3E381 Neural Computation

Lecture 11 Evolutionary Computation

11.1 Introduction

The principle of evolution is a unifying concept in biology that links every organism together through a historical chain of events. Every creature in the chain is the product of a long series of occurrences, many of them accidental, but all of them taking place under the stern selective pressure of the environment. Over many generations, random variation and natural selection have shaped the behaviours of individuals and species so that they can cope with the demands of their surroundings. Evolution is driven by the effects of physical laws acting on populations of different species and has resulted in many exotic and ingenious solutions to the problems of survival faced by different types of creature. The aim of evolutionary computation is to bring to computing the flexibility and ingenuity inherent in the evolutionary process and to use it to find solutions to problems that are very difficult using conventional computing methods.

In the very early days of evolutionary computation the method scored a significant win when some people working on magnetohydrodynamics used the approach to try to improve the design of a high-speed nozzle. What the technique came up with was something that looked so weird that the users spent quite a long time trying to find the bug in their program. Eventually they had to decide that their code was OK and they tried simulating the new nozzle – the simulations indicated that it was better than anything previously developed. So then they had to study the new structure and eventually found that, although it was a most unconventional structure, it actually was superior.

In this lecture I will confine the discussion of evolutionary computation mainly to genetic algorithms.

11.2 Genetic Algorithms

Genetic algorithms (GAs) employ evolution-like strategies to search for the best solution to a given problem. Thus, they are optimization algorithms and they seek the optimum using methods based upon the uncompromising technique of Mother Nature that we know as "survival of the fittest". GAs encode each point in the search space as a binary bit string that resembles a string of genes in a chromosome, so the bit string is usually referred to as a chromosome.

In passing, you may find it interesting to note that vertebrates have more than ten thousand genes and each gene can come in two or more forms (called *alleles*). These facts allow us to compute a lower bound on the total number of distinct individual human beings that could possibly be born. Let's assume 10,000 genes and two alleles per gene – this gives $2^{10,000} \approx 10^{3,000}$ different babies that could be born. This is a lower bound, so the actual figure would be somewhat bigger than this, but it gives you a reasonable idea of the number we're dealing with. Big numbers like this, of course, are not easy to grasp, but you will perhaps get some appreciation by noting that the number of elementary particles in the universe is "only" ten raised to the power of two or three hundred (I can't remember exactly). The chances of two people having identical genetic structure in the current world population of 5×10^9 therefore appear to be vanishingly small. But of course, if you indulge in in-breeding, as royal families have tended to do, the possibility of producing an exact replica of an ancestor is much greater. Hands up all those who'd like to see the return of Vlad the Impaler.

Now, as I said above, GAs encode each point they investigate in the space they are to search as a binary bit string. And each such point is assigned a value which is referred to as the *fitness* of that point. As an example, in a situation where you want to find the point where a mathematical function attains its maximum value, the fitness of a point would be the value of the function at that point. GAs maintain a population of points which evolves over a series of generations. In each generation, the GA constructs a new population using operations on the binary strings that are similar to those that take place in real chromosomes. Members of the population with higher fitness values are more likely to participate in the process of creating a new generation so there is a tendency for the population to improve its overall fitness.

In comparison to more traditional techniques for optimization, GAs differ in at least four fundamentally different ways:

1. The parameters of an optimization problem are coded as bit strings.
2. GAs search from a population of points, not a single point.
3. GAs use information from the objective function directly, not its derivatives or any other auxilliary knowledge.
4. GAs make their transitions probabilistically, not deterministically.

As a simple example, suppose we wish to maximize the function $f(x) = x^2$ on the integer interval [0,31]. The answer, of course, is dead obvious, but this example will serve to demonstrate some basic concepts. If we were employing a traditional algorithm to solve this problem, we would move cautiously from a single starting point to a point near by using some sort of transition rule. And we know this point-to-point method has problems because if you're trying to minimize a function it can get you stuck at a local minimum. (Equally, if you're trying to maximize, you can finish up at a local maximum.) By contrast, GAs work from a rich database of points simultaneously (a population of strings) and, when maximizing a function (as we wish to do here) they climb many peaks in parallel. This reduces the probability of getting stuck at a local maximum.

In the case of our example, the genetic algorithm will start with a population of strings and thereafter generate successive populations of strings. For this problem, where we are simply searching the integers in the range [0,31], we only require bit strings of length 5. Suppose we start with an initial population of size 4 with the bit strings generated randomly (say by flipping a coin 20 times) and taking the form : 01101, 11000, 01000 and 10011. From this starting population, successive populations will be generated using a genetic algorithm. The GA will use probabilistic transition rules to guide the search to a solution.

11.2.1 A simple genetic algorithm

The procedures followed by simple genetic algorithms are very straightforward, involving nothing more complex than the copying of strings and the swapping of partial strings. The explanation of why such a procedure works effectively is a little more subtle and we shall come to that in due course.

