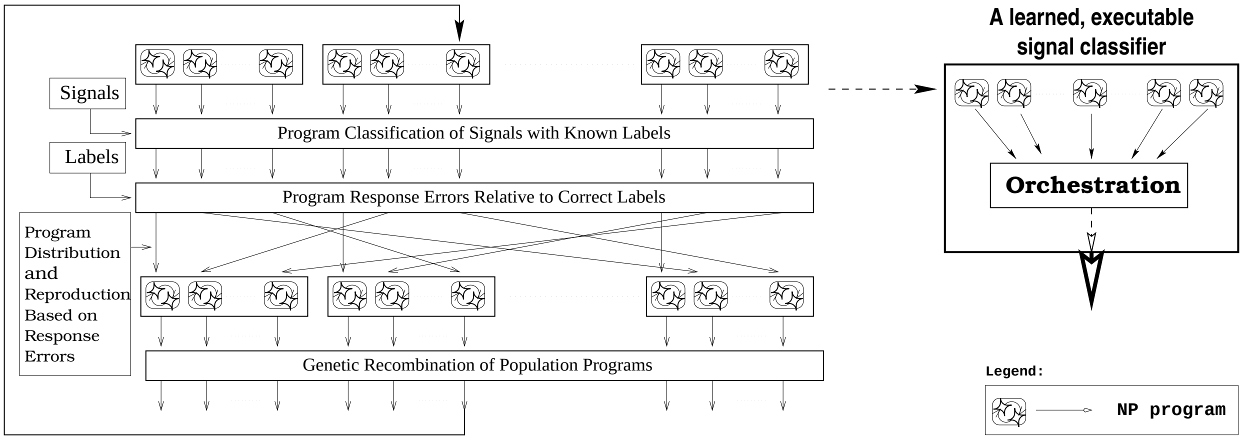


Figure 3.2: An Overview of the PADO internal workings.

This *hold-out set* of signals not included in the input signal set in no way affects the output of the PADO process, rather it is useful for determining how successfully the executable system output by the PADO process generalizes to unseen data. That is, after all, the real goal of the PADO system: to take as input a few signals and to learn enough from them to generalize to many other signals from the same problem domain (i.e. from the same signal/symbol distribution).

3.2 The PADO Internals

Clearly, any solution to the signal-to-symbol problem needs to be grounded in an algorithm that processes signal values from a signal. Consider the particular task of differentiating between many different signals. This task can be solved by learning a separate algorithm for discriminating signals of each signal class.

The new architecture of this thesis, PADO, is a technique for learning these algorithms directly so that there is no built-in commitment to the manner in which an algorithm investigates the signal and arrives at a decision. No features are chosen for PADO and no attention-focusing strategy is built in. PADO uses an evolutionary strategy to accomplish this learning process.

The goal of the PADO architecture is to learn to take signals as input and output correct labels. **When there are C classes to choose from, PADO learns C different, simpler**[4] *discrimination-solutions.* Discrimination-Solution i is responsible for taking a signal as input and returning a confidence that class i is the correct label.[5] Clearly, if all C discrimination-solutions worked perfectly, labeling each signal correctly would be as simple as picking the unique high confidence value. If, for example, discrimination-solution i returned a high confidence value, then the correct class label would necessarily be i. In practice, none of the C learned discrimination-solutions will work perfectly. This leads us to the recurring two questions of the PADO architecture:

1. "How does PADO learn good components (i.e. discrimination-solutions or programs)?"

2. "How does PADO orchestrate them for maximum effect?"

We will explain how PADO orchestrates these discrimination-solutions in Section 3.3. Now, let's delve into how one of these discrimination-solutions is built.

Discrimination-solution i is built out of a few programs. Each of these programs does exactly what the discrimination-solution as a whole does: it takes a signal as input and returns a confidence value that label i is the correct label. This use of multiple programs with the same general behavior is not redundant. In fact, this can be viewed as a "mixture of experts" problem. The insight is that many programs (models in the general case) that have slightly different behaviors but the same basic goal (i.e., have been trained to do well at the same task) can be coordinated so that the mixture of these experts does better than an individual expert by itself. Further information about these strategies can be seen in works such as [Littlestone, 1988], [Blum, 1995], and [Littlestone and Warmuth, 1994]. Evolution is a successful strategy for machine learning, and one of the insights of PADO is that a large group of experts is available as a side-effect of the population based learning. PADO's orchestration of these programs into a single discrimination-solution will be discussed in Section 3.3. Discrimination-solution i is then built out of a few programs by taking and combining (orchestrating) their responses. This orchestration will be discussed more in this chapter and the next.

The programs learned by PADO could be learned in a wide variety of representations. Appendix F gives details on representations that PADO has used in the past. Chapter 5 gives details on PADO's current algorithm representation: neural programming. To prevent a distracting, inserted discussion, details about neural programming will be postponed until that chapter. At the beginning of a learning session, the main population is filled with M

[4] The assumption being made is that it is easier to learn a problem of the form "Is this from class i or not?" than to learn a problem of the form "Which of C classes is this signal a member?"

[5] For this thesis, all the discrimination-solutions use the same range of confidence values. However, it would be simple to adjust PADO to work with varying ranges if necessary.

programs that have been randomly generated using a grammar for the legal syntax of the language[6]. All programs in this language are constrained by the syntax to return a number that is interpreted as a confidence value between some minimum confidence (MinConfidence) and some maximum confidence (MaxConfidence).

At the beginning of a new generation, each program in the population is presented with $|S|$ training signals and the $|S|$ confidence values it returns are recorded. Then the population is divided into C distinct groups of size M/C. The programs in group i are (within the constraint that each discrimination-solution is of size M/C) the programs that recognized class i better than any other class in the sense that they maximized a reward function **Reward** when $c = i$ (c is the class to which PADO is considering assigning program p)[7]. Table 3.1 shows this fitness computation in PADO.

int Reward (program p, class c, int Guess())	*Guess(p, j) is the*
Reward = 0;	*confidence program p*
<u>Loop</u> $j = 1$ to $\|S\|$	*returned for signal j*
<u>If</u> $(c = L_j)$ <u>Then</u>	
Reward = Reward + $((C-1) * Guess(p,j)$;	
<u>Else</u>	*C is the number of classes.*
Reward = Reward $- Guess(p,j)$;	L_j is the class label
return REWARD;	*for signal j.*

Table 3.1: The function for computing the fitness of program p in PADO if p is a discrimination program for class c.

Table 3.1 shows the procedure for computing the fitness for a program p under the assumption that p is a discriminator for signals of class c. This function is the sum of confidence errors across all $|S|$ training signals with the reward for those signals from class c weighted by a factor of $C-1$. Because programs may migrate from one pool to another, the fitness finally assigned to program p is the maximum of the returned values of Reward(p,c,Guess()) for $1 \leq c \leq C$.

The result of this training and division of the population can be described in another way. Imagine C different populations; the population i is composed of programs learning to discriminate signals of class i from all other signals. We will call such a sub-population a *discrimination pool.* Now we can measure each program and call its fitness the reward it gets from the Reward function just described. As will be described in a moment, the programs in these discrimination pools will undergo change over time (evolution is a form of search after all).

[6]PADO can accept non-random programs in its initial population. These can either be good programs from previous runs or even programs written by the user. This ability to express domain knowledge in code fragments to seed in the initial population is another example of EC's (and therefore PADO's) ability to accept a user's knowledge in a language which is natural to them.

[7]Though no detailed claim is being made, the PADO process of learning (evolving) and then deciding how best to interpret what has been learned (through the fitness "shuffling" of the Reward function) can be seen as a form of the EM algorithm.

After these changes to the discrimination pool programs, not only will some programs be better than others in discrimination pool i, but some programs will (with very high probability) have worse fitness than a program that guessed a class randomly would have. This form of divide and conquer for classification has the following property. Such a program (with fitness worse than random guessing) must *by definition* have better than random guessing fitness in some other discrimination pool j ($j \neq i$). So it would clearly be a benefit to move these very poor fitness programs to more appropriate discrimination pools. This is, effectively, what is happening in the processes described in the previous paragraph. This method for dividing the population into discrimination pools and then exchanging individuals when appropriate after the programs have been altered as part of the search process is called *discrimination pool guided migration*.

On signals that the program should return MaxConfidence for, the reward is multiplied by $C - 1$ so that, even though this only happens once in C times (assuming there are an equal number of training examples from each class), these signals will account for half the reward. This strategy values accuracy and coverage equally; the assumption is that it is as important to say "YES" when appropriate as it is to say "NO" when appropriate since these two cases are coverage and accuracy respectively. This normalization provides, on average, zero reward for a purely random classification strategy. Clearly, if the accuracy/coverage importance ratio is not 1 in a particular application, then it is trivial to modify.

Each discrimination pool is then sorted by increasing fitness and each program is ranked accordingly. C "mating pools" (temporary discrimination pools) are created in the following way. M/C times, K programs are chosen at random from Group$_i$. This method is called K-way tournament selection and is the fitness proportionate reproduction strategy of PADO.

Finally, the programs within each mating pool are subjected to crossover and mutation. All crossovers take place between two programs in the **same** mating pool. That means they are both recognizers of the same class. Much more will be said about how these search operators are constructed and used later in the thesis.

At this point the C mating pools are merged back into a new total population. Now the process of evaluation, reproduction, and recombination repeats. After many generations we find that the best programs in the population are much better than any that were created (randomly) at the start of the process.

To extract programs to use in the discrimination-solutions, we can pause the process after the evaluation step of a generation and copy out those programs that scored best or near best in each group i.

3.3 Orchestration

There are many cases in which a task to be approached with machine learning techniques can be or must be solved in more that one "piece." Learning a team of robotic soccer players is a good example of a task that could conceivably be done as a single agent, but lends itself very naturally toward learning sub-solutions (e.g., [Kitano *et al.*, 1997]) and *then* (or in addition) learning to ensure the mutual suitability of these sub-solutions [Andre and Teller, 1998]. This insurance of mutual suitability is the *orchestration problem*.

EC is a natural machine learning environment in which to find many behaviorally distinct models. PADO is a process of divide and conquer accomplished through the evolution of multiple pools of sub-solutions and the following orchestration of one or more learned programs from each pool.

One of the questions of this thesis is, "What opportunities are there for learning in the orchestration process and how much improvement can this learning provide?" The contributions of this part of the thesis will:

- Demonstrate the feasibility of PADO's divide and conquer strategy and suggest its preferability to unconstrained learning.
- Provide a specific justification for maintaining a population, a recurring issue in evolutionary computation.[8]
- Describe a number of specific techniques for orchestration learning and, through their successful application, demonstrate that orchestration is an important issue and that learned orchestration can dramatically improve generalization performance.

The basic justification for subdividing work as PADO does is this: it is usually preferable to search C spaces of size 2^j rather than one space of size 2^{Cj} $(j > 1)$. If we can find a good way to divide up a problem, then this kind of exponential reduction in computational effort may be possible, *at some cost in recombining these separate solutions afterwards*[9]. If we want to automate this process of learning, we had better be able to automatically pick this division into sub-problems. As was outlined in Section 3.2, signal classification for C classes is accomplished in PADO by the orchestration of C different discrimination-solutions. Each of these discrimination-solutions is composed of the B most fit programs from the corresponding group of the current generation.

Discrimination-solution$_i$ is built from the B programs that best[10] learned to recognize a signal from class i using a function F_L. For example, the B responses that the B programs return on seeing a particular signal can all be weighted, and their weighted average of responses interpreted as the confidence that Discrimination-solution$_i$ has that the signal in question is from class i. PADO does signal classification by orchestrating the responses of the C discrimination-solutions. On a particular test case, the function F_H (e.g., Weighted-MAX or Nearest-Neighbor) takes that function of the confidences from each Discrimination-solution$_i$ and selects one class as the signal class. Figure 3.3 pictures this orchestration learning process.

It is important to notice that the relation between the B responses in a particular discrimination-solution and the correct confidence may be *non-linear*. Similarly, the relation of the responses of the C discrimination-solutions to the correct symbol return value may be *non-linear*. For both these cases, the term non-linear refers here to the possibility that the mapping (function) between the response vector and the correct label is multi-modal. Much more will be said about orchestration and the issue of non-linearity in Chapter 4.

[8]See [Baluja and Caruana, 1995, Baluja, 1996] for arguments against populations.

[9]Teasing apart the separate costs of orchestration and cost saving of divide and conquer in PADO is outside the scope of this thesis but would be a valuable piece of future work on this subject.

[10]Based on the training results from that generation.

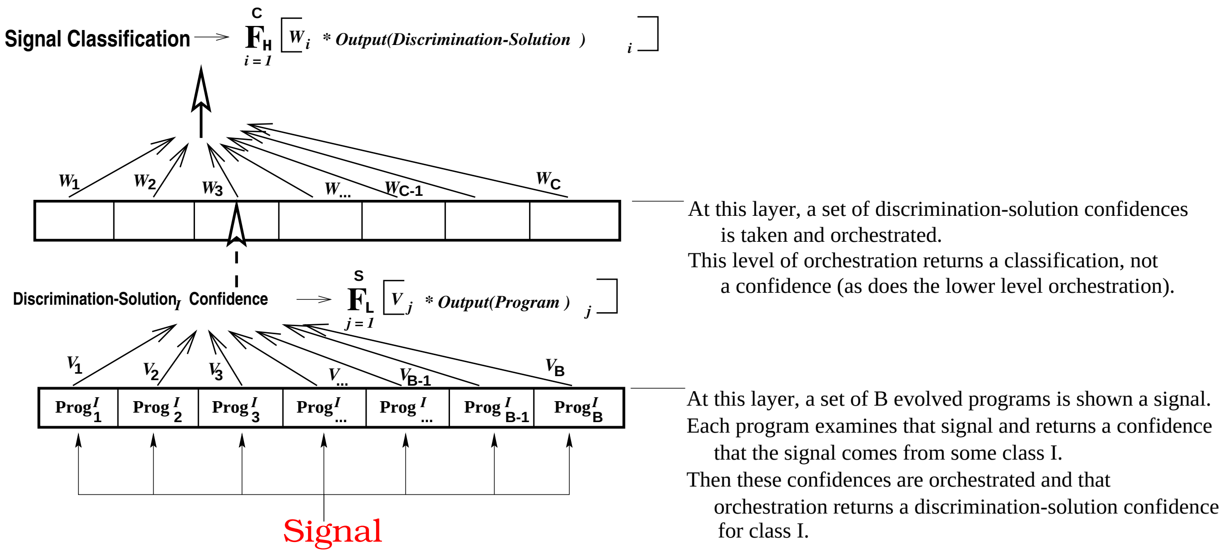


Figure 3.3: Orchestration in PADO. The weight vectors V_j and W_i are also learned.

3.4 Algorithm Evolution

3.4.1 Using GP to Evolve Turing Complete Programs

Genetic programming, as generally practiced today, does not evolve programs with access to both memory and recursion/iteration. Back in 1993, the two reasons for this were that neither memory nor looping/recursive constructs had really been worked into the paradigm. [Teller, 1994a] introduced expandable memory into GP ("indexed memory") and mentioned that a looping or recursive construct was all that was needed to make GP Turing complete. Later, [Teller, 1994b] gave a specific format for looping in GP and gave a formal proof that this new paradigm was Turing complete. This format included three simple rules not part of the contemporary GP toolbox. First, allow READ and WRITE non-terminals and allocate memory as needed for array positions specified by these two functions. Second, at the end of each tree evaluation, re-evaluate the tree (instead of stopping as a traditional tree-GP system would do). Third, stop this repeated evaluation only when "halt" is indicated by a specific value in a specific memory location.

This summary is not meant to suggest either that this was a great surprise to GP community or that this was first evolution of Turing complete systems (see Chapter 8 for previous examples). [Teller, 1994b] did, however, coincide with increased interest in Turing complete GP systems.

One of the main thrusts of this thesis is the efficient and effective evolution of algorithms that are written in a Turing complete language. It is known to be qualitatively harder to evolve such programs (relative to the functions traditionally evolved in GP). Appendix B includes a short discussion of why Turing complete programs are harder to learn/evolve and why they are worth the extra effort. Also, Section 7.5 demonstrates that the evolving programs really can take advantage of additional examination time when it is given to them.

3.4.2 Algorithm Representation in GP

PADO has moved through a series of representations for program evolution that will not be discussed in the main body of this thesis and are only summarized in Appendix F. [Teller, 1994b, Teller and Veloso, 1997, Teller, 1996] track the history of PADO program representations. PADO has moved to a still newer representation that is a significant departure from past work. This newest representation, *Neural Programming*, has brought PADO in line with the thesis objectives. Neural Programming is the culmination of a research program aimed at more control and better understanding of the evolutionary computation process. The entirety of Chapter 5 is devoted to describing this representation.

On the representation topic, it is important to give some idea about what the atomic units of computation in PADO have been in the past and are in this thesis. Table 3.2 shows the standard PADO actions. In seven domains that PADO was applied to, this exact set of actions was used for all experiments. This demonstrates that PADO can work on a variety of domains without altering its node action set, except for appropriate changes to the parameterized signal primitives.

Table 3.2: Standard PADO node actions

Continuous	Boolean	Choice	Memory	Signal PSP	Constant
Add	Max	If-the-Else	Read	SignalPrimitive_0	0..MaxValue
Sub	Min	PIFTE[11]	Write	...	
Mult	Less			...	
Div	Equal			...	
	Not			SignalPrimitive_i	

3.4.3 The Halting Problem

A succinct definition of the halting problem is "It is impossible to determine in any finite amount of time whether or not an arbitrary program written in a Turing complete language will halt on an arbitrary input." This poses a problem for PADO. In the evolution of algorithms we need to run each program in the population on a number of inputs and measure their fitness according to their responses to those inputs. If we have to wait for all of the programs to finish, and some may never finish, simulated evolution is sunk. The halting problem tells us that we cannot just "look at the program and see that it will never finish." We can do that for many programs ([Maxwell III, 1994]) but not for all of them. Since PADO has committed to evolving programs that have the potential to run forever, some addition must be made to the paradigm to address the halting problem.

The solution taken by PADO in all three of its representations to date is to require that evolving programs be anytime algorithms. This means that an answer can be extracted from a program at any point during the program's execution. In all three representations, this can be accomplished by simply expecting the program's response to be updated as it

[11] Probabilistic If-Then-Else. This has an interesting smoothing effect on program spaces.

improves and stored in a particular memory slot. Then when the environment reaches some time threshold, the answer is taken from that part of the program's memory, whether or not the program has halted.[12] This forces programs to learn not only to solve the problem, but also to store that solution in a specific piece of memory. This is an elegant way to solve the halting problem for simulated evolution but there is evidence that this method can hinder as well as help the learning system, depending on the problem to be solved. [Andre and Teller, 1996] has more details on these trade-offs.

This term "anytime algorithm" does not seem like an important distinction from traditional tree-structure GP program only because those programs do not run long enough until they halt to be an issue. But consider that in order to get the "answer" from a GP-tree, the entire tree must be evaluated. If the entire tree is not evaluated, the response is undefined. And defining a default response for a prematurely halted GP-tree is still not an *anytime* algorithm because there is no opportunity for the GP-tree to iteratively improve its response until that response is required.

3.5 Striping PADO Down to the Essentials

This chapter of the thesis has given details about the basic PADO structure. After this chapter, the thesis uses experimental evidence to demonstrate many of the claims (about PADO and neural programming) that are made. These demonstrations must be done in a specific instantiation of PADO, not a hypothetical or abstract version of PADO. This section gives details about the specific PADO system used to illuminate the more theoretical claims of this thesis. Thus, from this point forward in the thesis (unless otherwise noted) references to PADO refer to this specific thesis instantiation of the method.

Before the details of the current PADO system are described, it is worth mentioning why this particular PADO system was chosen for the thesis embodiment. As may be seen from the previous sections of this chapter, there are a number of issues (several of them novel) all of which have been part of the changing PADO system over the years. The more these disparate issues remain in PADO, the more complicated it would be to sort out the responsibility for success or failure in particular experiments.

This leads to the question, "Is this thesis primarily about the theory or primarily about the results?" The thesis is only valuable if both aspects exist in good measure, but the decision was made to focus on making clear the theoretical contributions, even at the expense of a few percentage points in the experimental results. As a result, while some of the issues discussed in the previous section are legitimately part of PADO, they were purposefully excluded from the implemented thesis version of PADO in order to maintain the desired clarity of contribution.

Orchestration for PADO

Clearly some specific representation had to be chosen for PADO. That representation, as has already been mentioned, is Neural Programming (NP). The entirety of Chapter 5 is devoted

[12] In fact, in the current PADO representation, Neural Programming, there is no built-in mechanism for halting since it was designed to be used in this anytime way.

to detailing the NP programming language.

PADO was described as orchestrating at many different levels and at many different times. The main choices were:

- How many programs to take from each discrimination pool? – (B)
- How to pick those programs?
- How to orchestrate those programs into a discrimination-solution? – (F_L)
- How to orchestrate these discrimination-solutions into a full PADO system? – (F_H)
- Does this orchestration happen during or after training (or both)?
- Does this orchestration use the training data or a separate hold-out set?

Here is what was chosen:

- One program from each discrimination pool. – ($B = 1$)
- How to pick this one program will be addressed in the next chapter.
- No orchestration necessary within Discrimination-solutions. – ($F_L = \text{NOP}$)
- How these C programs will be orchestrated is the subject of the next chapter.
- Orchestration will only take place during training.
- Orchestration will be based entirely on the same training data.

Program Representation in PADO

The impetus for PADO's move away from traditional GP representations was the desire to efficiently evolve programs written in a Turing complete language. The hypothesis is that many difficult problems (particularly ones in which foveation is required) are much easier to solve when iterations (or recursion) and memory are aspects of the learned model. In the first two PADO representations (described in Appendix F), memory was included in the learned models through the use of Indexed Memory primitives ([Teller, 1994a]). This was, for those representations, the most seamless way to add memory to the evolving programs.

However, when PADO moved to its current representation for evolving programs (NP), the data-flow nature of that representation created another kind of natural memory for the evolving programs. This memory source is the recurrent loop memory similar to memory use in recurrent systems like recurrent neural networks. Indexed Memory can still be used with evolving NP programs, but because it is largely redundant, indexed memory was dropped as a memory source for PADO. Section 6.6 demonstrates that indexed memory can still be used with the NP representation.

Guided Migration in PADO

Section 3.2 described how a C class classification problem is broken down in PADO to C distinct discrimination problems. The previous sections of this chapter also described the insight that:

> *If a program performs worse than random guessing in one discrimination pool, it must necessarily perform better than random guessing in some other discrimination pool.*

That insight, coupled with the empirical evidence that crossover and mutation often produce programs not only worse than their parents, but worse than random guessing on the same problem, lead to the process of *discrimination pool guided migration.* While this guided migration clearly makes good use of otherwise wasted information, it improves the evolutionary process in a complicated manner. In evolutionary computation the process of exchanging some individuals between relatively distinct populations is often called *niching* (after the term for the same phenomena in real flora and fauna), *demes* or the *island model* (see [Collins, 1992] for details). The thesis instantiation of PADO does not have discrimination pool guided mutation; the discrimination pools are entirely separate for the entire run of evolution.

Adding Noise to the Training Input Signals

PADO was designed to learn complex signal understanding problems even where there is very little to learn from (e.g., only 10-20 labeled signals from each class to be discriminated between). In much of the published work on PADO (e.g., [Teller and Veloso, 1997, Teller and Veloso, 1995d, Teller and Veloso, 1995a]) noise was added to the training signals in an effort to prevent (or at least forestall) overfitting to the training set. This addition was found to be quite effective in most of the domains PADO has been applied to over the years. Despite this overwhelming evidence that it improves performance, changing the training signals clouds the issue of what PADO can learn from small training sets and so PADO does not include any noise additions to the training signals.

Finally, there are as in any system, a number of parameters that must be set. Every effort was made to set these parameters in PADO to reasonable values and to leave them fixed for *all* the experiments described in this thesis. For completeness, Appendix C.2 has an annotated version of these parameters.

Chapter 4

PADO Orchestration

This thesis introduces particular aspects of the process of automatically sub-dividing a problem, developing multiple solutions to each sub-problem, and then *orchestrating* the sub-solutions into a complete solution. This chapter establishes through a series of experiments that this divide and conquer strategy can be done automatically, that the five specific techniques introduced for learning orchestration are effective, and that the orchestration of sub-solutions is a rich, intriguing area for machine learning study.

Figure 4.1 revisits PADO's general orchestration strategy as described in the previous chapter. In short, the lower level of orchestration pictured with the V weights takes a collection of evolved programs, each of which is trained to perform the same discrimination task, and orchestrates their responses on each particular signal to be identified into a single, aggregate response. How this aggregate response is obtained is the lower level orchestration problem in PADO.

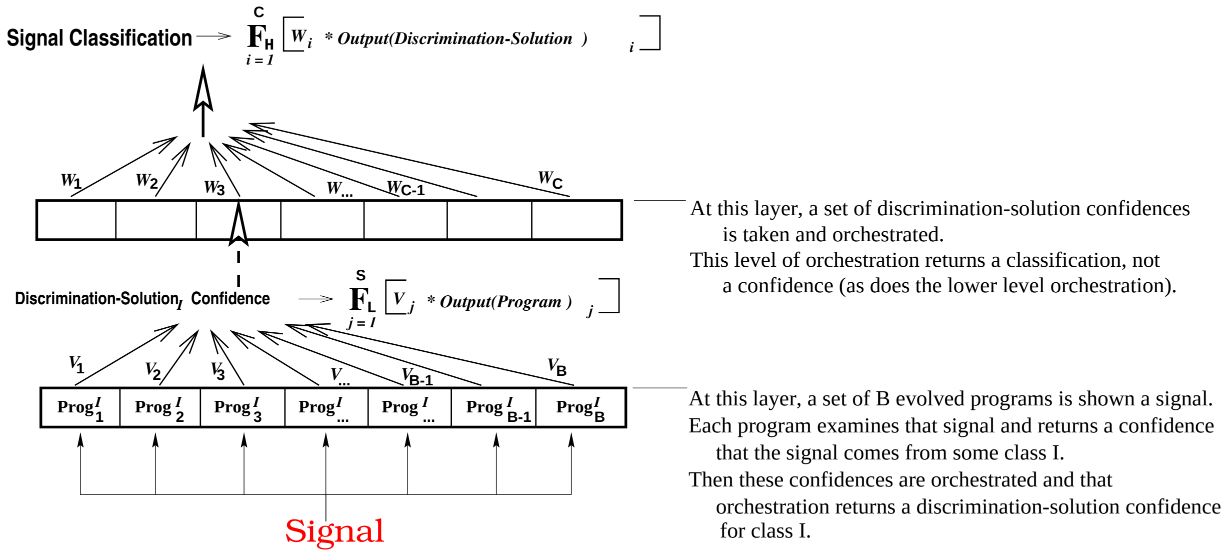


Figure 4.1: The two tier general orchestration process in PADO.

The higher level orchestration problem pictured with the W weights is as follows. How to combine these aggregate responses from each discrimination problem into a solution to the larger classification problem into which the multiple discrimination problems have been broken. In this higher level orchestration problem, the responses being orchestrated do not come from programs (or orchestrated groups of programs) trained to perform the same task (as in the lower level orchestration problem). Instead, the responses to be orchestrated at this higher level, come from solutions to *complementary* problems.

For this chapter, we will concentrate on the higher level orchestration in PADO. To focus in this way, we will make the simplifying assumption that exactly one program will be chosen from each discrimination pool. This means that the lower level orchestration function F_L is a NOP. This simplification of PADO's orchestration for explanation purposes is shown in Figure 4.2.

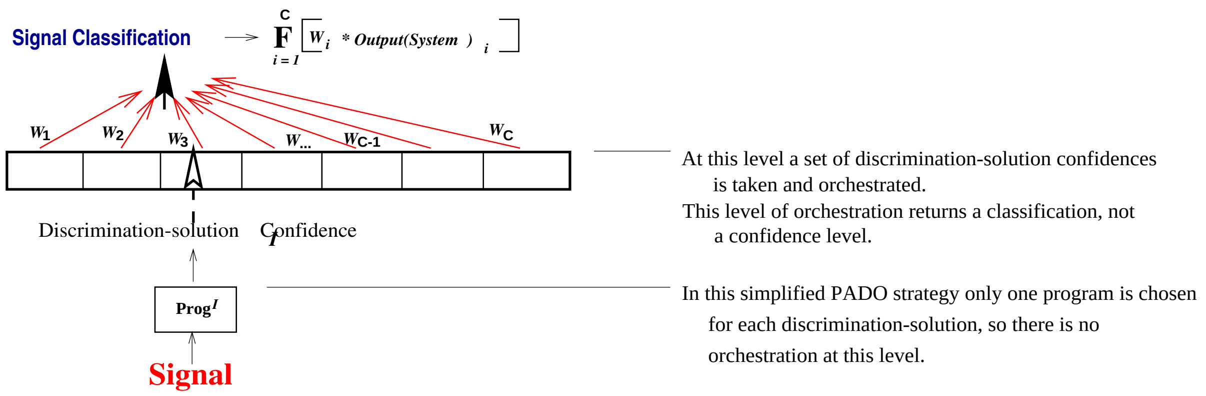


Figure 4.2: PADO's simplified orchestration strategy

This brings us to the topic of this chapter: how to learn to orchestrate the best possible team for classification. This chapter will address issues of which programs to orchestrate, how to orchestrate them, and how to bias and improve orchestration through evolution itself.

4.1 Orchestration Techniques

To understand the domain and range of this orchestration work, it will help to start by describing the orchestration options that we investigate in the course of this work. Most briefly, these are:

- **Fitness Proportionate Orchestration** – the reasonable but fixed control against which the learned orchestration paradigms are measured.
- **Evolved Orchestration** – implicit orchestration by evolution with a revised orchestration-sensitive fitness function.
- **Weight-Search Orchestration** – orchestration by the learning of parameters.
- **Program-Search Orchestration** – orchestration by the selection of appropriate programs from the discrimination pools.
- **Nearest-Neighbor Orchestration** – orchestration by the creation of an appropriate non-linear orchestration function.

The simplest option is to do no more than necessary. This *fitness proportionate orchestration* will be the baseline against which we will measure other techniques. In this fitness proportionate orchestration, PADO picks the best program from each discrimination pool and orchestrates them using a fixed function with fixed parameters (i.e., no learning). The next simplest strategy is to maintain the fitness proportionate orchestration whie incorporating its particular demands into the fitness function. This is *evolved orchestration*.

As long as there are a number of "parameters" or "coefficients" in the fixed orchestration function, another technique for improving orchestration could be to learn, for a particular set of programs, which coefficient values optimize the group's coordination. This is *learned weight orchestration*. Conversely, in *learned program orchestration*, with these coefficients fixed, as in the fitness proportionate orchestration case, we can learn which programs (one from each discrimination pool) would produce the best group for task coordination. It is possible to learn, for a fixed program set with fixed parameters, what *function* would optimize that particular group's coordination. This is *learned function orchestration*. The function approximator used in this chapter is a nearest-neighbor space.

These are orthogonal tactics for improving orchestration behavior. Future work could investigate *combinations* of these procedures in series or even in parallel (though this latter case could be more complex for some procedure combinations).

4.1.1 Fitness Proportionate Orchestration

This section describes the *fitness proportionate orchestration* procedure against which learned orchestration techniques will be measured and from which the learned orchestration techniques will start.

The first step in fitness proportionate orchestration is to pick the best program from each discrimination pool to be in the orchestrated set. Specifically, the procedure is to sort the programs in each discrimination pool according to training set fitness and choose the top ranked program from each discrimination pool. Remember, there is one discrimination pool for each class and the programs assigned to this pool are evolved with the purpose of creating a discrimination-solution for that class (i.e., solving the binary classification problem "Is this from class i or not?").

This selected program set is orchestrated with a simple function using reasonable, fixed coefficients (hereafter referred to as weights).

Each program p has a fitness, G_p (averaged over all training examples), that ranges between 0.0 and 1.0. Let us give each program p selected as part of the final PADO classifier an orchestration weight of $W_p = G_p$. For a particular test signal, each program is shown the signal and returns a response R_p between R^{min} and R^{max}. Let us refine R_p according to:

$$R_p^{\text{new}} := (W_p * R_p) - (1 - W_p) * \frac{R^{max} - R^{min}}{2} \tag{4.1}$$

Now if the program p from class i has the highest[1] R_p, PADO concludes the test signal is from class i. However, because of Equation 4.1, PADO "listens more attentively" to

[1] I.e., $R_p > R_{p'}$ for p' equal to all $C - 1$ other programs being orchestrated.

programs that are, on average, "more reliable" (i.e. have higher fitness). Notice that this fitness proportionate orchestration makes no attempt to find:

- a fitness function that promotes orchestration
- the optimal weight W_p for each program
- the optimal programs to orchestrate
- the optimal function on which the weights act

4.1.2 Evolved Orchestration

Clearly, an alternative to learning how best to orchestrate a number of programs, or which programs to orchestrate, is to try to change the basic learning of the programs so that the programs that perform best on the training set will also be the best orchestrated without further learning. Because PADO learns programs in an evolutionary framework, this amounts to incorporating the demands of a particular orchestration strategy into the *fitness function*.

At each generation, after computing the fitness of each evolving program, PADO can assign each program a new fitness based on its ability to orchestrate with high fitness individuals from the other discrimination groups.

For each discrimination pool i, PADO creates J biased-random[2] *groups* (using one program from each of the other $C-1$ discrimination pools) called $P_1^i..P_J^i$. For each discrimination group i, for each program p in the group:

$$G_p^{new} = \frac{\sum_{j=1}^{J} \mathbf{E}val(\{p\} \cup P_j^i, W)}{J} \tag{4.2}$$

$\mathbf{E}$val$(\{p\} \cup P_j^i, W)$ is the percentage of training examples that the orchestrated PADO program set of $\{p\} \cup P_j^i$ correctly classifies, orchestrated with the weight set W (such that $W_p = G_p$ for each program p in $\{p\} \cup P_j^i$). This value is an approximation to how well p "orchestrates in general", relative to other programs in the same discrimination pool.

From this point on, PADO follows the fitness proportionate orchestration strategy except that the program from each discrimination pool chosen for orchestration will be the *best* based on G_p^{new} rather than old fitness. In evolved orchestration, G_p^{new} replaces G_p in the entire process. So the number of copies that program p is likely to have in the next generation is a function of G_p^{new} and therefore a function of how well p "orchestrates" with programs from the other discrimination pools. p's fate is no longer a function of p's performance on its assigned discrimination task.

4.1.3 Weight-Search Orchestration

Another variation on the fitness proportionate orchestration in PADO is to try to find the best set of weights for the chosen set of programs to be orchestrated; this is *weight-search*

[2] Picked randomly, biased toward higher fitness individuals according to the same schedule shown in Figure 2.1. For this thesis, J was set to M/C.

orchestration. We get one degree of freedom for each program to be orchestrated by allowing each element W_p of the weight set W to vary between 0 and 1 (instead of simply setting it to G_p). There is a wealth of literature about parameter tuning and for this general situation (parameter tuning for linear function), techniques such as linear regression (e.g., [Press *et al.*, 1992, Rice, 1987]) have been well studied. The following method for tuning the parameters was chosen simply to parallel the method used in Section 4.1.4.

To begin, this strategy will create a set of programs Z to orchestrate. Z will be made of the single best program from each discrimination pool based on training set fitness. Next, the strategy will initialize the weight set W for orchestration such that $W_p = G_p$ for each program p selected for Z, just as in the fitness proportionate orchestration case. Now we can search for a better set of values for W. After initialization, the following three steps of a simple simulated annealing process are repeated Q_l times.

Initialize: $\vec{d} = 1$ and $W_{best} = W$

1. Pick i between 1 and C.
2. $W_i = W_i + \mathcal{O} * d_i$
3. <u>If</u> **E**val(Z,W) > **E**val(Z,W_{best}), <u>then</u> $W_{best} = W$ <u>else</u> $d_i = -d_i$;

$\mathcal{O}$ is a Gaussian random number generator. Step 1 is performed by choosing i from the range $[1..C]$ with a probability distribution equal to the sampled success rates at step 3 on previous selections of i in step 1. Step 3 is annealed so that local minima are partly avoided. This annealing takes the form of selecting the inferior option with probability equal to $\frac{Q_l - t}{2Q_l}$ $(1 \leq t \leq Q_l)$.

4.1.4 Program-Search Orchestration

Another variation on the fitness proportionate orchestration in PADO is to fix the weight set W at their default values ($W_p = G_p$ for each program p selected for orchestration) and to try to find the set of programs that best fit those weights; this is *program-search orchestration*. We get one degree of freedom per class (i.e. one per orchestration weight) by allowing the program to orchestrate for discrimination-solution i to vary over all of the programs in discrimination pool i. The rationale for picking some program other than the "best" program (highest fitness measured on the training set) from a discrimination pool is the following. Picking some other program may (or may not) reduce the individual discriminatory power of that one program discrimination-solution, but may decrease the non-linear relations in the orchestration process by enough to produce a net classification performance improvement for the entire PADO system.

To begin, this strategy will initialize the set Z to be the best program p from each discrimination pool based on standard fitness (G_p). The orchestration weight set, W, will be set for each program p to be orchestrated to its default value, G_p. Now we can search for a better program set Z ($Z = \{p_1, ..p_C\}$). After initialization, the following three steps of a simple simulated annealing process are repeated Q_l times.

