Chapter 1: Introduction to Genetic Algorithms

Genetic algorithms are search algorithms for discrete optimisation problems and are based on principles from evolutionary theory such as natural selection. In every generation, a new set of artificial creatures (strings) is created using parts of the fittest of the previous generations; an occasional new part (mutation) is tried for good measure. GAs are now established as a valid approach to problems requiring efficient and effective search and are finding windspread applications in areas of business, science and engineering. The reason for their increasing popularity is mainly twofold:

1. The algorithms are computationally simple yet powerful;
2. They are not fundamentally limited by restrictive assumptions about the search space such as continuity, differentiability and the existence of a unique local (and hence global) optimum point.

GAs differ from more traditional optimisation and search procedures in several fundamental ways:

1. GAs work with the coding of the decision variables, not the decision variables themselves.
2. GAs search from a population of points, not a single point.
3. GAs use objective function (fitness) information, not derivatives or other auxiliary information.
4. GAs use probabilistic transition rules, not deterministic rules.

GAs require the decision variables of the optimisation problem to be coded as a finite length string over some finite alphabet.

Before introducing some of the concepts associated with Genetic Algorithms (GAs) we look briefly at optimisation in general and introduce some concepts and notation associated with solving optimisation problems.

1.1 What is Optimisation?

Optimisation is the process of attempting to find a *best* or *optimal* solution to the problem under consideration. This problem can arise in many different ways and is usually a mathematical model of a problem which can come from such diverse areas as manufacturing, transportation, construction, telecommunications, financial planning and control engineering, to name a few.

In optimisation we usually solve a deterministic finite dimensional optimisation problem in an attempt to obtain the "best" solution to the problem. Before attempting to solve the optimisation problem we require:

1. A description of the *decision variables.* For example, if there are n quantifiable decisions to be made, then these can be represented by the decision variables $x_1, x_2, \cdots, x_n$.
2. A description of the set $\Omega \in \mathbb{R}^n$, of *admissible (permissible, feasible) decision variables.* Typically, Ω is defined implicitly via functional relations called *constraints.*
3. A measure of performance called an *objective function.* This objective function can be described by a function $f : \Omega \to \mathbb{R}$.

   The problem can then be stated in mathematical terms as

$$\text{minimise} \qquad f(\boldsymbol{x}) \quad \text{over} \quad \boldsymbol{x} \in \Omega.$$

   Note that, if the problem is to maximise some function $p(\boldsymbol{x})$ (such as maximising profit) this is equivalent to minimising $-p(\boldsymbol{x})$.

1.2 Algorithms for Optimisation

Optimisation problems can be separated into two main groups: discrete and continuous.

Discrete Optimisation Problems

Discrete optimisation problems are usually ones in which there are a finite number of choices for the decision variables $x_1, x_2, \cdots, x_n$. Examples are transportation and assignment problems, shortest path and maximum flow problems, other problems on graphs or networks and integer programming problems. Notice that it is possible to approximate all continuous problems by discrete ones. Consider the following example.

Example Suppose we want to solve the following (trivial) optimisation problem:

$$\text{minimise} \quad f(x) = x^2, \qquad -1 \leq x \leq 1.$$

We could encode the decision variable into a binary string of specified precision, for example, use 32 bits together with a sign bit. Each number in $[-1, 1]$ is now represented by a string of binary bits of length 33 and so there are 2^{33} possible choices for the value of the decision variable. The solution to this discrete problem will now give an approximation to the solution to the continuous problem.

Any discrete problem can be solved by simply enumerating all the feasible points, comparing the objective function at each point and choosing the best one. This approach, called *explicit enumeration*, is fine when the number of possibilities is small but in almost all practical problems the number of possibilities is far too large for this

approach to have any chance of success. We need to find algorithms which are more efficient than explicit enumeration for all but the most simple problems. Some discrete problems can be formulated as linear programming problems for which there are efficient solution methods (e.g. simplex method, Karmakar's interior point method). Other examples of efficient algorithms are shortest path algorithms, maximum flow algorithms and critical path planning. However, there are still many practical problems for which efficient algorithms are not available. One of the best known of these problems is the travelling salesman problem which is often used as a test example for algorithms.

When confronted with a problem for which no effective algorithm exists we often give up the search for an optimal solution and use approximate methods that hopefully give an acceptable solution in a reasonable time. The term *heuristic* is used to describe a method which seems to yield a good solution to the problem but cannot be guaranteed to produce an optimal solution. Examples of popular heuristics are simulated annealing and GAs. Both these methods use random choice as a tool in the search process. This has the advantage of helping prevent the algorithm getting stuck in a local minimum.

Nonlinear Programming Problems

Problems in which the variables and constraints can be described by continuous nonlinear functions (although, in practice we usually also assume they are differentiable) can be solved by using methods of calculus. These problems are called *nonlinear programming problems.* The general nonlinear programming problem can be expressed in the form:

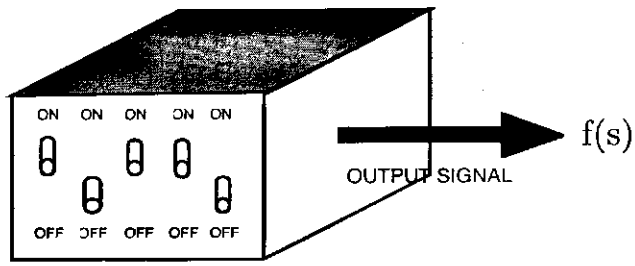
$$\begin{aligned} \text{minimise} \quad & f(\boldsymbol{x}), \qquad \boldsymbol{x} \in \mathbb{R}^n \\ \text{subject to} \quad & h_j(\boldsymbol{x}) = 0, \quad j = 1, 2, \ldots, m \\ & g_j(\boldsymbol{x}) \leq 0, \quad j = 1, 2, \ldots, p \end{aligned}$$

Since no realistic nonlinear programming problem can be solved analytically, the aim is to obtain a numerical algorithm which, when implemented, will provide the solution to the problem. Many of the algorithms are iterative, in that they will (hopefully) converge to the solution (or a good approximation) in a finite, or infinite, number of steps. In all the problems we consider it is usually known mathematically that a solution exists (for example, minimising a continuous function on a compact domain) but it is often quite difficult to prove that the algorithm will provide the solution we are seeking. Another desirable property that the algorithm should have is that it is reasonably efficient. It is no good having an algorithm that is guaranteed to give the correct solution but only after running on the fastest supercomputer longer than the current age of the universe.

Currently the most popular methods for solving general nonlinear programming problems are called sequential quadratic programming (SQP) methods and these are implemented in all major programming libraries (see 'fmincon' in Matlab). These are iterative methods in which a quadratic programming subproblem is solved at each step to obtain a sequence of approximations to the solution of the optimisation problem. There are many different implemetations of the SQP method and this is currently a very active research topic. The main limitation of these methods is that they can only guarantee to locate a local minimum point, whereas in almost all practical problems it is the global minimum point which is required. Also, the methods require continuity of the objective function and constraint functions and possibly continuity of their derivatives. In many practical problems this may not be the case.

1.3 A Simple Genetic Algorithm

Consider the black box switching problem illustrated in the diagram. There are 5 input switches and for each setting of the input switches s there is an output signal $f(s)$. The objective is to set the switches to obtain the maximum value of f. This is obviously a trivial problem but we shall use it to illustrate some of the ideas behind GAs.



First code the switches as a finite length string. an obvious way is to use a 1 to represent when the switch is on and a 0 when it is off. For example, 11110 codes the setting when the first four switches are on and the fifth is off. In our GA we start with a population of strings and then generate successive populations of strings. For this example we will begin with an initial population size of $n = 4$ which is generated at random by 20 coin tosses. Suppose the resulting initial population is:

01101 11000 01000 10011

We now define a set of simple operations that take this initial population and generate successive populations that (hopefully) improve over time.

A simple GA that yields good results in many practical problems is composed of three operations:

1. Reproduction
2. Crossover
3. Mutation

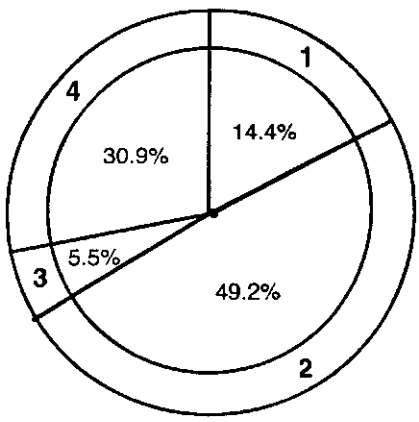
Reproduction is a process in which individual strings are copied according to their objective function values (biologists call this *fitness*). This means that strings with higher fitness have a higher probability of contributing to the next generation. This operator is an artificial version of the Darwinian process of natural selection.

The reproduction operator may be implemented in a variety of ways. In most cases we use the *roulette wheel* approach where each string is assigned a portion of the roulette wheel in proportion to its fitness. We then spin the roulette wheel n times to select n strings which are just copies of their parents. These form the mating pool for the next operation, which is crossover. We use our simple example of the black box to demonstrate this process.

Number	String	Fitness	% of Total	No. selected
1	01101	169	14.4	1
2	11000	576	49.2	2
3	01000	64	5.5	0
4	10011	361	30.9	1
Total		1170	100.0	4

Table 1.1 Sample problem strings and fitness values

Suppose our sample population has the fitness function values shown in the table above. We calculate the fitness proportions for each string which are shown as percentages in the table and assign these proportions to a roulette wheel. To reproduce we then spin the wheel four times.



Suppose this results in one copy of string 1, two copies of string 2, no copies of string 3 and one copy of string 4, as shown in the table. The population of the mating pool is then

01101 11000 11000 10011

After reproduction, crossover may proceed in two ways. Firstly, the newly selected strings are mated at random, that is, pairs of strings are chosen at random. In this example, there will be two pairs chosen for mating. Secondly, each pair of strings undergoes crossover as follows: an integer position k along the string is selected at random where $1 \leq k \leq l-1$, with l being the length of the string. Two new strings are created by swapping all characters between positions $k+1$ and l. For example, suppose the two strings

01101 11000

are selected for mating and the value of k, where $1 \leq k \leq 4$, is chosen at random to be $k = 4$. We then exchange the fifth character between the two strings to obtain the two new strings

01100 11001

This completes the crossover operation.

The mechanics of reproduction and crossover are surprisingly simple, involving random number generation, string copies, and some partial string exchanges. The combined emphasis of reproduction and the randomised information exchange of crossover give GAs much of their power. That these two simple operations can lead to a such a powerful search mechanism is surprising and we will investigate later how this occurs.

The final operation is that of mutation, which plays only a secondary role to reproduction and crossover. In the simple GA that we are considering, mutation is the occasional random alteration of the value of a string position. In our example, this simply means changing a 0 to a 1 and vice versa. Mutation is needed because the operations of reproduction and crossover occasionally may lose some potentially useful information and the mutation operator helps protect against this. The frequency of mutation is usually very small in most GAs, for example, one mutation per thousand bits is a typical value.

1.4 Continuing our Example of a Simple GA

Let us now take our switching box example and use it as the setting for a specific optimisation problem. Consider the problem of maximising $f(x) = x^2$, where x is an integer, $0 \leq x \leq 31$. We can describe the simple GA in six steps.

Step 1: Encoding of the decision variable

In our trivial problem, the coding is obvious. We simply represent x as a binary unsigned integer of length 5. This problem is therefore equivalent to the switching box problem we considered earlier. The results are shown in Table 1.2.

Step 2: Select the initial population

This is as described previously. Let us suppose that the initial population is the same as before.

String No.	Initial Popn.	x	$f(x)$	% of Total	No. Sel.	Mating Pool	Mate	C'over Site	New Popn.	x	$f(x)$
1	01101	13	169	14.4	1	01101	2	4	01100	12	144
2	11000	24	576	49.2	2	11000	1	4	11001	25	625
3	01000	8	64	5.5	0	11000	4	2	11011	27	729
4	10011	19	361	30.9	1	10011	3	2	10000	16	256
Sum			1170	100.0	4						1754
Average			293								439

Table 1.2 Simulation of a simple GA

Step 3: Calculate the fitness proportions for the population

Table 1.2 shows the decoded values of x together with the fitness values $f(x)$. Notice that the values of $f(x)$ are the same as we used for the black box values. The fitness proportions for the strings are calculated and shown as percentages.

Step 4: Reproduction

We select the mating pool for the next generation by spinning the roulette wheel four times. This results in the mating pool shown in Table 1.2.

Step 5: Crossover

This occurs in two steps. First, pairs of strings are randomly chosen for mating. The results of this are shown in the column headed "Mate". Second, the selected site for each crossover is chosen at random. This results in the first mating pair having a crossover site of $k = 4$ and the second mating pair having a crossover site of $k = 2$. The crossover is then performed and the results are shown in the table.

Step 6: Mutation

With 4 strings of 5 bits each we have 20 possible sites for mutation. If we assume that the probability of mutation is 0.001 then we would expect 0.02 bits to mutate. Following our simulation it turns out that no bits mutate. Hence we have the final population for the next generation.

Following reproduction, crossover and mutation we can now assess the fitness of our new population. To do this we decode the new strings and calculate their fitness values. The results are shown in the last two columns of Table 1.2. We see that the population average fitness has improved from 293 to 439 in one generation. This improvement in fitness follows from the fact that the fittest string in the initial population receives two copies and in the resulting reproduction and crossover one of the resulting strings (11011) turns out to be a very good choice. This example is a good illustration of the processes involved in GAs and how they can lead to good results.

1.5 Similarity Templates and Schemas

GAs use only information about a population of strings and their fitness values, so how can this lead to an effective search procedure? To try and understand this, consider the strings and fitness values in Table 1.1. If we look at the four strings we start to see that certain string patterns seem to be associated with higher fitness values, for example, those strings with a '1' in the first position seem to do best. This suggests that there is information in the strings that can be used to advantage in the search procedure. To see how and precisely what information is considered by the simple GA we have described, we introduce the important concept of a *schema* or *similarity template*. (Note that the correct plural of schema is schemata, however we shall use the more commonly used word *schemas* to denote the plural of schema.)

A schema is a similarity template describing a subset of strings with similarities at certain string positions. We again consider only the binary alphabet $\{0, 1\}$ and append to it an extra symbol '$*$' which is a "wild card" symbol. We now create strings using the alphabet $\{0, 1, *\}$ and we think of a schema as a pattern matching device. For example, the schema $*101*$ describes the subset $\{01010, 01011, 10100, 10101\}$. The idea of a schema gives us a compact way to represent the similarities among the various strings.

Consider the previous example in which $l = 5$. Note that there are 3^5 different similarity templates and, in general, for alphabets with k symbols, there are $(k+1)^l$ schemas. Hence, there are more schemas than there are strings in the alphabet so, at first glance, we would seem to be making things more difficult by considering schemas. This is not the case as by considering similarities among the strings we admit a lot of new information to help direct the search. Just how much new information we admit

by considering the similarities among the strings depends on the number of unique schemas in the population.

Consider a single string of length 5, for example 10111. This string is a member of 2^5 schemas since each position may take on its actual value or a '$*$'. In general, a particular string of length l contains 2^l schemas. As a result, a population of size n will contain somewhere between 2^l and $n \cdot 2^l$ schemas, depending on the diversity of the strings in the population. This shows that a great deal of information about similarities is contained in even moderately sized populations.

The question we now ask is: how many of the schemas in a population are usefully processed by the GA? To answer this question we need to consider how the processes of reproduction, crossover and mutation affect the growth or decay of important schemas from generation to generation.

The effect of reproduction simply means that highly fit schemas are likely to be reproduced in increasing numbers. To see the effect of crossover, consider the two schemas $1 * * * 0$ and $* * 11*$. The first schema is much more likely to be disrupted by crossover than the second. Later, we will see that the first schema is an example of one with a relatively long defining length, while the second is an example of a schema with short defining length. Mutation normally occurs at a low rate, so it is unlikely to disrupt a particular schema very frequently. The conclusion of all this is that highly fit schemas with short defining length are propogated generation to generation in increasing numbers. All this takes place in the background, as the algorithm is actually only processing populations of strings. These highly fit schemas with short defining length are called *building blocks*. Later, we will see how the number of schemas which are actually processed usefully in each generation turns out to be proportional to n^3, whereas the number of function evaluations is n. It is this property that gives GAs their processing power.

Problems: Chapter 1

1. Consider a binary string of length 11 and the schema $1 * * * * * * * * * 1$. Under crossover with random crossover site selection, calculate the probability of this schema surviving crossover. Calculate survival probabilities for the following schemas: ****10*****, 11*********, ***111*****, ****1*0****, **1***1**0*.

2. Six strings have the following fitness function values: 5, 10, 15, 25, 50, 100. Under roulette wheel selection, calculate the expected number of copies of each string in the mating pool if a constant population size, $n = 6$, is maintained.

3. Suppose the probability of a mutation at a single bit position is 0.1. Calculate the probability of a 10-bit string surviving mutation without change. Calculate the probability of a 20-bit string surviving mutation without change. Recalculate these probabilities when the mutation probability is 0.01.

4. Consider the strings and schemas of length 11. For the following schemas, calculate the probability of surviving mutation if the probability of mutation is 0.1 at a single bit position: ***1**0****, 1*********0, ***111*****, *1000010*11. Recalculate these probabilities when the mutation probability is 0.01.

5. Use the Matlab random number generator 'rand' to generate 1000 numbers between 0 and 1. Keep track of how many numbers are generated in each of the four quartiles: $0-0.25, 0.25-0.5, 0.5-0.75, 0.75-1.0$, and compare the actual counts with the expected number. Is the difference within reasonable limits? How can you quantify whether the difference is reasonable?

6. Using the Matlab random number generator 'rand', write a function that generates a random integer between some specified lower limit and some specified upper limit. Test the function by generating 1000 numbers between 3 and 12. Keep track of the quantity of each number selected and compare these figures to the expected quantities.