A simple genetic algorithm that yields good results in many practical problems is composed of four operators:

1. *Fitness evaluation*: Fitness evaluation involves calculating a fitness value f_i for each member of the population, which is normally done by simply evaluating the objective function at the point represented by each member of the population.
2. *Selection*: Following evaluation, the selection operation is carried out to determine which members of the current population will participate in producing offspring for the next generation. Of course, they're all queuing up shouting Me! Me! but the selection operator has to be strict and members are selected for mating with a probability proportional to their fitness values f_i. The most common way to implement this is to set the selection probability equal to $f_i / \sum f_k$ where the summation is over the full population.
3. *Crossover*: This is basically the mating operation. To exploit the potential of the current gene pool, crossover operators are employed to generate new chromosomes that are likely to retain good features from the previous generation. This involves the swapping of substrings within the bit strings of each parent (eg the first *m* bits in the bit string of each parent might be exchanged). The effect of this is similar to that of mating in the natural world where parents pass on segments of their own chromosomes to their children.
4. *Mutation*: The selection and crossover operations on their own can occasionally lead to the search process concentrating overmuch on a relatively unproductive area of the search space. For this reason, a mutation operator, capable of spontaneously generating new chromosomes (and hence shifting the search to other areas) is included. The most common way of implementing mutation is to flip a bit with a probability set equal to a very low *mutation rate*. In this way, mutation takes place relatively rarely, but can be crucial to finding a good solution to a problem.

Let's now apply this algorithm to the simple problem I introduced above. We have an initial population of four bit strings, and the first thing we need to do is to evaluate their fitness. This is illustrated in Table 11.1 where, for instance, the fitness of 01101, which is the number 13 in decimal, is equal to $13^2 = 169$.

Table 11.1 Sample problem strings and fitness values

Number	String	Fitness	% of total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.0

One way of implementing the selection rule is to construct a weighted roulette wheel, as shown in Fig. 11.1. To understand this figure, note that string number 1 in the table above has fitness 14.4% of the total fitness of the population. On the roulette wheel, string number 1 is given 14.4% of the wheel. (I've taken this drawing from a book – ref[1] – and it's a pretty big 14.4% on the figure, but you should get the idea). Each spin of the wheel is intended to come up with string 1 with probability 0.144, with string 2 with probability 0.492 and so on. So, each time we want to select a member of the population to participate in the reproductive process, we spin the wheel and whichever member the wheel settles on is the next participant. This ensures that those strings with the higher fitness levels tend to have a higher number of offspring in the succeeding generation. Each time a string is selected for mating, a replica of that string is added to the mating pool.

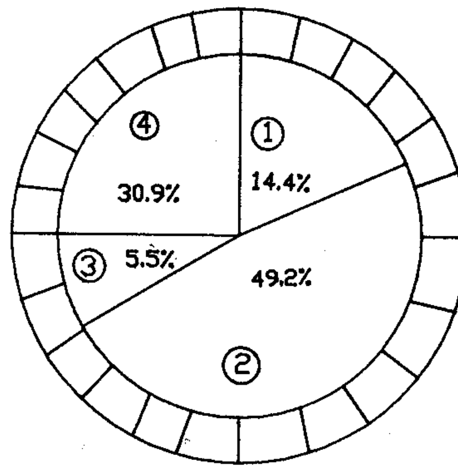


Figure 11.1

Once the mating pool is complete, pairs of members from the pool are selected at random to participate in the crossover operation. Each selected pair undergoes crossover as follows: an integer position k along a string is selected uniformly at random between 1 and $n-1$, where n is the string length. Two new strings are created by swapping the full set of bits between positions $k+1$ and n. To illustrate this, consider strings S_2 and S_4 from the initial population in our example:

$$S_2 = 1\ \ 1\ \ 0\ \ 0\ \ 0$$
$$S_4 = 1\ \ 0\ \ 0\ \ 1\ \ 1$$

and suppose that in choosing a random number between 1 and 4 we obtain the number k = 2. This means that we are going to swap the bits in positions higher than 2 in the string. This means that 3 bits will be involved in the crossover operation as indicated below:

$$S_2 = 1\ \ 1\,|\,0\ \ 0\ \ 0$$
$$S_4 = 1\ \ 0\,|\,0\ \ 1\ \ 1$$

and crossover leads to the swapping of these bits to give the strings below:

$$S'_2 = 1\ \ 1\ \ 0\ \ 1\ \ 1$$
$$S'_4 = 1\ \ 0\ \ 0\ \ 0\ \ 0$$

Most of us who have been fortunate enough to have a reasonable education have some understanding of the process of natural selection, but know rather less about mutation and its role in the evolutionary process. Our understanding is not assisted by B-grade movies that show things like rampant mutant eggplants devouring major parts of Tokyo or Chicago. Mutations in biological populations can be caused by radiation and chemical contamination, but most commonly arise due to slight errors occurring in the reproductive process. And they are remarkably common – roughly every fifth child carries some kind of mutated gene [2]. The vast majority are harmless, but some are not. Before these processes were at all understood (and they're far from fully understood

even now) events such as a family of in-bred black rats suddenly producing an albino (with no pigmentation) seemed quite unbelievable. Sometimes mutations are beneficial and this is why mutation has played a major role in the evolution of species.

