Active Systems Symposia GA Talk 1: Introduction to GAs

Paul A. Bromiley

Title

This is going to be the first of two talks on GAs, and will act as a simple introduction to the subject. Next week I will describe the work we have been doing on GA optimisation for the IRC project.

Overview

This presentation will start by describing the classic three operator GA, which is described in all the text books on the subject. The operators are reproduction, crossover and mutation. I will talk about how the operators work, and why we might expect the GA to optimise. I will then talk about the main problem with this sort of GA, the problem of premature convergence, which has led to much of the active research in the GA field, and describe some of the modifications people have introduced to try to avoid it. Finally, I will talk about multi-objective GAs, where the problem of premature convergence becomes a serious issue. That should set everything up for next weeks talk.

Optimisation Methods

The goal of optimisation is to find the maxima or minima of a function within some given range. There are two types of optima: local and global. Local optima are extremal points within some local neighbourhood, not on the boundary of that neighbourhood, whereas global optima are points which are truly the highest or lowest value the function can take. There are three main types of optimisation methods: calculus-based, enumerative and random. The aim in developing an optimisation method is to make it robust, by which we mean that it is both effective and efficient over a wide range of problem domains.

Calculus-Based Methods

Calculus-based methods assume that we can get meaningful derivatives of the function. With simple functions we can set the first derivative equal to zero and solve for the extremal points. Even if this is not possible, assuming we can get meaningful derivative values at any point, there are a wide range of calculus-based methods in the literature. They all boil down to more or less complicated hill-climbing: finding the local optimum by moving in the direction of steepest gradient. So, at this point we have two problems with calculus based methods: they require gradient information, which implies a smooth cost function, and they only find local optima, and so can't be considered robust.

Enumerative methods

Of course, it is possible to discretise the search space and calculate the cost at every point within some range, keeping a record of the best points. Schemes like this must be discounted since most search spaces are simply too large for them to be efficient, and the efficiency drops rapidly as the dimensionality of the search space grows.

Random methods

Truly random methods, such as random walks and random schemes that search and save the best result can't be expected to improve over enumerative search, but we have to separate these methods from randomised techniques. An example is simulated annealing, which attempts to simulate the process of annealing a crystalline material to achieve a single crystal structure. Consider a popular example: the Travelling Salesman Problem (TSP). Given a random arrangement of N cities, find the shortest path that visits each of the cities once and returns to the starting point. This problem is NP-complete, i.e. the computation time for an exact solution grows with N as exp(const. x N). The simulated annealing method starts with a random state, and then randomly changes portions of the state. We assign an energy to the state, which can simply be the total distance travelled. We want to minimise

this energy. Copying nature, we use the Boltzmann equation to describe the probability of the system moving to a new energy state, where p is the probability, E_1 and E_2 are the energies of the original and new state, T is the temperature and k is a constant. After each random change in the path, we calculate this probability. Notice that if the new path is shorter, the probability is one and we definitely move to it. However, even of the new path is longer we have a finite chance of moving to it. Therefore, the system can jump out of local minima. The temperature is slowly reduced until no further changes in the path can occur, and, given the right conditions, this will be the global optimum. Therefore, you can see that a randomised search is not necessarily a directionless search. The snag here is that the the right conditions include an annealing scheme that is problem specific, and thus requires experimentation.

Genetic Algorithms

So, we have many optimisation methods, but none of them meet the requirement for robustness: they are either too expensive, problem specific, or find only local minima. However, nature provides an example of an optimisation method that seems to simultaneously find all possible minima of a complex, multi-dimensional and dynamic cost function: the evolution of the species. Therefore, a class of optimisation methods have been developed that attempt to copy evolution, and these are known as genetic algorithms. The inspiration in GA research has therefore been twofold: to explain the processes of evolution through the abstraction of GAs, and to produce robust optimisation methods that simulate natural systems.

