ABSTRACT Genetic algorithms (GA) is an optimization technique for searching very large spaces that models the role of the genetic material in living organisms. A small population of individual exemplars can effectively search a large space because they contain *schemata*, useful substructures that can be potentially combined to make fitter individuals. Formal studies of competing schemata show that the best policy for replicating them is to increase them exponentially according to their relative fitness. This turns out to be the policy used by genetic algorithms. Fitness is determined by examining a large number of individual fitness cases. This process can be very efficient if the fitness cases also evolve by their own GAs.

1 Introduction

Network models, such as multilayered perceptrons, make local changes and so find local minima. To find more global minima a different kind of algorithm is called for. Genetic algorithms fill the bill. These algorithms also allow large changes in the state to be made easily. They have a number of distinct properties:

- They work with an encoding of the parameters.
- They search by means of a population of individuals.
- They use a *fitness function* that does not require the calculation of derivatives.
- They search probabilistically.

In broad outline the idea of a GA is to encode the problem in a string. For example, if the problem is to find the maximum of a scalar function $f(x)$, the string can be obtained by representing x in binary. A population represents a diverse set of strings that are different possible solutions to the problem. The fitness function scores each of these as to its optimality. In the example, the associated value of f for each binary string is its fitness. At each generation,

operators produce new individuals (strings) that are on average fitter. After a given set of generations, the fittest member in the population is chosen as the answer.

The two key parameters are the number of generations N_G and the population size N_p. If the population is too small, there will not be sufficient diversity in the strings to find the optimal string by a series of operations. If the number of generations is too small, there will not be enough chances to find the optimum. These two parameters are not independent. The larger the population, the smaller the number of generations needed to have a good chance of finding the optimum. Note that the optimum is not guaranteed; there is just a good chance of finding it. Certain problems will be very hard (technically they are called *deceptive* if the global minimum is hard to find). Note that for the solution to be not found, the probability of generating it from the population as it traverses the generations has to be small.

The structure of a GA uses a reproductive plan that works as shown in Algorithm II

The raw fitness score for the i^{th} individual, $h(i)$, is any way of assessing good solutions that you have. Of course the performance of the algorithm will be sensitive to the function that you pick. Rather than work with h, it is more useful to convert raw fitness to normalized fitness f, where

$$f(i) = \frac{h(i)}{\sum_{i=1}^{N_p} h(i)}$$

Since now $\sum_{i=1}^{N_p} f(i) = 1$, this method allows the fitness values to be used as probabilities.

Probabilistic operations enter the algorithm in three different phases. First, the initial population must be selected. This choice can be made randomly (or if you have some special knowledge of good starting points, these can be chosen). Next, members of the population have to be selected for reproduction. One way is to select individuals based on fitness, produce offspring, score the offspring, and then delete individuals from the resultant population

Algorithm I .1in The Genetic Algorithm

Choose a population size.
Choose the number of generations N_G.
Initialize the population.
Repeat the following for N_G generations:

1. Select a given number of pairs of individuals from the population probabilistically after assigning each structure a probability proportional to observed performance.
2. Copy the selected individual(s), then apply *operators* to them to produce new individual(s).
3. Select other individuals at random and replace them with the new individuals.
4. Observe and record the fitness of the new individuals.

Output the fittest individual as the answer.

based on $1 - f$. The third way probabilities enter into consideration is in the selection of the genetic operation to be used.

1.1 Genetic Operators

Now let's define the genetic operators. Let the symbol a denote a gene, and denote a chromosome by a sequence of genes $a_1 a_2 a_3 \ldots a_n$. The set of alleles $c_{i1}, \ldots, c_{ij_i}$ represents the possible codes that can be used at location i. Thus the gene will represent one of these

$$a_i \in c_{i1}, \ldots, c_{ij_i}$$

but to take a specific example, for an alphabet of the first three letters, the alleles would be $\{a, b, c\}$ at every position.

The first operator is *crossover*, wherein subsequences of two parent strings are interchanged. The point of the operator is to combine two good subsequences on the same string. To implement the operation, crossover points

Figure 1: Genetic operations on a three-letter alphabet of {a,b,c}. (*top*) Crossover swaps strings at a crossover point. (*middle*) Inversion reverses the order of letters in a substring. (*bottom*) Mutation changes a single element.

must be selected probabilistically. Then the substrings are swapped, as shown in Figure 1 (top).

Once good subsequences appear in an individual, it is advantageous to preserve them from being broken up by further crossover operations. Shorter subsequences have a better chance of surviving crossover. For that reason the *inversion* operation is useful for moving good sequences closer together. Inversion is defined in the middle section of Figure 1.

As the GA progresses, the diversity of the population may be removed as the individuals all take on the characteristics of an exemplar from a local mimimum. In this case *mutation* is a way to introduce diversity into the population and avoid local minima. Mutation works by choosing an element of the string at random and replacing it with another symbol from the code, as shown in the bottom section of Figure 1.

1.2 An Example

Consider finding the maximum of the function $f(x) = -x^2 + 12x + 300$ where x takes on integer values $0, \ldots, 31$. This example is easily solved with direct methods, but it will be encoded as a GA search problem to demonstrate the operations involved.

The first step in defining the GA is to code the search space. The encoding used for genetic operations is the binary code for the independent variable x. The value of $f(x)$ is the fitness value. Normalized fitness over the whole population determines the probability P_{select} of being selected for reproduction. Table 1 shows an initial condition for the algorithm starting with five individuals.

Now select two individuals for reproduction. Selections are made by accessing a random number generator, using the probabilities shown in the column

Table 1: Initial condition for the genetic algorithm example.

Individual	**Genetic Code**	x	$f(x)$	P_{select}
1	10110	22	80	0.08
2	10001	17	215	0.22
3	11000	24	12	0.01
4	00010	2	320	0.33
5	00111	7	335	0.36
Average			192	

Table 2: Mating process.

