Maintaining Diversity in Genetic Search

Michael L. Mauldin
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Genetic adaptive algorithms provide an efficient way to search large function spaces, and are increasingly being used in learning systems. One problem plaguing genetic learning algorithms is *premature convergence*, or convergence of the pool of active structures to a sub-optimal point in the space being searched. An improvement to the standard genetic adaptive algorithm is presented which guarantees diversity of the gene pool throughout the search. Maintaining genetic diversity is shown to improve off-line (or best) performance of these algorithms at the expense of poorer on-line (or average) performance, and to retard or prevent premature convergence.

1. Introduction

Genetic adaptive algorithms (GA's) are one solution to the *blackbox* learning problem — given a domain of input structures and a procedure for determining the value of an objective function on those structures, find a structure which maximizes (or minimizes) the objective function.

GA's are based on the observation that natural systems of evolving species are very efficient at adapting to changing environments. By simulating evolutionary processes, GA's can harness the power of population genetics to provide autonomous learning components for artificial systems. Genetic algorithms have been applied to widely varying problems in learning and adaptive control such as character recognition [6], state space learning [11], pattern tracking [10], discovery [7], maze running and poker betting [12], and gas pipeline operation [5].

2. Applicability of GA's

The most attractive feature of GA's is the flexibility of the technique. As long as there is an objective performance measure, genetic search through the function space will find better and better solutions. No initial knowledge of the domain is required, and as long as the objective function is not completely random, the underlying structure of the problem assures that GA's will outperform random search. Of course, some domains have objective functions which are amenable to more specialized and more efficient search techniques. For example, where the objective function is quadratic, special numerical analysis techniques can quickly find the optimum point in the space. If the function is differentiable, gradient search works equally fast. If the function is at least unimodal, hill climbing search is very effective.

In complicated domains, though, these specialized techniques break down quickly. The conditions for which GA's perform well are much less rigid, and empirical studies have shown that on complicated domains, GA's outperform both specialized and random searches [2]. Bethke's thesis characterized the set of functions which are genetically optimizable in terms of the Walsh transforms of the function, and shows that the coefficients involved can be estimated during the search to determine whether the function can be optimized genetically [1].

3. The Basic Genetic Algorithm

The following steps are common to all genetic adaptive algorithms. They are motivated by the study of population genetics, and most of the same intuitions and terms apply.

1. Choose a representation language for describing the possible behaviors of the organisms you wish to study, and then encode this language in strings of binary digits (some genetic algorithms use other alphabets, but bit strings or bit strings with DON'T CARE symbols are the most common internal representations). Each string represents one point in the function space being optimized.
2. Choose an objective (or payoff) function which assigns a scalar payoff to any particular bit string, using the mapping you chose in step 1. This can be the cost of a solution to an economic problem, the final score of a game playing program, or some other measure of performance. This score is usually called a *fitness* rating.
3. Generate an initial population of strings (often at random, but the system can be given *a priori* knowledge by including some individual strings already known to perform well).
4. Evaluate each string using the payoff function to assign it a non-negative fitness. Better strings receive higher fitness ratings (when using GA's to minimize an objective function, a transformation is applied to the result to derive an increasing function).
5. Repeatedly generate a new population. Select one or more parent strings from the population using weighted probabilities so that the chance of being selected as a parent is proportional to the fitness of the string. Then apply one or more *genetic operators* to generate one or more new strings. There are many possible operators, but the two basic ones are *crossover* and *mutation*.
6. Select an equal number of strings in the current population to "die" and replace these with the newly generated strings. Some GA's generate only one new string at a time; others generate a whole new population at each step.

7. Now evaluate each of the new strings to assign each of them a fitness value, and go back to step 5.

The best source of information about GA's is Holland's *Adaptation in Natural and Artificial Systems* [6]. Holland uses terms borrowed from Mendelian genetics to describe the process:

- Each position in the string is called a *gene*.
- The possible values of each gene are called *alleles*.
- A particular string is called a *genotype*.
- The population of strings also called the *gene pool*.
- The organism or behavior pattern specified by a genotype is called a *phenotype*.
- If the organism represented is a function with one or more inputs, these inputs are called *detectors*.

Each genotype represents a particular point in the function space being optimized, and the goal of the search is to find points in the space with the largest objective values (better performance). Although this formulation of genetic algorithms is very similar to earlier evolutionary models (*eg* [3], [4]), there is one subtle difference: the introduction of the crossover operator. Early programs were usually "random generation with preservation of the best." This corresponds to a GA where the only genetic operator used is mutation. But mutation does not take advantage of the knowledge already present in the gene pool.

