An Explanation of the Genetic Algorithm

Abstract

The genetic algorithm specifies to the computer what is to be done rather than how to do it. Genetic operators function as a generalized technique which can be applied to any class of problems. For this reason they are similar in nature to simple enumerative (brute-force) searches though they exhibit the efficiency of standard problem-solving algorithms.

The genetic algorithm searches the domain of action of a given problem by evaluating all solutions in a small subset of all possible solutions. By using only three of the seven types of genetic operators proposed by Holland — reproduction, crossover, and mutation — the subset of solutions moves through the domain of action. This movement focuses on better-performing solutions which enables the genetic algorithm to identify progressively better and better solutions.

The purpose of this paper is to outline John Holland’s original plan for the genetic algorithm, to illustrate its action through simplified examples and to discuss some of the underlying mathematical assumptions the genetic algorithm is built upon.

A slightly different version of this paper is available at http://kirsch.cs.washington.edu/pardoe. This version contains examples and dynamic tables with a variety of functions. The text and code of the website follows as an appendix. Copies of the TeX source and web pages will be available by e-mail from <apardoe@acm.org>.

Andrew Pardoe
Computer programming is naturally done in a deterministic fashion — humans design plans based upon our ability to build a proof step by step. The programmer provides an algorithm as a series of instructions to the machine: assign this value, subtract that amount, branch if the answer is equal to zero. These instructions can be charted according to the flow of control in the program. Each step is determined: no matter what time $t$ at which you step into the algorithm you will always get the same action at time $t+1$. The machine is told how to handle each condition which it will encounter.

One of the motivations John Holland had in the development of the genetic algorithm was to program the computer by specifying what is to be done rather than how to do it. The genetic algorithm was developed as a general technique for solving difficult problems rather than as an algorithm which solves a single given problem. Its operation is more similar to a brute-force enumerative plan than an iterative algorithm in that it searches a given domain of action without needing special instructions specific to that problem. Dijkstra's minimum-path building algorithm, for example, builds a set of minimum paths by processing the edges and nodes of a graph. The algorithm contains special knowledge of a graph's structure and builds a solution one node at a time. It cannot be used for any problem which cannot be represented as a minimum-path problem. Genetic algorithms are applicable to graph problems — and many other problems — without any special knowledge of what the components of a graph are or how they are arranged. The fact that genetic algorithms are a technique rather than a domain-specific plan means that they can be applied to a great range of problems.

As they are also instances of general purpose techniques, enumerative plans work well for some small problems. For example, most people take the same route from home to work every day. There are (usually) a number of routes available, each with varying distances and traffic throughput. How does the average person choose the best route to or from work? A reasonable plan is to take the surface streets one day, the highway the next, and an arterial the third. Eventually one can settle on the perceived fastest ways to and from work. This is an example of an effective enumerative plan: we take all of the reasonable routes from one place to another and test each one individually. When the tests are completed (possibly multiple times) a decision is made as to which is best.

This simple commute problem is a modified version of the Traveling Salesman Problem. More properly stated the Traveling Salesman Problem is an attempt to find the "best" (cheapest, fastest, shortest, etc.) Hamiltonian cycle on a weighted graph $n$ vertices. Assuming that we can get directly from any one city to any other we have $\binom{n}{2}$ possible edges to consider of which we want to include $n$ in our tour. The commute problem is a problem of finding the shortest path, similar to the Travelling Salesman Problem. (With a commute, of course, one often takes the same route both to and from work as opposed to a Travelling Salesman Problem which requires a Hamiltonian cycle.)

What is the total number of possible tours which we may need to inspect for the solution? Asking the same question with different terms, what is the domain of action for this problem? Given that we wish to start in a particular city (presumably where the salesman lives and works) the problem reduces to an arrangement of the remaining $n-1$ cities. Thus we have $(n-1)!$ possible tours. For some small number of cities this problem is trivial but for a tour of only 12 cities we have almost 40 million possible combinations to consider!

Unfortunately there exists no algorithm which will solve the Traveling Salesman Problem in a deterministic fashion. Moreover it is obvious from the size of the domain of action (40 million possible routes) that an enumerative plan would be useless. Enumeration may work if we are lucky enough to find a good route quickly but we must expect to search the entire domain of action in order to find the best solution. Searching over 40 million possible routes and assigning a relative fitness to each route would take an incredible amount of processing time.

If we look at the problem logically, however, we can immediately eliminate most of our choices. Even a child with a map and a pencil could sketch out a general travel plan which would eliminate the majority of the 40 million routes. One simple example is that under very few circumstances would we wish to fly from Seattle to New York to Portland to Washington, D.C. Yet an enumerative plan would evaluate over 40 thousand plans with include this sequence.

What advantage does a person with a pencil and a map have over a computer? Our intelligence acts in subtle ways. From the first instance — as soon as our eyes start following the lines drawn on the map — we eliminate plans which backtrack over great distances as unreasonable. When particularly attractive or
unattractive combinations arise we take note and reinforce these in future attempts at routing. We focus on
the probable solutions which, for whatever motivation, seem reasonable. In short, we adapt our planning by
considering the perceived qualities of solutions and partial solutions we have encountered before. This is in
direct opposition to the action of an enumerative plan, which is completely unaffected by the history of its
search.

The genetic algorithm searches through the domain of action in a similar fashion by attempting to
find the best solutions in a biased fashion. It is designed to give increasingly more representation to those
solutions which appear to be more fit. The technique by which it gives increasing representation is modelled
after Darwinian evolution: the fittest individual solutions survive in the domain of action and the weakest are
eliminated. This process is similar as well to a human's decision to eliminate Seattle to New York to Portland
as an unreasonable route: as the genetic algorithm develops new solutions it will spend proportionately
less time searching over the worst solutions. By removing less fit solutions from the domain of action
the likelihood of more fit solutions — and possibly the perfect solution — increases exponentially. This
progressive modification of the domain of action to suit new circumstances is the basic feature of adaptation.

Holland identifies five obstacles which face adaptive plans. These obstacles are problems which must
be well-handled for an adaptive plan to be a considered an effective yet still general technique for problem
solving. The problems are: ¹

1. The domain of action \( \alpha \) — the set of potential solutions — is too large to test efficiently.
2. The member "solutions" \( A \subset \alpha \) in the domain of action are so complicated that it is difficult to figure
   out where credit for a potential solution's performance is due.
3. The performance measure \( \mu \) — or relative fitness — is cumbersome to use because of its complexity.
4. The performance measure varies over time or space so that inconsistent results are obtained.
5. The environment presents too great an amount of information which must be filtered and sorted for
   relevance.

The genetic algorithm addresses all of these issues successfully. Though it is not appropriate for all
problems it is truly an efficient general-purpose technique for obtaining solutions to combinatorially explosive
problems. The purpose of this paper is to outline the structure Holland's genetic algorithm, illustrate its
action through a simple example and discuss some of the underlying mathematical assumptions it is built
upon.

A simple adaptive plan

Adaptation is one of the salient features of an intelligent agent. We adapt our actions based upon our
experiences constantly. Suppose, for the sake of example, that you are sitting in front of a two-armed slot
machine with a pocket full of nickels (\( N \) nickels, to be exact.) Each arm of the machine is labelled with
that arm's probability of paying off. Unfortunately the labels read "Payoff probability: \( \mu_1 \)" and "Payoff
probability: \( \mu_2 \)" with no explanation of what numeric values are associate with each \( \mu \). How are you
to maximize your profit?

Given that you will play all of your nickels (otherwise, why are you sitting in front of a slot machine?)
your safest algorithm would be to split the nickels into two piles, each of size \( n = N/2 \). Play \( n \) of the nickels
on \( \mu_1 \) and \( n \) on \( \mu_2 \) to obtain expected winnings of \( N \cdot (\mu_1 + \mu_2) / 2 \).

What if, as you play, you notice that \( \mu_1 \neq \mu_2 \)? That is to say, you find yourself winning repeatedly from
the one arm and losing repeatedly from the other. Most likely you would change your strategy and place all
of your bets with the winning arm. This instinctive action leads us to a better plan: divide your trials into
a first phase, one of exploration, and a second phase, of exploitation.

The method in detail is this: allocated \( n < N \) nickels for exploration. Play \( n/2 \) nickels on each arm,
keeping track of your winnings as you play. These nickels will earn the same expected winnings as the safe
strategy discussed above: \( n \cdot (\mu_1 + \mu_2) / 2 \). Play your remaining \( N - n \) nickels on the observed — and thus
assumed — better arm. Assuming that \( \mu_1 \) is the observed better arm your expected winnings in this round
of exploitation are \( (N - n) \cdot \mu_1 \). Your total expected winnings with this strategy are the sum of these:
\( (N - n) \cdot \mu_1 + n \cdot (\mu_1 + \mu_2) / 2 \) which reduces to \( (N - n) \cdot \mu_1 + \frac{n}{2} \cdot \mu_2 \). As \( n \) approaches 0 this algorithm is
perfect, provided that \( \mu_1 > \mu_2 \). It is obviously more risky — with a shorter phase of experimentation you

¹ Holland, p. 5
are less sure of the relative values of $\mu_1$ and $\mu_2$. As $n$ approaches $N$ this algorithm is optimally safe, but not as beneficial — you play fewer nickels on the perceived better arm.

Next assume you are playing with this strategy when you notice something odd. The arm which was doing so well for you suddenly starts to fail. As you have kept careful notes you easily determine that $\mu_1$ appears to be lower than $\mu_2$. You have made the wrong choice. What will you do? Obviously, switch arms. Your plan can be constantly adaptive. Essentially you can experiment and exploit simultaneously for a progressively adaptive plan.

This is the heart of the genetic algorithm. By constantly allocating a greater number of trials to those solutions which appear to be more fit we hope to maximize our gains through adaptation. We are not by any means guaranteed of success. In our example of the two-armed bandit only an infinite number of trials will establish with absolute certainty the greater of $\mu_1$ and $\mu_2$. This exhaustive plan of experimentation, however, is certainly less rewarding than constant analysis and adaptation provided that $\mu_1$ and $\mu_2$ are dissimilar.

**Generalized genetic operators**

The problem of the two-armed slot machine is an extremely simple problem. There are only two solutions to the problem of which arm to play. Real-world problems, like the Traveling Salesman Problem, have a much larger domain of action.

Holland defines a set of generalized genetic operators by which we can search and manipulate the sample space. The most commonly used operators are reproduction, crossover and mutation. Other operators such as inversion, dominance-change, segregation and translocation, intrachromosomal duplication and deletion are less well known. The first three operators alone provide enough power for a simple genetic algorithm to be effective.

**Reproduction**

An enumerative plan acts upon the entire set of possible solutions $\alpha$, individually inspecting each solution $A \subset \alpha$ until a perfect solution is found. A genetic algorithm acts at any time upon a manageable subset of all possible solutions. This subset is referred to as the population $\beta$. It is expected that as the genetic operators are applied to $\beta$ that the entire space of $\alpha$ will be sampled. The reproduction operator does not increase the scope of $\beta$, however. Its role is to identify which solutions are more fit and increase their proportional representation in the population so that they are acted on by other genetic operators relatively more frequently than less fit solutions.

The algorithm by which reproduction is done is as follows. First, use an appropriate fitness function to assign a probability of selection to each member of the population. Second, select one structure from the population probabilistically. Third, select one member of the population in a random, uniform fashion. Lastly, replace the second member with a copy of the first.  

It is clear that the reproduction operator can be viewed as a simple transition matrix with an easily determined steady state. Take as example a population with size $n$ and average fitness $x$. Say that a member $m \in \beta$ has fitness $y$. We would expect that after a large number of cycles of reproduction have taken place we would have $\frac{n}{x}$ copies of $m$ in the population of size $n$. The same is true for any member of $n$ provided that member had some representation in $n$ at every stage.

This is a defining point for the reproduction operator. While the population cannot be so large that it is impossible to process its members it must be large enough to represent the diversity of its members. A fit member which was represented only once in the initial population could be made extinct at the uniform elimination step (which is done without regard to fitness.) In this case we have no possibility of obtaining a steady state which is representative of the relative fitnesses of the original population’s members. If the member lost was a fit member we have no chance of identifying it as a possible solution.

By keeping the population $\beta$ at a manageable level, however, we address the first obstacle that genetic algorithms were designed to overcome. If we can examine a representative set of $\alpha$ in $\beta$ we no longer have to worry that the domain of action $\alpha$ is too large to test efficiently.