Chapter 2: Computer Implementation of a Simple GA

In this chapter we implement the simple GA described in Chapter 1. We write Matlab code which applies the operations of reproduction, crossover and mutation to two problems:

1. The optimisation of a simple function of a single variable coded as an unsigned binary integer;
2. The optimisation of a function of several (two) variables where the variables are constrained to belong to specified intervals.

We also discuss some implementation issues such as coding of strings, and the mapping of objective functions to fitness functions.

2.1 General Structure of the Simple GA

We will write our program as a function "SimpleGA" which will have the following general structure shown in Figure 2.1. We may make modifications to this structure at a later stage if we deem it advantageous to do so.

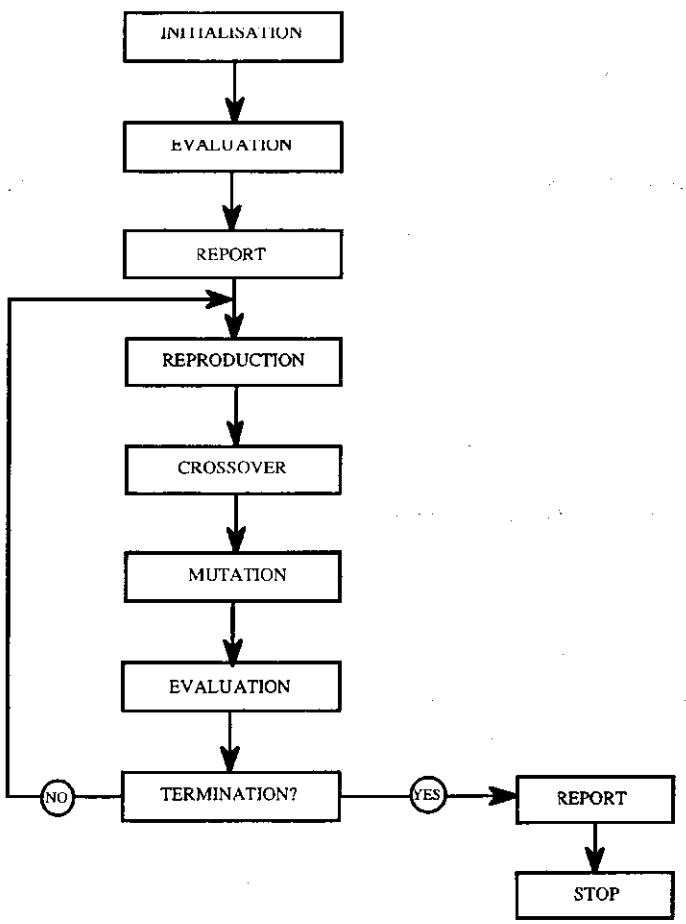


Figure 2.1 General Structure of the Simple GA

We now examine and flesh out each of the modules in our program one-by-one. During this process it will become apparent what inputs and outputs will be required by "SimpleGA" so we will leave this until the end.

Initialisation

In our first simple problem we will assume that we are dealing with a single variable coded as an unsigned binary string. The input required for Initialisation will therefore be the length of the string - call this variable 'lbits'. In the initialisation we produce the initial population, hence we require the number of strings in the population - call this variable 'popnsize'. Even though the initialisation for this problem is trivial we will code it as a function so that it can be generalised in later examples.

$function\ popn = Initialise(popnsize, lbits)$
$\%\ Initialise\ generates\ the\ initial\ population$

$popn = rand(popnsize, lbits) < 0.5;$

$\%\ End\ of\ Initialise$

Evaluation

Here we evaluate the fitness of the individual strings in the population. To do this we first decode the strings into real numbers in function 'Decode' and then call the objective function to evaluate their fitness. We will call the function describing the objective function 'Objfun'. At this time we may want to evaluate other population statistics such as the mean fitness of the population, the maximum fitness and the string and/or real variable corresponding to the maximum fitness for the current generation. These statistics can be saved for use by 'Report'.

$function\ [xpopn, fitness, xopt, maxf, meanf] = Evaluate(popn, lbits)$
$\%\ Evaluate\ returns\ the\ fitness\ of\ the\ current\ population\ after\ first$
$\%\ decoding\ the\ strings,\ together\ with\ other\ population\ statistics$

$xpopn = Decode(popn);$
$fitness = Objfun(xpopn, lbits);$
$meanf = sum(fitness)/size(popn, 1);$
$[maxf, imax] = max(fitness);$
$xopt = xpopn(imax);$

$\%\ End\ of\ Evaluate$

Reproduction

The function 'Reproduce' will use roulette wheel selection to produce the mating pool from the previous population. The numbers of strings selected from the previous population is saved in 'select'. The mating pool is then randomly re-ordered so that the first string is to be mated with the second string, etc.

```
function [matingpairs, select] = Reproduce(popn, fitness)
% Reproduce uses roulette wheel selection to produce the mating pairs
% for crossover.
% select contains the numbers of each string in popn which has been added to
% the mating pool

normfit = fitness/sum(fitness);
partsum = 0;
randnums = rand(size(fitness));
count(1) = 0;
matepool = [ ];

for i = 1 : length(fitness)
    partsum = partsum + normfit(i);
    count(i + 1) = length(find(randnums < partsum));
    select(i, 1) = count(i + 1) − count(i);
    matepool = [matepool; ones(select(i, 1), 1) ∗ popn(i, :)];
end

% Now re − order the strings for mating so that the string in row 1
% is to be mated with the string in row 2, etc.
[junk, mating] = sort(rand(size(matepool, 1), 1));
matingpairs = matepool(mating, :);

% End of Reproduce
```

Crossover

Crossover uses the simple crossover operation described in Chapter 1. Crossover occurs with probability p_c on the pairs of strings chosen for mating by Reproduce.

```
function offspring = Crossover(popn, pc)
% Crossover creates offspring from a population (ordered mating pool)
% using crossover with probability pc.
% Crossover sites are chosen at random, so there are half as many sites
% as there are strings in the population
```

$lbits = size(popn, 2);$
$sites = ceil(rand(size(popn, 1)/2, 1) * (lbits - 1));$
$sites = sites. * (rand(size(sites)) < pc);$

$for\ j = 1 : length(sites);$
$\quad offspring(2 * j - 1, :) = [popn(2 * j - 1, 1 : sites(j))\ popn(2 * j, sites(j) + 1 : lbits)];$
$\quad offspring(2 * j, :) = [popn(2 * j, 1 : sites(j))\ popn(2 * j - 1, sites(j) + 1 : lbits)];$
end

$\%\ End\ of\ Crossover$

Mutation

We use single bit mutation where each bit of each string can mutate with probability p_m.

$function\ newpopn = Mutation(offspring, pm)$
$\%\ Mutation\ changes\ a\ gene\ of\ the\ offspring\ with\ probability\ pm.$

$mutate = find(rand(size(offspring)) < pm);$

$\%\ mutate\ contains\ the\ positions\ of\ the\ genes\ to\ be\ mutated\ as\ a\ column\ vector$
$\%\ going\ down\ the\ columns\ of\ the\ matrix\ offspring$
$newpopn = offspring;$
$newpopn(mutate) = 1 - offspring(mutate);$

$\%\ End\ of\ Mutation$

Report

Report outputs the current population details as well as the details of the fittest string in each generation. Report is called initially and also after the final generation. This routine is rather problem specific and may have to be changed for different problems.

$function\ Report(gen, popn, xpopn, fitness, meanfhistory, maxfhistory, xopthistory)$
$\%\ Report\ outputs\ the\ population\ (strings\ and\ reals)\ and\ the\ fitness\ values\ for\ the$
$\%\ current\ generation,\ together\ with\ a\ history\ of\ mean\ and\ maximum\ fitness\ values$

$disp(' '); disp(' '); disp(' ');$
$disp([sprintf('\qquad\qquad Generation\ \%5.0f', gen)]);$
$disp(' ');$

```
disp(['          string          x          fitness']);

% Compress the matrix popn into a vector of character strings
popnstring = char(48 + popn);

% Output the current population details
for i = 1 : size(popn, 1)
    disp([popnstring(i,:) sprintf('%16.8g %16.8g', xpopn(i), fitness(i))]);
end

disp(' ');
disp('          Fitness History Statistics');
disp('Generation     Mean Fitness     Max Fitness     Optimal x');
history = [[0 : gen]' meanfhistory maxfhistory xopthistory];
disp([sprintf(' %5.0f %16.6g %16.6g %16.6g n ', history')]);

% End of Report
```

Termination

In this specific example we will run the program for a fixed number of generations so there is no need for a separate function to determine when to terminate the program. In general, 'Termination' will include some stopping criteria.

Bits and Pieces

We still have two remaining functions: 'Decode' and 'Objfun'. 'Decode' is called by function Evaluate and we will only give the code for our specific example, in which the single variable is coded as an unsigned binary integer. Later, we will give a more general version of this function.

```
function xpopn = Decode(popn)
% Decode converts a population (popn) of strings from binary to
% integer assuming all variables are nonnegative

lbits = size(popn, 2);
twopowers = 2.^(lbits - 1 : -1 : 0);
xpopn = popn * twopowers';

% End of Decode
```

It remains to define the objective function (fitness) for the problem. In our simple problem we shall consider the function

$$f(x) = \left(\frac{x}{c}\right)^{10}$$

where c is scaling constant so that we don't have to deal with large function values (more about that later). If we take the string length as $lbits = 30$ then, by setting $c = 2^{30} - 1$, the optimal value of f will be $f(x) = 1$ when $x = 2^{30} - 1$.

```
function fitness = Objfun(xpopn, lbits)
% Objfun evaluates the objective function (fitness) values of a decoded
% population
% This example is f(x) = (x/c)^10, where c = 2^lbits - 1

c = 2^lbits - 1;
fitness = (xpopn/c).^10;

% End of Objfun
```

The Main program

It now remains to decide on the inputs and outputs and write the main program. From what has gone before, it is clear that the input variables for our problem are: population size (*popnsize*), string length (*lbits*), p_c and p_m. We shall also include the number of generations (*numgens*) we will be running the algorithm. For output variables, although 'Report' will give us information, we may want to save the decision variables and fitness values of the final population (*xpopn* and *fitness*), the mean and maximum fitness values for each generation and the x values giving the maximum fitness for each generation (*xopt*).

```
function [xpopn, fitness, meanf, maxf, xopt] = onevarSimpleGA(popnsize, lbits, ...
        pc, pm, numgens)
% onevarSimpleGA minimises a simple function of one variable coded as an
% unsigned binary integer

meanfhistory = [ ];
maxfhistory = [ ];
xopthistory = [ ];

% Generate the initial population
gen = 0;
popn = Initialise(popnsize, lbits);

% Obtain fitness statistics for the initial population and save the history
% parameters for use in Report
[xpopn, fitness, xopt, maxf, meanf] = Evaluate(popn, lbits);
xopthistory = [xopthistory; xopt];
```

```
maxfhistory = [maxfhistory; maxf];
meanfhistory = [meanfhistory; meanf];

% Call Report for the printout of the initial population and the initial population
% statistics
Report(gen, popn, xpopn, fitness, meanfhistory, maxfhistory, xopthistory);

% Main generation loop
for gen = 1 : numgens

    % Reproduce
    matingpairs = Reproduce(popn, fitness);

    % Crossover
    offspring = crossover(matingpairs, pc);

    % Mutate
    popn = mutation(offspring, pm);

    % Obtain fitness statistics for the current population and save the history
    % parameters for use in Report
    [xpopn, fitness, xopt, maxf, meanf] = Evaluate(popn, lbits);
    xopthistory = [xopthistory; xopt];
    maxfhistory = [maxfhistory; maxf];
    meanfhistory = [meanfhistory; meanf];

end

% Call Report for the printout of the final population and a summary of the
% population statistics over all generations
gen = numgens;
Report(gen, popn, xpopn, fitness, meanfhistory, maxfhistory, xopthistory);

xopt = xopthistory;
maxf = maxfhistory;
meanf = meanfhistory;

% End of onevarSimpleGA
```

2.2 Running the Test Problem

As mentioned previously, our test problem is

$$\text{Maximise} \quad f(x) = \left(\frac{x}{c}\right)^{10}, \qquad c = 2^{30} - 1$$

where x is coded as an unsigned binary integer. We we will use the following values in our simulation

$$popnsize = 30, \quad lbits = 30, \quad pc = 0.6, \quad pm = 0.0333$$

Note that, with $lbits = 30$, there are $2^{30} = 1.07 \times 10^{10}$ points in the space and so explicit enumeration of all the points is already becoming prohibitive, even for this simple problem.

Generation 0

string	x	fitness
011111101011010101110110001110	5.3145537e+08	0.00088240779
100111111001001000100110000001	6.6928883e+08	0.0088539506
010001010111001010111111001010	2.9128699e+08	2.1587986e-06
100111000100110101101010001110	6.5557979e+08	0.0071987464
010101110010100110001011101001	3.6558513e+08	2.0935653e-05
010001100001100010101100100111	2.9400554e+08	2.3689532e-06
101100001100111110011010001111	7.4159886e+08	0.02469971
101101101100011001000110100100	7.6661188e+08	0.03441571
011111000000011010010010111100	5.202014e+08	0.00071238785
100010010000111011001110101111	5.7486226e+08	0.0019348141
000010111010011100011000010010	48875026	3.8183112e-14
011101101001111110001010010010	4.9754178e+08	0.00045634873
000011111001000110110000111111	65301567	6.9220915e-13
010101100110000010000010010100	3.6229135e+08	1.9124097e-05
101100000100110000001010010111	7.3944335e+08	0.023991112
100111100001010101100100011001	6.6305052e+08	0.0080624589
000001010001011101110011000011	21355715	9.6859555e-18
011010111011010000011100110011	4.5174149e+08	0.00017374052
110010110001110111000001110001	8.5193125e+08	0.098865961
010000010101100110000011000100	2.7409632e+08	1.174988e-06
110101111100101110011000011101	9.0511107e+08	0.18114238
100111100010110111001100000010	6.6345037e+08	0.0081112112
010001110101110001011101000001	2.9930886e+08	2.8326773e-06
100001100111100101101110101000	5.6402628e+08	0.0015995385
111111111000000011111011010000	1.0716608e+09	0.98078668
100010010011101100100100110101	5.7558866e+08	0.0019594023
111000110000111001111111111101	9.5234457e+08	0.30126066
001100111010001100101011010011	2.1658287e+08	1.1149164e-07
100011100000110010010011000101	5.9579719e+08	0.0027668512
100100011111001100100110000111	6.1215783e+08	0.0036277337

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x
0	0.056385	0.980787	1.07166e+09

We will run our simple GA for ten generations and have a look at the results. The initial generation details are shown above. We see that we have been fortunate in choosing an initial population with a string of fitness 0.980787, which is very high. The corresponding x value, when divided by 2^{30} gives a value of 0.99806, which is very close to the optimal value. The average fitness of the population is, however, very low at 0.056385.

Generation 10

string	x	fitness
11111111100010010111011011011	1.0718e+09	0.98206208
11111101100100001111010011000	1.0635342e+09	0.90889986
11011011100011010010011011011	9.2086524e+08	0.2152594
11111111110000111111011001001	1.072758e+09	0.99087476
11111111101110010111010011000	1.0725859e+09	0.98928645
11111111100101000111011001101	1.0719802e+09	0.98371454
11111011100000110111010011000	1.0643616e+09	0.91599558
11111111100010000111011011001	1.0717831e+09	0.98190726
11111011110000111111011001001	1.0559807e+09	0.84637202
11011101100000000111111011011	9.2905443e+08	0.23518676
11100111100000011111010001011	9.710132e+08	0.36580751
11111011100000000101111000010001	1.0548792e+09	0.83758482
11011111100110001111101101001	9.3783625e+08	0.25838747
11111111100111100100101001001	1.072141e+09	0.98519037
11111111001100011111100001000	1.072054e+09	0.98439187
11110111100010000101101101001	1.0382272e+09	0.71437322
11101111100000000100110000010000	1.0045456e+09	0.51368591
11111111000000100101100101010	1.069586e+09	0.96196312
11111111000000000010110110101	1.0695534e+09	0.9616696
11111111101110100101100101001	1.0726007e+09	0.98942275
11110111100010100101101001000	1.0382599e+09	0.71459822
11110110100010010111101100100	1.0340513e+09	0.68615469
11011011000000001111101011010	9.2905439e+08	0.23518667
11111111100010111101111101101	1.0718392e+09	0.98242101
11111111100010101001101101000	1.0718184e+09	0.98223088
11111111100010001001101101000	1.0717857e+09	0.98193063
11111111100010001111101001100	1.0717918e+09	0.98198641
01111111100011010010011011011	5.3498928e+08	0.00094287053
11111111100001001010101001100	1.0717211e+09	0.98133925
11111111100111101101100001000	1.07215e+09	0.98527381

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x
0	0.056385	0.980787	1.07166e+09
1	0.438815	0.99207	1.07289e+09
2	0.635871	0.992018	1.07288e+09
3	0.857955	0.991995	1.07288e+09
4	0.86909	0.991995	1.07288e+09
5	0.836053	0.991636	1.07284e+09
6	0.825284	0.994061	1.0731e+09
7	0.820968	0.994064	1.0731e+09
8	0.811156	0.985029	1.07212e+09
9	0.722945	0.984785	1.0721e+09
10	0.771803	0.990875	1.07276e+09

After ten generations, the Fitness History Statistics show that the algorithm converged very quickly and the optimum fitness value of 0.994064 occurred in generation seven before crossover and/or mutation has destroyed the optimum string. Surprisingly, the population average fitness peaked at the fourth generation.

2.3 Fitness Functions and Fitness Scaling

In GAs we always maximise fitness, so if our optimisation problem is stated as a minimum problem than we know this can be easily converted to a maximum problem by taking the negative of the original objective function. However, GAs also require the fitness function to be nonnegative so we often have to map the objective function into an appropriate fitness function.