The crossover operator mixes building blocks (sets of alleles) which have been generated during the course of the search; this approach exploits regularities in the environment to produce new genotypes which are more plausible than mutation alone would provide. Holland uses the term *schemata* to describe these building blocks, and he showed that each schema tends to increase or decrease its presence in the gene pool in proportion to its past performance [6]. Since this happens for each subset of the space simultaneously, there is an immense amount of implicit parallelism in the search for better genotypes [12].

4. Examples of GA's

Smith's maze and poker betting programs and Goldberg's gas pipeline program (mentioned in Section 1) all use sets of production rules as an internal representation. Production rules are encoded as bit strings; the left-hand side of each rule matches one or more input detectors, and the right-hand side emits a binary message which encodes the desired action. Holland's *classifier systems* [8] are also rule based, but the messages emitted by the right-hand side of each rule are fed back to a global message list. Messages on this list activate more production rules to give the system the ability to represent feed-back loops and state memory.

The poker betting problem was based on Waterman's work on draw poker betting [13]. For this problem, seven input detectors were specified: (1) the value of the hand, (2) the size of the pot, (3) the size of the last bet, (4) the likelihood that the opponent is bluffing, (5) the "pot odds," (6) the number of cards drawn by opponent, and (7) a measure of the opponent's playing style. The right-hand side of each rule can specify one of four actions: drop, call, bet low, or bet high. Smith's system learned enough about poker betting over the course of 4000 trials to generate bets in accordance with accepted poker axioms 82% of the time. By contrast, Waterman's system achieved 86% agreement only with the help of an additional decision matrix not available to the genetic system [12].

5. Function Optimization as Sandbox

Much work on genetic algorithms has focused on function optimization. By using various test functions as environments, the effects of domain features such as linearity, differentiability, continuity, modality, and dimensionality can be studied in isolation [2]. When optimizing functions which map points in $\Re^n$ to $\Re$, the following representation is commonly used: each point in the domain is an n-tuple of real numbers, each real number is represented as a fixed binary number, and the binary representations are concatenated together to form a bit string. The fitness of the string is the value of the function at the original point.

Two different performance measures are commonly used to analyze the effectiveness of function optimizers: *on-line* and *off-line* performance. On-line performance is simply the mean of all trials, while off-line performance is the mean of the best previous trial at each time (or trial) t. On-line performance is an appropriate measure for a task such as gambling or economics where learning must be done while performing the task at hand. Off-line performance only considers the best behavior of the system, and is more appropriate for systems which either train to solve a problem, or systems which have a model of the domain.

More formally, if $f_e(t)$ denotes the average value of trial t on functions f_i, and if the goal is to minimize each function, we have the following definitions:

$$\text{On-line performance} \quad x_e(t) = \frac{1}{t} \cdot \sum_{i=1}^{t} f_e(i)$$

$$\text{Off-line performance} \quad x_e^*(t) = \frac{1}{t} \cdot \sum_{i=1}^{t} f_e^*(i)$$

$$\text{Best so far} \quad f_e^*(i) = \min_{j=1,i} f_e(j)$$

6. Premature Convergence

In genetic search, the process converges when the elements of the gene pool are identical, or nearly so. Once this occurs, the crossover operator ceases to produce new individuals, and the algorithm allocates all of its trials in a very small subset of the space. Unfortunately, this often occurs before the true optimum has be found; this behavior is called *premature convergence*. The mutation operator provides a mechanism for reintroducing lost alleles, but it does so at the cost of slowing down the learning process. DeJong suggests adding a *crowding factor* which affects the replacement algorithm. Rather than merely replacing a single individual, select a small subset of the gene pool, and replace the string most similar to the newly generated string. This method has the advantage that it does not introduce wild mutations, but unfortunately it does not *guarantee* that alleles won't be lost, it merely reduces the probability of loss, delaying but not preventing premature convergence.

7. Diversity

The intuitive reason for premature convergence is that the individuals in the gene pool are too "alike." This realization suggests that one method for preventing this convergence is to assure that different members of the gene pool are different. Since each structure is represented as a

bit string, it suffices to check that whenever a new structure is added to the pool that it differs from from every other structure by at least one bit. If the new individual is identical to another member of the gene pool, randomly change one bit, and repeat until the result differs from every other member of the pool.

A more general method is to define a metric over the space of structures and assure at each point that the distance between any two structures is greater than some minimum distance. The most obvious metric is to use the Hamming distance between the bit strings representing each structure (*ie* the number of bits which do not match). So that a large uniqueness value does not preclude search in a small subspace at the end of the search, the uniqueness value of k bits is slowly decreased to one bit as the search proceeds. If the decrease is linear in the number of bits, we have the following equation for n trials:

$$\text{k bit decreasing uniq.} \quad \text{Hamming}(g_i, g_j) > \left\lceil \frac{k \cdot (n - t)}{n} \right\rceil$$

Thus at the start of the search the space is sampled over a relatively coarse "grid," and as the search progresses, the grid size is gradually reduced until adjacent points are considered. This process bears a striking similarity to *simulated annealing*, with the minimum distance being analogous to the decreasing *temperature* used during the annealing process. But unlike simulated annealing, genetic search with decreasing uniqueness retains the parallel flavor of genetic search, while simulated annealing is a fundamentally serial process.