Mating Pair	**Site**	**New Individual**	$f(x)$	P_{select}
00010	2	10010	192	0.14
10001	2	00001	311	0.23

Table 3: The genetic algorithm example after one step.

Individual	**Genetic Code**	x	$f(x)$	P_{select}
1	10010	18	192	0.14
2	10001	17	215	0.16
3	00001	1	311	0.23
4	00010	2	320	0.23
5	00111	7	335	0.24
Average			275	

P_{select}. Suppose that individuals 2 and 4 are picked.

The operation we will use is crossover, which requires picking a locus on the string for the crossover site. This is also done using a random number generator. Suppose the result is two, counting the front as zero. The two new individuals that result are shown in Table 2 along with their fitness values. Now these new individuals have to be added to the population, maintaining population size. Thus it is necessary to select individuals for removal. Once again the random number generator is consulted. Suppose that individuals 1 and 3 lose this contest. The result of one iteration of the GA is shown in Table 3.

Note that after this operation the average fitness of the population has increased. You might be wondering why the best individuals are not selected at the expense of the worst. Why go to the trouble of using the fitness values? The reason can be appreciated by considering what would happen if it turned out that all the best individuals had their last bit set to 1. There would be no way to fix this situation by crossover. The solution has a 0 in the last bit position, and there would be no way to generate such an individual. In this case the small population would get stuck at a local minimum. Now you see why the low-fitness individuals are kept around: they are a source of diversity with which to cover the solution space.

2 Schemata

In very simple form, the example exhibits an essential property of the genetic coding: that individual loci in the code can confer fitness on the individual. Since the optimum is 6, all individuals with their leading bit set to 0 will be fit, regardless of the rest of the bits. In general, the extent to which loci can confer independent fitness will simplify the search. If the bits were completely independent, they could be tested individually and the problem would be very simple, so the difficulty of the problem is related to the extent to which the bits interact in the fitness calculation.

A way of getting a handle on the impact of sets of loci is the concept of *schemata* (singular: schema).[1] A schema denotes a subset of strings that have identical values at certain loci. The form of a schema is a template in which the common bits are indicated explicitly and a "don't care" symbol ($*$) is used to indicate the irrelevant part of the string (from the standpoint of the schema). For example, $1*101$ denotes the strings $\{10101, 11101\}$. Schemata contain information about the diversity of the population. For example, a population of n individuals using a binary genetic code of length l contains somewhere between 2^l and $n2^l$ schemata.

Not all schemata are created equal, because of the genetic operations, which tend to break up some schemata more than others. For example, $1****1$ is more vulnerable to crossover than $*11****$. In general, short schemata will be the most robust.

2.1 Schemata Theorem

To see the importance of schemata, let's track the number of representatives of a given schema in a population. It turns out that the growth of a particular schema in a population is very easy to determine. Let t be a variable that denotes a particular generation, and let $m(S,t)$ be the number of schema exemplars in a population at generation t. To simplify matters, ignore the effects of crossover in breaking up schema. Then the number of this schema in the new population is directly proportional to the chance of an individual being picked that has the schema. Considering the entire population, this is

$$m(S,t+1) = m(S,t)n\frac{f(S)}{\sum_i f_i}$$

because an individual is picked with probability $\frac{f(S)}{\sum_i f_i}$ and there are n picks. This equation can be written more succinctly as

$$m(S,t+1) = m(S,t)\frac{f(S)}{f_{ave}}$$

To see the effects of this equation more vividly, adopt the further simplifying assumption that $f(S)$ remains above the average fitness by a constant amount. That is, for some c, write

$$f(S) = f_{ave}(1 + c)$$

Then it is easy to show that

$$m(S, t) = m(S, 0)(1 + c)^t$$

In other words, for a fitness that is slightly above the mean, the number of schema instances will grow exponentially, whereas if the fitness is slightly below the average (c negative), the schema will decrease exponentially. This equation is just an approximation because it ignores things like new schema that can be created with the operators, but nonetheless it captures the main dynamics of schema growth.

2.2 The Bandit Problem

The upshot of the previous analysis has been to show that fit schemata propagate exponentially at a rate that is proportional to their relative fitness. Why is this a good thing to do? The answer can be developed in terms of a related problem, the two-armed bandit problem (Las Vegas slot machines are nicknamed "one-armed bandits"). The two-armed slot machine is constructed so that the arms have different payoffs. The problem for the gambler is to pick the arm with the higher payoff. If the arms had the same payoff on each pull the problem would be easy. Just pull each lever once and then pull the winner after that. The problem is that the arms pay a random amount, say with means m_1 and m_2 and corresponding variances σ_1 and σ_2.

This problem can be analyzed by choosing a fixed strategy for N trials. One such strategy is to pull both levers n times (where $2n < N$) and then pull the best for the remaining number of trials. The expected loss for this strategy is

$$L(N, n) = |m_1 - m_2|\{(N - n)p(n) + n[1 - p(n)]\} \quad (1)$$

where $p(n)$ is the probability that the arm that is actually worst looks the best after n trials. We can approximate $p(n)$ by

$$p(n) \approx \frac{e^{-x^2/2}}{\sqrt{2\pi}x}$$

where

$$x = \frac{m_1 - m_2}{\sqrt{\sigma_1^2 + \sigma_2^2}}\sqrt{n}$$

With quite a bit of work this equation can be differentiated with respect to n to find the optimal experiment size n^*. The net result is that the total number of trials grows at a rate that is greater than an exponential function of n^*. More refined analyses can be done, but they only confirm the basic result: Once you think you know the best lever, you should pull that lever an exponentially greater number of times. You only keep pulling the bad lever on the remote chance that you are wrong. This result generalizes to the k-armed bandit problem. Resources should be allocated among the k arms so that the best arms receive an exponentially increasing number of trials, in proportion to their estimated advantage.