Initialize: $Z_{best} = Z$

1. Pick i between 1 and C.

2. Exchange Z_i[3] for another program from the discrimination class i pool.

3. If $\mathbf{E}\text{val}(Z,W) > \mathbf{E}\text{val}(Z_best,W)$, then $Z_{best} = Z$

Step 1 is performed by choosing i from the range $[1..C]$ with a probability distribution equal to the sampled success rates at step 3 on previous selections of i in step 1. Step 2 is performed with a bias toward more highly fit programs using the schedule shown in Figure 2.1. Step 3 is annealed so that local minima are partly avoided. This annealing take the form of selecting the inferior option with probability equal to $\frac{Q_l - t}{2Q_l}$ $(1 \leq t \leq Q_l)$.

4.1.5 Nearest-Neighbor Orchestration

The introduced weight-based orchestration schemes explored in the previous few sections are *linear orchestrations*. An example will highlight the distinction between these and *non-linear orchestration* strategies; the subject of this section.

Suppose that when the programs chosen to discriminate for classes i and j *both* return high confidence, it is the case 93.2% of the time that the signal in question is really from class k, and it is the case 99.1% of the time when, in addition, the class k program confidence value is particularly low. In other words, there may be a set of signals from class k that are not successfully recognized by the class k program discriminator, but these same confusing characteristics cause a reliable pattern of responses in the other program discriminators.

In order to take advantage of these additional indicators[4], PADO needs to employ a function approximator that can capture non-linear relationships. Nearest-Neighbor is such a function approximator and we will now describe how it can be incorporated as an orchestration strategy.

Let us assume, as we did in the fitness-proportionate orchestration and in the weight-search orchestration, that we will simply choose one program from each discrimination pool and that that program will be the single best program for that generation (as measured on the training set).

For a C class classification problem, we will create a C-dimensional nearest-neighbor space called $\vec{\mathcal{N}}$. Now PADO fills this space with labeled points. For each labeled training input S_i, PADO knows how each of these programs to be orchestrated responded. If we call the label (correct class) for this training example L_i and we call these C responses from these C programs $R^1_{S_i}$ through $R^C_{S_i}$, then we can set the point in C-dimensional space $\mathcal{N}[R^1_{S_i}, ..., R^C_{S_i}]$ to have the value L_i. After doing this for all the available training inputs $(1 \leq i \leq |S|)$, we have a picture of how this group of programs interacts over a sampling of inputs. The process of building this nearest-neighbor space is outlined in Table 4.1.

[3] Changing Z_i from p to p' changes W_i from G_p to $G_{p'}$.

[4] The non-linear relations between program responses that were missed by the other orchestration methods described in this chapter.

For each labeled training input S_i with class label L_i
  For all of the programs to orchestrate ($1 \leq p \leq C$)
    Let $R^p_{P_i}$ be the response of program p to training input S_i
  $\mathcal{N}[R^1_{S_i}, ..., R^p_{S_i}, ..., R^C_{S_i}] = L_i$

Table 4.1: The process of building the nearest-neighbor space for orchestration.

Extracting the answer (label) from a nearest-neighbor space is a well studied process and PADO uses, rather than adds to, this field. Therefore, those readers familiar with the use of nearest-neighbor spaces can now safely skip on to Section 4.1.6.

Extracting the Answer from Nearest-Neighbor Space

Now that PADO has this structure, $\vec{\mathcal{N}}$, a process of choosing a class on an unlabeled test input must be described. On test input E_i, the set of programs will have a set of responses that corresponds to the point $[R^1_{E_i}, ..., R^p_{E_i}, ..., R^C_{E_i}]$ in the nearest-neighbor space. For real-world problems, the chances are very low that this point in the space will already have a label. This means that PADO must look to nearby points to determine a likely class. There are many such k-nearest-neighbor strategies. One of the simplest, and the one PADO uses is to give every labeled point a vote, but to weight that vote by $1/r^2$ where r is the distance in C-dimensional space between the labeled point in question and the point $[R^1_{E_i}, ..., R^p_{E_i}, ..., R^C_{E_i}]$.

There are two important issues surrounding how to maximize the efficiency of prediction for a nearest-neighbor space that relate to normalization of the dimensions. Essentially, the first problem is if the program discriminator for class p always says, for example, either 101 or 107, but is always right[5], it would be nice to count this dimension in the NN space as more important than some other dimension l in which the program always says either 101 or 107 but those responses only have a 0.75 correlation with the correct discrimination confidences. PADO solves this problem by weighting each dimension's value by G_p, the fitness of that dimension's representative program.

The second issue is that if there is a discriminator program for class i that always responds with either 101 or 107 with 0.75 correlation to the correct discrimination confidence, and there is another program for class l that responds with either 0 or 250 with the same correlation to the correct discrimination confidences, we would like to weight those NN dimensions equally since they convey the same amount of statistical information. However, because we are measuring distances, the program for class l will have a larger effect on the class chosen than will the program for class i. The solution in this case is to determine VAR_p, the variance of the $R^p_{S_i}$ over all i, for p between 1 and C. Then each dimension p can be weighted by $1/\mathrm{VAR}_p$. This normalization is always the right thing to do *when the dimensions are independent.* Even though the programs never explicitly interact and are working on different discrimination problems, because they are responding to the same set of inputs

[5]That is, there is a perfect correlation between the program's response value and the correct discrimination confidence.

their responses will not be independent. Empirically however, this normalization is quite close to the sort of normalization determined by more complex methods such as PCA.

Table 4.2 shows a nearest-neighbor based algorithm for choosing a class label for an unlabeled test signal.

For each program p to orchestrate $(1 \leq p \leq C)$
    Let R_E^p be the response of program p to test input E
Let δ_E be the point $[R_E^1,...,R_E^j,...,R_E^C]$ in C-dimensional space

For each labeled training input S_i with class label L_i
    Let δ_{S_i} be the point $[R_{S_i}^1, ..., R_{S_i}^C]$ in C-dimensional space
    Let D be the weighted, normalized **squared**, Cartesian distance between δ_E and δ_{S_i}
        For each program p to orchestrate $(1 \leq p \leq C)$
            $D = D + (R_E^p - R_{S_i}^p)^2 * \frac{G_p^2}{\mathrm{VAR}_p}$
    Tabulate a vote for label L_i with strength equal to $\frac{1}{D}$

Select the label that received the highest vote strength

Table 4.2: The process of using the nearest-neighbor orchestration for PADO classification.

The process shown in Table 4.2 simply works as follows. When a particular set of programs return confidences about some particular testing signal, the distance is measured in the C-dimensional nearest-neighbor space ($\vec{\mathcal{N}}$) between that point and every labeled training point. This distance is weighted along each dimension by the fitness of the program responsible for that dimension and normalized by the variance of the training signal responses along that dimension to remove one source of "noise" from the function. Each labeled training point in the NN space votes for its label (class) with a vote strength of $\frac{1}{D}$. The label (class) that receives the largest total vote strength is chosen as the signal class to predict.

4.1.6 Other Orchestration Techniques

Example Combination Strategies

It is possible to combine orchestration variations like the ones described above in a variety of ways. For example, we can first search through program space with a set of fixed weights (Section 4.1.4), and once we find this cooperative set of programs we can then search through weight space (Section 4.1.3) for this discovered set of programs in order to find the optimal weight set for those programs.

Another combination possibility is to evolve better orchestration (Section 4.1.2) and then at orchestration time, to do a search to find the best program set (Section 4.1.4).

Many combination strategies can be interleaved rather than done in series. It is easy to imagine, for example, interleaving search in program space and weight space.

One more tangential use of the orchestration information that is worth mentioning is the idea of *Orchestration Based Elitism*. In normal elitism in evolutionary computation, the "best" one or few individuals in the population are guaranteed to survive unchanged into the next generation. Typically the "best" individuals are defined by their fitness values. In Orchestration Based Elitism, the "best" individuals, for elitism purposes, could be treated to be those programs that won whatever orchestration strategy was currently in use. If what the system really cares about is getting the best orchestrated group, then perhaps it is those programs that win the orchestration search that should most certainly be preserved. Since PADO does not use elitism, this comparison would be extraneous here but is certainly worth pursuing in future work.

Mixture of Experts

Past work certainly exists on combining multiple models learned for the same purpose so that their combined response is, on average, better than that of any one of the models (e.g., [Littlestone, 1988, Littlestone and Warmuth, 1994, Blum, 1995, Bielak, 1993a]).

There are two main levels of orchestration in PADO. At the lower level (F_L), multiple programs can be chosen from a single discrimination pool and their responses orchestrated to obtain a single response for that discrimination-solution. Previous work in PADO (e.g., [Teller and Veloso, 1997]) has demonstrated that this level of orchestration can be effective. In addition, the fact that evolution is used as the learning force in PADO provides a ready source of programs that orchestrate at this level. However, both because PADO can function when a single program is selected from each discrimination pool, and because finding the right mixture of experts is not the primary work of this thesis, this aspect of PADO was dropped from the thesis instantiation of PADO as described in the previous chapter. Future work in this area should certainly include the incorporation of other work on mixtures of experts into this lower level PADO orchestration.

The second level of PADO (F_H) orchestrates the different discrimination-solutions (independent of how many evolved programs make up each of those discrimination-solutions). At this level, much of the work on combining the responses of multiple experts is not applicable because these discrimination-solutions are not responding to the same question. This is more a situation of divide and conquer, in which this second level of orchestration is the conquer step after the divided sub-problems have been solved. PADO does address this problem and orchestration at this level is the subject of this chapter.

4.2 Experimental Comparisons

4.2.1 Example Domain

This thesis devotes an entire chapter (Chapter 7) to demonstrating the range and domain of PADO. There are also a number of issues in this thesis for which empirical comparisons are useful. For these comparisons, a manufactured domain presented in this section is used throughout the rest of this thesis. This section presents that domain here because the first experiment using this domain takes place in Section 4.2.2.

Before we present the domain, let us examine the criteria for a good domain in which to test many of the claims of this thesis. This domain should be:

- A domain in which a near perfect solution can often be reached within the generation bound.
- A domain in which a near perfect solution cannot be reached without a reasonable amount of learning.
- A domain in which a near perfect solution is impossible without reference to multiple, local signal aspects. In other words, it should be impossible to entirely solve the problem by the global applications of the user provided parameterized signal primitives (or by any single local application of the same).
- A domain in which there is sufficient complexity to highlight many of the distinctions the thesis will bring up.
- A domain that is simple enough that we can have high confidence that it is entirely understood. This is important because in real world domains, it is nearly impossible to account for all empirical data oddities that are caused by unexpected characteristics of the domain itself.

The manufactured domain this thesis uses has four classes. An example signal from each of the four classes, from both the training and testing sets is shown in Figure 4.3.

As you can see in Figure 4.3, the distinguishing signal feature in this domain is *slope*. A high level correct classifier of the four classes is:

> Class 1 has a positive slope all the way through. Class 2 has negative slope for the first half of the signal and then a positive slope for the second half. Class 3 has a negative slope all the way through. Class 4 has a positive slope followed by a negative slope.

Each signal is a vector of 256 values, each of which is an integer between 0 and 255. These are the sources of noise in this domain:

- The slopes vary uniformly within the ranges [-0.156,-0.078] and [0.078,0.156]
- The midline of the signal (mean of all signal values) before noise is added, varies uniformly within the range [75,190]
- Each point of the signal varies uniform randomly from the "correct" line (defined by the midline height and the chosen slopes) by an offset in the range [-25,25]

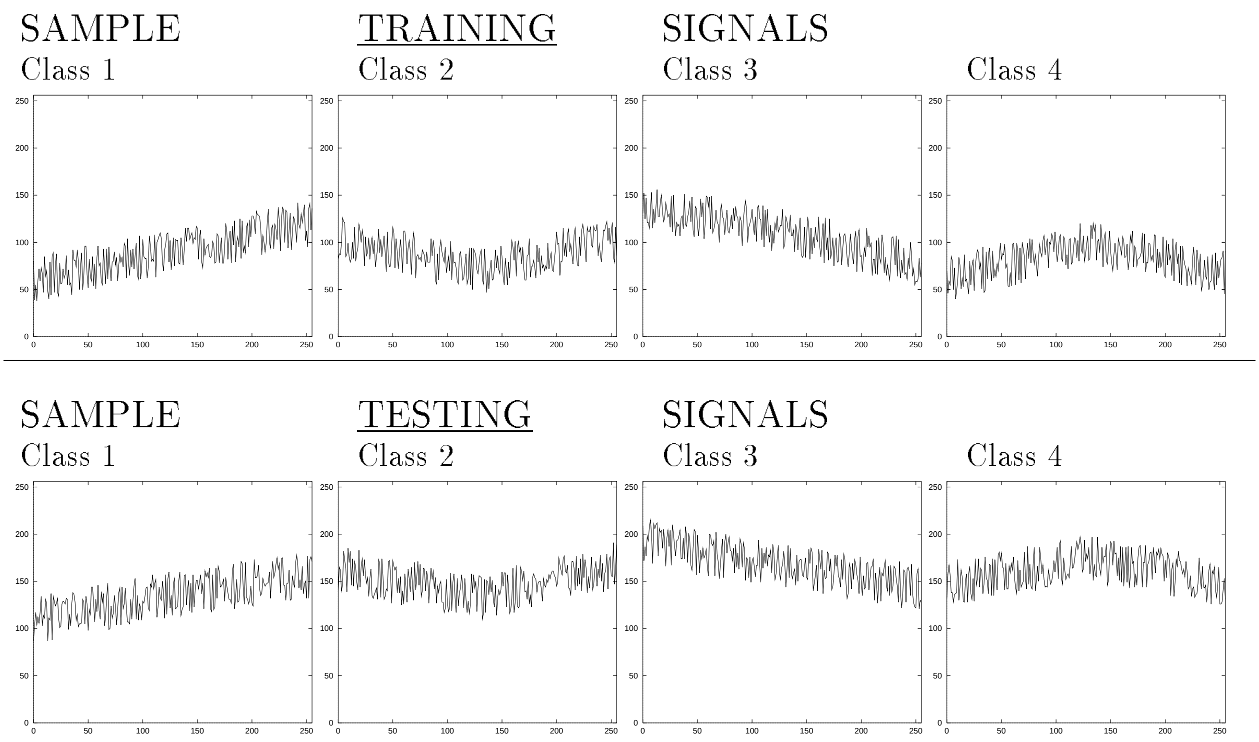


Figure 4.3: Example testing and training signals from the example domain.

The difficulty of the domain is, of course, largely dependent on the PSPs given to the evolving programs. Imagine a PSP called SLOPE(x, y) such that SLOPE(x, y) returned the average slope of the signal segment beginning with Signal[x] and ending with Signal[y]. Such a PSP would obviously be very useful. With such a function at its disposal, figuring out if a signal was from class 3 could be as simple as: **OUTPUT** $\leftarrow$ **((SLOPE(0,128)** $>$ **0) and (SLOPE(128,256)** $<$ **0))**.

In order to make the problem a little more difficult, **SLOPE(x, y)** will not be provided as a PSP. The two PSPs provided to PADO for this particular domain are:

PSP-POINT(x) this function takes a point on the signal and returns the value/height of the signal at that point.

PSP-AVERAGE(x,y) this function takes a region on the signal and returns the average value/height of the signal in that region.

Notice that estimating the slope using *PSP-POINT(x)* will take many function calls given the amount of noise in the system. Estimating the slope for half the signal using *PSP-AVERAGE(x,y)* can take as few as two PSP-AVERAGE function calls, but these must be carefully chosen. If PSP-AVERAGE(x,y) is applied to more than about 1/4 of the signal, it becomes useless for this domain.

Unless otherwise specified, all experiments described with this signal used a training set size of 200 signals and a testing set size of 500 signals. Because signals from this domain can easily be generated, these sets are generated randomly for each run to further smooth out the effects of randomly choosing an abnormal sample for either the training or testing set. This domain will be referred to, for the remainder of this thesis, as the *Generic Signal Domain*.

4.2.2 Experiments with Orchestration

The experiment discussed in this section is the four class classification problem, the *Generic Signal Domain*.[6]

The two important experimental questions to ask are "Do any of these learned orchestration strategies perform more *efficiently* than the fitness proportionate orchestration strategy?" and "Do any of these learned orchestration strategies perform more *effectively* than the fitness proportionate orchestration strategy?" For this thesis, these two terms will be understood to mean:

efficiency the mean time (usually measured in generations) it takes the strategy to reach a particular level of generalization performance.

effectiveness the mean level of generalization performance reached by a particular time threshold (usually MAX-GENERATIONS).

In Figure 4.4, we can see that there is indeed something to be gained by learning a better orchestration. The *weight-search orchestration* performs approximately 10% more efficiently than the fitness proportionate orchestration and the *program-search orchestration* performs approximately 20% more efficiently than the fitness proportionate orchestration. These results make sense as there is only so much additional information that can be gained within the context of this linear orchestration.

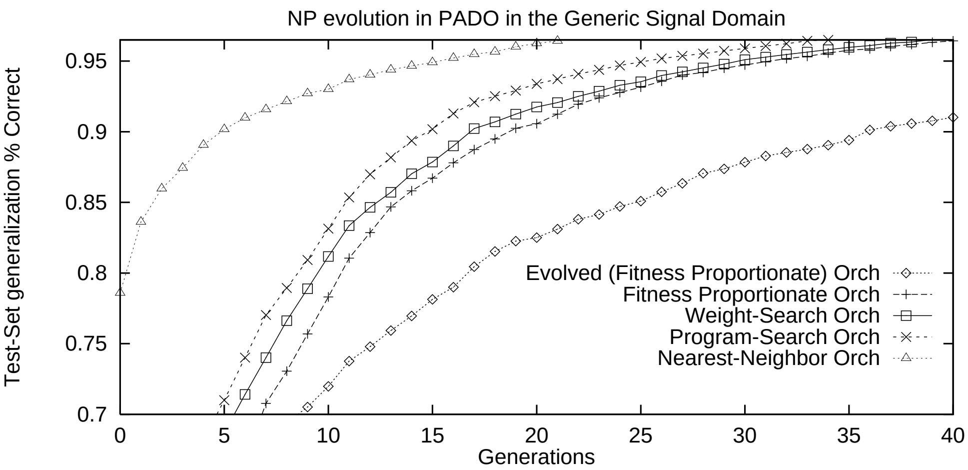


Figure 4.4: Comparison of the efficiency of four different orchestration techniques.

[6]This section explores differences in PADO orchestration using the NP representation and the Internal Reinforcement procedures. These are described in detail in later chapters and the results of this section are almost entirely independent of whether IRNP is on or off.

The *nearest-neighbor orchestration*, however, learns twice as efficiently as the fitness proportionate orchestration, making it clear that there are often non-linear interactions between these discrimination-solution responses that do convey additional information. It is worth noting that all of these orchestration options incur a negligible computational cost relative to the cost of evolving multiple discrimination-solutions.

The *evolved orchestration* is actually **less** efficient than doing no learning. This is such an important (and oddly positive) result that an explanation will be saved for Section 4.2.3.

In Figure 4.5, we can see that orchestration affects not only how quickly PADO can learn to generalize to a certain performance level, but that it also affects the generalization performance that PADO achieves in the end. The fitness proportionate orchestration is the control for this study and therefore the standard against which the other methods are to be compared. In this particular domain, the *weight-search orchestration* reduces the generalization error by about 10% by generation 40. Similarly, the *program-search orchestration* reduces the generalization error by about 20%. Again, because these two strategies are forced to explore within the context of the function "weighted voting," there is only so much they can do to improve upon the fitness proportionate orchestration.

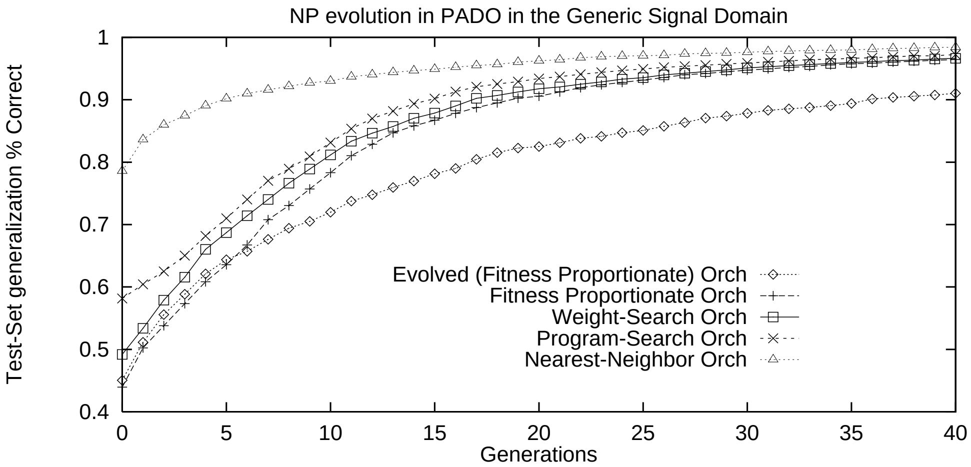


Figure 4.5: Comparison of the effectiveness of four different orchestration techniques.

For this problem, the *nearest-neighbor orchestration* reduces the generalization error by almost a factor of 3 (which in this case is a reduction of about 3% of the error).

The *evolved orchestration* is actually **less** effective than doing no learning. This is caused by the same effect alluded to above and will be explained in full in Section 4.2.3.

The practical conclusion of this section is that the nearest-neighbor orchestration is the best of the options put forward by this chapter. As will become clear later in the thesis, the orchestration strategy used can have an effect on other aspects of PADO other than just the effectiveness and efficiency. For that reason the rest of this thesis will report results using both the *nearest-neighbor orchestration* and the *weight-search function*[7]

[7]Because they're fairly close in performance, *weight-search orchestration* was chosen over *program-search orchestration* simply because it can be computed slightly faster.

4.2.3 Discussion

Evolved Orchestration, as it was defined in Section 4.1.2, adjusted the fitness values of the programs in the population to reflect their ability to "generally orchestrate," and then it applied the fitness proportionate orchestration with those new fitness values. Keep in mind that while this adjustment of fitness value affects the course of evolution, it does not immediately change anything but the relative fitness values of the programs.

Evolved Orchestration could just as well have been defined such that after the fitness adjustment, any of the other learning based orchestration strategies could have been used. Before the explanation for how this fitness adjustment hurts PADO's performance (and why), Figure 4.6 shows all four of the orchestration strategies and their evolved orchestration counterparts. Notice that in *all four cases*, the evolved orchestration fitness adjustment hurts PADO's performance.

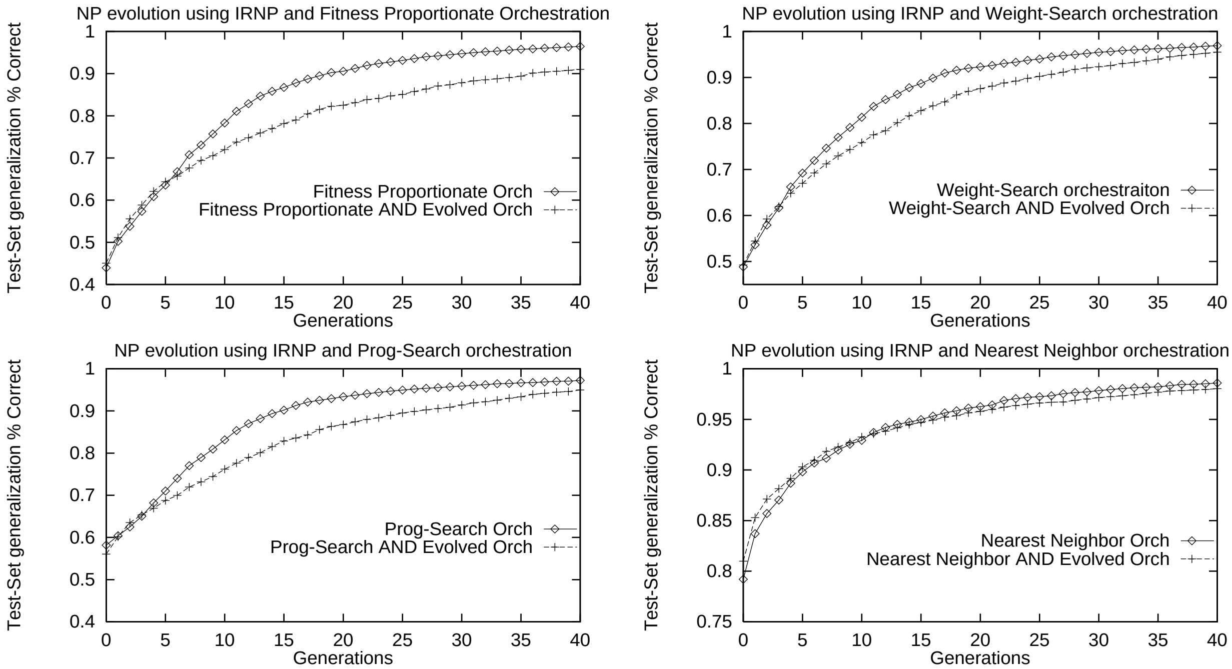


Figure 4.6: Comparison of the effectiveness of four different orchestration techniques with and without *evolved orchestration*.

Here is an explanation for why putting the orchestration directly into the fitness function actually *hurts* PADO's performance. PADO begins by sub-dividing its population, and thereby the problem, into several easier sub-problems. This subproblem division is reasonable for classification and PADO enforces it throughout the entire run. The chapter has shown that the orchestration phase of PADO is not trivial, but because it is tractable, there is more to be lost than gained by removing the useful constraint on the algorithm discovery phase of PADO: only solve a particular discrimination problem.

When fitness gets tied to orchestration instead of discrimination, *PADO loses exactly these constraints* and ends up searching a much larger space. So we argue that PADO may have gained something through evolved orchestration, but *at the cost of losing the whole mechanism of divide and conquer* that made orchestration important in the first place.

The major characteristics of the curves in Figure 4.5 are that the evolved orchestration does worse than the fitness proportionate orchestration, the Weight-Search and Program-Search orchestration methods do only mildly better than the fitness proportionate orchestration, and the Nearest-Neighbor orchestration method is markedly more effective and markedly more efficient. None of these effects seem to be empirically specific to this domain. The same effects, with similar magnitudes, have been observed in two other different domains. This section has already addressed why the evolved orchestration is less successful than any orchestration method that does not include it. The other two major characteristics just mentioned also deserve discussion.

The reason that the Weight-Search and Program-Search orchestration methods are only mildly better than the fitness proportionate orchestration is that the fitness proportionate orchestration is a very good strategy, given the constraints of a "weighted max vote" orchestration paradigm. As has been noted, since this orchestration paradigm is linear, the default weights (a direct function of the fitnesses of the programs chosen) are quite often exactly the right weights to have. So in reality, about 10% of the runs evolve programs that can be orchestrated noticeably better with the Weight-Search or Program-Search strategies. But since each point in the curves in Figure 4.5 is an average over many runs, and since these two strategies were of negligible value in about 90% of those runs, the curves express this as a mild average performance improvement.

The explanation for the success of the Nearest-Neighbor strategy is exactly the opposite. The Nearest-Neighbor orchestration strategy discards the linear orchestration assumption and in so doing takes advantage of the non-linear relationships between the program responses on particular signal inputs. As will be addressed in Chapter 7, the advantages of the Nearest-Neighbor strategy are diminished in domains where there is very little training data. This is so because the Nearest-Neighbor orchestration strategy needs to fill its space with labeled training points and when there are few training points to use, the nearest-neighbor space has low density and thus less non-linear information of which to take advantage.

Chapter 5

Neural Programming

5.1 Introduction

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. In this chapter, we introduce "Neural Programming," a connectionist representation for evolving parameterized programs. Neural Programming allows for the generation of credit and blame assignment in the process of learning programs. We further introduce "Internal Reinforcement" as a general informed feedback mechanism for Neural Programming. We present the Internal Reinforcement process and demonstrate its increased learning rate through empirical results.

This chapter introduces a new machine learning representation for complex programs. This new representation, *Neural Programming* (NP), has been developed with the goal of incorporating a principled updating procedure into genetic programming (GP). Neural Programming is a connectionist programming language that has been designed to make *internal reinforcement*, until now unaccomplished in genetic programming, possible.

Genetic programming is a successful representative of the machine learning practice of *empirical credit-blame assignment* [Angeline, 1993]. Empirical credit-blame 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-blame assignment*. In explicit credit-blame 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. The goal of this work is to bridge this credit-blame assignment gap by finding ways in which explicit and empirical credit-blame assignment can find mutual benefit *in a single machine learning technique.*

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 how to work complex functions into the middle of an otherwise homogeneous network of simple non-linear functions is far from obvious. Yet gradient-descent learning procedures like backpropagation in ANNs are an extremely powerful idea. Backpropagation is not only a kind of 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 this *internal reinforcement* is to provide a reasoned method to guide search in the field of program induction. When hill-climbing in any space, it is always possible to sample a few points and then choose 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 it can work much more effectively with *internal reinforcement*. We introduce internal reinforcement as a program evolution approximation to the program space gradient function. As will become clear in the following chapter, internal reinforcement is a partial, not complete solution to this problem.

In short, in GP, it would be desirable to be able 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).

This chapter contributes the Neural Programming representation as a new, connectionist representation for evolving programs. In Chapter 6, 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 Chapter 7, we demonstrate the effectiveness of both the representation and its associated internal reinforcement strategy through several experiments on illustrative signal classification problems.

5.2 The NP Representation

The essence of a programming language is one or more basic constructs and one or more 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 incorporated into 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 perform a **flow of data**, rather than the flow of control as in typical programming languages. The nodes in a neural program can compute arbitrary functions of the inputs. So a node can still 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 of neural programs are given later in this chapter. Table 5.1 enumerates the important characteristics of the NP representation.

• 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 thesis OUT is a simple weighted average. Each value is weighted by the timestep it appears on.[a] [a]More detail on this can be found in Appendix C.3.2.

Table 5.1: The critical characteristics of the NP representation.

Throughout this thesis, 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) is a piece of code written by a user of PADO that expresses a way of extracting information of the input signal in a parameterized form. Example PSPs 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 preprocessing. As has been discussed already, the salient distinction here is the parameterization of input "features."

The topology of NP programs is not rigidly fixed to the semantics of the function each node happens to execute. This means that, for example, a node that only "needs" two inputs (e.g., a node executing the function DIVIDE) may have more input arcs than it can use (e.g., it has four input arcs and simply ignores two of them) or have fewer than it can use (e.g., it has only one input and so returns some default value).[1]

Each NP node may have many output arcs. (See Figure 5.2 for a simple example.) The multiple forked distribution of good values from any point in the program is a valuable aspect

[1]Implementation detail: the arcs are numbered in increasing chronological order of their connection to the node and the input parameters of a function simply grab arcs to treat as inputs starting with the earliest connected arc.

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. GP program representations can profit by incorporating this fan-out which is an advantage of connectionist representations.

In a data flow machine in which function evaluation at the nodes is instantaneous, there are two distinct options for computing the output of a node from its inputs. The first is that all nodes act simultaneously on the outputs generated on the previous timestep. In other words, there is no order to the evaluations of nodes in a program: they evaluate in parallel. In the second case, the nodes are evaluated in a particular order, so that if $D_{x,t}$ indicates the evaluation of node x on timestep t, the evaluation order is $D_{0,0}$, $D_{1,0}$, ..., $D_{N_p,0}$, $D_{0,1}$, $D_{1,1}$, ... $D_{N_p,T}$ where N_p is the number of nodes in the hypothetical program p. For illustrative purposes, in the examples below we assume that the NP programs are evaluated according to the first rule: all nodes evaluate in parallel. In point of fact, PADO is implemented with the second variety of data flow.

Given that there is a timestep threshold imposed on the evolving programs, 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 to these questions are provided in Section 7.5.

There are two dominant forms of change that evolving programs typically undergo: crossover and mutation. Mutation is the change of one (usually atomic) part of the program to another aspect of the same type. Crossover is the sexual reproduction of two programs where two programs "mate" by exchanging program material between them.

> *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.*

5.3 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.

5.3.1 Example 1: The Fibonacci Series

Figure 5.1 shows an extremely simple NP program. This program computes the Fibonacci series and sends each successive element out of the program fragment on Arc4.

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

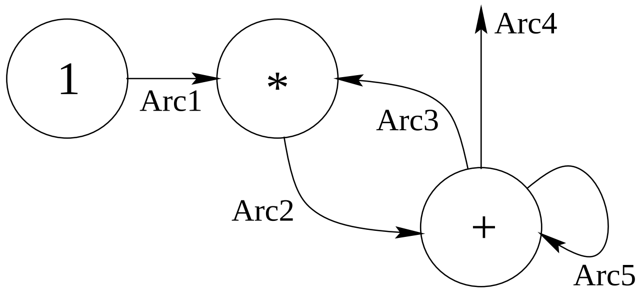


Figure 5.1: A simple NP program that computes the successive elements of the Fibonacci series. *All input/arc values are 1 on the first time step.*

the value 1 when a program starts up. Table 5.2 shows how the values of the arcs change over time.

Step	Arc 1	Arc 2	Arc 3	Arc 4	Arc 5
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
...					

Table 5.2: Progression of arc values over time for the simple NP program shown in Figure 5.1.

5.3.2 Example 2: The Golden Mean

Let us now change slightly the computation of the simple NP program from example 1. Instead of outputing a list of exponentially increasing values (as in the program shown in Figure 5.1) let us design an NP program that approximates the "Golden Mean"[2] 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) $\mathrm{fib}(i)$ and $\mathrm{fib}(i-1)$ as they are computed (shown in Figure 5.2).

Table 5.3 shows how this computation plays out through the arcs as the timesteps pass.

5.3.3 Example 3: Foveation

Foveation is the process that changes 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) only fills about 5% of our field of view.

[2]The golden mean is $\frac{1\pm\sqrt{5}}{2}$. This example program approximates $\frac{1+\sqrt{5}}{2}$. Note that $\lim_{n\to\infty} \frac{\mathrm{fib}(i)}{\mathrm{fib}(i-1)} = \frac{1+\sqrt{5}}{2}$.

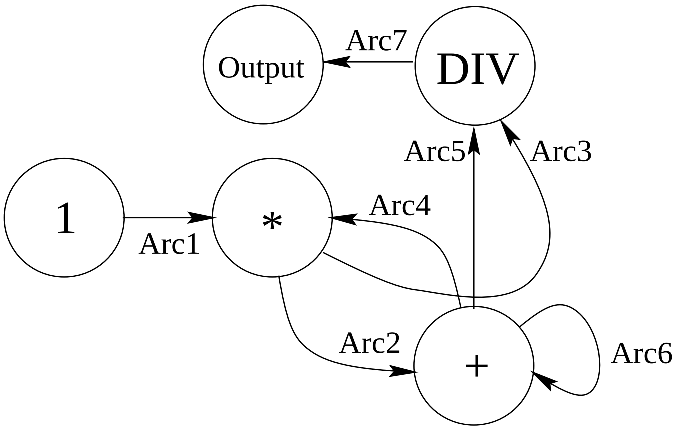


Figure 5.2: A simple Neural Program that iteratively improves an approximation to the golden mean. *This program assumes that all input values are* **1** *on the first time step.*

Step	Arc 1	Arc 2	Arc 3	Arc 4	Arc 5	Arc 6	Arc 7	OUTPUT	Golden Mean
0	1	1	1	1	1	1	1	NA	1.618034
1	1	1	1	2	2	2	1	1	1.618034
2	1	2	2	3	3	3	2.0	1.5	1.618034
3	1	3	3	5	5	5	1.5	1.5	1.618034
4	1	5	5	8	8	8	1.6667	1.5417	1.618034
5	1	8	8	13	13	13	1.6000	1.5533	1.618034
...									

Table 5.3: Progression of arc values over time for the simple NP program shown in Figure 5.2.

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 user-defined PSP 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 5.3 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. This means that PSP-Variance, at each time step, computes its function

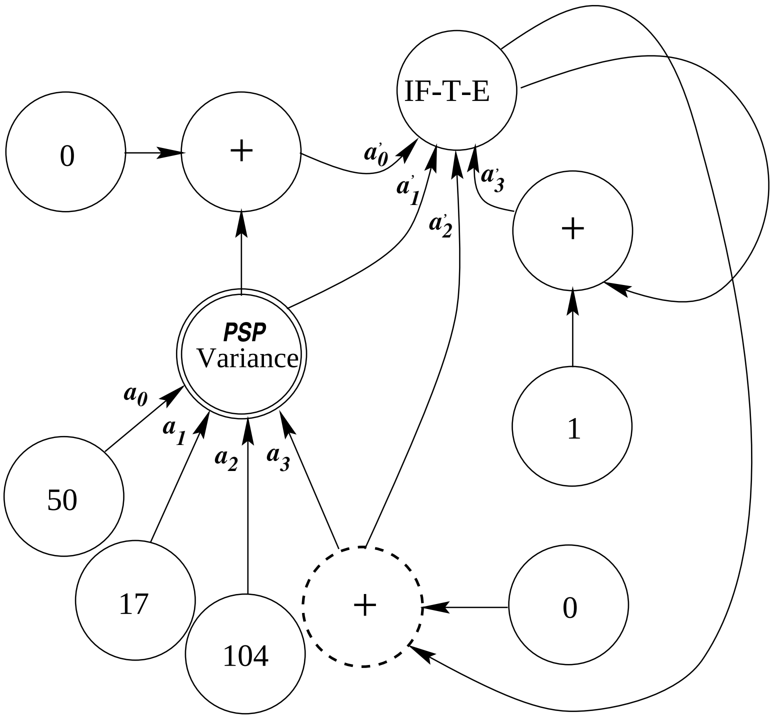


Figure 5.3: A simple NP program fragment. The output value from the dashed circle node is being iteratively refined to minimize the value returned by the PSP-Variance node.