For maximisation problems if the objective function $u(x)$ can take on negative values then we let the fitness function be defined by

$$f(x) = \begin{cases} u(x) + C_{min}, & \text{if } u(x) + C_{min} > 0 \\ 0, & \text{otherwise} \end{cases}$$

where C_{min} is prescribed initially or calculated during the algorithm. For example, C_{min} could be the absolute value of the smallest (negative) value of $u(x)$ in the current or previous generation.

For minimisation problems, if $g(x)$ is the objective function to be minimised, we define the fitness function by

$$f(x) = \begin{cases} C_{max} - g(x), & \text{if } C_{max} - g(x) > 0 \\ 0, & \text{otherwise} \end{cases}$$

where C_{max} again could be prescribed initially or calculated during the algorithm. For example, C_{max} could be the largest value of $g(x)$ in the current or previous generation.

The simple GA we have described often exhibits some undesirable properties in the reproduction phase. In the first few generations of the GA it is possible for a few highly fit individuals to dominate the selection process and this can lead to premature convergence. In later generations, the population average fitness may be close to the population maximum fitness and this leads to the emergence of an essentially random search behaviour. In both these cases, scaling of the fitness function can help. Here we describe a process called *linear scaling* which is designed to try and overcome these problems.

Let f denote the fitness function, which is assumed nonnegative, so a mapping of the type descibed above may have already occurred. With linear scaling, the scaled fitness function is then

$$g = af + b$$

where the coefficients a and b are to be appropriately chosen. Now, we want the average fitness of the population to remain the same after scaling so that the average string will contribute one expected offspring to the mating pool. In order to limit the number of expected offspring by the fittest string in the early generations we choose $g_{max} = Cf_{avg}$ where C is usually taken to be 2. In the later generations, when the population average fitness may be close to the population maximum fitness, this mapping will tend to stretch the fitness values. With these conditions, the values of a and b in the linear scaling formula are:

$$a = \frac{f_{avg}(C-1)}{f_{max} - f_{avg}}, \qquad b = f_{avg}(1-a)$$

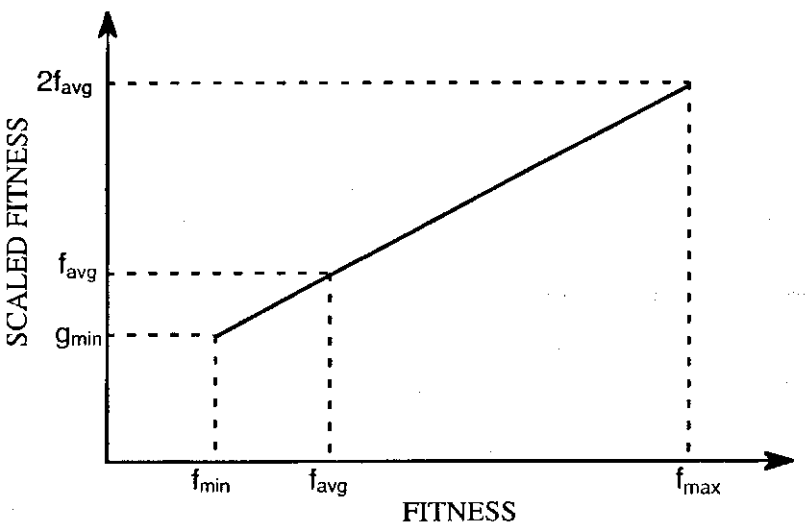


Figure 2.2 Linear scaling under normal conditions

Figures 2.2 and 2.3 show the different situations that can occur with this scaling function using the value $C = 2$. Figure 2.2 shows a typical situation early in the run where the population maximum fitness is well above the average fitness. In this case, fitness scaling does the required job of limiting reproduction of the few highly fit strings in the population. Figure 2.3 shows a situation that can develop in later generations where the average fitness is close to the maximum fitness and there are a few strings with fitness well below the average. In this case, the scaled fitness function can produce negative values which is unacceptable. There are various ways to try and combat this problem. One possible solution when negative values occur is to still map the average fitness to the average of the scaled fitness but, instead of mapping the maximum fitness to C times the maximum, map the minimum fitness to zero. The parameters a and b in the linear scaling formula now become:

$$a = \frac{f_{avg}}{f_{avg} - f_{min}}, \qquad b = f_{avg}(1-a)$$

This modified scaled fitness function is shown in Figure 2.3.

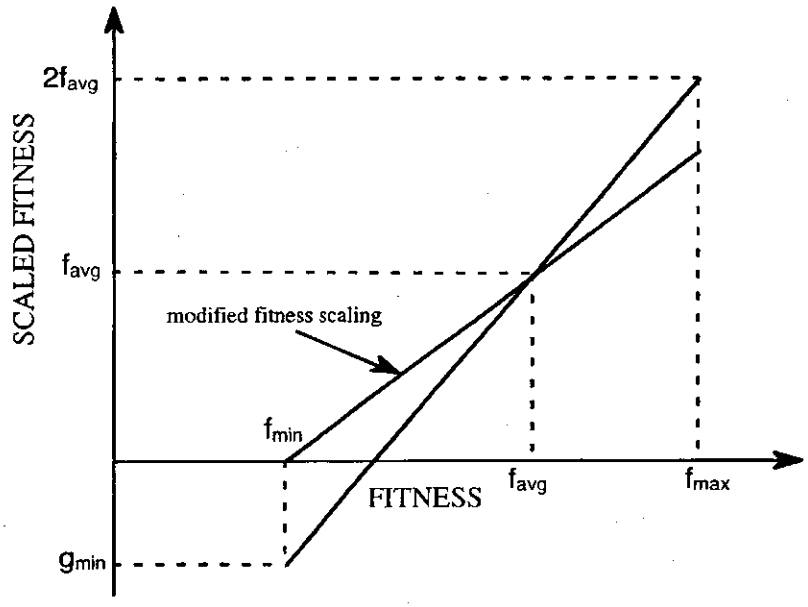


Figure 2.3 Linear scaling with negative scaling values and the modified function

A coding of this scaling function, called 'Scalefitness' is given below. This routine will be called from 'Reproduce' in our simple GA, as the scaled fitness values are used only for the roulette wheel selection.

```
function scfitness = Scalefitness(fitness, fmultiple)
% Scalefitness performs linear scaling on the fitness values and returns the
% results in scfitness

% Calculate the parameters a and b
favg = sum(fitness)/length(fitness);
[fmax, i] = max(fitness);
[fmin, j] = min(fitness);
a = favg * (fmultiple - 1)/(fmax - favg);
b = favg * (1 - a);
if a * fmin + b < 0
    a = favg/(favg - fmin);
    b = favg * (1 - a);
end

% The scaled fitness
scfitness = a * fitness + b * ones(size(fitness));

% End of Scalefitness
```

2.4 Codings and Multivariable Problems

In the simple example studied in this chapter we used a binary coding where a single real variable x was coded as an unsigned binary integer. This coding can easily be extended to the more general case when $a \leq x \leq b$, for given constants a and b. In fact, we will see shortly how to code real multi-dimensional decision variables into binary strings.

The question we first want to address is: What is the best coding? For some problems there may be a natural coding, but for many problems there are many possible codings. Remember, for the GA to work successfully, we require building blocks (short, highly fit schemas) to lead to optima. In the early days of GAs, there were two principles which governed the choice of a coding for a particular problem.

1. The user should select a coding so that short, low-order schemas are relevant to the underlying problem.
2. The user should select the smallest alphabet that permits the natural expression of the problem.

The first principle is difficult to satisfy in practice, but it essentially means that bit positions that are related should be close together. We will look later at ways of rearranging the ordering of a string coding. The second principle leads naturally to binary codings in many problems and explains our preoccupation with binary numbers. Further justification for using binary codings can be seen by comparing the number of schemas for a binary coding (3^l) with the number for a nonbinary coding ($(k+1)^l$). It is easy to show that the binary alphabet offers the maximum number of schemas per bit of information of any coding.

Now let us consider the binary coding for a multivariable problem with decision variable $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$ where $a_i \leq x_i \leq b_i$. Each component of $\boldsymbol{x}$ will be encoded as a binary string of length l_i. The length of each string is determined by the required precision. For example, if we require five decimal place precision for the variable x_i then l_i is chosen to be the smallest integer satisfying:

$$10^5(b_i - a_i) \leq 2^{l_i} - 1.$$

With this choice of l_i, the decoding from a binary string to a real number is given by:

$$x_i = a_i + \text{decimal}(\text{string}_i)\left(\frac{b_i - a_i}{2^{l_i} - 1}\right)$$

where, decimal(string$_i$) means the decimal integer value of the ith binary string.

For the simple GA that we have programmed there are a few changes that have to be made to accommodate multivariable optimisation problems. Firstly, the input parameter *lbits* now has to be a vector with components l_i, $i = 1, \ldots, m$. Secondly,

the function 'Decode' has to be generalised to implement the above formula. There will also have to be additional inputs into our main function 'SimpleGA', which are the lower and upper bounds on the variables. The new form of 'SimpleGA' will now be

$$function\ [xpopn, fitness, meanf, maxf, xopt] = multivarSimpleGA(popnsize, ... \\ lbits, pc, pm, numgens, vlb, vub)$$

where *lbits* is a row vector of length m and *vlb* and *vub* are row vectors of length m containing the lower and upper bounds of the real variables x_i, $i = 1, 2, \ldots, m$. A coding for the general version of function 'Decode' is shown below.

```
function xpopn = Decode(popn, lbits, vlb, vub)
% Decode converts a population (popn) of strings from binary to real.
% Each string in popn is of length sum(lbits) and consists of m = length(lbits)
% substrings which decode to the variables x_1, ..., x_m

% First decode each substring to an unsigned decimal integer : xint
index1 = 1;
index2 = 0;
for i = 1 : length(lbits)
    index2 = index2 + lbits(i);
    twopowers = 2.^(lbits(i) - 1 : -1 : 0);
    xint(:, i) = popn(:, index1 : index2) * twopowers';
    index1 = index1 + lbits(i);
end

% Now calculate the x values
factor = (vub - vlb)./(2.^lbits - 1);
xpopn = ones(size(popn, 1), 1) * vlb + xint * diag(factor);

% End of Decode
```

Other minor changes will have to be made to other parts of our program, for example, in 'Report', where the output format of the variables x will have to be changed. Note that fitness scaling is not included in this program.

Example Consider the following problem:

$$\begin{aligned} \text{Maximise} \quad & f(x_1, x_2) = 21.5 + x_1 \sin(2\pi x_1) + x_2 \sin(10\pi x_2) \\ \text{subject to} \quad & -3.0 \leq x_1 \leq 12.1 \\ & 4.1 \leq x_2 \leq 5.8 \end{aligned}$$

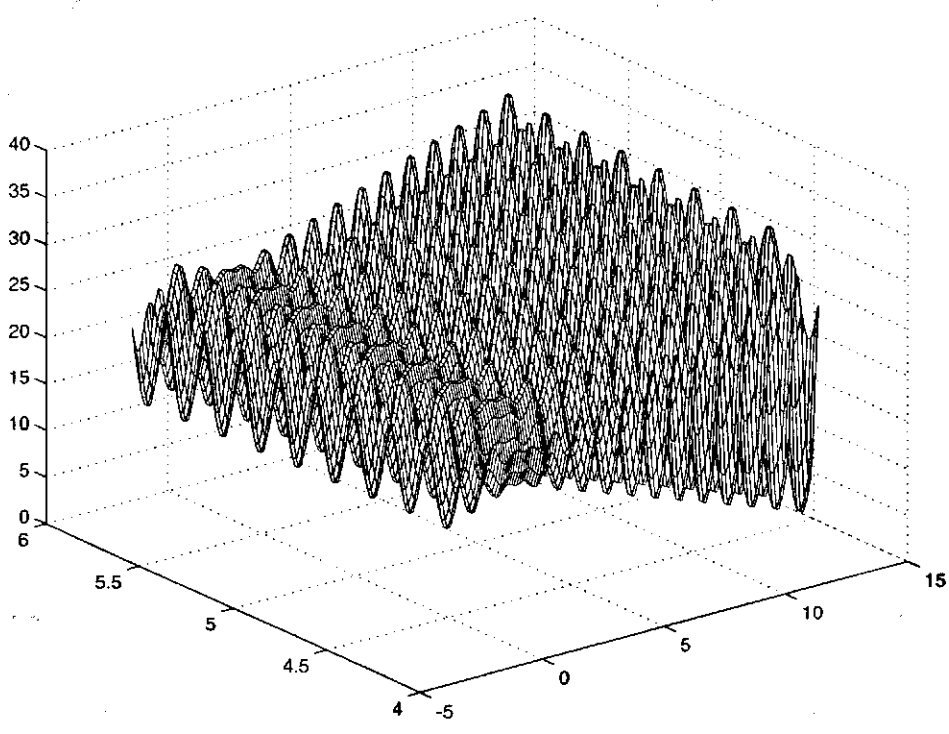


Figure 2.4 Mesh plot of the function $f(x_1, x_2)$

Figure 2.4 shows a surface plot of the function and we can see that the function has many local peaks of similar height making this a difficult problem for most optimisation techniques to solve.

Let us code the decision variables into binary strings with four decimal place precision. This gives the sub-string lengths as $l_1 = 18$ and $l_2 = 15$ and so the total string length is 33 bits. We use a population size of 10 and values of $p_c = 0.5$ and $p_m = 0.01$. We will initially run the program for 10 generations but later increase this number. The details of this run are shown on the next page.

From the output we can see that the mean population fitness actually dropped in the first few generations before increasing, while the maximum fitness seems to get stuck after seven generations. If we run the program for 100 generations then the results improve dramatically. The mean fitness peaks at 35.2285 in generation 65 while the optimum fitness value occurs in generation 82 and has a value of 37.7597 with $\boldsymbol{x} = (11.2008, 5.6455)$. You should try different runs of this program, varying the values of *numgens*, p_c and p_m. Remember to reset the random number generator before each run so as to be able to compare the results.

Generation 0

string	fitness	x	
00110110010111010111011111001110	23.2517	0.206654	5.6912
10101001011110111111111100110000001	18.9683	6.99691	5.71377
00010100000001011101111101010110 0	15.6529	-1.81898	4.93239
10101000011010011101100000011101 1	14.163	6.93378	4.74058
01110101111100110110110100101010 0	20.4985	3.95725	5.29971
01110000001001111011000100000000 1	19.3833	3.61538	5.40165
10100110110000001000101110010000 0	16.6137	6.8358	4.40714
10001011110101010011101111110001 1	28.1121	5.24794	5.69229
01111010010011110101101000000000 0	24.2217	4.21438	4.79065
10101000000110000000011100101100 1	19.2316	6.91494	4.29056

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x	
0	20.0097	28.1121	5.24794	5.69229

Generation 10

string	fitness	x	
01101110110001011011101110000111 0	23.9122	3.53381	5.68124
00110000110000000100100100001010 0	17.4555	-0.124443	4.34011
00110000000000000001101111100010 0	26.281	-0.168739	4.84066
10110000110000000100100100001010 0	20.7146	7.42559	4.34011
10110000000000000001101100001010 0	29.1223	7.38129	4.81825
00110000110000000110100111100111 0	23.6678	-0.124443	5.21306
01110010000001110111110100100111 0	13.8448	3.72591	5.72441
01101110110001011011101110000111 0	23.9122	3.53381	5.68124
10110000000000000001101111100111 0	31.1655	7.38129	4.84118
10110000000000000001101101100111 0	30.2191	7.38129	4.8279

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x	
0	20.0097	28.1121	5.24794	5.69229
1	21.1405	28.1391	5.24794	5.69214
2	18.3244	23.6502	3.85131	5.63808
3	18.6192	22.1903	3.608	5.68259
4	19.675	25.5737	11.5073	5.26619
5	20.6879	27.2951	3.06067	4.84118
6	20.5021	27.2951	3.06067	4.84118
7	21.9399	31.1655	7.38129	4.84118
8	21.0691	31.1655	7.38129	4.84118
9	22.5961	31.1655	7.38129	4.84118
10	24.0295	31.1655	7.38129	4.84118

Problems: Chapter 2

1. A search space contains $2,097,152$ points. A binary-coded genetic algorithm is compared to an octal-coded genetic algorithm. Calculate and compare the following quantities in the two cases:
   (a) Total number of schemas.
   (b) Total number of search points.
   (c) Number of schemas contained within a single individual.
   (d) Upper and lower bounds on the number of schemas in a population of size $n = 50$.

2. Create a Matlab function 'mutation' that takes a population of strings and a specified mutation probability p_m and returns the mutated population, together with the indices of the mutated bits. The function should complement a particular bit value with mutation probability p_m. Test the function by performing 1000 calls to mutation using the the following strings of length 10: $1011101011, 0000110100$ and mutation probabilities of $p_m = 0.01$ and $p_m = 0.001$. Compare the realised number of mutations with the expected number.

3. A function $f(x, y, z)$ is to be minimised, where $-20 \leq x \leq 125$, $0 \leq y \leq 1.2 \times 10^6$ and $-0.1 \leq z \leq 1$. The desired precision for x, y and z are 0.5, 10^4 and 0.001, respectively. Design a concatenated, binary coding for this problem. What is the minimum number of bits required to obtain the desired precision? With the selected coding, determine the bit strings that represent each of the following points: $(-20, 0, -0.1), (125, 1.2 \times 10^6, 1), (50, 10^5, 0.597)$. (You should be able to check your answer for the third point by using 'function Decode'.)

Chapter 3: Mathematical Foundations of Genetic Algorithms

In Chapter 1 we introduced the notions of similarity templates, or schemas, and building blocks, which were short, highly fit schemas. We now present a more rigorous analysis of GAs and see how they exploit in parallel the many similarities contained in these building blocks. This analysis leads to the fundamental theorem of GAs.

3.1 Some Notation

To enable us to analyse the growth and decay of the many schema contained in a population, we need to introduce some simple notation.

