e-PGPathshala

Subject : Computer Science

Paper: Machine Learning

Module: Genetic Algorithms - II

Module No: CS/ML/22

Quadrant I – e-text

Welcome to the e-PG Pathshala Lecture Series on Machine Learning. In this module we will discuss some more details about the operators of Genetic Algorithms like crossover, mutation and selection. We will also look into some preliminaries of genetic programming.

Learning Objectives:

The learning objectives of this module are as follows:

- To explore the operators of Genetic Algorithms like crossover, mutation and selection
- To understand the fitness function and the termination condition
- To understand Genetic Programming

22.1 Introduction

A genetic algorithm maintains a population of candidate solutions for the problem at hand, and evolves to find the optimized solution by iteratively applying a set of stochastic operators. The stochastic processes include selection, recombination and mutation. Selection replicates the most successful solutions found in a population at a rate proportional to their relative quality in terms of their fitness values. Recombination decomposes two distinct solutions and then randomly mixes their parts to form novel solutions. Mutation on the other hand randomly perturbs a candidate solution. We will look at the above aspects and other important issues associated with genetic algorithms including the way the problem is represented.

22.2 Representation of the Candidate Solution

Genetic Algorithm must represent the possible candidate solutions.The way the chromosome is encoded/decoded seriously affects precision and accuracy of the genetic approaches. The most critical decision in any application, is the decision of the best way to represent a candidate solution of the algorithm. This representation encodes the phenotype (candidate solution) to a string corresponding to the genotype (Figure 22.1)

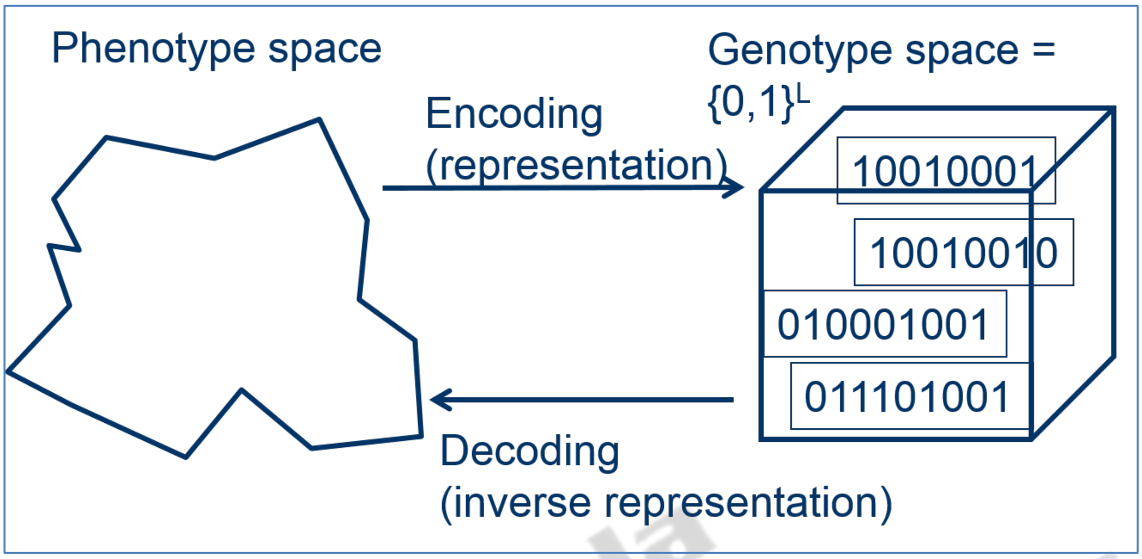
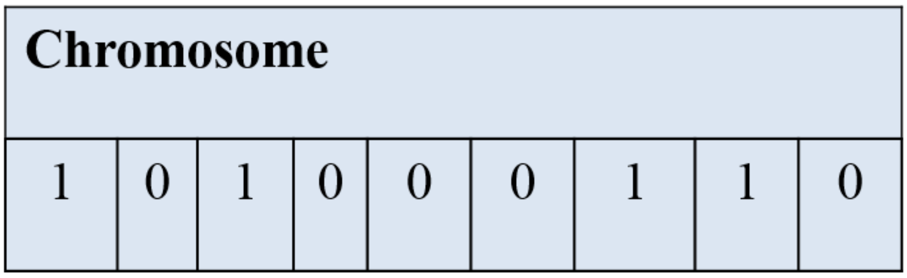


