1. GAs: What Are They?

Z. Michalewicz, 1992
GA + DS = Evolution Programs

> Paradoxical as it seemed, the Master
> always insisted that the true reformer
> was one who was able to see that everything
> is perfect as it is — and able to
> leave it alone.
>
> Anthony de Mello, *One Minute Wisdom*

There is a large class of interesting problems for which no reasonably fast algorithms have been developed. Many of these problems are optimization problems that arise frequently in applications. Given such a hard optimization problem it is often possible to find an efficient algorithm whose solution is approximately optimal. For some hard optimization problems we can use probabilistic algorithms as well — these algorithms do not guarantee the optimum value, but by randomly choosing sufficiently many "witnesses" the probability of error may be made as small as we like.

There are a lot of important practical optimization problems for which such algorithms of high quality have become available [33]. For instance we can apply simulated annealing for wire routing and component placement problems in VLSI design or for the traveling salesman problem. Moreover, many other large-scale combinatorial optimization problems (many of which have been proved NP-hard) can be solved approximately on present-day computers by this kind of Monte Carlo technique.

In general, any abstract task to be accomplished can be thought of as solving a problem, which, in turn, can be perceived as a search through a space of potential solutions. Since we are after "the best" solution, we can view this task as an optimization process. For small spaces, classical exhaustive methods usually suffice; for larger spaces special artificial intelligence techniques must be employed. Genetic Algorithms (GAs) are among such techniques; they are stochastic algorithms whose search methods model some natural phenomena: genetic inheritance and Darwinian strife for survival. As stated in [34]:

> "... the metaphor underlying genetic algorithms is that of natural evolution. In evolution, the problem each species faces is one of

Good GA metaphor

> searching for beneficial adaptations to a complicated and changing environment. The 'knowledge' that each species has gained is embodied in the makeup of the chromosomes of its members."

The idea behind genetic algorithms is to do what nature does. Let us take rabbits as an example: at any given time there is a population of rabbits. Some of them are faster and smarter than other rabbits. These faster, smarter rabbits are less likely to be eaten by foxes, and therefore more of them survive to do what rabbits do best: make more rabbits. Of course, some of the slower, dumber rabbits will survive just because they are lucky. This surviving population of rabbits starts breeding. The breeding results in a good mixture of rabbit genetic material: some slow rabbits breed with fast rabbits, some fast with fast, some smart rabbits with dumb rabbits, and so on. And on the top of that, nature throws in a 'wild hare' every once in a while by mutating some of the rabbit genetic material. The resulting baby rabbits will (on average) be faster and smarter than these in the original population because more faster, smarter parents survived the foxes. (It is a good thing that the foxes are undergoing similar process — otherwise the rabbits might become too fast and smart for the foxes to catch any of them).

A genetic algorithm follows a step-by-step procedure that closely matches the story of the rabbits. Before we take a closer look at the structure of a genetic algorithm, let us have a quick look at the history of genetics (from [180]):

> "The fundamental principle of natural selection as the main evolutionary principle has been formulated by C. Darwin long before the discovery of genetic mechanisms. Ignorant of the basic heredity principles, Darwin hypothesized fusion or blending inheritance, supposing that parental qualities mix together like fluids in the offspring organism. His selection theory arose serious objections, first stated by F. Jenkins: crossing quickly levels off any hereditary distinctions, and there is no selection in homogeneous populations (the so-called 'Jenkins nightmare').
>
> It was not until 1865, when G. Mendel discovered the basic principles of transference of hereditary factors from parent to offspring, which showed the discrete nature of these factors, that the 'Jenkins nightmare' could be explained, since because of this discreteness there is no 'dissolution' of hereditary distinctions.
>
> Mendelian laws became known to the scientific community after they had been independently rediscovered in 1900 by H. de Vries, K. Correns and K. von Tschermak. Genetics was fully developed by T. Morgan and his collaborators, who proved experimentally that chromosomes are the main carriers of hereditary information and that genes, which present hereditary factors, are lined up on chromosomes. Later on, accumulated experimental facts showed Mendelian laws to be valid for all sexually reproducing organisms.

> However, Mendel's laws, even after they had been rediscovered, and Darwin's theory of natural selection remained independent, unlinked concepts. And moreover, they were opposed to each other. Not until the 1920s (see, for instance the classical work by Četverikov [26]) was it proved that Mendel's genetics and Darwin's theory of natural selection are in no way conflicting and that their happy marriage yields modern evolutionary theory."

Genetic algorithms use a vocabulary borrowed from natural genetics. We would talk about *individuals* (or *genotypes*, *structures*) in a population; quite often these individuals are called also *strings* or *chromosomes*. This might be a little bit misleading: each cell of every organism of a given species carries a certain number of chromosomes (man, for example, has 46 of them); however, in this book we talk about one-chromosome individuals only. (For additional information on *diploidy* — pairs of chromosomes — dominance, and other related issues, in connection with genetic algorithms, the reader is referred to [72].) Chromosomes are made of units — *genes* (also *features, characters, or decoders*) — arranged in linear succession; every gene controls the inheritance of one or several characters. Genes of certain characters are located at certain places of the chromosome, which are called *loci* (string positions). Any character of individuals (such as hair color) can manifest itself differently; the gene is said to be in several states, called *alleles* (feature values).