Genetic Algorithms: Outline

There are a huge range of variations of GAs, but I am going to start with the simplest possible example, using only three operators. Firstly, we code up the parameters of the optimisation problem we are interested in as a binary string. It will turn out that the way in which we do this coding is important, but ignore the subtleties for the moment. The strings are analogous to chromosomes, and the bits on the strings are analogous to genes. In the case of binary strings, each gene has two alleles, 0 and 1. We initialise a population of random strings: populations sizes are typically of the order of 100 individual strings.

Genetic Algorithms: Example

We can illustrate the process with a simple problem taken from Goldberg. Suppose we want to optimise x^2 over the range 0 to 31. We can code the parameter x as a simple five-bit binary string, from 0 at 00000 to 31 at 11111. We generate a random population of four strings, as shown here, then decode them and calculate their fitness.

Genetic Algorithms: Example 2

Then we apply three operators: reproduction, crossover and mutation. In the reproduction stage, we make a new population containing copies of the original strings. The probability of reproducing a string is proportional to its fitness, so fitter strings have more copies in the mating pool than less fit strings. The simplest way of implementing this is known as roulette wheel selection. Divide the fitness of each string in the population by the total fitness to create a proportion. Next, create a roulette wheel, where each string is allocated its proportion of the wheel. Spin the wheel to generate random choices of strings, weighted by their proportion of the fitness of the population, until enough strings have been chosen to fill the mating pool. The table here shows the strings we end up with in the mating pool. Notice that there are two copies of the fittest string, and none of the least fit string. Also note that roulette wheel selection is noisy: the number of copies of some string in the mating pool is never exactly equal to its proportional fitness.

Genetic Algorithms: Example 3

The next operator is crossover. We randomly pair off the strings to create mating pairs. Then, for each mating pair, we randomly select a position along the string. We then create new strings consisting of the first part of the first string with the second part of the second string, and vice-versa. Repeating with all strings, we create a new population.

The final operator is mutation. We randomly select genes in the population and flip their value, according to some mutation probability, which is typically kept quite low. This allows us to explore the properties of alleles of genes that were not present in the original population, and thus increases the coverage of the search space. Given a low probability of mutation, we do not mutate any of the genes in the example population.

Schemata

Now, look at the new population generated in the simple example. In only a single iteration, the average fitness of the population has increased from 576 to 729. What has happened? The best string in the initial population was 11000. Since it was fittest, it received two copies in the mating pool. One of these combined with the string 10011 to give 11011, which turns out to be very fit. Overall, a visual inspection shows that most of the increase in average fitness is coming from the propagation of strings containing 11—, and the best string comes from a combination of this pattern with —11.

Although it is risky to draw conclusions from a single trial of a stochastic process, this implies that the GA is working by combining fit sub-strings, and preferentially propagating the fittest, leading to the concept of schemata. A schema is a similarity template describing a set of strings with similarities at certain points. For the other points, we need an extra symbol in our genetic alphabet: use *. So, the schema 11*** describes all strings with 1's in the first two positions. Since we have an alphabet of cardinality 3 for describing schemata, there are 3^l possible schemata. Note that a binary string of length l contains 2^l possible schemata (each position may take on its actual value or the don't care symbol) so, depending on population diversity, the population of n individuals contains between 2^l and $n2^l$ schemata. A schema has two obvious properties: the number of bits it specifies, known as the order o(H), and the distance between the outlying bits, known as the defining length d(h). For example, the schema 1**0* has order 2 and defining length 3.

Schemata 2: Reproduction

Suppose there are m examples of the schema H in the population at time t, writing m=m(H, t). After an iteration of the GA, we will have m=m(H, t+1) examples of the same schema in the population. Since they are copied according to their fitness with probability $p_i = \frac{f_i}{\sum f_i}$, and putting f(H) as the average fitness of the strings containing H, we can write $m(H, t+1) = m(H, t).n.f(H)/\sum f_j$. Putting $\bar{f}$ as average fitness of the population $\sum f_j/n$, we have

$$m(H, t+1) = m(H, t)\frac{f(H)}{\bar{f}} \tag{1}$$

So, the number of any given schema in the population grows as the ratio of its average fitness to the average fitness of the population under the action of reproduction alone. Also note that this process is being carried out on every schema in the population in parallel during a single iteration of the algorithm. Assuming that a given schema retains average fitness a fixed constant c above the average of the population, we get

$$m(H, t_1) = m(H, t)\frac{\bar{f} + c\bar{f}}{\bar{f}} = (1 + c)m(H, t) \tag{2}$$

giving

$$m(H, t) = m(H, 0)(1 + c)^t \tag{3}$$

So, above average fitness schemata grow in frequency in the population exponentially, and below average schemata decrease in the same fashion.