**Fitness functions**

The largest problem that any enumerative plan faces — reproduction included — is the identification of an adequate fitness function. This is the most domain-specific problem of genetic algorithms. Some problems

---

2 Holland, p. 91
have easily defined fitness functions such as “maximize $f(x) = x^2$.” Other problems are complex but still quantifiable, such as the Traveling Salesman Problem. An appropriate fitness function would be to sum distance (or cost) on the chosen edges and assign higher reproductive probabilities to the lower sums.

The problem of the fitness function is an asset when it deals with less well-defined problems. Simple problems such as “maximize $f(x) = x^2$” are obviously better solved with algebra or calculus. Even NP-complete problems such as the Traveling Salesman Problem can be handled fairly well through the intelligent conversion of the problem to a similar problem which can be solved deterministically in reasonable time. For example, to solve the Traveling Salesman Problem one can apply an algorithm such as Dijkstra’s minimum paths algorithm to obtain a minimum spanning tree. Once we have this tree we can convert it into a tour by traversing every edge of the minimum spanning tree twice. We can then eliminating duplicated edges by finding another edge which will bypass the duplicated edges. Assuming that the triangular inequality holds (any two edges of a triangle are greater in sum than the third edge along) we can in this manner we obtain a tour of the vertices which is no more than 1.5 times the optimal length.\footnote{TSP algorithm from Manber, pp. 365 - 367}

These two problems are well-defined problems. Although the latter is difficult it is still easy to express exactly what conditions are beneficial to a good solution. In the absence of this sort of simple definition of the problem an enumerative plan — and the genetic algorithm — has an advantage. A grade can be assigned for something which has many diverse factors. Think, for example, of how a professor assigns a student’s grade based on homework, exams and projects. In an even more complicated problem we grade other people based upon their perceived qualities when we fall in love. Ranking is a natural process for humans to do and it is generally not difficult to codify our biases. While a fitness function for an ill-defined problem may not be perfect the genetic algorithm will find an answer to the question we ask it to answer. By handling problems with cumbersome measures of performance $\mu$ the genetic algorithm overcomes the third obstacle.

**Crossover**

Reproduction works on entire members of the population by recopying entire solutions based upon their relative fitnesses. While the best solution to the problem exists in the domain of action $\alpha$ it may not exist in any given population $\beta \subset \alpha$. If it does not appear in the first population $\beta(0)$ reproduction will not introduce it into any population $\beta(t)$. Crossover allows some variation in the existing population by separating and recombining pieces of solutions. It directly parallels the biological idea of exchanging alleles between pairs of chromosomes. By viewing solutions as pieces of solutions rather than entire solutions we can reduce the complexity of a potential solution $A \subset \beta$ to better assign credit for the solution’s fitness. This addresses the second issue outlined by Holland as an obstacle for genetic algorithms.

The generalized plan for crossover is as follows. First, select two members of the population, usually at random. Then select at random a position at which to cross over. Lastly, recombine the first half of the first member with the second half of the second member and vice versa. In this operation two new members with the characteristics of both old members have been formed.

How is it that the characteristics of a solution are carried over into the new, crossed-over members of the population? Obviously we can no longer define a solution as an atomic structure because some part carries its characteristics. This is similar to the biological idea that alleles, not chromosomes, determine our genetic makeup. In order to account for the idea that solutions are not atomic Holland introduces the concept of schemata.

Schemata give us a way to associate a combination of attributes with the performance of a solution. Going back to the Traveling Salesman Problem, a subroute of Seattle to Portland to San Francisco may prove to be an especially effective subroute regardless of how New York, Chicago, Cleveland, Pittsburgh and Toronto are arranged. Any solution including the subroute Seattle - Portland - San Francisco will enjoy the benefits of this efficient segment of travel.

Crossover has three direct effects on a set of schemata represented in a population. The first is that new instances of existing schema will be created which have potentially different fitness values associated with them. For example, the Seattle - Portland - San Francisco route may be paired up with or split away from an efficient Great Lakes segment. Either action will produce an instance of the schema with a different fitness value. If the new solution’s fitness is greater the new solution provides the schema a better chance at reproduction.
The second direct effect of crossover is the generation of new schema. For example, a segment from Seattle - Portland - San Francisco may be pasted directly in front of a segment from San Jose to Los Angeles. This new, longer schema may in turn help other routes as an extremely efficient segment.

The third effect of crossover on schemata is the destruction of individual copies of long schemata. As the crossover site is selected at random without regard to any schema, one which spans both sides of the crossover site will be destroyed. For example, a schema which consists of “start in Seattle and go to Portland. At the end of the trip, go to Anchorage and return to Seattle from there” has little chance of surviving crossover. Most likely, however, both the initial and terminal subsegments will appear in the final solution if it is adequately efficient of its own accord. Also, if multiple copies of the schema appeared in the population it is unlikely that crossover would affect all of them simultaneously.

Reproduction with crossover

Operating together reproduction and crossover provide the two main advantages of genetic algorithms, intrinsic parallelism and compact storage of information. Each crossing over of dissimilar members of a population of length \( l \) will create up to \( 2^l \) new schemata which are not referenced in the original population. For a length \( l \) of only 20 a single crossing over processes over a million schemata.\(^4\) Further, the population \( \beta \subset \alpha \) which is being processed by these two genetic operations contains all of the information about the population being processed. If a schema \( \xi \) has relative fitness \( \mu_\xi \) there will be \( \mu_\xi/\bar{\mu} \) copies of \( \xi \) in the population \( \beta \) (where \( \bar{\mu} \) is the average fitness of the population \( \beta \).) Moreover, “the state of the algorithm at the beginning of any cycle includes not only the population \( \beta(t) \) but also the retained performances” of its members.\(^5\) Thus the storage requirements of the genetic algorithm are no greater than the size of the population on which it works. The size of the population — its carrying capacity, for a biological parallel — can be scaled to fit the processing power available.

The environment in which a genetic algorithm works is its population \( \beta \), composed completely of member solutions \( A \subset \alpha \) which are in turn composed of schemata. Not only does this create an algorithm which has extremely efficient storage requirements but the environment has essentially no noise. This addresses the fifth obstacle outlined by Holland.

Repeated crossing-over with reproduction produces a steady state in which any schema \( \xi \) appears with a probability \( \lambda(\xi) \). The expected time until an occurrence of \( \xi \) is the reciprocal \( 1/\lambda(\xi) \). Thus repeated crossing-over creates a steady state which cycles through all possible schemata in the domain of action \( \alpha \). Those schemata with short defining length \( l \) are given greater representation as they are less likely to be disrupted by crossover but each schema represented in the population will appear independently of the action of any other schema in the population.\(^6\) Thus each schema \( \xi \subset \alpha \) is independently exponentially distributed in the population \( \beta \).

That the rate of representation of a schema based upon its individual fitness is exponentially distributed with relation to its fitness is a key point of the genetic algorithm. This fact allows us to see that the population \( \beta \) moves through the domain of action \( \alpha \) with increasing speed toward likely solutions. As the criterion of average fitness \( \bar{\mu} \) is constantly rising a fit solution must be exponentially better each generation. Thus a genetic algorithm will focus on good solutions more quickly than an iterative algorithm.

Of possibly more importance, as the spread of the population \( \beta \) through the domain of action \( \alpha \) is diffuse local optima are reached simultaneously and independently. More importantly, the genetic algorithm does not get trapped on any local optimum because further improvements are being explored elsewhere. As the average fitness of the population is being increased constantly as time elapses “schemata must meet progressively higher criteria to attain (or retain) a high ranking.”\(^7\) This addresses the fourth (and last) obstacle that Holland set out to overcome: the fitness function need not change as the solution changes so that spatial and temporal measures are of reduced importance in the operation of the algorithm.

While it is a weak parallel it can be enlightening to view the action of a genetic algorithm as being similar to a well-managed diverse mutual fund. The growth (or loss) of any individual holding compounds as the interest is reinvested. The fund manager applies the fitness function of rate of return to remove less.

---

4 Holland, p. 99  
5 Holland, p. 93  
6 Holland, p. 103  
7 Holland, p. 105
fit holdings from the population and invest in new markets which are similar to ones which are performing well.

**Mutation**

While it is the most familiar of genetic operators, mutation is in practice infrequently used. It is by definition a blind enumerative plan and for that reason it is inefficient. Going back again to our example of the Traveling Salesman Problem, a plan based upon mutation is similar to taking a route and changing one city at a time at random in the hopes of finding a perfect route. This is equivalent to selecting routes one at a time from the domain of action α. This is obviously an inefficient process. Yet, in common thinking about evolution, mutation plays a larger-than-life role.

The creation of new schema is handled well by crossover when pieces of solutions are recombined in a previously unseen arrangement. The role of mutation is to allow for the possibility that a lost allele can reappear if it has been eliminated from the population β. Once an allele has been removed from the population crossover and reproduction have no way of reintroducing it. In the pure view of crossover and reproduction diffusing the population β throughout the domain of action α mutation is strictly a source of loss. This loss may be necessary, however, to escape a false peak in performance. If mutation is infrequent it does not greatly affect the performance of the population as a whole.

**A handworked example**

David Goldberg works through a simple example of a genetic algorithm in his book *Genetic Algorithms in Search, Optimization, and Machine Learning* to illustrate the basic function of genetic algorithms. His early work focused on the practical application of genetic algorithms to real-world problems. In this example he uses only the three genetic operators we have discussed so far, reproduction, crossover and mutation. While our discussion of these operators has so far been extremely general, in practice some details of the operations change to allow them to be more easily applied to the problem at hand. A simple example can help to illustrate the power of the genetic algorithm as well as set the basis for Goldberg's Schema Theorem which measures exactly the performance of his simplified genetic algorithm.

The example Goldberg uses manipulates a string of five bits (length $l = 5$) and a population of four strings ($|β| = 4$). Obviously the domain of action is $|α| = 2^5 = 32$. That is to say that there are 32 unique instances of solutions $A ⊆ α$. Likewise in this example Goldberg uses a problem which defines its own fitness function: $f(x) = x^2$. Use of a simple function allows a clear path of progression from low average fitness $\bar{f}$ to a perfect solution. Of course we realize instantly that the ideal string is $11111_{\text{ten}} = 31_{\text{ten}}$, whose square is 961. The genetic algorithm will obtain this result from a random pool of candidates.

Other fitness functions and a dynamic operation of this simple example are available on the web at http://kirsch.cs.washington.edu/pardo. The use of other fitness functions shows how the growth of the fitness of solutions varies with the amount of definition given by the function.

The first step is to determine who will be able to reproduce. We compute the decimal value and fitness of each string. For our search for a maximal square, the fitness is the value of the string squared. Other applications of genetic algorithms would supply a fitness function which accurately maps the desired goal of that algorithm to a subject-appropriate relative fitness score. The resulting fitness values are shown in the following table.

<table>
<thead>
<tr>
<th>String</th>
<th>Decimal</th>
<th>Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101</td>
<td>13</td>
<td>169</td>
</tr>
<tr>
<td>11000</td>
<td>24</td>
<td>576</td>
</tr>
<tr>
<td>01000</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>10011</td>
<td>19</td>
<td>361</td>
</tr>
</tbody>
</table>

The sum of fitness values $\Sigma f$ is 1170, leading to an average fitness value $\bar{f}$ of 293.

We now look at the fitness of each string to determine the probability of its being selected.

---

6 Goldberg, pp. 33 - 35
<table>
<thead>
<tr>
<th>String</th>
<th>Decimal</th>
<th>Fitness</th>
<th>Probability of Selection</th>
<th>Expected Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1</td>
<td>13</td>
<td>169</td>
<td>0.14</td>
<td>0.58</td>
</tr>
<tr>
<td>1 1 0 0 0</td>
<td>24</td>
<td>576</td>
<td>0.49</td>
<td>1.97</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>8</td>
<td>64</td>
<td>0.06</td>
<td>0.22</td>
</tr>
<tr>
<td>1 0 0 1 1</td>
<td>19</td>
<td>361</td>
<td>0.31</td>
<td>1.23</td>
</tr>
<tr>
<td><strong>Average fitness</strong></td>
<td></td>
<td><strong>1170 / 4 = 293</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Our first string, 01101, has a fitness of 169 in a population where the total fitness is 1170. The probability of selecting this string from this population is thus \(0.14 = \frac{169}{1170}\). As we have 4 strings from which to choose we expect to have \(0.58 = 0.14 \cdot 4\) copies of this string in the reproduction pool. This figure can also be obtained by dividing the individual fitness \(f_i = 169\) by the average fitness \(\bar{f} = 293\).

