Machine Learning 3: 121-138, 1988

Learning with Genetic Algorithms: An Overview

KENNETH DE JONG (KDEJONG@GMU90X.GMU.EDU)
Computer Science Department, George Mason University, Fairfax, VA 22030, U.S.A.

(Received: January 15, 1988)
(Revised: May 20, 1988)

Keywords: Genetic algorithms, competition-based learning, learning task programs, classifier systems

Abstract. Genetic algorithms represent a class of adaptive search techniques that have been intensively studied in recent years. Much of the interest in genetic algorithms is due to the fact that they provide a set of efficient domain-independent search heuristics which are a significant improvement over traditional "weak methods" without the need for incorporating highly domain-specific knowledge. There is now considerable evidence that genetic algorithms are useful for global function optimization and NP-hard problems. Recently, there has been a good deal of interest in using genetic algorithms for machine learning problems. This paper provides a brief overview of how one might use genetic algorithms as a key element in learning systems.

1. Introduction

The variety and complexity of learning systems makes it difficult to formulate a universally accepted definition of learning. However, a common denominator of most learning systems is their capability for making *structural changes* to themselves over time with the intent of improving performance on tasks defined by their environment, discovering and subsequently exploiting interesting concepts, or improving the consistency and generality of internal knowledge structures.

Given this perspective, one of the most important means for understanding the strengths and limitations of a particular learning system is a precise characterization of the structural changes that are permitted and how such changes are made. In classical terms, this corresponds to a clear understanding of the space of possible structural changes and the legal operators for selecting and making changes.

This perspective also lets one more precisely state the goal of the research in applying genetic algorithms to machine learning, namely, to understand when and how genetic algorithms can be used to explore spaces of legal structural changes in a goal-directed manner. This paper summarizes our current understanding of these issues.

2. Exploiting the power of genetic algorithms

Genetic algorithms (GAs) are a family of adaptive search procedures that have been described and extensively analyzed in the literature (De Jong, 1980; Grefenstette, 1986; Holland, 1975). GAs derive their name from the fact that they are loosely based on models of genetic change in a population of individuals. These models consist of three basic elements: (1) a Darwinian notion of "fitness," which governs the extent to which an individual can influence future generations; (2) a "mating operator," which produces offspring for the next generation; and (3) "genetic operators," which determine the genetic makeup of offspring from the genetic material of the parents.

A key point of these models is that adaptation proceeds, not by making incremental changes to a single structure (e.g., Winston, 1975; Fisher, 1987), but by maintaining a population (or database) of structures from which new structures are created using genetic operators such as crossover and mutation. Each structure in the population has an associated *fitness* (goal-oriented evaluation), and these scores are used in a *competition* to determine which structures are used to form new ones.

There is a large body of both theoretical and empirical evidence showing that, even for very large and complex search spaces, GAs can rapidly locate structures with high fitness ratings using a database of 50–100 structures. Figure 1 gives an abstract example of how the fitness of individuals in a population improves over time. Readers interested in a more detailed discussion of GAs should see Holland (1975), De Jong (1980), and Grefenstette (1986).

The purpose of this paper is to understand when and how GAs can lead to goal-directed structural changes in learning systems. We are now in a position to make some general observations, which we will explore in more detail in subsequent sections.

The key feature of GAs is their ability to exploit accumulating information about an initially unknown search space in order to bias subsequent search into useful subspaces. Clearly, if one has a strong domain theory to guide the process of structural change, one would be foolish not to use it. However, for many practical domains of application, it is very difficult to construct such theories. If the space of legal structural changes is not too large, one can usually develop an enumerative search strategy with appropriate heuristic cutoffs to keep the computation time under control. If the search space is large, however, a good deal of time and effort can be spent in developing domain-specific heuristics with sufficient cutoff power. It is precisely in these circumstances (large, complex, poorly understood search spaces) that one should consider exploiting the power of genetic algorithms.

