[illegible] 96-10

[illegible]

Genetic Algorithm Based Learning

Kenneth De Jong
George Mason University

Abstract

This chapter describes a subarea of machine learning which is actively exploring the use of genetic algorithms as the key element in the design of robust learning strategies. After characterizing the kinds of learning problems motivating this approach, a brief overview of genetic algorithms is presented. Three major approaches to using genetic algorithms for machine learning are described and an example of their use in learning entire task programs is given. Finally, an assessment of the strengths and weaknesses of this approach to machine learning is provided.

1. Introduction

The explosive growth of interest in machine learning has lead to a rich diversity of approaches to the design of learning systems. One of the consequences of this diversity is the formation of subgroups which share common interests such similarity-based learning, explanation-based learning, neural net learning, and so on. The advantage of such groups is the ability to focus intensively on a highly specialized set of issues and make considerable progress in understanding machine learning in that context. The primary danger of such groups is that the high degree of specialization can lead easily to a state in which the communication of ideas and results among subgroups is difficult.

One of the goals of this book (and the two preceding ones) is to provide cogent descriptions of specialty areas in machine learning in terms that "outsiders" can understand and assess the strengths and limitations of a particular approach. This chapter attempts to achieve that goal for an active subgroup concerned with machine learning from an *adaptive systems* perspective as initially proposed by Holland (1975), and with a special interest in the use of *genetic algorithms* as a key element in the design of learning strategies. In keeping with present convention, this approach will be referred to as Genetic Algorithm Based Learning (GABL).

2. An Adaptive Systems Perspective

The casual and imprecise usage of the term "learning" in everyday life seems to confound attempts to provide a succinct, all-inclusive definition for it. Faced with this difficulty, the operational workaround of the machine learning research community has been to focus on *useful and interesting aspects* of learning without concern for capturing all facets. From an adaptive systems perspective, that focus is on *systems which are capable of making changes to themselves over time with the goal of improving their performance on the tasks confronting them in a particular environment.*

This view reflects several biases of the adaptive systems community. Note first the *performance-oriented* nature of the definition. There is much less emphasis on evaluating learning in terms of changes to internal structures and much more emphasis on evaluating learning in terms of changes in performance over time. This focus on performance is in some sense a pragmatic reflection of our own difficulty in dealing with complex problems. It is generally beyond our current abilities to characterize the behavior of a system of any complexity by means of a static analysis of its internal structure regardless of whether the system was built by hand or constructed using some automated (learning) techniques. Although we can develop a set of tools which are capable of catching syntactic and simple semantic problems via a static analysis of the system, this is invariably complemented by the development of an extensive "test suite" of problems for empirical validation.

This is closely related to another issue: *when does learning stop*? The answer from an adaptive systems perspective is: *never*! Consider the construction of a self-improving diagnostic expert system from this perspective. If such a system is designed to continue learning "in the field", there is less concern about anticipating (and validating) all possible situations *before* the system is released. A strong motivation for this point of view comes from the observation that the world (environment) in which such systems perform is *itself* very seldom static. Designing systems to adapt to such environmental changes rather than requiring manual intervention would seem preferable.

A second bias reflected in the definition of learning given above is the emphasis on *self-modification*. The mental image of an adaptive system is that of a "black box" whose internal state is not directly accessible to anything in its environment, including teachers. Advice or any other form of feedback is presented in terms of an interface language which must be interpreted and integrated internally by the adaptive system itself.

If one attempts to formalize these ideas, the adaptive system model which generally emerges is an abstraction of an autonomous robot equipped with a fixed set of detectors (sensors) and effectors (operators) which are presumed to be useful primitives for improving performance in the task domain defined by the environment. As has been the case in other machine learning areas, it is useful (as illustrated in Figure 1) to separate out the problem solving component (whose performance is to be improved) from the learning component which is charged with finding ways of improving the performance of the problem solver. Notice that both components have detectors and effectors appropriate to their role.

Since the goal of the learning subsystem is to improve the performance of the problem solving component, its detectors must provide a means for measuring changes in performance over time. This is typically formalized by breaking performance into two components: internal measurements relating to resources used within the robot to accomplish the task, and external measurements involving task-related aspects such as the number of correct classifications, the final score of a game, etc. Similarly, the effectors of the learning component are separated into internal effectors for making changes to the task subsystem to improve performance and external

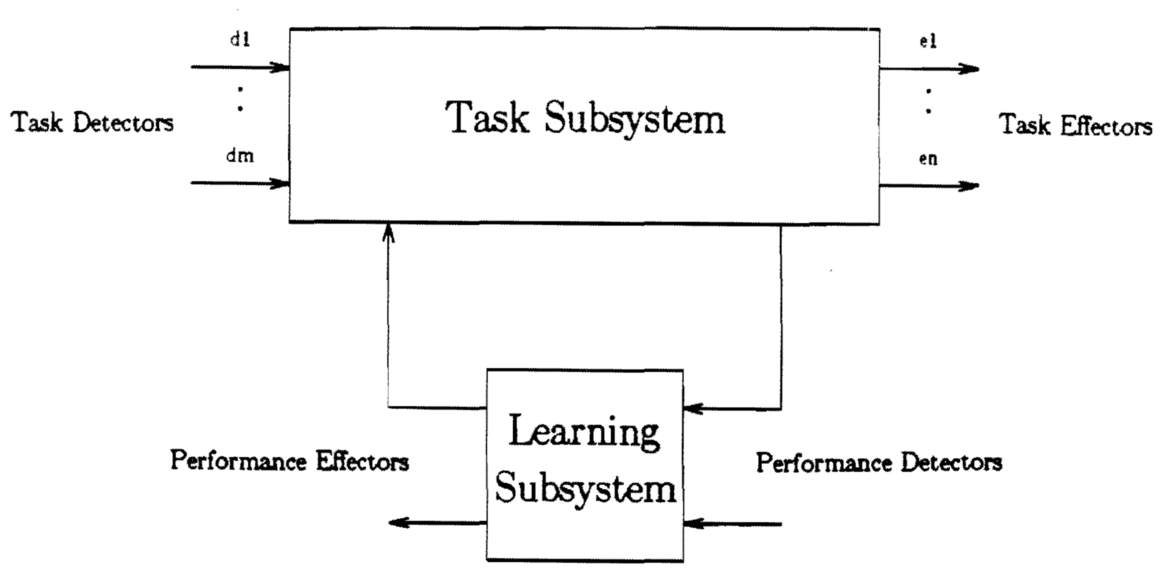

Figure 1: Model of an Adaptive System

effectors capable of making changes to the environment needed to activate further evaluation of the task subsystem.

As an example of such an adaptive system, one might consider a rule-based diagnostic expert system as the task subsystem augmented with a learning subsystem which *continually* monitors the performance of the expert system, making changes (when appropriate) to the rule base in an attempt to improve diagnostic performance.

An obvious and important issue is the extent to which domain-specific knowledge must be used in the construction of learning algorithms which are capable of effecting significant performance improvements. One can easily imagine the two extremes which occur in most areas of AI: very general methods which have a wide range of applicability, but are weak in the sense that they exhibit intolerably slow rates of learning; and very problem-specific techniques which are capable of achieving highly efficient learning, but have little use in other problem domains. The remainder of this chapter will be devoted to exploring a family of learning techniques which fall somewhere in between these two extremes. The power of these learning strategies comes from the use of an intriguing class of adaptive search techniques called *genetic algorithms* which have been studied since the early 1970s. The next section provides a brief overview of genetic algorithms and can be skipped by an already informed reader.