Each genotype (in this book a single chromosome) would represent a potential solution to a problem; an evolution process run on a population of chromosomes corresponds to a search through a space of potential solutions. Such a search requires balancing two (apparently conflicting) objectives: exploiting the best solutions and exploring the search space [21]. Hillclimbing is an example of a strategy which exploits the best solution for possible improvement; on the other hand, it neglects exploration of the search space. Random search is a typical example of a strategy which explores the search space ignoring the exploitations of the promising regions of the space. Genetic algorithms are a class of general purpose (domain independent) search methods which strike a remarkable balance between exploration and exploitation of the search space.

GAs have been quite successfully applied to optimization problems like wire routing, scheduling, adaptive control, game playing, cognitive modeling, transportation problems, traveling salesman problems, optimal control problems, database query optimization, etc. (see [15], [20], [43], [72], [78], [81], [82], [134], [132], [160], [189], [190]). However, De Jong [43] warned against perceiving GAs as optimization tools:

> "...because of this historical focus and emphasis on function optimization applications, it is easy to fall into the trap of perceiving GAs *themselves* as optimization algorithms and then being surprised and/or disappointed when they fail to find an 'obvious' optimum in a particular search space. My suggestion for avoiding this perceptual trap is to think of GAs as a (highly idealized) simulation of a natural

> process and as such they embody the goals and purposes (if any) of that natural process. I am not sure if anyone is up to the task of defining the goals and purpose of evolutionary systems; however, I think it's fair to say that such systems are *not* generally perceived as functions optimizers".

On the other hand, optimization is a major field of GA's applicability. In [162] *(1981) Schwefel said:*

> "There is scarcely a modern journal, whether of engineering, economics, management, mathematics, physics, or the social sciences, in which the concept 'optimization' is missing from the subject index. If one abstracts from all specialist points of view, the recurring problem is to select a better or best (according to Leibniz, optimal) alternative from among a number of possible states of affairs."

During the last decade, the significance of optimization has grown even further — many important large-scale combinatorial optimization problems and highly constrained engineering problems can only be solved approximately on present day computers.

Genetic algorithms aim at such complex problems. They belong to the class of probabilistic algorithms, yet they are very different from random algorithms as they combine elements of directed and stochastic search. Because of this, GA are also more robust than existing directed search methods. Another important property of such genetic based search methods is that they maintain a population of potential solutions — all other methods process a single point of the search space.

Hillclimbing methods use the iterative improvement technique; the technique is applied to a single point (the current point) in the search space. During a single iteration, a new point is selected from the neighborhood of the current point (this is why this technique is known also as neighborhood search or local search [112]). If the new point provides a better[1] value of the objective function, the new point becomes the current point. Otherwise, some other neighbor is selected and tested against the current point. The method terminates if no further improvement is possible.

It is clear that the hillclimbing methods provide local optimum values only and these values depend on the selection of the starting point. Moreover, there is no information available on the relative error (with respect to the global optimum) of the solution found.

To increase the chances to succeed, hillclimbing methods usually are executed for a (large) number of different starting points (these points need not be selected randomly — a selection of a starting point for a single execution may depend on the result of the previous runs).

The simulated annealing technique [1] eliminates most disadvantages of the hillclimbing methods: solutions do not depend on the starting point any longer

[1] smaller, for minimization, and larger, for maximization problems.

and are (usually) close to the optimum point. This is achieved by introducing a probability p of acceptance (i.e., replacement of the current point by a new point): $p = 1$, if the new point provides a better value of the objective function; however, $p > 0$, otherwise. In the latter case, the probability of acceptance p is a function of the values of objective function for the current point and the new point, and an additional control parameter, "temperature", T. In general, the lower temperature T is, the smaller the chances for the acceptance of a new point are. During execution of the algorithm, the temperature of the system, T, is lowered in steps. The algorithm terminates for some small value of T, for which virtually no changes are accepted anymore.

As mentioned earlier, a GA performs a multi-directional search by maintaining a population of potential solutions and encourages information formation and exchange between these directions. The population undergoes a simulated evolution: at each generation the relatively "good" solutions reproduce, while the relatively "bad" solutions die. To distinguish between different solutions we use an objective (evaluation) function which plays the role of an environment.

An example of hillclimbing, simulated annealing, and genetic algorithm techniques is given later in this chapter (Section 1.4).

