Genetic Algorithms

1. Introduction

Genetic Algorithms (GA's) are a relatively new type of algorithm. (Fraser, 1957 and 1960) and (Bremermann, 1958) proposed similar algorithms which simulated genetic systems and much seminal work was also conducted by (Holland, 1992 – reprinted) and his students and colleagues at the University of Michigan in the 1960's and 1970's. Holland's book of 1975 (Holland, 1992 - reprinted) is recognised as one of the seminal work for GA's.
If you want a gentle introduction to GA's, as well as some history of the people involved (particularly John Holland), you might want to see (Levy, 1993).

The handout for this part of the course is presented as follows.
Firstly, an overview of GA's is given. Following this the GA algorithm is presented.
Next the three main parts of a GA are discussed; these being the evaluation module, the population module and the reproduction module.
And finally a list of references is presented that the interested student might find useful. The books I have listed are considered to be the "standard" textbooks for GA's.
Please note, that this handout is largely based on (Davis, 1991), as I find this gives a more complete description than (Russell, 1995), which is the book we have used for much of the course. I also find that (Davis, 1991) is very easy to read and is a very good introductory book to the subject.
Another good introduction is chapter 1 of (Goldberg, 1989). You might also like to look at (Michalewicz, 1996), (Michalewicz, 2000) and (Mitchell, 1996) which are also "standard" GA textbooks.

2. GA's – What are they?

Genetic Algorithms are based on the principles of survival of the fittest; sometimes called natural selection. GA's, informally, work as follows.
Many potential solutions to the problem are created. Each solution is evaluated to see how good it is. The best solutions are allowed to breed with each other.
This cycle continues in the hope that, as we are breeding with the good solutions, we will gradually breed better and better solutions. And the evidence supports that this is the case for a wide variety of problems.

As GA's have their origins in genetics the terms have carried through into the computer counterparts.

Each solution is normally called a chromosome (or an individual). Each chromosome is made up of genes, which are the individual elements that represent the problem. The collection of chromosomes is called a population.
In a problem such as the travelling salesman problem (TSP) the genes would be individual cites. A chromosome would be a complete tour and the population would be a number of complete tours.

As stated above, GA's work by breeding the best individuals from the current population. Let's consider this is a bit more detail.

It is common to have a population of one hundred. We'll work with that figure for now.

The initial population of one hundred chromosomes is normally created randomly. Many problems are represented as bit strings, but this need not be the case and other representations have been used.
The individuals that make up the population are then evaluated. That is, some function considers each chromosome and assigns it a value depending on how good an answer it is to the problem. The chromosomes are then allowed to breed. But what does this mean?
Firstly we need to choose suitable "parents." Overall we want to choose the fittest members of the population to breed but, as in life, everybody must be given the chance to breed. Just because an individual has a low evaluation does not mean he/she does not have something to contribute to society.

Therefore, in choosing parents, it is normally done in such a way that they are chosen in proportion to their evaluation rating. In doing this the fitter individuals are more likely to breed but the weaker members of the population also have the opportunity.

Having chosen our parents we now allow them to breed. This means (normally) two offspring are produced from the two parents. The children consist of genetic material taken from both parents. How the genetic material is distributed can be done in a number of ways, which we discuss below.

However, we do not breed all the time. There is a probability associated with each breeding pair as to whether they produce children or not. The probability of breeding is usually set to something like sixty percent but other figures are often experimented with.

Occasionally, as in life, a mutation occurs and GA's also mimic this.
Mutation happens with low probability and how the mutation occurs depends on the coding that is being used. If the problem is being represented by bit strings then mutation is fairly easy to implement. It can simply look at each bit in the chromosome and decide (with some low probability) if the bit should be replaced with a randomly produced bit.

The last point that should be made about GA's is that they have no knowledge about the problem it is trying to solve. The only part of the GA that has some domain knowledge is the evaluation function. This function is given a chromosome and passes back an evaluation for the chromosome. It is this evaluation rating that the breeding mechanism uses in deciding which chromosomes should breed.
The breeding mechanism has no knowledge about the problem. In its simplest form a GA is just manipulating bit strings.

3. The GA Algorithm

A Genetic Algorithm can be implemented using the following outline algorithm

1. Initialise a population of chromosomes
2. Evaluate each chromosome (individual) in the population
    2.1. Create new chromosomes by mating chromosomes in the current population (using crossover and mutation)
    2.2. Delete members of the existing population to make way for the new members
    2.3. Evaluate the new members and insert them into the population
3. Repeat stage 2 until some termination condition is reached (normally based on time or number of populations produced)
4. Return the best chromosome as the solution.