Taking probability at face value we can round off all of the expected counts to see that the string 01000 will not survive to reproduce. It is replaced by the most fit string, 11000, for which there is an expected count of almost 2. Of course there is a chance 01000 will survive to reproduce: one of the main features of genetic algorithms is that they are nondeterministic — things can happen differently each time the algorithm is run.

Now that we have identified the class of strings which are to reproduce we can examine the main operator of the genetic algorithm, crossover. We choose a mate for each string which will reproduce at random. For each pair we choose the crossover site at random. We then paste the head of one string on the tail of the other and vice versa. Our crossover will look like the following. Note that second and third strings are the same as the third string did not survive and the second was strong enough to be represented an above-average number of times.

<table>
<thead>
<tr>
<th>Mating Pool</th>
<th>Mate</th>
<th>Crossover</th>
<th>New</th>
<th>x</th>
<th>Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1</td>
<td>2</td>
<td>4</td>
<td>0 1 1 0 0</td>
<td>12</td>
<td>144</td>
</tr>
<tr>
<td>1 1 0 0 0</td>
<td>1</td>
<td>4</td>
<td>1 1 0 1 1</td>
<td>25</td>
<td>625</td>
</tr>
<tr>
<td>1 1 0 0 0</td>
<td>3</td>
<td>2</td>
<td>1 1 0 1 1</td>
<td>27</td>
<td>729</td>
</tr>
<tr>
<td>1 0 0 1 1</td>
<td>4</td>
<td>2</td>
<td>1 0 0 0 0</td>
<td>16</td>
<td>256</td>
</tr>
<tr>
<td><strong>Average fitness</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>1764 / 4 = 439</strong></td>
</tr>
</tbody>
</table>

In one period of reproduction and crossover we have increased the average fitness of the population from 293 to 439, an increase of 50%. We have increased the maximum fitness represented by 153 which is 16% of the maximum attainable fitness (a "perfect" score is 961.)

Under the process of reproduction solely the growth of a string is related directly to its fitness. It is selected to reproduce with probability \(p_i = f_i / \Sigma f_j\) where \(f_i\) is the fitness of the individual and \(\Sigma f_j\) is the total fitness of all strings in the population. The growth of a string can then be modelled as a simple difference equation

\[
m(H, t + 1) = m(H, t) \frac{f(H)}{\bar{f}}
\]

where \(m\) represents the number of string \(H\) in a population at time \(t\). This equation says simply that a string which is twice above average will be represented twice as often in the new population. Similarly, a string which is a constant amount below

Assume that some string is much better than the others by a factor of a constant, \(c\). Then the fitness of the individual is equal to \(c\) times the average fitness at each time step \(f(H) = c \bar{f}\). Thus the difference equation modelling the growth of the string can be reduced:

\[
m(H, t + 1) = m(H, t) \frac{f(H)}{\bar{f}} = m(H, t) \frac{c \bar{f}}{\bar{f}} = m(H, t) \cdot (1 + c).
\]

Starting at time zero we obtain \(m(H, t) = m(H, 0) \cdot (1 + c)^t\) which is the discrete analog of the exponential.\(^7\) Thus reproduction leads to exponentially increasing representation of fit solutions much as interest accrues.

\(^7\) Goldberg p.30
on money. Rather than rate $r$ we have the constant $c$ which is determined by the amount of definition given by the fitness function.

**Schemata**

Why is it that the string 11000 was extremely fit for reproduction whereas the string 01000 was not? Obviously having a 1 in the highest bit made for the largest number $x$ and thus the largest score from the fitness function $f(x) = x^2$. This property of having a one in the highest can be represented by the schema 10000 where the character 0 represents a wildcard or "don't care" character. The schema 00000 is obviously an unfit schema. The first string, however, survived to reproduce despite containing it. In many generations of reproduction we would expect to see the schema 00000 disappear completely. In fact the probability of it being selected for reproduction from our new population (after crossover) is only 0.08.

This is how schemata allow us to identify and assign credit to certain characteristics of the solution rather than the solution as a whole. Any class of strings beginning with 10000 would be expected to perform well. In this way rather than chasing down only highly fit solutions which may have some fatal flaw we are able to target a class of highly fit solutions. Any problem which has an area of local minima surrounding a peak or any problem with a "false maximum" peak will be easier to solve when annealing by patterns than when using a simple hillclimbing algorithm of complete solutions.

While working with schemata adds some granularity of control it also adds a layer of complexity to the system: in a binary alphabet of 0’s and 1’s there are only $2^5 = 32$ possible strings of length five. When we move to schemata and add a "don’t care" symbol of 0 we increase the possible schemata to $3^5 = 243$. We have 243 schemata to search but still only 32 possible solutions.

Schemata also have their own characteristics which influence the behaviour of an SGA. Consider a schema such as 10000. This schema, while beneficial at both specified positions, would not survive any crossover. The string length is five and the specified length of the schema itself is five. Any crossover site destroys the schema. Likewise, a schema such as 11010 would have difficulty surviving crossover because it can only survive crossover in the fourth position.

The number of positions specified in a schema is referred to as its order. The number of positions between the first and last specified positions is its defining length. The effects of genetic operators such as crossover, mutation and inversion lead to the survival of low-order schemata with short defining lengths.

This fact is shown if we look again at our initial population of 01010, 11000, 01000 and 10011. We identify three schemata to follow through the crossover: 10000, 01000 and 10000.

<table>
<thead>
<tr>
<th>Original Strings</th>
<th>Fitness $f(x) = x^2$</th>
<th>Expected Count</th>
<th>Mating Pool &amp; Crossover Population</th>
<th>New Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1</td>
<td>169</td>
<td>0.58</td>
<td>0 1 1 0 1</td>
<td>0 1 1 0 0</td>
</tr>
<tr>
<td>1 1 0 0 0</td>
<td>576</td>
<td>1.97</td>
<td>1 1 0 0 0</td>
<td>1 1 0 0 1</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>64</td>
<td>0.22</td>
<td>1 1 0 0 0</td>
<td>1 1 0 1 1</td>
</tr>
<tr>
<td>1 0 0 1 1</td>
<td>361</td>
<td>1.23</td>
<td>1 0 0 1 1</td>
<td>1 0 0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schema represented in Strings</th>
<th>Schema Average Fitness $f(H)$</th>
<th>Expected Count</th>
<th>Represented in Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>469</td>
<td>3.20</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>10000</td>
<td>320</td>
<td>2.18</td>
<td>2, 3</td>
</tr>
<tr>
<td>10000</td>
<td>576</td>
<td>1.97</td>
<td>4</td>
</tr>
</tbody>
</table>

The schemata shown are represented in the four strings a total of five times. To find each schema’s average fitness we average the fitnesses of the strings in which they appear. To find the number of copies of each schema which should be represented in the mating pool we take the ratio of the schema’s average fitness to the average population fitness and multiply by the number of copies of the schema present in the initial population. Thus for the schema 10000 with fitness 469 but present in two strings we expect to see $2 \cdot 469/293 = 3.20$ copies in the mating pool. In our example this schema is in fact represented three times in the mating pool.

Crossover, by selecting a site at random to separate a string, will favor those schemata with low order and short defining length. The first schema, with order 1, is impervious to disruption by crossover and thus we keep our expected count of 3.20 representations of this schema in the new (crossed-over) population. The
second schema, with defining length two, will only be disrupted one time in four selections of crossover sites. We thus expect to see $2.18 \cdot \frac{1}{4} = 1.64$ representations of it in the new population. The last schema, however, has the same order as the second (2) but has a defining length of five. Any crossover site will disrupt this schema. Thus we expect to see $1.97 \cdot \frac{0}{5} = 0$ copies of this schema in the crossed-over population.

Looking at the new population we see that our expectations have been satisfied in our experiment. After crossover the first schema, $\circ \circ \circ \circ$, is represented three times (in strings 2, 3 and 4.) The second schema, $\circ 1 \circ \circ \circ$, appears twice (in strings 2 and 3)—we had expected 2.18 copies. The last schema with the defining length of five appears only once in the crossed-over population while we had expected two copies. The long defining length made this last schema more likely to be destroyed by crossover.

We calculated survival probability above by looking at the number of possible crossover sites which is always $l - 1$. Given that there will be crossover of a particular schema the probability of its destruction will be the difference of its defining length, $\delta(H)$ and the number of crossover sites. The probability of its survival is then $1 - \frac{\delta(H)}{l - 1}$. If there is a given probability that a schema will be selected for crossover of $p_c$ then the probability $p_s$ of it surviving a round of reproduction with random crossover is

$$p_s \geq 1 - p_c \cdot \frac{\delta(H)}{l - 1}$$

Mutation does not appear in our handworked example because of its relative infrequency. It does occur, however, and thus can be assigned a probability $p_m$. Thus the probability that mutation will not occur is $(1 - p_m)$ for each of the fixed positions in the schema. Those schemata with greater defining length $\delta(H)$ are referred to as being of a greater order. If a schema is of order $o(H)$ then the probability that it will survive mutation is $(1 - p_m)^{o(H)}$. For small values of $p_m$ this can be approximated by $1 - o(h) \cdot p_m$.

If crossover, reproduction and mutation are independent these all work together to produce what Goldberg calls the Schema Theorem.

$$m(H, t + 1) \geq m(H, t) \cdot \frac{f(H)}{f} \cdot \left[1 - p_c \cdot \frac{\delta(H)}{l - 1} - o(H) p_m\right]$$

This theorem counts the number $m$ of representations of a schema $H$ in a population at time $t$. It takes into account the relative fitness $f(H)/f$, crossover which is a function of defining length $\delta$ of the schema and the total length $l$ of a member solution $A \subset \alpha$ and the chance of destruction by mutation as a function of the order of the schema $o(H)$.

What is most important about the Schema Theorem is that it arrives at the same conclusion as Holland's generalized operators — that solutions are given exponentially increasing (or decreasing) representation in the population as a function of their relative fitness.

Other genetic operators

Obviously Goldberg's use of only a simplified version of reproduction and crossover produces results quickly. In more complicated problems those two operators in conjunction with infrequent mutation provide enough power to obtain a solution. Holland outlines a series of genetic operators which are not as commonly used, including inversion, dominance-change, segregation and translocation, intrachromosomal duplication and deletion.

Inversion is the idea of twisting a solution in order to reduce the defining length of a member schema. For example, on our Traveling Salesman Problem we had discussed a solution which included Seattle - Portland ... Anchorage - Seattle. The problem with this schema, though it is highly fit, is that its defining length is the same length as the solution so any crossover is guaranteed to destroy it. In biology, inversion, by rearranging the alleles on a strand of DNA, protects schemata of long order by reducing their defining lengths.

While Holland proposed inversion as one of the main operators of genetic algorithms it has not been widely adopted because of its complexity. While it is simple in theory and has a direct biological parallel the position of any allele is a necessary aspect the schema's fitness. For example, a tour of twelve cities beginning with Seattle - Portland - Anchorage - Seattle ... is absurd.

In order to address this problem Holland introduces the idea of detectors. Any allele is associated with a detector so that it can carry its relative position along with it. Thus our schema of Seattle - Portland

---

8 Goldberg, p. 32
Anchorage - Seattle can be represented as Seattle₁ - Portland₂ - Anchorage₁₁ - Seattle₁₂. "Though inversion alters the linkage of schemata, it does not alter the subsets of α which they designate." Obviously, however, detectors add another layer of complexity to the implementation of the genetic algorithm.

The main role of dominance-change is to reduce the losses resulting from mutation. Dominance requires the use of a pair of alleles at each detector so that we have a double strand for each member solution \( A \subseteq α \). There is no longer a direct relation between an allele and the fitness of \( A \) in that an allele is only tested or represented when it is in a matching pair. The benefit of dominance is that in order to ensure the survival of a given allele in the presence of mutation we can reduce the mutation rate by the factor of a square. If the adaptive plan is to provide at least one copy of each allele every \( T \) generations and there are \( M \) homologous pairs of alleles the mutation rate can be reduce from \( 1/MT \) without dominance to \((1/MT)^2\) with dominance. Thus mutation can fulfill its role of assuring the presence of alleles without as much disturbance in the population through the destructive action of mutation. Obviously, again, dominance-change requires a great increase in the complexity of implementation of the genetic algorithm.