Figure 22.1 Encoding

22.2.1 Binary Encoding

The most widely used representation used in genetic algorithms is calculating values by decoding a binary string. Binary-Coded GAs must decode a chromosome into a candidate solution, evaluate the candidate solution and return the resulting fitness back to the binary-coded chromosome representing the evaluated candidate solution. If we were to use binary-coded representations we would first need to develop a mapping function form our genotype representation (binary string) to our phenotype representation that is our candidate solution. An example is given in Figure 22.2. Let us consider the problem of counting the ones in a chromosome (Onemax problem). Here the number of ones in the genotype given in Figure 22.2 is 4 and hence the phenotype is 4 irrespective of where the ones occur.



Chromosome								
1	0	1	0	0	0	1	1	0

Figure 22.2 Binary Encoding

22.2.2 Representation - Many-Character and Real Valued Encoding

The most natural way to carry out the encoding is to use an alphabet of many characters or real numbers to form chromosomes. Real-Coded GAs can be regarded as GAs that operate on the actual candidate solution that is the phenotype (Figure 22.3). For Real-Coded GAs, no genotype-to-phenotype mapping is needed. For the genotype given in Figure 22.3, we assume that the problem is addition of weights. In this case the phenotype is 623.16.

Chromosome								
32.15	25	40	67	25.01	67	50	89	128

Figure 22.3 Real-Valued Encoding

22.2.3 Representation - Tree Encoding

This type of tree encoding is used in the process of genetic programming (GP) (to be discussed later). This encoding allows the search space to be open-ended since the tree can grow large in uncontrolled ways and prevent the formation of more structured, hierarchical candidate solutions (Figure 22.4). A genetic algorithm can be used to 'evolve' an expression tree to create a very close fit to the data.

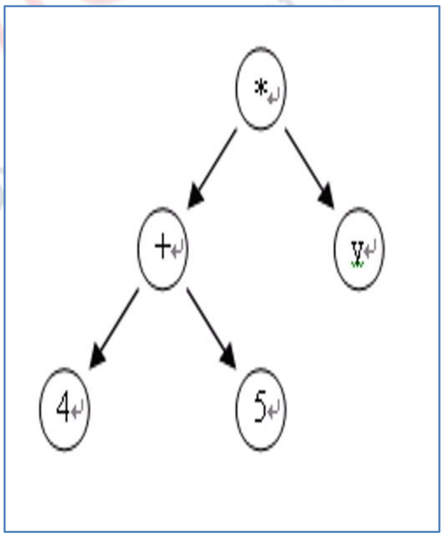


Figure 22.4 Tree encoding

22.2.4 Representation - Permutation Encoding

Permutation encoding is used in scheduling and ordering problems. (E.g., Tabu search). In this representation every chromosome is a string of numbers, which represents numbers in a sequence and this representation is generally used in “ordering problems”. This kind of encoding usually needs to make some corrections after the crossover and mutation operating procedures because any transformation might create an illegal situation (Figure 22.5). An illegal situation in this case is the repetition of the gene “1” in the chromosome which is not allowed in a permutation problem.

Chromosome									
5	6	1	1	4	3	2	8	7	…………

Figure 22.5 Permutation Encoding

22.3 Genetic Operators - Reproduction (Selection)

As already discussed in the previous module, the purpose of selection is to focus the search in promising regions of the space and is inspired by Darwin’s theory “survival of the fittest”. In this section we will discuss possible selection methods.

22.3.1 Roulette Wheel Selection

The most common method of selection is the roulette wheel selection. In roulette wheel selection, individuals are given a probability of being selected that is directly proportionate to their fitness. Two individuals are then chosen randomly based on these probabilities and produce offspring. This method is called the roulette wheel method because each individual is assigned a slice of a circular “roulette wheel”, where the area of the slice is proportional to the individual’s fitness. The wheel is spun N times, where N is the number of individuals in the population. The slice underneath the wheel when it stops determine which individual becomes a parent.

An example is given in Figure 22.6. Here A has a fitness value of 3, B has a fitness value of 2 and C has a fitness value of 1. This spinning introduces the probability aspect. On each spin, the individual under the wheel’s marker is selected to be in the pool of parents for the next step. The selective capability depends on the variance of the fitness in the population. The population converges when the variance of the fitness is low that is the difference between the fitness of the individuals is small. This is because when the probabilities of the different individuals selected are similar it is difficult to select a better individual from individuals with similar fitness values.