3. A Brief Overview of Genetic Algorithms

Many AI problems can be viewed as searching a space of legal alternatives for the best solution one can find within reasonable time and space limitations. Path planning in robotics and move selection in board games are familiar examples. What is required for such problems are techniques for rapid location of high quality solutions in search spaces of sufficient size and complexity to rule out any guarantees of optimality. When sufficient knowledge about such search spaces is available *a priori*, one can usually exploit that knowledge to develop problem-specific strategies capable of rapidly locating "satisficing" solutions. If, however, such *a priori* knowledge is unavailable, acceptable solutions are typically only achieved by *dynamically* accumulating information about the problem and using that knowledge to control the search process. Problems of this character are not hard to find. Robot path planning in an unstructured environment and games whose strategies involve identifying and exploiting the characteristics of an opponent are excellent examples. Inferring an acceptable set of classification rules from training examples without significant amounts of domain knowledge is a familiar example from classical machine learning problems.

Problems of this type which require exploitation of dynamically accumulating knowledge to control the search process are called *adaptive search problems*. Genetic algorithms are of considerable interest in this context because they represent a reasonably general, yet efficient, family of adaptive search techniques which produce acceptable performance over a broad class of problems.

3.1. Adaptive Search Techniques

To motivate the discussion of genetic algorithms consider for a moment several other strategies one might employ in dealing with combinatorially explosive search spaces about which little can be assumed to be known *a priori*. One approach might be to employ some form of random search. This can be effective if the search space is reasonably dense with acceptable solutions, so that the probability of finding one is reasonably high. However, in most cases such approaches fail to generate acceptable solutions in a reasonable amount of time because they make no use of the accumulating information about the search space to increase the probability of finding acceptable solutions.

An alternative approach is to use some form of hill climbing in which better solutions are found by exploring only those solutions which are "adjacent" to the best found so far. Techniques of this type work well on search spaces with relatively few hills, but frequently get "stuck" on local peaks which are still below an acceptable level of performance.

One way of attempting to avoid some of these problems is to combine these two strategies in creative ways such as including some random samples in addition to adjacent points while hill climbing, or restarting hill climbing from a randomly selected point when it appears to be stuck on a local peak. Although such variations frequently improve the robustness of an adaptive search strategy, they still can generate unacceptable performance because of their inability to exploit accumulating *global* information about the search space.

Statistical sampling techniques are typical alternative approaches which emphasize the accumulation and exploitation of more global information. Generally they operate by dividing the space into regions to be sampled. After sufficient sampling, most regions are discarded as unlikely to produce acceptable solutions. and the remaining regions are subdivided for further sampling, resulting in further discarding of regions. and so on. Such strategies are usually successful when the space to be searched can be divided into a reasonable number of useful subregions. They are much less effective when little is known about the appropriate granularity of the subregions, or when the required granularity is so fine that the cost of accumulating the necessary statistics for, say, 100,000 subregions is unacceptable.

Genetic algorithms, introduced initially by Holland (1975), provide an alternative approach to adaptive search problems that has proven to be a robust and effective strategy over a broad range of problems. The power of genetic algorithms comes from the fact that they blend in a natural way elements of the preceding strategies (random search, hill climbing, and sampling) together with a fourth important idea: *competition.*

3.2. The Anatomy of a Genetic Algorithm

Genetic algorithms (GAs) derive their name from the fact that they are loosely based on models drawn from the area of population genetics. These models were developed to explain how the genetic material in a population of individuals changes over time. The basic elements of these models consisted of: 1) a Darwinian notion of "fitness" which governed the extent to which an individual could influence future generations; 2) the notion of "mating" to produce offspring for the next generation; and 3) the notion of genetic operators which determine the genetic makeup of offspring from the genetic material of the parents.

These ideas can be used as components of an adaptive search procedure in the following way. Consider each point in the space to be searched as a legal instance of genetic material. Assume that for each such point, a fitness measure can be invoked to assess the quality of the solution it represents. Adaptive searching of the solution space is then achieved by simulating the dynamics of population development as illustrated in Figure 2.

The process begins by randomly generating an initial population M(0) of (typically 50-100) individuals whose genetic material represents sample points in the solution space. Each individual **m** in the population is evaluated by invoking the fitness function **u** to measure the quality

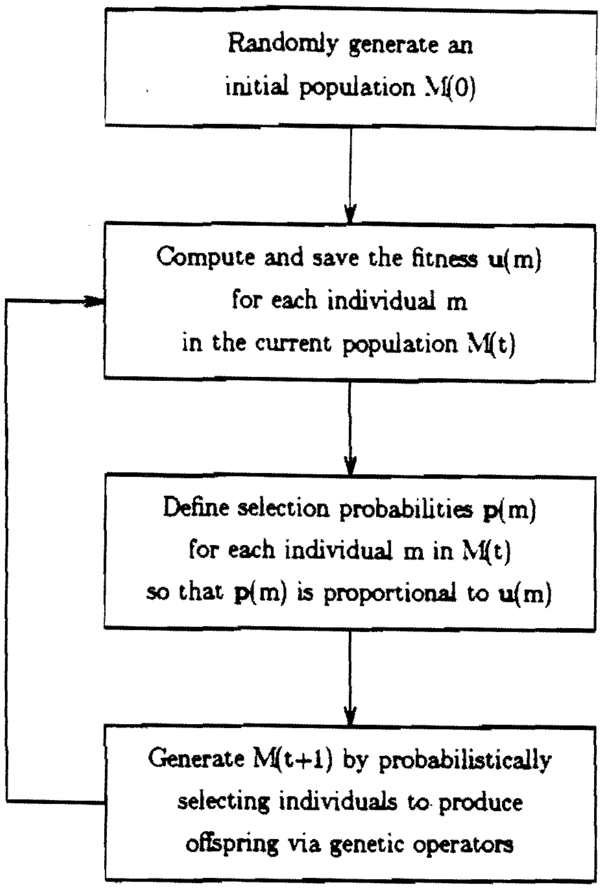
Randomly generate an initial population M(0)
Compute and save the fitness u(m) for each individual m in the current population M(t)
Define selection probabilities p(m) for each individual m in M(t) so that p(m) is proportional to u(m)
Generate M(t+1) by probabilistically selecting individuals to produce offspring via genetic operators

Figure 2: Basic Structure of Genetic Algorithms

of the solution that individual represents. A selection probability distribution p is then defined over the current population M(t) as follows: for each individual m in M(t), let p(m) = $\frac{u(m)}{\sum_j u(m_j)}$. Intuitively, this defines a probability distribution in which an individual's chance of being selected, p(m), is proportional to its observed fitness u(m). Finally, the next generation M(t+1) is produced by selecting individuals *via the selection probabilities* to produce offspring via genetic operators.

To get a feeling for how GAs work, note that the selection probabilities are defined in such a way that the expected number of offspring produced by an individual is proportional to its associated performance (fitness) value. This can be seen by considering the process of selecting individuals for reproduction as N samples from M(t) with replacement using the selection probabilities. The expected number of offspring from individual m_i is given by

$$O(m_i) = N * p(m_i) = N * \frac{u(m_i)}{\sum_j u(m_j)} = \frac{u(m_i)}{\frac{1}{N} * \sum_j u(m_j)}$$

which indicates that individuals with average performance ratings produce on the average one offspring while better individuals produce more than one and below average individuals less than one. Hence, with no other mechanism for adaptation, reproduction proportional to performance results in a sequence of generations M(t) in which the best individual in M(0) takes over a larger and larger proportion of the population.

However, in nature as well as in these artificial systems, offspring are almost never exact duplicates of a parent. It is the role of genetic operators to exploit the selection process by producing new individuals which have high-performance expectations. The choice of operators is motivated by the primary mechanisms of nature: crossover, mutation, and inversion.