In the simple GA, mutation is the occasional random alteration of the value of a bit in a bit string (ie from 1 to 0 or vice-versa). For a given bit string, each bit is flipped with probability p where p is necessarily small – in practical situations where the bit string length L is generally quite large, it is common to set $p = 1/L$.

Other types of genetic operator that occur in biology, besides the four discussed in this section, have been tried as contributors to genetic algorithms, but have been found to contribute nothing especially useful.

Let us now continue with our example.
The process commenced with the random selection of an initial population of the four bit strings in Table 11.1. This initial population is repeated in Table 11.2 where some additional information is provided.

String Number	Initial Population	Decimal Equivalent	$f(x) = x^2$	Selection Probability $f_i/\sum f_k$	Expected Selection count	Actual count (from wheel)
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4.00	4
Average			293	0.25	1.00	1
Maximum			576	0.49	1.97	2

Table 11.2 First steps in applying genetic algorithm

The mating pool for the next generation is selected by spinning the weighted roulette wheel four times. Thus, the mating pool contains the same number of bit strings as are present in the initial population but, as can be seen in Table 11.2, the method of selection for the mating pool (random selection with replacement) allows strings to be selected more than once. Strings 1 and 4 are selected once, string 2 is selected twice and string 3 not at all. Clearly strings with a higher fitness are more likely to be selected and those with a lesser fitness are less likely. But the randomness of the procedure still allows for the possibility of low-fitness individuals being included in the mating pool. In larger, more practical, problems many relatively low fitness individuals would participate in the mating process, thus maintaining the population diversity that is necessary for the best solutions to be found.

So, we now have a mating pool of strings and we can imagine them actively looking for mates within the pool. Mating to produce offspring is implemented by the crossover operation and is carried out in two steps: (i) strings are mated randomly using a random number generator to pair off the happy couples, and (ii) the crossover operation is applied to mated couples by use of a random number generator to select the crossover point. This is illustrated in Table 11.3, where the random pairs are strings 1 & 2 and strings 2 & 4. For the first pair the crossover takes place after the fourth bit and for the second pair, crossover takes place after the second bit. (Note that you could equally well say that the crossover takes place *before* the fourth bit for the first pair and, for the second pair, before the second bit – either way you finish up with the same offspring.) These operations lead to the new population shown in column 4 of the table.

Mating pool showing crossover point	String Number	Crossover Point	New population	Decimal value	$f(x) = x^2$
0 1 1 0 | 1	1	4	0 1 1 0 0	12	144
1 1 0 0 | 0	2	4	1 1 0 0 1	25	625
1 1 | 0 0 0	2	2	1 1 0 1 1	27	729
1 0 | 0 1 1	4	2	1 0 0 0 0	16	256
Sum					1754
Average					439
Maximum					729

Table 11.3 Production of a new population from the mating pool

At this point one would normally implement the mutation operation, too. Note that I said above that in practical problems the mutation probability is commonly set equal to $1/L$ where L is the string length. But here the string length is only 5 so $1/L = 0.2$. Experience has shown that if a mutation probability greater than 0.1 is employed in a genetic algorithm, the algorithm does little more than random search – the mutations occur so frequently that they obliterate the effects of crossover. So, for a small problem such as this one, we must choose a more sensible mutation probability, say $p = 0.01$. But with this choice, the probability of any of the 20 bits being flipped in the new population is only 0.2, so we can assume that no changes are brought about by mutation.

Following selection, crossover and mutation, the new population is ready to be assessed. This we can do in this example by determining the decimal value x of each string in the new population and determining its fitness value $f(x) = x^2$. The fitness values are shown in the right-hand column of Table 11.3. The most significant values are the maximum and average values of fitness which can be compared with the corresponding values of the initial population which are shown in the middle column of Table 11.2.

Perhaps you expected this kind of outcome. But let's just review the main contributing factors. The best string in the initial population (11000) was selected twice for the mating pool because of its high fitness level. When the random selection process in the mating pool caused it to combine with the string having the next highest fitness (10011), one of the two strings resulting from the crossover operation comes out with an excellent fitness value. This crossover is at position 2 in each string (note that this position was determined randomly) and creates the best string so far (11011).

Let's now try to look at the broader picture. Just what is going on here? We have a search process based only upon bit strings and fitness values, but what is it about these two things that guides the search towards better fitness values? If we look at the initial population and its fitness values we can see that strings starting with a 1 are better than strings starting with a 0. For the example fitness is given by $f(x) = x^2$ so we know, in this case, that strings starting with a 1 will do better. But in general, the situation is not so simple. However, by studying the behaviour of substrings as a genetic algorithm progresses, we can arrive at a (relatively simple) mathematical explanation of why genetic algorithms work, ie why they reliably provide solutions to optimization problems. To do this we need to introduce the concept of a *schema* (plural *schemata*). The so-called schema theorem gives us the explanation of why genetic algorithms work reliably.

11.2.2 Schemata and the fundamental theorem of genetic algorithms

The idea of a schema was introduced by the guy who first dreamed up genetic algorithms, namely John Holland, who worked at the University of Michigan. His ideas on GAs were developed during the 1960s and he first described the schema in [3]. His first accessible journal article on the subject of GAs is [4] and his first book on the subject is [5].