The structure of a simple genetic algorithm is the same as the structure of any evolution program (see Figure 0.1, Introduction). During iteration t, a genetic algorithm maintains a population of potential solutions (chromosomes, vectors), $P(t) = \{x_1^t, \ldots, x_n^t\}$. Each solution x_i^t is evaluated to give some measure of its "fitness". Then, a new population (iteration $t + 1$) is formed by selecting the more fit individuals. Some members of this new population undergo reproduction by means of crossover and mutation, to form new solutions. Crossover combines the features of two parent chromosomes to form two similar offspring by swapping corresponding segments of the parents. For example, if the parents are represented by five-dimensional vectors $(a_1, b_1, c_1, d_1, e_1)$ and $(a_2, b_2, c_2, d_2, e_2)$, then crossing the chromosomes after the second gene would produce the offspring $(a_1, b_1, c_2, d_2, e_2)$ and $(a_2, b_2, c_1, d_1, e_1)$. The intuition behind the applicability of the crossover operator is information exchange between different potential solutions.

Mutation arbitrarily alters one or more genes of a selected chromosome, by a random change with a probability equal to the mutation rate. The intuition behind the mutation operator is the introduction of some extra variability into the population.

A genetic algorithm (as any evolution program) for a particular problem must have the following five components:

- a genetic representation for potential solutions to the problem,
- a way to create an initial population of potential solutions,
- an evaluation function that plays the role of the environment, rating solutions in terms of their "fitness",

- 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,

$$\begin{aligned} x_i &= \tfrac{2i-1}{20} + \epsilon_i, \text{ for } i = 1, 2, \ldots \\ x_0 &= 0 \\ x_i &= \tfrac{2i+1}{20} - \epsilon_i, \text{ for } i = -1, -2, \ldots, \end{aligned}$$

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.

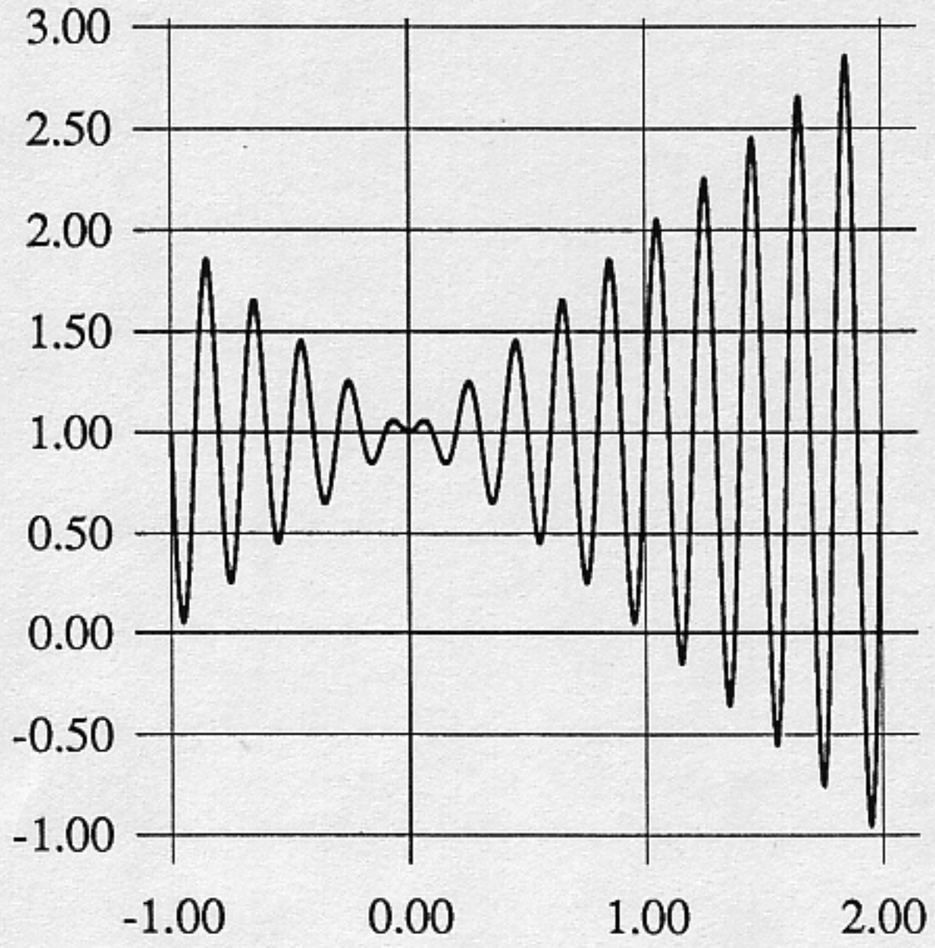

Fig. 1.1. Graph of the function $f(x) = x \cdot \sin(10\pi \cdot x) + 1.0$

1.1.1 Representation

May need to gradually increase the size of bit string, i.e. improve the precision.