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 Table 5.4). Note that IF-T-E (If-Then-Else) is the function "if ($a_0 < a_1$) then return a_2 else return a_3." Assuming again that all arcs are initialized to 1, this program finds a one-sided local minimum of PSP-Variance with respect 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 NP foveating: using the values it perceives to focus upon further investigation of the input in question.

VAR$_0$ = 1
VAR$_t$ = **PSP-Variance**(50,17,104,$a_{3,t}$)
IF (VAR$_{t-1}$ < VAR$_t$)
    THEN $a_{3,t+1} = a_{3,t}$
    ELSE $a_{3,t+1} = a_{3,t} + 1$

Table 5.4: Pseudo-code for the behavior of the NP program fragment in Figure 5.3. In this figure, VAR$_t$ denotes the value of VAR at time t, and $a_{3,t}$ denotes the value of argument a_3 at time t.

5.4 Evolving Subroutines

Each node may have multiple outputs. Since one of the advantages of substructure to evolving programs is the availability of regularity [Koza, 1994], this opportunity to spread computed values to multiple locations simultaneously is an alternative to explicit subroutines in evolving programs.

Here is another way to understand the value of this kind of programmatic fan-out. In biology, the process of *branch duplication* takes place in which a gene (or at least a region of DNA) is duplicated within a chromosome. This redundancy is almost never harmful because these regions of DNA encode for proteins, and encoding for them more than once does not reduce the abundance of these proteins. Now, however, biological mutation has the opportunity to change one of these regions encoding for a new, "experimental" protein without disrupting the supply of the original, useful protein. In the same way, if there is a particularly good "idea" present in an NP program, this "idea" (i.e., series of output values from a node) can be distributed, through the multiple output arcs, to a number of different parts of the program.

Further, an additional output arc can be added to a node that has been identified as the source of one of these "good ideas." This node can lead into an unused or under-used part of the program. This area of the program now has the opportunity to create some added value using this input. If it does, learning has succeeded; the performance has improved. If it does not succeed, then no harm has been done to the program because the "good idea" is still being used correctly in other parts of the program. Notice that this "parallelization" of evolution (i.e., independence of sub-parts of the program) is accomplished because NP programs are data-flow rather than control-flow programs and because there can be multiple OUTPUT nodes in a single NP program.

Our work as reported in this thesis contributes a specific technique to identify these "good ideas" (Section 6.2.1). Section 6.3 makes explicit how unused or under-used areas of a program can be identified and exactly how the "good idea" node and the unused/under-used program areas can be connected. For completeness, however, it is worth mentioning at least one method for adding explicit hierarchy into the NP paradigm. The only insight required is that *each node computes a function and that that function can be co-evolved.* Figure 5.4 below pictures this embedding process.The general process of co-evolution of subroutines is dealt with in detail in a number places (notably [Koza, 1994]). These same pieces of wisdom are equally applicable to evolving hierarchy within a single NP program.

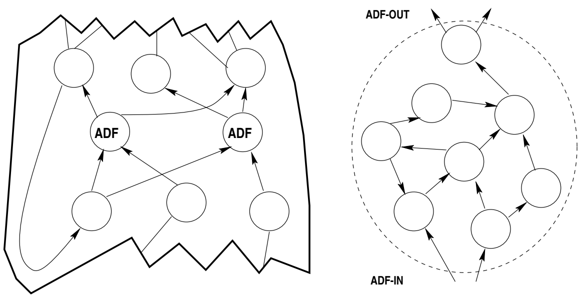


Figure 5.4: How ADFs (automatically defined functions) could be implemented in NP.

Chapter 6

Internal Reinforcement in NP

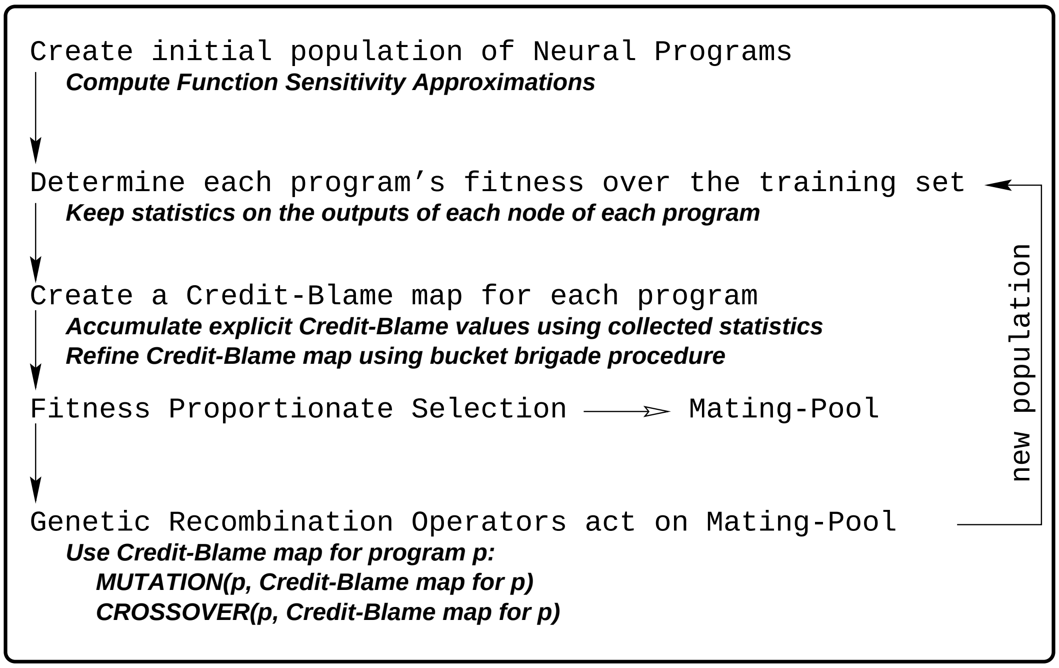
6.1 Introduction

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*. Programs need to become new programs. These program transformations have a specific goal which is to produce programs that are better, which is to say programs that 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 EC. 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 thesis: *internal reinforcement*.

Now that we have introduced the neural programming representation, we can describe a mechanism to accomplish *internal reinforcement*. There are two main stages in Internal Reinforcement of Neural Programs (IRNP). 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.

Table 6.1 shows the evolutionary learning process for NP and how IRNP fits into that picture. One Credit-Blame map is created *for each program in the population* and when the time comes to perform genetic recombination (the search method in EC) on a particular program, the Credit-Blame map *for that particular program* is used.



IRNP additions to EC

Table 6.1: The high level flow of NP learning.

6.2 Creating a Credit-Blame Map

As was described in Section 3.2, 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). Therefore, let us consider an abstract input to output mapping to be learned by the neural programs.

6.2.1 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[1] all the values node x outputs into a single value H_x^i. 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 6.1.

[1]The compression function used in this thesis is *mean*.

$$\mathrm{CS}_x = \left| \frac{E(\vec{H}_x - \mu_{\vec{H}_x}) * E(\vec{L} - \mu_{\vec{L}})}{\sigma_{\vec{H}_x} * \sigma_{\vec{L}}} \right| \tag{6.1}$$

This credit score for each node is an indication of how valuable that node is to the program. It is certainly 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 definition*, 100% correct if it has a node with a credit score of 1 and that node is connected to an output node[2]. 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 large 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. $\mathrm{CS}_z = 0.97, \mathrm{CS}_x = 0.14, \mathrm{CS}_y = 0.07$). 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[3] 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, (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 6.2.3.

6.2.2 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,

[2]This is true under the condition that there are no other arcs in the program that terminate in an OUTPUT node.

[3]We can say that nodes x and y are partly responsible for the credit score node z receives because, by definition, the output of the function XOR is dependent on its inputs and CS_z is, by definition, dependent on the output of XOR. The reason we do not say that nodes x and y are entirely responsible for CS_z is that the function at node z is also an important factor (over and above the inputs node z receives from nodes x and y) in determining the value of CS_z.

"What is the responsibility of each input parameter for the output value produced by each function?"

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 two questions that function sensitivity approximation can automatically answer about a black box function's relation to its inputs are "How many and how few parameters can it take (min and max arity)?" and "How sensitive is the output value to changes in its inputs?" This discovered sensitivity is a substitute to the derivative of the function in question.

Let us say that the sensitivity of some function f with respect to one of its arguments (call that argument 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 (e.g., [-1000,1000]).

Before describing this discovery of a function's input sensitivity, the issue of nondeterminism must be visited. Does the proposed technique for investigating "black box" functions still work if some of the functions are nondeterministic? The good news is "yes." PSP-Variance is such an example (see Section 5.3.3). PSP-Variance takes four input parameters and returns the variance of pixel intensity in the rectangular region described by the four parameters. PSP-Variance(10,15,101,219) is an image region that contains 18,564 pixels. A fairly accurate value for the pixel intensity variance in that region can be achieved by sampling a small fraction of those 18,564 pixels.

Table 6.2 shows the simple process for finding the sensitivity of each parameter of a general, possibly nondeterministic function f. The procedure in Table 6.2 is performed for all values of A between 1 and the maximum arity of the function. We do not have to determine this maximum arity ourselves. The key insight is that, finding the sensitivity gives us the minimum and maximum arities for each function since, for example, the maximum arity is, by definition, that parameter number above which further parameters have a sensitivity of zero. Nondeterminism can also be handled by this process and is adjusted for through the calculation of "Noise" as shown in Figure 6.2.

As an example, consider the function "ADD" for which the user introduced a ceiling so that any set of numbers that sums to a number greater than that ceiling effectively sums to exactly that ceiling[4]. Not having given it a lot of thought, we would have simply described all the parameters of ADD as equally important (which is true within the sampling error) and all 100% sensitive to changes in any of those parameters. However, when running our function sensitivity approximation procedure as we introduced, we find that this is not true. Table 6.3 shows the values returned by the procedure. It makes sense (after looking at line four of Table 6.3) that with four parameters that vary randomly in the legal range of values, a random change in one of those parameters has only a 6.9% chance of affecting the value returned by this application of the function "ADD."

[4]Implementation detail: specifically the legal range of values used in this thesis was [0..256]. This naturally caused the ADD ceiling enforced to be the value 256.

Noise := 0; Sensitive := 0;
DO Q_s **times**
  Let A be the arity of function f
  Let $\vec{a}$ be the input vector for function f
  Pick uniform random values $a_1, a_2, ..., a_A$ for $\vec{a}$
  $\text{Result}_0 := f(\vec{a})$
  $\text{Result}_1 := f(\vec{a})$
  Change $\vec{a}$: parameter $a_i \leftarrow$ random value
  $\text{Result}_2 := f(\vec{a})$
  If ($\text{Result}_0 \neq \text{Result}_1$) Noise := Noise + 1
  If ($\text{Result}_0 \neq \text{Result}_2$) Sensitive := Sensitive + 1
$\mathcal{S}_{f,A,i} := \frac{\text{Sensitive}-\text{Noise}}{Q_s}$

Table 6.2: Function Sensitivity Approximation: the process for finding $\mathcal{S}_{f,A,i}$, the sensitivity of a particular parameter a_i for some function f that is given a parameter vector with A elements.

# Params	arg 1 sensitivity	arg 2 sensitivity	arg 3 sensitivity	arg 4 sensitivity
1	0.996312			
2	0.655752	0.659795		
3	0.246964	0.259000	0.240082	
4	0.068905	0.068309	0.064403	0.076065

Table 6.3: **ADD** (min arity is **1** and max arity is **4**)

The benefits of Function Sensitivity Approximation are particularly clear in the context of a function such as "if-then-else." IF-Then-Else is the function "if ($a_0 < a_1$) then return a_2 else return a_3". Left to figure it out for ourselves, we originally assigned the four sensitivities as (1.0,1.0,0.5,0.5). a_2 and a_3 are certainly equally important and each has a sensitivity of 0.5. The first two parameters, however, only matter *with respect to each other*. So for two random values a_0^0 and a_1^0, changing a_0^0 to some new random value a_0^1 has only a 33% chance of changing the value of the relevant test: ($a_0 < a_1$). The procedure outlined in Table 6.2 discovered this counterintuitive result automatically as shown in Table 6.4.

Function sensitivity approximation is useful exactly because it works without prior information about the function to be analyzed. This means that function sensitivity approxima-

# Params	arg 1 sensitivity	arg 2 sensitivity	arg 3 sensitivity	arg 4 sensitivity
4	0.322822	0.326114	0.503329	0.490406

Table 6.4: **IFTE** (IF ($X < Y$) Then U Else V) (min arity is **4** and max arity is **4**)

# Params	arg 1 sensitivity	arg 2 sensitivity	arg 3 sensitivity	arg 4 sensitivity
4	0.853429	0.887229	0.846371	0.875599

Table 6.5: **PSP-VARIANCE** (min arity is **4** and max arity is **4**)

tion also works on user-defined functions. The Parameterized Signal Primitive *PSP-Variance*, an example user defined function used by NP, also produces informative sensitivity values.

As just described, all functions are evaluated within the context of the training examples. This does not affect many functions (e.g., "ADD") but certainly has an effect on an input-sensitive function like "PSP-Variance." PSP-Variance requires four parameters and this is easily detected. More interestingly, Table 6.5 shows that the second and fourth parameters to PSP-Variance are slightly more sensitive to changes than are the first and third parameters. This does not "mean" anything to NP and IRNP, it is taken for the process of evolution as a fact about the world. But we can step back and see that, because PSP-Variance interprets its four inputs (x_1, y_1, x_2, y_2) as a rectangle in a video image with upper left corner (x_1,y_1) and lower right corner (x_2,y_2), this 3% additional sensitivity for the second and fourth parameters tells us that the particular images in this domain tend to be very slightly more variable along the Y-axis. In itself this does nothing to help solve the pattern recognition problem, but it is an interesting side-effect of this process of automatic discovery of function argument sensitivity.

6.2.3 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 Table 6.6. The credit scores are refined according to the network topology and sensitivity of the node functions.

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 $\mathcal{S}_{f_y,A_y,i}$ = Sensitivity of f_y (relative to A_y and i)
    $\mathbf{CS}_x = \mathbf{MAX}(\mathbf{CS}_x,\ \mathcal{S}_{f_y,A_y,i} * \mathbf{CS}_y)$

Table 6.6: The process of bucket brigading the Credit Scores (CS) throughout an NP program.

The high level structure of the procedure presented in Table 6.6 is as follows. For each node for each output arc from that node, the node's credit-score will be 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 *times* the sensitivity of that destination node to that particular output arc. We explain this process in detail through a series of questions and answers designed to identify the important elements of this procedure.

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[5] halts and typically in only a few passes.

Note that because of the way $\mathcal{S}_{f,A,i}$ is defined (see Table 6.2), parameter a_i is very occasionally replaced *by itself* and so SENSITIVITY is usually 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, 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 no 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.[6] The sensitivity discovery process described in Table 6.2 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*.

[5]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 Table 6.6). 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 $\frac{N_p}{\epsilon}$.

[6]This is of course, not entirely true. There are always some more "popular" numbers in a system, but this regularity is often caused by the *interactions* between the various functions and this is sufficiently complicated by the fact that the uniform approximation was adopted.

Finally, consider the equation for refining the credit scores: $\mathrm{CS}_x = \mathbf{MAX}(\mathrm{CS}_x, \mathcal{S}_{f_y,A_y,i} * \mathrm{CS}_y)$. "Why should CS_x be set to the maximum of itself and $\mathcal{S}_{f_y,A_y,i} * \mathrm{CS}_y$?" 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 the output of a node could be "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 its individual credit score and 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 $\mathrm{CS}_x = \mathrm{MAX}(\mathrm{CS}_x,\ F_r(\mathrm{CS}_y))$ 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 6.2.2. 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 $F_r(\mathrm{CS}_y)$; it is $\mathcal{S}_{f_y,A_y,i} * \mathrm{CS}_y$.

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.

6.2.4 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. While, on the other hand, when 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.

6.3 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 program updates 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. In brief, mutation is the process of recombination of a single genotype and crossover is the process of recombination of two or more genotypes, through genetic material exchange. Considerable information is available on these methods through sources such as [Koza, 1992].

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, 1996, Teller, 1996]). This work has reaped some success, but because of the co-evolutionary nature of the work, it has not yielded much insight into the basic mechanism of crossover. IRNP has the potential not only to improve on the existing GP mechanism, but also to help *explain* the central mystery of GP, namely crossover. This is one of the interesting directions for future work.

6.3.1 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 mentioned above. Notice that when the terms “large” and “low” are used (as opposed to the unambiguous terms “highest” and “lowest”), it 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**

1. Pick a node x with a large credit score.
2. Pick a node y with a low credit score and A inputs such that y would still be sensitive to input a_{A+1}.
3. Add an arc (x, y).

- **Delete an Arc**

1. Pick a node y with a low credit score such that y would still be sensitive to its inputs if one were removed.
2. Pick a node x with a low credit score such that there exists an arc (x, y).
3. Delete arc (x, y).

- **Swap Two Arcs** (see Figure 6.1)

1. Let x be the node with highest CS_x.
2. Let (x, y) be the output arc of x to a node y that minimizes CS_y.
3. For all arcs (u, v) such that v is an OUTPUT node, pick the arc (u, v) that minimizes CS_u.
4. Delete arcs (x, y) and (u, v), and create arcs (x, v) and (u, y).

- **Change a Node Function**

1. Pick a node x that has a low credit score and such that (x, y) exists and creates input a_i to node y and $\mathcal{S}_{f_y,A_y,i} > 0$.
2. Change the function that x computes to another function of similar or lower arity.

- **Add a Node**

1. Create a new node z with f_z, a randomly selected function.
2. Let A be the arity of f_z, and let O_z be the number of output arcs from z.
3. Find high credit score nodes x_1, ...x_A and create the arcs (x_1, z) ... (x_A, z).
4. Find low credit score nodes y_1, ... y_{O_z} such that $\mathcal{S}_{f_{y_i},A_{y_i}+1,A_{y_i}+1} > 0$ for $(1 \leq i \leq O_z)$.
5. Create the arcs (z, y_1) ... (z, y_{O_z})

- **Delete a Node**

1. Pick a low credit score node x.
2. 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 "vanilla" 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.

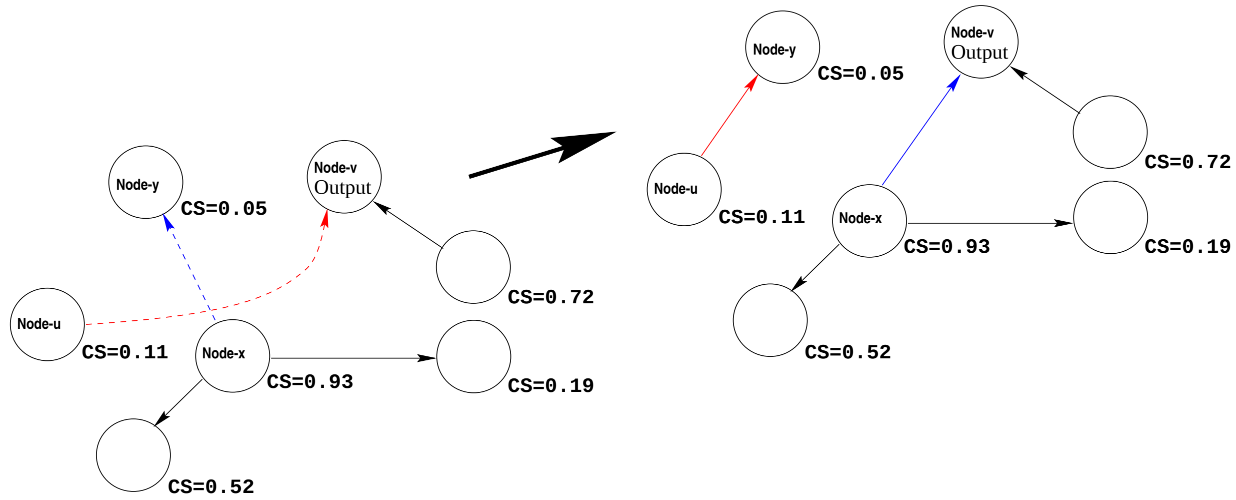


Figure 6.1: The *Swap Two Arcs* mutation procedure.

6.3.2 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 6.2 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].

We keep this underlying mechanism and present an IRNP procedure that selects "good" program fragments to exchange. This means that IRNP has, as it's 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 thesis work and contributions, this aspect of the use of credit-blame assignment has been left as future work.

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 the context of crossover we take the credit score of an arc to be the credit score of its destination 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. Table 6.7 outlines this IRNP crossover procedure.

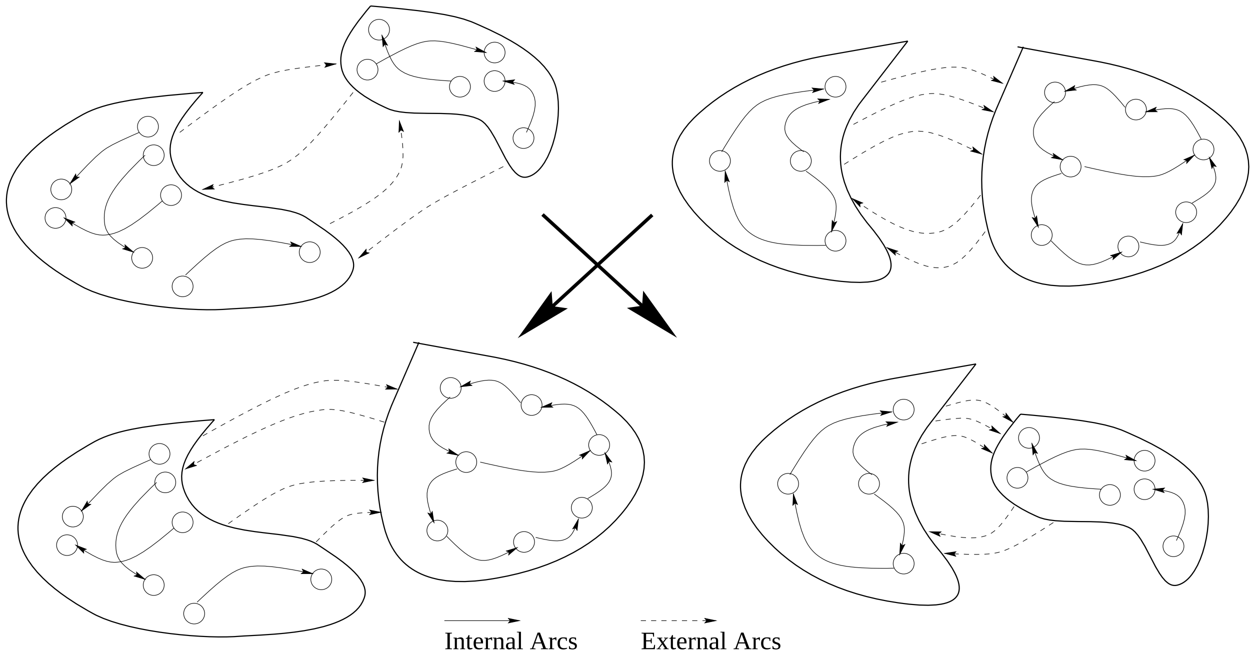


Figure 6.2: Crossover in NP: A single graph of nodes and arcs is fragmented with a cut into two fragments such that every node in the original graph is now either in Fragment_1 or Fragment_2 and every arc is either an *internal* or *external* arc.

6.4 Exploration vs. Exploitation Within a Program

This thesis has already touched on the issue of exploration vs. exploitation within the search process. A similar tension exists within the recombination of a single program. On the one hand, it seems clear that IRNP should leave alone the "best" parts of the program and focus its changes on the "worst" 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 that **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.[7]

The second problem with seeing IRNP's job as simply focusing on the "bad" parts of a program is that, 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 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 these issues. With regards to the second problem, IRNP *does*

[7]This claim assumes that there are no functions that have side-effects that affect later computation within the program. An example of such functions are the indexed memory *READ* and *WRITE* functions.

```
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
```

Table 6.7: The IRNP process for choosing a "good" fragment of a program to exchange through crossover.

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 of 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 $\rightarrow$ MULT is "less damaging" than ADD $\rightarrow$ If-Then-Else).

IRNP also addresses the node participation problem. If the Credit-Blame map had a *participation* flag for each program node, IRNP could take advantage of these flags by, for example, simply augmenting the mutation policies described in Section 6.3.1. This is exactly the case. We now describe how these flags are set and the modification to the mutation and crossover policies.

These participation flags are set using the process shown in Table 6.8.

Now that the Credit-Blame map includes these participation flags, the mutation and crossover operators can be adjusted to take advantage of them.

Here are the revised mutation procedures:

- **Add an Arc**
  1. Pick a node x with a large credit score.
  2. 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}
  3. Add an arc (x, y)
- **Delete an Arc**

```
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
```

Table 6.8: The procedure for assigning the *participation flags* to nodes in each program's Credit-Blame map.

1. 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**$_y = 1$.
2. Pick a node x with a low credit score such that there exists an arc (x, y).
3. Delete arc (x, y).

- **Swap Two Arcs** (see Figure 6.1)

1. Let x be the node with highest CS_x.
2. Let (x, y) be the output arc of x to a node y that minimizes CS_y.
3. For all arcs (u, v) such at v is an OUTPUT node, pick the arc (u, v) that minimizes CS_u.
4. Delete arcs (x, y) and (u, v) and create arcs (x, v) and (u, y).

- **Change a Node Function**

1. Pick a node x that has a low credit score such that (x, y) exists and creates input a_i for node y, $\mathcal{S}_{f_y,A_y,i} > 0$, <u>and Participation$_y = 1$</u>.
2. Change the function that x computes to another function of similar or lower arity.

- **Add a Node**

1. Create a new node z with f_z, a randomly selected function.
2. Let A be the arity of f_z and let O_z be the number of output arcs from z.
3. Find high credit score nodes x_1, ...x_A and create the arcs (x_1, z) ... (x_A, z).

4. 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 $[1..O_z]$.
5. Create the arcs (z, y_1) ... (z, y_{O_z})

- **Delete a Node**
  1. Pick a low credit score node x with Participation$_x = 1$.
  2. Remove x and arcs (x, y) and (z, x) for all nodes y and all nodes z in the program.

The process for picking good fragments to exchange in crossover now needs to be modified slightly. Table 6.9 describes this modified decision process with the modification shown in bold.

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$\in${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

Table 6.9: The IRNP process for choosing a "good" fragment of a program to exchange through crossover.

6.5 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 6.2.3 actually does spread the credit score values out to aspects of the program that previously were not rewarded. This section illustrates the spreading of this value using a real snap-shot during the IRNP in a normal PADO run. Table 6.10 shows a typical (though small) NP program from Generation 8 of a run learning to classify signals from the generic signal domain. Appendix E shows this program including its arcs.

Table 6.10 shows the Credit Scores for each node at an intermediate stage in the credit-blame assignment process as described in this chapter. Namely, the credit scores shown in Table 6.10 have undergone the process of Section 6.2.1, but not the process of detailed in

Node	CreditScore	Function	Node	CreditScore	Function	Node	CreditScore	Function
0	0.000000	060	1	0.000000	Clock	2	0.000000	144
3	0.000000	Clock	4	0.000000	211	5	0.000000	Clock
6	0.000000	094	7	0.000000	182	8	0.000000	145
9	0.000000	165	10	0.000000	182	11	0.000000	045
12	0.000000	036	13	0.206935	Output	14	0.256266	PSP-Pnt
15	0.000000	Divide	16	0.697369	Output	17	0.130193	Add
18	0.196399	PSP-Max	19	0.357697	Add	20	0.000000	Multiply
21	0.331573	Output	22	0.225413	PSP-Pnt	23	0.039100	Multiply
24	0.314305	Subtract	25	0.000000	If-T-E	26	0.225413	Multiply
27	0.238077	If-T-E	28	0.000000	Split	29	0.091500	PSP-Max
30	0.735100	Split	31	0.433461	Subtract	32	0.120807	Add
33	0.000000	Subtract	34	0.433530	PSP-Pnt	35	0.004648	Add
36	0.000000	Multiply	37	0.000000	Add	38	0.282299	PSP-Max
39	0.225413	Multiply	40	0.000000	If-T-E	41	0.016246	Add
42	0.699286	Output	43	0.331573	Subtract	44	0.000000	Add
45	0.000000	Add	46	0.000000	Add			

Table 6.10: A sample NP program (without the arcs) at the end of Generation 8 of evolution, **after** the Credit Scores have been assigned, but **before** the bucket brigade refinement has taken place. These are the explicit credit scores.

Section 6.2.3. Table 6.11 shows this same NP program after the bucket brigade refinement process has taken place.

The bold faced credit scores in Table 6.11 are those values that changed during the bucket brigade credit score refinement 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 refinement 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 in itself the highest correlation node in the program.

6.6 IRNP and Indexed Memory

It should have been made clear by this point in the thesis that NP programs are nearly Turing complete in that they have a sufficiently complex function set, memory, and iteration.[8] 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 which will be used throughout the rest of this thesis, 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 restricted ranges of these values. This is *implicit memory use* (i.e., memory use through the representation itself) rather than *explicit memory*

[8] See Appendix B for complete details.

Node	CreditScore	Function	Node	CreditScore	Function	Node	CreditScore	Function
0	0.000000	060	1	0.000000	Clock	2	0.000000	144
3	**0.668100**	Clock	4	0.000000	211	5	**0.118573**	Clock
6	**0.432689**	094	7	**0.085486**	182	8	**0.001133**	145
9	**0.085897**	165	10	**0.166522**	182	11	**0.694260**	045
12	**0.162969**	036	13	**0.253838**	Output	14	0.256266	PSP-Pnt
15	**0.039802**	Divide	16	0.697369	Output	17	**0.640655**	Add
18	**0.685729**	PSP-Max	19	0.357697	Add	20	0.000000	Multiply
21	**0.681141**	Output	22	**0.667553**	PSP-Pnt	23	0.039100	Multiply
24	**0.685848**	Subtract	25	0.000000	If-T-E	26	**0.250230**	Multiply
27	0.238077	If-T-E	28	**0.425428**	Split	29	0.091500	PSP-Max
30	0.735100	Split	31	0.433461	Subtract	32	0.120807	Add
33	0.000000	Subtract	34	**0.433530**	PSP-Pnt	35	0.004648	Add
36	**0.263743**	Multiply	37	**0.685487**	Add	38	0.282299	PSP-Max
39	**0.669771**	Multiply	40	0.000000	If-T-E	41	**0.655078**	Add
42	0.699286	Output	43	**0.683490**	Subtract	44	**0.643737**	Add
45	**0.671457**	Add	46	**0.000158**	Add			

Table 6.11: A sample NP program (without the arcs) at the end of Generation 8 of evolution, **after** both the Credit Scores have been assigned, **and after** the bucket brigade refinement has taken place. *The values in bold are the values changed by the refinement process.*

use.

Indexed Memory [Teller, 1994a] 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-ARRAY[O_0]. WRITE(O_0,O_1) returns the value stored in MEMORY-ARRAY[O_0] and has the side-effect of updating MEMORY-ARRAY[O_0] to it's new value: O_1. Indexed Memory has been extensively studied in GP (e.g., [Teller, 1994a, Andre, 1995, Langdon, 1995, Langdon, 1996]) and has demonstrated itself to be a valuable form of memory use for evolving programs. As such, this thesis is enhanced by a momentary diversion to talk about the incorporation of such an explicit memory use in the context of IRNP.

IRNP makes an assumption that is not completely general which is that none of the functions have side-effects that affect later computation or the eventual response of the program. An example of such a function that violates this assumption is the function WRITE(O_0,O_1) since it has a side-effect which is to alter the program's indexed memory structure. The natural question is "can IRNP still operate with these functions even though they violate a basic assumption built into IRNP?"

In order to answer this question, we can perform a controlled experiment. We will use the generic signal domain first described in Section 4.2.1 and the problem will be the same: "classify to which of the four classes a particular signal belongs." We will simply add two new functions to the function set used in this domain: READ(O_0) and WRITE(O_0,O_1). These two functions will act upon an indexed memory array of 10 cells.

It is a well known phenomenon in evolutionary computation that adding functions to the function set available to an evolving system often has a dramatic effect on the system's performance [Andre and Teller, 1996]. So, the control for this experiment will be a set of runs in which, to the usual function set we add the two functions: READ-FAKE(O_0) and WRITE-FAKE(O_0,O_1), both of which simply return 0. Notice that these two "fake" functions do not violate the side-effect assumption IRNP makes.

Both of these experiments are run in the context of IRNP, so the differences in the graph are caused by the benefit IRNP can derive from the "real" READ and WRITE functions and the harm that violating the side-effect assumption causes to the workings of IRNP. Figures 6.3 and 6.4 show the results for this experiment for two different orchestration strategies.

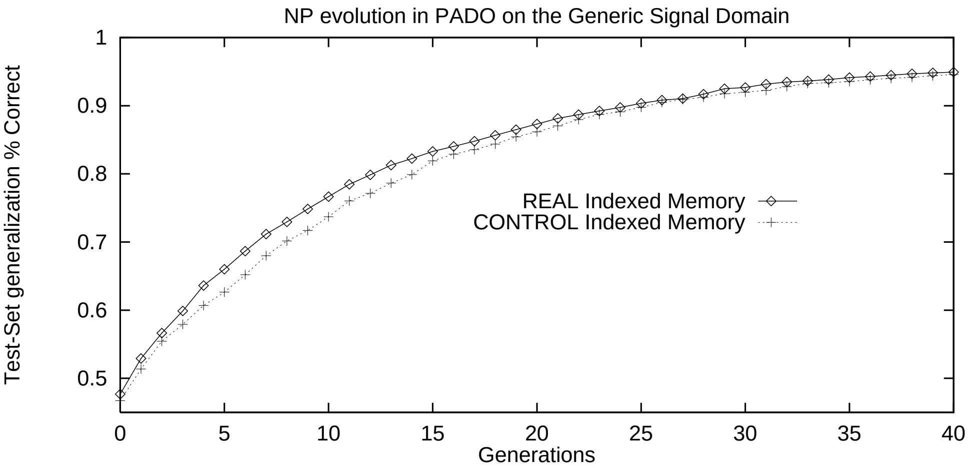


Figure 6.3: IRNP working with Indexed Memory and "Control" Indexed Memory, both using Search-Weight orchestration.

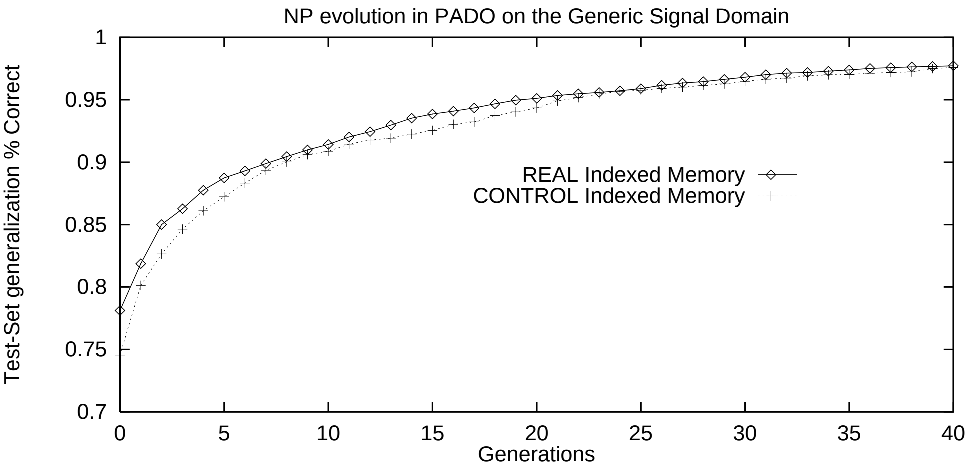


Figure 6.4: IRNP working with Indexed Memory and "Control" Indexed Memory, both using Nearest Neighbor orchestration.

We can see in Figures 6.3 and 6.4 that PADO performs very slightly better with READ and WRITE than with READ-FAKE and WRITE-FAKE. This is good news for two reasons: first, it means that PADO, NP, and IRNP can take some minimal advantage of the indexed memory when it is made available. Second, and much more importantly though, it is clear that the introduction of functions that violate the IRNP side-effect assumption do not cause IRNP to break down.

The performance improvements due to READ and WRITE are tiny. This is because there is already memory in use in the NP programs, the implicit memory use in the data-flow memory on the arcs. This, combined with the fact that this domain does not require a large amount of memory to solve the problem, makes the READ and WRITE functions redundant, so we can hardly expect more improvement than Figures 6.3 and 6.4 show us.

The results of this section should not be taken to suggest that IRNP cannot be expanded to include indexed memory as the dominant form of memory use, only that it has been left for future work. In principle, memory of any form (and indexed memory in particular) can be examined during execution just like the rest of the program. Indexed memory cells could, for example, be treated as time-delay arcs; to describe it in terms of the memory use currently in place in NP in PADO. Once the implementation had been augmented with that capability, the rest of the IRNP procedure worked as described in this chapter.

6.7 IRNP in Tree based GP

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 that 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. Let us see how IRNP can be simply translated to the tree-GP representation.