Schemata are sometimes referred to as similarity templates because they capture similarities between bit strings in terms of their substrings. A particular schema is a particular substring with don't cares in all the other bit positions. Thus, if we are particularly interested in the 111 substring at the centre of a string of length 5 bits, we would write the schema for this as *111*, where the asterisk is used to represent don't cares.

The specified variables in a schema (as against the don't cares) need not be contiguous (ie consecutive) in a schema as in *111*. Thus, in a 7-bit string, one of the many possible schemas is 011*1** and another is 1****1*. Note that the schema 011*1** has more specified values than the schema 1****1*. Note also that the specified values in the schema 1****1* span more bit positions than those in the schema 011*1**. These are meaningful concepts in relation to the study of schemata and are given special names.

The *order* of a schema H, denoted $o(H)$ is the number of specified bits (ie the number of 1s and 0s) in the schema. Thus, the order of the schema 1*1**** is 2 and the order of 011*1** is 4.

The *defining length* of a schema H, denoted by $\delta(H)$, is the difference between the bit positions of the first and last specified bit in the schema. For example, the schema 011*1** has defining length $\delta = 4$ $(= 5 - 1)$ and the schema 1*1**** has defining length $\delta = 2$. Note in particular that the schema ***0*** has defining length $\delta = 0$ because the first and last specified bits are in the same position.

Schemata provide a basis for analyzing the effect of the selection, crossover and mutation operations used in genetic algorithms. But before we proceed with this, we need just a little more notation. We will let A_i represent bit string i from the total population of bit strings $\boldsymbol{A}$ and will represent the fitness of A_i by f_i. We are now in a position to begin investigating the effect of the selection process on the expected number of schemata in the population of bit strings.

Suppose that at a given time step t there are m examples of a particular schema H contained within the population $A(t)$. The time step determines the generation we are dealing with (ie $t = 0$, we are dealing with the

initial population, $t = 1$ the first generation, etc.). Since m depends on both the schema H that we are interested in and upon the time step, we write $m = m(H, t)$. We will now investigate how this number changes as the selection process is implemented. For the moment, we will ignore the effects of crossover and mutation.

During selection, a string is chosen for the mating pool according to its fitness, so that string A_i is selected with probability $f_i / \sum f_k$. After selecting a population of size n (with replacement, so that some strings can be selected more than once) from the population $A(t)$ (also of size n) the number of examples of schema H will be $m(H, t+1)$. And since we are for the moment ignoring both the effects of crossover and mutation, this is the number of examples of schema H that we can expect in the population at generation $t+1$. And if $f(H)$ represents the average fitness of the strings containing schema H at time t, we can write, for the expected number of schemata H at time $t+1$.

$$m(H,t+1) = m(H,t) \cdot n \cdot \frac{f(H)}{\sum f_k} \tag{11.1}$$

Equation (11.1) holds because for each string containing schema H at generation t, the probability of selection on the basis of fitness is $f(H)/\sum f_k$ and n such selections are made.

Now note that the average fitness of the whole population can be written

$$\overline{f} = \sum f_k / n \tag{11.2}$$

so that (11.1) can be written

$$m(H,t+1) = m(H,t)\frac{f(H)}{\overline{f}} \tag{11.3}$$

and what (11.3) states is that any schema H grows in numbers proportional to the ratio of the average fitness of H to the average fitness of the population as a whole. Put another way, schemata with fitness values above the population average increase in numbers in the next generation and those with below average fitness decrease in numbers.

To gain an appreciation of the speed of growth of populations with above average fitness, let's make the simplifying assumption that a particular schema H remains above the average fitness level by a constant amount. That is

$$f(H) = (1 + c)\overline{f} \tag{11.4}$$

where c is a constant. With this assumption, the difference equation (11.3) becomes

$$m(H, t+1) = (1 + c) \cdot m(H, t) \tag{11.5}$$

Starting with t = 0, this gives $m(H, 1) = (1 + c) \cdot m(H, 0)$, $m(H, 2) = (1 + c)^2 \cdot m(H, 0)$ and so on, so that in general,

$$m(H, t) = (1 + c)^t \cdot m(H, 0) \tag{11.6}$$

Equation (11.6) describes a geometric progression, which is the discrete version of the exponential for continuous functions. So what this means essentially is that for schemata that have fitness somewhat above the average, the selection process will arrange for their numbers to increase exponentially, and equally, for schemata with fitness a little below average (c negative), the selection process will ensure that their numbers decrease exponentially.

Equation (11.6) applies only to populations generated by the selection procedure. We must now investigate how the processes of crossover and mutation affect this result.

Although the selection procedure clearly does a good job in ensuring the survival (and flourishing) of the fittest, it does nothing to promote exploration of new regions of the search space, since it does no more than select

existing strings to put in the mating pool. It is the process of mating that leads to crossovers taking place and thereby providing opportunities for a wider exploration of the search space.