At the same time, one must understand the price to be paid for searching poorly understood spaces. It typically requires 500–1000 samples before genetic algorithms have sufficient information to strongly bias subsequent samples into useful subspaces. This means that GAs will not be appropriate search procedures for learning domains in which the evaluation of 500–1000 alternative structural changes is infeasible. The variety of current activity in using GAs for machine learning suggests that many interesting learning problems

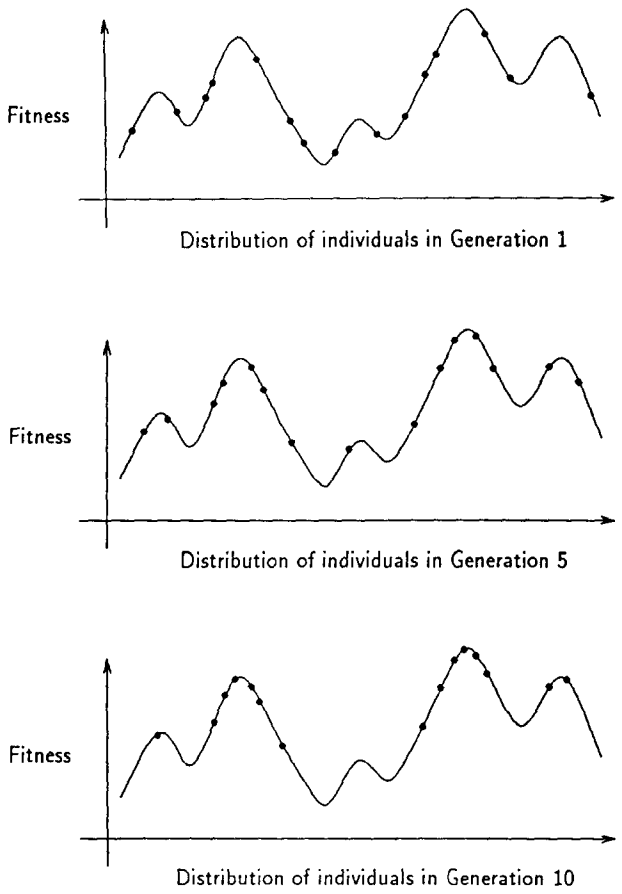

Figure 1. An abstract example of adaptive search using genetic algorithms.

fall into this category; i.e., involving large, complex, poorly understood search spaces in contexts that permit sampling rates sufficient to support GAs.

In discussing these activities, it will help to have a more concrete model of the architecture of a learning system that uses genetic algorithms. The simplest GA-based learning systems to describe are those whose goals are *performance oriented.* In this framework, the environment defines one or more tasks to be performed, and the learning problem involves both skill acquisition and skill refinement. It is generally useful to separate such systems (at least conceptually) into two subsystems as illustrated in Figure 2: a GA-based learning component charged with making appropriate structural changes, and a task component[1] whose performance-oriented behavior is to be improved.

[1] Within the machine learning literature, this is often called the *performance* component.

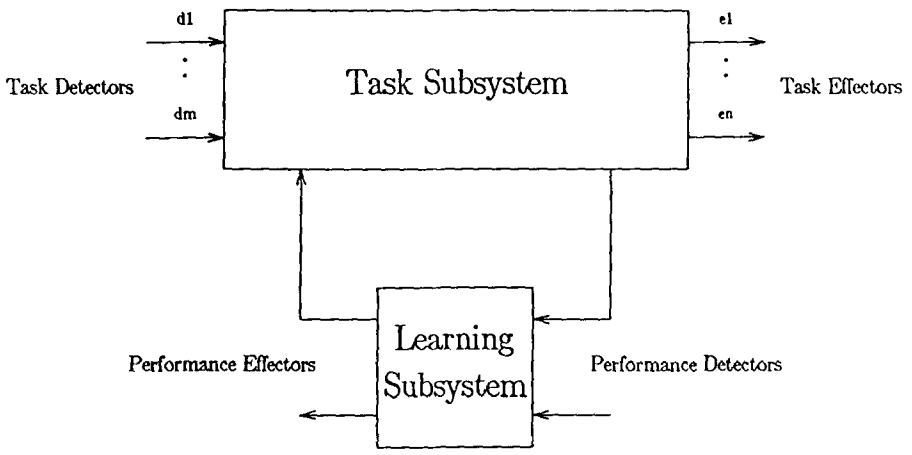

Figure 2. A performance-oriented learning system.

We are now in a position to describe how one might exploit the power of genetic algorithms in a learning system of the type depicted in Figure 2. The key idea is to define a space of *admissible* structures to be explored via GAs. Each point in this space represents the "genetic material" of a task subsystem in the sense that, when injected with this structure, its task performance is now well defined and can be measured. The learning strategy involves maintaining a population of tested structures and using GAs to generate new structures with better performance expectations.

