CHAPTER 1

INTRODUCTION

Genetic algorithms (GAs) are numerical optimisation algorithms inspired by both natural selection and natural genetics. The method is a general one, capable of being applied to an extremely wide range of problems. Unlike some approaches, their promise has rarely been over-sold and they are being used to help solve practical problems on a daily basis. The algorithms are simple to understand and the required computer code easy to write. Although there is a growing number of disciples of GAs, the technique has never attracted the attention that, for example, artificial neural networks have. Why this should be is difficult to say. It is certainly not because of any inherent limits or for lack of a powerful metaphor. What could be more inspiring than generalising the ideas of Darwin and others to help solve other real-world problems? The concept that evolution, starting from not much more than a chemical "mess", generated the (unfortunately vanishing) bio-diversity we see around us today is a powerful, if not awe-inspiring, paradigm for solving any complex problem.

In many ways the thought of extending the concept of natural selection and natural genetics to other problems is such an obvious one that one might be left wondering why it was not tried earlier. In fact it was. From the very beginning, computer scientists have had visions of systems that mimicked one or more of the attributes of life. The idea of using a population of solutions to solve practical engineering optimisation problems was considered several times during the 1950's and 1960's. However, GAs were in essence invented by one man—John Holland—in the 1960's. His reasons for developing such algorithms went far beyond the type of problem solving with which this text is concerned. His 1975 book, *Adaptation in Natural and Artificial Systems* [HO75] (recently re-issued with additions) is particularly worth reading for its visionary approach. More recently others, for example De Jong, in a paper entitled *Genetic Algorithms are NOT Function Optimizers* [DE93], have been keen to remind us that GAs are potentially far more than just a robust method for estimating a series of unknown parameters within a model of a physical system. However in the context of this text, it is this robustness across many different practical optimisation problems that concerns us most.

So what is a GA? A typical algorithm might consist of the following:

1. a number, or population, of guesses of the solution to the problem;
2. a way of calculating how good or bad the individual solutions within the population are;
3. a method for mixing fragments of the better solutions to form new, on average even better solutions; and
4. a mutation operator to avoid permanent loss of diversity within the solutions.

With typically so few components, it is possible to start to get the idea of just how simple it is to produce a GA to solve a specific problem. There are no complex mathematics, or torturous, impenetrable algorithms. However, the downside of this is that there are few hard and fast rules to what exactly a GA is.

Before proceeding further and discussing the various ways in which GAs have been constructed, a sample of the range of the problems to which they have been successfully applied will be presented, and an indication given of what is meant by the phrase "search and optimisation".

1.1 Some Applications of Genetic Algorithms

Why attempt to use a GA rather than a more traditional method? One answer to this is simply that GAs have proved themselves capable of solving many large complex problems where other methods have experienced difficulties. Examples are large-scale combinatorial optimisation problems (such as gas pipe layouts) and real-valued parameter estimations (such as image registrations) within complex search spaces riddled with many local optima. It is this ability to tackle search spaces with many local optima that is one of the main reasons for an increasing number of scientists and engineers using such algorithms.

Amongst the many practical problems and areas to which GAs have been successfully applied are:

- image processing [CH97,KA97];
- prediction of three dimensional protein structures [SC92];
- VLSI (very large scale integration) electronic chip layouts [COH91,ES94];
- laser technology [CA96a,CA96b];
- medicine [YA98];
- spacecraft trajectories [RA96];
- analysis of time series [MA96,ME92,ME92a,PA90];
- solid-state physics [SU94,WA96];
- aeronautics [BR89,YA95];
- liquid crystals [MIK97];
- robotics [ZA97, p161-202];
- water networks [HA97,SA97];
- evolving cellular automaton rules [PA88,MI93,MI94a];
- the architectural aspects of building design [MIG95,FU93];
- the automatic evolution of computer software [KO91,KO92,KO94];
- aesthetics [CO97a];
- jobshop scheduling [KO95,NA91,YA95];
- facial recognition [CA91];
- training and designing artificial intelligence systems such as artificial neural networks [ZA97, p99-117,WH92, KI90,KI94,CH90]; and
- control [NO91,CH96,CO97].