In order to understand how these genetic operators produce high quality offspring, we need to briefly discuss how points in the solution space S are represented internally as genetic material. The simplest genetic algorithms represent a point in S as a single string of length L taken from some alphabet of symbols. Hence a particular solution s_i in S is represented internally as an individual $m_j = g_{j1} g_{j2} \ldots g_{jL}$ where the symbols g_{jk} play the role of genes. With this simple representation two basic genetic operators are used: crossover and mutation. The crossover operation works as follows. Whenever an individual m_i is selected from the current population M(t) to undergo reproduction, a mate m_j is also selected from M(t). Their offspring m_k is produced by concatenating segments of m_i with segments from m_j. The segments are defined by selecting at random a small number (typically 1 or 2) of crossover points from the L-1 possible crossover points. Figure 3 illustrates how an offspring might be generated using two crossover points.

$$\begin{array}{l} m_i = \text{ABCDEFGH} \\ m_j = \text{abcdefgh} \end{array} \implies m_k = \text{ABCdefGH} \text{ via crossover}$$

$$m_k = \text{ABCdefGH} \implies m_l = \text{AyCdefGH} \text{ via mutation}$$

Figure 3: Simple Crossover and Mutation Operators

Thus, the strategy employed by crossover is to construct new individuals from existing high-performance individuals by recombining subcomponents. Notice, however, that crossover will explore only those subspaces of the search space S which are already represented in M(t). If, for example, every individual in M(t) contains an N in the first gene position, crossover will never generate a new individual with an M (or any other legal gene value) in that position. A subspace may not be represented in M(t) for several reasons. It may have been deleted by selection due to associated poor performance. It may also be missing because of the limited size of M(t). In the basic GAs this problem is resolved via the second genetic operator: *mutation*.

The mutation operator generates a new individual by independently modifying one or more of the gene values of an existing individual as illustrated in Figure 3. In nature as well as these artificial systems, the probability of a gene undergoing a mutation is less than .001 suggesting that it is not a primary genetic operator. Rather, it serves to guarantee that the probability of searching a particular subspace of S is never zero.

3.3. An Analysis of Genetic Algorithms

The behavior of GAs has been formally analyzed in a variety of ways (see, for example, Holland (1975), De Jong (1975), or Bethke (1980)). The intent here is to provide the reader with a brief intuitive idea of how GAs adaptively search large, complex spaces in a relatively efficient manner.

A good way to get a feeling for the dynamics of the search process is to focus attention on certain groups of hyperplanes defined over the internal string representation of the search space. For simplicity of notation, assume that each point is S is uniquely represented by a binary string of length L. Let 0 - - ... - denote the set of points in S whose binary string representation begins with a 0. Since we are focusing on only one position in the string (in this case the first position), 0 - - ... - is called a first-order hyperplane and it contains exactly one half of the points in S. The hyperplane 1 - - ... - contains the remaining points, and together, they define a first-order *partition* of S. Of course, we could just as well have focused on any other position on the string and derived another first-order partition of S (for example, - 0 - ... - and - 1 - ... -). Similarly, one can study second-order partitions of S by fixing two of the string positions such as {00 - ... -, 01 - ... -, 10 - ... -, 11 -...-}. In general, if the binary strings are of length L, there are $\binom{L}{N}$ distinct Nth-order partitions of S.

The characterization of GAs can now be intuitively stated as follows: for any reasonably low-order hyperplane partition of S, GAs use their population M(t) of samples from S to bias subsequent search toward hyperplanes in the partition with higher expected payoff in terms of the fitness measure **u**. The power of this approach (this heuristic, if you like) is that this dynamic shifting of search occurs *simultaneously* in all low-order hyperplane partitions of S. As the search proceeds, the population M(t) reflects this bias in the sense that members of M(t) increasingly share common substrings. This growing homogeneity of M(t) indicates a reduction in the scope of the search (a focus of attention) toward high-payoff hyperplanes. As this homogeneity increases, certain positions on the strings become fixed throughout the entire population M(t). This has the effect of reducing higher-order partitions into lower-order ones which, in turn, are now subject to the same payoff bias, resulting in a further reduction in the scope of the search, and so on.

The overall effect is to produce a search technique which rapidly adapts to the characteristics of S as defined by **u** in order to home in on high-performance objects in S. Figure 4 provides a simple way to visualize this adaptive search strategy on a solution space S whose performance measure **u** defines hilly surface over S. The important point here is that GAs make no *a priori* assumptions about the characteristics of this surface. Rather, the initial population consists of a random sample of solution points in S. Using the performance feedback provided by **u**, subsequent generations "home in" on the high performance regions of S.

Note that, although our discussion here has been confined to simple, fixed length string representations of the solution space and to the simplest forms of genetic operators, similar analyses have been done for other genetic operators and more complex representations of S. These characterizations of GAs are supported by a large body of experimental work in which GAs have been applied to a broad range of problems including difficult global function optimization problems, NP-hard problems, and model-fitting problems. An excellent coverage of these activities is provided in the proceedings of the Genetic Algorithms Conferences (Grefenstette (1985), Grefenstette (1987), and Schaffer (1989)). However, our interest here is in exploring how GAs can be used to design systems that learn. The remainder of the chapter addresses this issue.

4. Using GAs for Machine Learning

We can begin to conceptualize how one might exploit the power of GAs in a learning system by referring back to the architecture of an adaptive system illustrated in Figure 1. By clearly separating out the task component from the learning component, one can focus on the ways in which the learning component can effect changes in the behavior of the task subsystem so as to improve performance over time.

In considering what kinds of changes might be made to the task component, there are a

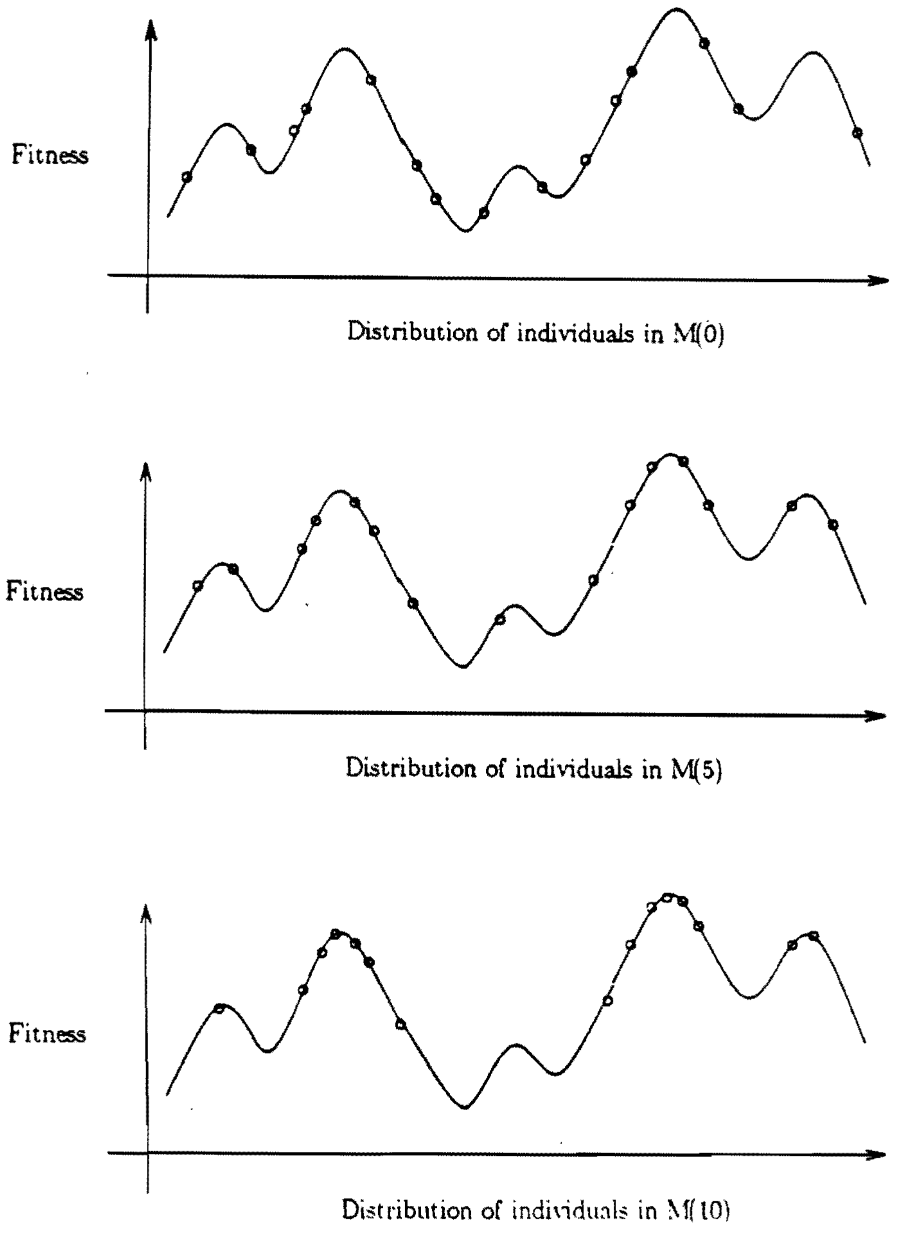
Fitness
Distribution of individuals in M(0)
Fitness
Distribution of individuals in M(5)
Fitness
Distribution of individuals in M(10)