Segregation and translocation act as an exceptional sort of crossover where linkages can be created between member solutions \( A \subseteq α \). With these operations the genetic algorithm can transfer genes from one linkage set to another. The benefit of this comes when schemata from one context can potentially be useful in another context. This can be efficient when fitness depends upon many independent factors.

Intercromosomal duplication and deletion operate by providing multiple copies of alleles on the same \( n \)-tuple. These \( n \)-tuples of sets of alleles are mapped directly onto the original structures in the population. This is in concept similar to providing dominance for sets of alleles rather than individual alleles. In order to provide a mechanism for decreasing the number of copies of sets of alleles Holland introduces the idea of deletion which removes duplicates from the chromosomes.

While these other genetic operators provide increases in power when necessary the three operators discussed in detail, reproduction, crossover and mutation, provide enough power to solve complex and combinatorially explosive problems. In customizing the genetic algorithm for a particular problem, however, these ideas can be of use. With crossover, for example, the idea of detectors can be useful.

Think of a game with a variety of square tiles which have one of four designs on each of their corners. The game is played by arranging the tiles in a certain way so that no two designs are next to each other. A genetic algorithm is ideal to solve this puzzle as it mimicks likely human behavior of playing with good permutations of the positions of the tiles.

In programming a crossover for this game one cannot simply swap one set of tiles from one solution to the board for another without the possibility of deleting one type of tile and duplicating another. A solution for this problem is not a permutation of the tiles themselves but of their positions. The use of detectors can in this example allow a simple crossover to be performed safely by assigning a unique type to each of the original tiles. They can then each be identified by their detector to assure that each is represented in the final solution.

Similarly the ideas presented in this last set of genetic operators can potentially provide great increases in power for particular problems. The triad of reproduction, crossover and mutation are effective for solving complex problems but represent a poor model of computational biology. We have much more to learn about the genetic algorithm.

**Benefits of the genetic algorithm**

Genetic algorithms as a generalized technique offer the benefits of simple enumerative plans. We do not have to carefully analyze the structure of a problem before attempting to solve it. The genetic algorithm will search through the domain of action with knowledge of the solutions limited to a definition of the problem contained in the fitness function. The combinatorial explosion of solutions works for the system rather than against it, which is the case for most enumerative algorithms.

If we have special knowledge about the problem a deterministic solution may be more appropriate. A genetic algorithm to determine shortest paths would likely be less efficient than Dijkstra's algorithm. But

---

9 Holland p. 107  
10 Holland, pp. 114 - 115  
11 Holland, p. 116  
13 Holland, p. 117  

11
we cannot directly apply Dijkstra's algorithm to the Traveling Salesman Problem. And we certainly cannot apply it to problems of a tile game, image recognition or even finding the maximum of a simple algebraic function. Genetic algorithms operate without an understanding of how individual solutions are created.

While Holland originally intended for the genetic algorithm to mimic adaptation in biological evolution Goldberg and others have had great success with a vastly simplified version of the genetic algorithm. A effective generalized genetic plan can be created using only the operators of reproduction, crossover and mutation. These three offer a set of solutions whose fitness increases exponentially proportional to the definition offered by the fitness function. More power is offered, however, through the other genetic operators proposed by Holland.

We have learned a great deal about the power of genetic algorithms in the quarter century since their introduction. We have much more to learn, not only about the genetic algorithm itself but about the operation of probabilistic plans in general. We naturally attempt to build solutions by programming every step. The genetic algorithm shows us that sometimes it's more effective to step back and let guided chance discover a solution.
References


An Explanation of the Genetic Algorithm

The genetic algorithm specifies to the computer what is to be done rather than how to do it. Genetic operators function as a generalized technique which can be applied to any class of problems. For this reason they are similar in nature to simple enumerative (brute-force) searches though they exhibit the efficiency of standard problem-solving algorithms.

The genetic algorithm searches the domain of action of a given problem by evaluating all solutions in a small subset of all possible solutions. By using only three of the seven types of genetic operators proposed by Holland—reproduction, crossover, and mutation—the subset of solutions moves through the domain of action. This movement focuses on better-performing solutions which enables the genetic algorithm to identify progressively better and better solutions.

The purpose of this paper is to outline John Holland's original plan for the genetic algorithm, to illustrate its action through simplified examples and to discuss some of the underlying mathematical assumptions the genetic algorithm is built upon.

The paper is divided into six sections, listed below. This paper was typeset in \TeX{} and translated into HTML by \TeX{}-II. The original typeset paper is available as a Postscript file or as a \TeX{} source file.

Andrew Pardoe

\begin{itemize}
  \item Introduction The genetic algorithm as it compares to other algorithms and plans
  \item A simple adaptive plan Example of an adaptive algorithm
  \item Generalized genetic operators Description and examination of the most commonly known genetic operators
  \item A handworked example Reproduction and crossover worked as a dynamic example
  \item Other genetic operators Genetic operators which are not commonly known
  \item Conclusion Benefits of the genetic algorithm
  \item References References cited
\end{itemize}
Example: A two-armed bandit

Below is a simple two-armed slot machine. If you've got a nickel, try your luck and pull an arm! It's a binary slot machine so it'll pay off big if you get 1-1-1. Anything else, however, and you lose your nickel. Don't worry, the payoff rates are golden on this machine!

The arm on the left will payoff with a probability equal to $\mu_1$, the one on the right with probability equal to $\mu_2$.

Care to know the relative values of $\mu_1$ and $\mu_2$? Of course you would. That's why we've supplied a running tally of your wins and losses on each arm. Big of us, huh? Well you could click the "Cheat" button below but I wouldn't let anybody see you doing that . . .

$\mu_1$ [Pull 1 1 0] Pull $\mu_2$
You have $0.89$
$\mu_1$: Win 4 Lose 5 Total 9 $\mu = 0.44$
$\mu_2$: Win 2 Lose 4 Total 6 $\mu = 0.33$
Cheat Click to reveal $\mu_1$ 0.44 and $\mu_2$ 0.33

A simple adaptive plan

Adaptation is one of the salient features of an intelligent agent. We adapt our actions based upon our experiences constantly. Suppose, for the sake of example, that you are sitting in front of a two-armed slot machine with a pocket full of nickels ($N$ nickels, to be exact.) Each arm of the machine is labelled with that arm's probability of paying off. Unfortunately the labels read "Payoff probability: $\mu_1$" and "Payoff probability: $\mu_2$" with no explanation of what numeric values are associate with each $\mu$. How are you to maximize your profit?

Given that you will play all of your nickels (otherwise, why are you sitting in front of a slot machine?) your safest algorithm would be to split the nickels into two piles, each of size $n = N/2$. Play $n$ of the nickels on $\mu_1$ and $n$ on $\mu_2$ to obtain expected winnings of $N \cdot (\mu_1 + \mu_2) / 2$.

What if, as you play, you notice that $\mu_1 \neq \mu_2$? That is to say, you find yourself winning repeatedly from the one arm and losing repeatedly from the other. Most likely you would change your strategy and place all of your bets with the winning arm. This instinctive action leads us to a better plan: divide your trials into a first phase, one of exploration, and a second phase, of exploitation.

The method in detail is this: allocated $n < N$ nickels for exploration. Play $n/2$ nickels on each arm, keeping track of your winnings as you play. These nickels will earn the same expected winnings as the safe strategy discussed above: $n \cdot (\mu_1 + \mu_2) / 2$. Play your remaining $N-n$ nickels on the observed—and
thus assumed—better arm. Assuming that \( \mu_1 \) is the observed better arm your expected winnings in this round of exploitation are \((N-n) \cdot \mu_1\). Your total expected winnings with this strategy are the sum of these:

\[
(N-n) \cdot \mu_1 + n \cdot (\mu_1 + \mu_2)/2
\]

which reduces to \((N - \frac{n}{2}) \cdot \mu_1 + \frac{n}{2} \cdot \mu_2\). As \( n \) approaches 0 this algorithm is perfect, provided that \( \mu_1 > \mu_2 \). It is obviously more risky—with a shorter phase of experimentation you are less sure of the relative values of \( \mu_1 \) and \( \mu_2 \). As \( n \) approaches \( N \) this algorithm is optimally safe, but not as beneficial—you play fewer nickels on the perceived better arm.

Next assume you are playing with this strategy when you notice something odd. The arm which was doing so well for you suddenly starts to fail. As you have kept careful notes you easily determine that \( \mu_1 \) appears to be lower than \( \mu_2 \). You have made the wrong choice. What will you do? Obviously, switch arms. Your plan can be constantly adaptive. Essentially you can experiment and exploit simultaneously for a progressively adaptive plan.

This is the heart of the genetic algorithm. By constantly allocating a greater number of trials to those solutions which appear to be more fit we hope to maximize our gains through adaptation. We are not by any means guaranteed of success. In our example of the two-armed bandit only an infinite number of trials will establish with absolute certainty the greater of \( \mu_1 \) and \( \mu_2 \). This exhaustive plan of experimentation, however, is certainly less rewarding than constant analysis and adaptation provided that \( \mu_1 \) and \( \mu_2 \) are dissimilar.
ADAPTIVE.HTML

<html>
<HEAD>

<SCRIPT LANGUAGE="JavaScript">

var mu1, mu2, w1, w2, l1, l2, t1, t2, money;

function roundTwo(x) {
  // Rounds to two decimal places and shows three digits. As JavaScript is not
  // statically typed it defaults to different types (number/string) depending
  // upon the object's magnitude.
  x=Math.round(x * 100) / 100;
  if (x > 9) return x;
  if (x == 0) return "0.00";
  // If x is a whole number, paste on .00
  if ((x * 10) == (Math.round(x) * 10)) return x + ".00";
  // If x has one decimal place paste on another 0
  if ((x * 100) == (Math.round(x * 10) * 10)) x = x + "0";
  if ((x < 1) & (x > 0.09)) x = "0" + x;
  return x;
}

function updateTotals(){
  document.foo.w1.value = w1;
  document.foo.w2.value = w2;
  document.foo.l1.value = l1;
  document.foo.l2.value = l2;
  document.foo.t1.value = t1;
  document.foo.t2.value = t2;
  if (t1 != 0)
    document.foo.p1.value = roundTwo(w1/t1);
  if (t2 != 0)
    document.foo.p2.value = roundTwo(w2/t2);
  document.foo.money.value = "$" + roundTwo(money);
}

function clear(){
  document.foo.m1.value =
  document.foo.m2.value =
  document.foo.p1.value =
  document.foo.p2.value =
  document.foo.one.value =
  document.foo.two.value =
  document.foo.thr.value = ""
  w1 = w2 = l1 = l2 = t1 = t2 = 0;
  money = 1;
  mu1 = Math.random() * .2 + .3;
  mu2 = Math.random() * .2 + .3;
  if (Math.abs(mu1 - mu2) < .05)
    if (mu1 > mu2) {
      mu1 += .025;
      mu2 -= .025;
    } else {
      mu1 -= .025;
      mu2 += .025;
    }
  updateTotals();
}

</SCRIPT>
</HEAD>
</html>
function pause(interval){
  for (var x=0; x < 1000 * interval; x++) {}
  return;
}

function scrollA(){
  document.foo.one.value = Math.floor(Math.random() * 8) + 2;
  pause(Math.round(Math.random() * 7));
}

function scrollB(){
  document.foo.two.value = Math.floor(Math.random() * 8) + 2;
  pause(Math.round(Math.random() * 7));
}

function scrollC(){
  document.foo.thr.value = Math.floor(Math.random() * 8) + 2;
  pause(Math.round(Math.random() * 7));
}

function scrollABC(){
  scrollA(); scrollB(); scrollC();
}

function scrollACB(){
  scrollA(); scrollC(); scrollB();
}

function scrollBAC(){
  scrollB(); scrollA(); scrollC();
}

function scrollBCA(){
  scrollB(); scrollC(); scrollA();
}

function scrollCAB(){
  scrollC(); scrollA(); scrollB();
}

function scrollCBA(){
  scrollC(); scrollB(); scrollA();
}

function stop(arm){
  // Stop generates a guaranteed loss. Two is the XOR of One and Three so if
  // One and Three are both 1's then 2 is a 0. To allow for 010 we take a
  // chance at sticking this value in. We take a random toss to see if we have
  // a real winner: 111

  switch(arm){
  case 1:
    t1++;
    probability = mu1;
    break;
  case 2:
    t2++;
    probability = mu2;
  }
document.foo.one.value = Math.round(Math.random());
document.foo.two.value = Math.round(Math.random());
document.foo.thr.value = document.foo.one.value ^ document.foo.thr.value;
if (document.foo.one.value + document.foo.thr.value == 0)
document.foo.two.value = Math.round(Math.random());

// 111 is awarded proportional to the probability of the button pressed
if (Math.floor(Math.random() + probability) == 1) {
  money += .05;
  switch(arm){
  case 1:
    w1++;
    break;
  case 2:
    w2++;
  }
document.foo.one.value = document.foo.two.value =
document.foo.thr.value = 1;
} else {
  money -= .05;
  switch(arm){
  case 1:
    l1++;
    break;
  case 2:
    l2++;
  }
}

updateTotals();

function play(arm){
  if (money <= 0) return;
  var y, z;
  z = Math.round(Math.random() * 5) + 7;
  for (var x = 0; x < z; x++) {
    y = Math.floor(Math.random() * 6);
    switch (y) {
    case 0:
      scrollABC();
      break;
    case 1:
      scrollACB();
      break;
    case 2:
      scrollBAC();
      break;
    case 3:
      scrollBCA();
      break;
    case 4:
      scrollCAB();
      break;
    case 5:
      scrollCBA();
      break;
    }
function showMu(){
    document.foo.mu1.value = roundTwo(mu1);
    document.foo.mu2.value = roundTwo(mu2);
}

</SCRIPT>
<BODY onLoad="clear()">

<TITLE>A simple adaptive plan</TITLE>
<meta name="GENERATOR" content="ThH 2.79">

<h3>Example: A two-armed bandit</h3>

Below is a simple two-armed slot machine. If you've got a nickel, try your luck and pull an arm! It's a binary slot machine so it'll pay off big if you get 1-1-1. Anything else, however, and you lose your nickel. Don't worry, the payoff rates are golden on this machine!