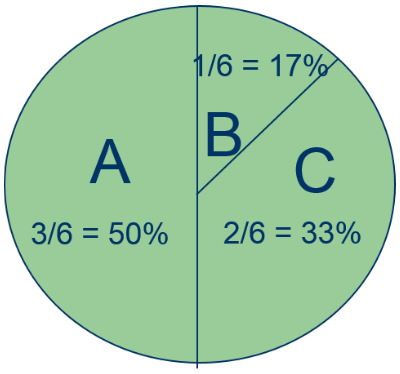


Figure 22.6 Roulette Wheel

22.3.2 Fitness Scaling

There are a number of disadvantages associated with selection based on proportion of fitness as is the case with Roulette wheel. These selection methods cannot be used on directly for minimization problems, but needs to be transformed to an equivalent maximization problem. There is a domination of super individuals in early generations and slow convergence in later generations. Loss of selection pressure that is search direction as population converges is another problem. Fitness scaling has often been used in early days to combat the above problems. We will discuss two types of fitness scaling namely simple scaling and sigma scaling

22.3.2.1 Simple scaling:

In the case of simple scaling the ith individual's fitness is defined as:

$f_{scaled}(t) = f_{original}(t) - f_{worst}(t),$

where t is the generation number and $f_{worst}(t)$ is the fitness of the worst individual so far seen.

22.3.2.2 Sigma scaling:

In the case of sigma scaling the ith individual's fitness is defined as*:*

$$ExpVal(i,t) = \begin{cases} 1 + \dfrac{f(i) - \bar{f}(t)}{2\delta(t)} & if\,\delta(t) \neq 0 \\ 1.0 & if\,\delta(t) = 0 \end{cases}$$

This method keeps the selection pressure relatively constant. An individual's expected value is a function of its fitness, the population mean, and the population standard deviation.

22.3.3 Tournament selection

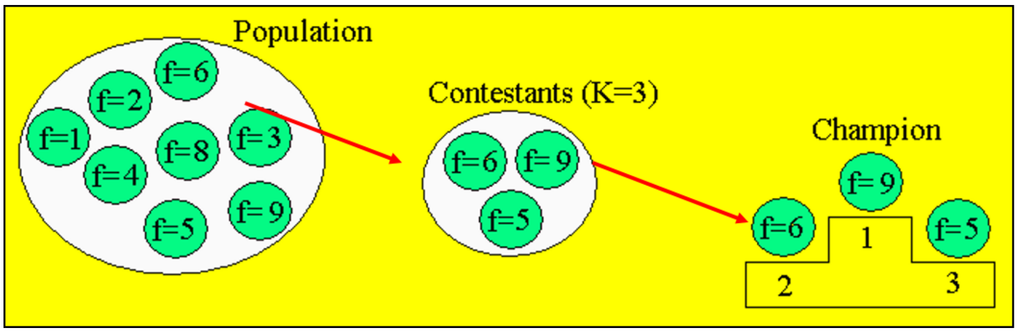


Figure 22.7 Tournament Selection

In this method of selection individuals are chosen at random from the population. A random number *r* is then chosen between 0 and 1. If *r*<*k* (where *k* is a parameter, e.g., 0.75), and the fitter of the individuals is selected as parent. This method has a much faster and better performance than Roulette-Wheel selection. In the example given in Figure 22.7, 3 contestants (f=5,f=6 and f=9) are chosen at random and the fittest among these three (f=9) is chosen as the parent.

22.3.4 Elitism

Elitism is the concept by which while selecting for constructing a new population, we allow the best candidates from the current generation to be present in the next unaltered. This method forces the GA to retain some number of the best individuals at each generation. The best individuals significantly improves the GA's performance.

22.3.5 Steady-State selection

In this method only a few individuals are replaced in each generation that is usually a small number of the least fit individuals are replaced by offspring resulting from crossover and mutation of the fittest individuals. In another method called steady state with no duplicates we do not allow children that are duplicates of chromosomes which already exist in the population. This method is used in evolving rule-based systems in which incremental learning is used.

22.4 Genetic Operator - Crossover

Crossover is concept from genetics and is analogous to reproduction and biological crossover. Crossover combines genetic material from two parents, in order to produce superior offspring. There are many types of crossover depending on the number of points used for the operation.

