- genetic operators that alter the composition of children during reproduction,
- values for various parameters that the genetic algorithm uses (population size, probabilities of applying genetic operators, etc.).

We discuss the main features of genetic algorithms by presenting three examples. In the first one we apply a genetic algorithm for optimization of a simple function of one real variable. The second example illustrates the use of a genetic algorithm to learn a strategy for a simple game (the prisoner's dilemma). The third example discusses one possible application of a genetic algorithm to approach a combinatorial NP-hard problem, the traveling salesman problem.

1.1 Optimization of a simple function

In this section we discuss the basic features of a genetic algorithm for optimization of a simple function of one variable. The function is defined as

$$f(x) = x \cdot \sin(10\pi \cdot x) + 1.0$$

and is drawn in Figure 1.1. The problem is to find x from the range $[-1..2]$ which maximizes the function f, i.e., to find x_0 such that

$$f(x_0) \geq f(x), \text{ for all } x \in [-1..2].$$

It is relatively easy to analyse the function f. The zeros of the first derivative f' should be determined:

$$f'(x) = \sin(10\pi \cdot x) + 10\pi x \cdot \cos(10\pi \cdot x) = 0;$$

the formula is equivalent to

$$\tan(10\pi \cdot x) = -10\pi x.$$

It is clear that the above equation has an infinite number of solutions,

$x_i = \frac{2i-1}{20} + \epsilon_i$, for $i = 1, 2, \ldots$
$x_0 = 0$
$x_i = \frac{2i+1}{20} - \epsilon_i$, for $i = -1, -2, \ldots$,

where terms ϵ_i represent decreasing sequences of real numbers (for $i = 1, 2, \ldots$, and $i = -1, -2, \ldots$) approaching zero.

Note also that the function f reaches its local maxima for x_i if i is an odd integer, and its local minima for x_i if i is an even integer (see Figure 1.1).

Since the domain of the problem is $x \in [-1..2]$, the function reaches its maximum for for $x_{19} = \frac{37}{20} + \epsilon_{19} = 1.85 + \epsilon_{19}$, where $f(x_{19})$ is slightly larger than $f(1.85) = 1.85 \cdot \sin(18\pi + \frac{\pi}{2}) + 1.0 = 2.85$.

Assume that we wish to construct a genetic algorithm to solve the above problem, i.e., to maximize the function f. Let us discuss the major components of such a genetic algorithm in turn.

"Genetic Algorithms + Data Structures = Evolution Programs"
Z. Michalewicz Springer-Verlag 1992

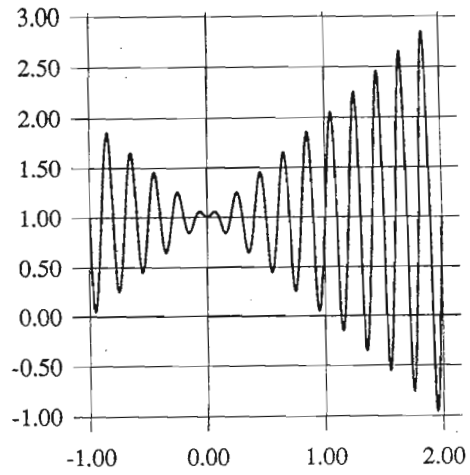

Fig. 1.1. Graph of the function $f(x) = x \cdot \sin(10\pi \cdot x) + 1.0$

1.1.1 Representation

We use a binary vector as a chromosome to represent real values of the variabl x. The length of the vector depends on the required precision, which, in thi example, is six places after the decimal point.

The domain of the variable x has length 3; the precision requirement implie that the range $[-1..2]$ should be divided into at least $3 \cdot 1000000$ equal size ranges This means that 22 bits are required as a binary vector (chromosome):

$$2097152 = 2^{21} < 3000000 \leq 2^{22} = 4194304.$$

The mapping from a binary string $\langle b_{21} b_{20} \ldots b_0 \rangle$ into a real number x fron the range $[-1..2]$ is straightforward and is completed in two steps:

- convert the binary string $\langle b_{21} b_{20} \ldots b_0 \rangle$ from the base 2 to base 10:

  $$(\langle b_{21} b_{20} \ldots b_0 \rangle)_2 = (\textstyle\sum_{i=0}^{21} b_i \cdot 10^i)_{10} = x',$$

- find a corresponding real number x:

  $$x = -1.0 + x' \cdot \frac{3}{2^{22}-1},$$

  where -1.0 is the left boundary of the domain and 3 is the length of th domain.