Figure 4: Adaptive Search via Genetic Algorithms

variety of strategies of increasing sophistication and complexity. The simplest and most straightforward approach is to have GAs make changes to a set of parameters which control the behavior of a pre-developed, parameterized task program. A second and more interesting approach is to make changes to more complex data structures such as "agendas" which control the behavior of the task subsystem. A third intriguing but difficult approach is to make changes to the task program itself. Each of these possibilities is explored in more detail in the following sections.

4.1. Changing Parameters

The primary advantage of this approach to effecting behavioral changes is that it immediately places us on the familiar terrain of parameter optimization problems for which there is considerable understanding and guidance, and for which the simplest forms of GAs can be used. It is easy at first glance to discard this approach as trivial and not at all representative of what is meant by "learning". But note that significant behavioral changes can be achieved within this simple framework. Samuel's checker player is a striking example of the power of such an approach. If one views the adjustable weights and thresholds as parameters of a structurally fixed neural network, then much of the neural net learning research also falls into this category.

So, how does one use GAs to quickly and efficiently search parameter spaces for combinations of parameters which improve the performance of the task subsystem? The simplest and most intuitive approach is to think of the parameters as genes and the genetic material of individuals as a fixed length string of genes, one for each parameter. Then crossover generates new parameter combinations from existing good combinations in the current database (population) and mutation provides new parameter values.

There is a good deal of experimental and theoretical evidence to support the surprising rate at which GAs can home in on high performance parameter combinations (see, for example, De Jong (1975), Brindle (1980), or Grefenstette (1985a)). Typically, even for large search spaces (e.g., 10^{30} points), acceptable combinations are being found after only 10 simulated generations. To be fair, however, there are several issues which can catch a GA practitioner off guard when attacking a particular parameter modification problem.

The first issue has to do with the number of distinct values that genes can have. With population sizes generally in the 50-100 range, a given population can usually represent only a small fraction of the possible gene values. Since the only way of generating new gene values is via mutation, one can be faced with following dilemma. If the mutation rate is too low, there can be insufficient global sampling to prevent premature convergence to local peaks. However, by significantly increasing the rate of mutation one reverts to a form of random search which decreases the probability that new individuals will have high performance. Fortunately, this problem has both a theoretical and a practical solution, although it is not usually obvious to the casual reader.

Holland (1975) has provided an analysis of GAs which suggests that they are most effective when the number of values a gene can take on is small and that binary (two-valued) genes are in some sense optimal for GA-style adaptive search. This theoretical result translates rather naturally into what has now become standard practice in the GA community. Rather than representing a 20 parameter problem internally as strings of 20 genes (with each gene taking on many values), a binary string representation is chosen in which parameters are represented as *groups* of binary-valued genes. Although the two spaces are equivalent in that they both represent the same parameter space, GAs perform significantly better on the binary representation because, in addition to mutation, crossover is now generating new parameter values each time it combines part of a parameter's bits from one parent with those of another.

The easiest way to illustrate this point is to imagine the extreme case of a system in which there is only one parameter to be adjusted, but that parameter can take on 2^{30} distinct values. Representing this problem internally as a 1-gene problem renders crossover useless and leaves mutation as the only mechanism for generating new individuals. However, by choosing a 30-gene binary representation, crossover plays an active and crucial role in generating new parameter values with high performance expectations.

A second issue that arises in this context is that of convergence to a global optimum. Can we guarantee or expect with high probability that GAs will find the undisputed best combination of parameter settings for a particular problem? The answer is "yes and no"! One can show theoretically that every point in the search space has a non-zero probability of being sampled. However, for most problems of interest, the search space is so large as to make it impractical to wait long enough for guaranteed global optimums. A much better way to view GAs is as a powerful sampling heuristic which can rapidly find high quality solutions in large complex spaces.

In summary, one effective approach to machine learning is to restrict the kinds of changes that the learning component can make to a task program to that of parameter modification, and use GAs to quickly locate useful combinations of parameter values. The interested reader can see De Jong (1980) or Grefenstette (1985a) for more detailed examples of this approach.

4.2. Changing Data Structures

However, there are many problems for which such a simple approach is inappropriate in the sense that "more significant" structural changes to task programs seem to be required. Frequently in these situations a more complex data structure is intimately involved in controlling the behavior of the task, and so the most natural approach is to have GAs make changes to these key structures. A good example of problems of this type occur when the task systems whose behavior is to be modified are designed with top level "agenda" control mechanism. Task programs for traveling salesman problems, bin packing, and scheduling problems are frequently organized in this manner as well as systems driven by decision trees. In this context GAs are expected to select data structures to be tested, evaluated, and subsequently used to fabricate better ones.

This approach at first glance may not seem to introduce any difficulties as far as using GAs, since it is usually not hard to "linearize" these data structures, map them into a string representation which can be manipulated by GAs, and then reverse the process to produce new data structures for evaluation. However, there are some issues here as well which the designer of a learning system must be familiar with in order to make effective use of GAs.

Just as we saw in the previous section on searching parameter spaces, these issues center around the way in which the space (in this case, a space of data structures) to be searched is represented internally for manipulation by GAs. It is not difficult to invent internal string representations for agendas and other complex data structures which have the following property: almost every new structure produced by the standard crossover and mutation operators is an internal representation of an *illegal* data structure!

My favorite example of this is to consider how one might use GAs to quickly find good agendas (tours) for a traveling salesman who needs to visit N cities exactly once minimizing the distance traveled. The most straightforward approach would be to internally represent a tour as N genes and the value of each gene is the name of the next city to be visited. Notice, however, that GAs using the standard crossover and mutation operators will explore the space of all *combinations* of city names when, in fact, it is the space of all *permutations* which is of interest. The obvious problem is that as N increases, the space of permutations is a vanishingly small subset of the space of combinations, and the powerful GA sampling heuristic has been rendered impotent by a poor choice of representation.

Fortunately, a sensitivity to this issue is usually sufficient to avoid it in one of several ways. One approach is design an alternate representation for the same space for which the traditional genetic operators are appropriate. This has been done for a variety of problems of this type including the traveling salesman problem (see, for example, Grefenstette (1985b), Goldberg (1985b)), or Davis (1985)).

An equally useful alternative is to select different genetic operators which are more appropriate to "natural representations". For example, in the case of traveling salesman problems, a genetic-like inversion operator (which can be viewed as a particular kind of permutation operator) is clearly a more "natural" operator. Similarly, representation-sensitive crossover and mutation operators can be defined to assure offspring represent legal points in the solution space.

The key point to be made here is that there is nothing sacred about the traditional string-oriented genetic operators. The mathematical analysis of GAs indicates that they work best when the internal representation encourages the emergence of useful building blocks which can be subsequently combined with other to produce improved performance. String representations are just one of many ways of achieving this.