In the light of this result let us return to the analysis of schemata. In particular, consider schema that compete with each other. For example, the following schemata all compete with each other:

$$**0*00*$$
$$**0*01*$$
$$**0*10*$$
$$**0*11*$$
$$**1*00*$$
$$**1*01*$$
$$**1*10*$$
$$**1*11*$$

Do you see the relationship between the bandit problem and schema? If these schemata act independently to confer fitness on the individuals that contain

them, then the number of each schema should be increased exponentially according to its relative fitness. But this is what the GA is doing!

Summary You should recognize that all of the foregoing discussion has not been a proof that GAs work, but merely an argument. The summary of the argument is as follows:

- GAs seem to work. In practice, they find solutions much faster (with higher probability) than would be expected from random search.
- If all the bits in the GA encoding were independent, it would be a simple matter to optimize over each one of the bits independently. This is not the case, but the belief is that for many problems, one can optimize over subsets of bits. In GAs these are schemata.
- Short schemata have a high probability of surviving the genetic operations.
- Focusing on short schemata that compete shows that, over the short run, the fittest are increasing at an exponential rate.
- This has been shown to be the right thing to do for the bandit problem, which optimizes reward for competing alternatives with probabilistic payoffs that have stable statistics.
- Ergo, if all of the assumptions hold (we cannot tell whether they do, but we suspect they do), GAs are optimal.

3 Determining Fitness

In the simple example used to illustrate the basic mechanisms of the GA, fitness could be calculated directly from the code. In general, though, it may depend on a number of *fitness cases* taken from the environment. For example, suppose the problem is to find a parity function over binary inputs of length four. All 2^4 inputs may have to be tested on each individual in the

population. In these kinds of problems, fitness is often usefully defined as the fraction of successful answers to the fitness cases. When the number of possible cases is small, they can be exhaustively tested, but as this number becomes large, the expense of the overall algorithm increases proportionately. This is the impetus for some kind of approximation method that would reduce the cost of the fitness calculation. The following sections describe two methods for controlling this cost.

3.1 Racing for Fitness

One of the simplest ways of economizing on the computation of fitness is to estimate fitness from a limited sample. Suppose that the fitness score is the mean of the fitnesses for each of the samples, or fitness cases. As these are evaluated, the estimate of the mean becomes more and more accurate. The fitness estimate is going to be used for the selection of members of the population for reproduction. It may well be that this fitness can be estimated to a useful level of accuracy long before the entire retinue of fitness cases has been evaluated.[2] The estimate of fitness is simply the average of the individual fitness cases tested so far. Assuming there are n of them, the sample mean is given by

$$\bar{f} = \frac{1}{n} \sum_{i=1}^{n} f_i$$

and the sample variance by

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (f_i - \bar{f})^2$$

Given that many independent estimates are being summed, make the assumption that the distribution of estimates of fitness is normal, that is, $N(\mu, \sigma)$. Then the marginal posterior distribution of the mean is a Student distribution with mean $\bar{f}$, variance s/n, and $n-1$ degrees of freedom. A Student distribution is the distribution of the random variable $t = \frac{(\bar{f}-\mu)\sqrt{n-1}}{s}$, which is a way of estimating the mean of a normal distribution that does not depend

on the variance.[3] The cumulative density function can be used to design a test for acceptance of the estimate. Once the sample mean is sufficiently close to the mean, testing fitness cases can be discontinued, and the estimate can be used in the reproduction phase.

Example: Testing Two Fitness Cases Suppose that we have the sample means and variances of the fitness cases for two different individuals after ten tests. These are $\bar{f}_1 = 21$, $\bar{f}_2 = 17$, and $s_1 = 3$. Let us stop testing if the two means are significantly different at the 5% level. To do so, choose an interval $(-a, a)$ such that

$$P(-a < t < a) = 0.95$$

Since $n - 1 = 9$, there are 9 degrees of freedom. A table for the Student distribution with nine degrees of freedom shows that

$$P(-2.26 < t < 2.26) = 0.95$$

Now for the test use the sample mean of the second distribution as the hypothetical mean of the first; that is,

$$t = \frac{(\bar{f}_1 - \bar{f}_2)\sqrt{n-1}}{s_1}$$

With $\bar{f}_1 = 21$, $\bar{f}_2 = 17$, and $s_1 = 3$,

$$t = 4$$

which is outside of the range of 95% probability. Thus we can conclude that the means are significantly different and stop the test.

3.2 Coevolution of Parasites

Up to this point the modeling effort has focused on a single species. But we know that "nature is red in tooth and claw"; that is, the competition between different species can improve the fitness on each. To demonstrate

Figure 2: An eight-element sorting network. The eight lines denote the eight possible inputs. A vertical bar between the lines means that the values on those lines are swapped at that point. The disposition of the particular input sample is tracked through the network by showing the results of the swaps.

this principle we will study the example of efficient sorting networks. This example was introduced by Hillis,[4] who also introduced new features to the basic genetic algorithm.

The idea of a sorting network is shown in Figure 2. Numbers appear at the input lines on the left. At each crossbar joining two lines, the numbers are compared and then swapped if the higher number is on the lower line (and otherwise left unchanged).

Recalling the basic result from complexity analysis that sorting takes at least $n \log n$ operations, you might think that the best 16-input network would contain at least 64 swaps.[5] This result holds in the limit of large n. For small 16-input networks, a solution has been found that does the job in 60 swaps. Hillis shows that a GA encoding can find a 65-swap network, but that introducing parasites results in a 61-swap network.

The encoding for the sorting problem distinguishes a model genotype from a phenotype. In the genotype a chromosome is a sequence of pairs of numbers. Each pair specifies the lines that are to be tested for swapping at that point in the network. A diploid encoding contains a second chromosome with a similar encoding.

To create a phenotype, the chromosome pair is used as follows. At each gene location if the alleles are the same—that is, have the same pairs of numbers—then only one pair is used. If they are different, then both are used (see Figure 3). The result is that heterozygous pairs result in larger sorting networks, whereas homozygous pairs result in shorter networks. The advantage of this encoding strategy is that the size of the network does not have to be explicitly included in the fitness function. If a sequence is useful, it is likely to appear in many genes and will thus automatically shorten the network during the creation of the phenotype.