8. Methodology

To evaluate the usefulness of uniqueness, a learning program was written which implemented five search algorithms: (1) standard genetic search with replacement of worst (Holland R_1), (2) k bit decreasing uniqueness, (3) DeJong's *crowding factor*, (4) random search, and (5) parallel hill climbing search. These last two algorithms were included as controls to verify the general utility of genetic algorithms. The hillclimbing search used was simple random mutation with preservation of best.

Since search is a stochastic process, each algorithm was run 10 times with 10 different random seeds. The initial population depended only on the random seed, not the specific algorithm; therefore for any one seed, every algorithm started with the same initial population. This reduced the chance that an unusual initial population distribution would favor one particular algorithm. Each of these 10 runs was for 5000 trials. The domain for the test was a set of five test functions used by DeJong in his study [2]. Complete descriptions of each function are given in [9]. This "environment" included functions which were continuous and discontinuous, convex and non-convex, unimodal and multimodal, quadratic and non-quadratic, of low, medium, and high dimensionality, and both with and without noise.

9. Results

Table 9-1 shows each algorithm's global performance, *ie* the sum of its scores on the five test functions. Since this was a minimization problem, smaller numbers indicate better performance. Figure 9-1 shows the "Best so far" curves for each algorithm. This is simply a graph of the best point in the space found at that point (this is the first derivative of the off-line performance curve).

Global Performance	On-line	Off-line	Best
Random Search:	239.336	7.972	5.761
Hill Climbing:	18.871	2.103	-2.810
Holland R1:	5.218	1.227	-0.207
Crowding Factor 4:	3.886	-0.577	-2.351
Uniqueness 1 bit:	16.938	-0.017	-2.481
Uniqueness 2 bits:	15.803	-0.140	-2.209
Uniqueness 4 bits:	17.755	-1.341	-3.646
Uniqueness 8 bits:	31.300	-2.152	-4.779
Uniqueness 12 bits:	50.447	-2.776	-5.267
Crowding 4 + Uniq 12:	9.447	-2.689	-5.046

Table 9-1: Global Performance

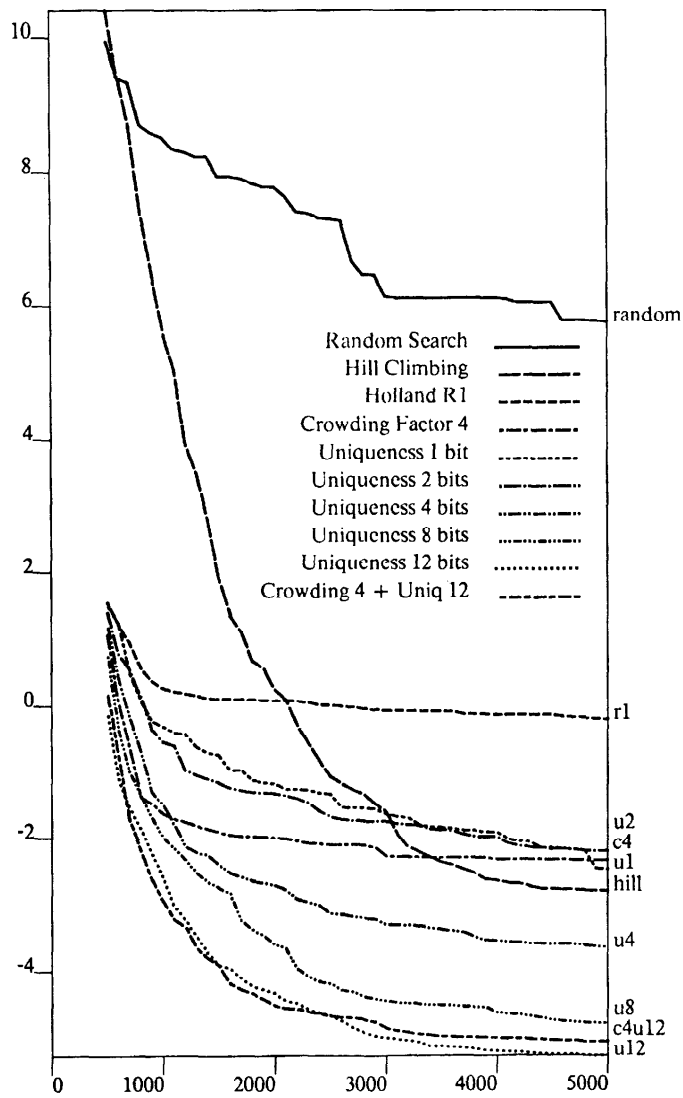

Figure 9-1: Global Best Found

The data in Table 9-1 clearly show that increasing the uniqueness parameter improves the off-line performance at the expense of poorer on-line performance. The only limit seems to be that the Hamming distance between two bit strings can be no greater than the length of the strings (so for a gene pool of size M and strings of k bits, the maximum uniqueness would be $k \cdot \log_2 M$).