For example, a chromosome

(1000101110110101000111)

represents the number 0.637197, since

$$x' = (1000101110110101000111)_2 = 2288967$$

and

$$x = -1.0 + 2288967 \cdot \frac{3}{4194303} = 0.637197.$$

Of course, the chromosomes

(0000000000000000000000) and (1111111111111111111111)

represent boundaries of the domain, -1.0 and 2.0, respectively.

1.1.2 Initial population

The initialization process is very simple: we create a population of chromosomes, where each chromosome is a binary vector of 22 bits. All 22 bits for each chromosome are initialized randomly.

1.1.3 Evaluation function

Evaluation function *eval* for binary vectors $\mathbf{v}$ is equivalent to the function f:

$$eval(\mathbf{v}) = f(x),$$

where the chromosome $\mathbf{v}$ represents the real value x.

As noted earlier, the evaluation function plays the role of the environment, rating potential solutions in terms of their fitness. For example, three chromosomes:

$$\mathbf{v_1} = (1000101110110101000111),$$
$$\mathbf{v_2} = (0000001110000000010000),$$
$$\mathbf{v_3} = (1110000000111111000101),$$

correspond to values $x_1 = 0.637197$, $x_2 = -0.958973$, and $x_3 = 1.627888$, respectively. Consequently, the evaluation function would rate them as follows:

$$eval(\mathbf{v_1}) = f(x_1) = 1.586345,$$
$$eval(\mathbf{v_2}) = f(x_2) = 0.078878,$$
$$eval(\mathbf{v_3}) = f(x_3) = 2.250650.$$

Clearly, the chromosome $\mathbf{v_3}$ is the best of the three chromosomes, since its evaluation returns the highest value.

1.1.4 Genetic operators

During the reproduction phase of the genetic algorithm we would use two classical genetic operators: mutation and crossover.

As mentioned earlier, mutation alters one or more genes (positions in a chromosome) with a probability equal to the mutation rate. Assume that the fifth gene from the $\mathbf{v_3}$ chromosome was selected for a mutation. Since the fifth gene in this chromosome is 0, it would be flipped into 1. So the chromosome $\mathbf{v_3}$ after this mutation would be

$$\mathbf{v_3}' = (1110\mathbf{1}00000111111000101).$$

This chromosome represents the value $x_3' = 1.721638$ and $f(x_3') = -0.082257$. This means that this particular mutation resulted in a significant decrease of the value of the chromosome $\mathbf{v_3}$. On the other hand, if the 10th gene was selected for mutation in the chromosome $\mathbf{v_3}$, then

$$\mathbf{v_3}'' = (1110000001111111000101).$$

The corresponding value $x_3'' = 1.630818$ and $f(x_3'') = 2.343555$, an improvement over the original value of $f(x_3) = 2.250650$.

Let us illustrate the crossover operator on chromosomes $\mathbf{v_2}$ and $\mathbf{v_3}$. Assume that the crossover point was (randomly) selected after the 5th gene:

$$\mathbf{v_2} = (00000|01110000000010000),$$
$$\mathbf{v_3} = (11100|00000111111000101).$$

The two resulting offspring are

$$\mathbf{v_2}' = (00000|00000111111000101),$$
$$\mathbf{v_3}' = (11100|01110000000010000).$$

These offspring evaluate to

$$f(\mathbf{v_2}') = f(-0.998113) = 0.940865,$$
$$f(\mathbf{v_3}') = f(1.666028) = 2.459245.$$

Note that the second offspring has a better evaluation than both of its parents.

1.1.5 Parameters

For this particular problem we have used the following parameters: population size $pop\_size = 50$, probability of crossover $p_c = 0.25$, probability of mutation $p_m = 0.01$. The following section presents some experimental results for such a genetic system.

1.1.6 Experimental results

In Table 1.1 we provide the generation number for which we noted an improvement in the evaluation function, together with the value of the function. The best chromosome after 150 generations was

$$v_{max} = (11110011010001000000101),$$

which corresponds to a value $x_{max} = 1.850773$.