We consider, without loss of generality, strings to be constructed over the binary alphabet $V = \{0, 1\}$. We will refer to strings by capital letters and individual characters by subscripted lower-case letters. For example, the string 1 0 1 1 0 0 1 may be represented symbolically as

$$A = a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7$$

The strings in GAs are analogous to *chromosomes* in biological systems, while the a_i's are called *genes* and their values (0 or 1), *alleles*.

We use $t = 0, 1, \ldots$ (time) to denote the generation number and $\boldsymbol{A}(t)$ will then denote a population of strings at time t. If this population contains n strings: $A_1, A_2, \ldots, A_n$, then

$$\boldsymbol{A}(t) = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_n \end{bmatrix}.$$

As before, the '$*$' is used to denote the wild-card symbol which matches either a 0 or a 1 at a particular position. Recall that there are 3^l schemas defined over a string of length l and, more generally, for alphabets with k characters there are $(k+1)^l$ schemas. Also there are at most $n \cdot 2^l$ schemas contained in a population of size n because each string itself is a representative of 2^l schemas.

Suppose H is a schema taken from the three-letter alphabet $V+ = \{0, 1, *\}$. The *order* of H, denoted by $o(H)$, is the number of fixed positions in the template. For example, the order of the schema 0 1 1 $*$ 1 1 $*$ $*$ is 5 and the order of the schema 0 $*$ $*$ $*$ $*$ $*$ $*$ $*$ is 1. The *defining length* of H, denoted by $\delta(H)$, is the distance between the first and last string position. For example, the defining length of the schema 0 1 1 $*$ 1 1 $*$ $*$ is $\delta(H) = 6 - 1 = 5$ and the defining length of the schema 0 $*$ $*$ $*$ $*$ $*$ $*$ $*$ is $\delta(H) = 0$, since there is only a single fixed position.

Finally, let $m(H, t)$ represent the number of examples of a particular schema H contained within a population $\boldsymbol{A}(t)$ at time t.

3.2 The Fundamental Theorem

We now consider the individual and combined effect of reproduction, crossover and mutation on schemas contained within a population of strings.

Reproduction

If $A(t)$ contains the strings A_i, $i = 1, 2, \ldots, n$, with fitness f_i, then string A_i is selected with probability

$$\frac{f_i}{\sum_{j=1}^{n} f_j}.$$

The expected number of examples of a schema H at time $t+1$ is then

$$m(H, t+1) = m(H, t)\frac{n \cdot f(H)}{\sum_{j=1}^{n} f_j} = m(H, t)\frac{f(H)}{\bar{f}} \tag{3.1}$$

where $f(H)$ is the average fitness of the strings representing H at time t and $\bar{f}$ is the average fitness of the population at time t.

Consider the difference equation

$$x(t+1) = cx(t), \qquad t = 1, 2, \ldots$$

where c is a constant. The solution is

$$x(t) = x(0)c^t$$

and so, $x(t)$ grows exponentially for $c > 1$ and decreases exponentially for $c < 1$. Hence, from Eqn. (3.1) we see that the effect of reproduction is to allocate exponentially increasing numbers in future generations to schemas with above average fitness while allocating, in parallel, exponentially decreasing numbers to schemas with below average fitness.

Crossover

Suppose the string $A = 0\ \ 1\ \ 1\ \ 1\ \ 0\ \ 0\ \ 0$ has been selected for mating and crossover. Two representative schema are

$$H_1 = *\ \ 1\ \ *\ \ *\ \ *\ \ *\ \ 0 \quad \text{and} \quad H_2 = *\ \ *\ \ *\ \ 1\ \ 0\ \ *\ \ *$$

Clearly, H_2 is more likely to survive crossover than H_1. Let us estimate the probability of survival of the two schemas.

For H_1, an upper bound on the probability of destruction is $\frac{\delta(H_1)}{6}$, that is

$$\text{probability that } H_1 \text{ is destroyed by crossover} \leq \frac{\delta(H_1)}{6} = \frac{5}{6}.$$

The reason this is an upper bound is because we have ignored the possibility that A's mate also has a 0 in position 7. Hence, if p_s is the probability of survival of H_1, then

$$p_s \geq 1 - \frac{\delta(H_1)}{6} = 1 - \frac{5}{6} = \frac{1}{6}.$$

Similarly, for H_2 we have

$$p_s \geq 1 - \frac{\delta(H_2)}{6} = 1 - \frac{1}{6} = \frac{5}{6}.$$

In general, if l is the length of the strings, a lower bound on the probability of survival of a schema H under crossover is

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

If we introduce the possibility of crossover occurring with a probability p_c (in our earlier example, we took $p_c = 1$), then we have

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

The effects of reproduction and crossover can now be combined in Eqn. (3.1) to give

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\bar{f}} \left(1 - p_c \frac{\delta(H)}{l-1}\right) \tag{3.2}$$

Mutation

Suppose p_m is the probability that a single bit mutates. Then, the probability of a single bit surviving mutation is $1 - p_m$. For a given schema H, the probability of survival will depend on the number of fixed positions in the schema, that is, on $o(H)$. Hence, the probability of the schema surviving mutation is $(1-p_m)^{o(H)}$. Since p_m is always small, we often approximate this by $1 - o(H)p_m$. We can now combine this with the effects of reproduction and crossover to give the final equation, known as the *Schema Theorem* or the *Fundamental Theorem of Genetic Algorithms*,

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\bar{f}} \left(1 - p_c \frac{\delta(H)}{l-1}\right) [1 - p_m]^{o(H)}. \tag{3.3}$$

This theorem describes the growth of a schema from one generation to the next and essentially says that short, low order schemas whose average fitness stays above the mean will receive exponentially increasing numbers of samples in subsequent generations.

3.3 The Two-armed and k-armed Bandit Problem

We have seen that schemas of short defining length, low order and above average fitness receive exponentially increasing numbers of 'trials' (instances in the population) in future generations. We now examine why this is a good strategy by considering an important problem in statistical decision theory, the two-armed bandit problem and its extension, the k-armed bandit problem.

Suppose a gambler is given N coins with which to play a slot machine with two arms. One arm has a mean payoff rate of μ_1 with variance σ_1^2 and the other a mean payoff rate of μ_2 with variance σ_2^2 where $\mu_1 \geq \mu_2$. The gambler does not know these payoff rates or variances and the aim is to maximise the total payoff after the N trials. We now sketch the solution to this problem.

Suppose, initially we allocate n trials to each arm. We then observe the results and choose the arm with the best results to allocate the remaining $N - 2n$ trials. The question now is: What is the optimal value of n? There is always the possibility that after the n trials with each arm that the observed best arm is, in fact, the worst arm (that is, we guessed wrong). Let this probability be $q(n)$. Assuming we know N, μ_1, μ_2, σ_1 and σ_2, the expected loss after N trials (as compared with allocating all N trials to the best machine) is given by:

$$L(N, n) = q(n)(N - n)(\mu_1 - \mu_2) + (1 - q(n))n(\mu_1 - \mu_2).$$

We can find the value of n which minimises L by setting the derivative of L to zero and solving for n:

$$\frac{dL}{dn} = (\mu_1 - \mu_2)\left(1 - 2q(n) + (N - 2n)\frac{dq}{dn}\right) = 0.$$

To solve this equation we need to determine $q(n)$. This is quite difficult and so we will only give the final result.

The optimal allocation of trials n^*, which is the number of trials given to the worst observed arm, is approximated by

$$n^* \approx c \ln\left(\frac{N^2}{8\pi c^2 \ln(N^2)}\right),$$

where $c = \sqrt{\sigma_1/(\mu_1 - \mu_2)}$. If we solve for N from this equation, we obtain

$$N \approx e^{n^*/2c}\sqrt{8\pi c^2 \ln(N^2)}$$

which gives the number of trials given to the best observed arm as

$$N - n^* \approx e^{n^*/2c}\sqrt{8\pi c^2 \ln(N^2)} - n^*.$$

As n^* increases, the exponential dominates, and so,

$$N - n* \approx e^{n^*/2c}\sqrt{8\pi c^2 \ln(N^2)}.$$

The interpretation of this result is that we should allocate exponentially increasing trials to the observed best arm.

With a GA we are no longer solving a two-armed bandit problem. In the simple GA we consider the simultaneous solution of many multi-armed bandit problems. When the loss function for a k-armed bandit problem is minimised, the solution is similar to the 2-armed problem in that it implies that exponentially increasing numbers of trials be allocated to the observed best of the k arms. To see how the k-armed bandit problem is related to the processing of schemas in the GA we consider a specific example of a set of competing schemas.

Consider the following set of eight competing schemas that compete over the positions 2, 3 and 5:

$$\begin{array}{ccccccc}
* & 0 & 0 & * & 0 & * & * \\
* & 0 & 0 & * & 1 & * & * \\
* & 0 & 1 & * & 0 & * & * \\
* & 0 & 1 & * & 1 & * & * \\
* & 1 & 0 & * & 0 & * & * \\
* & 1 & 0 & * & 1 & * & * \\
* & 1 & 1 & * & 0 & * & * \\
* & 1 & 1 & * & 1 & * & *
\end{array}$$

These $2^3 = 8$ schemas compete with one another for population spots. In order to allocate population spots properly we need to allocate exponentially increasing trials to the observed best schemas just as in the k-armed bandit problem. We can think of this example as being the equivalent of a 8-armed bandit problem. One of the differences between our GA and a k-armed bandit problem is that, in the GA, there are a number of problems proceeding in parallel. For example, in schemas of length 7 and order 3, there are $\binom{7}{3} = 35$ different eight armed-bandit problems. In general, for schemas of order j over strings of length l, there are $\binom{l}{j}$ different 2^j-armed bandit problems.

3.4 Implicit Parallelism and Building Blocks

Each of the n individuals in a population of strings of length l contains 2^l schemas and so GAs essentially process between 2^l and $n \cdot 2^l$ schemas in parallel in each generation. This is called *implicit parallelism*. However, many of these schemas with long defining lengths will be destroyed by crossover and only those short, low-order and highly fit schema are sampled at the desirable exponentially increasing rate. These particular schemas, called *building blocks*, are sampled, recombined and resampled to form strings of potentially higher fitness.

There have been estimates made of the number of schemas processed in a useful manner (building blocks) by a GA. Despite the processing of only n strings in each generation, a GA usefully processes something like n^3 schemas. Hence, even though we perform computations proportional to the size of the population at each generation, we get useful processing of something like n^3 schemas in parallel with no extra work. This is where the power of GAs comes from.

The notion that GAs achieve optimality by the recombining of building blocks to form better and better strings is called the *building block hypothesis* and there have been many attempts in the research literature to justify this claim. There is now a great deal of evidence to suggest this is true for a variety of problem classes and many practical problems have now been solved using essentially the simple GA described in Chapter 1.

There are, however, problems which the simple GA finds difficult to solve. For example, functions and codings which contain isolated optima in which the best points tend to be surrounded by the worst. These functions and codings are called *GA-deceptive*. Note that one particular coding of a problem may be GA-deceptive whereas another coding may not.

3.5 The Minimal Deceptive Problem

Although many practical problems are not GA-deceptive it is important to understand better what makes a problem difficult for a simple GA. To this end, we try and construct a simple problem which violates the building block hypothesis. That is, we want short, low-order building blocks to lead to incorrect longer, high-order schemas. It can be shown that no order-1 problem can be GA-deceptive and that the smallest problem that is GA-deceptive is an order-2 problem. We call this the *minimal deceptive problem* (MDP).

Suppose we have a set of four order-2 schemas with fitness values as follows:

$$
\begin{array}{cccccccccccc}
* & * & * & 0 & * & * & * & * & * & 0 & * & f_{00} \\
* & * & * & 0 & * & * & * & * & * & 1 & * & f_{01} \\
* & * & * & 1 & * & * & * & * & * & 0 & * & f_{10} \\
* & * & * & 1 & * & * & * & * & * & 1 & * & f_{11} \\
 & & & | & \longleftarrow & & \delta(H) & & \longrightarrow & | & &
\end{array}
$$

where the fitness values are schema averages, assumed to be constant. To construct a problem which will deceive our GA into giving the wrong answer, we will assume that f_{11} is the global maximum and that the fitness f_{0*} of the order-1 schema $0*$ is greater than the fitness f_{1*} of the order-1 schema $1*$, that is,

$$\frac{f_{00} + f_{01}}{2} > \frac{f_{10} + f_{11}}{2}. \tag{3.4}$$

The aim is to try and fool the GA algorithm into converging to a non-optimal solution.

From Eqn. (3.4) we can deduce the following inequalities:

$$f_{10} < f_{01} \qquad \text{and} \qquad f_{10} < f_{00}.$$

This leaves two possibilities as to the relationship between f_{01} and f_{00}. These are:

Type 1: $f_{01} > f_{00}$.

Type 2: $f_{01} \leq f_{00}$.

We will now construct difference equation relationships for the expected proportions of each of the four schemas in successive generations, making the assumption that $p_m = 0$. We will not use Eqn. (3.3) as this only gives a lower bound, assuming that crossover destroys a schema whenever the crossing site is between the outermost bits of the schema. In this simple case we do not have to make this assumption and can calculate the exact probabilities.

In calculating the expected proportions of the schemas in successive generations we will use the following table which shows what happens when crossover occurs between each pair of schemas:

	0 0	0 1	1 0	1 1
0 0	S	S	S	0 1 1 0
0 1	S	S	0 0 1 1	S
1 0	S	0 0 1 1	S	S
1 1	0 1 1 0	S	S	S

The S in the above table denotes that the offspring are the same as the parents.

Now let P_{00}^t, P_{01}^t, P_{10}^t, P_{11}^t, denote the proportions of the four schemas in generation t. The probability of a crossing site falling between the two defining bits of each schema is

$$p_c \frac{\delta(H)}{l-1}.$$

We will assume that the probability of crossover is $p_c = 1$ and so, for this example,

$$p_c \frac{\delta(H)}{l-1} = 0.6.$$

Using the table, the proportions of the four schemas in generation $t+1$ are given by:

$$\begin{aligned}
P_{00}^{t+1} &= \frac{f_{00}}{\bar{f}} P_{00}^t \left(1 - 0.6\frac{f_{11}}{\bar{f}} P_{11}^t\right) + 0.6\frac{f_{01}f_{10}}{\bar{f}^2} P_{01}^t P_{10}^t \\
P_{01}^{t+1} &= \frac{f_{01}}{\bar{f}} P_{01}^t \left(1 - 0.6\frac{f_{10}}{\bar{f}} P_{10}^t\right) + 0.6\frac{f_{00}f_{11}}{\bar{f}^2} P_{00}^t P_{11}^t \\
P_{10}^{t+1} &= \frac{f_{10}}{\bar{f}} P_{10}^t \left(1 - 0.6\frac{f_{01}}{\bar{f}} P_{01}^t\right) + 0.6\frac{f_{00}f_{11}}{\bar{f}^2} P_{00}^t P_{11}^t \\
P_{11}^{t+1} &= \frac{f_{11}}{\bar{f}} P_{11}^t \left(1 - 0.6\frac{f_{00}}{\bar{f}} P_{00}^t\right) + 0.6\frac{f_{01}f_{10}}{\bar{f}^2} P_{01}^t P_{10}^t
\end{aligned} \tag{3.5}$$

where $\bar{f}$ is the average population fitness in generation t and is given by

$$\bar{f} = f_{00}P_{00}^t + f_{01}P_{01}^t + f_{10}P_{10}^t + f_{11}P_{11}^t.$$

We now show the results of several simulations of Eqns. (3.5). In each simulation, the individual schema fitnesses are all normalised with respect to f_{00}, for example, when we specify $f_{11} = 1.1$, this represents $f_{11} = 1.1f_{00}$. These results are shown in Figs. 3.1 and 3.2 for an example of Type 1. Fig 3.1 shows that, with equal initial proportions, $P_{11} \to 1$ as $t \to \infty$ showing that this case is not GA-deceptive although it appears at first that 0 1 does better only to lose out in the end to 1 1. Fig 3.2 shows the same case with unequal initial proportions (0 0 has been given 70% of the initial proportion). If we were to run the simulation for a limited time, it would appear that 0 1 is winning. However, 1 1 wins in the end, again showing that this case is not GA-deceptive. These results are indicative of the general situation as it has been proved in the literature that any Type 1 problem with nonzero initial proportions is not GA-deceptive.

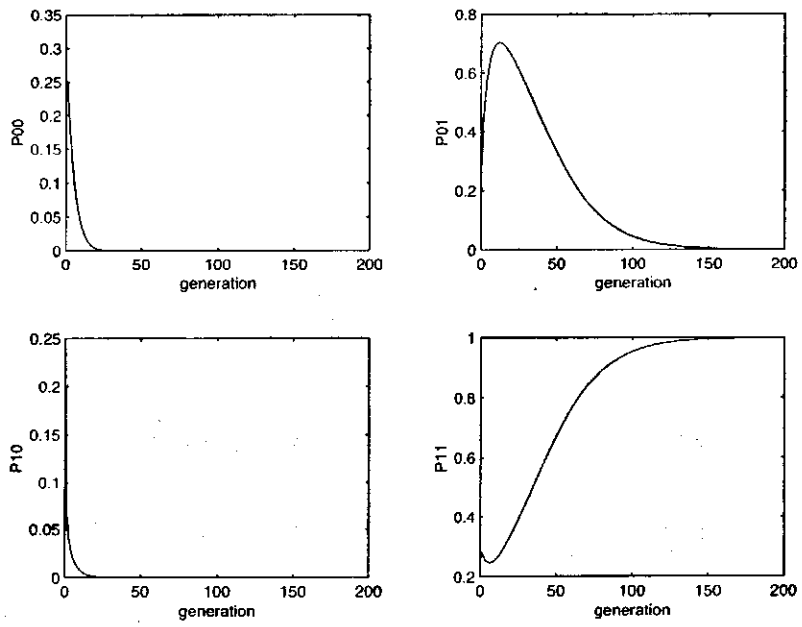


Figure 3.1 Type 1: $f_{01} = 1.05$, $f_{10} = 0$, $f_1 = 1.1$ and equal initial proportions.