To see how schemata can be affected by crossover, consider a particular string of length $L = 7$ and two of the schemata contained within the string:

$$\begin{array}{lcccccccc} A & = & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ H_1 & = & * & 1 & * & * & * & * & 0 \\ H_2 & = & * & * & * & 1 & 0 & * & * \end{array}$$

First make sure that you understand that the two schemata are contained in the string *A.* Then recall that crossover proceeds with the random selection of a mate (from the mating pool), the random selection of a crossover point and the exchange of substrings on one side of the crossover point – remember, it doesn't matter which side you choose to do the exchange on. Suppose we roll a die to determine the crossover point – note there are six possible crossover points in a string of length 7 – and let's suppose the die turns up a 3. This means that the crossover will take place to the right of bit 3:

$$\begin{array}{lcccccccc} A & = & 0 & 1 & 1\,| & 1 & 0 & 0 & 0 \\ H_1 & = & * & 1 & *\,| & * & * & * & 0 \\ H_2 & = & * & * & *\,| & 1 & 0 & * & * \end{array}$$

Now, we don't need to know the string that *A* will be mating with, because what we want to do here is to identify the kind of effect different types of mate will have on the schemata H_1 and H_2. First note that unless *A*'s mate is identical to *A* in bit positions 2 and 7 (and hence also contains schema H_1) the crossover process will have the effect of destroying schema H_1. This is because the 1 in position 2 and the 0 in position 7 will be placed in different offspring (they are on opposite sides of the separator symbol marking the crossover point). Secondly, observe that schema H_2 will survive regardless of the nature of *A*'s mate – this is because the 1 at position 4 and the 0 at position 5 will be transferred intact to a single offspring.

The above observations were based upon the specific crossover point between positions 3 and 4 in the string, but it should be clear that schema H_1 is generally less likely to survive crossover than schema H_2 because on average the crossover point is less likely to fall between the specified bits in H_2. To quantify this, note that schema H_1 has a defining length of 5 (recall that this is the difference between the bit positions of the first and last specified bits). If the crossover point is selected uniformly at random among the $L - 1 = 7 - 1 = 6$ possible sites, then clearly schema H_1 is destroyed with probability $p_d = \delta(H_1) / (L - 1) = 5/6$, unless A's mate also contains schema H_1 – what did John Howard say about mateship? Similarly, schema H_2 has defining length $\delta(H_2) = 1$ and can only be destroyed during the one event in six where the crossover point is selected to occur between positions 4 and 5 so that $p_d = 1/6$ and the probability of H_2 surviving is 5/6.

More generally, a lower bound on crossover survival probability p_s can be calculated for any schema. This will be a lower bound because, for simplicity of analysis, we will ignore the possibility that both mates contain the schema in question. So, we base our calculation on the fact that a schema will definitely survive when the crossover falls outside its defining length. In this case the survival probability for schema *H* can be written $p_s > 1 - \delta(H) / (L - 1)$ because the schema is open to destruction whenever crossover takes place at a point within its defining length. The greater than sign is used here because there is always the possibility of the mating pair each containing the schema *H*.

In the general application of a genetic algorithm, we may not always wish to apply crossover to a mating pair, so let's assume that crossover is applied to an arbitrary pair with probability p_c. The bound on survival probability for schema *H* then becomes

$$p_s > 1 - p_c \cdot \frac{\delta(H)}{L-1} \qquad (11.7)$$

We can now properly consider the combined effect of selection and crossover. And this is quite simple, if we assume independence of the processes of selection and crossover. Equation (11.3) gives us the growth rate for

schema H assuming no crossover. Equation (11.7) tells us the probability that H will survive the crossover operation and is therefore equal to the proportion of times, in the long run, that H survives crossover. So we can simply modify the growth rate for selection on its own (ie (11.3)) by multiplying by the proportion of times H survives crossover. This gives

$$m(H,t+1) > m(H,t) \cdot \frac{f(H)}{\overline{f}}\left[1 - p_c \cdot \frac{\delta(H)}{L-1}\right] \qquad (11.8)$$

To reiterate, the combined effect of crossover and selection is obtained by multiplying the expected number of schema H arising from selection alone by the survival probability under crossover. Equation (11.8), coupled with the development from (11.3) to (11.6) indicates that schemata with above-average fitness and short defining lengths will increase their numbers exponentially.

The last operation to consider is mutation which, in our scenario, is the random flipping of bits in a bit string. Let's suppose that the probability with which we flip any individual bit is p_m. In order for a particular schema H to survive, all of the specified bits in H must survive. The number of specified bits in schema H is the order of H, which we have denoted by $o(H)$. The survival probability for a single bit is $(1 - p_m)$ so the survival probability for the schema, which has $o(H)$ specified bits that it needs to preserve to survive, is $(1 - p_m)^{o(H)}$.