4.3. Changing Executable Code

Perhaps by now the reader is ready to reply that neither of the approaches just discussed "really" involves learning. Rather, the reader has in mind the ability to effect behavioral changes in the task system by making changes to the task program itself. I'm not sure that there is anything fundamentally different between interpreting an agenda and executing a Pascal program. However, I think that most everyone will agree that, in general, program spaces are very large and complex. In any case, there is good deal of interest in designing systems which learn at this level. The remainder of the chapter will discuss how GAs are used in such systems.

4.3.1. Choosing a Programming Language

It is quite reasonable view programs written in conventional languages like Fortran and Pascal (or even less conventional languages like Lisp and Prolog) as linear strings of symbols. This is certainly the way they are treated by editors and compilers in current program development environments. However. it is also quite obvious that this "natural" representation is a disastrous one as far as traditional GAs are concerned since standard operators like crossover and mutation seldom produce syntactically correct programs and. of those. even fewer which are semantically correct.

One alternative is to attempt to devise new language-specific "genetic" operators which preserve at least the syntactic (and hopefully, the semantic) integrity of the programs being manipulated. Unfortunately, the complexity of both the syntax and semantics of traditional languages makes it difficult to develop such operators. An obvious next step would be to focus on less traditional languages such as "pure" Lisp whose syntax and semantics are much simpler, leaving open the hope of developing reasonable genetic operators with the required properties. There have been a number of activities in this area (see, for example, Fujiko (1987) or Koza (1989)).

However, there is at least one important feature that "pure" Lisp shares with other more traditional languages: they are all procedural in nature. As a consequence most reasonable representations have the kinds of properties that cause considerable difficulty in GA applications. The two most obvious representation problems are order dependencies (interchanging two lines of code can render a program meaningless) and context sensitive interpretations (the entire meaning of a section of code can be changed by minor changes to preceding code, such as the insertion or deletion of a punctuation symbol). A more detailed discussion of these representation problems is presented in De Jong (1985).

These issues are not new and were anticipated by Holland to the extent that he proposed a family of languages called classifier languages which were designed to overcome the kinds of problems being raised here (Holland 1975). What is perhaps a bit surprising is that these classifier languages are a member of a broader class of languages which continues to reassert its usefulness across a broad range of activities (from compiler design to expert systems), namely, production systems (PSs) or rule-based systems. As a consequence, a good deal of time and effort has gone into studying this class of languages as a suitable language for use in evolving task programs with GAs.

4.3.2. Learning PS Programs

One of the reasons that production systems have been and continue to be a favorite programming paradigm in both the expert system and machine learning communities is that PSs provide a representation of knowledge which can *simultaneously* support two kinds of activities: 1) treating knowledge as data to be manipulated as part of a knowledge acquisition and refinement process, and 2) treating knowledge as an "executable" entity to be used to perform a particular task (see, for example, Newell (1977), Buchanan (1978), or Hedrick (1976)). This is particularly true of the "data-driven" forms of PSs (such as OPS5) in which the production rules which make up a PS program are treated as an unordered set of rules whose "left hand sides" are all independently and in parallel monitoring changes in "the environment".

It should be obvious that this same programming paradigm offers significant advantages for GA applications and, in fact, has precisely the same characteristics as Holland's early classifier languages. If we focus then on PSs whose programs consist of unordered collections (sets) of rules, we can then ask how GAs can be used to search the space of PS programs for useful rule sets.

To anyone whose has read Holland's book (Holland 1975), the most obvious and "natural" way to proceed is to represent an entire rule set as string (individual), maintain a population of candidate rule sets, and use selection and genetic operators to produce new generations of rule sets. Historically, this was the approach taken by De Jong and his students while at the University of Pittsburgh (see, for example, Smith (1980), or Smith (1983)) and has been dubbed "the Pitt approach".

However, during that same time period, Holland developed a model of cognition (*classifier systems*) in which each member of the population represents a rule, and the entire population corresponds to a single rule set (see, for example, Holland (1978)) and Booker (1982)). This quickly became known as "the Michigan approach" and initiated a continuing (friendly, but provocative) series of discussions concerning the strengths and weaknesses of the two approaches.

4.3.2.1. The Pitt Approach

If we adopt the view that each individual in a GA population represents an *entire* PS program, there are several issues which must be addressed. The first is the (by now familiar) choice of representation. The most immediate "natural" representation which comes to mind is to regard individual rules as genes, and entire programs as strings of these genes. Then, crossover serves to provide new combinations of rules and mutation provides new rules. Notice, however, that we have chosen a representation in which genes can take on many values. As discussed in the previous section on parameter modification, this can result in premature convergence when population sizes are typically 50-100. Since individuals represent entire PS programs, it is unlikely that one can afford to significantly increase the size of the population. Nor, as we have seen, does it help to increase the rate of mutation. Rather, we need to move toward an internal binary representation of the space of PS programs so that crossover is also involved in constructing new rules from parts of existing rules.

If we go directly to a binary representation, we must now exercise care that crossover and mutation are appropriate operators in the sense in that they produce new high-potential individuals from existing ones. The simplest way to guarantee this is to assume that all rules have a fixed-length, fixed-field format. Although this may seem restrictive in comparison with the flexibility and variability of OPS5 or Mycin rules, it has proven to be quite adequate when working at a lower sensory level. At this level one typically has a fixed number of detectors and effectors, so that condition-action rules quite naturally take the form of a fixed number of detector patterns to be matched together with the appropriate action. Many of the successful classifier systems make this assumption (see, for example, Wilson (1985) or Goldberg (1985a, 1989)).

It is not difficult, however, to relax this assumption and allow more flexible rule sets without subverting the power of the genetic operators. This is achieved by making the operators "representation sensitive" in the sense that they no longer make arbitrary changes to linear bit strings. Rather, the internal representation is extended to provide "punctuation marks" so that meaningful changes are made. For example, if the crossover operator chooses to break parent 1 on a rule boundary, it also breaks parent 2 on a rule boundary, and so on. This is the approach used successfully on the LS systems of Smith (1983) and Schaffer (1985).

A second representation-related issue which arises in the Pitt approach has to do with the number of rules in a rule set. If we think of rule sets as programs or knowledge bases, it seems rather silly and artificial to demand that all rule sets be the same size. Historically, however, all of the analytical results and most of the experimental work was done with GAs which maintained populations of fixed-length strings.

One can adopt the same view using the Pitt approach and require all rule sets (strings) to be the same fixed length. The justification is usually in terms of the advantages of having redundant copies of rules and having "workspace" within a rule set for new experimental building blocks without having to necessarily replace existing ones. However, Smith (1980) was able to show that the formal results could indeed be extended to variable length strings. He complemented those results with a GA implementation which maintained a population of variable length strings and which efficiently generated variable length rule sets for a variety of tasks. One of the interesting side issues of this work was the effectiveness of providing via feedback an "incentive" to keep down the size of the rule sets by including a "bonus" for achieving the same level of performance with a shorter string.

With these issues resolved, GAs have been shown to be surprisingly effective in producing non-trivial rule sets for such diverse tasks as solving maze problems, playing poker, and gait classification. The interested reader can see, for example, Smith (1983) or Schaffer (1985) for more details.

4.3.2.2. The Michigan Approach

A quite different approach to learning PS programs was developed by Holland and his colleagues while working on computational models of cognition. In this context, it seemed natural to view the knowledge (experience) of a particular person (cognitive entity) as a collection of rules which are modified over time via interaction with one's environment. Unlike genetic material, this kind of knowledge doesn't evolve over generations via selection and mating. Rather, it accumulates "in real time" as individuals struggle to cope with their environment. Out of this perspective came a family of cognitive models called *classifier systems* in which *rules* rather than *rules sets* are the internal entities manipulated by GAs. There are excellent descriptions of this approach in Machine Learning II (Holland 1986) and in Goldberg's recent book (Goldberg 1989). Wilson and Goldberg (Wilson 1989) also provide an excellent critical review of the classifier approach. So, the details of classifier systems need not be repeated here. Instead, I want to focus on how the two approaches differ.