IRNP has the following major components:

1. Observe the program as it runs and create a first approximation Credit-Blame map.

2. Do Sensitivity-based bucket brigade to redistribute the reward through the Credit-Blame map.

3. Use mutation and crossover operators that attend to the details of this Credit-Blame map.

Step 1 is independent of the topology of the programs. As long as each program is composed of distinct atomic functions (e.g., ADD, MULT, COS, 17), each distinct atomic function (a node in both NP and Tree-GP) can have statistic taken about its actions exactly as described in Section 6.2.1. These statistics can be turned into credit scores in exactly the same way with Tree-GP as with NP since, again, this has nothing to do with the topology of the programs.

Step 2 can also be accomplished in Tree-GP. Sensitivity Function Approximation can be done on **any** function. So the Tree-GP representation is no obstacle to discovering the sensitivity of the atomic functions out of which the programs are built. And the bucket brigade described in Section 6.2.3 relies only on the fact that there are nodes and arcs that describe the program. Whether those nodes and arcs happen to have loops (as they generally do in NP programs) or not (they clearly do not in a tree-structure) is immaterial to the algorithm. So the Credit-Blame can be created in exactly the same manner.

In Step 3 there must be some small deviation between the algorithms for internal reinforcement of the two representations, but the differences are superficial. Crossover in NP is a cut in a graph. This means that many arcs will be disrupted. In tree-GP, crossover always disrupts exactly one arc in each program. This actually simplifies the IR algorithm for Tree-GP. Now crossover need just minimize the credit score of the arc it is breaking while maximizing the average internal arc credit score. Mutation in IRNP, as described in Section 6.3.1, takes advantage of the fan-out of "good ideas" in NP. This cannot be done, by definition, in a tree-based representation. However, Tree-GP can be modified to a related representation, usually used just for space-saving reasons, called DAG-GP. In this representation, the GP programs are composed of nodes and arcs. Each node is still a terminal or non-terminal in GP-speak (i.e., an atomic function such as "ADD"). The only difference is that the topology of the programs is allowed to be a DAG instead of a tree. From a computational expressiveness point of view this makes no difference as there are no loops in a DAG. Now mutation can be redefined to take advantage of the Credit-Blame map and it can do many of the same procedures described in Section 6.3.1. Clearly the mutation procedures that refer to an OUTPUT node will need to be altered, but only slightly as there is a single "output" node for each Tree-GP program. That node is the root node.

This outline shows how IRNP can be successfully transferred to another representation for the evolution of programs.

Chapter 7

Experimental Results

7.1 Experimental Overview

The purpose of any set of experiments is to test a set of hypotheses. The goal of the experiments done in this thesis and shown in this chapter is to demonstrate several general attributes of the PADO, NP, and IRNP approaches:

- PADO successfully applies to a wide variety of signal domains and that the orchestration method employed matters.
- PADO can work equally well in domains with different numbers of classes.
- The IRNP procedure substantially improves learning across a variety of signal domains and this phenomenon is not overly sensitive to the orchestration method employed.

In a conscientious research approach, an attempt will be made to verify that the system created (and therefore the results produced) is not overly sensitive to changes in the parameters. To some extent the previous chapter has already addressed this problem. This chapter makes the same point but in a different manner. Unless otherwise stated, all the parameters in all the experiments discussed in this chapter were fixed to the same values. By demonstrating the different experimental goals across several experiments, without adjusting the parameters, it will become clear that the PADO and IRNP systems are not overly sensitive to those parameters. All these fixed values can be found in Appendix C.2. Table 7.1 gives the values for the most important parameters and Table 7.2 gives the fixed set of program primitives used in all the experiments.

A number of the parameters in Table 7.1 are very small for a specific reason. For example, in PADO a discrimination pool (sub-population) of population 250 programs is, by GP standards, quite small. The programs are only allowed to run for 10 timesteps before their answer is extracted from them. Given the complexity of the problems, this is a very minimal number of timesteps. The MaxNumberInputs need be no larger than four simply because no function (PSP or otherwise) used by the evolving programs happens to use more than four

[1] If-Then-Else(a_0, a_1, a_2, a_3) is defined as "if $a_0 < a_1$ then return a_2 else return a_3."

[2] Split(a_0) returns MaxCONST if $a_0 > \frac{\text{MaxCONST}-\text{MinCONST}}{2}$ and MinCONST otherwise.

CrossoverPercentChance	36
MutationPercentChance	60
PopulationSize	(250*NumClasses)
MaxNumberNodes	80
MinNumberNodes	10
NumberInTournament	5
NumTimeStepsToRun	10
MaxGenerations	80
MaxNumberOutputs	5
MaxNumberInputs	4

Table 7.1: Fixed experimental values for the most important PADO parameters.

Table 7.2: PADO program primitives used

Manipulation type					
Continuous	Add	Sub	Mult	Div	OUTPUT
Choice	If-Then-Else[1]	Split[2]			
Signal	SignalPrimitive_0	...	...	...	SignalPrimitive_i
Zero-Arity	0..MaxValue	Clock			

parameters. However, MaxNumberOutputs is limited to five and limiting the program's fan-out in this manner is another limitation on what it can practically be evolved to accomplish. In these experiments, OUTPUT simply outputs the average of the input values and adds that average (weighted by the time) to the accumulating weighted average response value (see Appendix C.3.2 for details). *Clock* simply returns the time step number on that time step.

There are two reasons for these limitations in the evolutionary process. The first is the simple fact that evolution is slow. These are complex programs examining large input signals over a period of time using computationally non-trivial PSPs. The end result is that a full run of an entire population to generation 80 can take 2 to 24 hours depending on the machine used and the size of the input signal.[3] This means that doing 60-75 independent runs for statistical significance can take as much as a month. And that is only one of the curves presented.

The second equally important reason for the parameter limitations set for PADO is to reduce the amount of overfitting that occurs. With the exception of the first experiment in this chapter (Section 7.2) the problems in these experiments are all real world problems. This means that there is a very limited amount of training data available. The less training available, the quicker a machine learning technique will tend to over fit the training data.

[3]For two extreme examples, a run to generation 40 of the generic signal domain on a Dec AlphaStation 250 4/266 takes about an hour. A run to generation 80 of the natural images domain on a Sparc5 takes about 20 hours.

The single easiest way to avoid (or at least put off) this difficulty is to reduce the number of free parameters in the model being learned. So, some of the parameters were fixed to these limited values to help avoid overfitting.

The summary of these experiments does indeed satisfy the demonstration goals for this thesis. Experiments are run on four totally dissimilar domains and the PADO approach does quite well in all four. Experiments are run on three different classification set sizes and the PADO approach does quite well in all cases. Several experiments show not only PADO run with IRNP, but without IRNP as well. In all those cases PADO does noticeably better when IRNP is active as part of the learning process. All these results are shown to be true for two very different orchestration techniques (Weight-Search and Nearest-Neighbor), using a fixed set of learning parameters. Notice in particular that empirically, *the harder the problem, the greater the efficiency gain provided by IRNP.*

A word of description about the method of presentation before we launch into the experiments. It is customary in EC (and GP in particular) to present either *the percentage of independent runs that resulted in a particular level of performance* or *the mean performance level achieved on average over many independent runs.* This thesis uses the latter method for presenting the experimental results. Be aware then, that the results presented are not the best PADO has ever done on a particular domain, but a report of the *kind of performance you can expect from PADO during an average run.* The detailed graphs that include statistical error information have been separated and grouped in Appendix H.

7.2 Generic Signal Domain

7.2.1 Description of the Domain and Problem

This domain was presented in detail in Section 4.2.1. Briefly, this domain is a manufactured signal type created to have a number of properties appropriate for testing the PADO system. An example signal from each of the four classes from both the training and testing sets is shown in Figure 7.1. The distinguishing signal feature in this domain is the *slope.* Table 7.3 shows the formal definition of these four signal classes. Informally, a high level correct classifier of the four classes is:

> Class 1 has a positive slope. Class 2 has negative slope followed by a positive slope. Class 3 has a negative slope. Class 4 has a positive slope followed by a negative slope.

These are the sources of noise in this domain:

- The slopes vary uniformly within the ranges [-0.156,-0.078] and [0.078,0.156]
- The midline of the signal (mean of all signal values) before noise is added, varies uniformly within the range [75,190]
- Each point of the signal varies uniform randomly from the "correct" line (defined by the midline height and the chosen slopes) by an offset in the range [-25,25]

Class 1	$f_1(x) = mx + b + n$	
Class 2	$f_2(x) = mx + b + n$	$000 \leq x \leq 127$
	$m'x + b + n$	$128 \leq x \leq 255$
Class 3	$f_3(x) = m'x + b + n$	
Class 4	$f_4(x) = m'x + b + n$	$000 \leq x \leq 127$
	$mx + b + n$	$128 \leq x \leq 255$

Where m, m', b, and n are random variables uniformly distributed over $[0.078, 0.156], [-0.156, -0.078], [75, 190], [-25, 25]$ respectively.

Table 7.3: Formal Definition of Generic Domain Signal classes.

7.2.2 Setting PADO up to Solve the Problem

In each of the two experiments in this section the total population size was 1000 (i.e., 250*4). Each point on each graph is an average of at least 75 independent runs. A total of 200 (50 from each of 4 classes) signals were used for training, and a separate set of 500 (125 from each of 4 classes) signals were withheld for testing afterwards.

The PSPs used in these experiments were as follows:

PSP-Point(a_0) takes a point on the signal and returns the value of the signal at that point (Signal[a_0]).

PSP-Average(a_0,a_1) takes an interval on the signal and returns the average value of the signal in the signal interval $[a_0, a_1]$.

7.2.3 The Results

Figures 7.2 and 7.3 show PADO's performance in this generic signal domain with and without IRNP in the context of the Weight-Search orchestration and Nearest-Neighbor orchestration methods respectively. Figure 7.4 shows PADO's performance for both IRNP curves out to generation 40 to demonstrate the effectiveness of the technique in this domain. Notice that unlike the rest of the experiments in this chapter, the experiments in this section are only run to generation 40. This is because the problem is sufficiently easy and therefore PADO has often produced a perfect solution within the first 40 generations.

Now that there are specific curves to examine, we can revisit how these figures are to be read. Each figure shows computational effort in generations (therefore also time) as the independent variable (x-axis) and generalization performance percent correct as the dependent variable (y-axis).

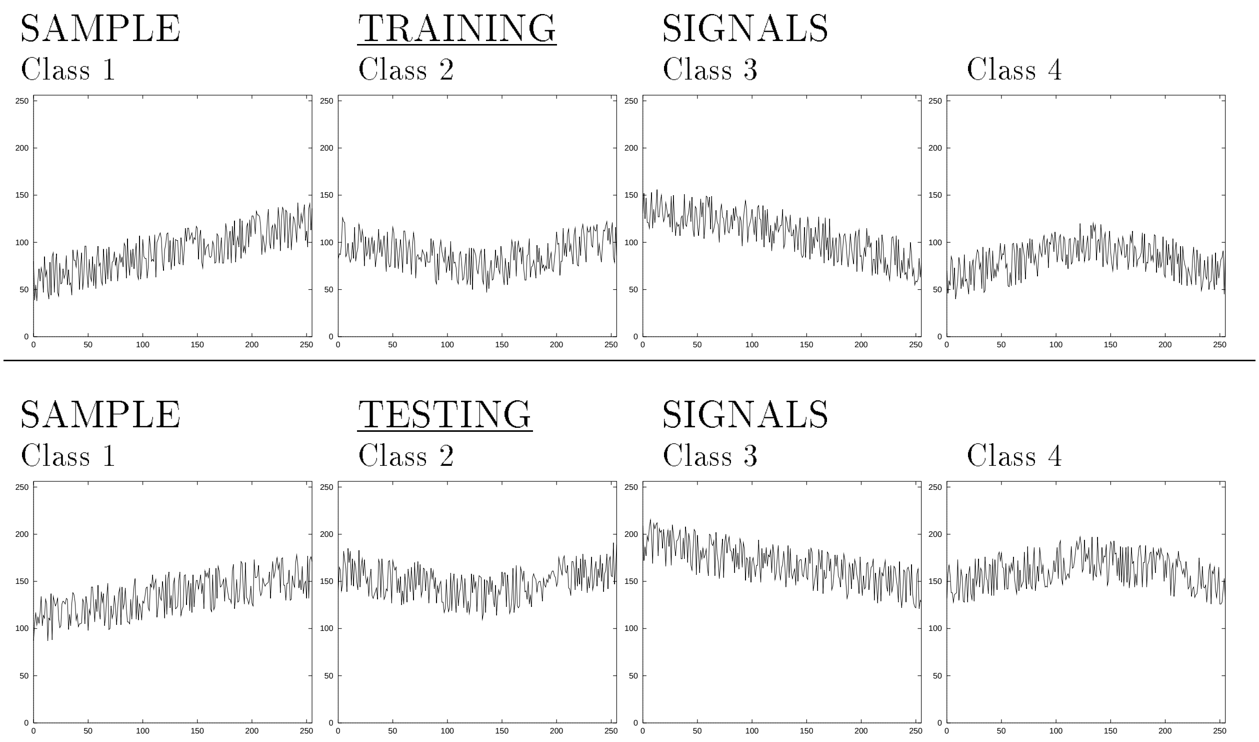


Figure 7.1: Example testing and training signals from the example domain.

Figures 7.2 and 7.3 compare PADO and NP with and without IRNP as a guiding light for the genetic recombination operators. We can see that in both cases (for two different orchestration strategies) IRNP improves the learning efficiency by a considerable margin. Figure 7.4 compares PADO, evolving NP programs, with these same two orchestration strategies. We can see that Nearest-Neighbor orchestration is better than Weight-Search orchestration and that IRNP has noticeably improved on the effectiveness as well as the efficiency of algorithm evolution.

Each point in each of these figures is an average over at least 75 independent trials. This means that a particular point on one of these graphs represents *the expected cost in computational effort required to reach a particular level of generalization performance.* Notice that by generation forty, both curves in Figure 7.4 have nearly, but not quite reached 100% generalization performance on the test set. This is not because PADO never entirely solves this problem. In fact PADO finds a perfect solution before generation 40 almost 50% of the time. However, as with any non-exhaustive search method, the possibility of local maximum exists for evolutionary computation. Some of the time PADO finds a solution that, for example, generalizes to 98.2% of the testing set but no more, and further learning effort does not improve upon this very good, but imperfect solution. In general, when looking at these graphs of PADO's performance, keep in mind that these curves represent not the worst or best that PADO has ever done, but the expected performance level for that expenditure of computational effort.

In Figures 7.2 and 7.3, the graphs show that IRNP provides slightly less than a factor two increase in *efficiency.* Efficiency, as it is referred to in this chapter, is speed measured in

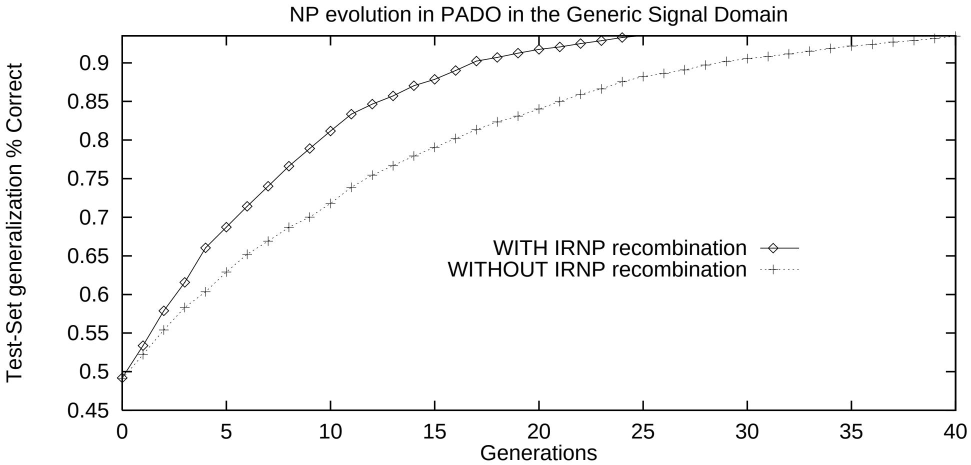


Figure 7.2: NP learning **with and without IRNP** using PADO Weight-Search orchestration.

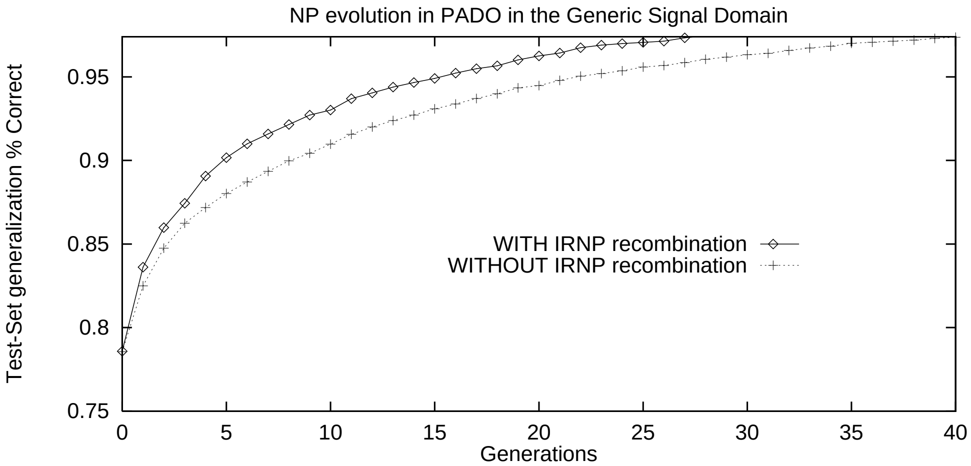


Figure 7.3: NP learning **with and without IRNP** using PADO Nearest-Neighbor orchestration.

performance per computational effort. Performance is measured in generalization percentage correct on the test-set. Computational effort is measured in generations.

The rest of the domains in this chapter are sufficiently difficult in that finding a perfect solution (i.e., a solution that generalizes perfectly) is incredibly unlikely. However, for this generic signal domain, it is fairly likely. Another piece of empirical evidence in favor of IRNP is that over 75 independent experiments, PADO without IRNP was only 58% as likely to find a perfect solution by generation 40 as PADO evolving in the context of IRNP. Empirically, the harder the problem, the greater the efficiency gain provided by IRNP.

7.2.4 Result Variance from Parameter Changes

Appendix C.2 describes the various parameters of the PADO, NP, and IRNP systems. There are a number of parameters (e.g., PopulationSize) that there is little reason to suspect IRNP is brittle with respect to. There are other parameters, however, for which it is possible that IRNP would be highly sensitive. The clearest examples of such parameters are the related

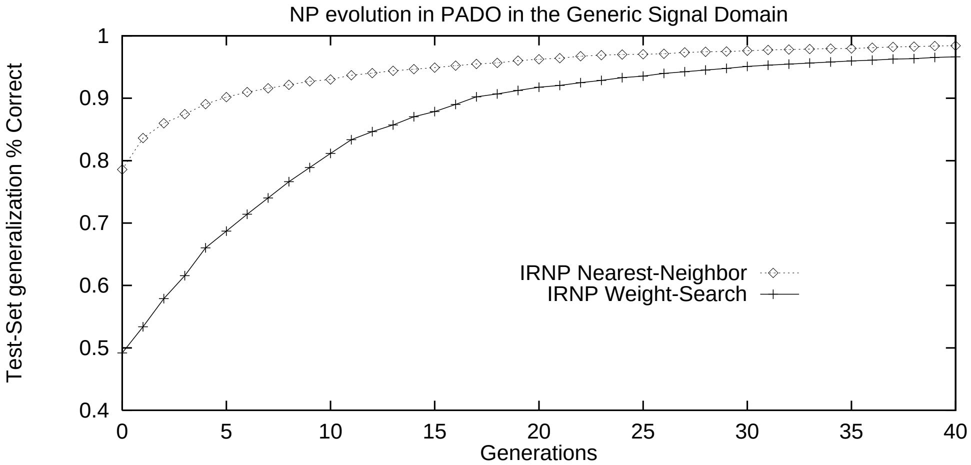


Figure 7.4: NP learning **with IRNP** using PADO Weight-Search and Nearest Neighbor orchestrations.

parameter: *CrossoverPercentChance* and *MutationPercentChance*.

For this thesis, *CrossoverPercentChance* has been fixed at 36% and *MutationPercentChance* has been fixed at 60%.[4] Figures 7.2, 7.3, and 7.4 were all generated with those particular values for *CrossoverPercentChance* and *MutationPercentChance*. This section presents a deviation from these values as a demonstration that IRNP provides significant benefits in both the crossover and mutation recombination strategies. To this end, the experiments presented in Figures 7.2, 7.3, and 7.4 are here redone with *CrossoverPercentChance* set to 60% and *MutationPercentChance* set to 36%. Figures 7.5, 7.6, and 7.7 show the results of these experiments.

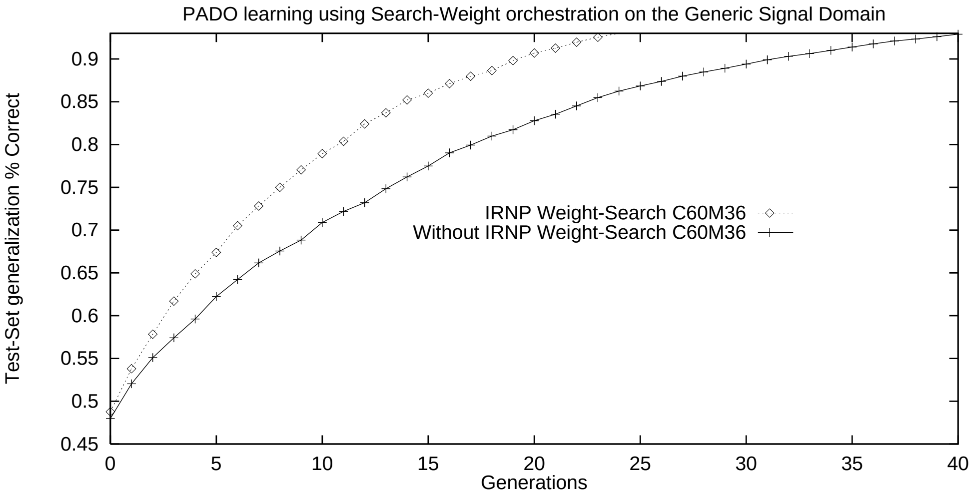


Figure 7.5: NP learning **with and without IRNP** using PADO Weight-Search orchestration **CrossoverPercentChance=60% and MutationPercentChance=36%**.

These changes to the values of crossover and mutation that are used in the rest of this thesis actually improve the results by a small margin.

[4]Both of these values were chosen for historical reasons [Teller and Veloso, 1996].

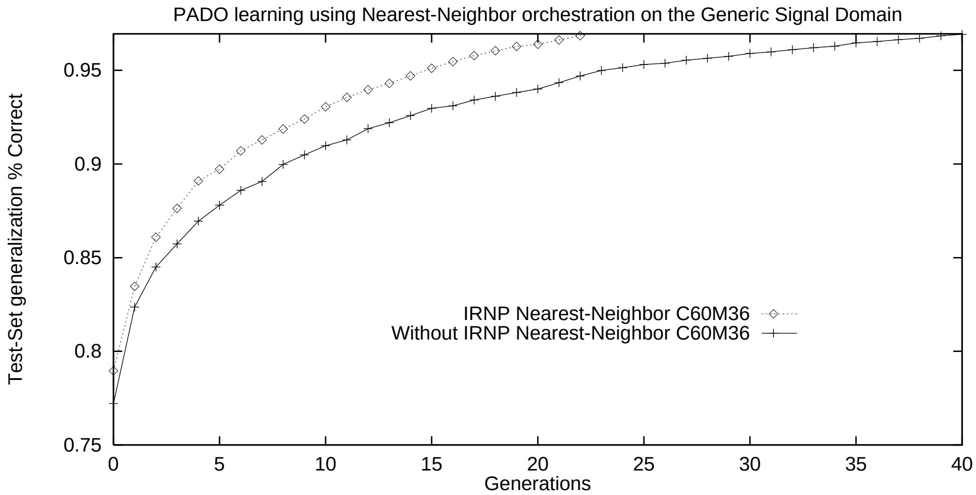


Figure 7.6: NP learning **with and without IRNP** using PADO Nearest-Neighbor orchestration **CrossoverPercentChance=60% and MutationPercentChance=36%**.

7.3 Natural Images

7.3.1 Description of the Domain and Problem

There are seven classes in the domain used in the following experiments. Figure 7.8 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.

7.3.2 Setting PADO up to Solve the Problem

In each of the two experiments 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 PSPs used in these experiments 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).

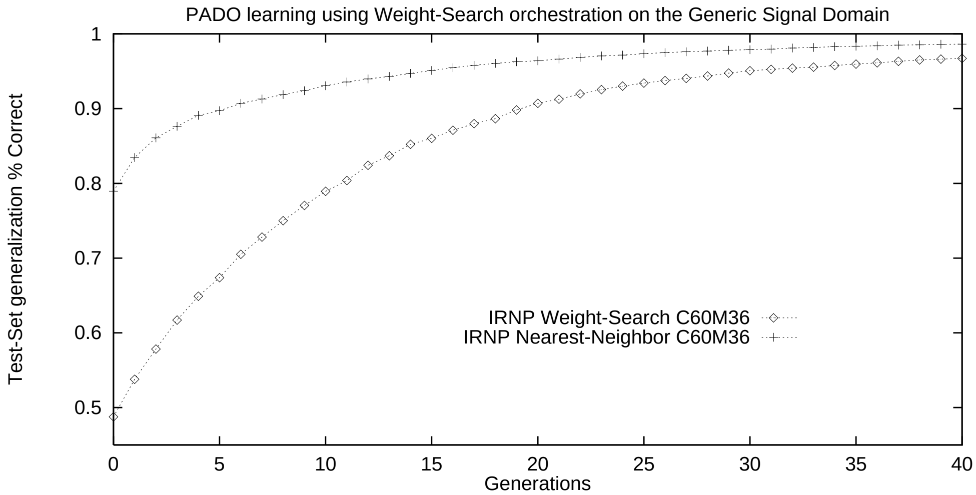


Figure 7.7: NP learning **with IRNP** using PADO Weight-Search and Nearest Neighbor orchestrations **CrossoverPercentChance=60% and MutationPercentChance=36%**.

PSP-Variance(a_0, a_1, a_2, a_3) returns the *variance* 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 difference 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).

7.3.3 The Results

In the following experiment, the generalization performance on a separate set of testing images was recorded during each run and Figure 7.9 plots the *mean* of each of these values.

In this experiment, the performance of the PADO system was achieved through a PADO orchestration strategy called Weight-Search orchestration (see Section 4.1.3 for details). Figure 7.9 shows the computational effort in generations required to reach a particular level of test-set generalization performance for NP learning with and without IRNP.

The most important feature of Figure 7.9 is that NP learns more than *twice as fast* when IRNP is applied to the recombination during evolution. Also notice that NP learns quite well on this difficult image classification problem. Random guessing in this domain would achieve only about 14.28% correct generalization performance.

In a second experiment from this same domain, the PADO system used Nearest-Neighbor orchestration to classify the images. Figure 7.10 shows the computational effort in generations required to reach a particular level of performance.

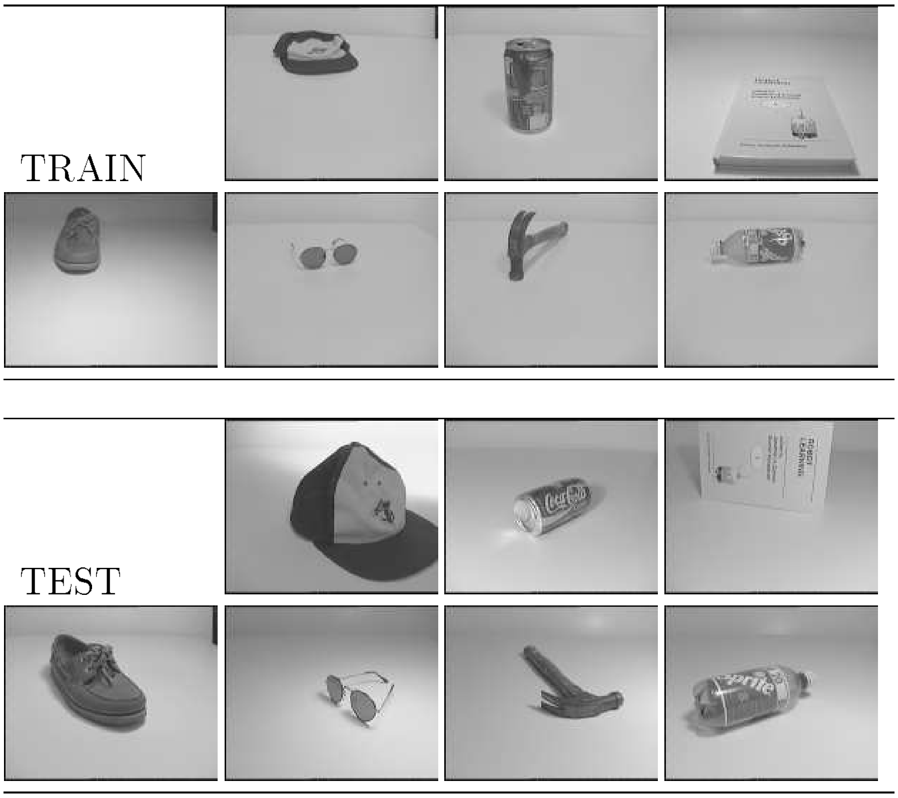


Figure 7.8: A random training and testing signal from each of the 7 classes in this classification problem.

Again NP learns more than *twice as fast* when IRNP is applied. The difference between using and not using IRNP in this experiment is a little less noticeable than in the previous experiment from this domain. This is entirely attributable to the difference in orchestration (since that is the only change from the first to the second experiment). Despite this small difference, it is encouraging that IRNP is robust to this aspect of the system that incorporates it. Also notice that NP learns even better than before on this difficult image classification problem.

Finally, Figure 7.11 shows both IRNP based curves for the two experiments together so that their performance level at generation 80 can be seen.

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

Another important point of discussion is the issue that while random guessing generates a generalization percent performance of only 14.28%, Figure 7.11 shows that in generation 0 both orchestration strategies produce higher performance than this value. The explanation for this is two fold. First, at the end of generation 0, some search has already been done; hundreds of random programs have been generated and the *best one* from each discrimination pool has been selected and used in orchestration and testing. Second, particularly

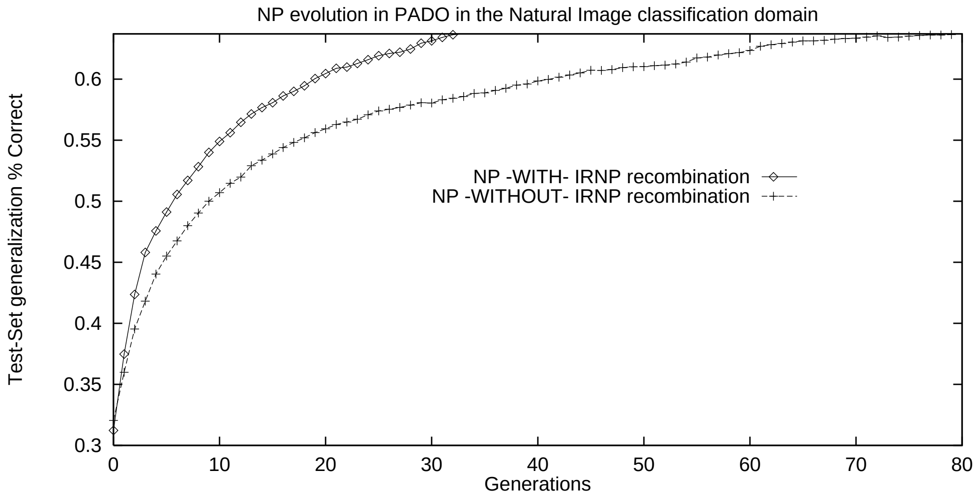


Figure 7.9: NP learning **with and without IRNP** using PADO Weight-Search orchestration.

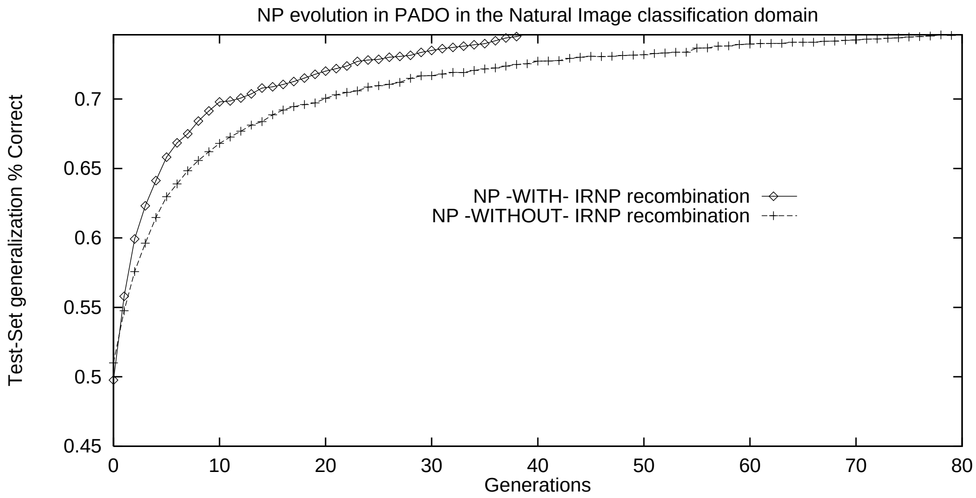


Figure 7.10: NP learning **with and without IRNP** using PADO Nearest Neighbor orchestration.

in the case of the nearest-neighbor orchestration strategy, while it helps for the individual discrimination subproblems to be solved in order to solve the overall classification problem, it is not necessary for this to happen for some non-random level of performance to be reached through the learning that takes place in the nearest-neighbor orchestration strategy.

7.4 Acoustic Signals

7.4.1 Description of the Domain and Problem

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 for about 500,000 bits per training or testing sound. 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

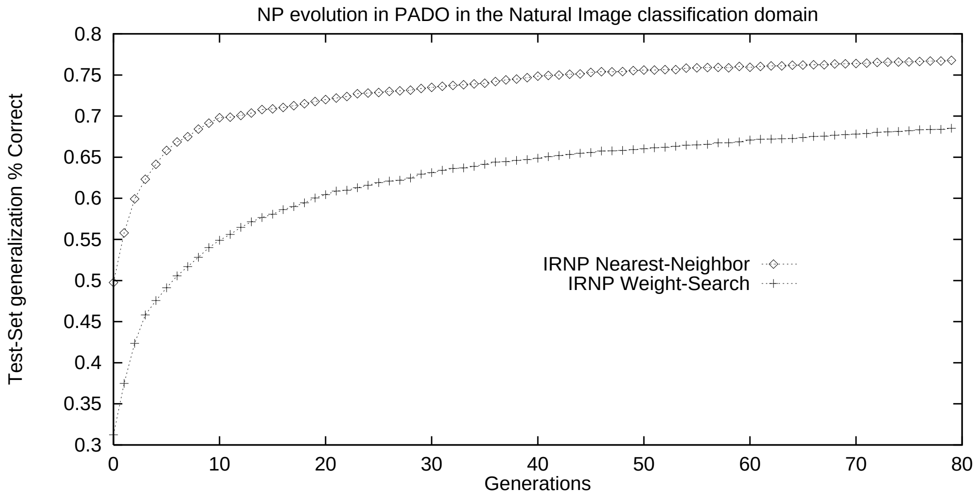


Figure 7.11: NP learning with IRNP using both PADO **Nearest Neighbor and Weight-Search** orchestration.

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.

7.4.2 Setting PADO up to Solve the Problem

In each of the two experiments 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 these experiments were as follows:

PSP-Point(a_0, a_1) returns the wave height at the moment in time specified 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 periods since it's return value will be, by definition, the waveform's midline.

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 classification experiment discussed in Section 7.3. This was not done to demonstrate the generality of these PSPs. On the contrary, this similarity in the experimental procedure was done to highlight how little was done to tune NP or PADO in order to achieve the reported results. PADO, using NP and 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 fitness used for evolutionary learning (training of the PADO programs) was based upon distance from returned confidence to the correct confidence for each training example. Given this model of one class chosen per sound, if PADO just guessed, it could achieve a generalization performance of 1/7 (14.28%) correct[5].

7.4.3 The Results

Figure 7.12 shows the generalization percent correct PADO reaches on average on each generation, with and without IRNP, using the Weight-Search orchestration.

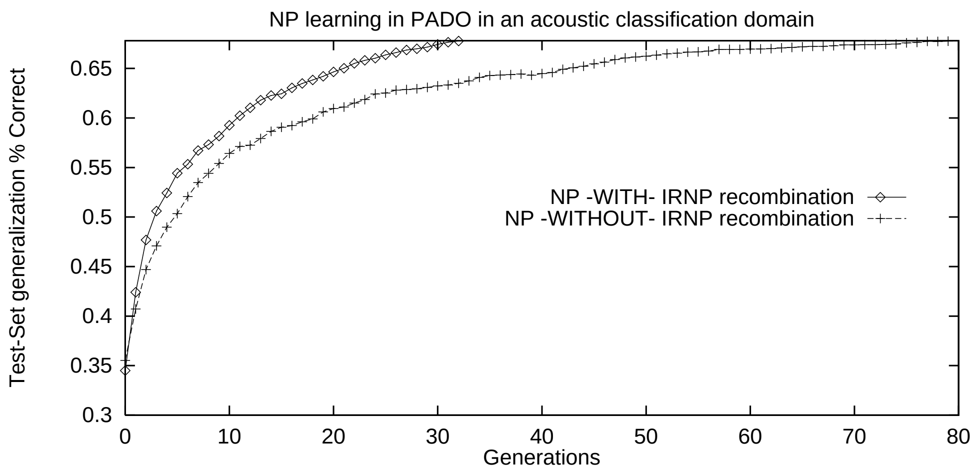


Figure 7.12: NP learning **with and without** IRNP using PADO Weight-Search orchestration.