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

The domain of the variable x has length 3; the precision requirement implies 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 from 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:

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

- 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 the 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

$$\mathbf{v_3}' = (1110\mathbf{1}0000011111000101).$$

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

$$\mathbf{v}_{max} = (11110011010001000001 01),$$

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

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

$$\mathbf{v}_1 = (11011010111010111111011011011),$$
$$\mathbf{v}_2 = (11100010010011011001010100011),$$
$$\mathbf{v}_3 = (00001000001100100000010001000),$$

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 optimization task. We use it only to illustrate the ideas behind these three algorithms. However, the interesting characteristic of the function f is that it has one global maximum for

$$\mathbf{v}_g = (11111111111111111111111111111),$$

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

$$\mathbf{v}_l = (00000000000000000000000000000),$$

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

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

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

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

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

```
procedure iterated hillclimber
begin
  t ← 0
  repeat
    local ← FALSE
    select a current string v_c at random
    evaluate v_c
    repeat
      select 30 new strings in the neighborhood of v_c
        by flipping single bits of v_c
      select the string v_n from the set of new strings
        with the largest value of objective function f
      if f(v_c) < f(v_n)
        then v_c ← v_n
        else local ← TRUE
    until local
    t ← t + 1
  until t = MAX
end
```

Fig. 1.2. A simple (iterated) hillclimber

The function $random[0,1)$ returns a random number from the range $[0,1)$. The (termination-condition) checks whether 'thermal equilibrium' is reached, i.e., whether the probability distribution of the selected new strings approaches the Boltzmann distribution [1]. However, in some implementations [2], this repeat loop is executed just k times (k is an additional parameter of the method).

The temperature T is lowered in steps ($g(T,t) < T$ for all t). The algorithm terminates for some small value of T: the (stop-criterion) checks whether the system is 'frozen', i.e., virtually no changes are accepted anymore.

As mentioned earlier, the simulated annealing algorithm can escape local optima. Let us consider a string

$$\mathbf{v}_4 = (11100000010011011001010100000),$$

with twelve 1s, which evaluates to $f(\mathbf{v}_4) = |11 \cdot 12 - 150| = 18$. For $\mathbf{v}_4$ as the starting string, the hillclimbing algorithm (as discussed earlier) would approach the local maximum

$$\mathbf{v}_l = (00000000000000000000000000000),$$

since any string with thirteen 1s (i.e., a step 'towards' the global optimum) evaluates to 7 (less than 18). On the other hand, the simulated annealing algorithm would accept a string with thirteen 1s as a new current string with probability

$$p = \exp\{(f(\mathbf{v}_n) - f(\mathbf{v}_c))/T\} = \exp\{(7-18)/T\},$$

```
procedure simulated annealing
begin
   t ← 0
   initialize temperature T
   select a current string v_c at random
   evaluate v_c
   repeat
      repeat
         select a new string v_n
            in the neighborhood of v_c
            by flipping a single bit of v_c
         if f(v_c) < f(v_n)
            then v_c ← v_n
            else if random[0,1) < exp{(f(v_n) − f(v_c))/T}
               then v_c ← v_n
      until (termination-condition)
      T ← g(T, t)
      t ← t + 1
   until (stop-criterion)
end
```

Fig. 1.3. Simulated annealing

which, for some temperature, say, $T = 20$, gives

$$p = e^{-\frac{11}{20}} = 0.57695,$$

i.e., the chances for acceptance are better than 50%.

Genetic algorithms, as discussed in Section 1.1, maintain a population of strings. Two relatively poor strings

$$\mathbf{v}_5 = (11111000000011011100111010000 0) \text{ and}$$
$$\mathbf{v}_6 = (00000000000110111001010111111 1)$$

each of which evaluate to 16, can produce much better offspring (if the crossover point falls anywhere between the 5th and the 12th position):

$$\mathbf{v}_7 = (11111000000110111001010111111 1).$$

The new offspring $\mathbf{v}_7$ evaluates to

$$f(\mathbf{v}_7) = |11 \cdot 19 - 150| = 59.$$

For a detailed discussion on these and other algorithms (various variants of hillclimbers, genetic search, and simulated annealing) tested on several functions with different characteristics, the reader is referred to [2].

1.5 Conclusions

The three examples of genetic algorithms for function optimization, the prisoner's dilemma, and the traveling salesman problem, show a wide applicability of genetic algorithms. However, at the same time we should observe first signs of potential difficulties. The representation issues for the traveling salesman problem were not obvious. The new operator used (OX crossover) was far from trivial. What kind of further difficulties may we have for some other (hard) problems? In the first and third examples (optimization of a function and the traveling salesman problem) the evaluation function was clearly defined; in the second example (the prisoner's dilemma) a simple simulation process would give us an evaluation of a chromosome (we test each player to determine its effectiveness: each player uses the strategy defined by its chromosome to play the game with other players and the player's score is its average over all the games it plays). How should we proceed in a case where the evaluation function is not clearly defined? For example, the Boolean Satisfiability Problem (SAT) seems to have a natural string representation (the i-th bit represents the truth value of the i-th Boolean variable), however, the process of choosing an evaluation function is far from obvious [46].

The first example of optimization of an unconstrained function allows us to use a convenient representation, where any binary string would correspond to a value from the domain of the problem (i.e., $[-1..2]$). This means that any mutation and any crossover would produce a legal offspring. The same was true in the second example: any combination of bits represents a legal strategy. The third problem has a single constraint: each city should appear precisely once in a legal tour. This caused some problems: we used vectors of integers (instead of binary representation) and we modified the crossover operator. But how should we approach a constrained problem in general? What possibilities do we have?