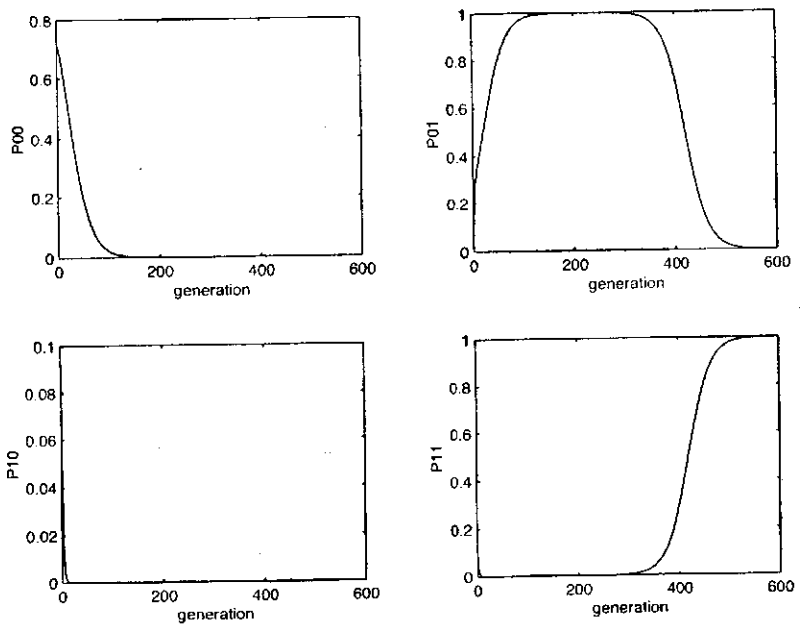


Figure 3.2 Type 1: $f_{01} = 1.05$, $f_{10} = 0$, $f_1 = 1.1$ and unequal initial proportions.

Fig. 3.3 shows a Type 2 simulation with equal initial proportions. Again, $P_{11} \rightarrow 1$ as $t \rightarrow \infty$ showing that the simple GA gives the correct solution to this problem. Fig. 3.4 shows the same simulation with unequal initial proportions. This time 0 0 becomes the dominant schema and the algorithm converges to this in the limit as $t \rightarrow \infty$ and so this case is GA-deceptive with the algorithm converging to the second best choice. Note that other simulations were continued well past 100 generations in this case just to verify that 0 0 is the dominant solution.

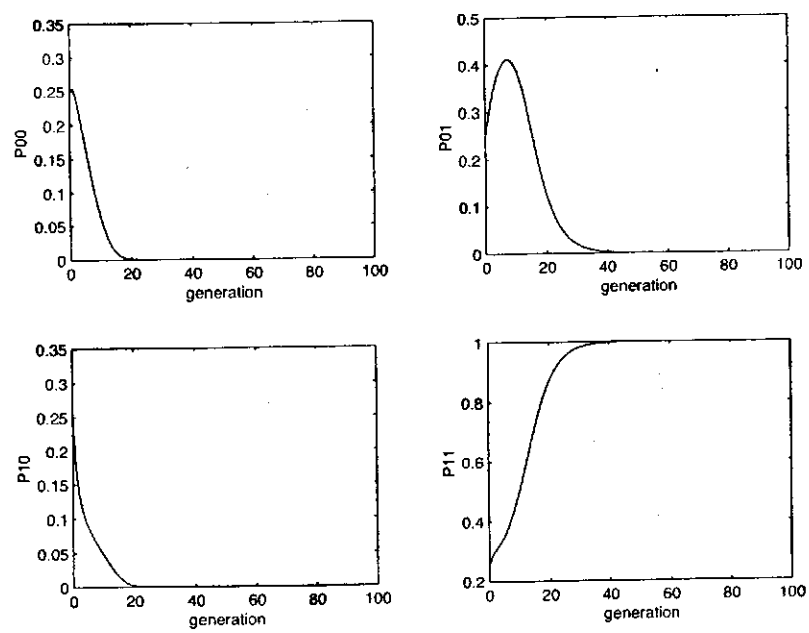


Figure 3.3 Type 2: $f_{01} = 0.9$, $f_{10} = 0.5$, $f_1 = 1.1$ and equal initial proportions.

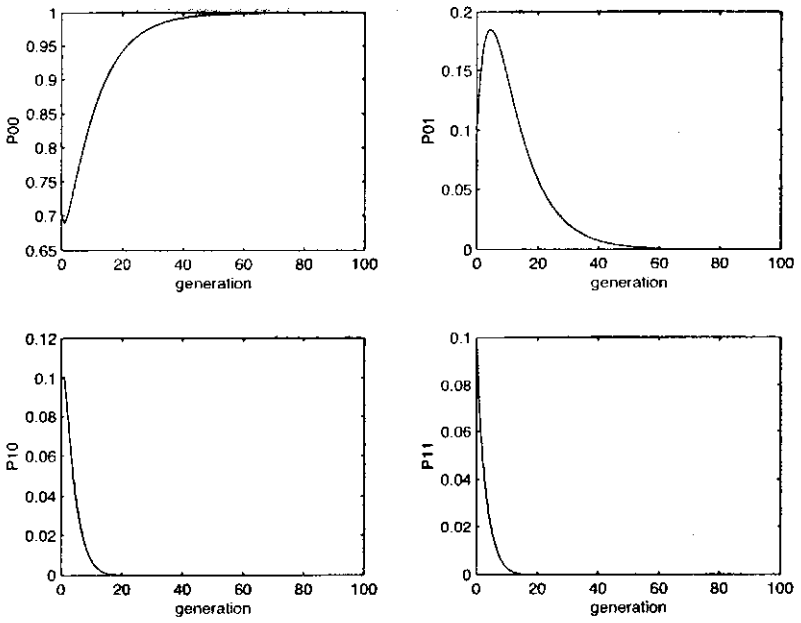
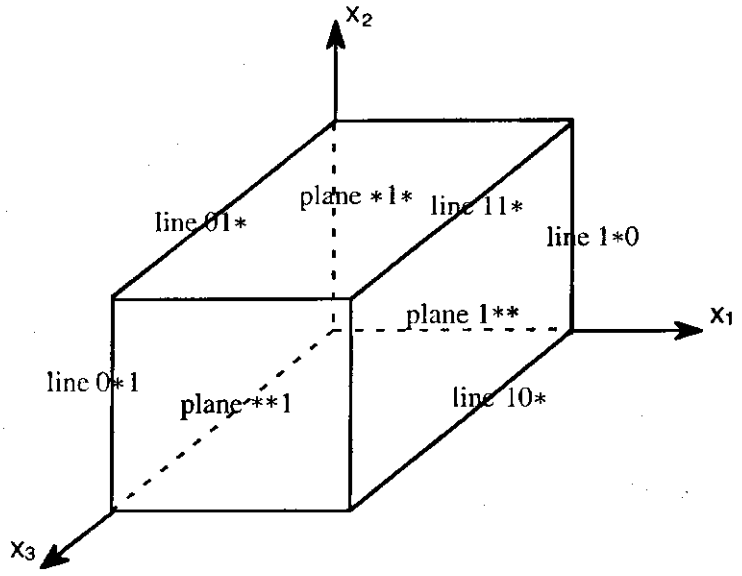


Figure 3.4 Type 2: $f_{01} = 0.9$, $f_{10} = 0.5$, $f_1 = 1.1$ and unequal initial proportions.

3.6 Similarity Templates as Hyperplanes

One can think geometrically of schemas as hyperplanes in space. Consider strings of length 3. Then specific strings can be thought of as points in $\mathbb{R}^3$. Schemas of order 2 are then equivalent to lines in space while schemas of order 1 are equivalent to planes. The schema of order 0, that is $*\ *\ *$, is the whole of $\mathbb{R}^3$. This is illustrated in the diagram below.



The result generalises to strings of length greater than 3. For strings of length n, the underlying space is $\mathbb{R}^n$, and schemas of various orders can be thought of as hyperplanes of various dimensions.

Problems: Chapter 3

1. Consider three strings $A_1 = 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1$, $A_2 = 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0$ and $A_3 = 0\ 1\ 0\ 0\ 0\ 0\ 1\ 1$ and six schemas $H_1 = 1 * * * * * * *$, $H_2 = 0 * * * * * * *$, $H_3 = * * * * * * 1\ 1$, $H_4 = * * * 0 * 0\ 0 *$, $H_5 = 1 * * * * * 1 *$ and $H_6 = 1\ 1\ 1\ 0 * * 1 *$.
   (a) Which schemas are matched by which strings?
   (b) What are the order and defining length of each of the schemas?
   (c) Estimate the probability of survival of each schema under mutation when the probability of a single mutation is $p_m = 0.001$.
   (d) Estimate the probability of survival of each schema under crossover when the probability of crossover is $p_c = 0.85$.

2. How many unique schemas exist within strings of length $l = 10$, 20 and 30 when the underlying alphabet is binary? How many unique schemas of order 3 exist in binary strings of length $l = 10$, 20 and 30?

3. Suppose a schema H when present in a particular string causes the string to have fitness 25% greater than the average fitness of the current population. If the destruction probabilities for this schema under mutation and crossover are negligible, and if a single representative of the schema is contained in the population at generation 0, determine when the schema H will overtake populations of size $n = 20, 50, 100, 200$.

4. Suppose a schema H when present in a particular string causes the string to have fitness 10% less than the average fitness of the current population. If the destruction probabilities for this schema under mutation and crossover are negligible, and if representatives of the schema are contained in 60% of the population at generation 0, determine when the schema H will disappear from populations of size $n = 20, 50, 100, 200$.

5. A fitness function for a single locus (one bit) GA has constant fitness values f_0 and f_1. Derive the difference equation for the expected proportion of 1's under reproduction alone and reproduction with mutation in an infinitely large population. Program the difference equation and calculate the expected proportion of 1's at generation 100 assuming equal proportions of 0's and 1's initially and a ratio of $f_1/f_0 = 1.1, 2, 10$ with (a) no mutation, and (b) mutation probability values of $p_m = 0.001, 0.01, 0.1$. Determine analytically the equilibrium proportion of 1's and compare with the numerical results.

Chapter 4: Function Optimisation

Much of the early work on GAs concentrated on function optimisation and this still remains one of the main applications areas today although practical applications can be found in many diverse areas. In this chapter, we look at the results of case studies by De Jong ("An Analysis of the Behavior of a Class of Genetic Adaptive Systems", PhD Dissertation, 1975) and Davis ("Handbook of Genetic Algorithms", 1991) on the optimisation of a variety of nonlinear functions using binary encodings.

Before discussing function optimisation we first look at extending and improving the simple GA described in Chapter 1 by including different selection, crossover and replacement stragies. We conclude the chapter with a discussion of how to solve constrained nonlinear programming problems using GAs.

4.1 Selection Methods

The first stage of the reproduction process is to select strings for mating. The first method proposed, and the one used in our simple GA, was fitness proportionate selection using roulette wheel selection. This stochastic method statistically results in the expected number of copies for each individual string. However, with relatively small populations typically used in GAs, the actual number of copies is often far from the expected value. To counter this a different method called *stochastic universal sampling* has been proposed.

Rather than spin the roulette wheel n times to obtain n parents for mating, this method spins the wheel once with n equally spaced pointers which are used to select the n parents. Using this method, the actual number of copies of each string selected for mating cannot differ by more than one from the expected number. The code for the function 'Reproduce' would only have to be modified slightly to incorporate this change. For example, the line of code:

$randnums = rand(size(fitness))$;

could be replaced by

$rr = rand$;
$spacing = 1/length(fitness)$;
$randnums = sort(mod(rr : spacing : 1 + rr - 0.5 * spacing, 1))$;

Ranking selection is a method in which the population is sorted according to its fitness values and then individual strings are assigned an offspring count according

to the ranking. If we have a population of n strings and sort from the best to the worst then the selection proportion assigned to the kth string is

$$p_k = q_{max} - \left(\frac{k-1}{n-1}\right)(q_{max} - q_{min})$$

where q_{max} and q_{min} are the selection proportions assigned to the most fit and the least fit strings, respectively. Since the proportions must sum to one, q_{max} and q_{min} must satisfy

$$q_{max} + q_{min} = \frac{2}{n}.$$

Ranking can be considered either as a fitness scaling technique or a selection technique. In ranking selection, we rank the parents according to their fitness and determine the number of copies according to their rank.

A method which is essentially deterministic but contains random features is *tournament selection.* In one variation of this method, a set of individuals (usually two) is randomly chosen from the population and the fittest one from the set is chosen to go into the mating pool. The set is returned to the original population and the process repeated until the mating pool is complete. There are several other variations of this method which have been proposed.

There are many other selection methods which have been proposed but the most popular appears to be stochastic universal sampling.

4.2 Crossover

Crossover is one of the main distinguishing features of GAs that make them different from other algorithms. Its main aim is to recombine building blocks on different strings. Single-point crossover, as incorporated into our simple GA, was inspired by biological processes and is the simplest form. It does, however, have some limitations, one being that it cannot combine all possible schemas. For example, the schemas 1 1 * * * * * * * * 1 and * * * * 1 * 1 * * * cannot be combined to form an instance of 1 1 * * 1 * 1 * * 1. Likewise, schemas with long defining length are likely to be destroyed under single-point crossover.

Single-point crossover assumes that short, low-order schemas are the building blocks of strings, but one does not generally know in advance what ordering of bits will group functionally related bits together. Also, single-point crossover tends to give the endpoints of strings preferential treatment as the segments exchanged between two parents always contain the endpoints of the strings.

One solution to these problems has been to use *two-point crossover*, in which two positions are chosen uniformly at random and the segments between them exchanged. Two-point crossover is less likely to disrupt schemas with large defining lengths and

can combine more schemas than one-point crossover. For example, if we apply two-point crossover to the two schemas 1 1 $*$ $*$ $*$ $*$ $*$ $*$ $*$ 1 and $*$ $*$ $*$ $*$ 1 $*$ 1 $*$ $*$ $*$ and the crossover points are at $k = 4$ and $k = 8$, this will produce the two schemas 1 1 $*$ $*$ 1 $*$ 1 $*$ $*$ 1 and $*$ $*$ $*$ $*$ $*$ $*$ $*$ $*$ $*$ $*$ $*$ and so, the schema 1 1 $*$ $*$ 1 $*$ 1 $*$ $*$ 1 is produced in one of the children.

There are schemas which two-point crossover cannot combine and there have been a number of experiments with different crossover operators. *Uniform crossover* has the ability to combine any schemas that do not differ in a single position. For each bit position on the two children we decide randomly which parent contributes its bit value to which child. The example below shows how this works, where a 1 in the template indicates that the first parent will contribute its bit value to the first child and the second parent will contribute its bit value to the second child, while a 0 in the template indicates that the first parent will contribute its bit value to the second child and the second parent will contribute its bit value to the first child.

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

Uniform crossover differs from one-point and two-point crossover in several ways. First, the location of the encoding of a feature on a chromosome is irrelevant to uniform crossover. With one-point or two-point crossover, the more bits that intervene in a schema, the less likely it is that the schema will be passed on to a child intact. Second, one-point and two-point crossover operators are more likely to preserve good features that are encoded compactly.

There is no simple answer as to which crossover operator to use. The success or failure of a particular crossover operator depends in complicated ways on the particular fitness function, the encoding and other details of the GA. Based on many invetigations, it appears that uniform crossover is favoured for small populations where the disruptiveness helps to sustain a widely explorative search. However, for large populations, the inherent diversity reduces the need for exploration and two-point crossover is generally recommended.

4.3 Population Replacement

In our simple GA, a generation was entirely replaced by its offspring created through selection, crossover and mutation. This is sometimes referred to as *nonoverlapping generations.* Because GAs are blind in nature, it is possible for the offspring to be worse than their parents and some fitter chromosomes may be lost from the evolutionary process. To overcome this problem, several different replacement strategies have been advocated in which a portion of the previous generation is retained in the new population. These are referred to as *overlapping generations.*

Looking at some of the earlier results we see that the best member of the population may fail to produce offspring that are at least as fit as itself in the next generation. The *elitist* strategy fixes this potential source of loss by copying the best member or best few members of each generation into the succeeding generation. The simplest strategy is for the elite strings to simply replace new parents at random. An alternative strategy is for the elite strings to replace the worst parents in the new population. While elitist strategies may increase the speed of dominance of a population by a super individual, they do appear to give improved performance in most cases.

Even with an elitist strategy, many of the best individuals may not reproduce at all, and their genes may be lost. One solution to this problem is to replace only one or two individuals at a time in the population. This has been called *steady-state reproduction.* A number (much less than the population size) of offspring are created and these replace the same number of the least fit individuals in the population. Steady-state GAs are often used in evolving rule-based systems (for example classifier systems) in which incremental learning is important.

When selection performs on an enlarged sampling space, both parents and offspring have the same chance of competing for survival. This strategy is sometimes referred to as $(\mu + \lambda)$ selection, where μ parents and λ offspring compete for survival and the μ best of offspring and old parents are selected as parents of the next generation. A special case of $(\mu + \lambda)$ selection is the elitist strategy described above, where λ is much smaller than μ. Another special case is when $\lambda = \mu$, that is, there is the same number of offspring as parents, as in our simple GA, and the best of the offspring and parents are chosen to form the next generation. One advantage of this latter strategy is that GA performance can be improved by increasing the mutation and crossover rates as we do not have to worry that the higher rates will destroy the fittest strings. A disadvantage of this strategy is that it is unable to deal with changing environments, especially within small populations.

4.4 De Jong and Function Optimisation

In 1975 De Jong completed his pioneering work on the application of GAs to function optimisation in which he examined and compared GA performance on five test problems. These functions, which are to be minimised, are given in Table 4.1. De Jong used binary encodings throughout.