1.2 Search Spaces

In a numerical search or optimisation problem, a list, quite possibly of infinite length, of possible solutions is being searched in order to locate the solution that best describes the problem at hand. An example might be trying to find the best values for a set of adjustable parameters (or variables) that, when included in a mathematical model, maximise the lift generated by an aeroplane's wing. If there were only two of these adjustable parameters, a and b, one could try a large number of combinations, calculate the lift generated by each design and produce a surface plot with a, b and *lift* plotted on the x-, y- and z-axis respectively (Figure 1.0). Such a plot is a representation of the problem's *search space*. For more complex problems, with more than two unknowns, the situation becomes harder to visualise. However, the concept of a search space is still valid as long as some measure of distance between solutions can be defined and each solution can be assigned a measure of success, or *fitness*, within the problem. Better performing, or fitter, solutions will then occupy the peaks within the search space (or *fitness landscape* [WR31]) and poorer solutions the valleys.

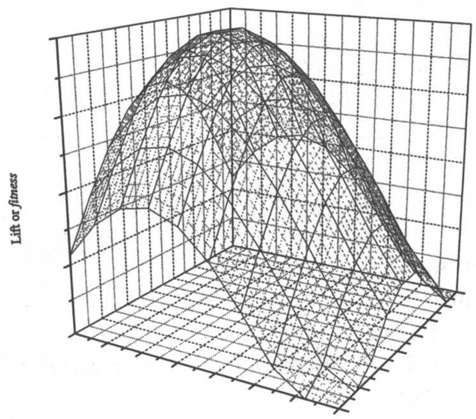

Figure 1.0. A simple search space or "fitness landscape". The lift generated by the wing is a function of the two adjustable parameters *a* and *b*. Those combinations which generate more lift are assigned a higher fitness. Typically, the desire is to find the combination of the adjustable parameters that gives the highest fitness.

Such spaces or landscapes can be of surprisingly complex topography. Even for simple problems, there can be numerous peaks of varying heights, separated from each other by valleys on all scales. The highest peak is usually referred to as the *global maximum* or *global optimum*, the lesser peaks as *local maxima* or *local optima*. For most search problems, the goal is the accurate identification of the global optimum, but this need not be so. In some situations, for example real-time control, the identification of **any** point above a certain value of fitness might be acceptable. For other problems, for example, in architectural design, the identification of a large number of highly fit, yet distant and therefore distinct, solutions (designs) might be required.

To see why many traditional algorithms can encounter difficulties, when searching such spaces for the global optimum, requires an understanding of how the features within spaces are formed. Consider the experimental data shown in Figure 1.1, where measurements of a dependent variable y have been made at various points j of the independent variable x. Clearly there is some evidence that x and y might be related through:

$$y_j = mx_j + c \ . \tag{1.1}$$

But what values should be given to m and c? If there is reason to believe that $y = 0$ when $x = 0$ (i.e. the line passes through the origin) then $c = 0$ and m is the only adjustable parameter (or *unknown*).

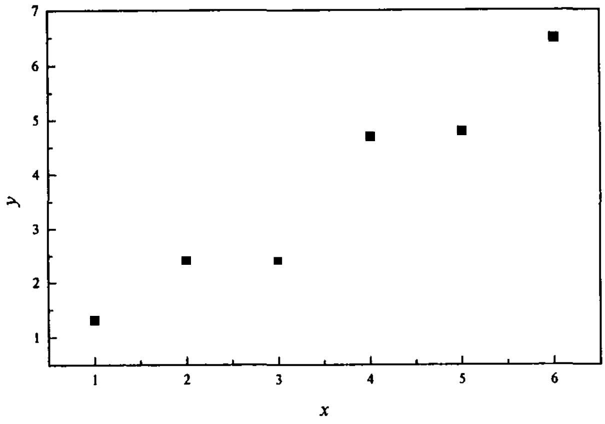

Figure 1.1. Some simple experimental data possibly related by $y = mx + c$.