As expected, $x_{max} = 1.85 + \epsilon$, and $f(x_{max})$ is slightly larger than 2.85.

Generation number	Evaluation function
1	1.441942
6	2.250003
8	2.250283
9	2.250284
10	2.250363
12	2.328077
39	2.344251
40	2.345087
51	2.738930
99	2.849246
137	2.850217
145	2.850227

Table 1.1. Results of 150 generations

1.2 The prisoner's dilemma

In this section, we explain how a genetic algorithm can be used to learn a strategy for a simple game, known as the prisoner's dilemma. We present the results obtained by Axelrod [6].

Two prisoners are held in separate cells, unable to communicate with each other. Each prisoner is asked, independently, to defect and betray the other prisoner. If only one prisoner defects, he is rewarded and the other is punished. If both defect, both remain imprisoned and are tortured. If neither defects, both receive moderate rewards. Thus, the selfish choice of defection always yields a higher payoff than cooperation — no matter what the other prisoner does — but if both defect, both do worse than if both had cooperated. The prisoner's dilemma is to decide whether to defect or cooperate with the other prisoner.

The prisoner's dilemma can be played as a game between two players, where at each turn, each player either defects or cooperates with the other prisoner. The players then score according to the payoffs listed in the Table 1.2.

Player 1	Player 2	P_1	P_2	Comment
Defect	Defect	1	1	Punishment for mutual defection
Defect	Cooperate	5	0	Temptation to defect and sucker's payoff
Cooperate	Defect	0	5	Sucker's payoff, and temptation to defect
Cooperate	Cooperate	3	3	Reward for mutual cooperation

Table 1.2. Payoff table for prisoner's dilemma game: P_i is the payoff for Player i

We will now consider how a genetic algorithm might be used to learn a strategy for the prisoner's dilemma. A GA approach is to maintain a population of "players", each of which has a particular strategy. Initially, each player's strategy is chosen at random. Thereafter, at each step, players play games and their scores are noted. Some of the players are then selected for the next generation, and some of those are chosen to mate. When two players mate, the new player created has a strategy constructed from the strategies of its parents (crossover). A mutation, as usual, introduces some variability into players' strategies by random changes on representations of these strategies.

1.2.1 Representing a strategy

First of all, we need some way to represent a strategy (i.e., a possible solution). For simplicity, we will consider strategies that are deterministic and use the outcomes of the three previous moves to make a choice in the current move. Since there are four possible outcomes for each move, there are $4 \times 4 \times 4 = 64$ different histories of the three previous moves.

A strategy of this type can be specified by indicating what move is to be made for each of these possible histories. Thus, a strategy can be represented by a string of 64 bits (or Ds and Cs), indicating what move is to be made for each of the 64 possible histories. To get the strategy started at the beginning of the game, we also need to specify its initial premises about the three hypothetical moves which preceded the start of the game. This requires six more genes, making a total of seventy loci on the chromosome.

This string of seventy bits specifies what the player would do in every possible circumstance and thus completely defines a particular strategy. The string of 70 genes also serves as the player's chromosome for use in the evolution process.

1.2.2 Outline of the genetic algorithm

Axelrod's genetic algorithm to learn a strategy for the prisoner's dilemma works in four stages, as follows:

1. Choose an initial population. Each player is assigned a random string of seventy bits, representing a strategy as discussed above.

2. Test each player to determine its effectiveness. Each player uses the strategy defined by its chromosome to play the game with other players. The player's score is its average over all the games it plays.

3. Select players to breed. A player with an average score is given one mating; a player scoring one standard deviation above the average is given two matings; and a player scoring one standard deviation below the average is given no matings.

4. The successful players are randomly paired off to produce two offspring per mating. The strategy of each offspring is determined from the strategies of its parents. This is done by using two genetics operators: crossover and mutation.

After these four stages we get a new population. The new population will display patterns of behavior that are more like those of the successful individuals of the previous generation, and less like those of the unsuccessful ones. With each new generation, the individuals with relatively high scores will be more likely to pass on parts of their strategies, while the relatively unsuccessful individuals will be less likely to have any parts of their strategies passed on.