Number	Function	Limits
F1	$f_1(\boldsymbol{x}) = \sum_{i=1}^{3} x_i^2,$	$-5.12 \leq x_i \leq 5.12$
F2	$f_2(\boldsymbol{x}) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2,$	$-2.048 \leq x_i \leq 2.048$
F3	$f_3(\boldsymbol{x}) = \sum_{i=1}^{3} \text{integer}(x_i),$	$-5.12 \leq x_i \leq 5.12$
F4	$f_4(\boldsymbol{x}) = \sum_{i=1}^{30} i x_i^4 + N(0,1),$	$-1.28 \leq x_i \leq 1.28$
F5	$f_5(\boldsymbol{x}) = 0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{ij})^6},$	$-65.536 \leq x_i \leq 65.536$

Table 4.1 Test functions for minimisation

Function F1 is a unimodal function (the level surfaces are spheres) with a global minimum of 0 at $\boldsymbol{x} = \boldsymbol{0}$. Function F2 is Rosenbrock's banana function with a global minimum of 0 at $\boldsymbol{x} = (1, 1)$. Function F3 is a three-dimensional step function, and so is discontinuous, with a global minimum of -25 which is achieved for $x_i \in [-5.12, -5]$. In function F4, $N(0, 1)$ is "white noise" of variance 1 and so F4 has a "noisy" global minimum near $\boldsymbol{x} = \boldsymbol{0}$. In function F5, the a_{ij} are randomly chosen in the interval $[-65.536, 65.536]$ and the function has 25 very sharp troughs of similar depth. All functions are nonnegative, but the problems have to be converted to maximisation problems by taking the negative of the function and adding an appropriate constant so as to ensure nonnegative fitness values.

In the study, De Jong used two performance measures to compare the effectiveness of different algorithms.

1. A measure of on-line performance using the performance measure

$$J_{on} = \frac{1}{T} \sum_{t=0}^{T} \bar{f}(t)$$

where $\bar{f}(t)$ is the average fitness value obtained in generation t. Thus, J_{on} is an average of all the fitness values up to and including generation T.

2. A measure of off-line performance using the performance measure

$$J_{off} = \frac{1}{T}\sum_{t=0}^{T} f_{max}(t)$$

where $f_{max}(t)$ is the maximum fitness value obtained in generation t. Thus, J_{off} is an average of the best fitness values up to and including generation T.

The Simple GA

The simple multivariable GA described in Chapter 2 was the first algorithm tested. As expected, larger populations lead to better off-line performance because of the larger pool of diverse schemas available in a larger population. On the other hand, smaller populations have the ability to change more rapidly and exhibit better initial on-line performance.

Mutation rates were varied from 0.001 to 0.1 ($p_m = 0.5$ is random search) using a fixed population size of $n = 50$ and $p_c = 1$. For on-line performance, the best results were almost always obtained with the lowest mutation value $p_m = 0.001$. For off-line performance, the results were not as conclusive, with $p_m = 0.01$ and 0.02 sometimes giving the best performance whereas, $p_m = 0.1$ always resulted in the worst performance.

De Jong also experimented with crossover probabilities. As a result of these studies, he recommended a crossover probability of $p_m = 0.6$ as a reasonable compromise between good on-line and off-line performance. Later studies have suggested that higher crossover rates are better when the sampling errors are reduced through the use of more accurate selection procedures, such as stochastic universal sampling.

Variations of the Simple GA

To improve performance of the simple GA, De Jong considered alternative selection and crossover operators. Among the variations he considered were:

1. Replacing roulette wheel selection with a process similar to stochastic universal sampling in which the actual number of offspring for the ith individual is limited to $f_i/\bar{f} + 1$. (Remember, in stochastic universal sampling, the actual number of offspring cannot differ by more than one from the expected number $f_i/\bar{f}$.)
2. Replacing one-point crossover with a generalised crossover model using multiple-point crossovers.

The new selection policy showed improved on-line and off-line performance compared to roulette wheel selection for all the test functions. However, multiple-point crossover degraded off-line and on-line performance as the number of crossover points

increased. A possible explanation of this can be given by counting the number of unique operators in crossover. In the case of simple crossover, there are $l-1$ operators, where l is the string length, and we choose between these operators at random. In multiple-point crossover, there are $\binom{l}{k}$ operators, where k is the number of crossovers. Hence, the number of different ways of choosing the k crossover points increases as k increases and each operator is less likely to be chosen during a particular cross, thus less structure can be preserved. With more mixing and less structure, these more involved crossover operators become more like a random shuffle and fewer important schemas can be preserved.

De Jong also compared several conventional algorithms for nonlinear optimisation with his GA which included the improved selection process. As expected, the GA outperformed the conventional algorithms on the multimodal function F5, but not on smooth, unimodal functions like F1 and F2. Of course, for smooth unimodal functions, there is no reason to use a GA in the first place.

4.5 Davis' Function Optimisation with Binary Encoding

In Davis' book, he discusses and compares the simple GA and many variations using both binary and real number encodings. In this section we will restrict our attention to algorithms using binary encodings and look at just a few variations of the simple GA. Real number encodings will be discussed later in the course.

Davis uses the function of two variables:

$$f6(x,y) = 0.5 - \frac{(\sin\sqrt{x^2+y^2})^2 - 0.5}{(1+0.001(x^2+y^2))^2}$$

as his test function. The aim is to maximise this function for $-100 \leq x, y \leq 100$. The problem has several features that make it an interesting test case. $f6$ is symmetric in x and y and has many local maxima and a unique global maximum at $x = y = 0$. The landscape is very hilly and the hill containing the global maximum occupies only a tiny fraction of the total area, thus making this a very difficult function for conventional optimisation methods to maximise. A plot of $f6$ for x varying while y is held constant at its optimal value is shown in Figure 4.1.

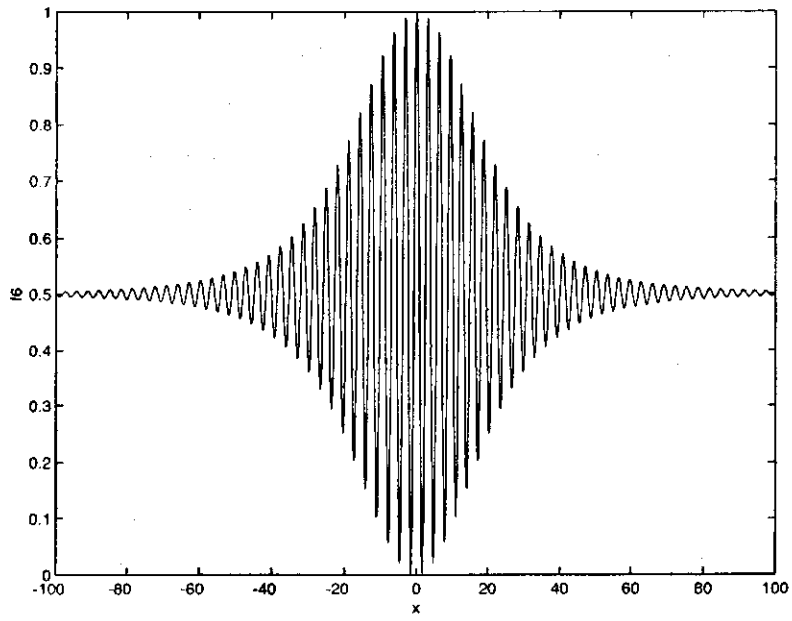


Figure 4.1 The test function $f6$ when y is constant ($y = 0$)

All the simulations will be for a population of $n = 100$ and with a fixed number of 4000 trials, that is, the simpleGA will create 40 generations of 100 individuals before stopping. The binary encoding used by Davis has 22 bits for both x and y, giving a total string length of 44 bits. This gives a precision of 4.8×10^{-5}, or slightly better than 4 decimal place accuracy.

The first run of our simple GA was with $p_c = 0.65$ and $p_m = 0.004$. A summary of the results for each generation are shown in Table 4.2. There are a number of observations we can make. Note that the best solution in the initial population disappeared in generation 2 and a better solution did not appear until generation 6. The best solution of $f6 = 0.960978$ with $(x, y) = (-2.63069, -5.74782)$ appeared in generation 16. From then on, the GA is searching around the region of the solution, as evidenced by an increasing average population fitness, with mutation the principal inducer of diversity. It would be better to terminate the algorithm after about 20 iterations and a better use of computer time might be to run from a different initial population or try some other method to refine the solution. Several other runs were tried and similar results were obtained.

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x	
0	0.503697	0.867592	4.64513	-11.7565
1	0.507463	0.867592	4.64513	-11.7565
2	0.514698	0.820131	-15.0977	-4.47166
3	0.498547	0.820131	-15.0977	-4.47166
4	0.501326	0.821075	-15.0977	-4.40943
5	0.502437	0.821075	-15.0977	-4.40943
6	0.500878	0.921811	-8.65238	-3.71592
7	0.513566	0.921811	-8.65238	-3.71592
8	0.52791	0.921811	-8.65238	-3.71592
9	0.545438	0.921811	-8.65238	-3.71592
10	0.54032	0.921811	-8.65238	-3.71592
11	0.578093	0.921811	-8.65238	-3.71626
12	0.555383	0.921811	-8.65238	-3.71626
13	0.595485	0.921811	-8.65238	-3.71626
14	0.63703	0.921811	-8.65238	-3.71626
15	0.649057	0.921811	-8.65238	-3.71511
16	0.6869	0.960978	-2.63069	-5.74782
17	0.707684	0.960978	-2.63069	-5.74782
18	0.742898	0.921811	-8.65238	-3.71459
19	0.745931	0.921811	-8.65238	-3.71473
20	0.740784	0.921811	-8.65238	-3.71473
21	0.765929	0.921811	-8.65238	-3.71473
22	0.789503	0.921811	-8.65276	-3.71363
23	0.828096	0.921811	-8.65276	-3.71363
24	0.813929	0.921811	-8.65238	-3.71473
25	0.830831	0.921811	-8.65238	-3.71473
26	0.848045	0.921811	-8.65286	-3.71363
27	0.858572	0.921811	-8.65286	-3.71363
28	0.853368	0.921811	-8.65286	-3.71363
29	0.861227	0.921811	-8.65286	-3.71363
30	0.855864	0.921811	-8.65238	-3.71473
31	0.851469	0.921811	-8.65238	-3.71502
32	0.858286	0.921811	-8.65238	-3.71463
33	0.866796	0.921811	-8.65238	-3.71363
34	0.869285	0.921811	-8.65391	-3.71153
35	0.87443	0.921811	-8.65391	-3.71153
36	0.868587	0.921811	-8.65391	-3.71153
37	0.903035	0.921811	-8.65247	-3.71363
38	0.898683	0.921811	-8.65276	-3.71363
39	0.878667	0.921811	-8.65247	-3.71363
40	0.853164	0.921811	-8.65247	-3.71363

Table 4.2 Fitness statistics for 40 generations of the simple GA

Improvements to the Simple GA

We have already discussed the reasons behind why fitness scaling might be effective. Davis tried two different scaling techniques and concluded that linear fitness scaling was the more effective of the two for this problem. For this problem, there is not a great deal of diversity in the fitness values in the initial population and so fitness scaling would not seem to be required. This also tends to be the case in later generations, but remember that fitness scaling also increases the discrimination between competitors that are close so that, in this situation, the best competitors reproduce disproportionately often.

The results of including linear fitness scaling are shown in Table 4.3. For this case, we increased the crossover probability to $p_c = 0.75$, while leaving the mutation

probability at $p_m = 0.004$. The results are significantly improved on the simple GA. The best solution of $f6 = 0.990257$ with $(x, y) = (-3.03724, -0.811506)$ appeared in generation 29. Several other runs were tried and similar results were obtained.

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x	
0	0.503697	0.867592	4.64513	-11.7565
1	0.505991	0.820017	-15.0977	-4.47776
2	0.50669	0.820069	-15.0977	-4.475
3	0.506627	0.820018	-15.0977	-4.47772
4	0.508189	0.820018	-15.0977	-4.47772
5	0.50865	0.816044	-15.0977	-4.62425
6	0.516753	0.816044	-15.0977	-4.62425
7	0.512653	0.815983	-15.0978	-4.62568
8	0.514911	0.705127	-21.7383	-4.50246
9	0.520749	0.719807	-17.6608	-5.58207
10	0.526513	0.719595	-21.7383	-4.01957
11	0.54161	0.792033	-15.226	-4.62425
12	0.553946	0.906381	-2.98832	-5.23555
13	0.555639	0.796486	-15.4883	-0.50633
14	0.557538	0.796488	-15.4883	-0.50652
15	0.559967	0.843862	-2.89056	-5.15697
16	0.570664	0.883502	-2.89056	-5.23613
17	0.583114	0.883502	-2.89056	-5.23613
18	0.56811	0.843888	-2.89056	-5.15702
19	0.572932	0.978888	-2.98793	-0.506568
20	0.591223	0.978884	-2.98793	-0.506473
21	0.613832	0.81409	-15.4395	-2.13292
22	0.623894	0.975807	-2.98798	-1.30441
23	0.648299	0.981279	-2.98879	-0.569654
24	0.654741	0.990232	-2.98879	-0.933576
25	0.684542	0.990249	-2.98879	-0.937915
26	0.740095	0.990249	-2.98879	-0.937915
27	0.743066	0.990249	-2.98879	-0.937915
28	0.759132	0.989885	-3.06356	-0.581861
29	0.753992	0.990257	-3.03724	-0.811506
30	0.791214	0.989813	-3.06356	-0.572372
31	0.790611	0.98921	-3.06327	-0.509763
32	0.793383	0.989121	-3.06351	-0.500083
33	0.808999	0.989927	-3.08006	-0.6953
34	0.790877	0.990213	-3.08592	-0.523162
35	0.794527	0.990217	-3.08616	-0.523162
36	0.816909	0.990217	-3.08616	-0.523162
37	0.853676	0.990155	-3.08602	-0.504565
38	0.856285	0.990155	-3.08602	-0.504565
39	0.862903	0.990162	-3.04906	-0.6953
40	0.83211	0.990228	-3.08602	-0.528216

Table 4.3 Fitness statistics for 40 generations of the simple GA with fitness scaling

Davis tried two-point and uniform crossover and concluded that uniform crossover is slightly superior for this problem. We will add uniform crossover to our simpleGA with fitness scaling and compare the results. One of the the difficulties when changing the operators is that the algorithm parameters p_c and p_m may no longer be optimal so we must vary these as well to try and find the values which work best for our algorithm.

The results of changing the crossover operator from one-point to uniform crossover are shown in Table 4.4. Fitness scaling is again included. For this case, we used the

probabilities of $p_c = 0.45$ and $p_m = 0.005$. The results show a slight improvement on the simple GA with fitness scaling. The best solution of $f6 = 0.993592$ with $(x, y) = (-0.0666857, 0.0443697)$ appeared in generation 39. Several other runs were tried with different initial populations. The results were variable, but one run discovered the excellent solution of $f6 = 0.999961$ at $(x, y) = (0.00603199, -0.00154972)$, in the 32nd generation.

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x	
0	0.503697	0.867592	4.64513	-11.7565
1	0.501753	0.740898	-1.55094	-12.8997
2	0.502289	0.920205	-5.28104	7.84843
3	0.510013	0.920205	-5.28104	7.84843
4	0.505201	0.771982	-11.5058	14.882
5	0.526082	0.953318	-6.17559	0.0729323
6	0.526152	0.819819	-5.15816	14.882
7	0.531687	0.796237	4.02439	14.9639
8	0.524537	0.757235	-9.85096	15.8586
9	0.516497	0.757235	-9.85096	15.8586
10	0.527599	0.757235	-9.85096	15.8586
11	0.527705	0.80801	-11.1014	6.48415
12	0.530047	0.80801	-11.1014	6.48415
13	0.52397	0.768153	-18.4077	3.55799
14	0.525164	0.761957	-18.6976	0.0857592
15	0.515236	0.749053	-9.78353	15.8525
16	0.540957	0.754437	-9.86006	0.630498
17	0.551491	0.761753	-9.81376	15.9137
18	0.556582	0.761753	-9.81376	15.9137
19	0.565747	0.771011	-11.3441	10.433
20	0.56454	0.772274	-9.95443	15.9999
21	0.570665	0.771022	-11.3441	10.4331
22	0.573517	0.771022	-11.3441	10.4331
23	0.566551	0.771022	-11.3441	10.4331
24	0.580253	0.795319	-9.78348	12.5339
25	0.569367	0.907932	-9.44174	1.39935
26	0.607883	0.907932	-9.44174	1.39935
27	0.621457	0.916101	-9.43945	1.05774
28	0.604054	0.92101	-9.39291	1.00892
29	0.574703	0.921087	-9.39138	1.00892
30	0.621237	0.965002	-3.14176	1.00892
31	0.646915	0.926389	-6.31435	1.44227
32	0.654452	0.92139	-9.39291	0.129724
33	0.663106	0.921416	-9.39062	0.270104
34	0.721541	0.989159	-3.16732	0.178647
35	0.743492	0.989823	-3.15511	0.178647
36	0.740725	0.989823	-3.15511	0.178647
37	0.742807	0.989823	-3.15511	0.178647
38	0.783238	0.990252	-3.14286	0.0931025
39	0.81302	0.993592	-0.0666857	0.0443697
40	0.846885	0.989823	-3.15511	0.178647

Table 4.4 Fitness statistics for 40 generations of the simple GA with linear scaling and uniform crossover

As mentioned in Section 4.3, the best member may fail to reproduce in the next generation. In Davis' simulations, he countered this by copying the best member of each generation into the next generation at the expense of one of the offspring, say

the one with least fitness. The results of this elitist replacement strategy are shown in Table 4.5, using the same values $p_c = 0.65$ and $p_m = 0.004$ as in the first run of the simple GA. Comparing with Table 4.2, we can see a definite improvement although convergence is still slow.