In considering the kinds of structural changes that might be made to the task subsystem, there are a variety of approaches of increasing sophistication and complexity. The simplest and most straightforward approach is for the GAs to alter a set of parameters that control the behavior of a predeveloped, parameterized performance program. A second, more interesting, approach involves changing more complex data structures, such as "agendas," that control the behavior of the task subsystem. A third and even more intriguing approach involves changing the task program itself. The following sections explore each of these possibilities in more detail.

3. Using genetic algorithms to change parameters

A simple and intuitive approach to effecting behavioral changes in a performance system is to identify a key set of parameters that control the system's behavior, and to develop a strategy for changing those parameters' values to improve performance. The primary advantage of this approach 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 (1959, 1967) 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 research on neural net learning also falls into this category.

How can one use genetic algorithms to quickly and efficiently search for combinations of parameters that improve performance? The simplest and most intuitive approach views the parameters as genes and the genetic material of individuals as a fixed-length string of genes, one for each parameter. The crossover operator then generates new parameter combinations from existing good combinations in the current database (population) and mutation provides new parameter values.

There is considerable evidence, both experimental and theoretical, that GAs can home in on high-performance parameter combinations at a surprising rate (De Jong, 1975; Brindle, 1980; Grefenstette, 1986). Typically, even for large search spaces (e.g., 10^{30} points), acceptable combinations are found after only ten simulated generations. To be fair, however, there are several issues that can catch a GA practitioner off guard when attacking a particular problem in parameter modification.

The first issue involves the number of distinct values that genes (parameters) can take on. 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 the following dilemma. If the mutation rate is too low, there can be insufficient global sampling to prevent premature convergence to local peaks. However, significantly increasing the rate of mutation can lead to a form of random search that 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 obvious to the casual reader.

Holland (1975) provides an analysis of GAs which suggests that they are most effective when each gene takes on a small number of values, 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), one uses a binary string representation that represents parameters as *groups* of binary-valued genes. Although the two spaces are equivalent in that both represent the same parameter space, GAs perform significantly better on the binary representation. This effect occurs because, in addition to mutation, crossover now generates new parameter values each time it combines part of a parameter's bits from one parent with those of another.

The simplest way to illustrate this point is to imagine the extreme case of a domain in which one must adjust a single parameter that can take on 2^{30} distinct values. Representing this problem internally as a one-gene problem renders crossover useless and leaves mutation as the only mechanism for generating new individuals. However, a 30-gene binary representation lets crossover play 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 both "yes" and "no." Theoretically, every point in the search space has a nonzero probability of being sampled. However, for most problems of interest, the search space is so large that it is impractical to wait long enough for guaranteed global optimums. A better view is that GAs constitute powerful sampling heuristics that can rapidly find high-quality solutions in complex spaces.

In summary, one simple but effective approach is to restrict structural change to parameter modification and to use GAs to quickly locate useful combinations of parameter values. De Jong (1980) and Grefenstette (1986) provide more detailed examples of this approach.

4. Using genetic algorithms to change data structures

There are many problems for which the simple parameter modification 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 uses GAs to alter these key structures. For instance, such problems occur when the task system whose behavior is to be modified is designed with a top-level "agenda" control mechanism. Systems for traveling-salesman, bin-packing, and scheduling problems are frequently organized in this manner, as are systems driven by decision trees. In this context GAs must select data structures to be tested, evaluated, and subsequently used to fabricate better ones.

At first glance, this approach may not seem to introduce any difficulties for genetic algorithms, since it is usually not hard to "linearize" these data structures, map them into a string representation that a GA can manipulate, and then reverse the process to produce new data structures for evaluation. However, again there are some subtle issues, and the designer must be familiar with them in order to make effective use of GA-based learning systems.

As in the previous section on 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. One can easily invent internal string representations for agendas and other complex data structures, but for many of these schemes, almost every new structure produced by the standard crossover and mutation operators represents an *illegal* data structure!

An excellent example of this problem arises in using GAs to find good agendas (tours) for a traveling salesman who needs to visit N cities exactly once while minimizing the distance traveled. The most straightforward approach would internally represent a tour as N genes, with the value of each gene indicating the name of the next city to be visited. However, notice that GAs using the standard crossover and mutation operators will explore the space of all N-tuples of city names (most of which are illegal tours) when, in fact, it is the

space of all *permutations* of the N city names that is of interest. The obvious problem is that, as N increases, the space of permutations becomes a vanishingly small subset of the space of N-tuples, and the powerful GA sampling heuristic has been rendered impotent by a poor choice of representation.