Now, as we have seen, the value of p_m is generally very small, so we can approximate $(1 - p_m)^{o(H)}$ by $1 - o(H) \cdot p_m$ (this is just the first-order Taylor series approximation as you should know). So now there are two ways in which a schema might fail to survive; one is incorporated into (11.7) and the other is the one we've just dealt with. Taking the two together, the probability that schema H will survive the crossover and mutation operations is now bounded as below:

$$p_s > 1 - p_c \cdot \frac{\delta(H)}{L-1} - o(H)p_m \qquad (11.9)$$

and inserting this in the obvious way in (11.8) gives

$$m(H,t+1) > m(H,t) \cdot \frac{f(H)}{\overline{f}}\left[1 - p_c \cdot \frac{\delta(H)}{L-1} - o(H)p_m\right] \qquad (11.10)$$

The inclusion of the mutation operation changes slightly our previous conclusions (following (11.8). We can now state that short, low-order schemata with above average fitness will increase exponentially in future generations. Equation (11.10) from which this conclusion is drawn is very important and is known as the *Fundamental Theorem of Genetic Algorithms*. It is also often referred to as the Schema Theorem.

11.3 Application to machine learning – the Learning Classifier System (LCS)

A major, and quite interesting application of GAs is to machine learning where the machine's behaviour is subject to a set of rules, and the machine modifies and improves these rules (so that it interacts better with its environment) over time.

In the more traditional rule-based systems (eg expert systems) the value of a given rule relative to other rules is fixed by the programmer in conjunction with the expert (or group of experts) whose knowledge is drawn upon to produce the rules. In a system that is to learn and adapt its own rules, we don't have the luxury of a group of experts to assist. The relative value of different rules is one of the key pieces of information that needs to be learnt. As we shall see, a system of payments and rewards, rather like a mini-economy, can be used to deal with this problem.

11.3.1 Basic system operation

A schematic diagram of the basic learning structure is shown in Fig. 11.2. Maybe we are best to think of the system in terms of a robot that is learning to survive in its environment, but it will become clear that there are many other applications for this kind of learning system. Information flows in from the environment through a set of detectors and is decoded into one or more messages. These messages are posted on a message list where they cause some of the rules stored in the classifier store to be activated. (Rather awkwardly, the stored rules have been

named "classifiers" and that's why they are held in the classifier store in the figure.) The overall system is known as a *Learning Classifier System* (LCS).

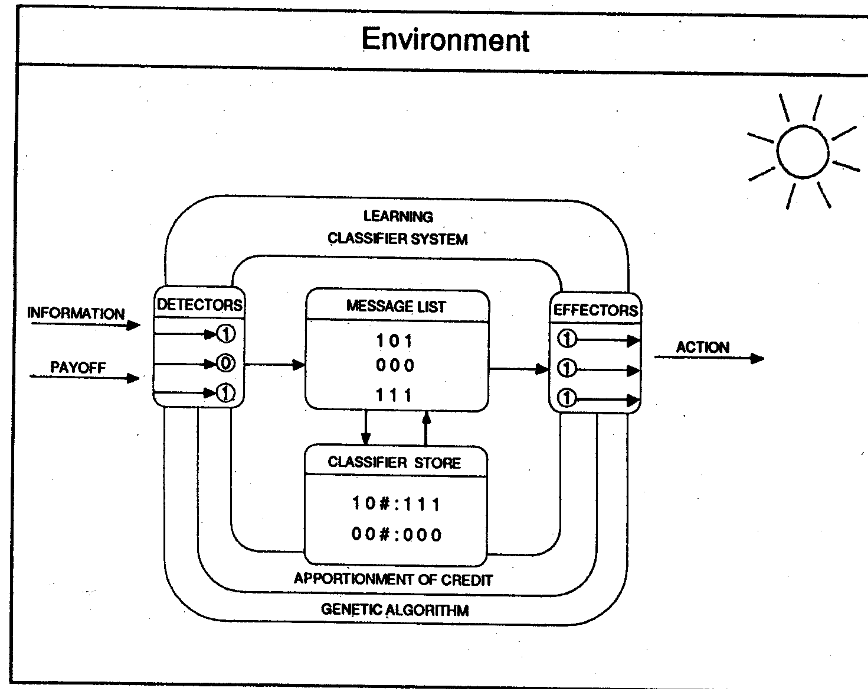


Figure 11.2

A message in the message list is simply a finite-length bit string. A classifier in the classifier store is a rule of the form "**if** a condition holds **then** perform a certain action". For instance, if the system were being used to control a mobile robot, a rule might be "if an object is to my right and in front of me then turn left". Such rules are stored as strings of bits and don't cares with a slash separating the condition from the action. Thus, the rule just stated might be stored in the form

1 1 / 1 0 0 0

where the # symbol is used for don't care. Here, the first 1 to the left of the slash represents "an object is present in front of me" and the second 1 represents "an object is present to my right". (Note how this is the kind of information you would expect to come in from the environment.) The action part, to the right of the slash might be encoded on the assumption that there are four discrete actions: turn left, turn right move forward and move backward. The action part in the above rule would then have the meaning "turn left, don't turn right, don't move forward and don't move backward".

A rule is activated by a message when the specified bits in its condition part match exactly with those in corresponding positions in the message. Thus, for instance, the condition # 0 1 # matches the message 0 0 1 0 but does not match the message 0 0 0 0.

Once a classifier's condition is matched, that classifier becomes a candidate to post its message to the message list at the next time step. Whether or not this will occur is determined by the current *strength S* of the classifier. In general, more than one rule will be activated by an arriving message and each matched classifier makes a bid (as in an auction) proportional to its strength. The winning classifier is allowed to post its action onto the message list and, in the simplest situation (which we will confine ourselves to), this action would be used to drive the effectors on the right of Fig. 11.2.