The answers are not easy; we explore these issues later in the book.

2. GAs: How Do They Work?

To every thing there is a season,
and a time to every purpose under the heaven:

A time to be born and a time to die;
a time to plant, and a time to pluck up
that which is planted;

A time to kill, and a time to heal;
a time to break down, and a time to build up.

The Bible, Ecclesiastes, 3

In this chapter we discuss the actions of a genetic algorithm for a simple parameter optimization problem. We start with a few general comments; a detailed example follows.

Let us note first that, without any loss of generality, we can assume maximization problems only. If the optimization problem is to minimize a function f, this is equivalent to maximizing a function g, where $g = -f$, i.e.,

$$\min f(x) = \max g(x) = \max\{-f(x)\}.$$

Moreover, we may assume that the objective function f takes positive values on its domain; otherwise we can add some positive constant C, i.e.,

$$\max g(x) = \max\{g(x) + C\}.$$

Now suppose we wish to maximize a function of k variables, $f(x_1, \ldots, x_k) : R^k \rightarrow R$. Suppose further that each variable x_i can take values from a domain $D_i = [a_i, b_i] \subseteq R$ and $f(x_1, \ldots, x_k) > 0$ for all $x_i \in D_i$. We wish to optimize the function f with some required precision: suppose six decimal places for the variables' values is desirable.

It is clear that to achieve such precision each domain D_i should be cut into $(b_i - a_i) \cdot 10^6$ equal size ranges. Let us denote by m_i the smallest integer such that $(b_i - a_i) \cdot 10^6 \leq 2^{m_i} - 1$. Then, a representation having each variable x_i coded as a binary string of length m_i clearly satisfies the precision requirement. Additionally, the following formula interprets each such string:

$$x_i = a_i + decimal(1001...001_2) \cdot \frac{b_i - a_i}{2^{m_i} - 1},$$

where $decimal(string_2)$ represents the decimal value of that binary string.

Now, each chromosome (as a potential solution) is represented by a binary string of length $m = \sum_{i=1}^{k} m_i$; the first m_1 bits map into a value from the range $[a_1, b_1]$, the next group of m_2 bits map into a value from the range $[a_2, b_2]$, and so on; the last group of m_k bits map into a value from the range $[a_k, b_k]$.

To initialize a population, we can simply set some *pop_size* number of chromosomes randomly in a bitwise fashion. However, if we do have some knowledge about the distribution of potential optima, we may use such information in arranging the set of initial (potential) solutions.

The rest of the algorithm is straightforward: in each generation we evaluate each chromosome (using the function f on the decoded sequences of variables), select new population with respect to the probability distribution based on fitness values, and recombine the chromosomes in the new population by mutation and crossover operators. After some number of generations, when no further improvement is observed, the best chromosome represents an (possibly the global) optimal solution. Often we stop the algorithm after a fixed number of iterations depending on speed and resource criteria.

For the selection process (selection of a new population with respect to the probability distribution based on fitness values), a roulette wheel with slots sized according to fitness is used. We construct such a roulette wheel as follows (we assume here that the fitness values are positive, otherwise, we can use some scaling mechanism — this is discussed in Chapter 4):

- Calculate the fitness value $eval(\mathbf{v}_i)$ for each chromosome $\mathbf{v}_i$ $(i = 1, \ldots, pop\_size)$.
- Find the total fitness of the population

$$F = \sum_{i=1}^{pop\_size} eval(\mathbf{v}_i).$$

- Calculate the probability of a selection p_i for each chromosome $\mathbf{v}_i$ $(i = 1, \ldots, pop\_size)$:

$$p_i = eval(\mathbf{v}_i)/F.$$

- Calculate a cumulative probability q_i for each chromosome $\mathbf{v}_i$ $(i = 1, \ldots, pop\_size)$:

$$q_i = \sum_{j=1}^{i} p_j.$$

The selection process is based on spinning the roulette wheel *pop_size* times; each time we select a single chromosome for a new population in the following way:

- Generate a random (float) number r from the range [0..1].
- If $r < q_1$ then select the first chromosome $(\mathbf{v}_1)$; otherwise select the i-th chromosome $\mathbf{v}_i$ $(2 \leq i \leq pop\_size)$ such that $q_{i-1} < r \leq q_i$.

Obviously, some chromosomes would be selected more than once. This is in accordance with the Schema Theorem (see next chapter): the best chromosomes get more copies, the average stay even, and the worst die off.

Now we are ready to apply the first recombination operator, crossover, to the individuals in the new population. As mentioned earlier, one of the parameters of a genetic system is probability of crossover p_c. This probability gives us the expected number $p_c \cdot pop\_size$ of chromosomes which undergo the crossover operation. We proceed in the following way:

For each chromosome in the (new) population:

- Generate a random (float) number r from the range [0..1];
- If $r < p_c$, select given chromosome for crossover.