Figure 7.13 shows the generalization percent correct PADO reaches on average on each generation, with and without IRNP, using the Nearest-Neighbor orchestration.

Figure 7.14 shows the generalization percent correct PADO reaches on average on each generation, with IRNP for both the Weight-Search and Nearest-Neighbor orchestration strategies. Notice that in these experiments, for both orchestration strategies, IRNP learning is almost *three times as efficient* as learning without it.

[5]Given that the training and test sets have an even number of signals from each class.

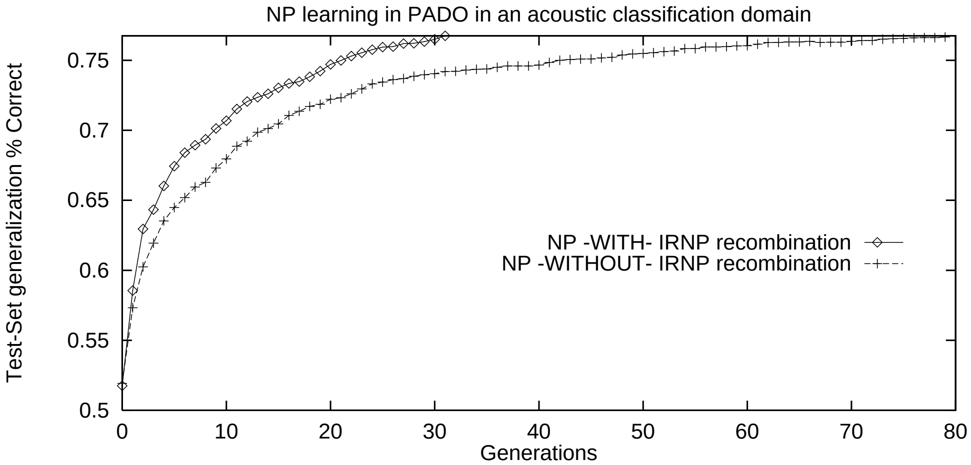


Figure 7.13: NP learning **with and without** IRNP using PADO Nearest Neighbor orchestration.

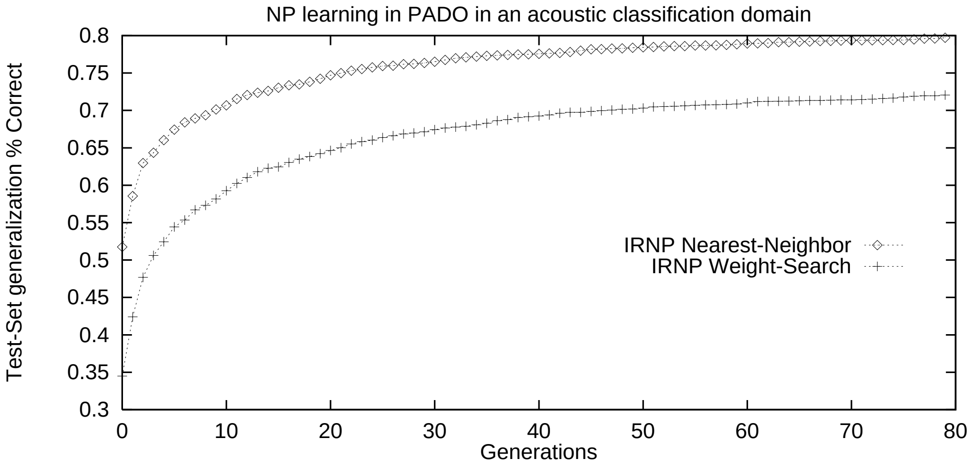


Figure 7.14: NP learning with IRNP using both PADO **Nearest-Neighbor** and **Weight-Search** orchestration.

7.5 Acoustic Signals Revisited

One of the most important implied aspects of this thesis 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, then increasing the amount of time (i.e., maximum timestep threshold, T) should often increase the evolved program performance. Having noted this, let us revisit the acoustic signal classification problem described in the previous section. As in the rest of this thesis, 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 efficiency and, more importantly, the effectiveness of NP learning within PADO.

Since this thesis 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[6]) will strengthen this argument. This is a critical point:

> The NP programs evolving in this section have the exact same number of degrees of freedom as in the previous section. Programs in all ways similar to those in the previous section are simply allowed to "think longer" about the signal in question. Therefore, improved performance in this experiment demonstrates that NP programs are making use of the looping/foveating aspects of the NP programming language.

7.5.1 Description of the Domain and Problem

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

7.5.2 Setting PADO up to Solve the Problem

In setting up PADO to solve this acoustic signal classification problem, every aspect 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.

7.5.3 The Results

Two facts are worth noting. The first is that doubling the timestep threshold almost[7] doubles the amount of time it takes PADO to complete a generation. Although this does not have much bearing on the effectiveness graphs below (Figures 7.17 and 7.18), the efficiency graphs do not reflect this increased time it takes to compute a generation (Figures 7.15 and 7.16). Each point in each of these figures is an average over at least 60 independent trials.

The second fact is that while it takes more time to compute the fitness 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 fitness takes twice as long to measure, not because twice as many decisions are made with the same information. This is significant by itself, but more significant 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.

[6]In Tree-GP, the standard method for giving a function more "time" to think about an input is to allow for larger functions to be created, but this has the obvious impact of increasing the number of degrees of freedom in the model being learned which makes "memorizing the training data" easier.

[7]There are a number of small costs associated with running the system that do not increase when each program's running time is doubled.

Figures 7.15 and 7.16 show the generalization percent correct PADO reaches on average on each generation with IRNP using this enlarged timestep threshold, Weight-Search orchestration and Nearest-Neighbor orchestration respectively.

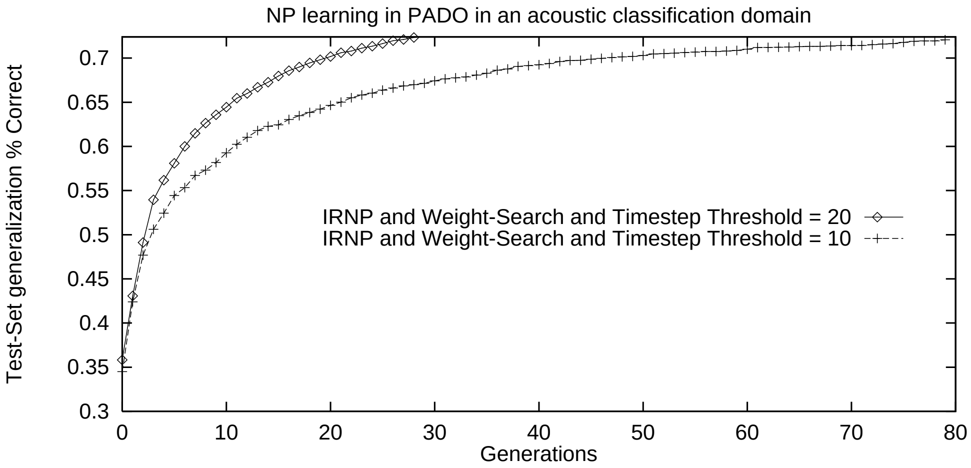


Figure 7.15: NP learning with IRNP using PADO Weight-Search orchestration and **Timestep Threshold = 10** *and* **20**.

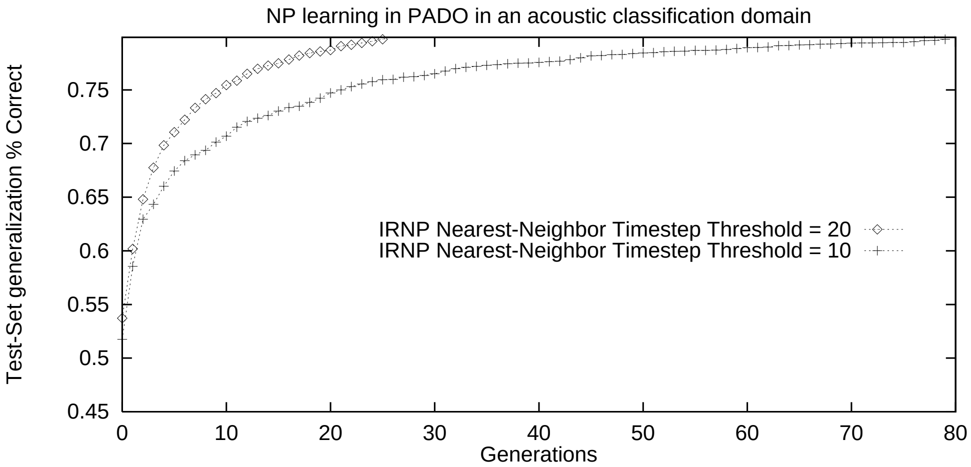


Figure 7.16: NP learning with IRNP using PADO Nearest Neighbor orchestration and **Timestep Threshold = 10** *and* **20**.

Figures 7.17 and 7.18 show the generalization percent correct PADO reaches on average on each generation, with IRNP for both the Weight-Search and Nearest-Neighbor orchestration strategies respectively, using this enlarged timestep threshold. In these two graphs the data has been redisplayed so that the eventual effectiveness of learning with the additional timesteps can be seen.

The results of these experiment are quite exciting. IRNP learning with a timestep threshold of $T = 20$ is *three times as efficient* 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 does NP learning without IRNP using $T = 10$ using only about **13%** of the effort!

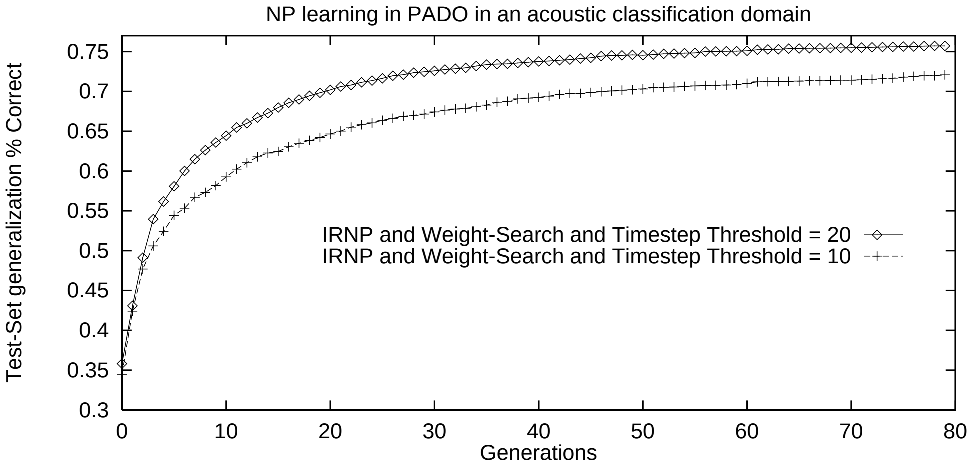


Figure 7.17: NP learning with IRNP using Weight-Search orchestration and different timestep thresholds.

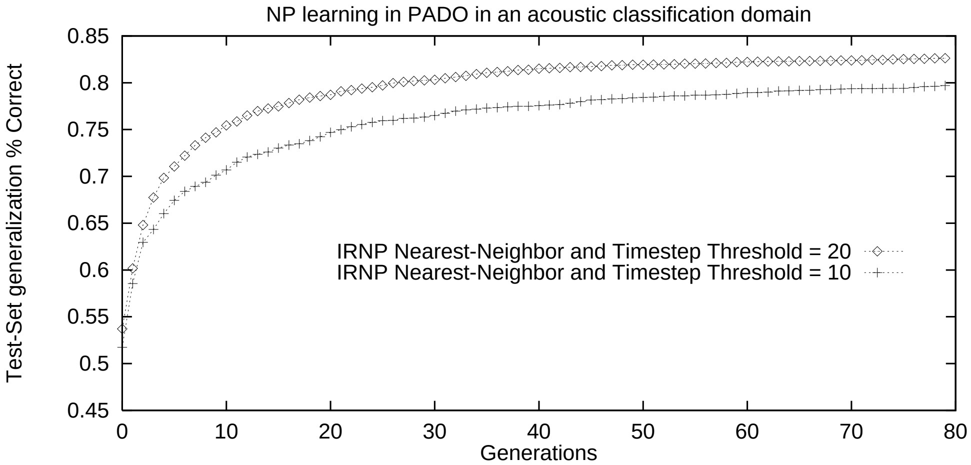


Figure 7.18: NP learning with IRNP using Nearest Neighbor orchestration and different timestep thresholds.

This phenomenon – increasing timestep threshold leading to increased generalization performance – may be asymptotic; above a certain amount of time, increasing NP's time threshold may not yield higher performance and near that certain amount of time, increasing the time threshold may have only a very slight positive impact on the performance. An interesting and important piece of future work is the further investigation of exactly how evolved NP programs use these looping/foveating properties of the NP language.

7.6 Protein Identification

7.6.1 Description of the Domain and Problem

Proteins provide the bulk of biological structure and biological functionality [Stryer, 1995]. There are two very different views of a protein: *primary structure* and *tertiary structure*[8]. In both cases, the building blocks for all proteins in all living organisms are *amino acids*. With exceedingly few exceptions, all proteins on earth are built from 20 amino acids that are symbolized by the 20 characters **A**, **B**, **C**, **D**, **E**, **F**, **G**, **H**, **I**, **K**, **L**, **M**, **N**, **P**, **Q**, **R**, **S**, **T**, **V**, **W**, and **Y**.

The primary structure of a protein is defined by the strongest bonds in the protein. This means that, by ignoring most of the interactions between the building blocks, the protein can be described as a single dimensional list of those 20 characters. This linear, 1-D structure never appears as such in nature; there are always 3-D folds and important secondary bonds. However, this listing of the amino acids is an extremely convenient and useful simplification of the protein structure. The tertiary structure is the 3-dimensional structure of the protein. In nature the chemical bonds in a protein cause it to fold up into something much more like a ball than a string.

As an example, let us look at the primary and tertiary structures for the protein BPTI (bovine pancreatic trypsin inhibitor). The primary structure for BPTI is shown in Table 7.4. The tertiary structure for BPTI is shown in Figure 7.19.

RPDFCLEPPYTGPCKARIIRYFYNAKAGLCQTFVYGGCRAKRNNFKSAE DCMRTCGGA

Table 7.4: The primary (1-D) representation for the protein BPTI: bovine pancreatic trypsin inhibitor.

It turns out that the function of a protein is almost entirely determined by its 3-dimensional structure. There is also a very strong correlation between the 1-dimensional representation of a protein and its 3-dimensional structure. This means that, by transitivity, the function of a protein (a primary subject of interest in biology) is largely determined by the 1-dimensional list of amino acids that describe the protein. Unfortunately, the two mappings just alluded to, from 1-D to 3-D and from 3-D to function, are both poorly understood.[9] As a result, computational biology has focused quite a bit of its attention on finding approximations to these mappings for different subsets of the protein domain. This process is often called the *protein folding problem* in computational biology.

Finding algorithms for computing the 3-D structure and function of a protein based on the 1-D structure of that protein seems at first like an odd waste of time. The reason this problem exists is because biologists can record the primary structure of a protein with immensely less effort and expense than recording it's 3-D structure or function. The process of recording a protein's primary structure is called *protein sequencing*.

[8] Tertiary structure is often called *conformation*.

[9] It is certainly the case that the mapping from the tertiary structure to the function of the protein is better understood than the 1-D to 3-D mapping being attempted in this experiment.

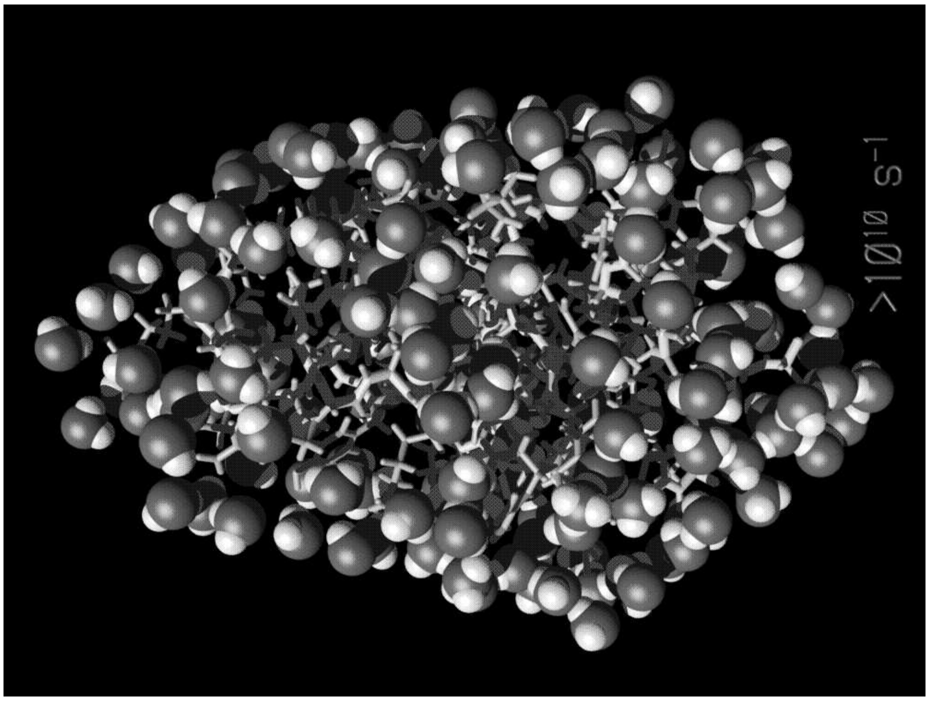


Figure 7.19: A rendered image of the 3D structure of the protein BPTI: bovine pancreatic trypsin inhibitor.

Because of this bias towards sequencing proteins and then trying to "guess" their 3-D structure and function from those sequences, the number of unlabeled protein sequences is increasing exponentially each year. The SWISS-PROT database [Bairoch and Boeckmann, 1991] is an example site at which such unlabeled proteins are placed. The following experiment is drawn from labeled protein sequences from that database.

Finding the 3-D structure and function of a sequenced protein is the goal, but either of those attributes could be described well below the level of detail necessary for a trained biologist to understand the basic nature, shape, and function of the protein. There are a range of mid-level descriptions of the 3-D structure that correspond well to particular functionality among most proteins. The specific experiment this section describes attempts to classify proteins into one of five different classes, where each class is such a useful, mid-level description of both 3-D structure and functionality. These five classes are: *extracellular*, *intracellular*, *nuclear*, *membrane integral*, and *membrane anchored*.

In 1997, a report appeared in the *Journal of Molecular Biology* that described an algorithm for predicting which of the five aforementioned classes an unlabeled protein falls into [Cedano *et al.*, 1997]. This algorithm, based on the statistical occurrences of the different amino acids, was reportedly trained on 1000 labeled proteins (proteins for which the 5-way classification was known) before it correctly identified the class for 76% of 200 "unlabeled" proteins (i.e. labels withheld until after testing). The goal of this experiment was to apply PADO to this problem using the same 1000 labeled proteins as a training set and the same 200 "unlabeled" proteins as a testing set.

7.6.2 Setting PADO up to Solve the Problem

As with all PADO experiments, the user must provide the training and testing signals. In this case, each signal to be classified is a string of characters representing the amino acid primary structure of the protein to be classified. The shortest signal in the training set is 17 characters (amino acids) long and the longest signal in the training set is 4911 characters long. In addition, the user must provide PADO parameterized signal primitives for examining the signal. The following PSPs were provided to PADO during these experiments:

PSP-Hydro(a_0, a_1) returns the amount of hydrophobicy/aliphaticy of the amino acids in the interval specified by $[a_0, a_1]$. There is wide disagreement about how this sort of thing should be measured. Appendix D.1 gives the hydrophobicity values chosen for this experiment.

PSP-Weight(a_0, a_1) returns the average weight (measured in *daltons*) of the specified interval $[a_0, a_1]$. These values are agreed upon and the individual values to be summed are shown in Appendix D.2.

PSP-Charged(a_0, a_1) returns the percentage of charged (+ AND -) amino acids that exist in the specified interval $[a_0, a_1]$. The charged amino acids are largely, but not entirely agreed upon. Appendix D.3 shows how this PSP is calculated.

PSP-Occurrence(a_0) returns the percentage of amino acid a_0 that appears in the signal to be classified.

PSP-VdWVolume(a_0, a_1) returns the average Van der Waals Volume in the interval $[a_0, a_1]$. The volume contribution of each amino acid is not well agreed upon. Van der Waals Volume is one of the fairly standard measures for the amount of space taken up by the different amino acids. Appendix D.4 gives the details on how this PSP is computed.

It is important to note that there is massive disagreement within biology about all of these "views" of a protein. For example, there is no consensus either about the exact relation of hydrophobicity and transmembrane proteins (class 4: membrane integral) or about the exact relation of the individual amino acids to the hydrophobicity of the protein. This means that the PSPs provided to PADO are certainly inferior to those that could have been created by a trained biologist studying this problem (e.g., [Cedano *et al.*, 1997]). It is exciting to see how well PADO does even under these circumstances, and it is very likely that under the guidance of a biologist PADO's performance in this domain would improve further.

In addition, these are by no means the best parameterized signal primitives to create either in terms of how they are parameterized (simply specifying sub-segments of the 1-D structure) or in terms of the general features to examine (e.g., hydrophobicity). The former was chosen because it is simple and clear, and the latter was chosen because these are the attributes of amino acids that get the most attention on the world wide web and so have the most information readily available.

7.6.3 The Results

In the previous several sections, the presentation of results focused on how much computational effort need be put in to achieve a particular level of generalization performance *on average*. This section will, in contrast, pick one highly successful set of NP programs that were learned together and orchestrated to achieve the results reported.[10] This set of programs is presented in Tables 7.5, 7.6, 7.7, 7.8, and 7.9.

Node	CreditScore	Function	Inputs	Output Arcs	Node	CreditScore	Function	Inputs	Output Arcs
1	0.116111	Sig-Len	4	57_2 44_1 45_1	5	0.059578	250	1	44_0
6	0.000716	596	2	65_2	10	0.116111	Sig-Len	3	58_0 57_1
12	0.176580	078	1	47_2	15	0.116111	Sig-Len	1	50_0
17	0.036305	377	1	61_0	18	0.011894	250	0	60_2
19	0.116111	Sig-Len	3	60_3 61_1 63_1 63_2 63_3	22	0.011435	377	1	69_0 60_1
23	0.116111	Sig-Len	0	55_3 49_1 49_0 48_0	25	0.051476	187	2	59_0 52_0 50_1
28	0.000650	187	0	65_3	40	0.011342	165	0	56_3
42	0.003700	752	4	57_0	44	0.177437	Split	3	47_1 45_2 56_1
47	0.177535	Output	4	54_1 55_0 47_0	50	0.154348	Split	4	54_0 59_3
52	0.004006	MULT	4	57_3	55	0.114990	Output	4	51_1 51_2 51_3 56_0
56	0.159417	ADD	4	54_3 55_1 55_2	57	0.056302	ADD	4	54_2 53_0
60	0.176813	SUB	4	47_3 52_1	61	0.066143	DIV	2	52_2 56_2 52_3
63	0.162016	ADD	4	60_0	65	0.010648	SUB	4	65_0 65_1 63_0

Table 7.5: A learned NP program to discriminate protein class **1** from all other classes.

Node	CreditScore	Function	Inputs	Output Arcs	Node	CreditScore	Function	Inputs	Output Arcs
0	0.016393	238	0	34_1	2	0.017864	245	1	43_1 50_1 70_1
3	0.305514	838	3	42_0 70_0 43_0 66_2	11	0.066670	030	2	73_1 44_1 62_1
14	0.299996	030	2	51_1 37_1	17	0.305474	915	3	65_1 42_3 71_1
19	0.020331	169	2	71_0 68_0	24	0.125573	Sig-Len	0	53_2 53_1 36_2 36_3 37_3
28	0.065365	169	1	37_2 73_2	34	0.262321	ADD	4	35_0 74_1
35	0.263357	Output	4	73_0	36	0.262287	SUB	4	35_1 46_1 34_0 35_3
42	0.306720	Output	4	42_1	46	0.018663	PSP-Weight	3	34_2 34_3
51	0.307870	PSP-Hydro	2	42_2 41_1 53_0	53	0.300901	SUB	4	51_0 35_2
55	0.017073	PSP-Occur	4	36_0 55_0	68	0.030845	SUB	2	38_1 53_3 59_0 58_1
69	0.016962	MULT	1	69_0 36_1 64_1	70	0.017931	PSP-VdW	3	46_0
73	0.263357	ADD	3	68_1 77_1					

Table 7.6: A learned NP program to discriminate protein class **2** from all other classes.

Node	CreditScore	Function	Inputs	Output Arcs	Node	CreditScore	Function	Inputs	Output Arcs
2	0.210147	Sig-Len	2	45_0 60_3	3	0.215147	Sig-Len	3	46_0 49_1
5	0.210147	Sig-Len	4	51_0 78_1	7	0.206148	592	1	43_1
13	0.210147	Sig-Len	2	62_2	14	0.047335	603	1	63_0
15	0.244704	CLOCK	3	61_1 64_1	16	0.250192	Sig-Len	1	67_0 67_2
19	0.251061	CLOCK	3	61_0 62_1 62_0 69_2	20	0.210147	Sig-Len	3	65_2 63_1
26	0.215120	491	2	73_0 46_3	27	0.151236	840	1	42_2 51_2 55_0
29	0.210147	Sig-Len	3	42_0 54_1	31	0.083166	CLOCK	4	55_1 52_0
34	0.206656	CLOCK	1	43_0	35	0.011626	371	2	62_3 73_1 42_3
37	0.094720	304	2	57_0	39	0.210147	Sig-Len	1	70_1 70_0
41	0.156044	PSP-Hydro	3	50_2 74_0 50_3	42	0.152511	MULT	4	41_0 74_1
43	0.215092	PSP-Weight	2	78_0 46_1	46	0.215997	Output	4	50_1
51	0.152053	Output	4	41_1	52	0.100396	PSP-Occur	1	77_0
53	0.149943	PSP-Weight	2	54_0 72_0 53_0	54	0.156064	PSP-Weight	2	50_0
57	0.143708	SUB	2	53_1	58	0.177966	PSP-Weight	3	42_1 59_0
59	0.214836	PSP-Occur	2	45_3 46_2 68_0	61	0.251127	PSP-Hydro	2	60_0 67_3 58_1 69_0
62	0.163411	ADD	4	47_1 76_1	63	0.140974	Split	2	77_1 60_2
64	0.249790	PSP-Hydro	2	66_0	66	0.250128	MULT	1	67_1
67	0.251180	Output	4	47_2 69_1	69	0.251889	Output	3	60_1
70	0.151416	PSP-Hydro	3	47_3 51_1	73	0.160914	MULT	2	76_0 44_0
74	0.096191	PSP-Hydro	3	57_1	76	0.244135	SUB	2	64_0 58_0
77	0.151435	ADD	2	51_3					

Table 7.7: A learned NP program to discriminate protein class **3** from all other classes.

Notice that in all five shown learned NP programs there are nodes "missing" (e.g., in Table 7.6, nodes 71 and 72 are not listed). This is because these programs have been reduced *using the Credit-Blame map* so that only the effective parts of the program are shown. The

[10] This alternate presentation is done because it corresponds best with the reporting method of the two research works against which the experimental results of this section are compared.

Node	CreditScore	Function	Inputs	Output Arcs	Node	CreditScore	Function	Inputs	Output Arcs
0	0.445867	328	3	20_0	1	0.539863	100	1	22_0 28_1
3	0.545543	100	1	33_0 31_0	4	0.306296	662	4	39_0
5	0.552905	100	1	42_0 48_1	7	0.544190	235	2	47_1
8	0.545530	942	0	47_0	9	0.000043	CLOCK	3	25_1 51_2
10	0.000028	624	1	25_2	11	0.281462	235	0	48_2
12	0.016601	100	1	29_0 24_1 52_1	14	0.000935	328	1	54_1 34_2 51_0
17	0.000329	870	2	24_2 23_3	18	0.000050	662	1	51_1 23_2
19	0.552736	Output	4	19_1	20	0.538241	PSP-Occur	4	22_1
21	0.552719	IFTE	4	19_0 43_1	22	0.552368	PSP-Hydro	3	21_0 23_1 30_0 28_3
24	0.183872	MULT	4	28_0	25	0.016665	MULT	3	52_0
28	0.552368	IFTE	4	19_2 19_3 21_1 21_2 21_3	31	0.543903	PSP-Occur	3	33_1
32	0.560383	Output	4	40_0	33	0.558179	PSP-Hydro	3	32_0 39_1 35_0
34	0.558106	MULT	3	32_3 34_0 43_0	35	0.551539	ADD	2	46_3 28_2
39	0.558036	DIV	3	32_1	40	0.551243	PSP-Occur	4	42_1
41	0.567945	Output	4	38_0	42	0.565711	PSP-Hydro	3	41_0 48_3
43	0.516104	MULT	3	34_1 44_1 48_0	47	0.567799	PSP-Weight	2	32_2 25_0 38_1
48	0.567872	IFTE	4	38_2 38_3 41_1 41_2 41_3	50	0.364228	ADD	4	50_1 50_0 35_1
51	0.025913	MULT	3	24_3 50_3	52	0.025138	ADD	2	50_2
54	0.000864	MULT	2	54_0 43_2					

Table 7.8: A learned NP program to discriminate protein class **4** from all other classes.

Node	CreditScore	Function	Inputs	Output Arcs
0	0.221809	653	0	20_0 46_2
4	0.336564	688	2	34_1 21_0 26_0
5	0.271669	103	1	38_0
10	0.338315	544	1	34_0
12	0.003878	257	2	44_0
16	0.358478	308	1	42_1 39_0 41_1
17	0.086478	Sig-Len	4	18_3 32_2
19	0.292845	PSP-Charge	2	22_0 19_0
20	0.233774	PSP-Charge	3	20_1 18_0
21	0.276418	MULT	3	19_1
22	0.293530	Output	2	30_0 31_0 27_0 21_1
30	0.294689	Output	4	18_1 28_1
33	0.293492	PSP-Charge	2	30_3
34	0.356565	PSP-Charge	2	35_0 36_0 41_0
35	0.272933	Output	3	18_2
36	0.357400	Output	2	39_1 36_1 42_0
38	0.272036	MULT	1	35_2
39	0.366780	PSP-Hydro	3	40_1 44_1 33_0
41	0.293455	PSP-Charge	2	30_1 30_2 31_1 31_2 31_3
42	0.292916	PSP-Charge	2	21_2 29_0 35_1 22_1 29_1
44	0.277030	MULT	2	33_1

Table 7.9: A learned NP program to discriminate protein class **5** from all other classes.

same has been done with the arcs. Only arcs that actually affect the program are shown. *This has reduced the complexity of the displayed programs by more than a factor two.* While this is not a focus of this thesis, one of our long term goals is to aid not only the learning of complex programs, but also the understandability of those learned programs after they have been learned. The complexity reduction demonstrated here is a first step in this exciting area for future work.

The first piece of good news is that, under the Nearest-Neighbor orchestration method, this set of learned NP programs generalizes to the test set with **69%** accuracy! This is 7% lower than the results recently reported by Cedano *et al.*, but to come even close to a publishable performance level in a problem acknowledged by the *Journal of Molecular Biology* to be difficult and important is a victory for machine learning. The fact that very little knowledge of the problem was input into the system (I am, after all, not a biologist) is some evidence that a biologist guiding PADO could have achieved results comparable to Cedano *et al.* The learned PADO system of NP programs shown and discussed in this section was learned using IRNP. It is worth noting that the highest accuracy PADO system developed without IRNP only generalized with 63% accuracy to the test-set. This 6% generalization improvement due to IRNP is significant in this domain a performance gap of a similar size is all that separates the program created by PADO from the results published in the Journal of Molecular Biology in 1997, as cited.

Soon after Cedano's 1997 paper came out, [Koza *et al.*, 1998] reported on a GP attempt to tackle a similar problem using the same training and testing data. This work reported on the problem of discriminating class 4, the *transmembrane domain*, from the other 4 classes in this same domain. After extensive evolution on an enormous population (500,000 functions), a function was evolved that generalized to the test set with an accuracy of **89%**. As a further study of the NP programs shown above, the learned NP program shown in Table 7.8 was tested on its ability to discriminate amino acid sequences belonging to class 4 from amino acid sequences from the other four classes. That NP program generalized to the test set with an **86%** accuracy, which is impressive given that the NP program was learned in a population of 250, not 500,000.

7.7 Hand-held Images

7.7.1 Description of the Domain and Problem

Elements in this experiment are 256x256 real video images with 256 level of grey. This domain has seven classes: *Bear, Long flute, Pan flute, Thermos, Book, Racket* and *Other*. The class *Other* is a collection of images which are empty or show a hand holding some object (like a cup or a ball) that is not one of the other six classes. All pictures are taken against a variety of solid colored backgrounds and contain part of a hand and arm. The hand holds one of the objects, often partially occluding it. The location and rotation of the object is only constrained so that the object is completely in the image. The lighting varies dramatically in intensity and position. The distance from the object to the camera ranges from 2.5 to 3.5 feet. The objects are never severely foreshortened. See Figure 7.20 for a sample training and testing image from each class.

7.7.2 Setting PADO up to Solve the Problem

In each of the two experiments in this section, the total population size was 1750 (i.e., 250∗7). Each point on each graph is an average of at least 65 independent runs. A total of 105 (15 from each of 7 classes) images were used for training and a separate set of 105 (15 from each of 7 classes) images were withheld for testing afterwards. Notice that this is less than a third as many training examples as was used in the experiment Section 7.3.

The PSPs used in these experiments 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 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).

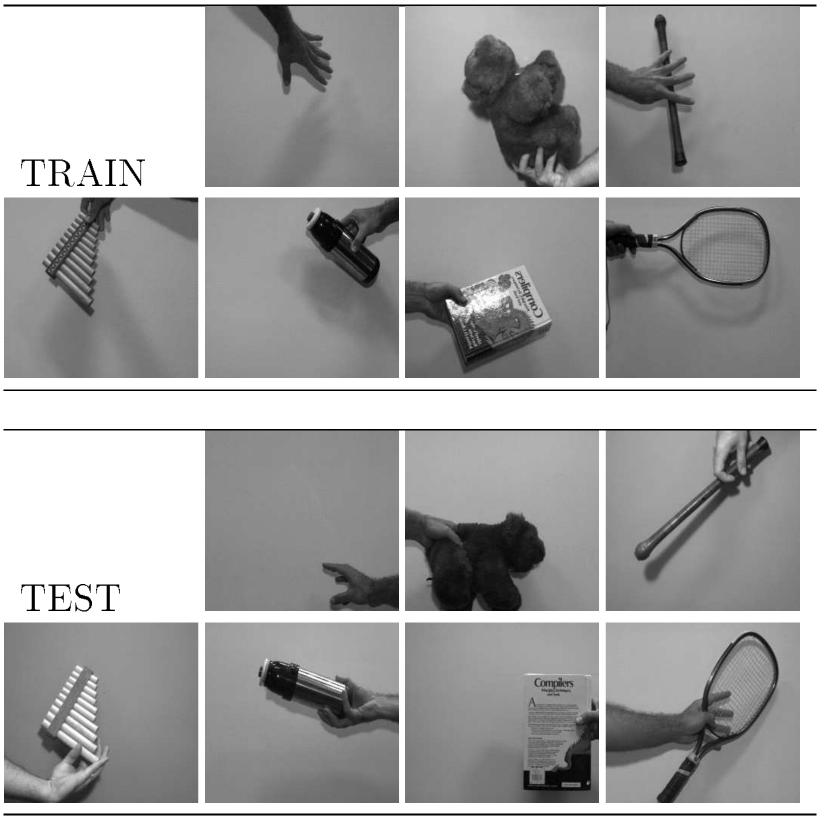


Figure 7.20: Random training and testing signals from the 7 classes in this domain.

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).

Notice that these are identical to the PSPs used in experiment Section 7.3.

7.7.3 The Results

Figure 7.21 plots the *mean* generalization performance on a separate set of testing images for each successive generation. Both curves in Figure 7.21 use IRNP; the difference in the curves is due to the use of the same pair of orchestration techniques used in the other experiments: Weight-Search and Nearest-Neighbor orchestration.

The most obvious feature of the curves in Figure 7.21 is that PADO's performance on this domain is significantly less good than in the other image domain presented in Section 7.3.