This is the basic GA algorithm. As we shall see below there are many parameters that we can use to affect this basic algorithm.

4. Evaluation Module

The evaluation module is responsible for evaluating a chromosome.
It is very important to note that this is the only part of the GA that has any knowledge about the problem that is to be solved. The rest of the GA modules are simply operating on (typically) bit strings with no information about the problem.
As the evaluation module needs to know about the problem a different evaluation module will be needed for each type of problem, but the rest of the algorithm can remain the same.
As an example, a GA that is trying to solve the travelling salesman problem (TSP) would evaluate each chromosome based on how long the tour distance was.
For those of you that don't know the travelling salesman problem it can be simply stated.
A salesman has to visit a number of cities. He can start at whichever city he likes but he must also end at that city. He must visit all the other cities only once. The problem is to find the shortest possible tour.

6. Population Module

6.1 Introduction

The population module is responsible for maintaining the population.
It does this using the following techniques and parameters

6.2 Initialisation Technique

This technique determines how the initial population is initialized. It is often the case than a random initialisation is done. In the case of a binary coded chromosome this means that each bit is initialised to a random zero or one.
But there may be instances where you initialise the population with some known good solutions.
This might be applicable where, for example, you know of a good solution but you want to try and improve on it. If you ***seed*** the population with these good solutions you should be able to discover better solutions.
Of course, this may not happen and the seed solutions may actually disappear from the population.
However, if you combine seeding the population with elitism (see below) then you are guaranteed to at least find the same good solutions and not have to suffer the embarrassment of finding a solution worse that than those you stared with.

6.3 Deletion Technique

This technique determines how the population is deleted at each generation of the GA.
Three common deletion techniques are

Delete-All : This technique deletes all the members of the current population and replaces them with the same number of chromosomes that have just been created.

Steady-State : This technique deletes *n* old members and replaces them with *n* new members. The number to delete and replace (*n*) at any one time is a parameter to this deletion technique.
Another consideration for this technique is deciding which members to delete from the current population. Do you delete the worst individuals, pick them at random or delete the chromosomes that you used as parents?

Steady-State-No-Duplicates : This is the same as the steady-state technique but the algorithm checks that no duplicate chromosomes are added to the population. This adds to the computational overhead but can mean that more of the search space is explored.

6.4 Parent Selection Technique

When breeding new chromosomes you need to decide which chromosomes to use as parents. You need to use the fittest individuals from the population but you also want to sometimes use less fit individuals so that more of the search space is explored and to increase the chances of producing promising off spring. The danger of always using, say, the top few chromosomes is that the population quickly converges to one of these individuals and you return an inferior solution.
Two common parent selection techniques are described below

Roulette Wheel Selection : The idea behind the roulette wheel selection parent selection technique is that each individual is given a chance to become a parent in proportion to its fitness evaluation. It is called roulette wheel selection as the chances of selecting a parent can be seen as spinning a roulette wheel with the size of the slot for each parent being proportional to its fitness. Obviously those with the largest fitness (and slot sizes) have more chance of being chosen.

Mathematically, the probability of a chromosome being chosen can be represented as follows.

$p_i = f_i / \sum f_i$ (for all $_i$)

where f_i are the individual fitnesses

Roulette wheel selection can be implemented as follows

1. Sum the fitness of all the population members. Call this f_{sum}.
2. Generate a random number r, between 0 and f_{sum}.
3. Add together the fitness of the population members (one at a time) stopping immediately when the sum is greater than r. The last individual added is the selected individual

As we saw from the lectures, the problem with roulette wheel selection is that one member can dominate all the others and get selected a high proportion of times. The reverse is also true. If the evaluation of the members are very close then they will have an almost equal chance of being selected. To get round these problems various fitness normalisation techniques can be used. These are described below

Tournament Selection : Another method of selecting parents, and which has been used with some success, is tournament selection. In effect, potential parents are selected and a tournament is held to decide which of the individuals will be the parent. There are many ways this can be achieved and two suggestions are

1. Select a pair of individuals at random. Generate a random number, R, between 0 and 1. If $R < r$ use the first individual as a parent. If the $R >= r$ then use the second individual as the parent. This is repeated to select the second parent. The value of r is a parameter to this method.
2. Select two individuals at random. The individual with the highest evaluation becomes the parent. Repeat to find a second parent.

6.5 Fitness Technique

As mentioned above, using the evaluation to choose parents can lead to problems. For example, if one individual has an evaluation that is higher than all the other members of the population then that chromosome will get chosen a lot and will dominate the population. Similarly, if the population has almost identical evaluations then they have an almost equal chance of being selected, which will lead to an almost random search.