One way of then finding m is simply to use a ruler and estimate the best line through the points by eye. The value of m is then given by the slope of the line. However there are more accurate approaches. A common numerical way of finding the best estimate of m is by use of a least-squares estimation. In this technique the error between that y predicted using (1.1) and that measured during the experiment, $\bar{y}$, is characterised by the objective function, Ω, (in this case the least squares cost function) given by,

$$\Omega = \sum_{j=1}^{n} \left(\bar{y}_j - y_j \right)^2 \tag{1.2}$$

where n is the number of data points. Expanding (1.2) gives:

$$\Omega = \sum_{j=1}^{n} \left(\bar{y}_j - \left(mx_j + c\right)\right)^2 .$$

As $c = 0$,

$$\Omega = \sum_{j=1}^{n} \left(\bar{y}_j - mx_j\right)^2 . \qquad (1.3)$$

In essence the method simply calculates the sum of the squares of the vertical distances between measured values of y and those predicted by (1.1) (see Figure 1.2). Ω will be at a minimum when these distances sum to a minimum. The value of m which gives this value is then the best estimate of m. This still leaves the problem of finding the lowest value of Ω. One way to do this (and a quite reasonable approach given such an easy problem with relatively few data points) is to use a computer to calculate Ω over a fine grid of values of m. Then simply choose the m which generates the lowest value of Ω. This approach was used together with the data of Figure 1.1 to produce a visualisation of the problem's search space—Figure 1.3. Clearly, the best value of m is given by $m = m^* \approx 1.1$, the asterisk indicating the optimal value of the parameter.

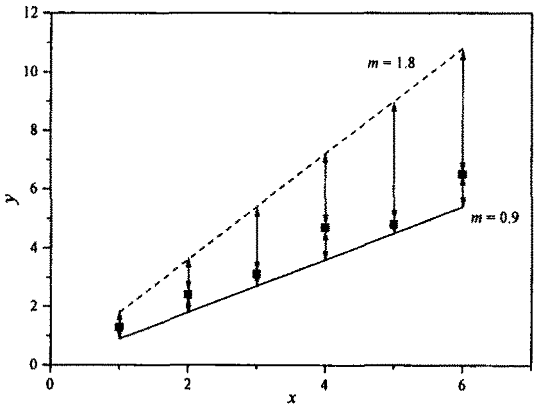

Figure 1.2. Calculating Ω for two values of m. Clearly m = 0.9 is the better choice as the sum of distances will generate a lesser value of Ω.

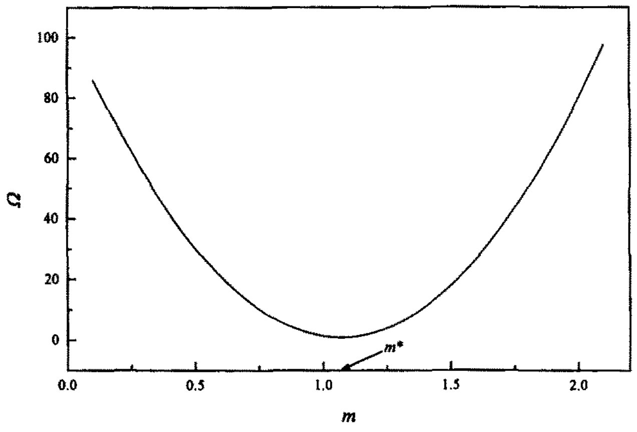

Figure 1.3. A simple search space, created from the data of Figure 1.1, Equation (1.3) and a large number of guesses of the value of *m*. This is an example of a minimisation problem, where the optimum is located at the lowest point.