Figure 3: The translation of genotype to phenotype used by Hillis demonstrated on a four-input sorting network. The genetic code contains a haploid representation with two possible swaps at each location. When the possible swaps are identical, only one is used in constructing the phenotype, resulting in a shorter network. In the example, the last pair result in two links, the second of which is redundant.

The fitness function is the percentage of the sequences that are sorted correctly. Experience shows that indexing the mating program with spatial locality is helpful. Rather than randomly selecting mates solely on fitness, each individual is assigned a location on a two-dimensional grid. The choice of mates is biased according to a Gaussian distribution, which is a function of grid distance.

Another nice property of the sorting networks is that just testing them with sequences of 0s and 1s is sufficient; if the network works for all permutations of 0s and 1s, it will work for any sequence of numbers.[6] Even so, it is still prohibitively expensive to use all 2^{16} test cases of 0s and 1s to evaluate fitness. Instead, a subset of cases is used. A problem with this approach is that the resultant networks tend to cluster around solutions that get most of the cases right but cannot handle a few difficult examples. This limitation is the motivation for parasites. Parasites represent sets of test cases. They evolve similarly, breeding by combining the test cases that they represent. They are rewarded for the reverse of the networks: the number of cases that the network gets wrong. The parasites keep the sorting-network population from getting stuck in a local minimum. If that outcome occurs, then the parasites will evolve test cases targeted expressly at this population. A further advantage of parasites is that testing is more efficient.

Notes

1. The exposition here follows David E. Goldberg's text *Genetic Algorithms in Search, Optimization, and Machine Learning* (Reading, MA: Addison-Wesley, 1989), which has much additional detail.

2. This idea has been proposed by A. W. Moore and M. S. Lee in "Effi-

cient Algorithms for Minimizing Cross Validation Error," *Proceedings of the 11th International Machine Learning Conference* (San Mateo, CA: Morgan Kaufmann, 1994). In their proposal they advocate using gradient search instead of the full-blown mechanics of a genetic algorithm, and show that it works well for simple problems. Of course, since gradient search is a local algorithm, its overall effectiveness will be a function of the structure of the search space.

3. H. D. Brunk, *An Introduction to Mathematical Statistics*, 3rd ed. (Lexington, MA: Xerox College, 1975).

4. Daniel W. Hillis, "Co-evolving Parasites Improve Simulated Evolution as an Optimization Procedure," in Christopher Langton et al., eds., *Artificial Life II*, SFI Studies in the Sciences of Complexity, vol. 10 (Reading, MA: Addison-Wesley, 1991).

5. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms* (Cambridge, MA: MIT Press, 1990).

6. Ibid.

ABSTRACT Genetic programming (GP) applies the genetic algorithm directly to programs. The generality of programs allows many different problems to be tackled with the same methodology. Experimental results show that even though GP is searching a vast space, it has a high probability of generating successful results. Extensions to the basic GP algorithm add subroutines as primitives. This has been shown to greatly increase the efficiency of the search.

4 Introduction

In genetic algorithms the problem is encoded as a string. This means that there is always a level of indirection whereby the meaning of the string has to be interpreted. A more direct way of solving a problem would be to have the string encode the *program* for solving the problem. Then the search strategy would be to have the genetic operations act on the programs themselves. The advantage of this approach would be that programs could rate themselves according to how well they are doing and rewrite themselves to do better. The difficulty is that, in general, making a change to a program will be disastrous to its function. To make it work, there must be some way of encoding changes that makes the genetic modifications less destructive. This is the goal of *genetic programming*, which works directly with programs encoded as trees instead of strings.

A beautiful idea is to use the LISP programming language.[1] This is a functional language based on lists. For example, the function call $f(x, y)$ is encoded simply as

```
(f x y)
```

which is a function followed by a list of arguments to that function. Running the program, or *evaluating the function*, results in the value of the function being computed. For example, evaluating the LISP list

```
(* 3 4)
```

results in a value of 12. Complex programs are expressed by using composite functions. For example,

```
(+ 2 (IF (> X 3) 4 7))
```

evaluates to 6 if X is greater than 3; otherwise it evaluates to 9. The list structure consists of *terminals* and *functions*. For instance, in the present example,

```
{x,2,3,4,7}
```

are terminals, whereas

```
{IF, +, >}
```

are functions.[2]

Every LISP program has this simple list structure. This has several advantages:

- The uniform list structure represents both program code and data. This simple syntax allows genetic operations to be simply defined.
- The format of the instructions is one of a function applied to arguments. This represents the control information in a common syntax, which is again amenable to modification with genetic operators.
- The variable-length structure of lists escapes the fixed-length limitation of the strings used in genetic algorithms.

5 Genetic Operators for Programs

A LISP program can be written as a nested list and drawn as a tree. The former fact is helpful because it allows the definition of the effects of operators on programs. The latter fact helps immensely because it allows the straightforward visualization of these effects. For example,

```
(* x (+ x y))
```

Figure 4: Functions in LISP can be interpreted as trees. Two examples are (∗ x (+ x y)) and (+ (∗ z y)(/ y x)).

Figure 5: The genetic programming crossover operator works by picking fracture points in each of two programs. Then these sublists are swapped to produce two new offspring.

and

```
(+ (* z y) (/ y x))
```

can be thought of as representing the programs that compute $x(x + y)$ and $zy + y/x$. These programs in their list structure can be interpreted also as trees, as in Figure 4.

Operators have a very elegant structure in the list format. The first step in applying an operator consists of identifying *fracture points* in the list for modification. These are analogous to crossover points in the string representation used by genetic algorithms. A fracture point may be either the beginning of a sublist or a terminal. The following examples illustrate the use of genetic operators.

To implement crossover, pick two individual programs. Next select any two sublists, one from each parent. Switch these sublists in the offspring. For example, pick y and (/ y x) from the two preceding examples. Then the offspring will be (∗ x (+ x (/ y x))) and (+ (∗ z y) y), shown in tree form in Figure 5.