In order to solve this problem, each chromosome is sometimes given two values, an evaluation and a fitness. The fitness is a normalised evaluation so that parent selection is done more fairly. Some of the methods for calculating fitness are described below.

Fitness-Is-Evaluation : It is common to simply have the fitness of the chromosome equal to its evaluation.

Windowing : The windowing evaluation technique takes the lowest evaluation and assigns each chromosome a fitness equal to the amount it exceeds this minimum.

Linear Normalization : The chromosomes are sorted by decreasing evaluation value. Then the chromosomes are assigned a fitness value that starts with a constant value and decreases linearly. The initial value and the decrement are parameters to the techniques.

6.6 Population Size

This parameter determines how many chromosomes should be in the population at any one time.

6.7 Elitism

It is sometimes the case that a good solution is found early on in the GA run but gets deleted from the population as the GA progresses. One solution is to "remember" the best solution found so far. Alternatively a technique called elitism can be used.
This technique ensures that the best members of the population are carried forward from one generation to the next.
It is usual to supply a parameter to the elitism function that says what percentage of the population should be carried over from one generation to the next.

7. Reproduction Module

7.1 Introduction

The reproduction module is responsible for the breeding of chromosomes. Typically, the reproduction module will ask the population module for two parents. These two parents will breed and the children will be passed back to the population module to be added to the population.
Reproduction is done by operators, with mutation playing a lesser, but still important role.

7.2 Operators

The first GA operator to be developed was one-point crossover. The other operators have been added as the GA field has developed. There have also been operators developed for specific problems. This really gets away from the GA ideal (where only the evaluation function has knowledge of the problem domain) and they will not be considered here.

One-Point Crossover

One-point crossover takes two parents and breeds two children. It works as follows

Parent 1	***1***	0	1	1	1	0	1
Parent 2	1	1	0	0	1	1	***0***

Child 1	1	0	0	0	1	1	0
Child 2	1	1	1	1	1	0	1

(ignore the highlighted bits for now)

- Two parents are selected.
- A ***crossover point*** is chosen at random (shown above by the dotted line).
- Child 1 is built by taking genes from parent 1 from the left of the crossover point and genes from parent 2 from the right of crossover point.
- Child 2 is built in the same way but it takes genes from the left of the crossover point of parent 2 and genes from the right of the crossover point of parent 1.

You may have seen a powerpoint demonstration of one point crossover in the lectures.

Two-Point Crossover

Two-point crossover works in a similar way as one point crossover but two crossover points are selected. It is also possible to have *n*-point crossover. Two-point crossover is considered beneficial when, for example, the highlighted bits (in the above example) would provide a good solution if they were next to each other

(in fact this is a bad example and really we should be looking at several bit combinations; called schemata; which would benefit from being closer). Using one-point crossover this can never happen but two-point crossover will allow this.

Uniform Crossover

For each bit position of the two children we decide, at random, which parent will contribute its bit value to that child.
The algorithm can be implemented as follows.

Parent 1	1	0	1	1	1	0	1
Parent 2	1	1	0	0	1	1	0
Template	0	1	1	0	0	1	0

Child 1	1	0	1	0	1	0	0
Child 2	1	1	0	1	1	1	1

- Two parents are selected
- A template is created which consists of random bits
- Child 1 receives bits from parent 1 indicated by a one in the template and bits from parent 2 indicated by a zero in the template.
- Child 2 is built in the same way but receives bits from parent 1 where there is a zero in the template and bits from parent 2 when there is a one in the template.

Order-Based Crossover

The problem with the above crossover operators is that they can lead to illegal solutions for some problems. Take, for example, the travelling salesman problem (TSP). A chromosome will be coded as a list of towns. If we allow the one-point crossover operator we can (and almost definitely will) produce an illegal solution. In this instance we will duplicate some cities and delete some others.
There are two approaches to dealing with this problem. We can implement a repair function which takes a potential solution and examines it. If it finds it is an illegal solution it carries out a repair to ensure that it is valid. Obviously, the type of repair that is carried out is problem dependent.
Another approach to this problem is to develop crossover operators that do not produce illegal solutions. This can be done by using domain information but we can also develop generic crossover operators that work on a whole range of problems. Order-based crossover is one of these operators.