Fortunately, sensitivity to this issue is usually sufficient to avoid it in one of several ways. One approach is to design an alternative representation of the same space for which the traditional genetic operators are appropriate. GA researchers have taken this approach on a variety of such problems, including the traveling-salesman problem (e.g., see Grefenstette, Gopal, Rosmaita, & Van Gucht, 1985).

An equally useful alternative is to select different genetic operators that 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, one can define representation-sensitive crossover and mutation operators to assure that offspring represent legal points in the solution space (e.g., see Goldberg & Lingle, 1985; Davis, 1985).

The key point 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 that can subsequently be combined with each other to improve performance. String representations are just one of many ways of achieving this goal.

5. Using genetic algorithms to change executable code

So far we have explored two approaches to using GAs to effect structural changes to task subsystems: (1) by changing critical parameter values, and (2) by changing key data structures. In this section we discuss a third possibility: effecting behavioral changes in a task subsystem by changing the task program itself. Although there is nothing fundamentally different between a task program that interprets an agenda data structure and one that executes a LISP program, generally the space of structural changes to executable code is considerably larger and very complex. In any case, there is a good deal of interest in systems that learn at this level, and the remainder of the paper will discuss how GAs can be used in such systems.

5.1 Choosing a programming language

Since our goal is to use genetic algorithms to evolve entire task programs, it is important to choose a task programming language that is well suited to manipulation by genetic operators. At first glance, this does not seem to be much of an issue, since programs written in conventional languages like FORTRAN and PASCAL (or even less conventional ones like LISP and PROLOG) can be viewed 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 clear that this "natural" representation is disastrous

for traditional GAs, since standard operators like crossover and mutation produce few syntactically correct programs and even fewer that are semantically correct.

One alternative is to devise new language-specific genetic operators that preserve at least the syntactic (and hopefully, the semantic) integrity of the programs being manipulated. Unfortunately, the syntactic and semantic complexity of traditional languages makes it difficult to develop such operators. An obvious next step would be to focus on less traditional languages with simpler syntax and semantics (e.g., "pure" LISP), thus having the potential for reasonable genetic operators with the required properties. There have been a number of activities in this area (e.g., see Fujiki & Dickinson, 1987).

However, pure LISP shares an important feature with more traditional languages: it is procedural in nature, and procedural representations have properties that cause difficulty for GA applications. One obvious problem involves order dependencies; interchanging two lines of code can render a program meaningless. Another is the occurrence of context-sensitive interpretations; minor changes to a section of code, such as the insertion or deletion of a punctuation symbol, can change the entire meaning of the succeeding code. De Jong (1985) presents a more detailed discussion of these representation problems.

These representational issues are not new. Holland (1975) anticipated them and proposed a family of languages (called *broadcast* languages) that were designed to overcome the problems described above. It is now clear that broadcast languages are a subset of a more general class of languages known as *production systems* (Newell, 1973; Neches, Langley, & Klahr, 1987). Production systems (PSs) continue to reassert their usefulness across a wide range of activities, from compiler design to expert systems; thus, a good deal of time and effort has gone into studying their use in evolving task programs with genetic algorithms.

5.2 Learning production-system programs

One reason that production systems have emerged as a favorite programming paradigm in both the expert system and machine learning communities is that they provide a representation of knowledge that 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 in performing a particular task (Buchanan & Mitchell, 1978; Hedrick, 1976). This is particularly true of data-driven PSs such as OPS5 (Forgy, 1981), in which the production rules making up a program are treated as an unordered set of rules whose left-hand sides independently and in parallel monitor changes in the environment.

It should be obvious that this same programming paradigm offers significant advantages for GA applications. In fact, it has precisely the same characteristics as Holland's early broadcast languages. As a consequence, we will focus on PSs whose programs consist of unordered rules, and describe how GAs can be used to search the space of PS programs for useful rule sets.

To anyone who has read Holland (1975), a natural way to proceed is to represent an entire rule set as a string (an 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 (e.g., see Smith, 1980, 1983), which gave rise to the phrase "the Pitt approach."