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x	
0	0.503697	0.867592	4.64513	-11.7565
1	0.507463	0.867592	4.64513	-11.7565
2	0.521815	0.867592	4.64513	-11.7565
3	0.532329	0.868933	4.64513	-11.7443
4	0.537641	0.868948	4.64513	-11.7441
5	0.541165	0.926931	4.64509	-4.51162
6	0.572436	0.926931	4.64509	-4.51162
7	0.594639	0.954654	-4.12691	-4.60475
8	0.593846	0.954994	-4.12691	-4.60742
9	0.622498	0.954994	-4.12691	-4.60742
10	0.660932	0.954994	-4.12691	-4.60742
11	0.664423	0.955016	-4.1271	-4.60742
12	0.65959	0.955058	-4.1271	-4.60775
13	0.70687	0.9627	4.25451	-4.60308
14	0.752182	0.96277	4.25933	-4.60742
15	0.80518	0.962727	4.25451	-4.60546
16	0.800866	0.962733	4.25527	-4.60546
17	0.801097	0.96277	4.25933	-4.60746
18	0.797825	0.96277	4.25933	-4.60746
19	0.787489	0.96277	4.25933	-4.60746
20	0.776605	0.96245	4.46684	-4.38345
21	0.773659	0.96245	4.46684	-4.38345
22	0.753026	0.951945	-4.12686	-4.58524
23	0.781912	0.96277	4.25933	-4.60746
24	0.770833	0.96277	4.25933	-4.60746
25	0.795764	0.962768	4.25933	-4.60689
26	0.769603	0.96277	4.25933	-4.60746
27	0.781374	0.962765	4.25933	-4.60622
28	0.790545	0.96277	4.25933	-4.60746
29	0.791565	0.962776	4.25933	-4.61118
30	0.811625	0.962776	4.25933	-4.61104
31	0.788062	0.96277	4.25933	-4.60746
32	0.770054	0.962349	4.29595	-4.60622
33	0.791557	0.962666	4.45173	-4.41001
34	0.781544	0.962776	4.25933	-4.61104
35	0.767559	0.962776	4.46699	-4.4101
36	0.774543	0.962776	4.25933	-4.61085
37	0.826198	0.962776	4.46699	-4.4101
38	0.822024	0.962776	4.46699	-4.4101
39	0.852891	0.962776	4.46699	-4.4101
40	0.826052	0.962776	4.46699	-4.4101

Table 4.5 Fitness statistics for 40 generations using a simple elitist replacement strategy

Although Davis did not use this strategy, we will try a more aggressive replacement strategy in which the parents and offspring compete for all the population spots, that is, $\mu + \lambda$ selection with $\mu = \lambda$. This allows us to use higher crossover and mutation probabilities and the values of $p_c = 0.75$ and $p_m = 0.01$ seem to be

close to optimal. The results of the first run are shown in Table 4.6 and are similar to those in Table 4.5. However, a second run with a different initial population now gives very impressive results even though we have not included fitness scaling. This is shown in Table 4.7.

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x	
0	0.503697	0.867592	4.64513	-11.7565
1	0.508326	0.867592	4.64513	-11.7565
2	0.509802	0.867592	4.64513	-11.7565
3	0.53039	0.868933	4.64513	-11.7443
4	0.561234	0.921625	5.82707	7.37765
5	0.59649	0.921625	5.82707	7.37765
6	0.672782	0.950617	5.21481	-3.69651
7	0.752719	0.921625	5.82707	7.37765
8	0.78568	0.921633	5.82707	7.37803
9	0.821143	0.921633	5.82707	7.37803
10	0.785656	0.921811	5.82707	7.39653
11	0.763477	0.950584	5.215	-3.69651
12	0.780457	0.950549	5.215	-3.6968
13	0.801794	0.950617	5.21481	-3.69651
14	0.852029	0.950617	5.21481	-3.69651
15	0.854064	0.953766	5.21481	-3.66867
16	0.822871	0.962424	5.21481	-3.52895
17	0.841983	0.962424	5.21481	-3.52895
18	0.818125	0.962424	5.21481	-3.52895
19	0.819909	0.962424	5.21481	-3.52895
20	0.831519	0.962427	5.21472	-3.52895
21	0.862256	0.962627	5.21481	-3.51675
22	0.904606	0.962775	5.1904	-3.52895
23	0.873061	0.962775	5.1904	-3.52895
24	0.86957	0.962775	5.1904	-3.52895
25	0.883483	0.962775	5.1904	-3.52895
26	0.899208	0.962776	5.19059	-3.52895
27	0.864422	0.962776	5.19059	-3.52895
28	0.85123	0.962776	5.19059	-3.52895
29	0.878127	0.962776	5.1904	-3.53048
30	0.867648	0.962776	5.1904	-3.53048
31	0.899257	0.962776	5.1904	-3.53048
32	0.892281	0.962776	5.1904	-3.53048
33	0.905438	0.962776	5.1904	-3.53048
34	0.880897	0.962776	5.1904	-3.53048
35	0.834948	0.962776	5.1904	-3.53048
36	0.901784	0.962776	5.1904	-3.53029
37	0.914267	0.962776	5.19097	-3.52933
38	0.907366	0.962776	5.19097	-3.52933
39	0.887017	0.962776	5.19097	-3.52933
40	0.885118	0.962776	5.19097	-3.52933

Table 4.6 Fitness statistics for 40 generations choosing the best offspring and parents to form the next generation - first run

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x	
0	0.498652	0.890776	3.13227	-5.76427
1	0.49634	0.887869	3.14448	-5.76427
2	0.515782	0.890055	3.13532	-5.76427
3	0.542739	0.93341	3.14448	-5.63891
4	0.568121	0.960416	3.14448	-5.37436
5	0.583212	0.955109	3.14753	-1.0844
6	0.646628	0.960549	3.14448	-5.37603
7	0.744575	0.95005	3.14448	-5.5682
8	0.78392	0.956143	3.14457	-1.08421
9	0.842531	0.989737	3.14753	-0.303149
10	0.842104	0.990168	3.14753	-0.107837
11	0.839353	0.98977	3.14753	-0.29552
12	0.891083	0.990168	3.14753	-0.107074
13	0.867687	0.990168	3.14753	-0.107074
14	0.891209	0.990169	3.14753	-0.106311
15	0.858183	0.990265	3.13227	-0.107837
16	0.878565	0.99027	3.13227	-0.123954
17	0.927549	0.99027	3.13227	-0.123954
18	0.889455	0.990284	3.13456	-0.151134
19	0.938775	0.990284	3.13685	-0.107837
20	0.939114	0.990284	3.1369	-0.107837
21	0.935044	0.990284	3.13456	-0.156665
22	0.943409	0.990284	3.13685	-0.107837
23	0.930018	0.990284	3.1369	-0.100732
24	0.971233	0.990284	3.13685	-0.0987768
25	0.936272	0.999751	0.0120401	-0.0101805
26	0.92873	0.999751	0.0120401	-0.0101805
27	0.935169	0.990284	3.13685	-0.10097
28	0.935309	0.998658	0.0120401	-0.0345945
29	0.919122	0.999845	0.0120401	-0.0030756
30	0.938004	0.999751	0.0120401	-0.0101805
31	0.937849	0.99974	0.0120401	-0.010705
32	0.909272	0.999844	0.0120401	-0.00326634
33	0.928413	0.999861	0.00593662	-0.0101805
34	0.938213	0.999849	0.0116587	-0.00388622
35	0.934315	0.999845	0.0120401	-0.0030756
36	0.943323	0.999844	0.00841618	-0.0092268
37	0.943836	0.999849	0.0116587	-0.00388622
38	0.929207	0.999849	0.0116587	-0.00388622
39	0.946492	0.999907	0.00879765	-0.00388622
40	0.954651	0.999914	0.00841618	-0.00388622

Table 4.7 Fitness statistics for 40 generations choosing the best offspring and parents to form the next generation - second run

4.6 Constraints

We have so far only considered GAs for solving unconstrained optimisation problems, or those with simple bounds on the decision variables. Many practical problems contain one or more constraints that must also be satisfied.

The general nonlinear programming problem is:

$$\begin{aligned} \text{minimise} \quad & f(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Omega \subset \mathbb{R}^n \\ \text{subject to} \quad & g_j(\boldsymbol{x}) = 0, \quad j = 1, 2, \ldots, m \\ & g_j(\boldsymbol{x}) \geq 0, \quad j = m+1, m+2, \ldots, p \end{aligned}$$

where Ω is usually defined by upper and lower bounds on the decision variables.

The main difficulty with applying GAs to these problems is how to handle constraints because the genetic operators often yield infeasible offspring. In recent years, several techniques have been proposed for handling constraints with GAs. These techniques can be roughly classified in four groups as follows:

Rejecting strategy. All the infeasible chromosomes created during the GA are discarded. This method works reasonably well when the feasible region is convex and constitutes a reasonable part of the search space Ω. This method can be very inefficient in that most of the created chromosomes may have to be discarded. Also, the system may be able to reach the optimum more easily if it is possible to cross an infeasible region.

Repairing strategy. This involves taking an infeasible chromosome and generating a feasible one through some repairing procedure. This can be easy to accomplish with many combinatorial optimisation problems and some reports indicate that this approach has been successful with these problems. Repairing strategy depends on the existence of a deterministic repair procedure to convert an infeasible offspring to a feasible one. The weakness of the method is its problem dependence, as a specific repair algorithm needs to be designed for each particular problem. Where the constraints are complex, the process of repairing infeasible chromosomes may be as complex as solving the original problem.

Modifying genetic operator strategy. Knowledge-based genetic operators can be designed for specific problems which maintain the feasibilty of chromosomes. This has proved popular in recent years and are claimed to be more reliable than GAs based on penalty functions. As with the previously mentioned techniques, the genetic search is confined within the feasible region.

Penalty strategy. The other three strategies have the advantage that they never generate infeasible solutions but have the disadvantage that they do not consider points outside the feasible region. For highly constrained problems, feasible solutions may be difficult to find if we just confine genetic search within feasible regions. The penalty technique is the most common technique used by GAs to handle constraints in constrained optimisation problems.

For *penalty function methods*, if $P(\boldsymbol{r}, \boldsymbol{x})$ is the penalty function, then most methods minimise the unconstrained function:

$$q(\boldsymbol{r}, \boldsymbol{x}) = f(\boldsymbol{x}) + P(\boldsymbol{r}, \boldsymbol{x}).$$

There are many different functional forms that have been proposed for $P(\boldsymbol{r}, \boldsymbol{x})$, but many of them are similar to, or a variant of, the function

$$P(\boldsymbol{r}, \boldsymbol{x}) = \sum_{j=1}^{m} r_j [g_j(\boldsymbol{x})]^2 + \sum_{j=m+1}^{p} r_j [\min\{0, g_j(\boldsymbol{x})\}]^2$$

where the r_j are parameters to be determined. It then remains to decide what value or values to assign to the penalty parameters r_j. Some methods choose different values of r_j for each constraint, which reflect by how much a constraint is violated, while others use the same values of r_j for each constraint. In most cases, the values of the r_j are inceased dynamically during the algorithm, that is, $r_j = r_j(t)$ is a nondecreasing function of t, the generation number. Problems that can occur in choosing the $r_j(t)$ are, if $r_j(t)$ is too small then the final solution may not satisfy the constraints and, if $r_j(t)$ is too large, the search may not converge to the correct solution.

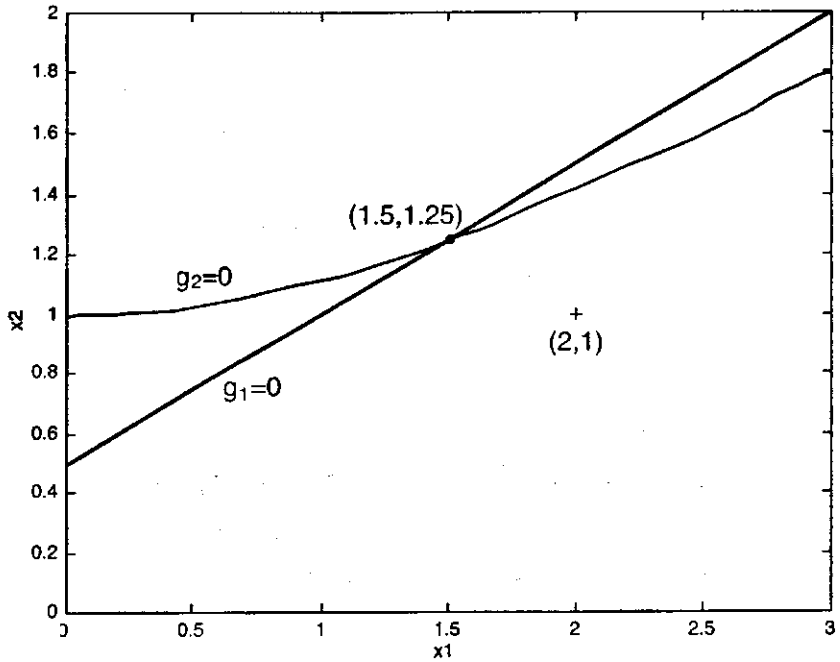


Figure 4.2 The constrained optimisation problem

We shall attempt to solve the following problem.

$$\begin{aligned} \text{minimise} \quad & f(\boldsymbol{x}) = (x_1 - 2)^2 + (x_2 - 1)^2 \\ \text{subject to} \quad & g_1(\boldsymbol{x}) = x_1 - 2x_2 + 1 = 0 \\ & g_2(\boldsymbol{x}) = \frac{x_1^2}{4} - x_2^2 + 1 \geq 0 \end{aligned}$$

This problem has a solution of $f = 0.3125$ at $(x_1, x_2) = (1.5, 1.25)$ and can be solved easily by standard derivative based optimisation methods but we will use it as an illustration of applying GAs to constrained optimisation problems. Bounds on the variables will be set at $0 \leq x_1, x_2 \leq 3$.

We first have to set up the problem for solution using a GA. The penalty function will be

$$P(r, \boldsymbol{x}) = r_1 g_1(\boldsymbol{x})^2 + r_2[\min\{0, g_2(\boldsymbol{x})\}]^2.$$

We will take $r_1 = r_2$ and vary r during the algorithm, so that $r = 1$ initially and is then increased by a multiple of 10 every n_r iterations until it reaches 10^4. We shall use the modified objective function for the maximisation problem:

$$q(r, \boldsymbol{x}) = C - f(\boldsymbol{x}) - P(r, \boldsymbol{x})$$

where C is a constant chosen so that the fitness values remain positive. For this example, the maximum value of $f(\boldsymbol{x}) + P(1, \boldsymbol{x})$ is 97 when $(x_1, x_2) = (0, 3)$ so we set $C = 100$. A check is included in the algorithm to increase C if the fitness becomes negative. The objective function and constraints are specified in two different user functions and the penalty function and modified objective function are formed in 'Evaluate'. The function 'Report' will also be different for this problem, as it will need to report values of the objective function and constraints rather than the string fitnesses.

The problem was run with a population size of 40 and values of $p_c = 0.5$ and $p_m = 0.02$, using $\mu + \lambda$ selection with $\lambda = 5$, that is, the best 5 parents of the previous generation are combined with the offspring of the current generation. Fitness scaling was also included. The same penalty parameter $r = r_1 = r_2$ was used for both constraints and was initially set at 1 and increased by a factor of 10 every 20 iterations. Table 4.8 shows the output after 90 generations. We see that, as r increases, the equality constraint is more closely satisfied, while the inequality constraint also approaches zero. A solution very close to being optimal is obtained after 84 generations in which the (absolute) value of the equality constraint is 0.0002136 and the inequality constraint is satisfied with a value of 6×10^{-6} The corresponding function value is 0.3121, very close to the optimal value of 0.3125.