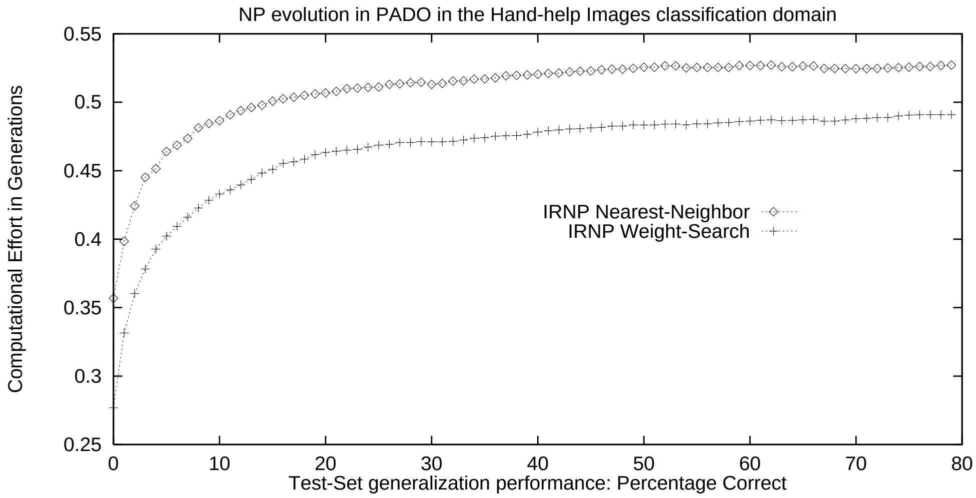


Figure 7.21: Hand-held Images: NP learning with IRNP using both PADO **Nearest Neighbor and Weight-Search** orchestration.

More significantly, PADO performs less well in this domain than previous incarnations of PADO have done on this exact same domain [Teller and Veloso, 1997]. Both of these phenomena can be explained by the same underlying cause: scarcity of training data.

As noted in Section 7.7.1, there are very few training examples given to PADO. Learning the difference between these sort of video images is a very difficult problem. Given the PSPs PADO is given (that clearly do not include any domain specific information), the problem is much harder. Learning to discriminate these classes from each other is yet again harder with only 15 examples from each class to learn from (compared with 50 examples from each class in Section 7.3). In [Teller and Veloso, 1997], this problem was addressed by adding to PADO a whole regime for adding noise to video images. This regime effectively increased the number of available training examples from each class from 15 to about 75. This increase translated directly into the approximately 10% performance advantage that this older version of PADO exhibited over the thesis instantiation of PADO. Still it is encouraging to see that despite all of these difficulties, PADO still manages to find a general classification system well above the 14.28% guess a class at random would achieve.

7.8 Experiments Summary

The following general conclusions can be drawn from the empirical evidence presented in this chapter:

- PADO was run successfully on a variety of different signals:
  - Generic signal domain
    – Generic signal domain artificially generated – four classes.
  - Natural Images I
    – Natural Images I for visual object recognition – seven classes.

  - Acoustic Signals
    – Acoustic signals of army noises for classification – seven classes.
  - Amino-Acid Sequences
    – Protein subsequence identification – five classes.
  - Natural Images II
    – Natural Images II for recognition of hand-held objects – seven classes.

- IRNP increases the learning rate across domains.
- NP programs perform better when given more time to examine the signals.
- The orchestration method employed has a significant impact on PADO's performance.
- IRNP is not significantly sensitive to the orchestration method employed.
- PADO is not significantly sensitive to its learning parameters.

Chapter 8

Related Work

We now situate our work on PADO, Neural Programming, and Internal Reinforcement within the larger context of machine learning in AI. While it is nearly impossible to read everything that has been written on a field the size of algorithm evolution, we believe that PADO, NP, and IRNP represent important original contributions to the field. Therefore, this chapter focuses on the similarity between our work and other research programs rather than their differences.

In general, mechanisms already exist with at least some of the stated goals of this research program. The most popular mechanism with many of the same goals as PADO is Artificial Neural Networks (ANNs). ANNs do take signals as input and produce a much smaller signal (interpretable as a symbol) as output (e.g., [Pomerleau, 1992, Intrator *et al.*, 1995, Sebald *et al.*, 1991]). To some extent the ANN mechanism works independent of the signal type, *as long as proper pre-processing of the signal and ANN parameter tuning is done.* However, it is also fair to say that ANNs pay a price for increased signal size which is always linear and often closer to quadratic in the input size increase. This price is not only in the time it takes to train the extra weights introduced by increasing input size, but more importantly, this increased number of degrees of freedom means that overfitting happens early, unless significantly more training data is provided. This relationship (larger input signals $\rightarrow$ larger required training set sizes) forces a fairly low upper-bound on the size of signal elements that can currently be given to an ANN tractably. In addition, ANNs are committed to the specific real-valued representation that drives their learning process. This means that they can accept any signal type *that can be preprocessed to fit into this input paradigm.* This is another restriction that the ANN process brings upon itself. PADO seeks to overcome, among others, these two obstacles: sensitivity to signal size, and static use of domain knowledge (pre-processing).

Because PADO aims to contribute both to algorithm evolution in machine learning and to automatic signal understanding in the signal processing field, the next two subsections will give background on those two areas of research.

8.1 Algorithm Evolution

Neural Programming is an extension of the genetic programming paradigm. Genetic programming (GP) is a term for the automatic generation of programs by means of natural selection. Genetic programming as a term began with [Koza, 1992], but the origins of algorithm evolution are much older. As long back as the 1960's, work like [Fogel *et al.*, 1966] was laying the ground work for what genetic programming has become today. [Cramer, 1985] was the work that formed the bridge from traditional GA to the field that has since become genetic programming.

The NP and IRNP approaches are used, in the context of our work, for signal classification problems. Genetic programming has been applied to a number of visual and acoustic classification problems. In most of these cases, this work has looked at small images (e.g., font bitmaps [Andre, 1994, Koza, 1994]). In the cases where larger signals are examined, GP is almost always used as an aid, not the actual program that examines the signal directly (e.g., [Nguyen and Huang, 1994, Tackett, 1993, Daida, 1996]). Time series prediction has also been the topic of some GP research, but again, successful only in specific cases (e.g., [Oakley, 1994]). The use of arbitrary memory was introduced into genetic programming in [Teller, 1994a] and later papers have examined aspects of memory and data structures in evolved programs (e.g., [Langdon, 1995, Andre, 1995, Andre and Teller, 1996, Brave, 1996a, Langdon, 1996]). [Teller, 1994b] demonstrated that GP, with a few paradigmatic additions, was Turing complete. Some research has been done on recursion and looping in GP (e.g., [Kinnear, Jr., 1993, Brave, 1996c, Langdon, 1995]), but how to tractably evolve complex programs with iteration and/or recursion and extensive, effective memory use is still very much an open question.

GP is by no means the only machine learning method that employs evolution as a learning model. Of the most immediate relation to this research, Evolutionary Programming (EP) was first described as a method for learning finite state machines (FSMs) [Fogel *et al.*, 1966]. More recently EP has, as a field, moved to evolving vectors of real numbers. Some work in EP still is related to aspects of PADO and the NP representation, and IRNP. [Fogel *et al.*, 1995] is a good example of this work. In [Fogel *et al.*, 1995], finite state machines are evolved and used to predict time series data. FSMs are topologically identical to NP programs. What happens at each node though is entirely different. FSMs are flow of control and the "computation" done at each node is simply to decide which node (state) to pass control to next based on the next input symbol. FSMs can also be described as data-flow machines, but the effect is the same. [Fogel *et al.*, 1995] is notable in that a self-adaptation strategy is used to try to improve the mutation operator acting on these FSMs which has a similar goal to that of IRNP, though the method for solving the problem is very different.

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.

8.2 Signal Understanding

Signal Understanding[1] is a large field and certainly cannot even be summarized here. There are a number of related fields all of which, to some extent, can be thought of as addressing the signal understanding problem: Computer Vision, Machine Learning, Digital Signal Processing, Pattern Recognition, etc. Each has been applied to some of the sorts of signal understanding problems addressed in this thesis.

From an AI perspective, the signal understanding field can be divided into two subfields: human solutions and machine learning. The eventual goal of machine learning in signal understanding is both to improve the performance of human created solutions as well as to act as a labor saving device where human created solutions are too expensive to obtain or are needed in situations in which humans cannot participate. The state of the art is far from but moving towards this goal. This thesis avoids the topic of human coded solutions because it is not the primary goal of this thesis to surpass human coded solutions in important signal processing problems. In fact there is a synergistic, not competitive, relationship between this work and the main stream of fields such as computer vision.

There are some signal-specific approaches to solving the signal understanding problem. For example: computer vision. The bulk of the lessons learned in computer vision are not transferable to other signal understanding areas such as the deciphering of acoustic information. Generally, such signal-specific fields are fields in which the understanding about that specific signal type is driven by the innovations of the researchers.

On the other end of the spectrum is machine learning. A good part of the effort in machine learning is aimed at attacking one part or another of the signal understanding problem. In machine learning signal understanding the primary goal is not to increase understanding about the signal type, but for the researchers to innovate in the machine learning techniques so that the machine learning systems themselves can learn more about the signals in question. At this level of description, the work of this thesis falls on this, the ML, end of the spectrum.

Another distinction in the signal understanding field is between identifying the important symbols associated with a particular symbol (often called *Pattern Recognition*) and the problem of actually locating, specifying, or otherwise annotating specific aspects *within* a signal (often called *Object Recognition* in the computer vision field). In this thesis, the applications of PADO focus on classification of the whole signals rather than dissecting them, but PADO could as easily be applied to these related "symbol location" problems.

Within machine learning, ANNs have enjoyed the largest share of success stories in both visual and acoustic signal domains (certainly the two most important and popular signal domains). It would require the entirety of a thesis this length to adequately summarize a topic as large as ANN activity in visual and acoustic domains in the last 10 years. Two of the visual areas in which ANNs have appropriately received the most attention are in the autonomous road following domain (e.g., [Pomerleau, 1992]) and the face detection and/or recognition domain (e.g., [Hancock, 1990, Intrator *et al.*, 1995, Viola, 1993, Turk and Pentland, 1991,

[1] Signal understanding is a potentially over-general term which is used on purpose in this thesis. Signal classification is certainly a major form of signal understanding. However, if, for example, we map an image to a continuous variable (e.g., temperature) then we are doing signal-to-symbol mapping without a finite set of classes. PADO, in its final form, is intended to be able to capture many types of signal understanding.

Rowley *et al.*, 1995, Sung and Poggio., 1994]). In the general acoustic domain, ANNs have been applied to problems such as phoneme classification (e.g., [Waibel *et al.*, 1989]) and word classification (e.g., [Tebelski, 1995]). In all of these cases, preprocessing was required for the neural network and most of the expertise of the research involved understanding not only the domain, but how it interfaced with the ANN topology and reinforcement policy.

Conrads, Nordin, and Banzhaf have done some nice work on speech discrimination using GP [Conrads *et al.*, 1998]. Jason Daida did work on image recognition [Daida, 1996]. Walter Tackett did some feature discovery and image discrimination work finding tanks in forest scenes [Tackett, 1993]. Nordin and Banzhaf did some work on image and sound compression using GP [Nordin and Banzhaf, 1996]. Poli has done some good work on the evolution of image filters for analysis of MRI images [Poli, 1996a].

8.3 Orchestration

This is not the first paper to discuss the orchestration problem in its general form. [Selfridge, 1966] is an interesting description of this exact problem from over thirty years ago. Nor is this research the first time that the fitness function in evolutionary computation has included some measure of "cooperation." [Haynes *et al.*, 1995, Luke and Spector, 1996, Haynes and Sen, 1996] are examples of the evolution of behavior coordination. In these papers, the "teams" are explicitly grouped, leading to a natural incorporation of cooperation into the fitness function. [Wolpert, 1992] gives a very thorough theoretical account of "stacked generalization." The general conceit of stacked generalization is that instead of having a learning algorithm entirely solve a problem, one or more models can be used to partially solve the problem. Then, the output of that model(s) can be "stacked" as inputs to a new learner. Though the description is very different, the orchestration problem can be seen as a specific difficulty in stacked generalization. This work has attempted to address some of these specific difficulties.

8.4 Function Sensitivity Approximation

There is an entire field of study called *Automatic Program Differentiation.* For a good overview of this field, try [Griewank and Corliss, 1991]. The tools and techniques from this field are generally more thorough and exact than the Function Sensitivity Approximation presented in this thesis. Briefly, this section will present the attributes of Function Sensitivity Approximation that distinguish it from the bulk of the work done in Automatic Program Differentiation.

Firstly, much of the work done in Automatic Program Differentiation seeks to use the chain rule to develop symbolic partial derivatives of the outputs with respect to the inputs (e.g., [Chavent *et al.*, 1996]). A lot of the work in this field does, however, concern itself with differentiating non-differentiable programs (e.g., [Beck and Fischer, 1995]. One of the basic assumptions made by this field is that they have access to the code to be differentiated. This is not an assumption violated by the needs of PADO and NP, but it is an assumption that Function Sensitivity Approximation does not need to make.

With the exception of one or two large complicated systems (e.g., [Bischof *et al.*, 1992]), none of the applications developed in this field works well with very large numbers of inputs. For example, imagine a four parameter PADO PSP examining a video image. On one level, it has four parameters. On another very real level however, it has 65,536 inputs if each video image is 256x256 pixels, since this video image can change independent of the four input parameters and independent of the PSP code in question. This is a clear advantage of Function Sensitivity Approximation.

The derivative of an input is a *local* property. If small perturbations to an input have no change, the attitude of the Automatic Program Differentiation field is that they can get no information about what large changes might do. This is true, but since what NP needs is to know the *sensitivity* (which is not, as just pointed out, the same as the derivative), this is a serious limitation.

And finally, the techniques developed by this field usually assume that the application for doing the automatic differentiation will have to be run *once per new control path* in the program [Griewank and Corliss, 1991]. Of course for complex programs this could take an extremely long time. Function Sensitivity Approximation is not free from this problem, but it does not need to explicitly wait until all control sequences are seen before it has an approximation to the sensitivity (by definition).

In summary, the needs of NP for refining the Credit-Blame map to evolve Turing complete programs is "how likely is a *random change* in input a_i to cause the function output to change *at all*." The reason Function Sensitivity Approximation has not duplicated the work of this field is largely because this is not the typical problem they set out to solve.

8.5 Work Related to Thesis Contributions

In order to frame this section's descriptions of related work, here is a summary of the contributions of this thesis:

- Algorithm Evolution
  - Evolution of multiple evolved specialist programs.
  - **Neural Programming**: Algorithm representation for program evolution that facilitates internal reinforcement.
  - **Internal Reinforcement**: use of a Credit-Blame map for focused, principled genetic search operators.
    - **Function Sensitivity Approximation**: automatic analysis of function input/output sensitivity.
    - **Credit-Blame map**: creation and sensitivity-refinement of credit-blame assignment of program internals based on program behavior.
- Learned Signal Understanding
  - **Orchestration**: Classification decomposition and discrimination task solution synthesis.

- **Parameterized Signal Primitives**:
  * Arbitrary signal size and type independent of program size.
  * Expert knowledge input at the symbolic (programmatic) level.

Both GP (e.g., [Koza, 1992, Banzhaf *et al.*, 1998]) and ANNs (e.g., [Rumelhart *et al.*, 1986]) continue to be well investigated. 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. To the best of our knowledge, in the context of ANNs and principled update policies like backpropagation, these investigations have not yet been extended to arbitrary, potentially non-differentiable functions like those typically used by human programmers and by GP programs.

An important current aspect of the PADO architecture is the orchestration of many programs of similar behavioral goals into a single sub-system of the total PADO system. These sub-systems are then orchestrated into the total PADO signal understanding system. This is not the first use of multiple experts to improve the performance of a system. In AI, the general concept dates back at least as far as the pandemonium architecture [Selfridge, 1966].

Papers like [Blum, 1995] point out the advantages of "asking the opinion of more than one expert." The Winnow [Littlestone, 1988] and Weighted-Majority [Littlestone and Warmuth, 1994] algorithms, for example, utilize this very fact to, in some case, improve performance of the underlying algorithm. [Bielak, 1993b] is a thesis in which the possibility of using multiple experts for separate domains, or different modalities of the same domain, can be coordinated, often with large positive effect. PADO's orchestration, which is an extension of these ideas, specifically notices and incorporates the readily available pool of different experts found in an evolutionary setting.

Iterative deepening in genetic programming has been a peripheral focus of this thesis work. Iterative deepening in program space is another interesting way to try to address the evolution of algorithms in a principled way. This idea has been investigated in the ADATE work [Olsson, 1995]. In ADATE, iterative deepening is done on minimum description length codes, where the codes are made up of program encodings and associated errors. In ADATE this encoding is in the form of ML programs. So ADATE is doing iterative deepening in a sort of algorithm evolution.

The ADATE work creates a very different representation for algorithm evolution with the goal of improving the efficiency of the search [Olsson, 1995]. In ADATE, programs are evolved in pure functional ML (i.e., no loops, only recursion). ADATE does not use mutation and crossover, but instead defines a set of *transformations* that are always syntactically legal, type legal, and, with some of the transformations, guaranteed to produce a new program that performs at least as well on the training set (though of course that guarantee does not extend to a random sampling from the same distribution of inputs). However, these program transformations do not react to the *behavior* of the programs, nor target specific aspects of a program for change because of those observed behaviors.

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.

The fact that its similarity to NP is in representation, not in use or objectives (i.e., IRNP), [Poli, 1996b] is worthy of note. Poli describes Parallel Distributed Genetic Programming (PDGP), a type of genetic programming which can be used for the development of parallel programs in which symbolic and neural processing elements can interact. PDGP is based on a graph-like representation of the parallel programs. These programs can be changed through crossover and mutation operators in the style of the traditional GP paradigm. [Angeline, 1997a] is another recent example of GP moving from the traditional tree structured representation out into the more general world of general graph representation for program evolution.

[Baxter and Bartlett, 1998] is an example of using nearest neighbor as the top level discriminator for classification. This particular work focuses on a distance metric called *Canonical Distortion Measure* and shows that this is optimal for 1-NN when the function classes can be expressed as linear combinations of a fixed set of features. The particular classification example used in this work is a Japanese Kanji OCR problem.

There are certainly other explorations of algorithm evolution in which the representation is changed with positive results. One of the most notable examples is [Nordin, 1997b], which describes evolution or induction of register machine code. This machine learning technique (called CGPS) is a variant of genetic programming applied to evolution of binary machine code for a real computer (a Sparc 5). This technique turns out to be very machine efficient. This research work includes applications to robot control as well as image and sound processing.

In GP, the focus of investigation for increased power of the technique has not generally been on changing the GP representation or on finding principled (non-random) update policies, as in our work. Some work has, however, been done in these areas. [Rosca and Ballard, 1996] describes a process for trying to find sub-functions in an evolving GP function that are more likely than randomly selected ones to contribute positively to fitness when crossed-over into other programs. [Angeline, 1996] 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.

As was described in Section 6.1, issues of credit-blame assignment are central to the NP representation and the IRNP procedure. The IRNP procedure uses a form of bucket-brigade to deal with part of the credit assignment problem. The contribution of this work is, however, not the bucket brigade solution to the credit assignment problem. The contributions of NP and IRNP do include the identification of a credit assignment problem in EC and the application of the bucket-brigade algorithm to help tackle that issue.

When looking at related work in the credit assignment area, even restricting our attention to the field of AI, the problem of credit assignment has been discussed in a wide variety of contexts. The bucket-brigade algorithm is one of the oldest versions 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 (though that is not the historical order of the two ideas) [Watkins, 1989]. Back propagation is another form of TDM and so the connection can also be made to the bucket-brigade algorithm. This connection is brought out in works such as [Cribs and Smith, 1996]. For an excellent short introduction to some of these issues, see [Wilson and Goldberg, 1989].

Chapter 9

Conclusions and Future Work

9.1 Conclusions

9.1.1 Reviewing the Motivation and Goals

One of the original motivations for this thesis was the impression that, given the slow progress of existing machine learning techniques in the general signal-to-symbol domain, a significantly different direction would be worth exploring.

Evolutionary computation, and genetic programming in particular, were chosen as the general models on which to base this work. GP was selected because it was a relatively unexplored area and because genetic programming has the opportunity to develop much richer learned models for addressing the signal-to-symbol problem than many of the more established learning mechanisms.

The other primary motivation for this thesis was the conviction that genetic programming has been under-appreciated as an AI technique because the unfocused nature of the genetic search operators makes the GP field seem unfinished. This led naturally to a research agenda of finding a way to introduce principled internal reinforcement into the GP process.

Once the decision had been made to create *PADO*, a domain independent signal-to-symbol mapping tool using GP, and to use PADO as a vehicle for exploring intelligent search in program space, a number of secondary motivations were created for the thesis. PADO ought, when possible, to maintain the advantages of current machine learning techniques for signal-to-symbol mapping and to avoid the disadvantages of those techniques where possible.

An important advantage of popular existing machine learning techniques is that *why* the techniques work is well understood, thereby generating faith in those methods. One of the primary motivations of this thesis has been to provide a credit-blame assignment approach to program evolution along with a more principled search mechanism using that credit-blame assignment. The goal has been to create the groundwork for this same kind of understanding that can allow GP to move in from the margin of the field of machine learning. This work has not focused simply on gradient descent in program space, but rather on the problem of how to approach gradient descent in program space within the context of evolution. This thesis has bridged the credit-blame assignment gap by finding a way in which explicit and empirical credit-blame assignment can find mutual benefit *in a single machine learning technique.*

9.1.2 Reviewing the Approach

PADO stands for Parallel Algorithm Discovery and Orchestration and has been specifically designed for supervised learning of signal classification problems. PADO's high level approach is to divide a given C class signal classification problem into C different simpler discrimination tasks. PADO learns programs to solve each of these discrimination tasks using simulated evolution as the learning mechanism. Then these evolved programs, specialists in different areas of the classification problem, are orchestrated into a single system that classifies signals from the domain in question. This orchestration can happen at two different levels and in a variety of different ways, as presented during this thesis. A nearest-neighbor orchestration strategy was presented that favored the most reliable of the experts, but also took advantage of non-linearities in the ways these evolved specialist programs tended to disagree on confusing signals.

Within the context of PADO, this thesis 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 focused, 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 Function Approximation* was also introduced as a separate contribution of our work. Sensitivity-based bucket-brigade for refining each program's Credit-Blame map leads to a credit-blame assignment of sufficient detail to allow internal reinforcement to perform focused, beneficial search operations during the algorithm evolution.

9.1.3 Reviewing the Contributions

The following is an annotated enumeration of the major contributions of this thesis:

- Algorithm Evolution
  - Evolution of multiple evolved specialist programs.
  - **Neural Programming**: Algorithm representation for program evolution that facilitates internal reinforcement.
  - **Internal Reinforcement**: use of a Credit-Blame map for focused, principled genetic search operators.
    - **Function Sensitivity Approximation**: automatic analysis of function input/output sensitivity.
    - **Credit-Blame map**: creation and sensitivity-refinement of credit-blame assignment of program internals based on program behavior.
- Learned Signal Understanding
  - **Orchestration**: Classification decomposition and discrimination task solution synthesis.
  - **Parameterized Signal Primitives**:

    * Arbitrary signal size and type independent of program size.
    * Expert knowledge input at the symbolic (programmatic) level.

- A complete implementation of PADO, NP, and IRNP has been created and tested successfully on a wide variety of real world domains

- Patent

  - Patent on the PADO mechanism for signal classification, learned intelligent recombination of programs, and related claims
  - *Software for Autonomously Learning to Discriminate Between Classes of any Signal Type/A Method of Autonomous Machine Learning.*
  - Issued 7/14/98, Patent Number: 5781698.

9.2 Future Work

9.2.1 Making Better Use of Available Tools

This thesis has made a number of assumptions and simplifications in order to make the important points clear. However, there is opportunity for further dramatic improvements in how PADO and NP act and this opportunity can even be translated into other algorithm evolution arenas.

For example, mutation in NP was described as a choice between six different mutation types and it was simply announced that, for this thesis, all six mutation types would be chosen with equal probability when a mutation was to take place. This is not the only option.

To begin with, it is empirically the case, that some of the mutations are simply more effective than others, or to put it differently, there are uneven probability weights for the six mutation types such that runs go "better" on average. The Swap-Arc mutation is an example mutation that benefits from added attention.

Of course there is a finer grain at which these probabilities can be adjusted. While it is possible to pick a set of uneven probabilities that work better across a large range of problems, for each domain these probabilities can be adjusted for even higher performance. A complete study has not been done but preliminary investigation indicates that this higher performance is likely to be achievable in this way.

An interesting area of future work related to these suggested speed-up mechanisms is the following. Instead of picking one probability vector for mutation for all problems, or even picking a new probability vector for each new domain, it is possible to pick a new probability vector for each generation (and this could be done domain dependently or independently). Table 9.1 shows how this would work.

The result of this process is that evolution will now have a detailed *Mutation-Schedule*. This mutation schedule can be used while it's being created and, after some point, it can be fixed and just used as is. This mutation schedule can be used domain independently or domain dependently. The goal of such a mutation schedule would be to pick up "rules" of

MutationSchedule[i][j] $= 0$ (for $1 \leq i \leq$ MaxGenerations and $1 \leq j \leq 6$)
For each learning session
For each generation i within a learning session
For each mutation to be done on program p with parent p'
Pick a mutation type k based on the probabilities from MutationSchedule[i]
Associate with the mutated program, mutation type k and the parent fitness $G_{p'}$
Test the new, mutated program to determine fitness G_p
$\mathbf{MutationSchedule[i][k] = f_{wa}(MutationSchedule[i][k], f_c(G_p, G_{p'}))}$

Table 9.1: Outline for the creation of a *Mutation Schedule*. Function f_{wa} is *weighted average* so that each trial is equally weighted as MutationSchedule picks up an increasing number of sample points. The f_c returns 0 if G_p is worse than $G_{p'}$ and otherwise returns the probability that G_p is actually better than $G_{p'}$ (as opposed to simply having had it's fitness inaccurately approximated as higher than $G_{p'}$).

the form "Early on, do a lot of changing what's **in** a node and then later in evolution, do more changing of how those things are **connected**."

At first glance, such a mutation schedule seems like its worst case performance is as good as using an even probability vector (since it can learn that that is what really drives evolution the fastest). But evolution is now quite so simple. The driving force for changing the mutation schedule is how much better the child turned out to be than the parent after the application of a particular mutation type. The hope is that by maximizing the size of these one step improvements in the program evolution process, the *end result* many generations later will be better. This is a greedy assumption that may be true but has never been generally demonstrated in EC.

9.2.2 Taking More Advantage of Sensitivity Information

Function Sensitivity Approximation provides, for each function, how sensitive the function's output is to each input parameter, *for a range of different numbers of inputs*. While each of these pieces of information is taken advantage of in the refinement of the Credit-Blame map, the relation between these different values for a single function is currently not exploited.

For example, many functions that can operate on a variable number of inputs become much less sensitive to each of the inputs as the number of inputs increases. MULT is such a function. It can take 2 or more inputs and return the multiplication of those inputs. To prevent overflow, there will be some return value above which that ceiling is returned instead of the actual product of the inputs. In such a case, as the number of inputs grows the sensitivity of the function to each particular input goes down. MEAN is an example of a function that does not act in this way (as long as the return values have a few digits of precision).

IRNP could be improved by watching for these changes in the input sensitivity of some functions and then adjusting mutation and crossover accordingly. As an example, suppose that there is a node with a low credit score and suppose this node's function is MULT. If this node has inputs from a number of high credit score nodes, IRNP would currently mutate the node function in an attempt to take better advantage of the incoming high credit values. But knowing that MULT is not very sensitive to any of its parameters when there are too many of them, it would be worth trying to remove one of the inputs (even if it comes from a high sensitivity node) in the hope that the MULT node responds more actively to the remaining inputs.

This "sensitivity to sensitivity" extends to the parameterized signal primitives as well. For example, suppose some visual processing primitive that acts on rectangular regions (a_0, a_1, a_2, a_3) of the input images is defined so that, if only three inputs (a_0, a_1, a_2) are provided then the PSP makes the assumption that $a_3 = 2 * a_1$. By looking at the range of sensitivity values for different numbers of inputs, IRNP could notice and take into account the fact that the output value is highly sensitive to the value of a_1 and that this effect can be smoothed out by adding a fourth parameter.

9.2.3 Extending IRNP Bucket Brigade

As described in this thesis, the Credit-Blame map for an NP program is created by first identifying program aspects (nodes) that are directly related to the task to be learned. Second, this Credit-Blame map is refined to reflect the topology of the program through a bucket brigade backpropagation of the credit scores accumulated in the first approximation to the Credit-Blame map. The effect of this second phase is to make sure that the Credit-Blame provides credit not only to the nodes that create "good" values, but also to nodes that contribute to the creation of those values.

The Credit-Blame map could be improved further through an additional step: *forward propagation of blame*. As described in this thesis, the blame that exists in the Credit-Blame map is only a lack of reward. If we could find something specific to blame a node for, we could explicitly refine its credit score to reflect this punishment. The key insight in forward propagation of blame is that nodes are "bad" if they receive "good" values and output "bad" values. This can be operationalized as shown in Table 9.2.

Until no further changes
  For each node x in the program
    For each arc (x, y) of that node
      if $(\mathrm{CS}_x > \mathrm{CS}_y)$
        $\underline{\mathbf{CS}_y = f_b(\mathbf{CS}_x, \mathbf{CS}_y)}$

Table 9.2: The process of bucket brigading the Credit Scores (CS) throughout an NP program for forward propagation of blame.

The real question of course is, what should f_b be? The problem is that if f_b always reduces CS_y in this case, then this is equivalent to simply setting CS_y to 0. This was left as a piece of future work exactly because this (always setting CS_y to 0) is undesirable.

9.2.4 The Next Level of Proactive Program Changes

One of the benefits of explicit credit-blame assignment is that, at least in some techniques (e.g., ANNs), it is possible to actually make a change to the model with the goal of adjusting specific values *in specific directions*. For example, in ANNs, backpropagation does exactly this: each node is labeled with a desired change in its output value *and then weights are altered to locally accomplish this change*.

The same functionality could be added to IRNP, though probably not as the dominant feature as it is in ANNs. There are some functions (e.g. ADD) for which the derivative exists. So if there is a node in an NP program and it has been determined that that node's output should be increased, this can be accomplished recursively by raising one or more of that node's input values. Such backprop-specific change suggestions can be passed through many functions (e.g. ADD, SUB, MULT, COS) including some functions whose derivative is undefined (e.g., IF-THEN-ELSE).

The reason that this is not currently the central theme of IRNP is that, because the NP representation accepts arbitrary functions, some of those functions will have the attribute that no amount of investigation (e.g., function sensitivity approximation) will illuminate how the inputs should be changed in order to affect the output in a specific way.

Take for example, the function PSP-AVERAGE that takes four inputs (a_0,a_1,a_2,a_3) and returns the average pixel intensity in the region of the current video signal described by the rectangle with upper left corner (a_0,a_1) and lower right corner (a_2,a_3). The derivative of this function may be as complicated as the set of images that the function will be exposed to! And this function, PSP-AVERAGE, is at least continuous; if one of the four parameters is changed by a small amount, the output will be fairly close to what it was before the change. There are some functions (e.g. PSP-MAX) which are discontinuous, which poses its own problems.

9.2.5 META-Orchestration

This thesis has described a number of orchestration strategies. All of these strategies were meant to be executed at the end of each generation and, for that reason, could not be excessively expensive. However, there is another kind or level of orchestration that would undoubtedly be a profitable avenue of future work. There is no reason that at the end of an entire learning episode (e.g., after 80 generations of evolution), a more expensive, more thorough form of orchestration could not be performed.

This META-Orchestration could take a number of directions. There could be a more thorough search of weight space or a more thorough search for a set of programs that work well together. This would be an appropriate point to orchestrate multiple programs from each discrimination pool as well as each of the C different discrimination-solutions. Nearest-Neighbor orchestration could be run and considerably more time could be spent customizing the warping of the space to maximize the expected out-of-sample performance.

Beyond more in-depth orchestrations in the veins already discussed, there are more complex options as well. In the extreme, the outputs of some of the best programs could be used to create values that instead of being directly orchestrated, are fed into another learning system as input. In fact both standard GP and ANNs are good candidates for such

META-learning, and of course this META-learning is just a more complex sort of learned orchestration.

Appendix A

Notation Table

Notation	Description
a_i	The ith input to a node
$a_{x,i}$	Input number i into a specific node x (when there is potential confusion)
A	Number of inputs into a node
B	Number of programs from each discrimination pool orchestrated by F_L.
C	The number of classification classes
CS_x	The credit score for node x
$D_{x,t}$	Indicates the evaluation of node x on timestep t
E	The testing signals
E_i	The ith testing signal
$\|E\|$	The number of testing signals
F_L	The lower level orchestration procedure
F_H	The higher level orchestration procedure
G_p	The fitness for NP program p
H_x^i	Compressed value of what node x output during examination of S_i
i, j	Generic loop variables
J	Number of biased-random groups P_j^i. $(1 \leq j \leq J)$
K	The number of programs in each reproduction selection tournament
L	The labels associated with the training signals
L_i	The ith training signal label
$\|L\|$	The number of labels (always the same as $\|S\|$)
M	The number of programs in the population
N_p	The number of nodes in program p
$\vec{\mathcal{N}}$	The space used for computing the Nearest-Neighbor quantities
O	A node output

Notation	**Description**
$\mathcal{O}$	Gaussian random number generator
p	An NP program
P_j^i	Evolved-Orchestration group j for discrimination pool i
Q	Generic loop range maximum
R	The response of an evolved program
$\mathcal{S}_{f,A,i}$	The sensitivity of function f with A inputs to input number a_i
S	The training signals
S_i	The ith training signal
$\lvert S\rvert$	The number of training signals
t	The $time$ loop variable
T	Time loop range maximum (MaxTimeSteps)
x, y, z, u, v	An NP program node
(x, y)	The directed arc from node x to node y
$\vec{W}, \vec{V}$	A vector of weights
Z	The set of programs to test during orchestration learning
$\mathrm{OUT}, \mathrm{fib}, f, \ldots$	Arbitrary functions
α	Measure of drag on an airplane wing under design
β	Measure of lift on an airplane wing under design
δ	A point in $\vec{\mathcal{N}}$ space
ϵ	Threshold for ending the Credit-Blame map refinement process

Appendix B

Turing Complete Programs

The terms *program* and *function* have been used almost interchangeably in GP to date. However, for historical reasons within GP, these are the two names which can best be used to distinguish two very different kinds of languages/representations and the evolving of individuals written in them. The following quote from the latest text book on genetic programming acknowledges this important distinction (using s-expression as a synonym for "function").

> *"The topic we discuss in this section is that PADO works not with s-expressions but with programs."*
>
> – From *Genetic Programming: An Introduction*, 1998.

function is, mathematically, a *many-to-one* mapping from a range to a domain that has, among other things, an execution time that is a bounded function of the size of the element chosen from the range as input. *Modulo-7(x)* is a function. *Arc-Tan(x)* is a function. *Balanced-Parentheses(<some-string>)* is **not** a function. A function is computationally equivalent to a regular language. Functions are often referred to as *elementary functions*. An elementary function is any of the set of recognized elementary functions such as +, *, **COS**, etc. or a composition of such functions.

program is a Turing complete entity composed of elementary functions, loops (and/or recursion), and an inexhaustible supply of state (memory). For a succinct introduction to Turing completeness, try [Hopcroft and Ullman, 1979].

The hierarchy of languages and their computational expressiveness is shown in Figure B.1. See [Hopcroft and Ullman, 1979] for a primer on these distinctions.

A good question is "So? Does this really matter?" The answer, in both theory and in practice is "Yes. It does matter." It matters because programs written in a Turing complete language are harder to evolve and because they are more powerful computationally (computationally expressive).

Evolutionary computation, like most forms of machine learning, depends on the ability to hill-climb. That is, success is predicated on the idea that you can make small syntactic changes to a hypothesis space function/program and that these changes will result in

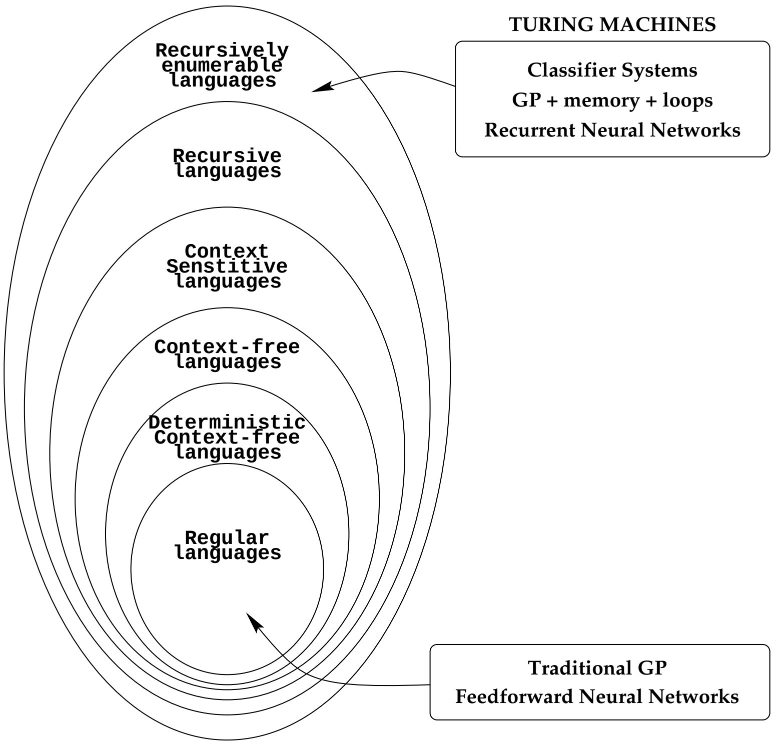


Figure B.1: Expressiveness of the Representation - The Chomsky Hierarchy

semantic (i.e., behavioral) changes small enough that hill-climbing can occur. Figure B.2 illustrates this issue.