However, during the same time period, Holland developed a model of cognition (classifier systems) in which the members of the population are individual rules and a rule set is represented by the entire population (e.g., see Holland & Reitman, 1978; Booker, 1982). This quickly became known as "the Michigan approach" and initiated a friendly but provocative series of discussions concerning the strengths and weaknesses of the two approaches. Below we consider each framework in more detail.

5.2.1 The Pitt approach

If we adopt the view that each individual in a GA population represents an *entire* PS program, there are several issues that must be addressed. The first is the (by now familiar) choice of representation. The most immediate representation that comes to mind is to regard individual rules as genes and to view entire programs as strings of these genes. Crossover then serves to provide new combinations of rules and mutation provides new rules. However, notice 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 (Forgy, 1981) or MYCIN (Buchanan & Shortliffe, 1984) 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 an action appropriate for those conditions. Many of the successful classifier systems rely on this assumption (Wilson, 1985; Goldberg, 1985).

However, it is not difficult to relax this assumption and allow more flexible rule sets without subverting the power of the genetic operators. One can achieve this by making the operators "representation sensitive," in the sense that they no longer make arbitrary changes to linear bit strings. Rather, one extends the internal representation to provide punctuation marks so that only meaningful changes occur. For example, if the crossover operator chooses to

break one parent on a rule boundary, it also breaks the other parent on a rule boundary. Smith (1983) and Schaffer (1985) have used this approach successfully in their LS systems.

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

One can adopt the same view using the Pitt approach and require all rule sets (strings) to have the same fixed length. This can be justified in terms of the advantages of having redundant copies of rules and having workspace within a rule set for new experimental building blocks without necessarily having to replace existing ones. However, Smith (1980) has extended many of the formal results on genetic algorithms to variable-length strings. He complemented these results with a GA implementation that maintained a population of variable-length strings and that efficiently generated variable-length rule sets for a variety of tasks. One interesting contribution of this work was a method for keeping down the size of the rule sets, based on 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 nontrivial rule sets for such diverse tasks as solving maze problems, playing poker, and classifying gaits. We direct the interested reader to Smith (1983) and Schaffer (1985) for more details.

5.2.2 The Michigan approach

Holland and his colleagues developed a quite different approach to learning production-system programs 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 that are modified over time via interaction with the environment. Unlike genetic material, this kind of knowledge does not evolve over generations via selection and mating. Rather, it accumulates in real time as the individual struggles to cope with his 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 genetic algorithms.

Classifier systems consist of a set of rules (*classifiers*) that manipulate an internal message list. The left-hand side of each classifier consists of a pattern that matches messages on the message list. The right-hand side of each classifier specifies a message to be posted on the message list if that classifier is allowed to fire. Interaction with the environment is achieved via a task-specific set of detectors that post detector messages on the message list, along with a set of task-specific effectors that generate external actions in response to posted messages. A classifier system is "perturbed" by the arrival of one or more detector messages indicating a change in the environment. This results in a sequence of rule firings as the contents of the message list changes, and it may result in one or more responses in the form of effector actions.

Learning in classifier systems is achieved by requiring that the environment provide intelligent feedback to the classifier system in the form of reward (punishment) whenever favorable (unfavorable) states are reached. Since an arbitrary number of rules can fire during the interval between two successive payoffs, a significant credit assignment problem arises in determining how payoff should be distributed. Holland (1986) has developed a "bucket brigade" mechanism for solving this problem. Based on a strong "service economy" metaphor, the bucket brigade distributes payoff (wealth) to those rules actively involved in sequences that result in rewards. Over time, wealthier rules become more likely to fire, since they are favored by the conflict-resolution mechanism.

As described, classifier systems are able to select useful subsets of rules from an existing rule set. However, additional behavioral improvements can be obtained by making changes to the rules as well. As the reader may have guessed, this is achieved by interpreting the wealth of individual rules as a measure of "fitness," and using genetic algorithms to select, recombine, and replace rules on the basis of their fitness.

There are a number of impressive examples of classifier systems that regulate gas flow through pipelines (Goldberg, 1985), control vision systems (Wilson, 1985), and infer Boolean functions (Wilson, 1987). Which approach is better, the Pitt or Michigan approach, in the sense of being more effective in evolving task programs? It is too early to answer this question or even to determine if the question is valid. 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 more useful for off-line environments in which more leisurely exploration and more radical behavioral changes are acceptable.