To implement inversion, pick one individual from the population. Next select two fracture points within the individual. The new individual is obtained by switching the delimited subtrees. For example, using (+ (∗ z y) (/ y x)), let's pick z and (/ y x). The result is (+ (∗ (/ y x) y) z), shown in tree form in Figure 6.

Figure 6: The genetic programming inversion operator works by picking two fracture points in a single program. These sublists are swapped to produce the new offspring.

Figure 7: The genetic programming mutation operator works by picking a fracture point in a single program. Then the delimited terminal or nonterminal is changed to produce the new offspring.

To implement mutation, select one parent. Then replace randomly any function symbol with another function symbol or any terminal symbol with another terminal symbol. That is, (+ (∗ z y)(/ y x)) could become (+ (+ z y)(/ y x)), shown in tree form in Figure 7. Mutation may also be implemented by replacing a subtree with a new, randomly generated subtree.

What you have seen is that the genetic operators all have a very simple implemetation in the LISP language owing to its list structure. But there is one caveat. These examples are all arithmetic functions that return integers, so there is no problem with swapping their arguments. However, there may be a problem if the results returned by the swapped functions are not commensurate. For example, if one function returns a list and it is swapped with another that returns an integer, the result will be a syntactically invalid structure. To avoid this error (at some cost), either all functions should return the same type of result, or some type checking has to be implemented.

6 Genetic Programming

The structure of the genetic programming algorithm, Algorithm II is identical to that of the basic genetic algorithm (Algorithm 12.1). The principal difference is the interpretation of the string as a program. This difference shows up in the evaluation of fitness, wherein the program is tested on a set of inputs by evaluating it.

As examples of genetic programming we consider two problems. One is the parity problem: Given a set of binary inputs of fixed size n the program must correctly determine their parity. Although it is easy to build a parity function without using GP, this problem is valuable as a test case as it is a difficult problem for learning algorithms. Knowing the parity of subsets of the input is insufficient to predict the answer.

Algorithm II Genetic Programming

Choose a population size N_p.
Choose the number of generations N_G.
Initialize the population.
Repeat the following for N_G generations:

1. Select a given number of pairs of individuals from the population probabilistically after assigning each structure a probability proportional to observed performance.
2. Copy the selected structure(s), then apply *operators* to them to produce new structure(s).
3. Select other elements at random and replace them with the new structure(s).
4. Observe and record the fitness of the new structure.

Output the fittest individual as the answer.

The second example is that of the Pac-Man video game. In this case the program must learn to control a robotic agent in a simulated environment that has rewards and punishments.

Example 1: Parity Consider the problem of discovering the even-n parity function. The even-3 parity function is shown in Table 4. For this problem the terminals are given by three symbols that represent the three possible inputs to be tested for parity:

$$T = \{D_0, D_1, D_2\}$$

The function set is given by the set of primitive Boolean functions of two arguments

$$F = \{AND, OR, NAND, NOR\}$$

Using a population size of 4,000, a solution was discovered at generation 5. The solution contains 45 elements (the number of terminals plus the number of nonterminals). The individual comprising the solution is shown in the

Table 4: The even-3 parity function.

Input	Output
000	1
001	0
010	0
011	1
100	0
101	1
110	1
111	0

following program. As you can see, this solution is not the minimal function that could be written by hand but contains many uses of the primitive function set, mostly because there is no pressure on the selection mechanism to choose minimal functions.

```
(AND (OR (OR D0 (NOR D1 D2)) D2)
      (AND (NAND (NOR (NOR D0 D2)
                      (AND (AND D1 D1) D1))
                 (NAND (OR (AND D0 D1) D2) D0))
(OR (NAND (AND (D0 D2)
                 (OR (NOR D0 (OR D2 D0) D1))
    (NAND (NAND D1 (NAND D0 D1)) D2)))
```

Example 2: Pac-Man As another illustration of the use of GP, consider learning to play the game of Pac-Man.[3] The game is played on a 31×28 grid, as shown in Figure 8. At every time step, the Pac-Man can remain stationary or move one step in any possible direction in the maze. The goal of the game is to maximize points. Points are given for eating the food pellets arrayed along the maze corridors. Energizers are special objects and are worth 50 points each. At fixed times $t = 25$ and $t = 125$ a very valuable piece of fruit worth $2,000$ points appears, moves on the screen for 75 time steps, and then disappears.

In the central den are four monsters. After the game begins they emerge from the den and chase the Pac-Man. If they catch him the game ends. Thus the game ends when the Pac-Man is eaten or when there is nothing left for him to eat. In this eat-or-be-eaten scenario the energizers play a valuable role. After the Pac-Man has eaten an energizer, all the monsters are vulnerable for 40 time steps and can be eaten. (In the video game the monsters turn blue for this period.) Thus a strategy for a human player is to lure the monsters close to an energizer, eat the energizer, and then eat the monster. Eating more than one monster is especially valuable. The first is worth 200 points, the next 400, the next 800, and the last 1600.

Figure 8: The board for the Pac-Man game, together with an example trace of a program found by genetic programming. The path of the Pac-Man is shown with vertical dots, and the paths of the monsters are shown with horizontal dots. In this example the Pac-Man starts in the middle of the second row from the bottom. The four monsters start in the central den. The Pac-Man first moves toward the upper right, where it captures a fruit and a pill, and subsequently attracts the monsters in the lower left corner. After eating a pill there it eats three monsters and almost catches the fourth.

Table 5: The GP Pac-Man nonterminals (control functions). (After Koza, 1992.)

Control Function Name	**Purpose**
If-Blue (IFB)	A two-argument branching operator that executes the first branch if the monsters are blue; otherwise the second.
If-Less-Than-or-Equal (IFLTE)	A four-argument branching operator that compares its second argument to its first. If it is less, the third argument is executed; otherwise the fourth is executed.