I think it is fair to say that most people who encounter classifier systems *after* becoming familiar with the traditional GA literature are somewhat surprised at the emergence of the rather elaborate "bucket brigade" mechanism to deal with apportionment of credit issues. In the traditional genetic view apportionment of credit is handled via the emergence of "co-adapted" sets of gene values in the population. The idea is that combinations of gene values which work well together have a higher than average likelihood of being represented in subsequent generations. As the frequency of a particular combination increases, it is also less likely that this set will be broken up via crossover since it is clearly the case that, if both parents have a co-adapted set, so will any offspring produced by crossover. Intuitively, these initially emerging co-adapted sets get combined with other sets, forming larger sets with improved performance which replace inferior ones, and so on.

If we now reinterpret these ideas in the context of PS programs, the process of interest is the emergence of co-adapted sets of *rules*. Because the Pitt approach maps *entire* PS programs into strings, one gets this co-adaption mechanism "for free". One can observe over time the emergence in the population of above average rule sets, which combine with other sets of rules to form larger rule sets with improved performance. Because in classifier systems the population represents a single PS program, co-adapted sets of rules must emerge from some other mechanism, namely the bucket brigade.

Which approach is better in the sense of being more effective in evolving PS programs? It is too early to tell what the answer might be or even if the question is a valid one. There are equally impressive examples of classifier systems which have solved problems involving the regulation of gas flow through pipe lines (Goldberg 1985a, 1989), controlling vision systems (Wilson 1985), and inferring boolean functions (Wilson 1987). The current popular view is that the classifier approach will prove to be most useful in an on-line, real-time environment in which radical changes in behavior cannot be tolerated whereas the Pitt approach will be useful with off-line environments in which more leisurely exploration and more radical behavioral changes are acceptable.

What I find exciting and provocative is that there are some recent developments which suggest that it might be possible to combine the two approaches in powerful and interesting ways (Grefenstette 1988, 1989).

4.3.3. PS Architecture Issues

So far we have been focusing on representation issues in an attempt to understand how GAs can be used to learn PS programs. The only constraint on production system architectures that has emerged so far is the observation that GAs are much more effective on PS programs which consist of unordered rules. In this section I will attempt to summarize any other implications that the use of GAs might have on PS architectural decisions.

4.3.3.1. The Left Hand Side of Rules

Many of the rule-based expert system paradigms (e.g., Mycin-like shells) and most traditional programming languages provide an IF-THEN format in which the left hand side is a boolean expression to be evaluated. This boolean sublanguage can itself become quite complex syntactically and can raise many of the representation issues discussed earlier. In particular, variable length expressions, varying types of operators and operands, and function invocations make it difficult to choose a representation in which the traditional string-oriented genetic operators are useful. However, genetic operators which "know about" the syntax of the boolean sublanguage can be developed and have been shown to be effective in producing high-level (symbolic) rule sets. The Samuel system (Grefenstette 1989) is an excellent example of this approach.

An alternate approach used in languages like OPS5 and SNOBOL is to express the "conditions" of the left hand side as patterns to be matched. Unfortunately, the pattern language used can easily be as complex as boolean expressions and in many cases more complex because of the additional need to "save" objects being matched for later use in the pattern or on the right hand side.

As we have seen, the GA implementor must temper the style and complexity of the left hand side with the need for an effective internal representation. As a consequence, most implementations have followed Holland's lead and have chosen the simple {0, 1, #} fixed length pattern language which permits a relatively direct application of traditional genetic operators. Combined with internal working memory, such languages can be shown to be computationally complete. However, this choice is not without problems. The rigid fixed length nature of the patterns can lead to very complex and "creative" representations of the objects to be matched. Simple relationships like "speed [illegible] 200" may require multiple rule firings and internal memory in order to be correctly evaluated. As discussed earlier, some of this rigidity can be alleviated by the use of context-sensitive genetic operators (Smith 1983). However, finding a more pleasing compromise between simplicity and expressive power of the left hand sides of rules is an active and open area of research.

A favorite cognitive motivation for preferring pattern matching rather than boolean expressions is the feeling that "partial matching" is one of the powerful mechanisms that humans use to deal with the enormous variety of every day life. The observation is that we are seldom in precisely the same situation twice, but manage to function reasonably well by noting its similarity to previous experience.

This has led to some interesting discussions as to how "similarity" can be captured computationally in a natural and efficient way. Holland and other advocates of the {0, 1, #} paradigm argue that this is precisely the role that the "#" plays as patterns evolve to their appropriate level of generality. Booker and others have felt that requiring perfect matches *even* with the {0, 1, #} pattern language is still too strong and rigid a requirement, particularly as the length of the left hand side pattern increases. Rather than returning simply success or failure, they feel that the pattern matcher should return a "match score" indicating how close the pattern came to matching. An important issue here which needs to be better understood is how one computes match scores in a reasonably general, but computationally efficient manner. The interested reader can see Booker (1985) for more details.

4.3.3.2. Working Memory

Another PS architectural issue revolves around the decision as to whether to use "stimulus-response" production systems in which left hand sides only "attend to" external events and right hand sides consist only of invocations of external "effectors", or whether to use the more general OPS5 model in which rules can also attend to and make changes to an internal working memory.

Arguments in favor of the latter approach involve the observation that the addition of working memory provides a more powerful computational engine which is frequently required with fixed length rule formats. The strength of this argument can be weakened somewhat by noting that in some cases the external environment *itself* can be used as working memory.

Arguments against including working memory generally fall along the lines of: 1) the application doesn't need the additional generality and complexity, 2) concerns about how one bounds the number of internal actions before generating the next external action (i.e., the halting problem), or 3) pointing out that most of the more traditional machine learning work in this area (e.g., Michalski (1983)) has focused on stimulus-response models.

Most of the implementations of working memory provide a restricted form of internal memory, namely, a fixed format, bounded capacity message list (Holland 1978, Booker 1982). However, it's clear that there are plenty of uses for both architectures. The important point here is that this choice is not imposed on us by GAs.

4.3.3.3. Parallelism

Another side benefit of PSs with working memory is that they can be easily extended to allow parallel rule firings. In principle, the only time conflict resolution (serialization) needs to occur is when an "external effector" is to be activated. Hence, permitting parallel firing of rules invoking internal actions is a natural way to extend PS architectures in order to exploit the power of parallelism. Whether this power is appropriate for a particular application is of course a decision of the implementor. An excellent example of a parallel implementation can be found in Robertson (1988, 1989).

4.3.4. The Role of Feedback

So far, in attempting to understand how GAs can be used to learn PS programs, we have discussed how PS programs can be represented and what kinds of PS architectures can be used to exploit the power of GAs. In this section we focus on a third issue: the role of feedback.

Recall that the intuitive view of how GAs search large, complex spaces is via a sampling strategy which is adaptive in the sense that feedback from current samples is used to bias subsequent sampling into regions with high expected performance. This means that, even though we have chosen a good representation and have selected an appropriate PS architecture, the effectiveness of GAs in learning PS programs will also depend on the usefulness of the information obtained via feedback. Since the designer typically has a good deal of freedom here, it is important that the feedback mechanism be chosen to facilitate this adaptive search strategy.

Fortunately, there is a family of feedback mechanisms which are both simple to use and which experience has shown to be very effective: *payoff functions*. This form of feedback uses a classical "reward and punishment" scheme in which performance evaluation is expressed in terms of a payoff value. GAs can use this information (almost) directly to bias the selection of parents used to produce new samples (offspring). Of course, not all payoff functions are equally useful in this role. It is important that the function chosen provide useful information early in the search process to help focus the search. For example, a payoff function which is zero almost everywhere provide almost no information for reducing the scope of search process.