5.3 Architectural issues for production systems

So far we have focused 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 is that GAs are much more effective on PS programs that consist of unordered rules. In this section we summarize some additional implications that the use of GAs might have on the design of PS architectures.

5.3.1 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 syntactically complex and can raise many of the representational issues discussed earlier. In particular, variable-length expressions, varying types of operators and operands, and function invocations make it difficult to choose a representation and/or a set of genetic operators that produce useful offspring easily and efficiently.

Languages like OPS5 and SNOBOL take an alternative approach, assuming the left-hand side is a pattern to be matched. Unfortunately, the pattern language can be as complex as Boolean expressions and in some cases is even

more complex, due to the additional need to save matched objects for later use in the pattern or in the right-hand side.

Consequently, 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, many implementations have followed Holland's lead and have chosen the simple {0, 1, #} fixed-length pattern language, permitting a direct application of traditional genetic operators, which were designed to manipulate fixed-length binary strings. When 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 require complex and creative representations of the objects to be matched. Simple relationships like "*speed* > 200" may require multiple rule firings and the use of internal memory to ensure correct evaluation. As discussed earlier, some of this rigidity can be alleviated by the use of context-sensitive genetic operators (Smith 1983). However, finding a better compromise between simplicity and expressive power of the left-hand sides is an active area of research.

A favorite psychological motivation for preferring pattern matching rather than Boolean expressions is the intuition that humans use the powerful mechanism of partial matching to deal with the enormous variety of every day life. Seldom are humans in precisely the same situation twice, but they manage to function reasonably well by noting the current situation's similarity to previous experience.

This has led to interesting discussions as to how GAs might capture similarity 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 wild-card symbol "#" plays as patterns evolve to their appropriate level of generality. Booker (1982, 1985) and others have suggested that requiring perfect matches *even* with the {0, 1, #} pattern language is still too 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 score indicating how close the pattern came to matching. This is an important issue, and we need more work on methods for computing match scores in a reasonably general but computationally efficient manner. We direct the interested reader to Booker (1985) for more details.

5.3.2 Working memory

Another PS architectural issue revolves around the decision about 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 OPS model, in which rules can also attend to elements in an internal working memory and make changes to that memory.

Arguments in favor of the latter approach observe 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 a working memory.

Arguments against including working memory generally fall along three lines: (1) the application does not need the additional generality and complexity; (2) concerns about bounding the number of internal actions before generating the next external action (i.e., the halting problem); or (3) the fact that most of the more traditional concept-learning work (e.g., Winston, 1975; Michalski, 1983) has focused on stimulus-response approaches.

Most GA implementations of working memory provide a restricted form of internal memory, namely, a fixed-format, bounded-capacity message list (Holland & Reitman, 1978; Booker, 1982). However, it is clear that there are many uses for both classes of architecture. The important point here is that this choice is not imposed by GAs themselves.

5.3.3 Parallelism in production systems

Another side benefit of PSs with working memory is that they can be easily extended to allow parallel rule firings (Thibadeau, Just, & Carpenter, 1982; Rosenbloom & Newell, 1987). In principle, the only time that serialization must occur is when an external effector is activated. Hence, permitting parallel firing of rules that invoke internal actions is a natural way to extend PS architectures in order to exploit the power of parallelism. Of course, the implementor must decide whether this power is appropriate for a particular application. What should be clear is that GAs can be applied equally well to parallel PS architectures, leaving the choice to the designer.

5.4 The role of feedback

In attempting to understand how GAs can be used to learn PS programs, we have discussed how such programs can be represented and what kinds of 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 one can view GAs as using an adaptive sampling strategy to search large, complex spaces. This sampling scheme 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 if one has chosen a good representation and has 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 on this dimension, it is important that he select a feedback mechanism that facilitates 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 employ this information (almost) directly to bias the selection of parents used to produce new samples (offspring). Of course, not all payoff functions are equally suited for this role. A good function will provide useful information early in the search process to help focus attention. For example, a payoff function that is nearly always zero provides almost no information for directing the search process.

The Michigan and Pitt approaches differ somewhat in the way they obtain payoff. 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, a good feedback mechanism will provide a relatively steady flow of payoff, rather than having long "dry spells." Wilson's (1985) "animat" environment is an excellent example of this style of payoff.