Results History Statistics

Gen	r	Mean Fitness	Max Fitness	f	Constraints		Optimal x	
0	1.0	88.0501	99.7167	0.2734	-0.04007	-0.09127	1.578	1.309
1	1.0	94.9451	99.7553	0.2334	0.1066	0.09538	1.578	1.236
2	1.0	97.5813	99.8269	0.1159	0.2392	0.1486	1.858	1.309
3	1.0	98.8841	99.8269	0.1159	0.239	0.1484	1.858	1.309
4	1.0	99.3897	99.8269	0.1159	0.239	0.1484	1.858	1.309
5	1.0	99.5031	99.83	0.1223	0.2183	0.1301	1.835	1.308
6	1.0	99.7476	99.8318	0.1461	0.1484	0.02694	1.86	1.356
7	1.0	99.7297	99.8319	0.1459	0.1492	0.02793	1.86	1.355
8	1.0	99.786	99.8319	0.1459	0.1492	0.02793	1.86	1.355
9	1.0	99.7867	99.8319	0.1459	0.1492	0.02793	1.86	1.355
10	1.0	99.7865	99.8319	0.1459	0.1492	0.02793	1.86	1.355
11	1.0	99.8164	99.8319	0.146	0.1488	0.02759	1.859	1.355
12	1.0	99.8018	99.8324	0.1317	0.1895	0.0845	1.854	1.332
13	1.0	99.8104	99.8325	0.1377	0.1726	0.05956	1.86	1.343
14	1.0	99.7967	99.8326	0.1435	0.1548	0.03855	1.852	1.349
15	1.0	99.8279	99.8326	0.1435	0.1548	0.03855	1.852	1.349
16	1.0	99.7481	99.8328	0.1394	0.1668	0.05412	1.854	1.343
17	1.0	99.7739	99.8328	0.1394	0.1668	0.05412	1.854	1.343
18	1.0	99.7941	99.8328	0.1394	0.1668	0.05427	1.853	1.343
19	1.0	99.7912	99.8329	0.1388	0.1682	0.0567	1.852	1.342
20	10.0	99.7943	99.833	0.1381	0.1702	0.07406	1.817	1.323
21	10.0	99.6218	99.833	0.1381	0.1702	0.07406	1.817	1.323
22	10.0	98.9009	99.833	0.1381	0.1702	0.07406	1.817	1.323
23	10.0	478.716	499.767	0.1932	0.0481	-0.04118	1.712	1.332
24	10.0	592.535	599.768	0.1999	0.04755	-0.03067	1.688	1.32
25	10.0	162.003	599.768	0.1999	0.04755	-0.03067	1.688	1.32
26	10.0	511.686	599.768	0.1999	0.04755	-0.03067	1.688	1.32
27	10.0	511.371	599.768	0.1999	0.04755	-0.03067	1.688	1.32
28	10.0	161.796	599.768	0.1999	0.04755	-0.03067	1.688	1.32
29	10.0	162.012	599.768	0.1999	0.04755	-0.03067	1.688	1.32
30	10.0	424.223	599.768	0.1999	0.04755	-0.03067	1.688	1.32
31	10.0	161.915	599.768	0.1999	0.04755	-0.03067	1.688	1.32
32	10.0	162.071	599.768	0.1999	0.04755	-0.03067	1.688	1.32
33	10.0	162.204	599.768	0.1999	0.04755	-0.03067	1.688	1.32
34	10.0	162.219	599.768	0.1999	0.04755	-0.03067	1.688	1.32
35	10.0	162.17	599.768	0.1999	0.04755	-0.03067	1.688	1.32
36	10.0	162.242	599.768	0.1999	0.04755	-0.03067	1.688	1.32
37	10.0	599.678	599.768	0.1998	0.04791	-0.03018	1.688	1.32
38	10.0	512.133	599.768	0.1998	0.04791	-0.03018	1.688	1.32
39	10.0	511.993	599.768	0.1998	0.04791	-0.03018	1.688	1.32
40	100.0	162.187	599.768	0.1998	0.04791	-0.03018	1.688	1.32
41	100.0	687.501	699.481	0.2001	0.04718	-0.03097	1.688	1.32
42	100.0	173.717	699.481	0.2001	0.04718	-0.03097	1.688	1.32
43	100.0	4568.68	4799.48	0.2001	0.04736	-0.03073	1.688	1.32
44	100.0	1096.97	4799.48	0.2001	0.04736	-0.03073	1.688	1.32
45	100.0	859.668	4799.48	0.2001	0.04736	-0.03073	1.688	1.32
46	100.0	5454.33	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
47	100.0	1031.13	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
48	100.0	947.952	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
49	100.0	4884.53	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
50	100.0	1034.79	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
51	100.0	947.897	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
52	100.0	947.906	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
53	100.0	4971.97	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
54	100.0	4884.74	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
55	100.0	4971.66	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
56	100.0	1022.52	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
57	100.0	1034.87	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
58	100.0	769.772	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
59	100.0	4882.23	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
60	1000.0	5140.52	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
61	1000.0	2732.28	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
62	1000.0	5258.27	5499.48	0.2004	0.04608	-0.0326	1.688	1.321
63	1000.0	45321.4	46697.6	0.3012	0.04599	0.05676	1.5	1.227
64	1000.0	8040.98	46697.6	0.3012	0.04599	0.05676	1.5	1.227
65	1000.0	47691.1	47797.4	0.3009	0.04746	0.05856	1.5	1.226
66	1000.0	7236.43	47797.4	0.3009	0.04746	0.05856	1.5	1.226
67	1000.0	47717	47797.6	0.3012	0.04599	0.05676	1.5	1.227
68	1000.0	6389.58	47797.6	0.3012	0.04599	0.05676	1.5	1.227
69	1000.0	8313.3	47797.6	0.3012	0.04599	0.05676	1.5	1.227

70	1000.0	45245.8	47797.6	0.3012	0.04599	0.05676	1.5	1.227
71	1000.0	50603.3	50799.7	0.312	0.0005799	0.0005189	1.5	1.25
72	1000.0	42445.5	50799.7	0.312	0.0005799	0.0005189	1.5	1.25
73	1000.0	44221	50799.7	0.312	0.0005799	0.0005189	1.5	1.25
74	1000.0	42640.8	50799.7	0.312	0.0005799	0.0005189	1.5	1.25
75	1000.0	60907.9	61299.7	0.312	0.0005799	0.0005189	1.5	1.25
76	1000.0	49192.9	61299.7	0.312	0.0005799	0.0005189	1.5	1.25
77	1000.0	9739.08	61299.7	0.312	0.0005799	0.0005189	1.5	1.25
78	1000.0	47810.6	61299.7	0.312	0.0005799	0.0005189	1.5	1.25
79	1000.0	46939.8	61299.7	0.312	0.0005799	0.0005189	1.5	1.25
80	10000.0	9926.51	61299.7	0.312	0.0005799	0.0005189	1.5	1.25
81	10000.0	438181	450500	0.312	0.0005799	0.0005189	1.5	1.25
82	10000.0	433185	450500	0.312	0.0005799	0.0005189	1.5	1.25
83	10000.0	399229	450500	0.312	0.0005799	0.0005189	1.5	1.25
84	10000.0	485033	487600	0.3121	0.0002136	6.107e-05	1.5	1.25
85	10000.0	453954	487600	0.3121	0.0002136	6.107e-05	1.5	1.25
86	10000.0	84652.9	487600	0.3121	0.0002136	6.107e-05	1.5	1.25
87	10000.0	454399	487600	0.3121	0.0002136	6.107e-05	1.5	1.25
88	10000.0	88641.9	487600	0.3121	0.0002136	6.107e-05	1.5	1.25
89	10000.0	451422	487600	0.3121	0.0002136	6.107e-05	1.5	1.25
90	10000.0	400130	487600	0.3121	0.0002136	6.107e-05	1.5	1.25

Table 4.8 Results for the constrained problem

Other runs were tried with different initial populations and gave results which were, in general, not quite as good as those shown in Table 4.8. It should be stressed that this type of problem is inherently difficult for GAs to solve because of the constraints. There have been many different methods proposed for solving problems with complex constraints but none have proved very successful on other than a limited range of problems.

Problems: Chapter 4

1. In a ranking selection method, assume the median population member is assigned one copy. If the highest ranking population member is assigned MAX copies, calculate a formula relating the population size and MAX to the number of copies, MIN, received by the lowest ranking population member. Assume a straight line variation in copy allocation. What restrictions, if any, exist on the values of MAX and MIN?

2. Six strings have the following fitness function values: $f_1 = 1, f_2 = 2, f_3 = 3, f_4 = 4, f_5 = 5, f_6 = 6$. Under roulette wheel selection, calculate the expected number of copies of the string with $f = 6$ in the mating pool if a constant population size, $n = 6$, is maintained. What is the minimum and maximum possible number of times the string might be picked? Repeat for stochastic universal sampling.

3. Calculate the probabilities of the following schemas surviving crossover under one-point, two-point and uniform crossover. Assume $p_c = 1$.

$$****1******, \qquad 1*********1, \qquad ****10*****$$

4. Estimate the probability of survival of a schema of length δ and order o under uniform crossover.

5. Consider the 2-variable maximisation problem discussed at the end of Chapter 2. Choose the replacement strategy in which the best of the parents and offspring form the next generation. Run the algorithm "binaryGA.m" and compare your results with those shown in Chapter 2. Make sure you choose optimal, or near optimal, values of p_c and p_m for each simulation.

Appendix: Review of Combinatorics and Probability

Here we review some elementary concepts in combinatorics and probability which will be required to help us understand the fundamental mathematics of GAs.

A.1 Permutations and Combinations

The number of permutations of n different objects is $n!$

Example Suppose we have 4 books on genetic algorithms, 5 books on control theory and 6 books on microelectronics. If we place the books on a bookshelf, how many arrangements of these books are there if (a) the books can be arranged in any order (b) books on each subject are grouped together?
(Answer: (a) $15! = 1,307,674,368,000$ (b) $3!4!5!6! = 12,441,600$)

Let $P(n,r)$ be the number of permutations of n objects taken r at a time. $P(n,r)$ is the number of unique (different) orderings of r objects chosen from n objects. Then

$$P(n,r) = n(n-1)\cdots(n-r+1) = \frac{n!}{(n-r)!}.$$

Example Suppose a squad of baseball players contains 15 players. How many 9-person batting orders can be formed from the squad?
(Answer: $P(15,9) = 1,816,214,400$)

Let $C(n,r)$ be the number of combinations among n objects taken r at a time. Note that, in combinations, the ordering of the objects is unimportant, hence

$$C(n,r) = \frac{P(n,r)}{r!} = \binom{n}{r} = \frac{n!}{r!(n-r)!}.$$

Example How many 11-person soccer teams can be formed from a squad of 15 players?
(Answer: $C(15,11) = \frac{15!}{11!4!} = 1365$)

A.2 Sample Space and Events

Consider an experiment whose outcome is not predictable with certainty in advance. Suppose the set of all possible outcomes is known in advance. This set is called the *Sample Space* of the experiment and denoted by S. Any subset E of the sample space is known as an *event.*

Example For a single roll of a die, the sample space is

$$S = \{1, 2, 3, 4, 5, 6\}.$$

An example of an event would be a roll of the die with value greater than 3, that is,

$$E = \{4, 5, 6\}$$

For any two events E and F of a sample space S, we define the new event $E \cup F$, called the *union* of E and F to consist of all outcomes in E or F or both. Similarly we define the new event EF, called the *intersection* of E and F to consist of all outcomes in both E and F. If $EF = \emptyset$, that is EF contains no outcomes, then E and F are said to be *mutually exclusive.*

For any event E we define the new event E^c, called the *complement* of E to consist of all outcomes in S that are not in E.

The above events can be represented graphically by using a *Venn diagram.*

A.3 Axioms of Probability

If an experiment is continually repeated under the exact same conditions, then for any event E, the proportion of time that the outcome is contained in E approaches some constant value as the number of repetitions increases. This constant limiting value is what we mean when we speak of the probability of an event, denoted by $P(E)$ which satisfies the following axioms:

Axiom 1 $0 \leq P(E) \leq 1$

Axiom 2 $P(S) = 1$

Axiom 3 For any sequence of mutually exclusive events $E_1, E_2, \ldots E_n$,

$$P\left(\bigcup_{i=1}^{n} E_i\right) = \sum_{i=1}^{n} P(E_i)$$

From these axioms we can easily obtain the following results:

$$P(E^c) = 1 - P(E)$$
$$P(E \cup F) = P(E) + P(F) - P(EF)$$

Suppose we have a situation where all events are equally likely, for example, tossing a coin or rolling a die. Then we can calculate the probability of an event E easily:

$$P(E) = \frac{\text{no. of points in } E}{\text{no. of points in } S}$$

In this case, the calculation of probabilities becomes a counting exercise.

Example What is the probability a head appears at least once in 10 tosses of a fair coin?

Solution: This problem is more easily solved by calculating the probability of the complement. Clearly,

$$P(\text{at least 1 head}) = 1 - P(\text{no heads}) = 1 - P(TTTTTTTTTT) = 1 - \left(\tfrac{1}{2}\right)^{10} \simeq 0.99902$$

Example What is the probability of being dealt a straight (including a straight flush) in 5 cards?

Solution: Consider the straight consisting of [A 2 3 4 5] in the sequence of cards

$$[\text{A } 2\ 3\ 4\ 5]\ 6\ 7\ 8\ 9\ 10\ \text{J Q K A}$$

Since there are 4 suits, each of the cards in the straight can be chosen 4 ways and so there are 4^5 possible straights consisting of [A 2 3 4 5]. Now there are 10 different straights and so the total number of all possible straights is $10 \cdot 4^5$. The total number of hands of 5 cards in a deck of 52 cards is $C(n, r) = \binom{52}{5}$ and so the probability of being dealt a straight is

$$\frac{10 \cdot 4^5}{\binom{52}{5}} \simeq 0.00394$$

A.4 Conditional Probability

Conditional probability deals with the occurence of an event given that a related event has occurred. The conditional probability of an event E given that an event F has occurred is denoted by $P(E|F)$ and we have the relationship

$$P(EF) = P(E|F)\,P(F)$$

For events E and F we have $E = (EF) \cup (EF^c)$. Since EF and EF^c are mutually exclusive, this leads to the relationship:

$$\begin{aligned} P(E) = P(EF) + P(EF^c) &= P(E|F)\,P(F) + P(E|F^c)\,P(F^c) \\ &= P(E|F)\,P(F) + P(E|F^c)(1 - P(F)) \end{aligned}$$

Example A student has a 75% chance of receiving an A in a genetic algorithms course and a 50% chance of receiving an A in a computational methods course . The student is only allowed to take one of the courses and makes the decision based on the toss of a coin. What is the probabilty of receiving an A?

Solution: Let A be the event that the student receives an A, G be the event the student takes the genetic algorithms course and C the event the student takes the computational methods course. Then,

$$P(A) = P(A|G)P(G) + P(A|C)P(C) = (0.75)(0.5) + (0.5)(0.5) = 0.625$$

From an earlier formula we can write

$$P(EF) = P(E|F)P(F) = P(F|E)P(E)$$

This leads to an important result in conditional probability known as *Bayes' rule*:

$$P(E|F) = \frac{P(EF)}{P(F)} = \frac{P(F|E)P(E)}{P(F|E)P(E) + P(F|E^c)P(E^c)}$$

Example A lab test is 99% effective in detecting a certain disease when it is, in fact, present. However, the test also yields a "false positive" result for 1% of the healthy persons tested. That is, if a healthy person is tested, then, with probability 0.01, the test result will imply the person has the disease. If 0.5% of the population actually has the disease, what is the probability that the lab test will be positive given that the person has the disease?

Solution: Let D be the event that the tested person has the disease and E the event that the test result is positive. Then, the desired probability is $P(D|E)$. From Bayes' rule:

$$\begin{aligned} P(D|E) &= \frac{P(E|D)P(D)}{P(E|D)P(D) + P(E|D^c)P(D^c)} \\ &= \frac{(0.99)(0.005)}{(0.99)(0.005) + (0.01)(0.995)} = 0.3322 \end{aligned}$$

Two events E and F are said to be *independent* when the conditional probability $P(E|F) = P(E)$. This then gives $P(EF) = P(E)P(F)$ for independent events.

Often we perform a sequence of experiments (trials) where each trial has a constant probability of success. Clearly, if $P(\text{success}) = p$ then $P(\text{failure}) = 1 - p$. If we perform a sequence of n trials then we may wish to know the probability of achieving exactly k successes. This is given by

$$P(k \text{ successes in } n \text{ trials}) = \binom{n}{k} p^k (1-p)^{n-k}$$

since a particular sequence of k successes and $n-k$ failures has probability $p^k(1-p)^{n-k}$ and there are exactly $\binom{n}{k}$ such sequences. This probability distribution is called a *binomial* probability distribution.

A.5 Random Variables

Quantities that are determined by the results of random experiments are called *random variables.* Since the value of a random variable is determined by the outcome of a random experiment, we assign probabilities to its possible values. A random variable that can take on at most a countable number of possible values is said to be *discrete.*

For a discrete random variable X, we define the *probability mass function* $p(x)$ of X by

$$p(x) = P(X = x).$$

If x represents the values of the X, then we must have

$$p(x) \geq 0 \qquad \text{and} \qquad \sum_x p(x) = 1.$$

If X is a discrete random variable, then the *expection* or *expected value* of X is denoted by $E[X]$ and defined by

$$E[X] = \sum_x xp(x).$$

Example What is the expected value when we roll a fair die?

Solution: Let X be the outcome when we roll the die.

$$E[X] = 1.\frac{1}{6} + 2.\frac{1}{6} + 3.\frac{1}{6} + 4.\frac{1}{6} + 5.\frac{1}{6} + 6.\frac{1}{6} = 3.5$$

We may also be interested in the expected value of some function, g, of a random variable X. Then,

$$E[g(X)] = \sum_x g(x)p(x).$$

From the previous example, it appears that the expected value of a random variable is, in some way, its average value. This result is known as the *strong law of large numbers* and is one of the most important limit theorems in probability. The statement of the theorem is:

Suppose we have a sequence of independent, identically distributed, random variables $X_1, X_2, \ldots, X_n$ with finite expected values. Then

$$\frac{X_1 + X_2 + \cdots + X_n}{n} \to E[X] \quad \text{as} \quad n \to \infty.$$

Problems

1. How many unique permutations may be formed from the letters of the following words?

   (a) turgid

   (b) sleeper

2. A credit card number is constructed as a 10-position code where each position is taken from the full alphabet (A-Z) or the decimal digits (0-9). How many different credit card numbers may be constructed in this manner? In storing the credit card code on a computer, the programmer wants to use the minimum number of bits to represent the code. Calculate the length of the minimum binary code required to hold the credit card code.

3. A committee consists of 13 first-year students, 6 second-year students, 7 third-year students and 5 final-year students. If a committee chairperson is chosen at random, what is the probability that the chairperson is (a) a final-year student; (b) a first-year student; (c) not a second-year student; (d) a first-year student or a third-year student?

4. In a poker hand of five cards, what is the probability of being dealt four of a kind?

5. A drawer contains 20 white socks and 10 black socks. If five socks are selected all at once and at random, what are the chances of picking exactly two black socks? How does the answer change if the socks are selected one at time and are replaced in the drawer after each selection?

6. On a multiple-choice exam with four alternative answers per question and five questions, what is the probability of getting four or more questions correct by random guessing?

7. A gambler has two coins in his left pocket. One is fair, the other is two-headed. A coin is selected at random from this pocket and tossed. Calculate the probability the coin will come up heads. If it does come up heads, what is the probability that the chosen coin is the fair coin?

8. A million-ticket lottery pays prizes as follows:

1	ticket pays	\$500,000
10	tickets pay	\$50,000
100	tickets pay	\$1,000
1,000	tickets pay	\$100
10,000	tickets pay	\$10

If a ticket costs \$2, what is the ticket holder's expected loss in this lottery? What is the probability of winning some prize?