This approach, of estimating an unknown parameter, or parameters, by simply solving the problem for a very large number of values of the unknowns is called an enumerative search. It is only really useful if there are relatively few unknown parameters and one can estimate Ω rapidly. As an example why such an approach can quickly run into problems of scale, consider the following. A problem in which there are ten unknowns, each of which are required to an accuracy of one percent, will require 100^{10}, or $1\text{x}10^{20}$, estimations. If the computer can make 1000 estimations per second, then the answer will take over $3\text{x}10^9$ years to emerge. Given that ten is not a very large number of unknowns, one percent not a very demanding level of accuracy and one thousand evaluations per second more than respectable for many problems, clearly there is a need to find a better approach.

Returning to Figure 1.3, a brief consideration of the shape of the curve suggests another approach: guess two possible values of m, labelled m_1 and m_2 (see Figure 1.4), then if $\Omega(m_1) > \Omega(m_2)$, make the next guess at some point m_3 where $m_3 = m_2 + \delta$, or else head the other way. Given some suitable, dynamic, way of adjusting the value of δ, the method will rapidly home in on m^*.

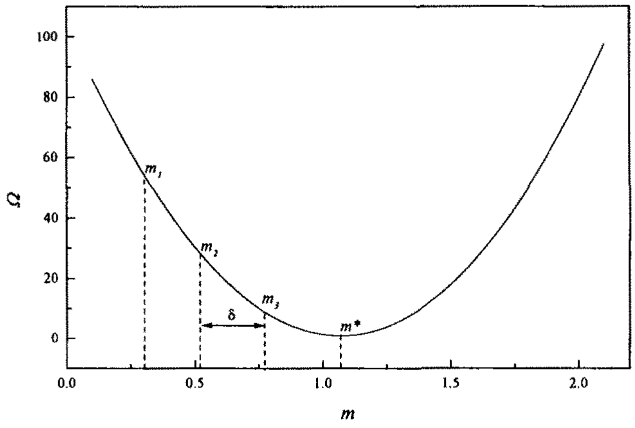

Figure 1.4. A simple, yet effective, method of locating m^*. δ is reduced as the minimum is approached.

Such an approach is described as a *direct* search (because it does not make use of derivatives or other information). The problem illustrated is one of minimisation. If $1/\Omega$ were plotted, the problem would have been transformed into one of maximisation and the desire would been to locate the top of the hill.

Unfortunately, such methods cannot be universally applied. Given a different problem, still with a single adjustable parameter, a, Ω might take the form shown in Figure 1.5.

If either the direct search algorithm outlined above or a simple calculus based approach is used, the final estimate of a will depend on where in the search space the algorithm was started. Making the initial guess at $a = a_2$, will indeed lead to the correct (or *global*) minimum, a^*. However, if $a = a_1$ is used then only a^{**} will be reached (a *local* minimum).

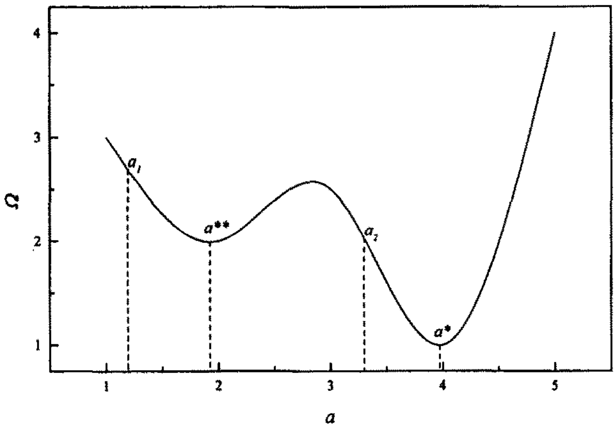

Figure 1.5. A more complex one-dimensional search space with both a global and a local minimum.

This highlights a serious problem. If the results produced by a search algorithm depend on the starting point, then there will be little confidence in the answers generated. In the case illustrated, one way around this problem would be to start the problem from a series of points and then assume that the true global minimum lies at the lowest minimum identified. This is a frequently adopted strategy. Unfortunately Figure 1.5 represents a very simple search space. In a more complex space (such as Figure 1.6) there may be very many local optima and the approach becomes unrealistic.