Let's now look a little more closely at the bidding procedure.

11.3.2 The reward system

Payments

When a new message comes in, it will in general match several classifiers. Each matched classifier makes a bid which is proportional to both its strength and its specificity (ie the number of its bits that have specific values rather than don't cares). Thus we can write

$$B_i = C_{\text{bid}} \times S_i \times NS_i \qquad (11.11)$$

where B_i is the bid value for classifier i, C_{bid} is the bid constant that applies to all classifiers, S_i is the strength of classifier i and NS_i is the number of specified bits in classifier i.

The winner of the auction is almost never selected on the basis of the highest bid alone. This is because it is possible that a moderately good, but highly specific rule could win the bidding in an early round against a less specific, but better rule. If good performance results the less good rule will be rewarded (see receipts below) thus increasing its strength and opening up the possibility of the less good rule beginning to dominate proceedings. To guard against this kind of thing, a degree of random noise is added to the bidding formula so that (11.11) becomes

$$B_i = C_{\text{bid}} \times S_i \times NS_i \times N(\sigma_{\text{bid}}) \qquad (11.12)$$

where $N(\sigma_{\text{bid}})$ is a zero-mean normal random variable with variance σ_{bid}.

Taxation

All classifiers have to pay a tax at each time step (this is deducted from their strength) and this is done so that the average strength of the population falls over time causing the more successful rules to stand out (because of the rewards they receive). The tax has to be very small (say 0.001) so that the ultimately successful rules have enough time to emerge from the pack.

Receipts

Classifiers can receive receipts from two sources. The first source is the environment – if the action of a given classifier results in a good outcome for the learning machine, then the strength of that classifier is increased in proportion to the outcome. This can be done automatically – eg a learning system in a robot can be rewarded automatically if the robot manages to pick something up.

Actions that result in poor performance are punished by decreasing classifier strength and the punishment can be quite severe. If a classifier leads to an action that is seen as highly undesirable, its strength will be reduced to zero so that it will never be selected again.

The second source of receipts occurs when it is required to reward a chain of actions that have led to a desirable outcome. For instance, if we wanted to train an LCS to play chess, we can only reward the system automatically if it successfully wins the game and this, of course, follows a long chain of actions. This is not entirely straightforward to implement, but we will ignore the difficulties here.

11.3.3 Creating new rules

After a reasonable time, the reward system will have differentiated between classifiers in the rulebase on the basis of strength. Strong classifiers will be those that increased the performance of the machine in its given environment. Classifiers that failed to perform will have a strength approaching zero.

In a complex system, the classifiers in the rulebase will represent only a fraction of the possible symbol structures in the problem space. Therefore at the beginning, when we commence with a set of random rules, we cannot expect to have the best ones. The reward system identifies the better ones in the rulebase, but we need to bring in some new ones to see if we can do any better. This is where the GA comes in.

Periodically, after allowing the reward system to operate for a period of time that allows good differentiation between rules, the GA is applied. The first step is to rank the N classifiers in the rulebase according to strength and then discard the bottom $N/2$ to make way for new rules.

