

On Using Genetic Algorithms to Search Program Spaces

Kenneth De Jong

Computer Science Department
George Mason University

1. Introduction

The variety and quality of the papers in this conference is strong evidence of the dramatic increase in interest in Genetic Algorithms (GAs) both as an object of study in their own right, and in terms of the applications to which they might be applied. At our last conference, I tried to summarize where we stood in terms understanding what these GAs were, what kinds of problems seemed well-suited for GA applications, and what the open research issues appeared to be. In this paper I will attempt the same sort of perspective for a particular class of problems which is receiving considerable attention today: using GAs to search *program spaces* to quickly locate programs which are capable of performing a task at an acceptable performance level.

It has been pointed out many times that, because it has been difficult in the past to find a forum in the existing journal/conference structure for reporting GA research, there is a fairly serious gap between the "common wisdom" held by members of the GA research community and the material available in the open literature. As a consequence, those who are new to the area find it difficult to ascertain who has been doing what and frequently get involved unnecessarily in rediscovering various aspects of undocumented "wisdom" regarding the implementation and application of GAs. This seems to be particularly true when attempting to use GAs to search program spaces. So, in the following sections I'll try to summarize what we know and what the open issues are for this particular use of GAs.

2. Conceptualizing the Problem

My favorite way of conceptualizing the problem of searching program spaces is to

clearly separate out the process which is executing a task program from the GAs being used to construct new (and hopefully better) task programs. If we make this separation, we can focus on two important and inter-related questions: what kind of changes to task programs are reasonable to attempt with GAs, and what kinds of task programming languages encourage (discourage) the use of GAs.

In considering what kinds of changes might be made to task programs, there appears to be a range of approaches of increasing sophistication and complexity. The simplest and most straight forward approach is to have GAs make changes to a set of parameters which control the behavior of a pre-developed, parameterized task program. The advantage to 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 classical 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 "searching program spaces". 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. So, one piece of common wisdom is that, if a particular GA application permits, formulating the task modification problem as a parameter optimization problem has significant advantages.

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 programs 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 and GAs are expected to construct agendas to be tested, evaluated, and subsequently used to fabricate better ones.

This approach at first glance may not seem to introduce any serious 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, experience has taught us otherwise. Representation issues now rear their ugly heads and we can easily slip into a situation in which GAs are rendered impotent by a poor internal representation of the space to be searched. One must be careful, for example, to either choose a string representation on which the traditional genetic operators like crossover and mutation make sense, or to invent new operators better suited to the representation and which also satisfy the fundamental schema theorems. Both of these alternatives can be difficult to achieve. My favorite example of this is the amount of time and effort that has been expended in finding a "good" way to use GAs to solve traveling salesman problems. I will not dwell on these representation issues here since they are discussed in more detail in a variety of papers both in this conference and the 1985 proceedings (see, for example, [Grefenstette85], [Goldberg85], and [Davis85]).

By now the reader is probably ready to reply that neither of the approaches just discussed "really" involves searching program spaces. Rather, the reader has in mind the use of GAs to derive task programs at the more fundamental level of manipulating the executable code itself. I'm not sure that there is anything fundamentally different between interpreting an agenda and executing a Pascal program. However, I do feel that by taking such a step we are introducing an additional level of complexity as far as GA applications are concerned, and that it is important to "look before we leap". Since there is good deal of interest in such applications, the remainder of the paper will discuss these issues.

3. Choosing a Programming Language

I believe that the best way to proceed is to focus on the second of the inter-related questions posed in the introduction: what kind of programming languages encourage (discourage) the use of GAs at this level. This approach should not be surprising since we are really just raising the representation issue in yet another form.

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. As a consequence it should be clear that traditional GAs are not particularly effective in searching spaces in which the payoff is zero almost everywhere!

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 including at least one paper in this conference [Fujiko87].

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 [DeJong85].

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 [Holland75]. 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 searching program spaces with GAs.

4. Searching PS Program Spaces

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, [Newell77], [Buchanan78], or [Hedrick76]). 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 seems to offer 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 [Holland75], 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, [Smith80, or [Smith83]]) and has been dubbed "the Pitt approach".

However, during that 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 (see, for example, [Holland78] and [Booker82]). 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.

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 relatively subservient role GAs play and the emergence of a "bucket brigade" to deal with apportionment of credit issues. A reasonable explanation for this shift is that Holland was focusing on developing models of cognition rather than developing adaptive search techniques for large, complex spaces.

In any case, both approaches have produced encouraging results in quite different contexts some of which are being presented at this conference. At the same time I think it is fair to say that there is still not enough experience to understand precisely the strengths, weaknesses, and tradeoffs involved in either of the approaches. 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.

There are also some recent developments which suggest that it might be possible to combine the two approaches in powerful and interesting ways [Grefenstette87].