1.2.3 Experimental results

Running this program, Axelrod obtained quite remarkable results. From a strictly random start, the genetic algorithm evolved populations whose median member was just as successful as the best known heuristic algorithm. Some behavioral patterns evolved in the vast majority of the individuals; these are:

1. Don't rock the boat: continue to cooperate after three mutual cooperations (i.e., C after (CC)(CC)(CC)[2]).

2. Be provokable: defect when the other player defects out of the blue (i.e., D after receiving (CC)(CC)(CD)).

3. Accept an apology: continue to cooperate after cooperation has been restored (i.e., C after (CD)(DC)(CC)).

4. Forget: cooperate when mutual cooperation has been restored after an exploitation (i.e., C after (DC)(CC)(CC)).

5. Accept a rut: defect after three mutual defections (i.e., D after (DD)(DD)(DD)).

For more details, see [6].

[2]The last three moves are described by three pairs $(a_1b_1)(a_2b_2)(a_3b_3)$, where the a's are this player's moves (C for cooperate, D for defect) and the b's are the other player's moves.

1.3 Traveling salesman problem

In this section, we explain how a genetic algorithm can be used to approach the Traveling Salesman Problem (TSP). Note that we shall discuss only one possible approach. In Chapter 10 we discuss other approaches to the TSP as well.

Simply stated, the traveling salesman must visit every city in his territory exactly once and then return to the starting point; given the cost of travel between all cities, how should he plan his itinerary for minimum total cost of the entire tour?

The TSP is a problem in combinatorial optimization and arises in numerous applications. There are several branch-and-bound algorithms, approximate algorithms, and heuristic search algorithms which approach this problem. During the last few years there have been several attempts to approximate the TSP by genetic algorithms [72, pages 166–179]; here we present one of them.

First, we should address an important question connected with the chromosome representation: should we leave a chromosome to be an integer vector, or rather we should transform it into a binary string? In the previous two examples (optimization of a function and the prisoner's dilemma) we represented a chromosome (in a more or less natural way) as a binary vector. This allowed us to use binary mutation and crossover; applying these operators we got legal offspring, i.e., offspring within the search space. This is not the case for the traveling salesman problem. In a binary representation of a n cities TSP problem, each city should be coded as a string of $\lceil \log_2 n \rceil$ bits; a chromosome is a string of $n \cdot \lceil \log_2 n \rceil$ bits. A mutation can result in a sequence of cities, which is not a tour: we can get the same city twice in a sequence. Moreover, for a TSP with 20 cities (where we need 5 bits to represent a city), some 5-bit sequences (for example, 10101) do not correspond to any city. Similar problems are present when applying crossover operator. Clearly, if we use mutation and crossover operators as defined earlier, we would need some sort of a "repair algorithm"; such an algorithm would "repair" a chromosome, moving it back into the search space.

It seems that the integer vector representation is better: instead of using repair algorithms, we can incorporate the knowledge of the problem into operators: in that way they would "intelligently" avoid building an illegal individual. In this particular approach we accept integer representation: a vector $\mathbf{v} = \langle i_1 i_2 \ldots i_n \rangle$ represents a tour: from i_1 to i_2, etc., from i_{n-1} to i_n and back to i_1 ($\mathbf{v}$ is a permutation of $\langle\ 1\ 2\ \ldots\ n\ \rangle$).

For the initialization process we can either use some heuristics (for example, we can accept a few outputs from a greedy algorithm for the TSP, starting from different cities), or we can initialize the population by a random sample of permutations of $\langle\ 1\ 2\ \ldots\ n\ \rangle$.

The evaluation of a chromosome is straightforward: given the cost of travel between all cities, we can easily calculate the total cost of the entire tour.

In the TSP we search for the best ordering of cities in a tour. It is relatively easy to come up with some unary operators (unary type operators) which would search for better string orderings. However, using only unary operators, there is a little hope of finding even good orderings (not to mention the best one) [70]. Moreover, the strength of genetic algorithms arises from the structured information exchange of crossover combinations of highly fit individuals. So what we need is a crossover-like operator that would exploit important similarities between chromosomes. For that purpose we use a variant of a OX operator [31], which, given two parents, builds offspring by choosing a subsequence of a tour from one parent and preserving the relative order of cities from the other parent. For example, if the parents are

$$\langle\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ \rangle \text{ and}$$
$$\langle\ 7\ 3\ 1\ 11\ 4\ 12\ 5\ 2\ 10\ 9\ 6\ 8\ \rangle$$

and the chosen part is

$$(4\ 5\ 6\ 7),$$

the resulting offspring is

$$\langle\ 1\ 11\ 12\ 4\ 5\ 6\ 7\ 2\ 10\ 9\ 8\ 3\ \rangle.$$

As required, the offspring bears a structural relationship to both parents. The roles of the parents can then be reversed in constructing a second offspring.