So, how are complex spaces to be tackled? Many possible approaches have been suggested and found favour, such as random searches and simulated annealing [DA87]. Some of the most successful and robust have proved to be random searches directed by analogies with natural selection and natural genetics—genetic algorithms.

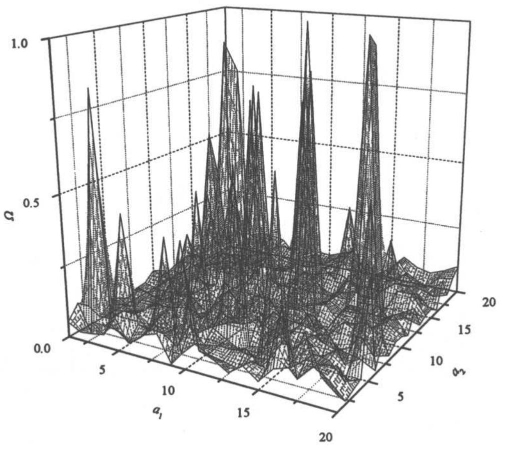

Figure 1.6. Even in a two-dimensional maximisation problem the search space can be highly complex.

1.3 GENETIC ALGORITHMS

Rather than starting from a single point (or guess) within the search space, GAs are initialised with a *population* of guesses. These are usually random and will be spread throughout the search space. A typical algorithm then uses three operators, *selection*, *crossover* and *mutation* (chosen in part by analogy with the natural world) to direct the population (over a series of time steps or *generations*) towards convergence at the global optimum.

Typically, these initial guesses are held as binary encodings (or *strings*) of the true variables, although an increasing number of GAs use "real-valued" (i.e. base-10) encodings, or encodings that have been chosen to mimic in some manner the natural data structure of the problem. This initial population is then processed by the three main operators.

Selection attempts to apply pressure upon the population in a manner similar to that of natural selection found in biological systems. Poorer performing individuals are weeded out and better performing, or *fitter*, individuals have a greater than average chance of promoting the information they contain within the next generation.

Crossover allows solutions to exchange information in a way similar to that used by a natural organism undergoing sexual reproduction. One method (termed *single point crossover*) is to choose pairs of individuals promoted by the selection operator, randomly choose a single locus (point) within the binary strings and swap all the information (digits) to the right of this locus between the two individuals.

Mutation is used to randomly change (flip) the value of single bits within individual strings. Mutation is typically used very sparingly.

After selection, crossover and mutation have been applied to the initial population, a new population will have been formed and the generational counter is increased by one. This process of selection, crossover and mutation is continued until a fixed number of generations have elapsed or some form of convergence criterion has been met.

On a first encounter it is far from obvious that this process is ever likely to discover the global optimum, let alone form the basis of a general and highly effective search algorithm. However, the application of the technique to numerous problems across a wide diversity of fields has shown that it does exactly this. The ultimate proof of the utility of the approach possibly lies with the demonstrated success of life on earth.

1.4 An Example

There are many things that have to be decided upon before applying a GA to a particular problem, including:

- the method of encoding the unknown parameters (as binary strings, base-10 numbers, etc.);
- how to exchange information contained between the strings or encodings;
- the population size—typical values are in the range 20 to 1000, but can be smaller or much larger;
- how to apply the concept of mutation to the representation; and
- the termination criterion.

Many papers have been written discussing the advantages of one encoding over another; or how, for a particular problem, the population size might be chosen [GO89b]; about the difference in performance of various exchange mechanisms and on whether mutation rates ought to be high or low. However, these papers have naturally concerned themselves with computer experiments, using a small number of simple test functions, and it is often not clear how general such results are. In reality the only way to proceed is to look at what others with similar problems have tried, then choose an approach that both seems sensible for the problem at hand and that you have confidence in being able to code up.

A trivial problem might be to maximise a function, $f(x)$, where

$$f(x) = x^2 \text{ ; for integer } x \text{ and } 0 \le x \le 4095.$$

There are of course other ways of finding the answer ($x = 4095$) to this problem than using a GA, but its simplicity makes it ideal as an example.