Order-based crossover ensures that, in the case of the travelling salesman problem, that no cities are duplicated or lost. It works as follows (assume the coding scheme represents cities)

Parent 1	A	B	C	D	E	F	G
Parent 2	E	B	D	C	F	G	A
Template	0	1	1	0	0	1	0

Child 1	E	B	C	D	G	F	A
Child 2	A	B	D	C	E	G	F

- Select two parents

- A template is created which consists of random bits
- Fill in some of the bits for child 1 by taking the genes from parent 1 where there is a one on the template (at this point we have child 1 partially filled, but it has some "gaps").
- Make a list of the genes in parent 1 that have a zero in the template
- Sort these genes so that they appear in the same order as in parent 2
- Fill in the gaps in child 1 using this sorted list.
- Create child 2 using a similar process

Partially Matched Crossover (PMX)

PMX is similar to order-based crossover in that it can be used for problems such as the travelling salesman problem. An example is probably the best way to shows how PMX operates. This example is taken from (Goldberg, 1989).
Given two parents, A and B, two crossover points are chosen.

Parent A	9	8	4	5	6	7	1	3	2	10
Parent B	8	7	1	2	3	10	9	5	4	6

Now, looking at B, we consider the genes between the crossover site. The first, 2, maps to 5 in parent A. Therefore, we swap gene 2 with gene 5 in parent B. Following the same principle we swap 3 and 6 and 10 and 7.
A similar procedure is followed for parent A (swapping 5 and 2, 6 and 3 and 7 and 10). The resulting chromosomes are shown below.

Parent A	9	8	4	2	3	10	1	6	5	7
Parent B	8	10	1	5	6	7	9	2	4	3

You can see, that what we have achieved is to duplicate the ordering in one gene in the other gene – and vice versa.
One of the benefits of PMX is that it tends to preserve absolute city positions, whilst taking ordering from the other parent. It is for this reason that this operator was used in a two dimensional stock cutting problem that I may have mentioned in the lectures.

Cycle Crossover

Yet another crossover operator that allows it to be used for the TSP is the cycle operator. This operates as follows. Again this example is taken from (Goldberg, 1989). The idea behind this crossover is that each city (to use TSP speak) must come from one parent or the other. It works as follows. Given two parents, A and B, we choose the first city from A.

Parent A	9	8	2	1	7	4	5	10	6	3
Parent B	1	2	3	4	5	6	7	8	9	10

As we have chosen 9 from A we now choose 1, as this maps onto the same position in B as in A. We place this city in the same position it currently occupies in A.

Parent A	9			1						

City 1 in A maps to city 4 in B, so we place city 4 in A in the same position it occupies. This gives

Parent A	9			1		4				

We continue this process once more and arrive at

Parent A	9			1		4			6	

At this point we should take city 9 and place it in A, but this has already been done, so we have completed a cycle; which is where this operator gets its name. The missing cities are filled from B giving,

Parent A	9	2	3	1	5	4	7	8	6	10

If we perform a similar crossover operation with B, we end up with

Parent B	1	8	2	4	7	6	5	10	9	3

7.3 Mutation

If we use a crossover operator such as one-point crossover we may get better and better chromosomes but the problem is, if the two parents (or worse – the entire population) has the same value in a certain position then one-point crossover will not change that. In other words, that bit will have the same value forever. Mutation is designed to overcome this problem and add some diversity to the population.

The most common way that mutation is implemented is to select a bit at random and randomly change it to a zero or a one. Note that using this method it is possible to change the bit to the same value that it already has. Some implementations “flip” the bit. This, in effect, doubles the mutation rate (see below).

Other mutation operators may swap parts of the gene or may develop problem specific mutation operators.

7.4 Mutation Rate

This is a parameter to the GA algorithm. If defines how often mutation should be applied. A typical value is 0.008. Therefore, when presented with a chromosome the mutation sweeps down the bit string and each bit has one chance in 8000 of being mutated.

7.5 Crossover Rate

The crossover rate defines how often crossover should be applied. A typical value if 0.6. This means that when presented with two parents there is a 60% chance that the parents will breed.

7.6 Dynamic Mutation and Crossover Rates

During GA research it has been found that crossover is more important in the early stages of the GA run, whilst the population searches large areas of the search space. Later in the GA run, when the GA is exploring a smaller area it has been found that mutation is more important.
With this in mind it is sometimes appropriate to change the mutation and crossover rates as the run progresses so that there is a low probability of mutation in the early stages and higher probability later in the run. Similarly the crossover rate starts off with a high probability and decreases as the run progresses.

8. Example

To give you some idea as to a GA operates, below is presented a simple example. This example is based on (Reeves, 1995), although we are going to use roulette wheel selection – as described above so that the example better relates to the rest of this handout.