22.4.1 Single-point crossover

In this type of crossover, given two parents, we select a random cut-point or crossover point. Then each chromosome is broken down into two parts, splitting at the crossover point. The method then recombines the first part of first parent with the second part of the second parent to create one offspring. Single-point crossover then recombines the second part of the first parent with the first part of the second parent to create a second offspring. In crossover a mathematical function is expressed as fitness to indicate success in life.

In the example shown in Figure 22.8, there are two parents P1 (0111011) and P2 (0101101) of seven bits and the single point crossover is after the third bit. Now the first three bits of P1 (011) and the last four bits of P2 (1101) together form the new child (0111101). Now the first three bits of P2 (010) and the last four bits of P1 (1011) together form the new child (0101011).

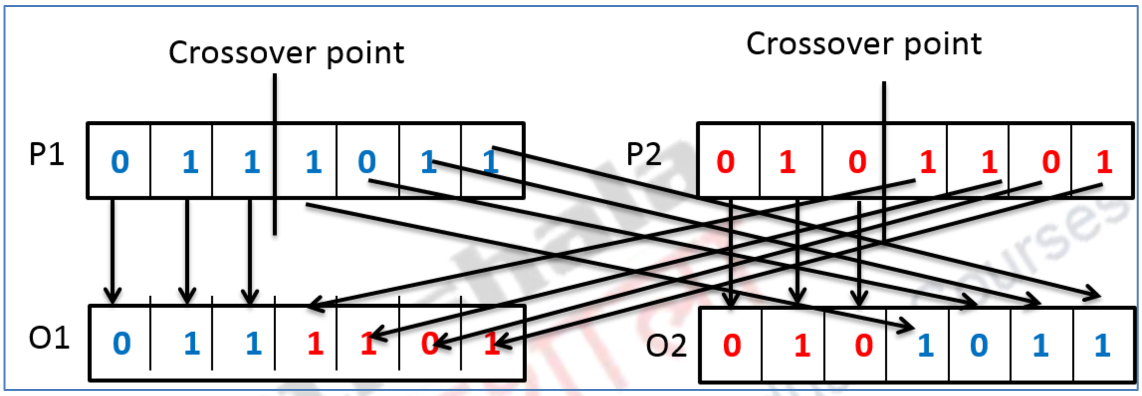


Figure 22.8 Single Point Crossover

One of the problems of single point crossover is the hitchhiking. The spurious correlations or hitchhiking is the phenomenon of some undesirable candidate solutions coupling with desirable ones during recombination, producing an above average individual, which later may get sampled again and so on. The other is the endpoint effect – the end point of the two parents is always exchanged, which is unfair to other points. The endpoint effect can be solved to a certain extent by using multiple-point crossover.

22.4.2 Multiple-point crossover

In multi-point crossover we have more than one point of crossover. Two-Point crossover is very similar to single-point crossover except that two cut-points are generated instead of one. Figure 22,9 shows a three point crossover after 2 bits, after 4 bits and after 6 bits. In this example the child O1 is formed by the first two bits of P1 (01), the next two bits of P2 (01), the next two bits of P1 itself (01) and finally the last bit of P2 (1) to form the new offspring O1 – 0101011. Similarly the child O2 is formed

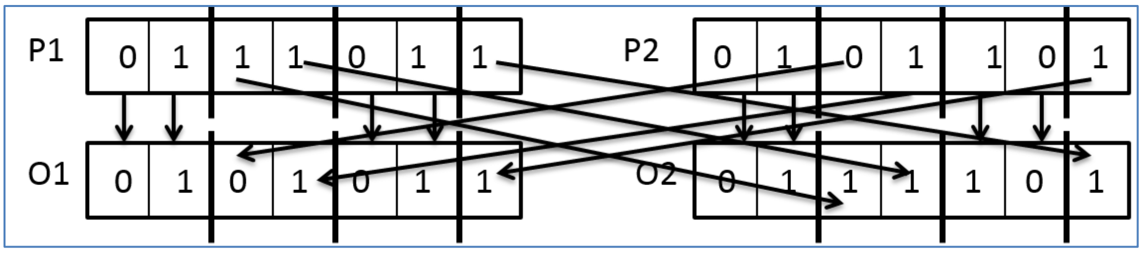


Figure 22.9 Multiple Point Crossover

22.4.3 Parameterized Uniform crossover

In this method the bits of the two parents are exchanged but only with a probability *p* (typically 0.5-0.8). Using this method results in there being no position bias, no endpoint effect etc.