The monsters are as fast as the Pac-Man but are not quite as dogged. Of every 25 time steps, 20 are spent in pursuit of the Pac-Man and 5 in moving at random. But since there are more of them, they can succeed by cornering the Pac-Man.

The parity problem was selected for its special appeal: its difficulty makes it a good benchmark. In the same way the Pac-Man problem is especially interesting as it captures the problem of survival in the abstract. The Pac-Man has to feed itself and survive in the face of predators. It has primitive sensors that can identify food and foes as well as motor capability. As such it is an excellent test bed to study whether GP can successfully find a good control strategy. Furthermore, the GP operations are so simple you can imagine how they might be done neurally in terms of changing wiring patterns introducing a changing control structure. Thus the evolutionary roots of the algorithm may be translatable into neural growth.

Tables 5 and 6 show the GP encoding of the problem. The two nonterminals appear as interior nodes of the tree and allow the Pac-Man to change behavior based on current conditions. The terminals either move the Pac-Man in the maze or take a measurement of the current environment, but they cannot call other functions.

Figure 9 shows the solution obtained in a run of GP after 35 generations.[4] Note the form of the solution. The program solves the problem almost by rote, since the environment is similar each time. The monsters appear at the same time and move in the same way (more or less) each time. The regularities in the environment are incorporated directly into the program code. Thus while the solution is better than what a human programmer might come up with, it is very brittle and would not work well if the environment changed significantly.

The dynamics of GP can also be illustrated with this example. Figure 10 shows how the population evolves over time. Plotted are the number of individuals with a given fitness value as a function of both fitness value and generations. An obvious feature of the graph is that the maximum fitness value increases as a function of generations. However, an interesting subsidiary feature of the graph is the dynamics of genetic programming. Just as in the analysis of GAs, fit schema will tend to increase in the population exponentially,

```
0 (IFB
1    (IFB
2      (IFLTE (AFRUIT)(AFRUIT)
3      (IFB
4          (IFB
5           (IFLTE
6             (IFLTE
                 (AGA)
                 (DISA)
7,8              (IFB (IFLTE
                         (DISF)
                         (AGA)
                         (DPILL)
9                        (IFLTE (DISU)(AGA)
                            (AGA)
10                          (IFLTE (AFRUIT)(DISU)
                               (AFRUIT)
10,9,8                         (DISA))))
8                      (IFLTE (AFRUIT)(RGA)
                         (IFB (DISA) 0)
8,7                       (DISA)))
6                (DPILL))
              (IFB
7                (IFB (AGA)
8                     (IFLTE
9                        (IFLTE
                            (IFLTE (AFRUIT)(AFOOD)(DISA)(DISA))
                            (AFRUIT)
                            0
9                           (IFB (AGA) 0))
                         (DPILL)
                         (IFLTE (AFRUIT)(DPILL)(RGA)(DISF))
8,7                      (AFRUIT)))
                 0)
          (AGA)
5         (RGA))
4     (AFRUIT))

3     (IFLTE
4         (IFLTE (RGA) (AFRUIT)(AFOOD)(AFOOD))
          (IFB(DPILL)(IFLTE (RGA)(APILL)(AFOOD)(DISU)))
5         (IFLTE
             (IFLTE (RGA)(AFRUIT)(AFOOD)(RPILL))
             (IFB (AGA) (DISB))
             (IFB (AFOOD) 2)
5            (IFB (DISB) (AFOOD)))
4,3       (IFB (DPILL) (AFOOD))))
2     (RPIL))
3     (IFB (DISB)
4          (IFLTE
              (DISU)
              0
              (AFOOD)
4,3,2         (AGA))))
2       (IFB (DISU)
3           (IFLTE
              (DISU)
              (DISU)
4             (IFLTE
5                 (IFLTE (AFRUIT)
                         (AFOOD)
                         (DPILL)
5                        (DISA))
                (AFRUIT)
                0
4               (IFB (AGA) 0))
3,2,1       (RGB))))
```

Figure 9: Program code that plays Pac-Man shows the ad hoc nature of the solution generated by GP. The numbers at the line beginnings indicate the level of indentation in the LISP code. (After Koza, 1992.)

Table 6: The GP Pac-Man terminals. (After Koza, 1992.)

Terminal Fuction Name	**Purpose**
Advance-to-Pill (APILL)	Move toward nearest uneaten energizer
Retreat-fom-Pill (RPILL)	Move away from nearest uneaten energizer
Distance-to-Pill (DPILL)	Distance to nearest uneaten energizer
Advance-to-Monster-A (AGA)	Move toward monster A
Retreat-from-Monster-A (RGA)	Move away from monster A
Distance-to-Monster-A (DISA)	Distance to monster A
Advance-to-Monster-B (AGB)	Move toward monster B
Retreat-from-Monster-B (RGB)	Move away from monster B
Distance-to-Monster-B (DISB)	Distance to monster B
Advance-to-Food (AFOOD)	Move toward nearest uneaten food
Distance-to-Food (DISD)	Distance to nearest food
Advance-to-Fruit (AFRUIT)	Move toward nearest fruit
Distance-to-Fruit (DISF)	Distance to nearest fruit

Figure 10: The dynamics of GP is illustrated by tracking the fitness of individuals as a function of generations. The vertical axis shows the number of individuals with a given fitness value. The horizontal axes show the fitness values and the generations. The peaks in the figure show clearly the effects of good discoveries. The fittest of the individuals rapidly saturate almost the whole population.

and that fact is illustrated here. When a fit individual is discovered, the number of individuals with that fitness level rapidly increases until saturation. At the same time the sexual reproduction operators are working to make the population more diverse; that is, they are working to break up this schema and increase the diversity of the population. Once the number of this schema has saturated, then the diversifying effects of the sexual reproduction can catch up to the slowed growth rate of individuals with the new schema.