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

We still have much to learn about the role of feedback, from both an analytical and an empirical point of view. Bethke (1980) has used Walsh transforms in formally analyzing the types of feedback information that are best suited for GA-style adaptive search. Recent experimental work by Grefenstette (1988) suggests one way to combine aspects of both the Michigan and Pitt approaches, employing a multilevel credit assignment strategy that assigns payoff to both rule sets and individual rules. This is an interesting idea that promises to generate a good deal of discussion, and it merits further attention.

5.5 The use of domain knowledge

Genetic algorithms are conventionally viewed as domain-independent search methods in that they can be applied with no knowledge of the space being searched. However, although no domain knowledge is required, there are ample opportunities to exploit domain knowledge if it is available. We have already seen some examples of how domain knowledge can be incorporated. A designer must 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. The choice of genetic operators is closely related to representation decisions, and a significant domain knowledge can also enter into their selection. Grefenstette et al. (1985) provide an excellent discussion of these issues.

A more direct example of domain knowledge involves the choice of the initial population used to start the search process. Although we have described the initial population as randomly selected, there is no reason to start with an empty slate if one has *a priori* information available that permits seeding the initial population with individuals known to have certain performance levels.

A third and more obvious way to exploit domain knowledge is by means of the feedback mechanism. As we have seen, the effectiveness of GAs depends on the usefulness of the feedback information provided. Even the simplest form of feedback (the payoff-only method) can and frequently does incorporate domain

knowledge into 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. Thus, in practice we see a variety of scenarios, ranging from the use of "vanilla" GAs with little or no domain-specific modifications to highly creative applications that incorporate a good deal of domain knowledge.

6. Summary and conclusions

We started this paper with the goal of understanding how genetic algorithms might be applied to machine learning problems. We suggested that a good way to answer this question was to visualize a system as consisting of two components: a task subsystem whose behavior is to be modified over time via learning, and a learning subsystem responsible for observing the task subsystem over time and effecting the desired behavioral changes. This perspective let us focus on the kinds of *structural* changes a learning subsystem might make to a task subsystem in order to effect *behavioral* changes. We identified three classes of structural changes of increasing complexity: parameter modification, data structure manipulation, and changes to executable code.

Having characterized learning in this way, we restated the problem in terms of searching the space of legal structural changes for instances that achieve the desired behavioral changes. If one is working in a domain 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 the learning algorithm to discover (infer) the important characteristics of the search space *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 effectively search spaces from each of the categories listed above.

At the same time, it is important to understand the limitations of this approach. We have seen that in most cases 500-1000 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. We have also seen that the difficulty of choosing a good internal representation for the space increases with the complexity of the search space. Similarly, care must be taken to provide an effective feedback mechanism.

Thus, genetic algorithms are best viewed as another tool for the designer of learning systems. Like the more familiar inductive techniques and explanation-based methods, GA is not the answer to all learning problems, but it provides an effective strategy for specific types of situations.

Acknowledgements

I would like to thank the editors of this issue, David Goldberg and John Holland, for useful comments on an earlier draft of the paper.

References

Bethke, A. (1980). *Genetic algorithms as function optimizers.* Doctoral dissertation, Department of Computer and Communications Sciences, University of Michigan, Ann Arbor.

Booker, L. B. (1982). *Intelligent behavior as an adaptation to the task environment.* Doctoral dissertation, Department of Computer and Communications Sciences, University of Michigan, Ann Arbor.

Booker, L. B. (1985). Improving the performance of genetic algorithms in classifier systems. *Proceedings of the First International Conference on Genetic Algorithms and Their Applications* (pp. 80–92). Pittsburgh, PA: Lawrence Erlbaum.

Brindle, A. (1980). *Genetic algorithms for function optimization.* Doctoral dissertation, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

Buchanan, B. G., & Mitchell, T. M. (1978). Model-directed learning of production rules. In D. Waterman and F. Hayes-Roth (Eds.), *Pattern-directed inference systems.* New York: Academic Press.

Buchanan, B. G., & Shortliffe, E. A. (Eds.). (1984). *Rule-based expert systems.* Reading, MA: Addison-Wesley.

Davis, L. (1985). Job shop scheduling with genetic algorithms. *Proceedings of the First International Conference on Genetic Algorithms and Their Applications* (pp. 136–140). Pittsburgh, PA: Lawrence Erlbaum.

De Jong, K. A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* Doctoral dissertation, Department of Computer and Communications Sciences, University of Michigan, Ann Arbor.