Figure 9-1 shows that the the standard R_1 algorithm learns very quickly until about 1000 trials, and then the curve levels off. Adding DeJong's crowding factor improves performance significantly, but the curve (marked **c4**) still levels off after 2950 and no improvement is found thereafter. The graph for uniqueness of 12 bits (marked **u12**) has the best off-line performance of any algorithm, and is still improving at the end of 5000 trials.

One surprising result is that the combination of a crowding factor of 4 *and* a uniqueness of 12 bits performed almost as well off-line as uniqueness of 12 alone, and had a substantially improved on-line performance over simple uniqueness. What happens is this: using a crowding factor greater than 1 means that any new string is likely to be similar to the string it replaces. Since the string being replaced was unique, there is a high probability that the new string will also be unique. Thus fewer mutations are required to maintain diversity, and on-line performance is not as badly degraded.

10. Summary

This study confirms earlier work which demonstrated the robustness of genetic search as a tool for function optimization. It was shown that guaranteeing genetic diversity by means of a decreasing uniqueness measure provides significantly improved off-line performance at the expense of much poorer on-line performance. This degraded on-line performance can be ameliorated by combining DeJong's crowding factor with uniqueness to produce a genetic adaptive algorithm with superior off-line performance and moderate on-line performance.

One avenue for future research is to consider metrics other than Hamming distance for defining uniqueness. Another possible variation is to decode the bits strings into the corresponding real numbers and use Euclidean distance as a measure. This would tend to violate the black-box model of genetic learning, but could be viewed as a genetic heuristic search. Another possible improvement to uniqueness would be a mutation operator which is not uniform over the whole bit string. It might be that a mutation operator which always reintroduces a lost allele would provide another performance boost.

Another interesting prospect is the author's conjecture that a diverse gene pool would be helpful in optimizing time-varying functions. Pettit has studied the usefulness of genetic algorithms for tracking changing environments. She concluded that the standard genetic search performed very poorly in tracking even slowly changing environments [10]. One problem is obvious — if the gene pool ever converges (even at the correct optimum!) all future trials will be allocated at the same point, and the time-varying peak will simply "move out from under it." If, on the other hand, the gene pool is kept diverse, the crossover operator will continue to generate new strings, and should be much more able to track the peak.

11. Acknowledgments

I would like to thank Stephen Smith for his suggestions and insights into the world of genetic learning, and especially for access to his collection of hard-to-find literature on genetic algorithms.

References

[1] Bethke, A.D., *Genetic Algorithms as Function Optimizers*, PhD dissertation, University of Michigan, January 1981.

[2] DeJong, K.A., *Analysis of the Behavior of a Class of Genetic Adaptive Systems*, PhD dissertation, University of Michigan, August 1975.

[3] Fogel, L.J., Owens, A.J., and Walsh, M.J., *Artificial Intelligence Through Simulated Evolution*, Wiley, New York, 1966.

[4] Friedberg, R.M., "A Learning Machine, Part 1," *IBM Journal of Research and Development*, Vol. 2, 1958 .

[5] Goldberg, D.E., *Computer-Aided Gas Pipeline Operation Use Genetic Algorithms and Rule Learning*, PhD dissertation, University of Michigan, 1983.

[6] Holland, J.H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

[7] Holland, J.H., "Adaptive Algorithms for Discovery and Using General Patterns in Growing Knowledge Bases," *Intl. Journal of Policy Analysis and Info. Systems*, Vol. 4, No. 2, 1980 .

[8] Holland, J.H., "Escaping Brittleness," *Proceedings of the Second International Machine Learning Conference*, July 1983 .

[9] Mauldin, M.L., "Using Diversity to Improve Off-line Performance of Genetic Search," Tech. report, Computer Science Department, Cargnegie-Mellon University, 1984.

[10] Pettit, E. and Swigger, K.M., "An Analysis of Genetic-Based Pattern Tracking and Cognitive-Based Component Models of Adaptation," *Proceedings AAAI-83*, August 1983 .

[11] Rendell, L.A., "A Doubly Layered Genetic Penetrance Learning System," *Proceedings AAAI-83*, August 1983 .

[12] Smith, S.F., "Flexible Learning of Problem Solving Heuristics Through Adaptive Search," *Proceedings IJCAI-83*, August 1983 .

[13] Waterman, D.A., "Generalized Learning Techniques for Automating the Learning of Heuristics," *Artificial Intelligence*, Vol. 1, 1970 .