7 Analysis

Since the performance of GP is too difficult to approach theoretically, we can attempt to characterize it experimentally.[5] The experimental approach is to observe the behavior of the algorithm on a variety of different runs of GP while varying N_G and N_p, and then attempt to draw some general conclusions. The most elementary parameter to measure is the probability of generating a solution at a particular generation k using a population size N_p. Call this $P_k(N_p)$, and let the cumulative probability of finding a solution up to and including generation i be given by

$$P(N_G, i) = \sum_{k=1}^{i} P_k(N_p)$$

One might think that P_k would increase monotonically with k so that using more generations would inevitably lead to a solution, but the experimental evidence shows otherwise. The fitness of individuals tends to

plateau, meaning that the search for a solution has gotten stuck in a local minimum. Thus for most cases P_k eventually decreases with k. The recourse is to assume that successive experiments are independent and find the optimum with repeated runs that use different initial conditions.

A simple formula dictates the number of runs needed based on empirical success measures. This process leads to a trade-off between population size and multiple runs.

Assuming the runs are independent, you can estimate the number needed as long as $P(N_p, i)$ is known. This process works as follows. Let the probability of getting the answer after G generations be P_A. Then P_A is related to the number of experiments r by

$$P_A = 1 - [1 - P(N_p, G)]^r$$

Since the variable of interest is r, take logarithms of both sides,

$$\ln(1 - P_A) = r \ln[1 - P(N_p, G)]$$

or

$$r = \frac{\ln \epsilon}{\ln[1 - P(N_p, G)]}$$

where ϵ is the error, $(1 - P_A)$.

These formulas allow you to get rough estimates on the GP algorithm for a particular problem, but a more basic question that you might have is about the efficacy of GP versus ordinary random search. Could one do as well by just taking the search effort and expending it by testing randomly generated examples? It turns out that the answer to this question is problem dependent, but for most hard problems the experimental evidence is that the performance of GP is much better than random search.[6]

8 Modules

A striking feature of the Pac-Man solution shown in Figure 9 is its nonhierarchical structure. Each piece of the program is generated without taking advantage of similarities with other parts of the code. In the same way the parity problem would be a lot easier to solve if functions that solved subproblems could be used. Ideally one would like to define a hierarchy of functions, but it is not at all apparent how to do so. The obvious way would be to denote a subtree as a function and then add it to the library of functions that can be used in the GP algorithm. In LISP, the function call (FUNC X Y) would simply be replaced by a defined function name, say FN001, whose definition would be (FUNC X Y), and the rest of the computations would proceed as in the standard algorithm. The problem with this idea is the potential explosion in the number of functions. You cannot allow functions of all sizes in the GP algorithm owing to the expense of testing them. Each program is a tree containing subprograms as subtrees. Naively testing all subtrees in a population increases the workload by a factor of N^2. However, it is possible to test small trees that have terminals at their leaves efficiently by using a data structure that keeps track of them.[7] This way the price of testing the trees is reduced to a manageable size. Thus to make modules, all trees of size less than M are tested for some M that is chosen to be practical computationally.

The modular version of GP for the parity problem tests each of these small subtrees to see if they solve the problem for their subset of inputs. If they do, then each of those subtrees is made a function with the number of arguments equal to the variables used in the subtree. This function is then added to the pool of nonterminals and can be used anywhere in the population. This process is recursive. Once a new function is added to the nonterminal pool, it can be used as part of yet newer function compositions. If later generations

Algorithm III Modular Genetic Programming

Choose a population size.
Choose the number of generations N_G.
Initialize the population.
Repeat the following for N_G generations:

1. Select a given number of pairs of individuals from the population probabilistically after assigning each structure a probability proportional to observed performance.
2. Copy the selected structure(s), then apply *operators* to them to produce new structure(s).
3. Select other elements at random and replace them with the new structure(s).
   (a) Test all the subtrees of size less than M. If there are any subtrees that solve a subproblem, add them to the nonterminal set.
   (b) Update the subtree data set.
4. Observe and record the fitness of the new structure.

Output the fittest individual as the answer.

solve a larger parity problem using the new function as a component, they can be treated in the same way: another new function is added to the nonterminal set. These constraints are formalized in Algorithm III

The modular approach works spectacularly well for parity, as shown in Table 7. Consider the entries for the 5 parity problem. Genetic programming without modules requires 50 generations, whereas the same problem can be solved in only 5 generations using modular GP. The reason, of course, is that functions that solve the parity of subsets of the data are enormously helpful as components of the larger solution.

Figure 11 shows the generation and use of new functions in the course of solving the 8 parity problem. In the course of the solution, functions that solved the 2, 3, 4, and 5 parity problem were generated at generations 0, 1, 3, and 7, respectively. The specific code for each of these functions is shown in Table 8. These functions may look cumbersome, but they are readily interpretable. Function F681 is the negation of a function of a single argument. Note that this was not in the initial function library. Function F682 computes the parity of its two arguments. Not surprisingly this function is extremely useful in larger parity problems, and you can see that it is used extensively in the final solution shown in the table.

Table 7: Comparison of GP and modular GP for parity problems of different sizes. Table entry is the generation at which the solution was found. Standard GP fails to find a solution to the even-8 parity problem after 50 generations, but modular GP solves the same problem at generation 10.

Method	**Even-3**	**Even-4**	**Even-5**	**Even-8**
GP	5	23	50	
Modular GP	2	3	5	10

Table 8: Important steps in the evolutionary trace for a run of even-8 parity. "LAMBDA" is a generic construct used to define the body of new functions. (From Rosca and Ballard, 1994.)