Now we mate selected chromosomes randomly: for each pair of coupled chromosomes we generate a random integer number pos from the range $[1..m-1]$ (m is the total length — number of bits — in a chromosome). The number pos indicates the position of the crossing point. Two chromosomes

$(b_1 b_2 \ldots b_{pos} b_{pos+1} \ldots b_m)$ and
$(c_1 c_2 \ldots c_{pos} c_{pos+1} \ldots c_m)$

are replaced by a pair of their offspring:

$(b_1 b_2 \ldots b_{pos} c_{pos+1} \ldots c_m)$ and
$(c_1 c_2 \ldots c_{pos} b_{pos+1} \ldots b_m)$.

The next recombination operator, mutation, is performed on a bit-by-bit basis. Another parameter of the genetic system, probability of mutation p_m, gives us the expected number of mutated bits $p_m \cdot m \cdot pop\_size$. Every bit (in all chromosomes in the whole population) has an equal chance to undergo mutation, i.e., change from 0 to 1 or vice versa. So we proceed in the following way.

For each chromosome in the current (i.e., after crossover) population and for each bit within the chromosome:

- Generate a random (float) number r from the range [0..1];
- If $r < p_m$, mutate the bit.

Following selection, crossover, and mutation, the new population is ready for its next evaluation. This evaluation is used to build the probability distribution (for the next selection process), i.e., for a construction of a roulette wheel with slots sized according to current fitness values. The rest of the evolution is just cyclic repetition of the above steps (see Figure 0.1 in the introduction).

The whole process is illustrated by an example. We run a simulation of a genetic algorithm for function optimization. We assume that the population size $pop\_size = 20$, and the probabilities of genetic operators are $p_c = 0.25$ and $p_m = 0.01$.

Let us assume also that we maximize the following function:

$$f(x_1, x_2) = 21.5 + x_1 \cdot \sin(4\pi x_1) + x_2 \cdot \sin(20\pi x_2),$$

where $-3.0 \leq x_1 \leq 12.1$ and $4.1 \leq x_2 \leq 5.8$. The graph of the function f is given in Figure 2.1.