In order to demonstrate the GA we are going to use a crossover probability, $P_C = 1.0$ and a mutation probability, $P_M = 0.0$. That is we always apply crossover, but never apply mutation.

Assume we have a function

$f(x) = x^3 - 60 * x^2 + 900 * x + 100$

and we wish to maximise it.
Further assume that we limit the value of x to 0..31. If you are interested, a spreadsheet is available on the web site for this course which shows this function. By looking at the spreadsheet and the associated graph, you can see the shape of the function as well as see where the maximum value for x lies. If you look at the spreadsheet you will also see why we limit x to 0..31 (see what happens when x > 40). Of course limiting x to a small value also allows us to demonstrate a GA more easily.

Assuming a binary representation we can represent x using five binary digits.

Assume we set our population size to four and generate the following random chromosomes.

Chromosome	Binary String	x	f(x)
P_1	11100	28	212
P_2	01111	15	3475
P_3	10111	23	1227
P_4	00100	4	2804
	TOTAL		**7718**
	AVERAGE		**1929.50**

The next stage is to decide which parents to breed. In this simplified GA we will choose two parents and let them create two new individuals. We will do this twice, at which point we will have created a new population.

The first round of breeding could do the following

Roulette Wheel	Parent Chosen	Crossover Point
4116	P_3	N/A
1915	P_2	1

Where Roulette Wheel is a random number between 0 and 7718
Parent Chosen is the parent chosen using the roulette wheel selection method
Crossover Point is a random number that decides where crossover will take place

Applying single point crossover we arrive at two children

C_1 = 11111
C_2 = 00111

We carry out the same procedure again to create C3 and C4

Roulette Wheel	Parent Chosen	Crossover Point
5802	P_4	N/A
1349	P_2	2

C_3 = 00111
C_4 = 01100

We can now re-evaluate these new solutions, which form a new population (and we call them P1..P4 again)

Chromosome	Binary String	x	f(x)
P_1	11111	31	131
P_2	00111	7	3803
P_3	00111	7	3803
P_4	01100	12	3998
	TOTAL		**11735**
	AVERAGE		**2933.75**

Although this is a very simple experiment we can already notice a few things.

- The average evaluation has risen.
- P_2, which was the strongest individual in the initial population, was chosen both times but we have lost it from the population.
- We have a value of x=7 in the population which is the closest value to 10 we have found.

You might like to continue with this experiment so that you fully understand it.

You might also like to answer this question.

Assume the initial population was 17, 21, 4 and 28. Using the same GA methods we used above (P_C = 1.0, P_M = 0.0), what chance is there of finding the global optimum.
The answer is at the end of this handout.

9. Schema Theorem

9.1 Introduction

When John Holland introduced the world to GA's he also developed a theoretical analysis which was based on a ***schema***; and thus the schema theorem. The questions he was trying to answer is how likely a schema is to survive from one generation to the next; and how many schema are likely to be present in the next generation.

Consider the following two individuals

0 0 0 1 0 1 1
0 1 0 0 0 0 1

Both of these are examples of the following schema

* * 0 * 0 * 1

where the '*' is a wildcard and can be replaced by a zero or a one. Another way of thinking of a schema is to consider it as a subset of all possible chromosomes.
The chromosomes also form parts of other schemata (for example the left most bits are the same in both chromosome).

If a chromosome is of length *n* then it contains 3^n schemata (as each position can have the value 0, 1 or *). This means that every time we evaluate the fitness of an individual we are getting information about each of the schemata that exists in the chromosome. In theory, this means that for a population of *M* individuals we are evaluating up to $M3^n$ schemata. Although, it should be borne in mind that some schemata will not be represented and others will overlap with other schemata. In fact, this is exactly what we want. We eventually want to create a population that is full of fitter schemata and we will have lost weaker schemata.

It is the fact that we are manipulating *M* individuals but $M3^n$ schemata that gives genetic algorithms what has been called ***implicit parallelism***.

We need to define three terms before we can continue our discussion.

Length : is defined as the distance between the start of the schema and the end of the schema minus one. This method of calculating the length of a schema is defined by (Goldberg, 1989). Therefore, the above schema has a length of 4.

Order : is defined as the number of defined positions. In the above schema the order is 3.

Fitness Ratio : is defined as the ratio of the fitness of a schema to the average fitness of the population.

Before looking at a little of the mathematics underlying schema theory, we can make an intuitive observation.