<p>The arm on the left will payoff with a probability equal to <font face="symbol">m</font><sub>1</sub>, the one on the right with probability equal to <font face="symbol">m</font><sub>2</sub>. Care to know the relative values of <font face="symbol">m</font><sub>1</sub> and <font face="symbol">m</font><sub>2</sub>? Of course you would. That's why we've supplied a running tally of your wins and losses on each arm. Big of us, huh? Well you <i>could</i> click the "Cheat" button below but I wouldn't let anybody see you doing that . . .

<FORM NAME = "foo">

<TABLE>
    <TR><TD><font face="symbol">m</font><sub>1</sub></TD>
    <TD><INPUT TYPE="button" NAME="go1" Value="Pull" onClick="play(1)"/></TD>
    <TD><INPUT TYPE=text NAME=one SIZE=2/></TD>
    <TD><INPUT TYPE=text NAME=two SIZE=2/></TD>
    <TD><INPUT TYPE=text NAME=thr SIZE=2/></TD>
    <TD><INPUT TYPE="button" NAME="go2" Value="Pull" onClick="play(2)"/></TD>
    <TD><font face="symbol">m</font><sub>2</sub></TD>
</TR>
</TABLE>

<TABLE>
    <TR><TD><font face="symbol">m</font><sub>1</sub> have</TD>
    <TD><INPUT TYPE=text NAME=money SIZE=5/></TD>
</TR>
</TABLE>

<TABLE>
    <TR><TD><font face="symbol">m</font><sub>1</sub> Win</TD>
    <TD><INPUT TYPE=text NAME=w1 SIZE=2/></TD>
    <TD>Loss</TD>
    <TD><INPUT TYPE=text NAME=l1 SIZE=2/></TD>
    <TD>Total</TD>
    <TD><INPUT TYPE=text NAME=t1 SIZE=2/></TD>
    <TD><font face="symbol">m</font><sub>2</sub> ?</TD>
    <TD><INPUT TYPE=text NAME=p1 SIZE=4/></TD>
    <TD><font face="symbol">m</font><sub>2</sub> Win</TD>
</TR>
</TABLE>
Adaptation is one of the salient features of an intelligent agent. We adapt our actions based upon our experiences constantly. Suppose, for the sake of example, that you are sitting in front of a two-armed slot machine with a pocket full of nickels (input $N$ nickels, to be exact.) Each arm of the machine is labelled with that arm’s probability of paying off. Unfortunately the labels read ‘‘Payoff probability:’’ and ‘‘Payoff probability:’’, with no explanation of what numeric values are associated with each ‘‘Payoff probability:’’. How are you to maximize your profit?

Given that you will play all of your nickels (otherwise, why are you sitting in front of a slot machine?) your safest algorithm would be to split the nickels into two piles, each of size $<i>N</i>$ / 2. Play $<i>n</i>$ of the nickels on ‘‘Payoff probability:’’ and $<i>n</i>$ on ‘‘Payoff probability:’’ to obtain expected winnings of $<i>N</i>$ / 2. (‘‘Payoff probability:’’ + ‘‘Payoff probability:’’) / 2.

What if, as you play, you notice that ‘‘Payoff probability:’’? That is to say, you find yourself winning repeatedly from the one arm and losing repeatedly from the other. Most likely you would change your strategy and place all of your bets with the winning arm. This instinctive action leads us to a better plan: divide your trials into a first phase, one of exploration, and a second phase, of exploitation.

The method in detail is this: allocated $<i>N</i>$ nickels for exploration. Play $<i>n</i>$ / 2 nickels on each arm, keeping track of your winnings as you play. These nickels will earn the same expected winnings as the safe strategy discussed above: $<i>N</i>$ / 2. (‘‘Payoff probability:’’ + ‘‘Payoff probability:’’). Play your remaining $<i>N</i>$ nickels on the observed $<i>N</i>$ such assumed better arm. Assuming that ‘‘Payoff probability:’’ is the observed better arm your expected winnings in this round of exploitation are ($<i>N</i>$ / 2). Your total expected winnings with this strategy are the sum of these: $<i>N</i>$ / 2. (‘‘Payoff probability:’’ $<i>N</i>$ / 2 + $<i>N</i>$ / 2).
Next assume you are playing with this strategy when you notice something odd. The arm which was doing so well for you suddenly starts to fail. As you have kept careful notes you easily determine that a shorter phase of experimentation you are less sure of the relative values of the algorithm is optimally safe, but not as beneficial as you play fewer nickels on the perceived better arm.

This is the heart of the genetic algorithm. By constantly allocating a greater number of trials to those solutions which appear to be more fit we hope to maximize our gains through adaptation. We are not by any means guaranteed of success. In our example of the two-armed bandit only an infinite number of trials will establish with absolute certainty the greater of and . This exhaustive plan of experimentation, however, is certainly less rewarding than constant analysis and adaptation provided that and are dissimilar.
A handworked example

David Goldberg works through a simple example of a genetic algorithm in his book *Genetic Algorithms in Search, Optimization, and Machine Learning* to illustrate the basic function of genetic algorithms. This early work focused on the practical application of genetic algorithms to real-world problems. In this example he uses only the three genetic operators we have discussed so far, reproduction, crossover and mutation. While our discussion of these operators has so far been extremely general, in practice some details of the operations change to allow them to be more easily applied to the problem at hand. A simple example can help to illustrate the power of the genetic algorithm as well as set the basis for Goldberg's Schema Theorem which measures exactly the performance of his simplified genetic algorithm.

The example Goldberg uses manipulates a string of five bits (length \( l = 5 \)) and a population of four strings (\( |\beta| = 4 \)). Obviously the size of the domain of action is \( |\alpha| = 2^5 = 32 \). That is to say that there are 32 unique instances of solutions \( \alpha \subset \alpha \). Likewise in this example Goldberg uses a problem which defines its own fitness function: \( f(x) = x^2 \). Use of a simple function allows a clear path of progression from low average fitness \( \bar{\mu} \) to a perfect solution. Of course we realize instantly that the ideal string is \( 1111_{\text{two}} = 31_{\text{ten}} \), whose square is 961. The genetic algorithm obtains this solution by searching a sample space starting with a random pool of candidates.

In our dynamic example we have four fitness functions, each of which grows at a characteristically different rate. By working the example with different fitness functions one can see that the amount of distinction conferred by the fitness function directly affects the rate of growth of representation for schema in the population. Goldberg shows that the representation of a schema grows at a rate of \( e^{ct} \), similar to the rate of growth we see in interest functions, \( e^{rt} \). Instead of having an interest rate \( r \) we use a constant \( c \) which is proportional to the amount of distinction given by the fitness function used. This point emphasizes again that while a fitness function must be accurate one should select a function to be as distinguishing as possible.

First, select a fitness function by clicking a radio button below. This function will be used to determine the relative fitness of strings in the population.

Next, create a population by clicking first button below. This population is a random set of strings five bits long. If you prefer you may hand-enter a set of strings or change strings which are there. When you finish editing the population hit the second button. Either button will calculate decimal values for the strings in the population. Note that if you make a mistake in entering your population (ie, use a character which is not a bit or use more or less than five bits per string) a population will be generated for you automatically.

Lastly, click the third button to apply the fitness function and calculate the probability of a string being selected in the reproduction pool. If you like you can change the fitness function and recalculate the fitmesses.

- \( f(x) = x \cdot \log_2(x) \)
- \( f(x) = 2x \)
- \( f(x) = x^2 \)
- \( f(x) = 2^x \)
<table>
<thead>
<tr>
<th>String</th>
<th>Decimal Value</th>
<th>Fitness Value</th>
<th>Selection Probability</th>
<th>Expected Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>10011</td>
<td>19</td>
<td>361</td>
<td>0.17</td>
<td>0.68</td>
</tr>
<tr>
<td>11110</td>
<td>30</td>
<td>900</td>
<td>0.42</td>
<td>1.68</td>
</tr>
<tr>
<td>00110</td>
<td>6</td>
<td>36</td>
<td>0.02</td>
<td>0.07</td>
</tr>
<tr>
<td>11101</td>
<td>29</td>
<td>841</td>
<td>0.39</td>
<td>1.57</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>534.5</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **OK** Automatically generate random population
- **OK** OR Calculate decimal with entered population
- **OK** THEN Apply fitness values and calculate expected count

Note that the reproduction population is not done exactly as Holland specified in the generalized reproduction operator above. Holland acts on one member of the population at a time for reproduction. In this example it is far easier to handle the entire population as a whole. We already need to apply the fitness function in order to select a member probabilistically. Rather than select one member of the population we instead generate a new entire population to which we will apply the other genetic operators. This method creates a clear division between generations which is not faithful to the biological model.

The selection probability for each member of the population represents the probability that this member will appear in the reproduction pool. It is equal to the ratio of an individual member's fitness to the total fitness of the population. The sum of all selection probabilities will sum to one.

The expected count column lists the amount of representation we expect each member to have in the reproduction pool. It is equal to the ratio of a member's fitness to the average fitness of the population. Obviously the sum of all expected counts will sum to the size of the population.

We can now create our mating pool which is the set of members selected probabilistically from the original population. If a member had a selection probability of 2.00 we would expect to see two copies of it in the mating pool. Click on the button labelled "Create mating pool" to do so. Again, if you wish to edit the strings by hand you may do so. (If you get "undefined" entries in your populations make certain that the original population in the table above has been created and rated for fitness.)

<table>
<thead>
<tr>
<th>Mating Pool</th>
<th>Mate &amp; Crossover (random)</th>
<th>Crossover Site</th>
<th>New Crossover Population</th>
<th>Decimal Value</th>
<th>Fitness Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110</td>
<td>3</td>
<td>2</td>
<td>11011</td>
<td>27</td>
<td>729</td>
</tr>
<tr>
<td>11101</td>
<td>4</td>
<td>3</td>
<td>11101</td>
<td>29</td>
<td>841</td>
</tr>
<tr>
<td>10011</td>
<td>1</td>
<td>2</td>
<td>10110</td>
<td>22</td>
<td>484</td>
</tr>
<tr>
<td>11101</td>
<td>2</td>
<td>3</td>
<td>11101</td>
<td>29</td>
<td>841</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>723.75</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **OK** Create mating pool by selecting probabilistically from original population
- **OK** THEN Find mate for each member and pick crossover sites
A handworked example

- **OK** AND Create new population from the crossed-over old population

Now that we have identified the class of strings which are to reproduce we can examine the action of the main operator of the genetic algorithm, crossover. Clicking the second button will choose a mate at random and a crossover site for the pair. The third button will then paste the head of one string onto the tail of its mate and vice versa. This button will also show the new decimal values, their relative fitnesses, the new average fitness and the difference.