Schemata 3: Crossover and Mutation

What about the other two operators? Crossover can clearly destroy some schemata, and the probability of this destruction will increase with the defining length, since there are more crossover points available to destroy it. If the defining length is d(H), there are d(H) crossover sites that can disrupt the schema, out of a possible (l-1) sites

in the string. To keep things general, suppose that crossover itself is performed with a probability p_c. Then, the probability of a schema surviving crossover is

$$p_s \leq 1 - p_c \frac{d(H)}{(l-1)} \tag{4}$$

This is only an upper bound, since the schema may survive crossover at a position between its specified positions if the other string contains the same values at some of the specified positions.

Finally, we have mutation, which is applied randomly with probability p_m to each bit in the population and flips the bit. The order of the schema o(H) tells us the number of specified bits, and so if each bit survives unchanged with probability $(1 - p_m)$, the probability of the whole schema surviving is

$$(1 - p_m)^{0(H)} \approx (1 - o(H).p_m) for p_m << 1 \tag{5}$$

remembering that we use low mutation probabilities.

Schemata 4: The Fundamental Theorem of Genetic Algorithms

Collecting terms, we get the Schema Theorem, also called the Fundamental Theorem of Genetic Algorithms

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\bar{f}} [1 - p_c \frac{d(H)}{l-1} - o(H) p_m] \tag{6}$$

This states that schema of low order and short defining length grow or decrease exponentially in the population in proportion to the ratio of their average fitness to the population average fitness. Schemata of low order and short defining length are given the name building blocks, and are recognised as the fundamental units being processed in a GA search.

Is this a good thing to do? It is usually justified using an example called the two-armed bandit problem. Given a two-armed slot machine where the arms pay off with reward μ_1 and μ_2, with standard deviations of σ_1 and σ_2 in the reward, where $\mu_1 > \mu_2$ but the labelling of the arms is not known, how do we divide N trials across the two arms to minimise losses. Holland (1975) calculated the optimal strategy, and determined that it involves allocating slightly higher than exponentially increasing trials to the observed best arm. A GA can be considered as a composition of many k-armed bandits. I am not going to go into this in any more detail.

How Many Schemata are Usefully Processed?

Given that the GA processes only low-order, short defining length schemata, how many are processed usefully? Ignoring mutation, and assuming simple crossover, we want to know how many schemata survive with probability greater than some constant p_s i.e. have error rate $\epsilon < (1 - p_s)$. Using the previous equation for p_s,

$$p_s = 1 - \frac{d(H)}{l-1} \tag{7}$$

we get

$$l_s < \epsilon(l-1) + 1 \tag{8}$$

where $d(H) = l_s - 1$

First count the number of schemata of length l_s or less. For a given length l_s anywhere in the string, there are $2^{(l_s-1)}$ possible schemata since we must hold one bit stationary to ensure the schemata match the string. There are then $(l - l_s + 1)$ positions in the string where the template can go, so we have

$$2^{(l_s-1)}(l - l_s + 1) \tag{9}$$

schemata of length l_s or less. If we pick a population size of $n = s^{l_s/2}$, we ensure that the are one or less schemata of order $l_s/2$ or more, and conclude that half are of higher order and half of lower order. If we take only the higher order ones, we set a lower bound on the number of schemata

$$n_s \geq n(l - l_s + 1)2^{(l_s - 2))} = \frac{(l - l_s + 1)n^3}{4} \tag{10}$$

So, the number of schemata usefully processed is of the order n^3, with a population size of n. The important point is that GAs process a large number of schemata in parallel whilst processing a much smaller number of strings. This property has been given the name implicit parallelism.