The longer the length of the schema, the more chance there is of the schema being disrupted by a crossover operation. This implies that shorter schemata have a better chance of surviving from one generation to the next. In turn, this implies that if we know that certain attributes of a problem fit well together then these should be placed as close as possible together in the coding.

This observation is also true for the order of the chromosome. If we are not worried about the number of defined positions (i.e. we allow as many '*' as possible) then a crossover operation has less chance of disrupting good schemata.

In short, intuitively, it would seem better to have short, low-order schema. At present, this observation is only based on empirical evidence (see for example Goldberg, 1989) but it is widely believed that these assumptions are true and the following theory makes some sense of this.

We can now present some of the mathematics that supports the schema theory. We will not be presenting the proofs to the theories – just discussing some of the main points.

9.2 Reproduction of Schemata

In the discussions above, although we discussed a reproduction module, we did not mention reproduction explicitly. In fact, using a technique where we choose parents relative to their fitness (e.g. roulette wheel selection), fitter schema will find their way from one generation to another.

Intuitively, if a schema is fitter than average then it should not only survive to the next generation but should also increase its presence in the population.

We can formalise this as follows

If Φ is the number of instances of any particular schema S within the population at time *t*, then

$$\Phi(S, t) > 0$$

At time $t + 1$ we would expect

$$\Phi(S, t+1) > \Phi(S)$$

to hold true for above average fitness schemata. It should be pointed out that this is only an estimate due to the random elements of the reproduction process.

We can go one stage further and estimate the number of schema present at $t+1$.

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

where *n* is the size of the population
f(S) is the fitness of the schema
Σf_i is the fitness of the population

This can be simplified to

$$\Phi(S,t+1) = \Phi(S,t)\frac{f(S)}{f_{avg}}$$

where f_{avg} is the average fitness of the population

If a particular schema stays a constant, *c*, above the average we can say even more about the effects of reproduction.

$$\Phi(S,t+1) = \Phi(S,t)\frac{(f_{avg} + cf_{avg})}{f_{avg}}$$

$$= \Phi(S, t)(1 + c)$$

If we now set $t = 0$, we can rewrite this formula as

$$\Phi(S, t) = \Phi(S, t)(1 + c)^t$$

Notice that the number of instances of a schema rises exponentially.

Therefore, we can conclude that reproduction allocates an exponentially increasing number of schemas with each generation.
On a spreadsheet, available via the course web site, this formula has been implemented so that you can experiment with various parameter values.

9.3 Probability of Non-Disruption through Crossover

Given a schema, what is the probability of it ***not*** being disrupted by a crossover operation? This can be calculated using the following formula, P_{NC} (probability of non-disruption by crossover)

$$P_{NC} = 1 - \frac{P_C l(S)}{n-1}$$

where
P_C is the probability of crossover,
$l(s)$ is the length of the schema,
and n is the length of the chromosome.

Take a look at this, where the schema is within the thicker border.

Assume $P_C = 1$. Then l(s) = 4 and n = 11.
The probability of the schema being disrupted by a crossover operation is 1- 1 x 4 / 10 = 0.6.

We can easily confirm this by seeing that there are six crossover positions, of a possible ten (we assume we do not pick crossover points at the "outside") that will not disrupt the schema.
If we now make things a little more complex by setting the crossover probability, P_C, to, say 0.6, we can recalculate P_{NC} as 0.76.

But, this is not the end of the story. We might choose a crossover point that is in the middle of the schema but the chromosome we are crossing over with might have the same schema. This will result in the schema remaining intact, even though crossover point is in the middle of the schemata, so we need to include this in the calculation.

The probability that the schema in the other parent is an instance of a ***different*** schema is given by (1-P[S,t]), where P[S, t] is the probability that the schema in the other parent is the same as the schema in the initial parent.

What we need to do is multiply our original definition of P_{NC} by the probability it is an instance of a different schema.

This leads to a new definition of P_{NC}.

$$P_{NC} = 1 - \frac{P_C l(S)}{n-1}(1 - P[S,t])$$

So, let's work through a couple of examples, to show this formula is doing what we require.

Using the schema shown above, assume the following
P_C = 1
$l(s)$ = 4
n = 11
$P[S, t]$ = 1 (i.e. the other parent's schema is the same as the initial parent – therefore we would expect the schema to appear in the next population)

If you run these figures through P_{NC} you will get 1. That is there is a probability of 1 (i.e. it ***will*** happen) that the schema will not be disrupted or, to put it another way, the schema is guaranteed to survive. If we change P[S, t] to zero the probability changes to 0.6, which is the same probability we had before when we just applied crossover.
It follows that if we set P[S, t] to values between zero and one then the overall probability of non-disruption will change accordingly.