A genetic algorithm based on the above operator outperforms random search, but leaves much room for improvements. Typical (average over 20 random runs) results from the algorithm, as applied to 100 randomly generated cities, gave (after 20000 generations) a value of the whole tour 9.4% above optimum.

For full discussion on the TSP, the representation issues and genetic operators used, the reader is referred to Chapter 10.

1.4 Hillclimbing, simulated annealing, and genetic algorithms

In this section we discuss three algorithms, i.e., hillclimbing, simulated annealing, and the genetic algorithm, applied to a simple optimization problem. This example underlines the uniqueness of the GA approach.

The search space is a set of binary strings $\mathbf{v}$ of the length 30. The objective function f to be maximized is given as

$$f(\mathbf{v}) = |11 \cdot one(\mathbf{v}) - 150|,$$

where the function $one(\mathbf{v})$ returns the number of 1s in the string $\mathbf{v}$.

For example, the following three strings

$\mathbf{v}_1 = (110110101110101111111101101011)$,
$\mathbf{v}_2 = (111000100100110111001010100011)$,
$\mathbf{v}_3 = (000010000011001000000010001000)$,

would evaluate to

$f(\mathbf{v}_1) = |11 \cdot 22 - 150| = 92$,
$f(\mathbf{v}_2) = |11 \cdot 15 - 150| = 15$,
$f(\mathbf{v}_3) = |11 \cdot 6 - 150| = 84$,

($one(\mathbf{v}_1) = 22$, $one(\mathbf{v}_2) = 15$, and $one(\mathbf{v}_3) = 6$).

The function f is linear and does not provide any challenge as an optimiz tion task. We use it only to illustrate the ideas behind these three algorithm However, the interesting characteristic of the function f is that it has one glob maximum for

$\mathbf{v}_g = (111111111111111111111111111111)$,

$f(\mathbf{v}_g) = |11 \cdot 30 - 150| = 180$, and one local maximum for

$\mathbf{v}_l = (000000000000000000000000000000)$,

$f(\mathbf{v}_l) = |11 \cdot 0 - 150| = 150$.

There are a few versions of hillclimbing algorithms. They differ in the wa a new string is selected for comparison with the current string. One version c a simple (iterated) hillclimbing algorithm (MAX iterations) is given in Figur 1.2 (steepest ascent hillclimbing). Initially, all 30 neighbors are considered, an the one $\mathbf{v}_n$ which returns the largest value $f(\mathbf{v}_n)$ is selected to compete with th current string $\mathbf{v}_c$. If $f(\mathbf{v}_c) < f(\mathbf{v}_n)$, then the new string becomes the curren string. Otherwise, no local improvement is possible: the algorithm has reache (local or global) optimum ($local$ = TRUE). In a such case, the next iteratio ($t \leftarrow t + 1$) of the algorithm is executed with a new current string selected a random.

It is interesting to note that the success or failure of the single iteratio of the above hillclimber algorithm (i.e., return of the global or local optimum is determined by the starting string (randomly selected). It is clear that if th starting string has thirteen 1s or less, the algorithm will always terminate in th local optimum (failure). The reason is that a string with thirteen 1s returns value 7 of the objective function, and any single-step improvement towards th global optimum, i.e., increase the number of 1s to fourteen, decreases the valu of the objective function to 4. On the other hand, any decrease of the numbe of 1s would increase the value of the function: a string with twelve 1s yields value of 18, a string with eleven 1s yields a value of 29, etc. This would pus the search in the "wrong" direction, towards the local maximum.

For problems with many local optima, the chances of hitting the globa optimum (in a single iteration) are slim.

The structure of the simulated annealing procedure is given in Figure 1.3.