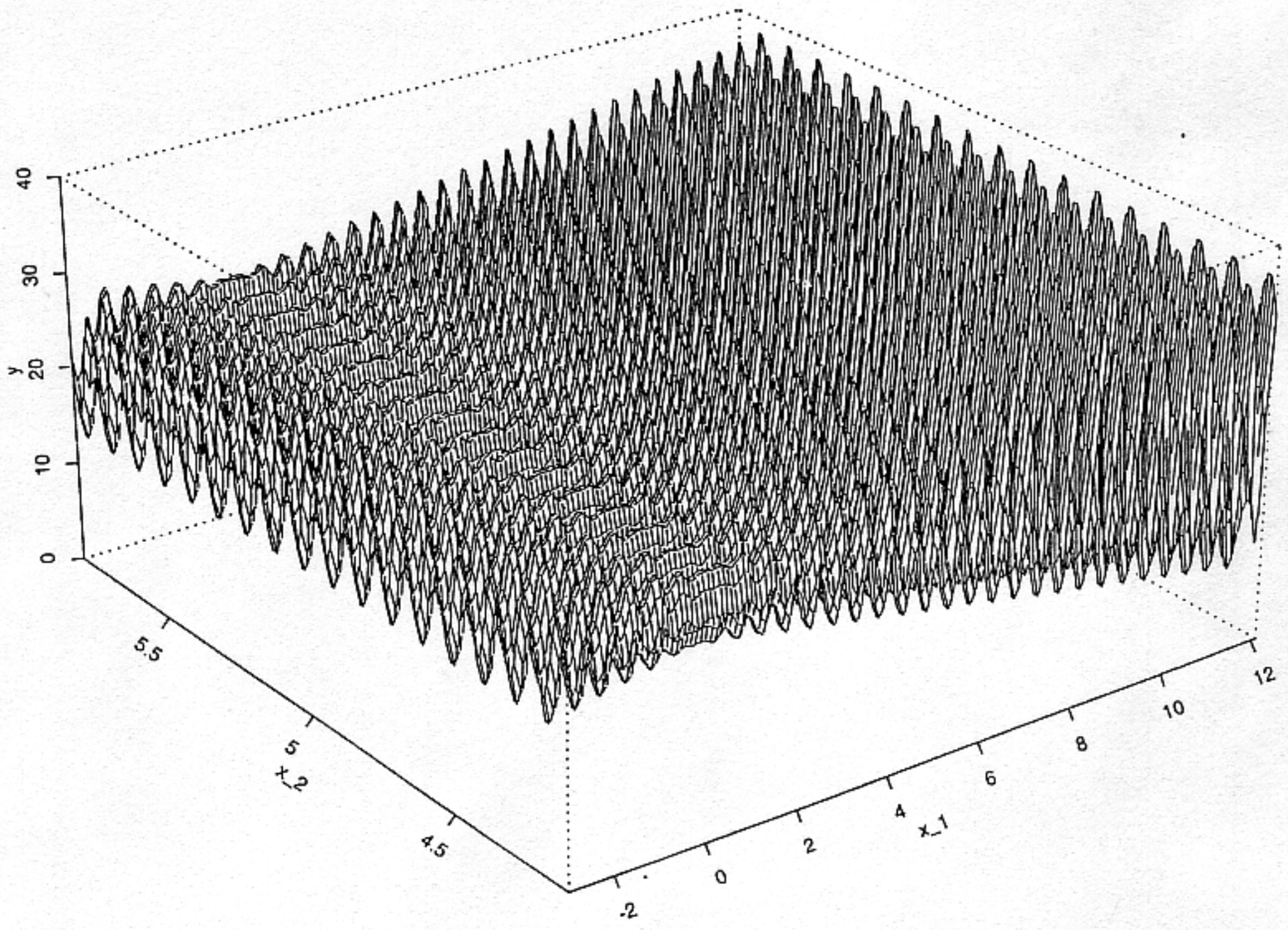

Fig. 2.1. Graph of the function $f(x_1, x_2) = 21.5 + x_1 \cdot \sin(4\pi x_1) + x_2 \cdot \sin(20\pi x_2)$

Let assume further that the required precision is four decimal places for each variable. The domain of variable x_1 has length 15.1; the precision requirement implies that the range $[-3.0, 12.1]$ should be divided into at least $15.1 \cdot 10000$ equal size ranges. This means that 18 bits are required as the first part of the chromosome:

$$2^{17} < 151000 \leq 2^{18}.$$

The domain of variable x_2 has length 1.7; the precision requirement implies that the range $[4.1, 5.8]$ should be divided into at least $1.7 \cdot 10000$ equal size ranges. This means that 15 bits are required as the second part of the chromosome:

$$2^{14} < 17000 \leq 2^{15}.$$

The total length of a chromosome (solution vector) is then $m = 18+15 = 33$ bits; the first 18 bits code x_1 and remaining 15 bits (19–33) code x_2.
Let us consider an example chromosome:

(010001001011010000111110010100010).

The first 18 bits,

010001001011010000,

represent $x_1 = -3.0 + decimal(010001001011010000_2) \cdot \frac{12.1-(-3.0)}{2^{18}-1} = -3.0 + 70352 \cdot \frac{15.1}{262143} = -3.0 + 4.052426 = 1.052426$.
The next 15 bits,

111110010100010,

represent $x_2 = 4.1 + decimal(111110010100010_2) \cdot \frac{5.8-4.1}{2^{15}-1} = 4.1 + 31906 \cdot \frac{1.7}{32767} = 4.1 + 1.655330 = 5.755330$.
So the chromosome

(010001001011010000111110010100010)

corresponds to $\langle x_1, x_2 \rangle = \langle 1.052426, 5.755330 \rangle$. The fitness value for this chromosome is

$$f(1.052426, 5.755330) = 20.252640.$$

To optimize the function f using a genetic algorithm, we create a population of $pop\_size = 20$ chromosomes. All 33 bits in all chromosomes are initialized randomly.

Assume that after the initialization process we get the following population:

$\mathbf{v}_1 = (100110100000001111111010011011111)$
$\mathbf{v}_2 = (111000100100110111001010100011010)$
$\mathbf{v}_3 = (000010000011001000001010111011101)$
$\mathbf{v}_4 = (100011000101101001111000001110010)$
$\mathbf{v}_5 = (000111011001010011010111111000101)$
$\mathbf{v}_6 = (000101000010010101001010111111011)$
$\mathbf{v}_7 = (001000100000110101111011011111011)$
$\mathbf{v}_8 = (100001100001110100010110101100111)$
$\mathbf{v}_9 = (010000000101100010110000001111100)$
$\mathbf{v}_{10} = (000001111000110000011010000111011)$
$\mathbf{v}_{11} = (011001111110110101100001101111000)$
$\mathbf{v}_{12} = (110100010111101101000101010000000)$
$\mathbf{v}_{13} = (111011111010001000110000001000110)$
$\mathbf{v}_{14} = (010010011000001010100111100101001)$
$\mathbf{v}_{15} = (111011101101110000100011111011110)$
$\mathbf{v}_{16} = (110011110000011111100001101001011)$
$\mathbf{v}_{17} = (011010111111001111010001101111101)$
$\mathbf{v}_{18} = (011101000000001110100111110101101)$
$\mathbf{v}_{19} = (000101010011111111110000110001100)$
$\mathbf{v}_{20} = (101110010110011110011000101111110)$