In chaotic systems, changes no matter how small, can cause unboundedly large changes in the resulting system's behavior. Regular Languages, the computational class of both Feedforward Neural Networks and Traditional GP, cannot support chaotic systems. Recursively Enumerable languages (Turing complete languages), the computational class of Recurrent Neural Networks and GP+memory+loops, can support chaotic systems. This means that hill-climbing works less well in the more expressive languages. In summary, evolution depends on hill-climbing. Hill-climbing depends on the tight coupling of syntactic and semantic changes. These changes are much less tightly coupled in more expressive languages (e.g.,

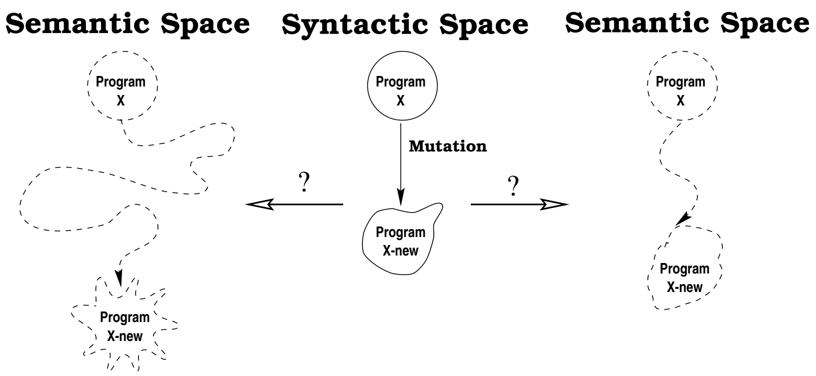


Figure B.2: The distinction between Syntax and Semantics for program evolution.

GP+memory+loops) than in less expressive languages (e.g., simple tree-GP).

The genetic "operators" are the source of search in algorithm evolution (see Chapter 2 for a review of this process). Operators are, by definition, used within the context of a representation for evolution. How "good" a representation is is only measurable *with respect to a particular set of operators.* The conclusion here, taken to heart by this entire thesis, is the idea that operators and representations should be designed *together* so that they coordinate correctly.

Given that programs written in Turing complete language are more difficult to evolve than programs written in less expressive languages, the other clear question is, "So why bother evolving them?" The answer to this is simple. First, programs are more computationally expressive than functions. This means there are things programs can learn that functions cannot. Second, and more immediately practical, even when a problem is posed so that a sufficiently large function could solve the problem, a program might be able to express a solution in much less space and therefore be seen as a better solution through appeals to Occam's Razor or the MDL principle.

B.0.6 The Necessary Theoretical Confession

Technically, theoretically, and correctly, the programs evolved in this thesis are *not* Turing complete. It is a meaningless statement to say that these evolving programs are "practically speaking" Turing complete since Turing completeness is a theoretical label.

There are two reasons that the NP programming language is not theoretically Turing complete: limited memory and limited running time. NP programs as they have been presented here (with or without the use of Indexed Memory) can and do use considerable amounts of memory, but not arbitrary amounts of memory. This could be easily changed by simply allowing the indexed memory to be extended indefinitely as those memory cells were indexed by the program. Clearly though, NP programs even without this addition have the *memory characteristic of programs written in Turing complete languages that, from a practical point of view, adds to the complexity that this thesis is, in part, trying to address.*

The second issue is how long the evolved NP programs are able to run. Programs written in a Turing complete language must have the capability to run for an arbitrarily, possibly infinite, period of time on any particular input. Practically, this is also not possible when trying to learn algorithms as Section 3.4.3 describes in detail. So, in practice, NP programs have a maximum time in which to complete their computation. This means that theoretically, NP programs are not of the same computational power as Turing machines. However, NP programs, even with this time limit have the *running time characteristic of programs written in Turing complete languages that, from a practical point of view, adds the complexity that this thesis is, in part, trying to address.* As good evidence of this, see Section 7.5 in which it is demonstrated that when additional time is made available to the evolving programs they do indeed use those additional iterations to improve their performance.

This is all to say that while it is theoretically inaccurate to say that this thesis is evolving Turing complete programs, this thesis makes that practical claim in the spirit that it communicates approximately where, on the grey continuum of Turing complete-ish languages, the NP programming language falls.

Appendix C

NP Implementation Details

C.1 Initializing an NP program

The results discussed in this thesis are surprisingly independent of the initialization of the NP programs. Never-the-less, this appendix describes the particular manner in which NP programs were initialized for experimental results of this thesis.

All programs are initialized in an identical, independent manner as follows:

1. Pick a number of nodes for program p to begin with, N_p. This value is chosen uniform randomly from the range [Min-Number-Nodes .. Max-Number-Nodes]
2. Assign an action to each node x, $1 \leq x \leq N_p$
   - With 33.3% probability, choose a constant uniform randomly from the range [Min-Constant .. Max-Constant]
   - With 66.7% probability, choose a function uniform randomly from the set of functions made available by the user.
3. Assign arcs for each node x
   (a) Pick a number of inputs for the node uniform randomly between the minimum arity for node x's action and Max-Number-Outputs
   (b) For each input arc needed, choose a *nearby* node y
      - A *nearby* node is a node chosen uniform randomly from the set of nodes within 10% of the same node number in order of creation (as defined in #2 above).
      - For example, if the program has 100 nodes and we're looking for nodes to act as source nodes for which node 72 will be the destination node, those nodes are chosen uniform randomly from the range [Node62..Node82].
   (c) Create the arc (y, x)

C.2 NP and PADO Knobs

This section is an itemization of the knobs in the implementation of PADO and NP. Many of them are only flags for turning on and off functionality that is compared in this thesis.

- Flags

InternalReinforcementMUT Internal Reinforcement for *Mutation* can be turned on and off.

InternalReinforcementCROSS Internal Reinforcement for *Crossover* can be turned on and off.

ClassificationFlag This flag switches PADO between classification and target value approximation modes.

ProgramResponse Only one of these two values can be defined.

UseOutputNodeResponse This flag sets PADO to read the program's response from the collected OUTPUT node values.

UseMemoryBasedResponse This flag sets PADO to read the program's response from Indexed Memory.

- Parameters

NumClasses (C)
The number of classes in the problem to be solved. This value varied throughout the thesis.

NumberTrainingCases $(|S|)$
This is the number of training signals made available to the evolving programs each generation. This number varied between experiments simply because the available data varied from domain to domain.

NumberTestingCases $(|E|)$
This is the number of distinct, unseen testing signals made available to the evolving programs each generation. This number varied between experiments simply because the available data varied from domain to domain. To maintain the meaningfulness of the experimental results, this value was always at least as large as NumberTrainingCases and often 2 to 5 times larger when the data permitted.

PercentChance
The sum of these two values equals the percent of the programs to move *transformed* to the next generation. Except for the section in which the sensitivity of PADO and NP to these values was explored, these two values were fixed to the (fairly arbitrary) values shown.

CrossOverPercentChance 36 (P_c)

MutationPercentChance 60 (P_m)

PopulationSize (250*NumClasses) (M)

For all experiments described in this thesis, this value was fixed to (250*NumClasses). There is nothing special about this value and fixing it across a number of problems from different domains demonstrates that PADO and NP are not overly sensitive to this parameter. Of course, as with all EC techniques, PADO and NP will work better if this value is increased.

NumberNodes (N_p)

NP programs can be of any size in theory, but in practice there must be some limit to how large these programs can grow. To avoid special casing in the NP code, having a minimum program size larger than 4 nodes allowed the system to run slightly faster. The following values were chosen and fixed for all the experiments done in this thesis. These values can obviously be changed and NP obviously performs better when it is allowed to created larger programs.

MaxNumberNodes 80

MinNumberNodes 10

NumberInTournament 5 (K)

This is the number of programs that are compared to pick one that is best to put into the mating pool. See Section 2.1.1. This value was fixed for all the experiments described in this thesis.

NumTimeStepsToRun 10 (T)

This is the number of time steps each program is allowed to run before the program response is extracted and used. NP programs increase their performance noticeably as this number is increased. This number was fixed for all the experiments described in this thesis except as described in Section 7.5.

MaxGenerations 80

Clearly, PADO and NP are independent of the length of time learning continues for. However, in practice, in order to do a statistically significant number of independent trials, this value must be reasonably small. 80 was chosen for historical reasons and fixed for all the experiments described in this thesis.

NumberOfMemoryCells 10

When Indexed Memory is on, it is practical to define some limit to the size of the indexed memory. This could be 10,000 and there is some evidence that this still works fine [Teller, 1994a]. However, this value was picked and fixed for all the experimental results in which Indexed Memory was used. See Section 6.6.

NumberOfFunctions 13

NP program can be built out of any number of atomic functions. In practice though, there must be some pool of such functions to draw from during the initialization and mutation phases of evolution. This value was fixed and used in all the experimental results discussed in this thesis except with the Generic Signal domain in which this value was 10.

MaxNumberOutputs 5
In theory an NP node can have up to (or even more than) MaxNumberNodes outputs. In practice, however, this is unnecessary and takes up a lot of space. This value was fixed (higher than the maximum number of inputs) and used in all the experiments in this thesis.

MaxNumberInputs 4
In theory, an NP node can have up to MaxNumberNodes inputs. In practice, however, this is unnecessary and takes up a lot of space. This value was fixed (lower than the maximum number of outputs) and used in all the experiments in this thesis.

MinFragmentSize 2
When doing crossover, it is convenient to define a minimum number of nodes that must remain in either fragment after the graph cut has been performed. This value was fixed and used in all the experiments in this thesis.

ConstantRange
In theory NP programs can operate with either floats or integers and over any range of numbers. In practice, some maximum and minimum values need to be chosen. The integer range [0..255] was chosen and fixed for all the experiments in this thesis.

MINCONST 0

MAXCONST 255

OrchestrationSearchSize 12000 (Q_l)
One of the Orchestration Options, Program-Search, is to chose a set of programs that work particularly well together (see Section 4.1.2). There needs to be a limit to the number of search steps in this process. This value is that limit and was fixed for all the experiments in this thesis.

OrchestrationTournamentSize 4
In Program-Search, this is the greediness parameter that adjust how new program groups are examined.

SearchOWeightsSize 12000 (Q_l)
One of the Orchestration Options, Weight-Search, is to choose a set of weights that help a fixed group of programs work particularly well together (see Section 4.1.3). There needs to be a limit to the number of search steps in this process. This value is that limit which was fixed for all the experiments in this thesis.

MutationTypes
Given that NP currently has six mutation types, when a particular mutation occurs one of these six mutation types must be chosen. The six values below are relative weights. Because they are all set to 1, all six have an equal probability of being chosen, as described in Section 6.3.1.

SwapArcChangeChance 1

FunctionChangeChance 1

AddNodeChangeChance 1

DeleteNodeChangeChance 1

AddArcChangeChance 1

DeleteArcChangeChance 1

MemAndOutputInitialVal (MINCONST+1)
When Indexed Memory is used, the memory cells must be initialized before an NP program begins execution. This is the value all the cells were initialized to in all thesis experiments that included indexed memory. This value is also used to set the initial values of all arcs and the inital value for each OUTPUT node.

CheckNumberFragments (2*MinNumberNodes)
When internal reinforcement crossover is in use, a certain number of cuts in a program must be examined before a "best" one is selected from among them. There are so many possible cuts that any value here would only be a small sampling. This value was chosen and fixed for all experiments in this thesis.

CheckHowManyNodes 50
When internal reinforcement is in use, a certain number of nodes must be examined before a mutation action can be taken. This value is a *percentage* of the total number of nodes that are to be examined in that particular program. This value was chosen and fixed for all experiments in this thesis.

It is interesting to note that we can make NP do population hill-climbing instead of evolution simply by setting:

- NumberInTournament → PopulationSize (M)
- CrossOverPercentChance → 0 (P_c)

C.3 Further Implementation Details

This section describes a few other important implementation details that will help to make clear how NP and PADO were constructed and the trade-offs involved.

C.3.1 Memory use with IRNP

The first important point to note is that IRNP takes an amount of memory that increases linearly with the number of nodes in a program and linearly in the number of training examples. The "outer loop" for EC training can either be of the form:

```
For All Training Examples
  For All Programs in the Population
    Measure This Program's fitness on this Training Example
```

or it can be:

```
For All Programs in the Population
  For All Training Examples
    Measure This Program's fitness on this Training Example
```

In the second case, the memory requirements for IRNP also increases linearly with the number of programs in the population. This is because IRNP can be done independently (i.e., in the same memory space) for each program (trained over the entire training set). But the reverse (each training set signal over all programs) is not true. IRNP is, by definition, computed over all of the training examples.

C.3.2 Implementation of the OUTPUT nodes in NP

This section presents the details of how the OUTPUT nodes in NP work in the thesis instantiation of PADO. All the details below are specific to this instantiation and could be quite different and possibly more effective in future implementations.

As was already discussed, each OUTPUT node on each timestep produces as output the mean of the values on its input arcs. Let us call this output value produced on timestep t to be O_t. As a *side-effect*, each OUTPUT node adds the value $(O_t * t)$ to a variable called *Response* and adds t to the variable *ResponseWeight*. When the timestep threshold is reached, the response of the program is taken to be *Response*/*ResponseWeight*. This means that every input into an OUTPUT is weighted equally, every OUTPUT node is weighted equally (relative to each other), and both of these factors are weighted by time so that the later values are linearly more important.

Notice also that this sort of an implementation is an anytime algorithm. That is, at any point during the NP calculation, an answer can be extracted and that answer is not simply a default value if the answer is not "ready" yet. Instead, NP programs with this implementation of the OUTPUT nodes are actually constantly improving their response and at any point, there is a working response ready to be used (i.e., *Response*/*ResponseWeight*) even before timestep threshold is reached.

Appendix D

Biology Data

D.1 Hydrophobicy/Aliphaticy

The computation for **PSP-Hydro** is the average of the hydrophobicity of each of the amino acids in the specified region. Table D.1 illustrates the lack of agreement on such a seemingly simple chemical property. For the protein folding experiment described in Section 7.6, the Kyte and Doolittle numbers were chosen (for no particular reason).

Amino Acid	Kyte/Doolittle	Engleman	Amino Acid	Kyte/Doolittle	Engleman
Phe	2.8	3.7	Met	1.9	3.4
Ile	4.5	3.1	Leu	3.8	2.8
Val	4.2	2.6	Cys	2.5	2.0
Trp	-0.9	1.9	Ala	1.8	1.6
Thr	-0.7	1.2	Gly	-0.4	1.0
Ser	-0.8	0.6	Pro	-1.6	-0.2
Tyr	-1.3	-0.7	His	-3.2	-3.0
Gln	-3.5	-4.1	Asn	-3.5	-4.8
Glu	-3.5	-8.2	Lys	-3.9	-8.8
Asp	-3.5	-9.2	Arg	-4.5	-12.3

Table D.1: **Kyte, J., Doolittle, R.F.**, A simple method for displaying the hydropathic character of a protein., J. Mol. Biol., 157, 105-132, 1982. **Engleman,D.M., Steitz, T.A., and Goldman, A.**, Identifying nonpolar transbilayer helices in amino acid sequences of membrane proteins. Annu. Rev. Biophys. Biophys. Chem., 15, 321-353, 1986.

D.2 Atomic Weights

The computation for **PSP-Weights** is the average of the mass of each of the amino acids in the specified region. These individual values are shown in Table D.2.

Residue	Symbols	Mass	Residue	Symbols	Mass
ALA	A	71.09	ARG	R	156.19
ASP	N	114.11	ASP	D	115.09
CYS	C	103.15	GLU	E	129.12
GLU	Q	128.14	GLY	G	57.05
HIS	H	137.14	ISO	I	113.16
LEU	L	113.16	LYS	K	128.17
MET	M	131.19	PHE	F	147.18
PRO	P	97.12	SER	S	87.08
THR	T	101.11	TRY	W	186.21
TYR	Y	163.18	VAL	V	99.14

Table D.2: The mass (in daltons) for each of the 20 amino acids. Data after Table 1.1 from T. E. Creighton, Proteins, 2nd ed., W. H. Freeman and Company, New York, 1993.

D.3 Charged Amino Acids

The computation for **PSP-Charged** is the average of "amount" of charge of each of the amino acids in the specified region. For this experiment, we took the simple approach that amino acids D and E contributed a negative charge of 1, H,R, and K contributed a positive charge of 1 each, and all other amino acids were charge neutral. This is a serious over-simplification.

D.4 Van der Waals Volume

The computation for **PSP-VdWVolume** is the average of the Van der Waals Volume of each of the amino acids in the specified region. These individual values are shown in Table D.3.

Residue	Symbols	Van der Waals Vol	Residue	Symbols	Van der Waals Vol
ALA	A	67	ARG	R	148
ASP	N	96	ASP	D	91
CYS	C	86	GLU	E	109
GLU	Q	114	GLY	G	48
HIS	H	118	ISO	I	124
LEU	L	124	LYS	K	135
MET	M	124	PHE	F	135
PRO	P	90	SER	S	73
THR	T	93	TRY	W	163
TYR	Y	141	VAL	V	105

Table D.3: The Van der Waals Volume for each of the 20 amino acids. Data after Table 1.1 from T. E. Creighton, Proteins, 2nd ed., W. H. Freeman and Company, New York, 1993.

Appendix E

Example Programs

This program is the full version of the program shown in Section 6.5.

Node	CreditScore	Function	Output Arcs	Node	CreditScore	Function	Output Arcs
0	0.000000	060	12_0 1_1 20_1 0_1	1	0.000000	CLOCK	1_0
2	0.000000	144	4_0	3	0.668100	CLOCK	19_2 4_2 45_0
4	0.000000	211	3_0 14_3	5	0.118573	CLOCK	15_0 27_2 19_3 5_0
6	0.432689	094	44_1 20_3 6_0 27_3	7	0.085486	182	29_0 32_0
8	0.001133	145	35_2 7_1	9	0.085897	165	29_1
10	0.166522	182	41_1	11	0.694260	045	37_0 42_0
12	0.162969	036	22_2 41_0	13	0.253838	OUTPUT	15_1 35_1 30_0 27_1
14	0.256266	SG-PNT	14_1 14_2	15	0.039802	DIV	46_0 17_0
16	0.697369	OUTPUT	17_2 42_1 14_0	17	0.640655	ADD	18_0 19_0 17_1
18	0.685729	SG-MAX	16_0 13_1 13_0	19	0.357697	ADD	27_0 6_1 19_1
20	0.000000	MULT	20_2 20_0	21	0.681141	OUTPUT	43_0 23_1 22_1
22	0.667553	SG-PNT	26_0 11_1 39_0	23	0.039100	MULT	29_3
24	0.685848	SUB	21_2 13_2 17_3 16_1	25	0.000000	IFTE	25_0
26	0.250230	MULT	0_0 13_3 25_2 25_1	27	0.238077	IFTE	7_0 8_3
28	0.425428	SPLIT	28_0 28_1 33_0 34_0	29	0.091500	SG-MAX	4_1 32_1
30	0.735100	SPLIT	16_3	31	0.433461	SUB	8_0 35_0 8_2 30_1 18_2
32	0.120807	ADD	29_2	33	0.000000	SUB	9_0
34	0.433530	SG-PNT	11_0 8_1 31_0	35	0.004648	ADD	10_0
36	0.263743	MULT	38_0	37	0.685487	ADD	16_2 11_2 36_0
38	0.282299	SG-MAX	38_1	39	0.669771	MULT	21_0 23_0 41_2
40	0.000000	IFTE	40_2	41	0.655078	ADD	22_0 44_0 40_1
42	0.699286	OUTPUT	21_1	43	0.683490	SUB	24_0
44	0.643737	ADD	10_1 18_1 40_0	45	0.671457	ADD	21_3 31_1
46	0.000158	ADD	23_2				

Appendix F

PADO History

This appendix includes only the briefest view of the two main previous representations used with PADO. For more information on either of these representations, the best source are the cited papers.

F.1 Historical Representations

The first PADO representation was a tree of nodes (an "S-expression") [Koza, 1992] that was *repeatedly evaluated.* Because it had this loop (repeated evaluation) and memory (READ and WRITE primitives) this representation was Turing complete. Figure F.1 pictures this first PADO representation. This algorithm representation (first proposed in [Teller, 1994b]) was chosen for its readability not for its ease of evolution. In other words, this representation was motivated by its nearness to the existing GP representation, not for any representation features which might lend themselves to faster, more effective, or more scrutable evolution. That made it an appropriate starting point for PADO. However, despite some success with the initial PADO system using this representation [Teller and Veloso, 1995c], a representation improvement was found to address other research goals for PADO, such as intelligent recombination and principled evolution. [Teller and Veloso, 1995c] contains empirical results for this first PADO representation on a number of different signal domains, including two visual domains for which empirical results are presented in this thesis.

[Teller and Veloso, 1995d] proposed a new representation for GP programs: an arbitrary graph structure. Each node, like traditional GP, corresponded to a primitive (now called an *action* since the old distinction of *terminals* and *non-terminals* arose from their relative positions in the program tree). Evaluation starts at a particular start node. At each new node, that node's action is performed and *one* of its directed outgoing arcs is chosen to represent the instantaneous change of program control to a new node. The decision about which of the arcs to choose is based on the state of the program at that moment. This process continues until a particular stop node is reached at which time the program's answer is taken either from the top of its argument stack or from some particular memory location. Figure F.2 shows a PADO program with this representation.

This representation has advantages over the traditional tree GP representation and empirically performs better. Works such as [Teller and Veloso, 1995b, Teller and Veloso, 1997,

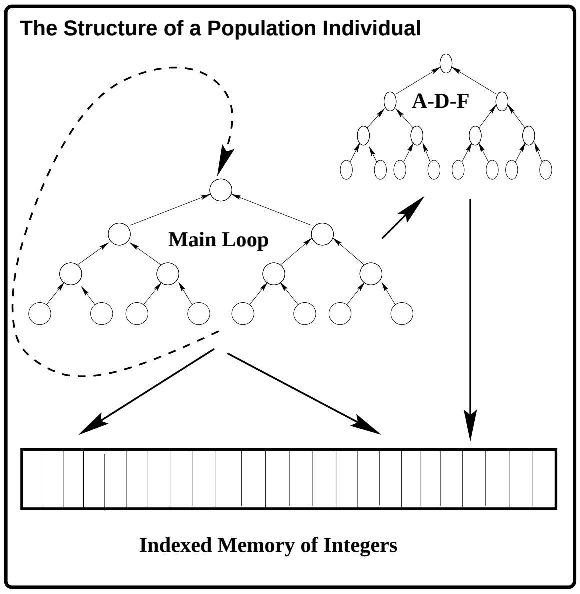


Figure F.1: The original basic structure of a PADO program.

Teller and Veloso, 1995d] present a range of positive empirical results for this second PADO algorithm representation on a number of domains including the two visual and one acoustic signal domains for which empirical results are given in this thesis. One of the long-term goals of the PADO work has been to find ways to improve the efficiency and effectiveness of the genetic operators by changing the representation of the evolving programs if necessary. This new representation lent itself to the learning of improved recombination (SMART) operators through co-evolution [Teller, 1996] (see section F.3) and thereby made progress toward a major goal. However, a typical "SMART" operator's process of finding an answer was quite inscrutable, because of the "flow of control" aspect of the representation. This caused a problem for humans wishing to inspect a learned program. But it was also a barrier to further improvement of the genetic operators which to be more effective must, we argue, necessarily "examine" the evolving programs in detail.

F.2 Evolution with Explicit Substructure

In the GP field, there has been considerable effort invested in researching how substructure (e.g., referenceable subroutines) can be evolved. Substructure is, of course, a means not an end. Modularity and hierarchy are the ends that substructure facilitates. The argument is that substructure makes the evolved code smaller, easier to read, and provides a regularity to programs that is advantageous when there is exploitable regularity in the problem to be solved. There is ample empirical evidence (see [Koza, 1994]) that this is indeed the case.

F.2.1 ADFs in PADO

In GP, an ADF (*automatically defined function*) refers to a piece of explicit substructure in the evolutionary process. For example, each program could have one ADF associated with

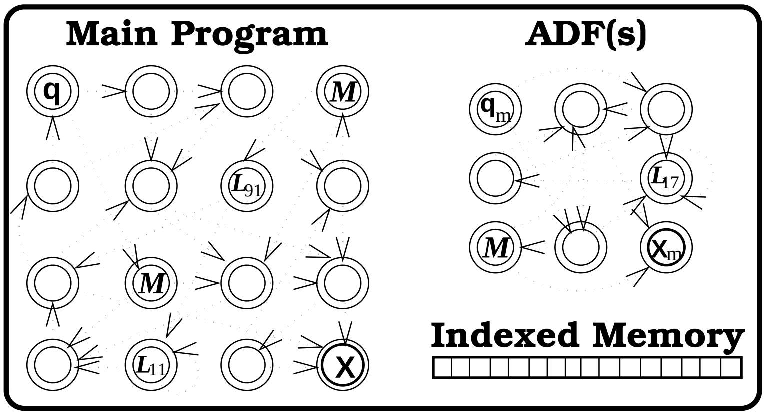


Figure F.2: The basic structure of a PADO program under the phase II representation. There can be one or more ADF programs for each PADO program. Each ADF program may be referenced from the Main program, another local ADF program, or a *Library* program.

it and that ADF could take, for example, two arguments. During evolution, each program may not only refer to its ADF as though that ADF were simply another two-arity function (e.g., ADD or MULT), but each program's ADF will also be subjected to genetic operators so that, over time, the ADF and the program may co-evolve to take advantage of one another in order to better solve the task at hand.

In the phase I and phase II representations of PADO, ADFs were used with the following distinction. Traditionally, GP ADFs may not call themselves since that is a form of recursion and so introduces one of the main evolutionary complications associated with Turing complete languages. However, since PADO was already explicitly dealing with these issues, allowing the ADFs for each program to contain self-references was no additional burden. In all other ways, the PADO ADFs were conceptually the same as traditional GP ADFs.

F.2.2 Libraries in PADO

The *Library* programs (e.g., L_{17} in Figure F.2) are globally available programs that can be executed at any time and from anywhere just like the private ADF programs [Teller and Veloso, 1997]. But unlike the ADF programs, where each ADF may be run only during the execution of the PADO program of which it is a part, the *Library* programs are publicly available to the entire population. [Angeline and Pollack, 1993] describes a similar concept called *Modules*. The distinction is primarily in how these public subroutines are created. Another related concept for identifying publicly useful subroutines can be found in [Rosca and Ballard, 1996].

Initially all Library programs are initialized to be random legal programs with the same characteristics as ADFs. At the end of every generation, the k worst Library programs are removed from the Library and replaced with the ADFs of the k most successful programs of the generation. The "goodness" of a Library program is the sum of the adjusted fitnesses of all programs that called it, multiplied by how often they called it. The adjusted fitness of each program is Rank[p] − (MaxRank − MinRank)/2. Notice that, unlike the MAIN and ADF programs, the individual Library programs do not evolve. Rather they are a storage place for some of the best "ideas" in the population and the bad ideas are moved out in favor of other ideas that have a better chance of being good. In this sense, the Library population *evolves* (in fact co-evolves with the main population and the SMART operator population) even though the individual library programs receive neither fitness proportionate reproduction or genetic recombination. Further details on this subject can be found in [Teller and Veloso, 1997].

F.3 Learned Algorithm Recombination Algorithms

One of the current debates in the field of GP is how to evolve structures that are more complicated than simple functional trees. One argument (typified by the cellular encoding work of [Gruau, 1994b]) argues that GP should be left as is, but that the evolved GP function tree should express *how* to build a more complicated structure (like a feed forward NN or an arbitrary graph). Another camp in this discussion (of which this work is a part) argues that these more complicated structures should be evolved directly, rather than indirectly. [Teller, 1996] and [Sims, 1994] are examples of such a view. In [Teller, 1996] we address the added complications that come with manipulation of more complex structures.

The basic function of the genetic recombination operator "crossover" is to take k (usually two) programs as input and to produce some (usually k) new programs as output that replace the input programs in the population from which they were drawn. The context of the PADO representation leads to the need for recombination (crossover) of two arbitrary graphs. Partly because the "right" way to do graph recombination is not obvious, learning how to do graph recombination intelligently is a natural desire.

The SMART operators are *programs* that learn to do this graph crossover better than recombination operators acting at random. These SMART recombination operators co-evolve with the main population. Like the programs in the main population, the programs in the SMART operator population may begin as randomly generated programs. The programs in this SMART operator population are tested by allowing them to *actually perform the recombinations on the main population*. Their fitness values are a function of the relative fitness of the programs they take as input and the fitness of programs they produce as output. [Teller, 1996] has a large amount of detail on this subject.

While [Teller, 1996] contains no proof that the hurdles of evolving a more complex representation have been cleared, it is a direction of research which may turn out to be as viable and considerably more flexible than the "growth phase" strategy. In addition, while we have only reported positive results using SMART operators for the arbitrary graph structured language, there is good reason to believe that this technique can be used effectively by itself or in conjunction with other attempts to make the search in EC more intelligent in other representations.

Appendix G

NP vs. CellularEncoding

Rather than directly evolving programs (even Turing complete programs) as neural programming does, it is possible to dissociate the genotype from the phenotype and so to evolve complex programs indirectly. Cellular encoding [Gruau, 1994a] is the most important example of this technique and it is worth highlighting the differences in perspective of neural programming and cellular encoding.

G.1 Overview of Cellular Encoding

There is a range of past and current research efforts that use GAs to directly evolve aspects of fixed topology neural networks. (e.g., [Porto *et al.*, 1995, Baluja, 1995, Chambers, 1995]). In contrast, *cellular encoding* is a technique for evolving a description of how to **build** an arbitrary ANN. "Building an arbitrary ANN" means the evolution of the ANN-topology, thresholds, and biases as well as the weights themselves. In cellular encoding, each population individual is a standard GP tree. The terminals and non-terminals are network-constructing, neuron-creating, and neuron-adjusting nodes.

The cellular encoding tree is **not** the ANN. It is the genotype. The ANN constructed by the application of a cellular encoded tree is the phenotype. The fitness of the genotype (the cellular encoding tree) is measured through the performance of the phenotype (the ANN) on the desired task.

The advantage of cellular encoding is that all of the standard GP techniques can be applied without modification to the evolution of a substantially different computation device (an ANN in this case). This is because cellular encoding is an *application* of tree-GP, not a new representation itself.

Reusable neural sub-networks (like ADFs) can be implemented in cellular encoding. Recursion can be implemented to create neural networks for high-order symmetry and high-order parity functions. Cellular encoding has been applied to the 2-pole-balancing problem [Whitley *et al.*, 1995], to six-legged creature walking behavior [Gruau and Quatramaran, 1996], and to Finite Automata development [Brave, 1996b].

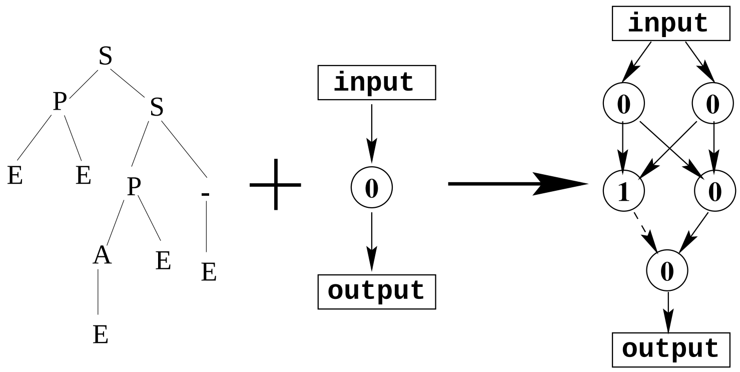


Figure G.1: Cellular encoding tree + embryonic neuron = NN

G.2 Thesis Corroboration from Cellular Encoding

As Section G.1 makes clear, cellular encoding is a method for indirectly learning models that cannot easily be learned in the traditional format of genetic programming. This is true and cellular encoding is certainly a valuable area of future work in genetic programming.

However, the fact that researchers have gone to such lengths to force the evolution of complex, non-tree based models into the tree-based format highlights the fact that genetic programming in its standard form is brittle and cannot directly evolve many of the kinds of programs of interest to researchers, *which is the basic thesis of this research.* This is not to say that cellular encoding does not work, but rather that the cellular encoding work corroborates the claim of this thesis that GP must be modified or augmented in order to get important new aspects into the paradigm.

Whether it is better to change the genetic programming paradigm to incorporate gradient descent style recombination operators like those presented in this thesis, or whether it is better to leave GP as it is and work on this "developmental process," as typified in cellular encoding is still an open research question.

Appendix H

Statistical Significance Information

Each of the following graphs shows the standard deviation **error** for each curve presented in the referenced figure in the main body of the thesis. Standard error is standard deviation **divided by** the square root of the number of independent samples for that point. The simplest way to read these curves is “There is only a 30% chance that the ’true’ curve is outside of the upper and lower STD curves shown in each figure.”

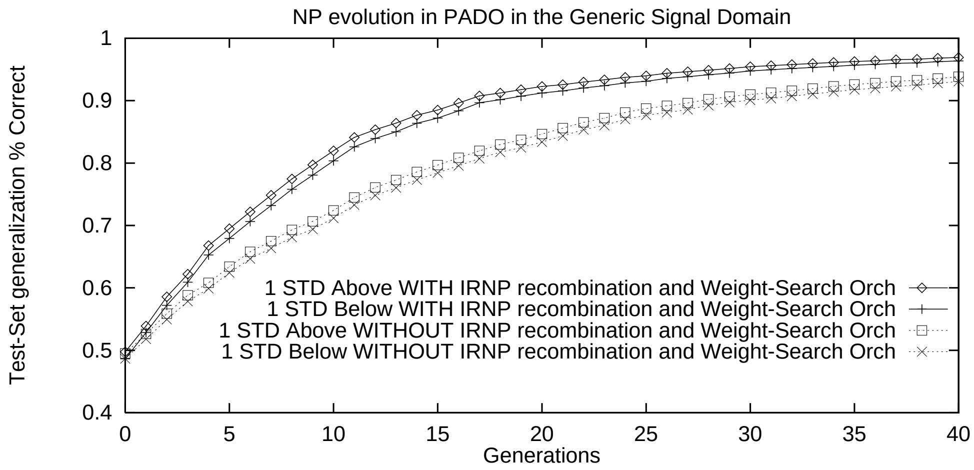


Figure H.1: The **standard error** curves for Figure 7.2.

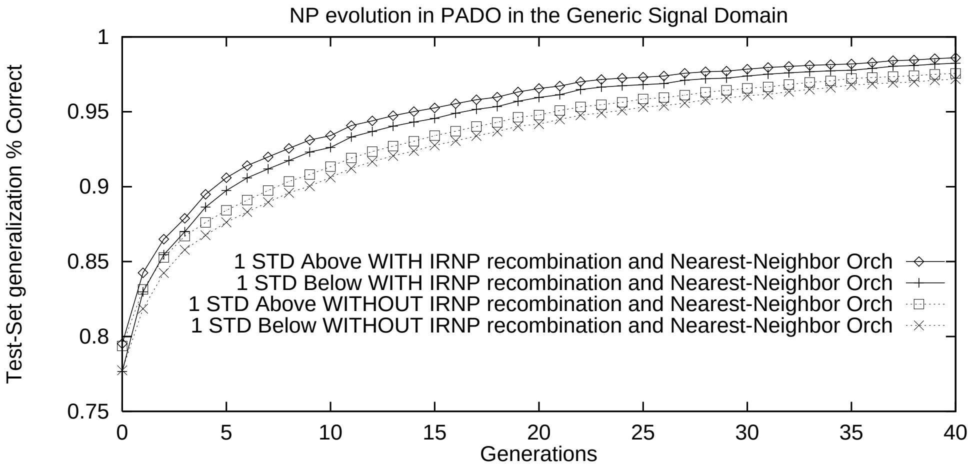


Figure H.2: The **standard error** curves for Figure 7.3.

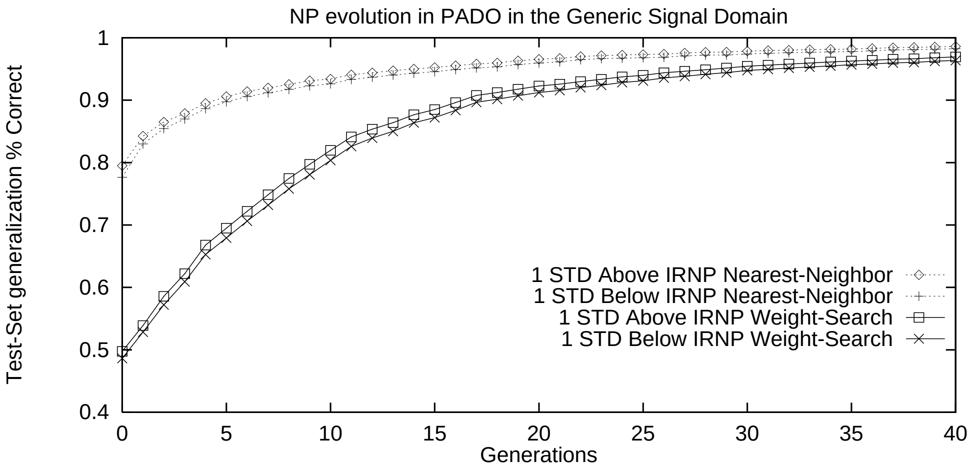


Figure H.3: The **standard error** curves for Figure 7.4.