With a high probability there is a relatively large positive number in the box labelled `Difference`. (If there's not, you most likely didn't click the buttons hard enough. Go back and try again.) In only one generation the average fitness of the population changed dramatically with the use of only two genetic operators, reproduction and selection. If the process were continued we would obviously focus on strings which are the better performers in the population as the weaker strings have been removed.

Under the process of reproduction solely the growth of a string is related directly to its fitness. It is selected to reproduce with probability \( p_i = \frac{f_i}{\Sigma f_j} \) where \( f_i \) is the fitness of the individual and \( \Sigma f_j \) is the total fitness of all strings in the population. The growth of a string can then be modelled as a simple difference equation

\[
m(H, t+1) = m(H, t) \cdot \frac{f(H)}{\bar{f}}
\]

where \( m \) represents the number of string \( H \) in a population at time \( t \). This equation says simply that a string which is twice above average will be represented twice as often in the new population. Similarly, a string which is a constant amount below

Assume that some string is much better than the others by a factor of a constant, \( c \). Then the fitness of the individual is equal to \( c \) times the average fitness at each time step \( t \): \( f(H) = c \bar{f} \). Thus the difference equation modelling the growth of the string can be reduced:

\[
m(H, t+1) = m(H, t) \cdot \frac{f(H)}{\bar{f}} = m(H, t) \cdot \frac{\bar{f} + c \bar{f}}{\bar{f}} = m(H, t) \cdot (1+c).
\]

Starting at time zero we obtain \( m(H, t) = m(H, 0) \cdot (1+c)^t \) which is the discrete analog of the exponential. Thus reproduction leads to exponentially increasing representation of fit solutions much as interest accrues on money. Rather than rate \( r \) we have the constant \( c \) which is determined by the amount of definition given by the fitness function.

- **Schema in the handworked example**
<table>
<thead>
<tr>
<th>String</th>
<th>Schema in Strings</th>
<th>Fitness</th>
<th>Expected Count</th>
<th>Crossover Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110</td>
<td>111*</td>
<td>2</td>
<td>900</td>
<td>1.68</td>
</tr>
<tr>
<td>11101</td>
<td>111**</td>
<td>2 4</td>
<td>870.5</td>
<td>1.63</td>
</tr>
</tbody>
</table>

Two of the better-performing schema from our above example. The above chart shows two strings from the original population and schema which are represented in those strings. The other strings they appear in, if any, are listed in the third column.

The average fitness column lists the average fitness of the schema's represented strings. For example, if the schema appears in strings 2 and 4 this column lists the average fitness of strings 2 and 4. From this we can calculate an expected count in the crossed-over population. The last column lists the schema's actual representation in the crossed-over population.

Why is it that the some strings were extremely fit for reproduction whereas others were not? Obviously having a 1 in the highest bit helps to make for the largest number \( x \) and thus the largest score from the fitness function \( f(x) = x^2 \). This property of having a one in the highest can be represented by the schema 1 2 2 2 where the character 2 represents a wildcard or "don't care" character. The schema 0 2 2 2 is obviously an unfit schema. The first string, however, survived to reproduce despite containing it. In many generations of reproduction we would expect to see the schema 0 2 2 2 disappear completely.

This is how schemata allow us to identify and assign credit to certain characteristics of the solution rather than the solution as a whole. Any class of strings beginning with 1 2 2 2 would be expected to perform well. In this way rather than chasing down only highly fit solutions which may have some fatal flaw we are able to target a class of highly fit solutions. Any problem which has an area of local minima surrounding a peak or any problem with a "false maximum" peak will be easier to solve when annealing by patterns than when using a simple hillclimbing algorithm of complete solutions.

While working with schemata adds some granularity of control it also adds a layer of complexity to the system: in a binary alphabet of 0's and 1's there are only \( 2^5 = 32 \) possible strings of length five. When we move to schemata and add a "don't care" symbol of 2 we increase the possible schemata to \( 3^5 = 243 \). We have 243 schemata to search but still only 32 possible solutions.

Schemata also have their own characteristics which influence the behaviour of an SGA. Consider a schema such as 1 2 2 1. This schema, while beneficial at both specified positions, would not survive any crossover. The string length is five and the specified length of the schema itself is five. Any crossover site destroys the schema. Likewise, a schema such as 1 1 0 1 2 would have difficulty surviving crossover because it can only survive crossover in the fourth position.

The number of positions specified in a schema is referred to as its order. The number of positions between the first and last specified positions is its defining length. The effects of genetic operators such as crossover, mutation and inversion lead to the survival of low-order schemata with short defining lengths.

We calculate the survival probability by looking at the number of possible crossover sites which is always \( l - 1 \). Given that there will be crossover of a particular schema the probability of its destruction will be the difference of its defining length, \( \delta(H) \) and the number of crossover sites. The probability of
its survival is then $1-\delta(H)/(l-1)$. If there is a given probability that a schema will be selected for crossover of $p_c$, then the probability $p_s$ of it surviving a round of reproduction with random crossover is

$$p_s \geq 1-p_c \cdot \frac{\delta(H)}{l-1}$$

Mutation does not appear in our handworked example because of its relative infrequency. It does occur, however, and thus can be assigned a probability $p_m$. Thus the probability that mutation will not occur is $(1-p_m)$ for each of the fixed positions in the schema. Those schemata with a greater defining length $\delta(H)$ are referred to as being of greater order. If a schema is of order $o(H)$ then the probability that it will survive mutation is $(1-p_m)^{o(H)}$. For small values of $p_m$ this can be approximated by $1-o(h)p_m$.

If crossover, reproduction and mutation are independent these all work together to produce what Goldberg calls the Schema Theorem.

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{f} \cdot \left[1-p_c \cdot \frac{\delta(H)}{l-1} - o(H)p_m\right]$$

This theorem counts the number $m$ of representations of a schema $H$ in a population at time $t$. It takes into account the relative fitness $f(H)/[\sum f]$, crossover which is a function of defining length $\delta$ of the schema and the total length $l$ of a member solution $A \subset \alpha$ and the chance of destruction by mutation as a function of the order of the schema $o(H)$. What is most important about the Schema Theorem is that it arrives at the same conclusion as Holland's generalized operators that solutions are given exponentially increasing (or decreasing) representation in the population as a function of their relative fitness.

Abstract • Introduction • A simple adaptive plan • Generalized genetic operators • Other genetic operators • Conclusion • References

Footnotes:

6 Goldberg, pp. 33 - 35

7 Goldberg, p. 30

8 Goldberg, p. 32
function clearForms(x) {
    // Clears forms down to form number x
    x--; for (var i=3; i > x; i--)
        document.forms[i].reset();
}

function roundTwo(x) {
    // Rounds to two decimal places and shows three digits. As JavaScript is not
    // statically typed it defaults to different types (number/string) depending
    // upon the object's magnitude.
    x=Math.round(x * 100) / 100;
    if (x > 9) return x;
    if (x == 0) return "0.00";
    // If x is a whole number, paste on .00
    if ((x * 10) == (Math.round(x * 10) * 10)) return x + ".00";
    // If x has one decimal place paste on another 0
    if ((x * 100) == (Math.round(x * 100) * 10)) x = x + "0";
    if ((x < 1) & & (x > 0.09)) x = "0" + x;
    return x;
}

function fitness(x) {
    // Fitness function switches based upon value of radio button
    if (document.functions.func[0].checked) return roundTwo(Math.log(x) * Math.LOG2E * x);
    if (document.functions.func[1].checked) return x * 2;
    if (document.functions.func[2].checked) return Math.pow(x, 2);
    if (document.functions.func[3].checked) return Math.pow(2, x);
}

function leadingZeros(x) {
    // Adds leading zeros to a string/number to make five digits
    if (x < 9999) x = "0" + x;
    if (x < 999) x = "0" + x;
    if (x < 99) x = "0" + x;
    if (x < 9) x = "0" + x;
    return x;
}

function randomString() {
    // Generates a random five-bit binary number then pads the number/string
    var bob=0;
    for (i=0; i < 5; bob += (Math.round(Math.random())) * Math.pow(10, i), i++)
        return leadingZeros(bob);
}

function createPopulation() {
    // Creates new random population
    clearForms(2);
    document.foo.pop1.value = randomString();
    document.foo.pop2.value = randomString();
    document.foo.pop3.value = randomString();
document.foo.pop4.value = randomString();
convertDecimal(document.foo);
}

function checkPopulation() {
// Verifies validity of user-input population. Concatenates all of the
// population members in populationIsValid() and then either converts
// the members to decimal or creates the population automatically

var checkString;

checkString = document.foo.pop1.value + document.foo.pop2.value +
document.foo.pop3.value + document.foo.pop4.value;

if ( populationIsValid(checkString) )
    convertDecimal(document.foo);
else
    createPopulation();
}

function populationIsValid(checkString) {
// Verifies validity of user-input population. Accepts all members of the
// population in one string then verifies that it contains 20 characters
// and that it does not match a regular expression which contains one or
// more characters which are not 0 or 1.

var notZeroOne = /[01]/;

if ( (checkString.length == 20) && !(notZeroOne.test(checkString)) )
    return true;
else
    return false;
}

function convertDecimal(form) {
// Converts population to decimal values

form.dec1.value = parseInt(form.pop1.value, 2);
form.dec2.value = parseInt(form.pop2.value, 2);
form.dec3.value = parseInt(form.pop3.value, 2);
form.dec4.value = parseInt(form.pop4.value, 2);

if (form == document.foo) clearFitness(form);
}

function clearFitness(form) {
// Clears fitness, selection probability and expected count values

form.fit1.value = form.fit2.value = form.fit3.value = form.fit4.value =
form.sel1.value = form.sel2.value = form.sel3.value = form.sel4.value =
form.cnt1.value = form.cnt2.value = form.cnt3.value = form.cnt4.value =
form.avg.value = "";
}

function applyFitness(form) {
// Applies the fitness function to the population

form.fit1.value = fitness(form.dec1.value);
form.fit2.value = fitness(form.dec2.value);
form.fit3.value = fitness(form.dec3.value);
form.fit4.value = fitness(form.dec4.value);

if (form == document.foo) probSelect();
function probSelect() {
    // probabilities of selection of each member
    var one = parseInt(document.foo.fit1.value);
    var two = parseInt(document.foo.fit2.value);
    var three = parseInt(document.foo.fit3.value);
    var four = parseInt(document.foo.fit4.value);
    var totalFitness = one + two + three + four;
    document.foo.sel1.value = roundTwo(one / totalFitness);
    document.foo.sel2.value = roundTwo(two / totalFitness);
    document.foo.sel3.value = roundTwo(three / totalFitness);
    document.foo.sel4.value = roundTwo(four / totalFitness);
    showCount(one, two, three, four, totalFitness);
}

function showCount(one, two, three, four, totalFitness) {
    // Calculates expected count of variables in reproducing population
    var averageFitness = totalFitness / 4;
    document.foo.cnt1.value = roundTwo(one / averageFitness);
    document.foo.cnt2.value = roundTwo(two / averageFitness);
    document.foo.cnt3.value = roundTwo(three / averageFitness);
    document.foo.cnt4.value = roundTwo(four / averageFitness);
    calculateAverage(document.foo);
}

function calculateAverage(form) {
    // Calculates average fitness for last row on each form
    var thisTotal = 0;
    thisTotal += parseInt(form.fit1.value);
    thisTotal += parseInt(form.fit2.value);
    thisTotal += parseInt(form.fit3.value);
    thisTotal += parseInt(form.fit4.value);
    form.avg.value = roundTwo(thisTotal / 4);
}

/* Second table's functions follow */

function showPool() {
    // Creates reproduction pool by pushing number 400 population members into
    // a stack. From the 400 four are randomly chosen. Choice is done with
    // replacement but this is negligible for 4 out of 400.
    var stack = [];
    for (i = 0; i < (document.foo.cnt1.value * 100); i++)
        stack.push(document.foo.pop1.value);
    for (i = 0; i < (document.foo.cnt2.value * 100); i++)
        stack.push(document.foo.pop2.value);
    for (i = 0; i < (document.foo.cnt3.value * 100); i++)
        stack.push(document.foo.pop3.value);
    for (i = 0; i < (document.foo.cnt4.value * 100); i++)
        stack.push(document.foo.pop4.value);
    document.bar.pool1.value = stack[Math.floor(400 * Math.random())];
    document.bar.pool2.value = stack[Math.floor(400 * Math.random())];
    document.bar.pool3.value = stack[Math.floor(400 * Math.random())];
    document.bar.pool4.value = stack[Math.floor(400 * Math.random())];
delete stack;
}

function chooseMatesAndSite() {
    // We choose the mate for #4 by picking a random number from 1 to 3.
    // In our simple population of four know the other three mates.
// Crossover site is assigned for each pair from two random sites.