22.4.4 Problems with crossover

Depending on coding, simple crossovers have a high chance to produce illegal offspring. E.g. in TSP with simple binary or path coding, most offspring will be illegal because not all cities will be in the offspring and some cities will be there more than once. Let us consider an example as given in Figure 22.10. Here each gene represents the city number and the fitness function is the sum of cost of

P1	3	6	5	8	4	1	9	7	2
P2	1	7	2	8	6	3	9	5	4
O1	**3**	**6**	**5**	8	**6**	**3**	9	**5**	4
O2	**1**	**7**	**2**	8	4	**1**	9	**7**	**2**

Figure 22.10 (a) TSP problem with One Point Crossover

travelling from one city to another in the given order. TSP tries to find an optimum order that the cost is minimal. However this problem also brings in an additional constraint that each city needs to be included and no city should be visited more than once, or every gene in a chromosome should occur only once. When we use the one point crossover, we can see that genes are repeated as shown in red in Figure 22.10 (a) and hence all genes are not represented in the chromosome.

P1	3	6	5	8	4	1	9	7	2
P2	1	7	2	8	6	3	9	5	4
Mask	0	1	0	1	0	0	1	1	0
O1	1	6	2	8	3	5	9	7	4
O2	1	5	2	8	6	3	9	7	4

Figure 22.10 (b) TSP problem with Uniform Crossover with Masking

In order to cater to the constraints of TSP we use uniform crossover with masking. E.g. in TSP with simple path coding where a mask is used and where if the mask is 1 cities are copied from one parent and whereas when mask is 0, the remaining cities are chosen in the order of the other parent as shown in Figure 22.10 (b)

22.5 Genetic Operator – Mutation

Mutation introduces randomness into the population and is a type of asexual reproduction. The idea of mutation is to reintroduce divergence into a converging population. However mutation is performed only on a small part of population, in order to avoid entering unstable state. In Binary-Coded GAs, each bit in the chromosome is mutated with probability p_{bm} which is known as the mutation rate. In this case for each bit we generate a random number r between 0 and 1 and if $r < m_r$ (i.e., mutation rate) then we flip the bit. For real number representation we randomly create a number for the mutated bit. In the case of tree representation there are two types of mutation namely tree mutation where we randomly create a branch replacing the current branch and point mutation where we change this bit value by random number/ random factor.

22.6 Fitness Function

Fitness function is an evaluation function that determines what solutions are better than others. It is computed for each individual. **Fitness functions** will differ according to the problem and encoding technique. **Fitness function** returns a single numerical value (**fitness**) which reflects the **utility** or the **ability** of the individual which that chromosome represents. Fitness function can calculate strength, weight, width, maximum load, cost, construction time or combination of all these. The fitness value is stored along with chromosome.

The fitness function defines the criterion for ranking potential hypotheses and for probabilistically selecting them for inclusion in the next generation population. If the task is to learn classification rules, then the fitness function typically has a component that scores the classification accuracy of the rule over a set of provided training examples. Other criteria includes the complexity or generality of the rule. When the bit-string hypothesis is interpreted as a complex procedure (e.g., when the bit string represents a collection of if-then rules), the fitness function measures the overall performance rather than performance of individual rules.

22.7 Ending iteration

When does the creation of new generations stop? We do not have an exact answer and we may not know during the design of simulation. Stopping iteration is generally based on three criteria. One criteria is whether the population converges that is the phenotypes of all individuals are the same. The next is evolutionary stability where the phenotype of all individuals does not change through several rounds and finally we use a fixed generation number.

Now the next issue is optimum achievement. We need to know whether we have got the fitness optimum when the iteration stops. We do not know even if the simulation has been completed especially for problems whose results are unknown.

We can try to guarantee the fitness achievement through different initialization design, different parameter settings, by concluding results from more experiments or by using more generations for each experiment.

22.8 Example: The MAXONE Problem

Suppose we want to maximize the number of ones in a string of *l* binary digits - Is it a trivial problem? Actually no though it may seem so because we know the answer in advance. We can map this problem to maximizing the number of correct answers, each encoded by 1, to yes/no difficult questions. For this problem an individual is encoded (naturally) as a string of *l* binary digits. The fitness *f* of a candidate solution is based on the number of ones in its genetic code.