If you experiment with these formulae you will observe that shorter schemata have a higher chance of surviving, which agrees with the intuitive comments we made earlier.
On a spreadsheet, available via the course web site, these formulae have been implemented so that you can experiment with various parameter values.

9.4 Probability of Non-Disruption through Mutation

We also need to consider the probability of a schema surviving a mutation operator. We'll work with this chromosome/schema

As mutation can be applied to all the genes in a chromosome we do not need worry about the length of the chromosome, nor do we need worry about the length of the schema. However, we are concerned with the order, $k(s)$. We may have a schema of length 100 but only of order 2. It is only the bits that are defined within the schema that are of concern to us. The "don't care" (*'s) can be mutated without affecting the schema.
Therefore the probability of a single bit within a schema surviving mutation is

$$1 - P_M$$

and the probability of the schema surviving mutation is

$$1 - P_M^{K(S)}$$

If we assume that $P_M = 0.01$ then the probability of the above schema surviving is 0.97.

Notice that if the schema had a higher order (say K(S) = 100), then the probability of the schema surviving is decreased 0.366, which demonstrates that short schema have a better chance of surviving.

We can approximate this formula so that

$$1 - P_M^{K(S)} \approx 1 - K(S)P_M$$

On a spreadsheet, available via the course web site, these formulae have been implemented so that you can experiment with various parameter values.

9.5 Schema Theory

The schema theory is simply a combination of the above and tells us how likely a particular schema is to survive into the next generation

$$\Phi(S,t+1) \geq \Phi(S,t)\frac{f(S)}{f_{avg}}\left[1-\frac{P_{Cl}(S)}{n-1}(1-P[S,t]) - \mathrm{K(S)P_M}\right]$$

In words, we are saying that the probability of a schema surviving is a function of the expected number of schema surviving reproduction, crossover and mutation.

At the start of this section we made an intuitive comment that short, low order schemata should have a better chance of surviving. If you followed the discussions above, I hope you can see that this is indeed the case.

On a spreadsheet, available via the course web site, the schema theory has been implemented so that you can experiment with various parameter values.

10. Coding Schemes

When applying a GA to a problem one of the decisions we have to make is how to represent the problem. The classic approach is to use bit strings and there are still some people who argue that unless you use bit strings then you have moved away from a GA.
In fact, researchers are gradually coming around to the view that bit strings are no better than any other representation. However, sometimes, bit strings are useful otherwise we get into all sorts of complications as to how to represent integers and how do we define a neighbourhood move.
Just consider the problems if we were dealing with real numbers and trying to define a neighbourhood. It is by no means impossible but bit strings seem like a good coding scheme if we can represent our problem using this notation.

Take a look at this table.

Decimal	Binary	Gray Code
0	000	000
1	001	001

2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

It shows the decimal numbers from (0..7), with their binary representations and also another possible binary representation (Gray codes).
Gray codes have the property that adjacent integers only differ in one bit position. Take, for example, decimal 3. To move to decimal 4, using binary representation, we have to change all three bits. Using the gray code only one bit changes.

(Hollstien, 1971) investigated the use of GAs for optimizing functions of two variables and claimed that a Gray code representation worked slightly better than the binary representation. He attributed this difference to the adjacency property of Gray codes. In general, adjacent integers in the binary representation often lie many bit flips apart (as shown with 3 and 4). This fact makes it less likely that a mutation operator can effect small changes for a binary-coded chromosome

A Gray code representation seems to improve a mutation operator's chances of making incremental improvements, and a close examination suggests why. In a binary-coded string of length N, a single mutation in the most significant bit (MSB) alters the number by 2^{N-1}. In a Gray-coded string, fewer mutations lead to a change this large. The use of Gray codes does, however, pay a price for this feature: those "fewer mutations" lead to much larger changes. In the Gray code illustrated above, for example, a single mutation of the left-most bit changes a zero to a seven and vice-versa, while the largest change a single mutation can make to a corresponding binary-coded individual is always four. One might still view this aspect of Gray codes with some favor: most mutations will make only small changes, while the occasional mutation that effects a truly big change may allow exploration of a new area of the search space.

If you are interested, the algorithm for converting between the Gray code described above (there are others) and the decimal binary representation is as follows.