var checkString;
checkString = document.bar.pool1.value + document.bar.pool2.value +
             document.bar.pool3.value + document.bar.pool4.value;

if (! ( populationIsValid(checkString) ) )
    showPool();

var mates = [];
var sites = [];
X1 = Math.floor(4 * Math.random()) + 1;
X2 = Math.floor(4 * Math.random()) + 1;
i = Math.floor(3 * Math.random()) + 1;
mates.push(i);
switch(i) {
    case 1:
        mates.push("2"); mates.push("3"); mates.push("4");
        sites.push(X1); sites.push(X2); sites.push(X2); sites.push(X1);
        break;
    case 2:
        mates.push("1"); mates.push("4"); mates.push("3");
        sites.push(X1); sites.push(X2); sites.push(X1); sites.push(X2);
        break;
    case 3:
        mates.push("4"); mates.push("1"); mates.push("2");
        sites.push(X1); sites.push(X1); sites.push(X2); sites.push(X2);
        break;
}
document.bar.mate1.value = mates.pop();
document.bar.mate2.value = mates.pop();
document.bar.mate3.value = mates.pop();
document.bar.mate4.value = mates.pop();
document.bar.site1.value = sites.pop();
document.bar.site2.value = sites.pop();
document.bar.site3.value = sites.pop();
document.bar.site4.value = sites.pop();
clearNewPopulation(document.bar);
}

function clearNewPopulation(form) {
    form.pop1.value = form.pop2.value = form.pop3.value = form.pop4.value =
    form.decl.value = form.dec2.value = form.dec3.value = form.dec4.value =
    form.fit1.value = form.fit2.value = form.fit3.value = form.fit4.value =
    form.avg.value = form.dif.value = "";
}

function newPopulation() {
    // Push old population into an array so we can reference it with index.
    var oldPop = [];
    oldPop.push(document.bar.pool1.value);
    oldPop.push(document.bar.pool2.value);
    oldPop.push(document.bar.pool3.value);
    oldPop.push(document.bar.pool4.value);

    splice = Math.pow(10, 5 - document.bar.site1.value);
document.bar.pop1.value = leading2Zeros(
    Math.floor(oldPop[0] / splice) * splice
    + oldPop[document.bar.mate1.value - 1] % splice);

    splice = Math.pow(10, 5 - document.bar.site2.value);
document.bar.pop2.value = leadingZeros(
    + oldPop[document.bar.mate2.value - 1] % splice);
splice = Math.pow(10, 5 - document.bar.site3.value);
document.bar.pop3.value = leadingZeros(
    + oldPop[document.bar.mate3.value - 1] % splice);
splice = Math.pow(10, 5 - document.bar.site4.value);
document.bar.pop4.value = leadingZeros(
    + oldPop[document.bar.mate4.value - 1] % splice);
convertDecimal(document.bar);
applyFitness(document.bar);
calculateAverage(document.bar);
fitnessDifference();
}

function fitnessDifference() {
    document.bar.dif.value = document.bar.avg.value - document.foo.avg.value;
    findBest();
}

function findBest(){
    // Finds the best two schema of the original population and their
    // representation in the populations

    var pattern1, pattern2;
    var all = new Array(4);
    all.push(document.foo.pop1.value);
    all.push(document.foo.pop2.value);
    all.push(document.foo.pop3.value);
    all.push(document.foo.pop4.value);
    all.sort();
document.frob.best1.value = all[3];
document.frob.best2.value = all[2];
pattern1 = findSchema(all[3]);
pattern2 = findSchema(all[2]);
if (pattern1 == pattern2) {
    document.frob.best2.value = all[1];
    pattern2 = findSchema(all[1]);
}
if (pattern1 == pattern2) {
    document.frob.best2.value = all[0];
    pattern2 = findSchema(all[0]);
}
delete all;
document.frob.schn1.value = putStars(pattern1);
document.frob.schn2.value = putStars(pattern2);
document.frob.orig1.value = representation(document.foo, pattern1);
document.frob.orig2.value = representation(document.foo, pattern2);
document.frob.cros1.value = representation(document.bar, pattern1);
document.frob.cros2.value = representation(document.bar, pattern2);
document.frob.ofit1.value = schemaFitness(document.foo,
    document.frob.orig1.value);
document.frob.ofit2.value = schemaFitness(document.foo,
    document.frob.orig2.value);
document.frob.expcl.value = roundTwo(document.frob.ofit1.value /
    document.foo.avg.value);
document.frob.expcl2.value = roundTwo(document.frob.ofit2.value /
    document.foo.avg.value);
}

function schemaFitness(whichForm, strings){
    // Calculates the fitness of a schema by averaging its strings' fitness
    var count = 0;
    var total = 0;

    if (strings.match(/1/) ) {
        count ++;
        total += parseInt(whichForm.fit1.value);
    }
    if (strings.match(/2/) ) {
        count ++;
        total += parseInt(whichForm.fit2.value);
    }
    if (strings.match(/3/) ) {
        count ++;
        total += parseInt(whichForm.fit3.value);
    }
    if (strings.match(/4/) ) {
        count ++;
        total += parseInt(whichForm.fit4.value);
    }
    return (total / count);
}

function representation(whichForm, pattern){
    var retval = "";

    if (pattern.test(whichForm.pop1.value)) retval += "1 ";
    if (pattern.test(whichForm.pop2.value)) retval += "2 ";
    if (pattern.test(whichForm.pop3.value)) retval += "3 ";
    if (pattern.test(whichForm.pop4.value)) retval += "4 ";

    return retval;
}

function putStars(word){
    // Replace doesn't work on regexps. This handles the replace function
    // manually by inspecting the regexp character by character. We then
    // don't have to search on terms like \d which are special in regexps.
    var wordStr = new String(word);
    var newWord = "";
    for (var i = 0; i < wordStr.length; i++) {
        if (wordStr.charAt(i) == "1") newWord += "1";
        if (wordStr.charAt(i) == "\d") newWord += "\";
    }
    return newWord;
}

function findSchema(member){
    // Finds the schema associated with a member. In a two-bit population the
    // schema is merely the presence of a highly-placed set of ones. This

// function by no means tests for all of the 200 possible schema.

var pattern;

pattern = /11111/;
if (pattern.test(member))
  return pattern;

pattern = /1111\d/;
if (pattern.test(member))
  return pattern;

pattern = /111\d\d/;
if (pattern.test(member))
  return pattern;

pattern = /11\d\d\d/;
if (pattern.test(member))
  return pattern;

pattern = /1\d1\d\d/;
if (pattern.test(member))
  return pattern;

pattern = /1\d\d\d\d/;
if (pattern.test(member))
  return pattern;

pattern = /\d1111/;
if (pattern.test(member))
  return pattern;

pattern = /\d111\d/;
if (pattern.test(member))
  return pattern;

pattern = /\d11\d\d/;
if (pattern.test(member))
  return pattern;

pattern = /\d1\d\d\d/;
if (pattern.test(member))
  return pattern;

pattern = /\d\d111/;
if (pattern.test(member))
  return pattern;

pattern = /\d\d11\d/;
if (pattern.test(member))
  return pattern;

pattern = /\d\d\1\d/;
if (pattern.test(member))
  return pattern;

pattern = /\d\d\d\1\d/;
if (pattern.test(member))
  return pattern;
if (pattern.test(member))
    return pattern;

pattern = /\d\d\d\d\d/;
if (pattern.test(member))
    return pattern;
}

</SCRIPT>

<TITLE>A handworked example</TITLE>
</HEAD>
<BODY onLoad="clearForms(0)">

<h2>A handworked example</h2>

<p>David Goldberg works through a simple example of a genetic algorithm in
his book &lt;i&gt;Genetic Algorithms in Search, Optimization, and Machine
Learning&lt;/i&gt; to illustrate the basic function of genetic algorithms. &lt;a
href="#thtPrefABB" name="thtPrefABB"&gt;&lt;sup&gt;6&lt;/sup&gt;&lt;/a&gt; This early work focused
on the practical application of genetic algorithms to real-world problems. In
this example he uses only the three genetic operators we have discussed so
far, reproduction, crossover and mutation. While our discussion of these
operators has so far been extremely general, in practice some details of the
operations change to allow them to be more easily applied to the problem at
hand. A simple example can help to illustrate the power of the genetic
algorithm as well as set the basis for Goldberg's Schema Theorem which
measures exactly the performance of his simplified genetic algorithm.</p>

<p>The example Goldberg uses manipulates a string of five bits (length
&lt;i&gt;n&lt;/i&gt; = 5) and a population of four strings (&lt;font
face="symbol">b&lt;/font&gt; &lt;font
face="symbol">c&lt;/font&gt;) = 4). Obviously the size of the domain of action is
&lt;font face="symbol">a &lt;/font&gt; &lt;font
face="symbol">b &lt;/font&gt; &lt;font
face="symbol">c &lt;/font&gt; = 2&lt;sup&gt;5&lt;/sup&gt; = 32. That is to say that there are
32 unique instances of solutions &lt;i&gt;&lt;/i&gt;.&lt;i&gt;&lt;font face="symbol">a &lt;/font&gt; &lt;font
face="symbol">b &lt;/font&gt; &lt;font
face="symbol">c &lt;/font&gt; = 1&lt;sup&gt;5&lt;/sup&gt;&lt;/i&gt;.&lt;i&gt; Likewise in this example Goldberg uses a problem
which defines its own fitness function: &lt;i&gt;f&lt;/i&gt;(&lt;i&gt;x&lt;/i&gt;)&lt;i&gt; =
1&lt;i&gt;x&lt;/i&gt;.&lt;i&gt;&lt;sup&gt;2&lt;/sup&gt;&lt;i&gt;.&lt;i&gt; Use of a simple function allows a clear path of
progression from low average fitness &lt;font face="symbol">a &lt;/font&gt; &lt;font
face="symbol">b &lt;/font&gt; &lt;font
face="symbol">c &lt;/font&gt; to a perfect solution. Of course we realize instantly
that the ideal string is 11111&lt;i&gt;&lt;sub&gt;1&lt;/sub&gt;&lt;i&gt;.&lt;i&gt;two&lt;/i&gt;&lt;/sub&gt; =
31&lt;i&gt;&lt;sub&gt;1&lt;/sub&gt;&lt;i&gt;ten&lt;/i&gt;&lt;/sub&gt;, whose square is 961. The genetic algorithm obtains
this solution by searching a sample space starting with a random pool of
candidates</p>

<p>In our dynamic example we have four fitness functions, each of which
grows at a characteristically different rate. By working the example with
different fitness functions one can see that the amount of distinction
conferred by the fitness function directly affects the rate of growth of
representation for schema in the population. Goldberg shows that the
representation of a schema grows at a rate of &lt;i&gt;c&lt;sup&gt;ct&lt;/sup&gt;&lt;/i&gt;, similar
to the rate of growth we see in interest functions, &lt;i&gt;c&lt;sup&gt;rt&lt;/sup&gt;&lt;/i&gt;.
Instead of having an interest rate &lt;i&gt;r&lt;/i&gt; we use a constant &lt;i&gt;c&lt;/i&gt; which
is proportional to the amount of distinction given by the fitness function
used. This point emphasizes again that a while a fitness function must be
accurate one should select a function to be as distinguishing as possible.</p>

<p>First, select a fitness function by clicking a radio button below. This
function will be used to determine the relative fitness of strings in the population.

Next, create a population by clicking first button below. This population is a random set of strings five bits long. If you prefer you may hand-enter a set of strings or change strings which are there. When you finish editing the population hit the second button. Either button will calculate decimal values for the strings in the population. Note that if you make a mistake in entering your population (i.e., use a character which is not a bit or use more or less than five bits per string) a population will be generated for you automatically.

Lastly, click the third button to apply the fitness function and calculate the probability of a string being selected in the reproduction pool. If you like you can change the fitness function and recalculate the fitnesses.