The way in which payoff is obtained differs somewhat depending on whether one is using the Pitt or Michigan approach. In classifier systems the bucket brigade mechanism stands ready to distribute payoff to those *rules* which are deemed responsible for achieving that payoff. Because payoff is the currency of the bucket brigade economy, it is important to design a feedback mechanism which provides a relatively steady flow of payoff rather than one in which there are long "dry spells". Wilson's "animat" environment is an excellent example of this style of payoff (Wilson 1985).

The situation is somewhat different in the Pitt approach in that the usual view of evaluation consists of injecting the PS program defined by a particular individual into the task subsystem and evaluating how well that program *as a whole* performed. This view can lead to some interesting considerations such as whether to reward programs which perform tasks as well as others, but use less space (rules) or time (rule firings). Smith (1980) found it quite useful to break up the payoff function into two components: a task-specific evaluation and a task-independent measure of the program itself. Although these two components were combined into a single payoff value, work by Schaffer (1985) suggests that it might be more effective to use a vector-valued payoff function in situations such as this.

There is still a good deal to be learned about the role of feedback both from an analytical and empirical point of view. Bethke (1980) has used Walsh transforms as the basis for a formal understanding of the kind of feedback information which is best suited for GA-style adaptive search. Recent experimental work suggests that it may be possible to combine aspects of both the Michigan and Pitt approaches via a multilevel credit assignment strategy which assigns payoff to both rule sets as well as individual rules (Grefenstette 1988, 1989). This is an interesting idea which will generate a good deal of discussion and merits further attention.

4.3.5. The Use of Domain Knowledge

It is conventional to view GAs as "weak" search methods in the sense that they can be applied without requiring any knowledge of the space being searched. However, a more accurate view is that, although no domain knowledge is required, there are ample opportunities to exploit domain knowledge if it is available. We have already seen a few examples of how domain knowledge can be used. As designers we select the space to be searched and the internal representation to be used by GAs. As discussed in the previous sections, such decisions require knowledge about both the problem domain and the characteristics of GAs. Closely related to representation decisions is the choice of genetic operators to be used. As we have seen, a good deal of domain knowledge can go into the selection of effective operators. Grefenstette (1987b) has an excellent discussion of this.

A more direct example of the use of domain knowledge involves the choice of the initial population used to start the search process. Although we have described this as a "random" initial population, there is no reason to start with an "empty slate" if there is *a priori* information available which permits the seeding of the initial population with individuals known to have certain kinds or levels of performance.

A third and perhaps the most obvious way of exploiting domain knowledge is by means of the feedback mechanism. As we have seen, the effectiveness GAs depends on the usefulness of the feedback information provided. Even the simplest form of feedback, namely payoff-only systems, can and frequently does use domain knowledge to design an effective payoff function. More elaborate forms of feedback such as the vector-valued strategies and multi-level feedback mechanisms discussed above provide additional opportunities to incorporate domain-specific knowledge.

In practice, then, we see a variety of scenarios ranging from the use of "vanilla" GAs with little or no domain-specific modifications to highly creative applications in which a good deal of domain knowledge has been used.

5. An Example: The LS-1 Family

In the previous section we described three broad classes of techniques for designing genetic algorithm based learning (GABL) systems. A common theme throughout that discussion was the representation issue: choosing a representation of the space to be searched which is both natural to the application and appropriate for GAs. In this section we illustrate these ideas by presenting in more detail one particular family of GABL systems, the LS-1 family (Smith 1980, 1983), which has been successfully applied to a wide variety of task domains.

LS-1 systems are designed to allow entire task programs to be learned. An LS-1 system consists of three basic components: a task subsystem, a critic, and a learning subsystem. The task subsystem is equipped with a fixed set of detectors (sensors) and effectors (actions) assumed to be useful for the task to be learned. Task programs consist of production rules whose left hand sides represent patterns to be matched against the current contents of working memory and the current external detector values. The right hand sides specify one of two types of actions to be taken: changes to working memory or activating an external effector. An execution cycle consists of firing *in parallel all* rules whose left hand side matches something in working memory. Conflict resolution is only performed when on a given execution cycle more than one request is made to activate an external effector. Conceptually, task programs have a strong OPS5-like flavor (unordered, forward chaining, pattern matching) without the restriction that only one rule can fire each cycle.

If we want to use GAs to evolve these rule-based task programs, we must choose an internal representation of the space to be searched. LS-1 uses "the Pitt approach" in which each individual in a GA population represents an entire task program and, hence, a given population represents (typically 50-100) competing task programs. Each task program is submitted to the critic subsystem which, in turn, injects the task program into the task subsystem and evaluates the "fitness" of the task program by observing its behavior on a series of tasks. After the entire population of task programs has been evaluated, a new generation of task programs is created by probabilistically selecting parents on the basis of fitness, and using crossover, inversion, and mutation to produce offspring which are "interesting" variants of their parents.

As discussed in the previous section, there are two ways of guaranteeing that the genetic operators produce viable offspring: 1) by building into the operators specific language and domain knowledge so that crossover, mutation, and inversion produce only syntactically and semantically correct rule sets: or 2) restricting the language so that the standard operators preserve syntactic and semantic integrity. LS-1 adopts the latter strategy, preferring to buy more domain independence at the cost of a less flexible language. It achieves this by requiring that every rule have an identical, fixed length format as follows:

pd1 pd2 ... pdn pw1 pw2 ... pwm --> some_action

where the pdi represent patterns to be matched against the n external detector values, and the pwj represent m patterns to be matched against the contents of working memory. LS-1 uses Holland's {0, 1, #} pattern language, so that the length of a particular pattern is determined by the binary representation of the objects to be matched. So, for example, the following rules:

```
        pd1     pd2       pw1            action
R1:   00#    #11##    1#1#01##  -->  010
R2:   1##    0011#    ##110##0  -->  110
```

might represent a situation in which the first pattern of each matches a 3-bit detector, the second pattern matches a 5-bit detector, the third pattern matches 8-bit working memory cells, and the right hand side selects 1 of 8 possible actions.

A rule set is therefore represented internally as a variable-length string of fixed-length rules. The important point is that the standard crossover and mutation operators always produce viable offspring. If we restrict inversion to occur only at rule boundaries, the same is is true for it.

So, how does one apply LS-1 to a particular task domain? First, the number and sizes of each external detector must be chosen (analogous to selecting feature vectors) and the set of legal operations (effectors) specified. Second, the number of internal detectors and actions on working memory must be specified (no internal activity corresponds to a stimulus-response system). Finally, a domain-specific critic must be developed to provide fitness feedback for each rule set generated during the learning process.

What is surprising is that, even though LS-1 assumes a fairly rigid rule format, there is considerable empirical evidence of its robustness in rapidly generating high-performance rule sets (task programs) for problem domains as diverse as maze puzzles, poker playing, human gait classification, VLSI layout problems, and network scheduling.

6. Summary and Conclusions

The goal of this chapter has been to understand how GAs might be used to design systems which are capable of learning. The approach has been to visualize learning systems as consisting of two components: a task subsystem whose behavior is to be modified over time via learning, and a learning subsystem whose job is to observe the task subsystem over time and effect the desired behavioral changes. This perspective focuses attention on the kinds of *structural* changes a learning subsystem might possibly make to a task subsystem in order to effect *behavioral* changes. Three classes of structural changes of increasing complexity were identified: parameter modification, data structure manipulation, and changes to executable code.

Having characterized learning in this way, the problem can be restated as one of searching the space of legal structural changes for instances which achieve the desired behavioral changes. In domains for which there is a strong theory to guide this search, it would be silly not to exploit such knowledge. However, there are many domains in which uncertainty and ignorance preclude such approaches and require learning algorithms to discover (infer) the important characteristics of these search spaces *while* the search is in progress. This is the context in which GAs are most effective. Without requiring significant amounts of domain knowledge, GAs have been used to rapidly search spaces from each of the categories listed above.