Schemata as Hyperplanes

As a final note on schemata, we can observe that a given schema encodes a hyperplane in the search space. Consider a unit cube. The schemata (*1*) encodes the plane forming the top surface of the cube. Therefore, we can consider the search as progressing independently along hyperplanes in the search space, combining the best results in each plane to generate global optima.

Codings

At this stage, we can reconsider the best way to code the parameters of the optimisation problem for GA search. We want to pick a coding that generates as many building blocks as possible, and allows the GA to manipulate them effectively. We can maximise the pool of building blocks by reducing the cardinality of the alphabet used, leading to the conclusion that a binary coding is the natural choice. To allow the GA to manipulate them effectively, they should be relevant to the problem and relatively independent.

Gray Codes

One alternative coding that has been used is Gray codes. The aim here is to impose a property of adjacency: examining the table, it is clear that adjacent integers differ by a single bit. Hollstien (1971) amongst others has claimed superiority for this type of coding. Thats all I'm going to say about codings, except to note that identification of the perfect coding in many cases would imply knowledge of the cost function that would make the optimisation redundant.

So, to summarise the first part of the talk, we know that a GA works by encoding the optimisation problem as a binary string, and repeatedly applying reproduction, crossover and mutation to a population of these strings. We know that this behaviour propagates building blocks (low-order, short defining length schemata) exponentially according to the ratio of their fitness compared to the population average fitness, and we know that of the order of n^3 of these building blocks are processed in parallel in a population of size n. The next question is: how well does this work?

Premature convergence

If this was the end of the story, then clearly there would not have been almost fifty years of GA research. Taking a simple example of $f(x) = x^10$, with parameters suggested from the literature, we get the following graph for the best-of-generation fitness and and average fitness. The GA converges quickly to solutions close to the optimum, but never reaches it. The population at this point is largely degenerate i.e. most genes are identical across all strings, implying that the GA has ceased to search the space. This is known as premature convergence, and can be a major problem. At the simplest level, its source is the reduction in the diversity in the population. We can always use the GA to indicate promising areas of the search space for simple hill-climbing algorithms such as simplex. However, loss of diversity is more of a problem on more complex cost functions, particularly if they are multimodal.

Sources of Diversity Loss

Research in the field has identified three main sources of loss of diversity: the variance of roulette wheel selection, selection pressure, and operator disruption. De Jong was the first researcher to identify noise in the selection process as a source of diversity loss. Roulette wheel selection is an inherently noisy process, as noted earlier. In

addition, we are estimating the fitness of any given schema from a finite population. These effects are termed selection noise. A second source of diversity loss is selection pressure: less fit schemata can be quickly eliminated from the population, even if they encode information concerning a global optimum. The third source of diversity loss is operator disruption: crossover and mutation can destroy schemata. Let's consider each of there effects in more detail.

Sources of Diversity Loss: Selection Noise

Roulette wheel selection is an inherently noisy process, with a high variance in the number of copies of a particular string added to the mating pool. Consider this test function, with two identically fit optima. Simple GAs will inevitably converge to one of the two optima, even though both are identically fit. The behaviour can be explained by a gambler's ruin argument: even if we find both optima, at each breeding step we are taking a gamble on destroying the strings representing one of the solutions. Eventually it will happen, and one of the global optima will be lost.

Sources of Diversity Loss: Selection Pressure

The second source of errors is selection pressure. Consider this function. We have a broad local minimum, and a narrow global minimum. Alleles coding for regions around the global minimum have less chance to be fit than those in the broad local minimum, as they have to get the answer nearly exactly right. Therefore, schemata encoding positions close to the global optimum will quickly be eliminated, and the search will collapse to the local optimum. We can conclude that the simple GA cannot maintain non-optimal solutions or sub-solutions, even when they may be components of a global optimum. Selection pressure will always be a problem where some solutions have fitness much higher than the population average, and such solutions are referred to as super-fit individuals.