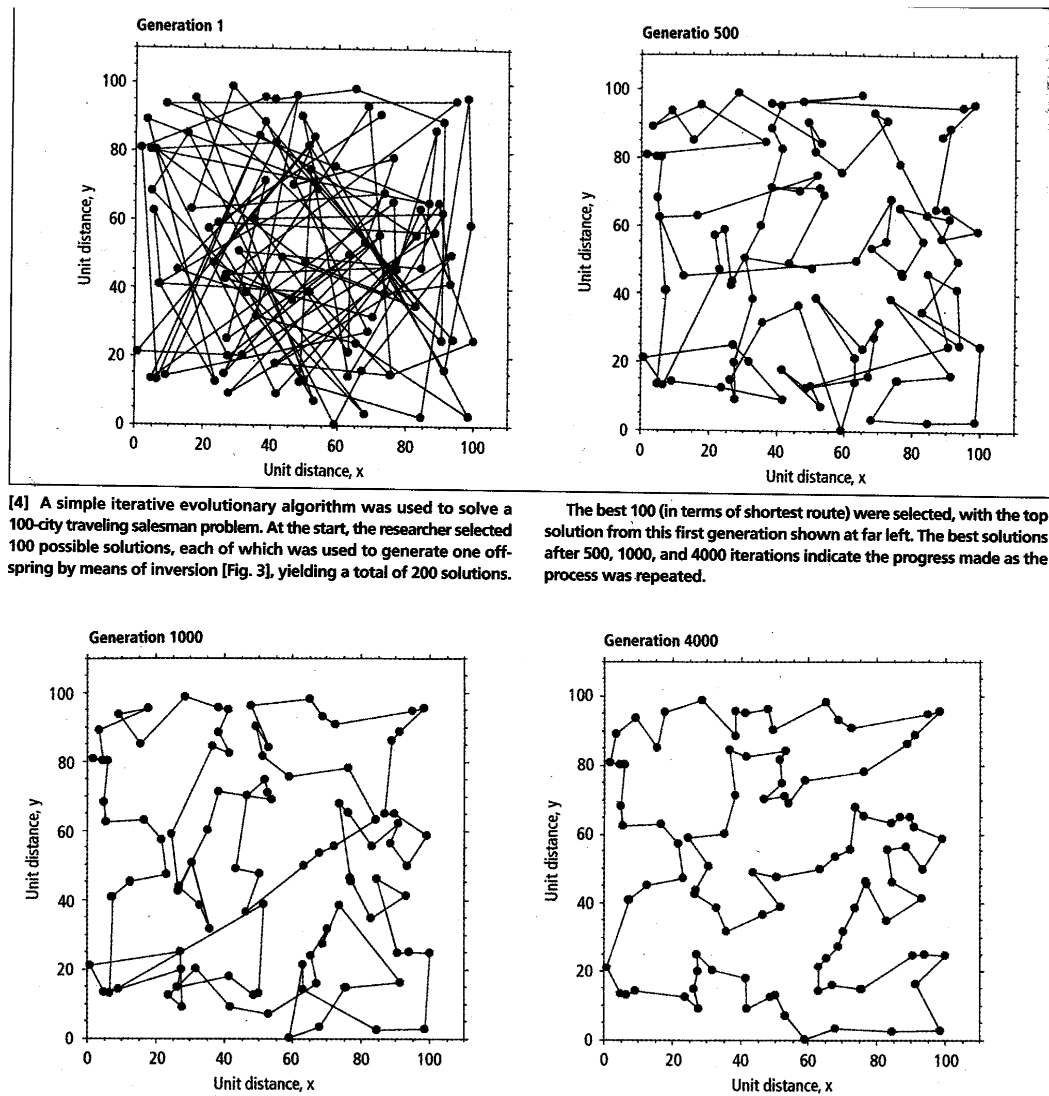
The GA is applied to the remaining $N/2$ rules in exactly the same way as described earlier, but with the fitness of a rule now measured by its strength. The strength of a new rule is then determined from the strengths of its parents using the crossover point to weight their two strengths. Thus, if we have a new rule consisting of the father's string to the left of the crossover point p and the mother's string to the right, the strength of the child is

given by $S_{child} = pS_{father} + (1-p)S_{mother}$. The mutation operation is also applied with a small probability. Note that the classifiers have three symbols: 0,1 and # so when a symbol needs to be changed it changes to one of the other two symbols with equal probability. The strength of new rules is not changed following application of mutation – the rewards system can be left to deal with that.

11.4 Concluding remarks

In this lecture I have described the basics of genetic algorithms. A GA is basically an optimization algorithm that draws upon nature's methods of natural selection to produce new solutions to problems. For this reason they have been very widely studied and applied. Because of their very general nature and the fact that they don't need special information like derivatives, the range of potential applications is virtually unlimited. Books such as [6,7,8] describe applications in a vast range of areas including power systems engineering, VLSI design, data mining, condition monitoring of machines, and so on.

Just to finish, I will illustrate for you performance of a GA applied to the best-known NP-complete problem, namely our old friend the traveling salesperson problem. This problem involves 100 cities distributed randomly over a square area. Fig. 11.3 and its caption explain the basic procedure. The program that was used for this example, which comes from [9] is written in Matlab and is available at www.natural-selection.com under the publications section. The GA searches the space of possible tours and eventually finds a very good one. In getting to the solution, the algorithm only examined about 400,000 of the 10^{150} possible tours.



[4] A simple iterative evolutionary algorithm was used to solve a 100-city traveling salesman problem. At the start, the researcher selected 100 possible solutions, each of which was used to generate one offspring by means of inversion [Fig. 3], yielding a total of 200 solutions. The best 100 (in terms of shortest route) were selected, with the top solution from this first generation shown at far left. The best solutions after 500, 1000, and 4000 iterations indicate the progress made as the process was repeated.

Figure 11.3

References

[1] D E Goldberg, “Genetic algorithms in search, optimization and machine learning”, Addison-Wesley, 1989.

[2] Encyclopaedia Britannica section on mutation.

[3] J H Holland, “Hierarchical descriptions of universal spaces and adaptive systems”, Technical Repoert ORA projects 01252 and 08226, Dept. of Computer and Communication Sciences, University of Michigan, 1968.

[4] J H Holland, “Genetic algorithms and the optimal allocation of trials”, SIAM Journal of Computing, **2**, 88-105, 1973.

[5] J H Holland, “Adaptation in natural and artificial systems”, University of Michigan Press, 1975.

[6] T Back, D B Fogel and Z Michalewicz, “Handbook of evolutionary computation”, Oxford University Press, 1997.

[7] C L Karr and L M Freeman, “ Industrial applications of genetic algorithms”,CRC Press, 1999.

[8] K Miettinen et al., “Evolutionary algorithms in engineering and computer science”, Wiley, 1999.

[9] D B Fogel, “What is evolutionary computation?” IEEE Spectrum, **37**, February 2000, 26-32.