Firstly, the exact form of the algorithm must be decided upon. As mentioned earlier, GAs can take many forms. This allows a wealth of freedom in the details of the algorithm. The following (Algorithm 1) represents just one possibility.

1. Form a population, of eight random binary strings of length twelve (e.g. *101001101010*, *110011001100*, ...).
2. Decode each binary string to an integer x (i.e. *000000000111* implies $x = 7$, *000000000000* = 0, *111111111111* = 4095).
3. Test these numbers as solutions to the problem $f(x) = x^2$ and assign a fitness to each individual equal to the value of $f(x)$ (e.g. the solution $x = 7$ has a fitness of $7^2 = 49$).
4. Select the best half (those with highest fitness) of the population to go forward to the next generation.
5. Pick pairs of *parent* strings at random (with each string being selected exactly once) from these more successful individuals to undergo single point crossover. Taking each pair in turn, choose a random point between the end points of the string, cut the strings at this point and exchange the tails, creating pairs of *child* strings.
6. Apply mutation to the children by occasionally (with probability one in six) flipping a *0* to a *1* or vice versa.
7. Allow these new strings, together with their parents to form the new population, which will still contain only eight members.
8. Return to Step 2, and repeat until fifty generations have elapsed.

Algorithm 1. A very simple genetic algorithm.

To further clarify the crossover operator, imagine two strings, *000100011100* and *111001101010*. Performing crossover between the third and fourth characters produces two new strings:

parents	**children**
000/100011100	*000001101010*
111/001101010	*111100011100*

It is this process of crossover which is responsible for much of the power of genetic algorithms.

Returning to the example, let the initial population be:

population member	**string**	x	**fitness**
1	*110101100100*	3428	11751184
2	*010100010111*	1303	1697809
3	*101111101110*	3054	9326916
4	*010100001100*	1292	1669264
5	*011101011101*	1885	3553225
6	*101101001001*	2889	8346321
7	*101011011010*	2778	7717284
8	*010011010101*	1237	1530169

Population members 1, 3, 6 and 7 have the highest fitness. Deleting those four with the least fitness provides a temporary reduced population ready to undergo crossover:

temp. pop. member	**string**	x	**fitness**
1	*110101100100*	3428	11751184
2	*101111101110*	3054	9326916
3	*101101001001*	2889	8346321
4	*101011011010*	2778	7717284

Pairs of strings are now chosen at random (each exactly once): 1 is paired with 2, 3 with 4. Selecting, again at random, a crossover point for each pair of strings (marked by a /), four new children are formed and the new population, consisting of parents and offspring only, becomes (note that mutation is being ignored at present):

population member	**string**	x	**fitness**
1	*11/0101100100*	3428	11751184
2	*10/1111101110*	3054	9326916
3	*101101/001001*	2889	8346321
4	*101011/011010*	2778	7717284
5	*111111101110*	4078	16630084
6	*100101100100*	2404	5779216
7	*101101011010*	2906	8444836
8	*101011001001*	2761	7623121

The initial population had an average fitness f_{ave} of 5065797 and the fittest individual had a fitness, f_{max}, of 11751184. In the second generation, these have risen to: $f_{ave} = 8402107$ and $f_{max} = 16630084$. The next temporary population becomes:

temp. pop. member	**string**	x	**fitness**
1	*110101100100*	3428	11751184
2	*101111101110*	3054	9326916
3	*101101011010*	2906	8444836
4	*111111101110*	4078	16630084

This temporary population does not contain a *1* as the last digit in any of the strings (whereas the initial population did). This implies that no string from this moment on can contain such a digit and the maximum value that can evolve will be *111111111110*—after which point this string will reproduce so as to dominate the population. This domination of the population by a single sub-optimal string gives a first indication of why mutation might be important. Any further populations will only contain the same, identical string. This is because the crossover operator can only swap bits between strings, not introduce any new information. Mutation can thus be seen in part as an operator charged with maintaining the genetic diversity of the population by preserving the diversity embodied in the initial generation. (For a discussion of the relative benefits of mutation and crossover, see [SP93a].)