- Label the bits of a binary-coded string B[i], where larger i's represent more significant bits
- Label the corresponding Gray-coded string G[i]
- Convert one to the other as follows
  - Copy the most significant bit
  - For each smaller i do G[i] = XOR(B[i+1], B[i]) (to convert binary to Gray)
    Or
  - B[i] = XOR(B[i+1], G[i]) (to convert Gray to binary)

For the interested student other references to Gray codes include (Horowitz, 1989), (Kozen, 1992), (Press, 1992) and (Reingold, 1977)

11. Hybridisation

Hybridisation is discussed in the simulated annealing handout and the information is not repeated here.

12. Glossary

This small glossary is supplied simply to collect some of the important terms together. Some terms, which you may read in the literature but not covered in this handout, are also given.

Allele	:	The possible values that can be taken by a gene are called alleles.
Chromosome	:	An individual within the population. You may also see the following terms, which means the same thing; individual, solution, strings and vectors.
Gene	:	Genes are the basic building blocks which form chromosomes. In the examples in this handout, we have mainly considered bit strings. Each bit is a gene.
Genotype	:	The genotype is the expression of the chromosome. The classical representation is bit strings but many other representations and data structures are possible.
Locus	:	The position of a variable within a chromosome is called its locus.
Phenotype	:	The phenotype is the physical expression of the chromosome. For example, a chromosome might consist of bit strings but it could actually represent integers or real numbers and those could represent anything.
Population	:	A set of chromosomes that represent a pool of solutions that are current being considered.

13. References

1. Coley, D. A. 1999. An Introduction to Genetic Algorithms for Scientists and Engineers, World Scientific Publishing
2. Davis, L., 1991. Handbook of Genetic Algorithms. Van Nostrand Reinhold
3. Fraser, A.S. 1957. Simulation of genetic systems by automatic digital computers. II. Effects of linkage on rates under selection. Australian J. of Biol Sci, vol 10, pp 492-499
4. Fraser, A.S. 1960. Simulation of genetic systems by automatic digital computers. IV. Epistatis. Australian J. of Biol Sci, vol 13, pp 329-346
5. Bremermann, H.J. 1958. The Evolution of Intelligence. The Nervous System as a Model of its Environment. Technical Report No. 1, Contract No. 477(17), Dept. of Mathematics, Univ. of Washington, Seattle.
6. Goldberg, D. 1989. Genetic Algorithms in Search, Optimization, and Machine Learning.
7. Gray, F. 1953. Pulse Code Communication, U. S. Patent 2 632 058, March 17, 1953.
8. Holland, John H. 1992. Adaptation in natural and artificial systems.
9. Hollstien, R.B. 1971. Artificial Genetic Adaptation in Computer Control Systems, PhD thesis, University of Michigan
10. Horowitz, P. and Hill, Winfield. 1989. The Art of Electronics. Second Edition, Cambridge University Press
11. Kozen, D. 1992. The Design and Analysis of Algorithms. Springer-Verlag, New York, NY
12. Levy, S. 1993. Artificial Life : The Quest for a New Creation. Penguin Books, pp 153-187
13. Michalewicz, Z. 1996. Genetic algorithms + Data Structures = Evolution Programs. 3rd ed. ISBN 3-540-60676-9
14. Michalewicz, Z and Fogel, D.B. 2000. How To Solve It. Springer-Verlag. ISBN 3-540-66061-5
15. Mitchell, M. 1996. An Introduction to Genetic Algorithms. MIT
16. Press, W.H. Press, et al. 1992. Numerical Recipes in C. Second Edition, Cambridge University Press
17. Reeves, C.R. 1995. Modern Heuristic Techniques for Combinatorial Problems. McGraw Hill
18. Reingold, E.M., et al.. 1977. Combinatorial Algorithms. Prentice Hall, Englewood Cliffs, NJ
19. Russell, S., Norvig, P. 1995. Artificial Intelligence A Modern Approach. Prentice-Hall. ISBN 0-13-103805-2

14. Answers

Assume the initial population was 17, 21, 4 and 28. Using the same GA methods we used above (P_C = 1.0, P_M = 0.0), what chance is there of finding the global optimum.

If we look at the values in binary we get

x	Binary
17	10001

21	10101
4	00100
28	11100

We know (although for any realistic problem we would not) that the global optimum is x = 10 which, in binary is 01010.
You can see that we need a 1 in positions 2 and 4 (counting from the right – although in this case it does not matter). In the initial population there is no individual with a 1 in position 2. This means that no matter how many times we apply single point crossover we will never be able to find the optimum. In addition, only the last individual has a 1 in position 4. This means that unless this individual is selected in the first round of breeding , then this bit will be lost. As the individual has an evaluation of 212 (compared to 17 = 2973, 21 = 1801 and 4 = 2804), it has the least chance of being selected.