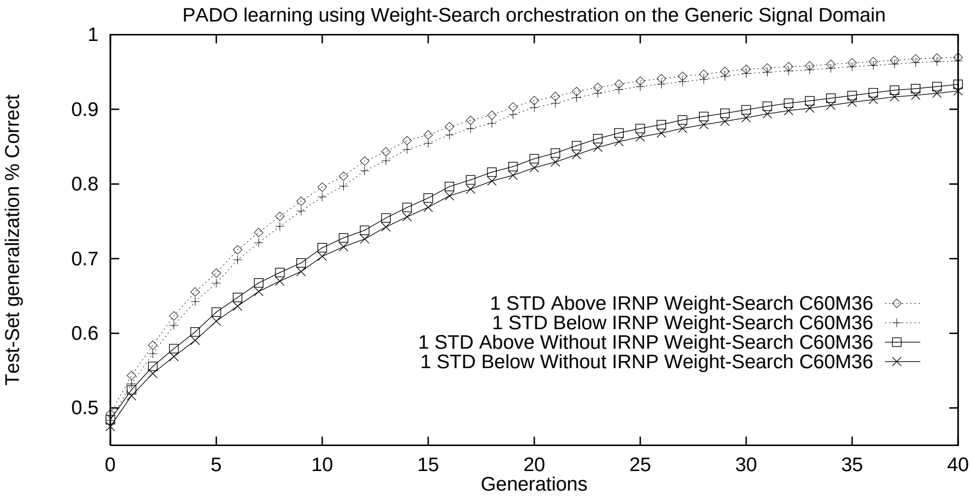


Figure H.4: The **standard error** curves for Figure 7.5.

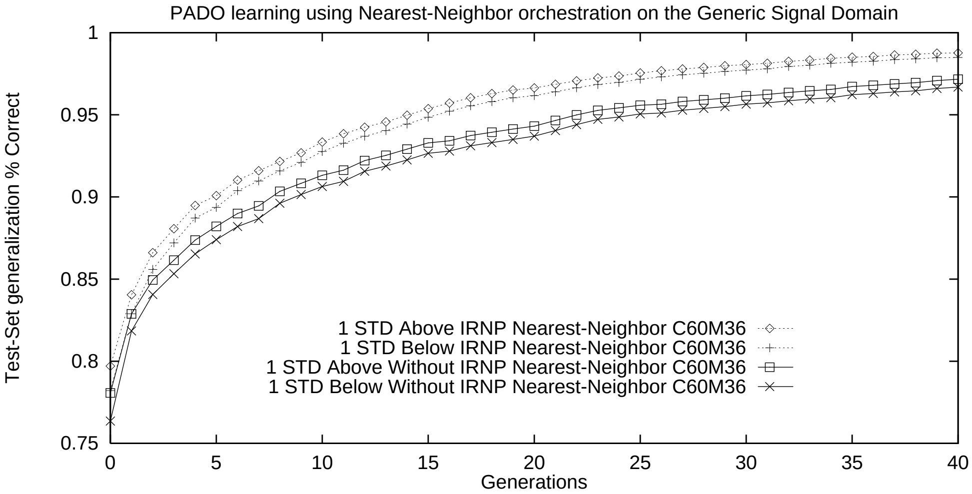


Figure H.5: The **standard error** curves for Figure 7.6.

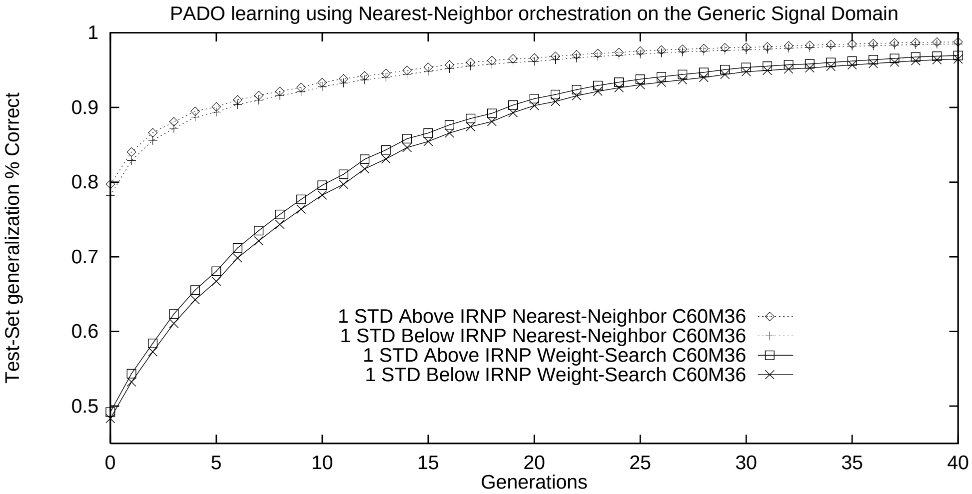


Figure H.6: The **standard error** curves for Figure 7.7.

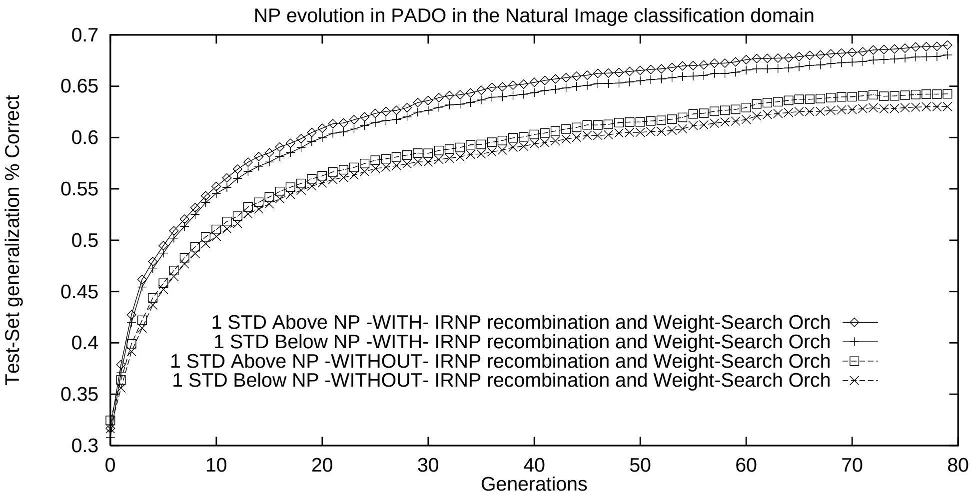


Figure H.7: The **standard error** curves for Figure 7.9.

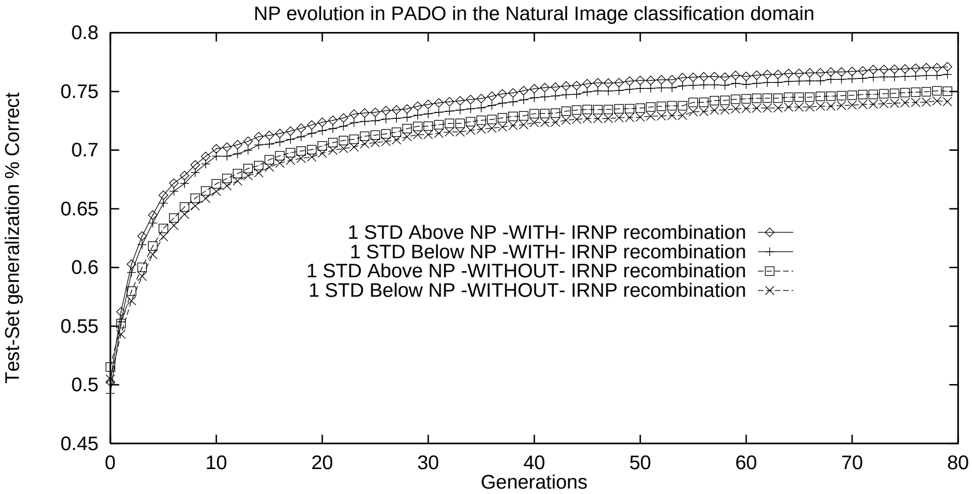


Figure H.8: The **standard error** curves for Figure 7.10.

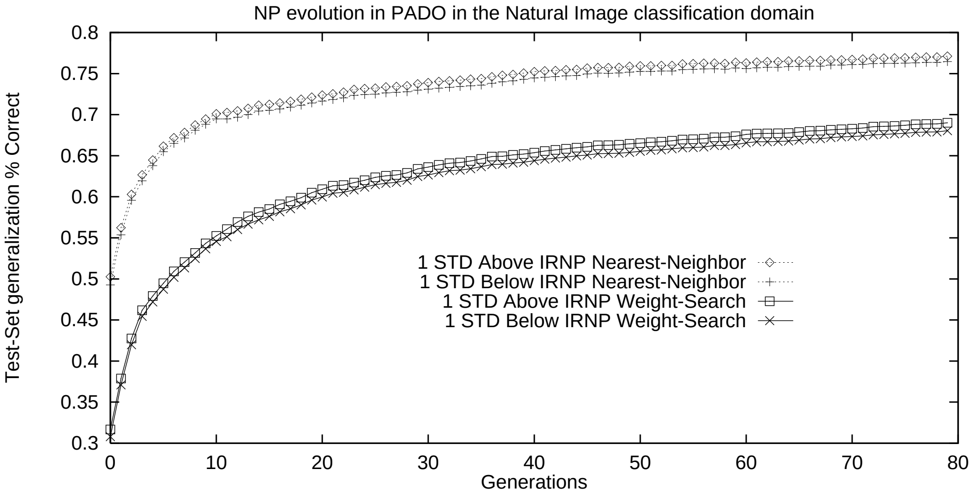


Figure H.9: The **standard error** curves for Figure 7.11.

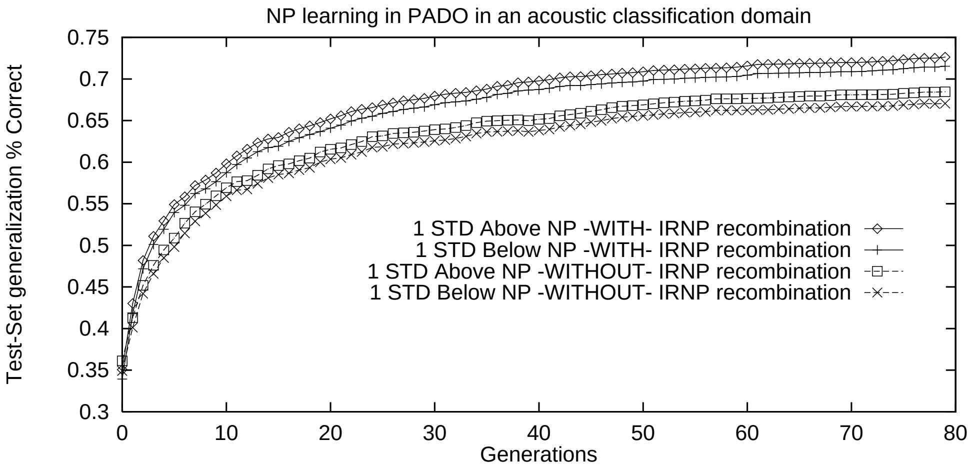


Figure H.10: The **standard error** curves for Figure 7.12.

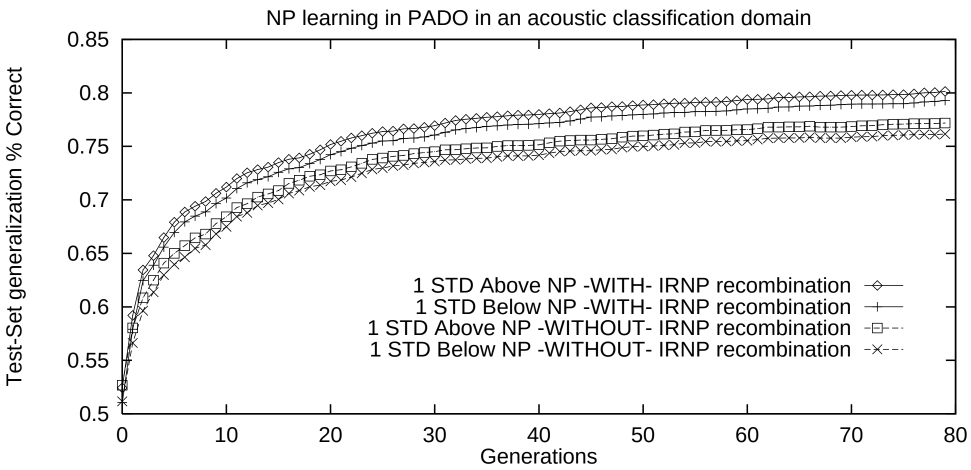


Figure H.11: The **standard error** curves for Figure 7.13.

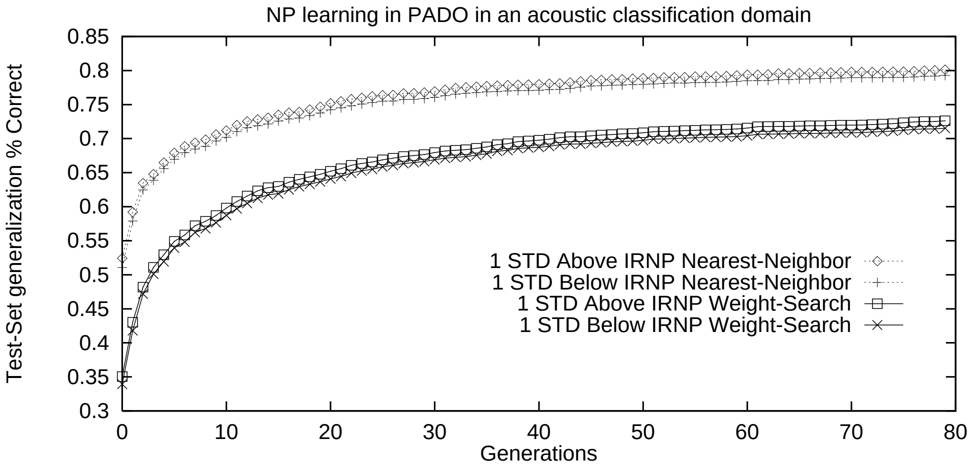


Figure H.12: The **standard error** curves for Figure 7.14.

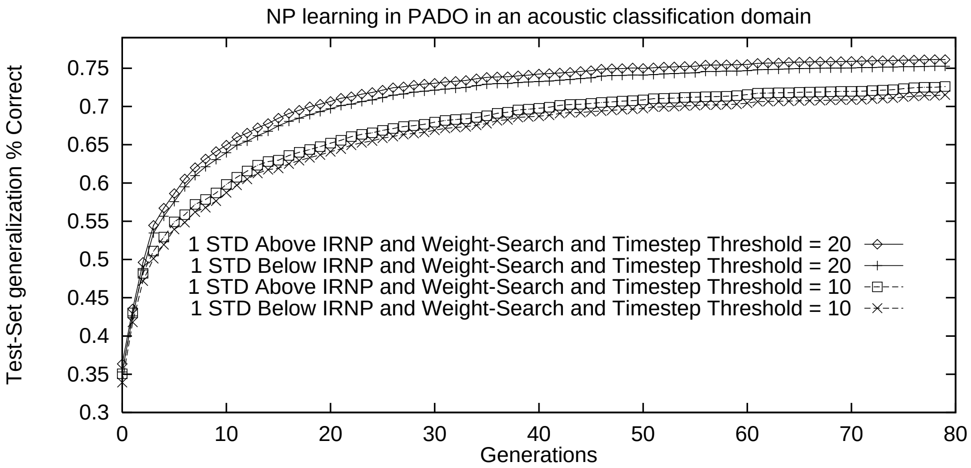


Figure H.13: The **standard error** curves for Figure 7.17.

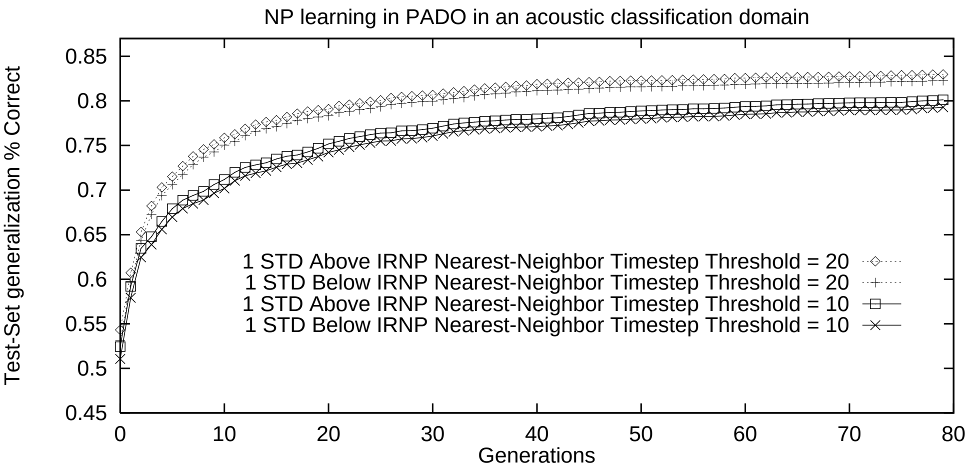


Figure H.14: The **standard error** curves for Figure 7.18.

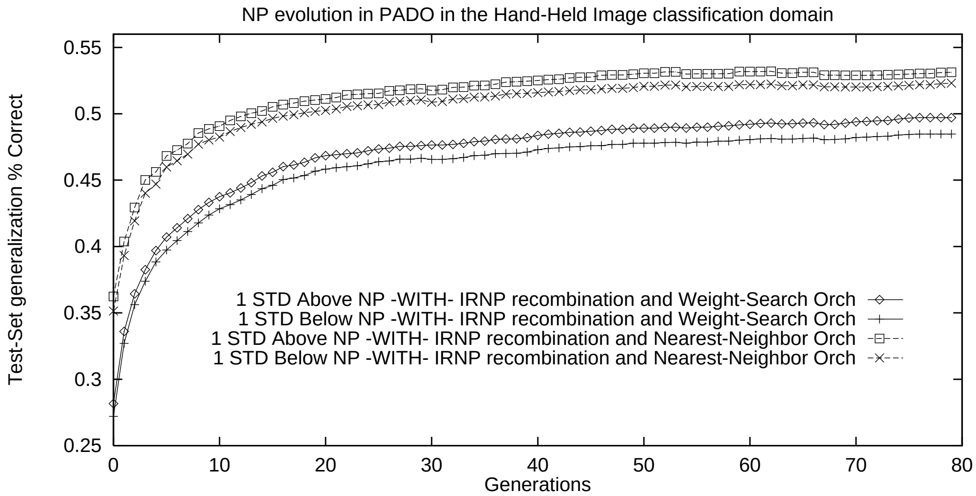


Figure H.15: The **standard error** curves for Figure 7.21.

Bibliography

[Altenberg, 1994] L. Altenberg. The evolution of evolvability in genetic programming. In Jr. K. Kinnear, editor, *Advances In Genetic Programming*, pages 47–74. MIT Press, 1994.

[Andre and Teller, 1996] D. Andre and A. Teller. A study in program response and the negative effects of introns in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 12, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Andre and Teller, 1998] David Andre and Astro Teller. Evolving team darwin united. In Minoru Asada, editor, *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, 1998.

[Andre *et al.*, 1996] David Andre, Forrest H. Bennett III, and John R. Koza. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 3–11, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Andre, 1994] David Andre. Automatically defined features: The simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them. In K. Kinnear, editor, *Advances In Genetic Programming*, pages 477–494. MIT Press, 1994.

[Andre, 1995] David Andre. The evolution of agents that build mental models and create simple plans using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 248–255, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Angeline and Kinnear, Jr., 1996] Peter J. Angeline and K. E. Kinnear, Jr., editors. *Advances in Genetic Programming 2*. MIT Press, Cambridge, MA, USA, 1996.

[Angeline and Pollack, 1993] Peter Angeline and J. Pollack. Evolutionary module acquisition. In D. Fogel, editor, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163. Evolutionary Programming Society, 1993.

[Angeline, 1993] P. Angeline. *Evolutionary Algorithms and Emergent Intelligence*. PhD thesis, Ohio State University, Computer Science Department, 1993.

[Angeline, 1996] P. Angeline. Two self-adaptive crossover operators for genetic programming. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2*. MIT Press, 1996.

[Angeline, 1997a] Peter J. Angeline. An alternative to indexed memory for evolving programs with explicit state representations. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 423–430, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Angeline, 1997b] Peter J. Angeline. Subtree crossover: Building block engine or macromutation? In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Bairoch and Boeckmann, 1991] A. Bairoch and B. Boeckmann. The swiss-prot protein sequence data bank: current status. *Nucleic Acids Research*, 22(17):3578–3580, 1991.

[Baluja and Caruana, 1995] Shumeet Baluja and Rich Caruana. Removing the genetics from the standard genetic algorithm. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 38–46. Morgan Kaufmann, 1995.

[Baluja, 1995] Shumeet Baluja. Artificial neural network evolution: Learning to steer a land vehicle. In Chambers [1995], chapter 1.

[Baluja, 1996] Shumeet Baluja. Genetic algorithms and explicit search statistics. In *Proceedings of the 1996 Neural Information Processing Society*. MIT Press, 1996.

[Banzhaf *et al.*, 1998] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, November 1998.

[Baxter and Bartlett, 1998] J. Baxter and P. L. Bartlett. The canonical distortion measure in feature space and 1-nn classification. In *Neural Information Processing Systems*, Cambridge, MA, 1998. MIT Press.

[Beck and Fischer, 1995] T. Beck and Herbert Fischer. The IF-problem in automatic differentiation. *Journal of Comput. Appl. Math.*, 50:119–131, 1995.

[Bielak, 1993a] Dennis C. Bielak. *Improving Classification with adaptive sysnthesis based on collective learning*. PhD thesis, George Washington University, 1993.

[Bielak, 1993b] Dennis Chester Bielak. *Improving Classification with adaptive synthesis based on collective learning*. PhD thesis, George Washington University, 1993.

[Bischof *et al.*, 1992] Christian H. Bischof, Alan Carle, George F. Corliss, and Andreas Griewank. ADIFOR: Automatic differentiation in a source translation environment. In Paul S. Wang, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 294–302, New York, 1992. ACM Press.

[Blum, 1995] A. Blum. Empirical support for winnow and weighted-majority based algorithms: results on a calendar scheduling domain. In *Proceedings of the Twelfth International Conference on Machine Learning*. Morgan Kaufmann, 1995.

[Brave, 1996a] Scott Brave. The evolution of memory and mental models using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 261–266, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Brave, 1996b] Scott Brave. Evolving deterministic finite automata using cellular encoding. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 39–44, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Brave, 1996c] Scott Brave. Using genetic programming to evolve recursive programs for tree search. In P. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10. MIT Press, Cambridge, MA, USA, 1996.

[Brooks, 1986] R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[Brooks, 1990] R.A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15, 1990.

[Brooks, 1997] R.A. Brooks. From earwigs to humans. *Robotics and Autonomous Systems*, 20(2–4):291–304, 1997.

[Cedano *et al.*, 1997] Juan. Cedano, Patrick Aloy, Josep Perez-Pons, and Enrique Querol. Relation between amino acid composition and cellular location of proteins. *Journal of Molecular Biology*, 266(3):594–600, 1997.

[Chambers, 1995] Lance Chambers, editor. *Practical handbook of genetic algorithms: new frontiers*, volume II. CRC Press, Inc., 1995.

[Chavent *et al.*, 1996] Guy Chavent, Jérôme Jaffré, Sophie Jégou, and Jun Liu. A symbolic code generator for parameter estimation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 129–136. SIAM, Philadelphia, Penn., 1996.

[Chellapilla, 1997] Kumar Chellapilla. Evolutionary programming with tree mutations: Evolving computer programs without crossover. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 431–438, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Collins, 1992] Robert Collins. *Studies in Artifical Evolution.* PhD thesis, University of California in LA, Department of Computer Science, 1992.

[Conrads *et al.*, 1998] M. Conrads, P. Nordin, and W. Banzhaf. Speech sound discrimination with genetic programming. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, LNCS, Paris, 14-15 April 1998. Springer-Verlag. Forthcoming.

[Cramer, 1985] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985.

[Cribs and Smith, 1996] H. Brown Cribs and Robert E. Smith. Classifier systems renaissance: New analogies, new directions. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 547–552, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Daida, 1996] Jason Daida. Algorithm discovery using the genetic programming paradigm: Extracting low-contrast curvilinear features from SAR images of arctic ice. In P. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 21. MIT Press, Cambridge, MA, USA, 1996.

[Dellaert and Beer., 1994] F. Dellaert and R.D. Beer. 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, 1994.

[Dennett, 1992] Daniel Dennett. *Consciousness Explained.* Little Brown and Co., 1992.

[Fogel *et al.*, 1966] L. Fogel, A. Owens, and M. Walsh. *Artifical Intelligence through Simulated Evolution.* New York: Wiley, 1966.

[Fogel *et al.*, 1995] L. Fogel, P. Angeline, and D. Fogel. An evolutionary programming approach to self-adaptation on finite state machines. In J. McDonnell, R. Reynolds, and D. Fogel, editors, *Proceedings of the 4th Annual Conference on Evolutionary Programming.* MIT Press, 1995.

[Gathercole and Ross, 1996] Chris Gathercole and Peter Ross. An adverse interaction between crossover and restricted tree depth in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 291–296, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Goldberg *et al.*, 1989] D. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. In *Complex Systems*, volume 3(5): 493-530. 1989.

[Goldberg, 1989] David Goldberg. *Genetic Algorithms: In search, optimization, and machine learning*. Addison-Wesley Press, 1989.

[Griewank and Corliss, 1991] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, Penn., 1991.

[Gruau and Quatramaran, 1996] Frederic Gruau and Kameel Quatramaran. Cellular encoding for interactive evolutionary robotics. Cognitive Science Research Paper 425, School of Cognitive and Computing Sciences, University of Sussex, Falmer, Brighton, Sussex, UK, 1996.

[Gruau, 1994a] F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France, 1994.

[Gruau, 1994b] Frederic Gruau. Genetic micro programming of neural networks. In Jr. Kenneth E. Kinnear, editor, *Advances In Genetic Programming*, pages 495–518. MIT Press, 1994.

[Hancock, 1990] P. J. B. Hancock. GANNET: Design of a neural network for face recognition by genetic algorithm. In *Proceedings of the IEEE Workshop on Genetic Algorithms, Simulated Annealing and Neural Networks*, University of Glasgow, Scotland, 1990.

[Haynes and Sen, 1996] Thomas Haynes and Sandip Sen. Evolving behavioral strategies in predators and prey. In Gerhard Weißand Sandip Sen, editors, *WEISS96*, pages 113–126. Springer Verlag, Berlin, 1996.

[Haynes *et al.*, 1995] Thomas Haynes, Roger Wainwright, Sandip Sen, and Dale Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies. In Stephanie Forrest, editor, *ICGA95*, pages 271–278, San Mateo, CA, July 1995. Morgan Kaufman.

[Hild and Waibel, 1993] H. Hild and A. Waibel. Multi-speaker/speaker-independent architectures for the multi-state time delay neural network. In *Proceedings of the Intern. Conference on Acoustics, Speech and Signal Processing*. 1993.

[Holland and Reitman, 1978] J.H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In *Pattern Directed Inference Systems*. Academic Press, 1978.

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

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

[Ikeuchi and Veloso, 1997] Katsu Ikeuchi and Manuela Veloso, editors. *Symbolic Visual Learning*. Oxford University Press, 1997.

[Intrator *et al.*, 1995] N. Intrator, D. Reisfeld, and Y. Yeshurun. Face recognition using a hybrid supervised/unsupervised neural network. In *Proceedings of the Face and Object Recognition Conference*, 1995.

[Kinnear, Jr., 1993] Kenneth E. Kinnear, Jr. Generality and difficulty in genetic programming: Evolving a sort. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 287–294, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[Kinnear, Jr., 1994] Kenneth E. Kinnear, Jr., editor. *Advances in Genetic Programming.* MIT Press, Cambridge, MA, 1994.

[Kitano *et al.*, 1997] Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The RoboCup synthetic agent challenge 97. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 24–29, San Francisco, CA, 1997. Morgan Kaufmann.

[Koza *et al.*, 1996] John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane. Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 123–131, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Koza *et al.*, 1998] John R. Koza, Forrest Bennett, and David Andre. Classifying proteins as extracellular using programmatic motifs and genetic programming. In *Proceedings of the IEEE International Conference on Evolutionary Computation.* IEEE Press, 1998.

[Koza, 1992] J. Koza. *Genetic Programming.* MIT Press, 1992.

[Koza, 1994] J. Koza. *Genetic Programming 2.* MIT Press, 1994.

[Langdon, 1995] William Langdon. Evolving data structures with genetic programming. In Stephanie Forrest, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms.* Morgan Kauffman, 1995.

[Langdon, 1996] William Langdon. Data structures and genetic programming. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2.* MIT Press, 1996.

[Littlestone and Warmuth, 1994] N. Littlestone and M. K. Warmuth. The weighted-majority algorithm. 108(2):212–261, 1994.

[Littlestone, 1988] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. In *Machine Learning*, pages 2:285–318. 1988.

[Luke and Spector, 1996] Sean Luke and Lee Spector. Evolving teamwork and coordination with genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 150–156, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Maxwell III, 1994] Sidney R. Maxwell III. Experiments with a coroutine model for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA*, volume 1, pages 413–417a, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Mitchell, 1997] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[Nguyen and Huang, 1994] T. Nguyen and T. Huang. Evolvable 3d modeling for model-based object recognition systems. In K. Kinnear, editor, *Advances In Genetic Programming*. MIT Press, 1994.

[Nordin and Banzhaf, 1996] Peter Nordin and Wolfgang Banzhaf. Programmatic compression of images and sound. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 345–350, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Nordin, 1997a] Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.

[Nordin, 1997b] Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, Krehl Verlag, Krehl Verlag, Postfach 51 01 42, D-48163 Muenster, GERMANY, 1997.

[Oakley, 1994] E. H. N. Oakley. The application of genetic programming to the investigation of short, noisy, chatoic data series. In T. C. Fogarty, editor, *Lecture Notes in Computer Science: Evolutionary Computation*. Springer-Verlag, 1994.

[Olsson, 1995] Roland Olsson. *Inductive Functional Programming using Incremental Program Transformation*. PhD thesis, University of Oslo, 1995.

[O'Reilly, 1995] Una-May O'Reilly. *An Analysis of Genetic Programming*. PhD thesis, Carelton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada, 22 September 1995.

[Poli, 1996a] Riccardo Poli. Genetic programming for image analysis. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 363–368, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Poli, 1996b] Riccarod Poli. 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, 1996.

[Pomerleau, 1992] Dean Pomerleau. *Neural Network Perception for Mobile Robot Guidance.* PhD thesis, Carnegie Mellon University School of Computer Science, 1992.

[Porto *et al.*, 1995] Vincent W. Porto, David B. Fogel, and Lawrence J. Fogel. Alternative neural network training methods. *IEEE Expert*, 10(3):16–22, 1995.

[Press *et al.*, 1992] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipies in C: the art of scientific computing.* Cambridge University Press, Cambridge, 2nd edition, 1992.

[Quinlan, 1986] J. R. Quinlan. Induction of decision trees. In *Machine Learning*, pages 81–106. Kluwer Academic Publishers, Boston, MA, USA, 1986.

[Rechenberg, 1965] I. Rechenberg. Cybernetic solution path of an experimental problem. *Ministry of Aviation, Royal Aircraft Establishment (U.K.)*, 1965.

[Rice, 1987] John A. Rice. *Mathematical statistics and data analysis.* The Wadsworth and Brooks-Cole Statistics-Probability Series. Wadsworth and Brooks-Cole Advanced Books and Software, Pacific Grove, California, 1987.

[Rosca and Ballard, 1996] Justinian P. Rosca and Dana H. Ballard. Discovery of subroutines in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA, 1996.

[Rowley *et al.*, 1995] H. Rowley, S. Baluja, and T. Kanade. Human face detection in visual scenes. In *Neural Information Processing Systems*, number 8. MIT Press, 1995.

[Rumelhart *et al.*, 1986] D.E. Rumelhart, G.E. Hinton, and R.J Williams. Learning internal represenations by error propogation. In *Parallel Distributed Processing.* MIT Press, Cambridge, MA, USA, 1986.

[Sebald *et al.*, 1991] A. Sebald, J. Schlenzig, and D. Fogel. Minimax design of cmac encoded neural controllers for systems with variable time delay. In R. R. Chen, editor, *Proceedings of the 25th Asilomar Conference on Signals, Systems, and Computers*, pages 551–555. Maple Press, 1991.

[Selfridge, 1966] Oliver Selfridge. Pandemonium: A paradigm for learning. In L. Uhr, editor, *Pattern Recognition.* Wiley, 1966.

[Sharman *et al.*, 1995] K. Sharman, A. Alcazar, and Y. Li. Evolving signal processing algorithms by genetic programming. In *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, 1995.

[Sims, 1994] Karl Sims. Evolving virtual creatures. In *Proceedings of the 21st International SIGGRAPH Conference.* ACM Press, 1994.

[Squires and Sammut, 1995] B. Squires and C. Sammut. Automatic speaker recognition: An application of machine learning. In *Proceedings of the Twelfth International Conference on Machine Learning*. 1995.

[Stryer, 1995] Lubert Stryer. *Biochemistry*. W.H. Freeman, 1995.

[Sung and Poggio., 1994] K. Sung and T. Poggio. Example-based learning for view-based human face detection. In *Technical Report A.I. Memo 1521, CBCL*. MIT, 1994.

[Sutton, 1988] R.S. Sutton. Learning to predict by the methods of temporal differences. In *Proceedings of the International Conference on Machine Learning*. AAAI Press, 1988.

[Tackett, 1993] Walter A. Tackett. Genetic programming for feature discovery and image discrimination. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kauffman, 1993.

[Tackett, 1994] Walter A. Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, 1994. Available as: Technical Report CENG 94-13. Dept. of Electrical Engineering Systems.

[Tebelski, 1995] Joe Tebelski. *Speech Recognition using Neural Networks*. PhD thesis, Carnegie Mellon University School of Computer Science, 1995.

[Teller and Veloso, 1995a] A. Teller and M. Veloso. Algorithm evolution for face recognition: What makes a picture difficult. In *Proceedings of the International Conference on Evolutionary Computation*. IEEE Press, 1995.

[Teller and Veloso, 1995b] A. Teller and M. Veloso. Language representation progression in PADO. In *AAAI Fall Symposium Series. AAAI Technical Report*. AAAI Press, 1995.

[Teller and Veloso, 1995c] A. Teller and M. Veloso. PADO: Learning tree structured algorithms for orchestration into an object recognition system. Technical Report CMU-CS-95-101, Computer Science Department, Carnegie Mellon University, 1995.

[Teller and Veloso, 1995d] A. Teller and M. Veloso. Program evolution for data mining. In Sushil Louis, editor, *The International Journal of Expert Systems. Third Quarter. Special Issue on Genetic Algorithms and Knowledge Bases.*, pages 216–236. JAI Press, 1995.

[Teller and Veloso, 1996] Astro Teller and Manuela Veloso. Neural programming and an internal reinforcement policy. In *First International Conference on Simulated Evolution and Learning*, pages 279–86. Springer-Verlag, 1996.

[Teller and Veloso, 1997] A. Teller and M. Veloso. PADO: A new learning architecture for object recognition. In K. Ikeuchi and M. Veloso, editors, *Symbolic Visual Learning*. Oxford University Press, 1997.

[Teller, 1994a] A. Teller. The evolution of mental models. In Kenneth E. Kinnear, editor, *Advances In Genetic Programming*, pages 199–220. MIT Press, 1994.

[Teller, 1994b] A. Teller. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the First IEEE World Congress on Computational Intelligence*, pages 136–146. IEEE Press, 1994.

[Teller, 1996] A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In K. Kinnear and P. Angeline, editors, *Advances in Genetic Programming 2*. MIT, 1996.

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

[Turk and Pentland, 1991] M. Turk and A. Pentland. Eigenfaces for recognition. In *Journal of Cognitive Neuroscience*, 1991.

[Viola, 1993] P. Viola. Feature-based recognition of objects. In *AAAI FSS on Machine Learning in Computer Vision*. AAAI, 1993.

[Waibel *et al.*, 1989] A.H. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang. Phoneme recognition using time-delay neural networks. In *IEEE Transactions on Acoustics, Speech and Signal Processing. 1989*. 1989.

[Watkins, 1989] Christopher J.C.H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, 1989.

[Whitley *et al.*, 1995] Darrell Whitley, Frederic Gruau, and Larry Pyeatt. Cellular encoding applied to neurocontrol. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 460–467, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Wilson and Goldberg, 1989] S. W. Wilson and D. E. Goldberg. A critical review of classifier systems. In *Proceedings of the Third International Conf. On Genetic Alogirithms*. Morgan Kauffman, 1989.

[Wilson, 1987] S.W. Wilson. Hierarchical credit allocation in a classifier system. In *Genetic Algorithms and Simulated Annealing*. Morgan Kaufman Publishers, 1987.

[Wolpert, 1992] D. H. Wolpert. Stacked generalization. In *Neural Networks*, volume 5, pages 241–259. Pergamon Press, 1992.