At the same time it is important to understand the limitations of such an approach. In most cases several thousand samples must be taken from the search space before high quality solutions are found. Clearly, there are many domains in which such a large number of samples is out of the question. At the same time, generating several thousand examples can frequently involve much less effort than building by hand a sufficiently strong domain theory. It is also true that choosing a good internal representation for the search space to tends to be a more difficult task as the complexity of the search space increases, thus reducing the effectiveness of

GAs. Finally, even with a good internal representation, care must be taken to provide an effective feedback mechanism. If, for example, the only feedback about a task program is "yes, it works" or "no, it fails", there is no information about the quality of partial solutions on which GAs depend to construct new trial solutions.

Hence, GABL should be viewed as another tool for the designer of learning systems which, like other more familiar tools such as similarity-based techniques and explanation-based approaches, is not the answer to all learning problems, but provides an effective strategy for specific kinds of situations.

References

Bethke, A. (1980). *Genetic Algorithms as Function Optimizers*, Doctoral Thesis, CCS Department, University of Michigan, Ann Arbor, MI.

Booker, L. B. (1982). *Intelligent Behavior as an Adaptation to the Task Environment.* Doctoral Thesis, CCS Department, University of Michigan, Ann Arbor, MI.

Booker, L. B. (1985). Improving the Performance of Genetic Algorithms in Classifier Systems. *Proc. First Int'l Conference on Genetic Algorithms and their Applications* (pp. 80-92). Pittsburgh, PA.

Brindle, A. (1980). *Genetic Algorithms for Function Optimization*, Doctoral Thesis, Department of Computing Science, University of Alberta, Alberta, Canada.

Buchanan, B., Mitchell, T.M. (1978). Model-Directed Learning of Production Rules. *Pattern-Directed Inference Systems*, Waterman, D. and Hayes-Roth, F. (Eds.), Academic Press.

Davis, L. D. (1985), Job Shop Scheduling Using Genetic Algorithms. *Proc. First Int'l Conference on Genetic Algorithms and their Applications* (pp. 136-140). Pittsburgh, PA.

De Jong, K. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Doctoral Thesis, CCS Department, University of Michigan, Ann Arbor, MI.

De Jong, K. (1980). Adaptive System Design: A Genetic Approach. *IEEE Trans. on Systems, Man, and Cybernetics* (pp. 556-574), Vol. 10. #9.

De Jong, K. (1985). Genetic Algorithms: a 10 Year Perspective. *Proc. First Int'l Conference on Genetic Algorithms and their Applications* (pp. 169-177). Pittsburgh. PA.

De Jong, K. (1987). On Using Genetic Algorithms to Search Program Spaces. *Proc. Second Int'l Conference on Genetic Algorithms and their Applications* (pp. 169-177). Cambridge, MA.

Fujiko, C. and Dickinson, J. (1987). Using the Genetic Algorithm to Generate Lisp Code to Solve the Prisoner's dilemma. *Proc. Second Int'l Conference on Genetic Algorithms and their Applications* (pp. 236-240). Cambridge, MA.

Goldberg, D. (1985a). Genetic Algorithms and Rule Learning in Dynamic System Control. *Proc. First Int'l Conference on Genetic Algorithms and their Applications* (pp. 8-15). Pittsburgh, PA.

Goldberg, D. and Lingle, R. (1985b). Alleles, Loci, and the Traveling Salesman Problem. *Proc. First Int'l Conference on Genetic Algorithms and their Applications* (pp. 154-159). Pittsburgh, PA.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing.

Grefenstette, J. (1985a). Optimization of Control Parameters for Genetic Algorithms. *IEEE Trans. on Systems, Man, and Cybernetics* (pp. 122-128), Vol. 16, #1.

Grefenstette, J., Gopal, R., Rosmita, B., and Van Gucht, D. (1985b). Genetic Algorithms for the Traveling Salesman Problem. *Proc. First Int'l Conference on Genetic Algorithms and their Applications* (pp. 160-168). Pittsburgh, PA. Lawrence Erlbaum Publishers.

Grefenstette, J. (1987a). Multilevel Credit Assignment in a Genetic Learning System. *Proc. Second Int'l Conference on Genetic Algorithms and their Applications* (pp. 202-209). Cambridge, MA. Lawrence Erlbaum Publishers.

Grefenstette, J. (1987b). Incorporating problems specific knowledge into Genetic Algorithms. *Genetic Algorithms and Simulated Annealing.* D. Davis (Ed.). Pitman Series on AI.

Grefenstette, J. (1988). Credit Assignment in Rule Discovery Systems. *Machine Learning Journal* (pp. 225-246), Volume 3, numbers 2/3, Kluwer Academic Publishers.

Grefenstette, J. (1989). A System for Learning Control Strategies with Genetic Algorithms. *Proc. Third International Conference on Genetic Algorithms* (pp. 183-90). Fairfax, VA. Morgan Kaufmann Publishers.

Hedrick, C. L. (1976). Learning Production Systems from Examples. *Artificial Intelligence*, Vol. 7.

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

Holland, J. H., Reitman, J. (1978) Cognitive Systems Based on Adaptive Algorithms. *Pattern-Directed Inference Systems* Waterman, D. and Hayes-Roth, F. (Eds.). Academic Press.

Holland, J. H. (1986). Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems. *Machine Learning: An Artificial Intelligence Approach*, Vol. II Michalski, R. S., Carbonell, J. G., and Mitchell, T. M (Eds.). Morgan Kaufman Publishing.

Holland, J. H., Holyoak, K., Hisbett, R., & Thagard, P. (1986). *Induction: Processes of Inference, Learning, and Discovery.* MIT Press. Cambridge, MA.

Koza, J. (1989). Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. *Proc. 11th Int'l Joint Conference on AI.* Detroit. MI.

Michalski, R. (1983). A Theory and Methodology of Inductive Learning. *Machine Learning: An Artificial Intelligence Approach.* Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (Eds.). Tioga Publishing.

Newell, A. (1977) Knowledge Representation Aspects of Production Systems. *Proc. 5th IJCAI* (pp. 430-437). Cambridge, MA,

Robertson, G. (1988). Parallel Implementation of Genetic Algorithms in a Classifier System. *Genetic Algorithms and Simulated Annealing.* D. Davis (Ed.). Pitman Series on AI.

Robertson, G. & Riolo, R. (1989). A Tale of Two Classifier Systems. *Machine Learning Journal* (pp. 139-160), Volume 3, numbers 2/3, Kluwer Academic Publishers.

Schaffer, D. (1985). Multiple Objective Optimization with Vector Evaluated Genetic Algorithms. *Proc. First Int'l Conference on Genetic Algorithms and their Applications* (pp. 93-100). Pittsburgh, PA.

Schaffer, D. (1989). Editor, *Proc. Third International Conference on Genetic Algorithms.* Morgan Kaufmann Publisher.

Smith, S. F. (1980). *A Learning System Based on Genetic Adaptive Algorithms.* Doctoral Thesis, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA.

Smith, S. F. (1983). Flexible Learning of Problem Solving Heuristics Through Adaptive Search. *Proc. 8th IJCAI* (pp. 422-425), Karlsruhe, Germany.

Wilson, S. (1985). Knowledge Growth in an Artificial Animal. *Proc. First Int'l Conference on Genetic Algorithms and their Applications* (pp. 16-23). Pittsburgh, PA.

Wilson, S. (1987). Quasi-Darwinian Learning in a Classifier System (pp. 59-65). *Proc. 4th Intl. Workshop on Machine Learning.* Irvine, CA.

Wilson, S. and Goldberg, D. (1989). A Critical Review of Classifier Systems *Proc. Third International Conference on Genetic Algorithms* (pp. 244-255). Fairfax, VA. Morgan Kaufmann Publishers.