<form name="functions">
<tr><td><input type=radio name=func value=log onClick="clearFitness(document.foo)"/>
  <input type=hidden name=f value=x&deg;ln(x)/2></td></tr>
<tr><td><input type=radio name=func value=n onClick="clearFitness(document.foo)"/>
  <input type=hidden name=f value=2x</td></tr>
<tr><td><input type=radio name=func value=square onClick="clearFitness(document.foo)"/>
  <input type=hidden name=f value=x^2</td></tr>
<tr><td><input type=radio name=func value=exp onClick="clearFitness(document.foo)"/>
  <input type=hidden name=f value=2e^x</td></tr>
</form>

<form name="foo">
<table frame=void>
<tr><td>String</td><td>Decimal</td><td>Fitness</td><td>Selection</td><td>Expected</td></tr>
<tr><td><input type=hidden name=pop1 SIZE=5></td></tr>
<tr><td><input type=hidden name=dec1 SIZE=2></td></tr>
<tr><td><input type=hidden name=fit1 SIZE=6></td></tr>
<tr><td><input type=hidden name=sel1 SIZE=4></td></tr>
<tr><td><input type=hidden name=cnt1 SIZE=4></td></tr>
<tr><td><input type=hidden name=pop2 SIZE=5"></td></tr>
<tr><td><input type=hidden name=dec2 SIZE=2></td></tr>
<tr><td><input type=hidden name=fit2 SIZE=6></td></tr>
<tr><td><input type=hidden name=sel2 SIZE=4></td></tr>
<tr><td><input type=hidden name=cnt2 SIZE=4></td></tr>
<tr><td><input type=hidden name=pop3 SIZE=5"></td></tr>
<tr><td><input type=hidden name=dec3 SIZE=2></td></tr>
<tr><td><input type=hidden name=fit3 SIZE=6></td></tr>
<tr><td><input type=hidden name=sel3 SIZE=4></td></tr>
<tr><td><input type=hidden name=cnt3 SIZE=4></td></tr>
<tr><td><input type=hidden name=pop4 SIZE=5"></td></tr>
<tr><td><input type=hidden name=dec4 SIZE=2></td></tr>
<tr><td><input type=hidden name=fit4 SIZE=6></td></tr>
<tr><td><input type=hidden name=sel4 SIZE=4></td></tr>
<tr><td><input type=hidden name=cnt4 SIZE=4></td></tr>
<tr><td><input type=hidden name=avg SIZE=6></td></tr>
</table>
</form>
Automatically generate random population<br>
<input type="button" name="buserpop" value="OK" onclick="checkPopulation()"><br>
OR<br>
Calculate decimal with entered population<br>
<input type="button" name="bfitness" value="OK" onclick="applyFitness(document.foon)">

THEN Apply fitness values and calculate expected count<br>

</form>

<p>Note that the reproduction population is not done exactly as Holland specified in the generalized reproduction operator above. Holland acts on one member of the population at a time for reproduction. In this example it is far easier to handle the entire population as a whole. We already need to apply the fitness function in order to select a member probabilistically. Rather than select one member of the population we instead generate a new entire population to which we will apply the other genetic operators. This method creates a clear division between generations which is not faithful to the biological model.</p>

<p>The selection probability for each member of the population represents the probability that this member will appear in the reproduction pool. It is equal to the ratio of an individual member’s fitness to the total fitness of the population. The sum of all selection probabilities will sum to one.</p>

<p>The expected count column lists the amount of representation we expect each member to have in the reproduction pool. It is equal to the ratio of a member’s fitness to the average fitness of the population. Obviously the sum of all expected counts will sum to the size of the population.</p>

<p>We can now create our mating pool which is the set of members selected probabilistically from the original population. If a member had a selection probability of 2.00 we would expect to see two copies of it in the mating pool. Click on the button labelled "Create mating pool" to do so. Again, if you wish to edit the strings by hand you may do so. (If you get "undefined" entries in your populations make certain that the original population in the table above has been created and rated for fitness.)</p>

<form name="bar">
<table frame=void>
<tr><td>Mating Pool</td><td>Mate</td><td>Crossover</td><td>New</td><td>Decimal</td><td>Site</td><td>Population</td><td>Value</td><td>Value</td></tr>
<tr><td>
<input type="text" name="pool1" size=5" />
</td><td>
<input type="text" name="mate1" size=1" /></td><td>
<input type="text" name="site1" size=1" /></td><td>
<input type="text" name="pop1" size=5" /></td><td>
<input type="text" name="dec1" size=5" /></td><td>
<input type="text" name="fit1" size=6" /></td><td>
</td><td>
</td></tr>
<tr><td>
<input type="text" name="pool2" size=5" />
</td><td>
<input type="text" name="mate2" size=1" /></td><td>
<input type="text" name="site2" size=1" /></td><td>
<input type="text" name="pop2" size=5" /></td><td>
<input type="text" name="dec2" size=5" /></td><td>
<input type="text" name="fit2" size=6" /></td><td>
</td><td>
</td></tr>
<tr><td>
<input type="text" name="pool3" size=5" />
</td><td>
<input type="text" name="mate3" size=1" /></td><td>
<input type="text" name="site3" size=1" /></td><td>
<input type="text" name="pop3" size=5" /></td><td>
<input type="text" name="</td><td>
</td><td>
</td></tr>
</table>
</form>
Create mating pool by selecting probabilistically from original population<br>

Find mate for each member and pick crossover sites<br>
Create new population from the crossed-over old population<br>

Now that we have identified the class of strings which are to reproduce we can examine the action of the main operator of the genetic algorithm, crossover. Clicking the second button will choose a mate at random and a crossover site for the pair. The third button will then paste the head of one string onto the tail of its mate and vice versa. This button will also show the new decimal values, their relative fitnesses, the new average fitness and the difference.

With a high probability there is a relatively large positive number in the box labelled `Difference.' (If there's not, you most likely didn't click the buttons hard enough. Go back and try again.) In only one generation the average fitness of the population changed dramatically with the use of only two genetic operators, reproduction and selection. If the process were continued we would obviously focus on strings which are the better performers in the population as the weaker strings have been removed.

Under the process of reproduction solely the growth of a string is related directly to its fitness. It is selected to reproduce with probability

Where $S$ is the total fitness of all strings in the population. The growth of a string can then be modelled as a simple difference equation:<br>

where $m(i)$ represents the number of string $H(i)$ in a population at time $i$. This equation says simply that a string which is twice above average will be represented twice as often in the new population. Similarly, a string which is a constant amount below
Assume that some string is much better than the others by a factor of a constant, $i &lt; c/i$. Then the fitness of the individual is equal to $i &lt; c/i$ times the average fitness at each time step $i &lt; t/i$: 

\[ i &lt; f/i (i &lt; H/i (i &lt; i)] = i &lt; c/i \]  

Thus the difference equation modelling the growth of the string can be reduced:

Starting at time zero we obtain $i &lt; m/i (i &lt; H/i, i &lt; t/i) = i &lt; m/i (i &lt; H/i, 0) &amp; 183; (1 + i &lt; t/i) / sup/i &lt; t/i / sup which is the discrete analog of the exponential. A href="#thREFnAAC" name="thRefAAC" sup/i &lt; t/i / sup /a> Thus reproduction leads to exponentially increasing representation of fit solutions much as interest accrues on money. Rather than rate $i &lt; r/i$ we have the constant $i &lt; c/i$ which is determined by the amount of definition given by the fitness function.

<h3>Schema in the handworked example</h3>

**FORM NAME = "frob"**
Two of the better-performing schema from our above example. The above chart shows two strings from the original population and schema which are represented in those strings. The other strings they appear in, if any, are listed in the third column.

The average fitness column lists the average fitness of the schema's represented strings. For example, if the schema appears in strings 2 and 4 this column lists the average fitness of strings 2 and 4. From this we can calculate an expected count in the crossed-over population. The last column lists the schema's actual representation in the crossed-over population.

Why is it that the some strings were extremely fit for reproduction whereas others were not? Obviously having a 1 in the highest bit helps to make for the largest number and thus the largest score from the fitness function \(<i>f</i>(<i>x</i>) = <i>x</i>^2</sup>\(^2</sup>. This property of having a one in the highest can be represented by the schema 1 \&164 \&164 \&164 \&164 \&164 where the character \&164 \&164 represents a wildcard or "don't care" character. The schema 0 \&164 \&164 \&164 \&164 \&164 is obviously an unfit schema. The first string, however, survived to reproduce despite containing it. In many generations of reproduction we would expect to see the schema 0 \&164 \&164 \&164 \&164 disappear completely.

This is how schemata allow us to identify and assign credit to certain characteristics of the solution rather than the solution as a whole. Any class of strings beginning with 1 \&164 \&164 \&164 \&164 would be expected to perform well. In this way rather than chasing down only highly fit solutions which may have some fatal flaw we are able to target a class of highly fit solutions. Any problem which has an area of local minima surrounding a peak or any problem with a "false maximum" peak will be easier to solve when annealing by patterns than when using a simple hillclimbing algorithm of complete solutions.

While working with schemata adds some granularity of control it also adds a layer of complexity to the system: in a binary alphabet of 0's and 1's there are only 2\(^5\) = 32 possible strings of length five. When we move to schemata and add a "don't care" symbol of \&164 we increase the possible schemata to 3\(^5\) = 243. We have 243 schemata to search but still only 32 possible solutions.

Schemata also have their own characteristics which influence the behaviour of an SGA. Consider a schema such as 1 \&164 \&164 \&164 1. This schema, while beneficial at both specified positions, would not survive any crossover. The string length is five and the specified length of the schema itself is five. Any crossover site destroys the schema. Likewise, a schema such as 1 1 0 1 \&164 would have difficulty surviving crossover because it can only survive crossover in the fourth position.

The number of positions specified in a schema is referred to as its order. The number of positions between the first and last specified positions is its defining length. The effects of genetic operators such as crossover, mutation and inversion lead to the survival of low-order schemata with short defining lengths.

We calculate the survival probability by looking at the number of possible crossover sites which is always 1\(^1\). Given that there will be crossover of a particular schema the probability
of its destruction will be the difference of its defining length, \(<font face="symbol">d</font>\) and the number of crossover sites. The probability of its survival is then \(<font face="symbol">1</font>\) if there is a given probability that a schema will be selected for crossover of \(<font face="symbol">i</font>\) then the probability \(<font face="symbol">i</font>\) of it surviving a round of reproduction with random crossover is \(<font face="symbol">1</font>\).

Mutation does not appear in our handworked example because of its relative infrequency. It does occur, however, and thus can be assigned a probability \(<font face="symbol">i</font>\) for each of the fixed positions in the schema. Those schemata with a greater defining length are referred to as being of greater \(<font face="symbol">i</font>\). If a schema is of order \(<font face="symbol">i</font>\) then the probability that it will survive mutation is \(<font face="symbol">1</font>\).

If crossover, reproduction and mutation are independent these all work together to produce what Goldberg calls the Schema Theorem.

This theorem counts the number \(<font face="symbol">m</font>\) of representations of a schema \(<font face="symbol">i</font>\) in a population at time \(<font face="symbol">t</font>\). It takes into account the relative fitness \(<font face="symbol">f</font>\) of \(<font face="symbol">i</font>\).
crossover which is a function of defining length \( d \) of the schema and the total length \( l \) of a member solution \( A \). What is most important about the Schema Theorem is that it arrives at the same conclusion as Holland's generalized operators; that solutions are given exponentially increasing (or decreasing) representation in the population as a function of their relative fitness.

<P>
<a href="index.html"&gt;Abstract&lt;/a&gt;
</P>&#149
<a href="intro.html">Introduction</a>&#149
<a href="adaptive.html">A simple adaptive plan</a>&#149
<a href="generalized.html">Generalized genetic operators</a>&#149
<a href="other.html">Other genetic operators</a>&#149
<a href="conclusion.html">Conclusion</a>&#149
<a href="references.html">References</a>&#149

<P>
h3>Footnotes:</h3>

<P>
<a name="tthPnttAAB"> </a><a href="#tthRefAAB">6</a><sup></sup><a href="#tthFntAAB">Goldberg, pp. 33 - 35</a>
</P>
<a name="tthPnttAAC"> </a><a href="#tthRefAAC">7</a><sup></sup><a href="#tthFntAAC">Goldberg p. 30</a>
</P>
<a name="tthPnttAAD"> </a><a href="#tthRefAAD">8</a><sup></sup><a href="#tthFntAAD">Goldberg, p. 32</a>

</BODY>