De Jong, K. (1980). Adaptive system design: A genetic approach. *IEEE Transactions on Systems, Man, and Cybernetics, 10,* 556–574.

De Jong, K. (1985). Genetic algorithms: A 10 year perspective. *Proceedings of the First International Conference on Genetic Algorithms and Their Applications* (pp. 169–177). Pittsburgh, PA: Lawrence Erlbaum.

Fisher, D. H. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine Learning, 2,* 139–172.

Forgy, C. L. (1981). OPS5 *user's manual* (Technical Report CMU-CS-81-135). Pittsburgh, PA: Carnegie Mellon University, Department of Computer Science.

Fujiki, C., & Dickinson, J. (1987). Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma. *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (pp. 236–240). Cambridge, MA: Lawrence Erlbaum.

Goldberg, D. E. (1985). Genetic algorithms and rule learning in dynamic system control. *Proceedings of the First International Conference on Genetic Algorithms and Their Applications* (pp. 8–15). Pittsburgh, PA: Lawrence Erlbaum.

Goldberg, D. E., & Lingle, R. (1985). Alleles, loci, and the traveling salesman problem. *Proceedings of the First International Conference on Genetic Algorithms and Their Applications* (pp. 154–159). Pittsburgh, PA: Lawrence Erlbaum.

Grefenstette, J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics, 16*, 122–128.

Grefenstette, J. J. (1988). Credit assignment in rule discovery systems based on genetic algorithms. *Machine Learning, 3*, 225–245.

Grefenstette, J., Gopal, R., Rosmaita, B., & Van Gucht, D. (1985). Genetic algorithms for the traveling salesman problem. *Proceedings of the First International Conference on Genetic Algorithms and their Applications* (pp. 160–168). Pittsburgh, PA: Lawrence Erlbaum.

Hedrick, C. L. (1976). Learning production systems from examples. *Artificial Intelligence, 7*, 21–49.

Holland, J. H. (1975). *Adaptation in natural and artificial systems.* Ann Arbor, MI: University of Michigan Press.

Holland J. H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). Los Altos, CA: Morgan Kaufmann.

Holland, J. H., & Reitman, J. S. (1978). Cognitive systems based on adaptive algorithms. In D. A. Waterman & F. Hayes-Roth (Eds.), *Pattern-directed inference systems.* New York: Academic Press.

Michalski, R. S. (1983). A theory and methodology of inductive learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* Los Altos, CA: Morgan Kaufmann.

Neches, R., Langley, P., & Klahr, D. (1987). Learning, development, and production systems. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development.* Cambridge, MA: MIT Press.

Newell, A. (1973). Production systems: Models of control structures. In W. G. Chase (Ed.), *Visual information processing.* New York: Academic Press.

Rosenbloom, P., & Newell, A. (1987). Learning by chunking: A production system model of practice. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development.* Cambridge, MA: MIT Press.

Samuel, A. L. (1959). Some studies of machine learning using the game of checkers. *IBM Journal of Research and Development, 3*, 210–229.

Samuel, A. L. (1967). Some studies of machine learning using the game of checkers, II - Recent progress. *IBM Journal of Research and Development, 11*, 601–617.

Schaffer, J. D. (1985). Multiple objective optimization with vector evaluated genetic algorithms. *Proceedings of the First International Conference on Genetic Algorithms and Their Applications* (pp. 93–100). Pittsburgh, PA: Lawrence Erlbaum.

Smith, S. F. (1980). *A learning system based on genetic adaptive algorithms.* Doctoral dissertation, Department of Computer Science, University of Pittsburgh, PA.

Smith, S. F. (1983). Flexible learning of problem solving heuristics through adaptive search. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (pp. 422–425). Karlsruhe, West Germany: Morgan Kaufmann.

Thibadeau, R., Just, M., & Carpenter, P. (1982). A model of the time course and content of reading. *Cognitive Science, 6*, 157–203.

Wilson, S. W. (1985). Knowledge growth in an artificial animal. *Proceedings of the First International Conference on Genetic Algorithms and Their Applications* (pp. 16–23). Pittsburgh, PA: Lawrence Erlbaum.

Wilson, S. W. (1987). Quasi-Darwinian learning in a classifier system. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 59–65). Irvine, CA: Morgan Kaufmann.

Winston, P. H. (1975). Learning structural descriptions from examples. In P. H. Winston (Ed.), *The psychology of computer vision.* New York: McGraw-Hill.