22.8.1 Initialization

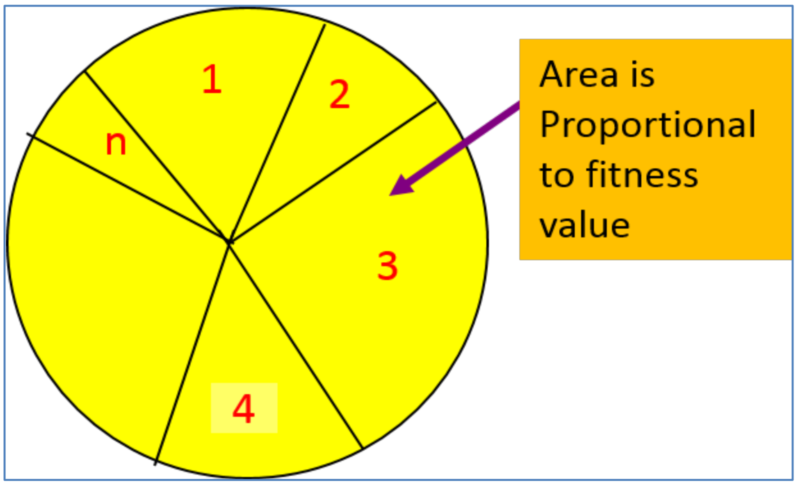
We start with a population of *n* random strings. Suppose that l = 10 and n = 6
Let us assume that we toss a fair coin 6 times and get the following initial population for this example (Figure 22.11):

s_1 = 1111010101	$f(s_1) = 7$
s_2 = 0111000101	$f(s_2) = 5$
s_3 = 1110110101	$f(s_3) = 7$
s_4 = 0100010011	$f(s_4) = 4$
s_5 = 1110111101	$f(s_5) = 8$
s_6 = 0100110000	$f(s_6) = 3$
	Total Fitness value = 34

Figure 22.11 The Initial Population with fitness value for each solution

22.8.2 Selection

Next we apply fitness proportionate selection with the roulette wheel method (Figure 22.12):



Individual i will have a probability to be chosen $\frac{f(i)}{\sum_i f(i)}$

Figure 22.12 Roulette Wheel Selection

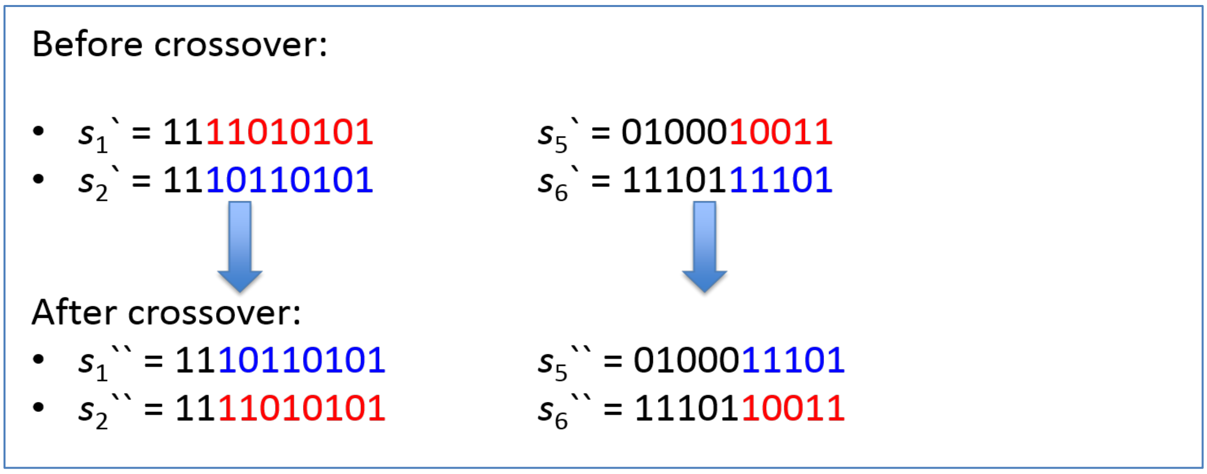
Suppose that, after performing selection, we get the following population (Figure 22.13):

s_1\` = 1111010101	(s_1)
s_2\` = 1110110101	(s_3)
s_3\` = 1110111101	(s_5)
s_4\` = 0111000101	(s_2)
s_5\` = 0100010011	(s_4)
s_6\` = 1110111101	(s_5)

Figure 22.13 Population after Selection

22.8.3 Crossover Operation

Next we mate strings for crossover. For each couple we decide according to crossover probability (for instance 0.6) whether to actually perform crossover or not. Suppose that we decide to actually perform crossover only for couples (s_1\`, $s_2$\`) and (s_5\`, $s_6$\`). For each couple, we randomly extract a crossover point, for instance 2 for the first and 5 for the second (Figure 22.14).