Sources of Diversity Loss: Operator Disruption

The final source of errors is operator disruption. Consider this cost function, with two global optima that share no common bit positions. We can ignore mutation, as it is applied in small doses. However, crossover between the global optima will always generate less fit solutions. Again, eventually we are going to completely eliminate some fit schemata from the population.

Diversity preservation

So, it is becoming clear that we need mechanisms to preserve diversity in the population. A wide variety of alternatives have been studied, so I think the best thing I can do is run through some of the more common ones briefly. I will cover alternative selection schemes, fitness scaling, crowding, niching, speciation, and restricted mating.

Alternate selection schemes

Having identified the variance of roulette wheel selection as a major source of diversity loss, a number of researchers have investigated methods for reducing the variance of roulette wheel selection. One example is due to De Jong, and is known as the expected value model. The expected number of offspring for each string $f/\bar{f}$ is calculated. Thereafter, each time the string is selected for mating and crossover its offspring count is reduced by 0.5, until it reaches zero and was made unavailable for mating. This forces the number of offspring of a given string to be less than $f/\bar{f} + 1$. Although this scheme reduces performance on simple unimodal functions, it improves performance over more complex functions, and resulted in less allele loss.

Fitness scaling

Basic roulette wheel selection results in two problems. Close to the start of a run, a few randomly generated schemata will be much fitter than the population average, and so will multiply quickly until they dominate the

population. Close to the end, all remaining schemata will be of about average fitness, so will add equally to the mating pool, causing the GA to degrade into a random search. Although I have used the words fitness and cost interchangeably up to this point, they are not identical. We can choose a scaling of the cost function to produce the fitness values which attempts to avoid these behaviours, imposing appropriate levels of competition throughout the run. A simple linear scaling will do. We want to keep the average fitness the same, and force the maximum fitness to be some multiple of the average. For population sizes of the order of 50 to 100, c_{mult} of 1.2 to 2 have been found to be successful. That way, super-fit individuals have their fitness advantage limited at the start of the run, but fitness differences are emphasised at the end of the run. Other models have also been tried: sigma and power law scalings are both popular. These methods are reviewed in Forrest (1985).

Crowding and preselection

De Jong developed the method known as crowding in order to improve GA performance on multi-modal cost functions. Although we have been talking exclusively about GAs with non-overlapping populations, i.e. all the parents are replaced by a new generation in each iteration, this is not a requirement of the method. We can replace only a subset of the population, leaving some parents intact, specifying the size of this subset using a parameter called the generation gap. Such GAs are termed steady state: those which replace the whole population are called generational. We are then free to choose which parents get replaced, and many techniques have been developed that aim to preserve diversity through this replacement step.

De Jong's technique involved replacing one the most similar individuals, taken from a subset of the whole population specified by a crowding factor. Similarity here is defined by bitwise comparison in Hamming space. This has an analogue in nature: individuals compete for limited resources within any given evolutionary niche, and stronger individuals force out weaker ones. De Jong found that this scheme gave nearly optimal performance on his multimodal test function. However, other authors (Mahfoud, 1992) have found that stochastic errors in replacement still lead to a significant amount of genetic drift. De Jong's scheme was a generalisation of the idea of preselection developed five years earlier by Cavicchio. Here, good individuals replaced their own parents in the population.

Niching and Speciation

Crowding and preselection are both simple examples of more general techniques called niching and speciation. The general idea is to modify the GA from a single population to a set of interacting populations, in the same way that nature contains multiple species to exploit different evolutionary niches. By controlling the interaction between niches, multiple local optima can be preserved in the population without destroying each other.

Goldberg and Richardson (1987) have developed a scheme called fitness sharing, which simulates this competition for resources directly. The fitness of any given individual is scaled by the sum of the fitnesses of individuals in the neighbourhood, scaled by some linear weighting with distance. This limits the number of individuals that can congregate in a single neighbourhood, promoting the development of other niches.