5. PS Architecture Issues

Even though we have narrowed the scope of the discussion to the problem of searching PS program spaces, there are a number of PS

architectural issues which still need to be addressed regardless of which (if any) of the preceding approaches are taken. In the following sections I will attempt to summarize those aspects which seem to arise most frequently.

5.1. Rule Set Sizes

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. However, the "path of least resistance" for GA applications leads one to precisely that point. If one adopts the Michigan approach, all implementations that I am aware of assume a fixed population size (i.e., a fixed number of classifiers). Most classical implementations of GAs as adaptive search procedures assume populations of fixed length strings. So, following the Pitt approach, using one of these implementations would require representing rule sets as fixed length strings. This issue is usually resolved (in keeping with one's philosophical and pragmatic perspectives) in one of several ways.

In classifier systems fixing the size of the rule set is viewed more in terms of setting an upper bound on the amount of classifier memory available in a cognitive model. Since humans seem to have the same sort of limitation, setting it to a reasonable, but fixed size seems to be quite justifiable (assuming mechanisms like generalization and forgetting for dealing with limited memory capabilities).

One can adopt the same view using the Pitt approach and require all rule sets (strings) to be the same fixed length. However, 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.

Historically, the strongest motivation for fixed length representations was that all the schema theorems were developed in this context. However, Smith [Smith80] 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 need to provide 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.

In summary, although constraints on rule set sizes may seem at first to be too simplistic and restrictive for GA applications, this has not proven to be the case in the work to date and can be relaxed if necessary.

5.2. Rule Formats

A similar concern arises when developing the format of the rules which make up the rule sets. Requiring the rule used in an expert system shell to conform to a rigid format consisting of a fixed number of slots on the left and right hand sides would be viewed as artificial and unduly restrictive. However, most GA-based applications do just that. Unfortunately, unlike the case of fixed length rule sets, it's much more difficult to justify such a restriction except on rather pragmatic grounds.

Whether one is treating rules or entire rule sets as individuals, one generally needs to have genetic operators like crossover and mutation occur at a finer level of granularity than just at rule boundaries. By requiring rules to be in a rigid, fixed length format, the usual genetic operators can be applied without much concern about producing syntactically or semantically meaningless offspring. If we increase the flexibility of the format by, for example, allowing a variable length left hand side, one is generally forced to modify the genetic operators to be "sensitive" to punctuation marks so that well-formed offspring are produced. However, introducing new operators means that care must be taken in ascertaining that they have the characteristics required by the schema theorems.

Most implementations of classifier systems and the more classical GA systems that I am aware of assume a fairly rigid format for reasons of simplicity. The usual (informal) arguments that are made to support the thesis that this is not a severe restriction generally take the form of: 1) the application doesn't require it or 2) one can achieve the same computational power (as the more flexible formats) by using multiple rule frings together with internal "working memory".

5.3. Working Memory

Another PS architectural dilemma 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., [Michalski83]) has focused on stimulus-response models.

Most of the implementations that I am aware of provide a restricted form of internal memory, namely, a fixed format, bounded capacity message list. However, it's clear that there are plenty of uses for both architectures. I'm just happy that this choice is not imposed on us by GAs.

5.4. Pattern Matching

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 and/or a set of genetic operators in which useful offspring are easily and efficiently produced.

The alternate approach used in languages like OPS5 and SNOBOL is to assume the left hand side is a pattern to be matched. Unfortunately, the pattern language 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.

Consequently, the GA implementor faces a fairly serious dilemma regarding the style and complexity of the left hand side of production rules. Most implementations that I am aware of 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. 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 > 200" may require multiple rule firings and internal memory in order to be correctly evaluated.

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 [Booker85] for more details.

In any case we need to understand better what is involved in choosing a syntax and semantics for the left hand side of rules which balances the need for simplicity of representation and the need for expressive power. In my view this is one of the most difficult and open issues in using GAs to search program spaces, and the key to successful applications.

6. Payoff Functions

In addition to choosing an appropriate representation language, careful thought must be given to the characteristics of the payoff function used to provide feedback regarding the performance of task programs. Strategies for designing effective payoff functions tend to differ somewhat depending on whether one is working with classifier systems or using the Pitt approach to evolving useful task programs.

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 function 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 [Wilson85].

The situation is somewhat different in the Pitt approach in that the usual view of evaluation consists of injecting the program defined by a particular individual into a task processor 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 [Smith80] found it necessary 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 are usually combined into a single payoff, recent work by Schaffer [Schaffer85] suggests that it might be more effective to use a vector-valued payoff function in situations such as this. To my knowledge no one has explored this possibility.

However, one of the more provocative papers in this area at this conference suggests that there is an opportunity to have "the best of both worlds" with a multilevel credit

assignment strategy which assigns payoff to both rule sets as well as individual rules [Grefenstette87]. This is an interesting idea which, I suspect, will generate a good deal of discussion and merits further attention.