Before crossover:

- $s_1` = 1111010101$ $s_5` = 0100010011$
- $s_2` = 1110110101$ $s_6` = 1110111101$

After crossover:

- $s_1`` = 1110110101$ $s_5`` = 0100011101$
- $s_2`` = 1111010101$ $s_6`` = 1110110011$

Figure 22.14 Crossover Operation for the Example

22.7.4 Mutation Operation

The final step is the random mutation where for each bit that we are to copy to the new population we allow a small probability of error (for instance 0.1) as shown in Figure 22.15. The total fitness value of the final population after one generation is 37.

Before applying Mutation	After applying Mutation	New Fitness value
$s_1``$ = 11101**1**0101	$s_1```$ = 11101**0**0101	$f(s_1```) = 6$
$s_2``$ = 1111**0**1010**1**	$s_2```$ = 1111**1**1010**0**	$f(s_2```) = 7$
$s_3``$ = 11101**1**11**0**1	$s_3```$ = 11101**0**11**1**1	$f(s_3```) = 8$
$s_4``$ = 0111000101	$s_4```$ = 0111000101	$f(s_4```) = 5$
$s_5```$ = 0100011101	$s_5```$ = 0100011101	$f(s_5```) = 5$
$s_6```$ = 11101100**0**1	$s_6```$ = 11101100**0**1	$f(s_6```) = 6$

Figure 22.15 Population after the Mutation Operation

In one generation, the total population fitness changed from 34 to 37, thus improved by ~9%. At this point, we go through the same process all over again, until a stopping criterion is met.

22.9 Benefits of GAs

GAs in general handle huge search spaces by trying to balance exploration and exploitation. The approach is easy to try since little initial knowledge is needed. It is a method that is easy to combine with other methods. GAs make it easier to provide many alternative solutions. They can continually evolve solutions to fit a continually changing problem.

22.10 Genetic Programming

In some programming languages such as LISP, the mathematical notation is written in prefix. Some examples of this:

+ 2 1	:	2 + 1
* + 2 1 2	:	2 * (2+1)
*+ - 2 1 4 9	:	9 * ((2 - 1) + 4)

As we have noted in data structures the prefix order is different to the generally used in order. Moreover the expression has no parenthesis and hence it is easier for programmers and compilers alike, because the order of precedence is not an issue. Therefore expression evaluation is easier. Now what is the connection with this and GAs? If for example you have numerical data and 'answers', but no expression to conjoin the data with the answers. A genetic algorithm can be used to 'evolve' an expression tree (Figure 22.16) to create a very close fit to the data. By 'splicing' and 'grafting' the trees and evaluating the resulting expression with the data and testing it to the answers, the fitness function can return how close the expression is.

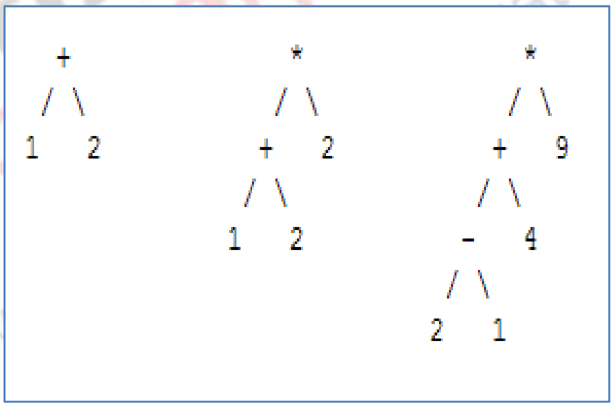


Figure 22.16 Expression Tree

The limitations of genetic programming lie in the huge search space the GAs have to handle, theoretically it can be an infinite number of equations. Normally before running a GA to search for an equation, the user tells the program which operators and numerical ranges to search under. Genetic programming has been used for stock market prediction, advanced mathematics and military applications

Summary

- Discussed the operators of Genetic Algorithms like crossover, mutation and selection
- Discussed the fitness function and the termination condition

- Discussed Genetic Programming