Generation 0. New functions
[F681]: (LAMBDA (D3) (NOR D3 (AND D3 D3)));
[F682]: (LAMBDA (D4 D3) (NAND (OR D3 D4) (NAND D3 D4)))

Generation 1. New function
[F683]: (LAMBDA (D4 D5 D7) (F682 (F681 D4) (F682 D7 D5)))

Generation 3. New functions
[F684]: (LAMBDA (D4 D5 D0 D1 D6) (F683 (F683 D0 D6 D1) (F681 D4) (OR D5 D5)));
[F685]: (LAMBDA (D1 D7 D6 D5) (F683 (F681 D1) (AND D7 D7) (F682 D5 D6)))

Generation 7. The solution found is: (OR (F682 (F682 (F683 D4 D2 D6) (NAND (NAND (AND D6 D1) (F681 D5)) D1)) (F682 (F683 D5 D0 D3) (NOR D7 D2))) D5)

Figure 11: Call graph for the extended function set in the even-8 parity example showing the generation when each function was discovered. For example, even-5 parity was discovered at generation three and uses even-3 parity discovered at generation one and even-2 parity and NOT, each appearing at generation zero. (From Rosca and Ballard, 1994.)

Algorithm IV Selecting a Subroutine

For individuals that have small height:

1. Select a set of promising individuals that have positive differential fitness.
2. Compute the number of activations of each individual, and remove individuals with inactive nodes.
3. For each individual with terminals T_s and block b do the following:
   (a) Create a new subroutine having a random subset of the terminals T_s with body (b, T_s).
   (b) Choose a program that uses the block b, and replace it with the new subroutine.

8.1 Testing for a Module Function

The previous strategy for selecting modules used all the trees less than some height bound. This strategy can be improved by keeping track of *differential fitness*. The idea is that promising modules will confer a fitness on their host that is greater than the average, that is,

$$\Delta f_s = f_{host} - f_{parents}$$

Thus a more useful measure of subroutine fitness is to select candidates that have a positive differential fitness. With this modification, you can use Algorithm IV to select subroutines.

8.2 When to Diversify

The distribution of the fitness among individuals in a population is captured by the fitness histogram, which plots, for a set of discrete fitness ranges, the number of individuals within each range. Insight into the dynamics of propagation of individuals using subroutines can be seen by graphing the fitness histogram as a function of generation, as is done in Figure 12. This figure shows that once a good individual appears it grows exponentially to saturate the population. These individuals are subsequently replaced by an exponential growth of a fitter individual. The stunning improvement is seen for the example of Pac-Man. Comparing this result with that of Figure 10, the plot of the fitness histogram shows that not only is the popluation more diversified, but also that the fitness increases faster and ultimately settles at a larger point.

Further insight into the efficacy of the modular algorithm can be obtained by studying the entropy of the population. Let us define an entropy measure by equating all individuals with the same fitness measure.

Figure 12: The dynamics of GP with subroutines is illustrated by tracking the fitness of individuals as a function of generations. The vertical axis shows the number of individuals with a given fitness value. The horizontal axes show the fitness values and the generations. The beneficial effects of subroutines can be seen by comparing this figure with Figure 13.7.

Figure 13: Entropy used as a measure to evaluate the GP search process. If the solution is found or if the population is stuck on a local fitness maximum, the entropy of the population tends to slowly decrease (*top*). In searching for a solution the diversity of a population typically increases, as reflected in the fluctuations in entropy in the modular case (*bottom*).

If fitness is normalized, then it can be used like a probability. Then entropy is just

$$E = \sum f_i \log f_i$$

When the solution is not found, as in nonmodular GP, the search process decreases the diversity of the population, as shown by the top graph in Figure 13. This result is symptomatic of getting stuck in a local fitness maximum. But the nonlocal search of modular GP continually introduces diversity into the population. This process in turn keeps the entropy measure high, as shown in the lower graph in Figure 13. This behavior can be used to detect when a solution is stuck. If the entropy is decreasing but the fitness is constant, then it is extremely likely that the population is being dominated by a few types of individuals of maximum fitness. In this case the diversity of the population can be increased by increasing the rate of adding new modules.

Finally, you can see the effects of the subroutine strategy from Table 9, which compares runs from GP with and without subroutines.

9 Summary

The features of GP search are qualitatively similar to that of GAs. Once an individual of slightly increased differential fitness is produced, its features tend to spread exponentially through the population. During this

Table 9: Comparison between solutions obtained with standard GP, subroutines, and a carefully hand-designed program. The GP parameters have the following values: population size = 500; crossover rate = 90%; reproduction rate = 10% (no mutation). During evolution, fitness is determined by simulation in 1 or 3 different environments. Each table entry shows the total number of points of an evolved Pac-Man controller for the corresponding GP implementation and, in parentheses, the generation when that solution was discovered. For all entries, the best results over 100 simulations were taken.

GP		**Subroutines**		**Hand**
1	3	1	3	
6380	4293	9840	5793	5630
(67)	(17)	(20)	(36)	

period the mechanism of sexual reproduction helps this process along. However, once the features saturate the population, sexual reproduction continues by breaking up program segments in the course of searching for new above-average features. Since most changes tend to be harmful, the course of sexual reproduction and schemata growth can be seen as being in dynamic balance.

In the world, nature provides the fitness function. This function is extremely complicated, reflecting the complexity of the organism in its milieu. Inside the GP alogorithm, however, fitness must be estimated, using both *a priori* knowledge and knowledge from the search process itself. The power of subroutines in this context is that they provide a way of pooling the results of the organism's experiments across the population. Thus they provide a way of estimating the effects of a schema en route to a solution.

Notes

1. John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Cambridge, MA: MIT Press, Bradford, 1992).

2. Actually, one does not put all the integers into the terminal set. Instead a function *random-integer* is used; whenever it is called, a random integer is generated and inserted into the program.

3. The Pac-Man example was introduced by Koza in *Genetic Programming* and refined by Justinian P. Rosca and Dana H. Ballard in "Discovery of Subroutines in Genetic Programming," Chapter 9 in P. Angeline and K. E. Kinnear, Jr., eds., *Advances in Genetic Programming 2* (Cambridge, MA: MIT Press, in press).

4. Koza, *Genetic Programming*.

5. This analysis was first done by Koza in *Genetic Programming*.

6. See, for example, Chapter 9 of Koza, *Genetic Programming*.

7. Justinian P. Rosca and Dana H. Ballard, "Genetic Programming with Adaptive Representations," TR 489, Computer Science Department, University of Rochester, February 1994.