The inclusion of mutation allows the population to leapfrog over this sticking point. It is worth reiterating that the initial population did include a *1* in **all** positions. Thus the mutation operator is not necessarily inventing new information but simply working as an insurance policy against premature loss of genetic information.

Rerunning the algorithm from the same initial population, but with mutation, allows the string *111111111111* to evolve and the global optimum to be found. The progress of the algorithm (starting with a different initial population), with and without mutation, as a function of generation is shown in Figure 1.7. Mutation has been included by visiting every bit in each new child string, throwing a random number between 0 and 1 and if this number is less than $^1/_{12}$, flipping the value of the bit.

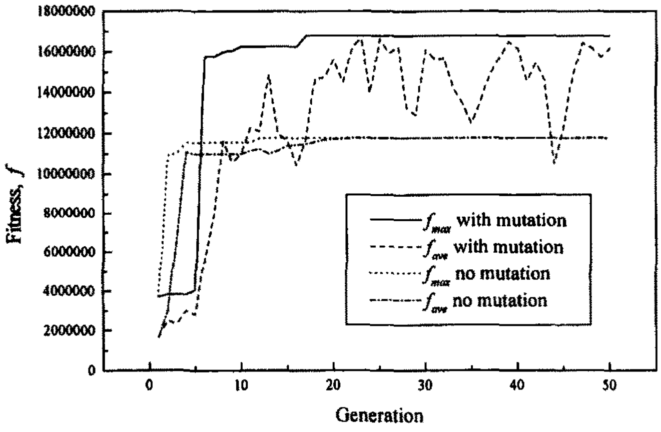

Figure 1.7. The evolution of the population. The fitness of the best performing individual, f_{max}, is seen to improve with generation as is the average fitness of the population, f_{ave}. Without mutation the lack of a *1* in all positions limits the final solution.

Although a genetic algorithm has now been successfully constructed and applied to a simple problem, it is obvious that many questions remain. In particular, how are problems with more than one unknown dealt with, and how are problems with real (or complex) valued parameters to be tackled? These and other questions are discussed in the next chapter.

1.5 SUMMARY

In this chapter genetic algorithms have been introduced as general search algorithms based on metaphors with natural selection and natural genetics. The central differences between the approach and more traditional algorithms are:

the manipulation of a population of solutions in parallel, rather than the sequential adjustment of a single solution; the use of encoded representations of the solutions, rather than the solutions themselves; and the use of a series of stochastic (i.e. random based) operators.

The approach has been shown to be successful over a growing range of difficult problems. Much of this proven utility arises from the way the population navigates its way around complex search spaces (or *fitness landscapes*) so as to avoid entrapment by local optima.

The three central operators behind the method are selection, crossover and mutation. Using these operators a very simple GA has been constructed (Algorithm 1) and applied to a trivial problem. In the next chapter these operators will be combined once more, but in a form capable of tackling more difficult problems.

1.6 EXERCISES

1. Given a string of length ten, what is the greatest value of an unknown Algorithm 1 can search for?

2. What is the resolution of Algorithm 1 when working with a string length of thirty?

3. Given a string length of 20 and a probability of mutation of $^1/_{20}$ per bit, what is the probability that a string will emerge from the mutation operator unscathed?

4. Implement Algorithm 1 on a computer and adapt it to find the value of x that maximises $\sin^4(x)$, $0 \le x \le \pi$ to an accuracy of at least one part in a million (Use a population size of fifty and a mutation rate of 1/(twice the string length).) This will require finding a transformation between the binary strings and x such that *000...000* implies $x = 0$ and *111...111* implies $x = \pi$.

5. Experiment with your program and the problem of Question 4 by estimating the average number of evaluations of $\sin^4(x)$ required to locate the maximum; (a) as a function of the population size, (b) with, and without, the use of crossover. (Use a mutation rate of 1/(twice the string length).)