Niching and Speciation 2

These results taken from Goldberg show the effect of sharing in a simple GA on a multimodal cost function. Without sharing, genetic drift causes the population to collapse to a single optima. With sharing, diversity is preserved and all optima of the cost function are maintained simultaneously in the population,

Mating Restriction

When the population reaches the point where multiple optima are simultaneously represented, any crossover between individuals on different optima will likely result in much less fit offspring. This has led to the development of mating restriction schemes, which aim to prevent such crossover events. Hollstein (1971) studied mating restriction schemes based on traditional methods in animal husbandry. He found that schemes which promote uniquely valuable individuals tend to improve performance on unimodal cost functions, whereas schemes which limit mating to like individuals tend to improve performance on multimodal cost functions. The scheme he settled on was called inbreeding with intermittent cross-breeding. Mating was restricted to related individuals as long as the family fitness continued to improve. If no improvement resulted, then cross-breeding of individuals from different families

was attempted.

So, you can see at this point that there is a vast field of research into alterations on the basic GA scheme, and that much of this research has been directed at diversity preservation. Many authors state that premature convergence is not a major problem, since the GA can always be used to identify promising areas of the search space for more convergent methods such as simplex. Also, for multimodal cost functions, multiple GA runs can be used to identify all of the global optima. However, when we attempt to optimise multi-objective cost functions, the problems get more serious.

Multi-objective optimisation

The GAs we have looked at so far all optimise a single fitness value, which is related to the underlying cost function in some way. However, the optimisation problem may contain multiple costs, and it may not be possible to combine them into a single number. Take a simple example again stolen from Goldberg. A widget manufacturer wishes to minimise both the expense and the time taken to produce widgets. There are five possible manufacturing methods, which result in the different values for expense and time taken. Plotting these on a graph, we can see that there are two type of points. Consider points E and D: there are other points, B and C respectively, which have both lower expenses and lower times. We call these solutions dominated. Now consider points A, B and C. For each of these, there is no other point which reduces both costs simultaneously. We call these non-dominated points. The set of all non-dominated points defines a surface in the multi-dimensional cost-function space called the Pareto Front. Now it is clear that, if we weight the individual objectives in some way and combine them into a single cost, we are finding a single point on the front. However, it is better to attempt to extract the entire front. Then the choice of which solution to use can be left to the widget manufacturer, based on other information.

GAs seem to be ideally suited to this type of optimisation. Since they use a population of individuals anyway, they can in theory find the whole front in a single run. However, this will only be true if diversity in the population can be preserved, otherwise all individuals will collapse to a single point on the front due to genetic drift.

Conclusions

That sets up everything nicely for next week's talk, so lets list some take-home messages. GAs work by coding the parameters of the optimisation problem as a string in some way, analogous to a chromosome. They then set up a randomly initialised population of these strings. In the simplest case, they then repeatedly apply three operators: reproduction, crossover and mutation, to generate a new population. Reproduction creates a mating pool of copies of these strings, with the number of copies proportional to the ratio of the string fitness to the average population fitness. Crossover mixes up portions of the strings in the mating pool, such that combinations of the best substrings are combined in an attempt to identify even fitter strings. Mutation is applied in small doses to prevent the population becoming degenerate.

GAs work by processing similarity templates called schemata that encode highly-fit sub-solutions. Only those schemata of low order and short defining length are processed effectively, but these schemata grow exponentially in the population. Highly fit, low order, short defining length schemata are called building blocks, and are the fundamental unit of GA search. In addition, of the order of n^3 schemata are processed in parallel for a population size of n, so much more information is available to the search than the fitness values of the population members, and we call this property implicit parallelism.

There are however, several sources of error in GA search: the variance of selection, selection pressure, and operator disruption. These errors result in genetic drift, the result of which is the population becoming degenerate. This can lead to premature convergence, and prevents multiple global optima being held in the population simultaneously. Whilst it can be argued that this is not a problem for single objective optimisation, it is a big problem in multi-objective optimisation, where we want to identify the Pareto front of all non-dominated solutions simultaneously. Much GA research focuses on new operators which prevent this collapse of the population diversity, and we have briefly looked at a few of them. Next week I will describe a multi-objective GA that we have developed on the IRC project, and which combines several of these adapted operators, together with a novel mating restriction, to prevent genetic drift.