7. Selecting Genetic Operators

It has been pointed out many times that there is nothing sacred about the traditional operators defined and analyzed by Holland. What *is* important is that we have criteria set by the schema theorems that such operators should meet. It is particularly tempting when using GAs to search program spaces to introduce new operators to deal with the complexity of the representations. One need not feel reluctant or apologetic about doing so. *However*, if changes are made to existing operators or new ones are introduced, it is important to verify that they aren't overly disruptive of the process of distribution of trials according to payoff and that they encourage the formation of building blocks. Without these basic properties, one is bound to be disappointed in the performance of GAs in searching program spaces.

8. Conclusion

So where does all this leave us? I think that it is quite clear that, as designers, if we can keep down the complexity of program spaces by using parameterized procedures or data structures which are easy to represent and manipulate with GAs, we have a much better chance for a successful GA application. If, on the other hand, the situation calls for evolving task programs at a more fundamental level, it seems equally clear that production system languages are currently the best choice for GA applications. Whether one chooses the Pitt or the Michigan approach is still a matter of individual preference. Perhaps by the time we get together again no such choice will be necessary.

References

- [Booker82] Booker, L. B., "Intelligent Behavior as an Adaptation to the Task Environment", Doctoral Thesis, CCS Department, University of Michigan, 1982.

- [Booker85] Booker, L. B., "Improving the Performance of Genetic Algorithms in Classifier Systems", Proc. Int'l Conference on Genetic Algorithms and their Applications, July, 1985.
- [Buchanan78] Buchanan, B., Mitchell, T.M., "Model-Directed Learning of Production Rules", in *Pattern-Directed Inference Systems*, eds. Waterman and Hayes-Roth, Academic Press, 1978.
- [Davis85] Davis, L. D., "Job Shop Scheduling Using Genetic Algorithms", Proc. Int'l Conference on Genetic Algorithms and their Applications, July, 1985.
- [DeJong85] De Jong, K., "Genetic Algorithms: a 10 Year Perspective", Proc. Int'l Conference on Genetic Algorithms and their Applications, July, 1985.
- [Fujiko87] Fujiko, C. and Dickinson, J., "Using the Genetic Algorithm to Generate Lisp Code to Solve the Prisoner's dilemma", Proc. Int'l Conference on Genetic Algorithms and their Applications, July, 1987.
- [Goldberg85] Goldberg, D. and Lingle, R., "Alleles, Loci, and the Traveling Salesman Problem", Proc. Int'l Conference on Genetic Algorithms and their Applications, July, 1985.
- [Grefenstette85] Grefenstette, J. et al., "Genetic Algorithms for the Traveling Salesman Problem", Proc. Int'l Conference on Genetic Algorithms and their Applications, July, 1985.
- [Grefenstette87] Grefenstette, J., "Multilevel Credit Assignment in a Genetic Learning System", Proc. Int'l Conference on Genetic Algorithms and their Applications, July, 1987.
- [Hedrick76] Hedrick, C.L., "Learning Production Systems from Examples", *Artificial Intelligence*, Vol. 7, 1976.
- [Holland75] J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
- [Holland78] Holland, J.H., Reitman, J., "Cognitive Systems Based on Adaptive Algorithms", in *Pattern-Directed Inference Systems*, eds. Waterman and Hayes-Roth, Academic Press, 1978.
- [Michalski83] Michalski, R., "A Theory and Methodology of Inductive Learning", in *Machine Learning: An Artificial Intelligence Approach*, eds. Michalski, Carbonell, and Mitchell, Tioga Publishing, 1983.
- [Newell77] Newell, A., "Knowledge Representation Aspects of Production Systems", Proc. 5th IJCAI, 1977.
- [Schaffer85] Schaffer, D., "Multiple Objective Optimization with Vector Evaluated Genetic Algorithms", Proc. Int'l Conference on Genetic Algorithms and their Applications, July, 1985.
- [Smith80] Smith, S. F., "A Learning System Based on Genetic Adaptive Algorithms", Doctoral Thesis, Department of Computer Science, University of Pittsburgh, 1980.
- [Smith83] Smith, S. F., "Flexible Learning of Problem Solving Heuristics Through Adaptive Search", Proc. 8th IJCAI, August 1983.
- [Wilson85] Wilson, S., "Knowledge Growth in an Artificial Animal", Proc. Int'l Conference on Genetic Algorithms and their Applications, July, 1985.

**GENETIC ALGORITHMS
AND THEIR APPLICATIONS:
Proceedings of the
Second International Conference on
Genetic Algorithms**

**July 28-31, 1987
at the
Massachusetts Institute of Technology
Cambridge, MA**

Sponsored By

American Association for Artificial Intelligence

Naval Research Laboratory

Bolt Beranek and Newman, Inc.

John J. Grefenstette
Naval Research Laboratory
Editor



LAWRENCE ERLBAUM ASSOCIATES, PUBLISHERS
1987 Hillsdale, New Jersey Hove and London