During the evaluation phase we decode each chromosome and calculate the fitness function values from (x_1, x_2) values just decoded. We get:

$$\begin{aligned}
eval(\mathbf{v}_1) &= f(6.084492, 5.652242) = 26.019600\\
eval(\mathbf{v}_2) &= f(10.348434, 4.380264) = 7.580015\\
eval(\mathbf{v}_3) &= f(-2.516603, 4.390381) = 19.526329\\
eval(\mathbf{v}_4) &= f(5.278638, 5.593460) = 17.406725\\
eval(\mathbf{v}_5) &= f(-1.255173, 4.734458) = 25.341160\\
eval(\mathbf{v}_6) &= f(-1.811725, 4.391937) = 18.100417\\
eval(\mathbf{v}_7) &= f(-0.991471, 5.680258) = 16.020812\\
eval(\mathbf{v}_8) &= f(4.910618, 4.703018) = 17.959701\\
eval(\mathbf{v}_9) &= f(0.795406, 5.381472) = 16.127799\\
eval(\mathbf{v}_{10}) &= f(-2.554851, 4.793707) = 21.278435\\
eval(\mathbf{v}_{11}) &= f(3.130078, 4.996097) = 23.410669\\
eval(\mathbf{v}_{12}) &= f(9.356179, 4.239457) = 15.011619\\
eval(\mathbf{v}_{13}) &= f(11.134646, 5.378671) = 27.316702\\
eval(\mathbf{v}_{14}) &= f(1.335944, 5.151378) = 19.876294\\
eval(\mathbf{v}_{15}) &= f(11.089025, 5.054515) = 30.060205\\
eval(\mathbf{v}_{16}) &= f(9.211598, 4.993762) = 23.867227\\
eval(\mathbf{v}_{17}) &= f(3.367514, 4.571343) = 13.696165\\
eval(\mathbf{v}_{18}) &= f(3.843020, 5.158226) = 15.414128\\
eval(\mathbf{v}_{19}) &= f(-1.746635, 5.395584) = 20.095903\\
eval(\mathbf{v}_{20}) &= f(7.935998, 4.757338) = 13.666916
\end{aligned}$$

It is clear, that the chromosome $\mathbf{v}_{15}$ is the strongest one, and the chromosome $\mathbf{v}_2$ the weakest.

Now the system constructs a roulette wheel for the selection process. The total fitness of the population is

$$F = \sum_{i=1}^{20} eval(\mathbf{v}_i) = 387.776822.$$

The probability of a selection p_i for each chromosome $\mathbf{v}_i$ $(i = 1, \ldots, 20)$ is:

$$\begin{aligned}
p_1 &= eval(\mathbf{v}_1)/F = 0.067099 & p_2 &= eval(\mathbf{v}_2)/F = 0.019547\\
p_3 &= eval(\mathbf{v}_3)/F = 0.050355 & p_4 &= eval(\mathbf{v}_4)/F = 0.044889\\
p_5 &= eval(\mathbf{v}_5)/F = 0.065350 & p_6 &= eval(\mathbf{v}_6)/F = 0.046677\\
p_7 &= eval(\mathbf{v}_7)/F = 0.041315 & p_8 &= eval(\mathbf{v}_8)/F = 0.046315\\
p_9 &= eval(\mathbf{v}_9)/F = 0.041590 & p_{10} &= eval(\mathbf{v}_{10})/F = 0.054873\\
p_{11} &= eval(\mathbf{v}_{11})/F = 0.060372 & p_{12} &= eval(\mathbf{v}_{12})/F = 0.038712\\
p_{13} &= eval(\mathbf{v}_{13})/F = 0.070444 & p_{14} &= eval(\mathbf{v}_{14})/F = 0.051257\\
p_{15} &= eval(\mathbf{v}_{15})/F = 0.077519 & p_{16} &= eval(\mathbf{v}_{16})/F = 0.061549\\
p_{17} &= eval(\mathbf{v}_{17})/F = 0.035320 & p_{18} &= eval(\mathbf{v}_{18})/F = 0.039750\\
p_{19} &= eval(\mathbf{v}_{19})/F = 0.051823 & p_{20} &= eval(\mathbf{v}_{20})/F = 0.035244
\end{aligned}$$

The cumulative probabilities q_i for each chromosome $\mathbf{v}_i$ $(i = 1, \ldots, 20)$ are:

$$
\begin{array}{llll}
q_1 = 0.067099 & q_2 = 0.086647 & q_3 = 0.137001 & q_4 = 0.181890 \\
q_5 = 0.247240 & q_6 = 0.293917 & q_7 = 0.335232 & q_8 = 0.381546 \\
q_9 = 0.423137 & q_{10} = 0.478009 & q_{11} = 0.538381 & q_{12} = 0.577093 \\
q_{13} = 0.647537 & q_{14} = 0.698794 & q_{15} = 0.776314 & q_{16} = 0.837863 \\
q_{17} = 0.873182 & q_{18} = 0.912932 & q_{19} = 0.964756 & q_{20} = 1.000000
\end{array}
$$

Now we are ready to spin the roulette wheel 20 times; each time we select a single chromosome for a new population. Let us assume that a (random) sequence of 20 numbers from the range [0..1] is:

0.513870	0.175741	0.308652	0.534534	0.947628
0.171736	0.702231	0.226431	0.494773	0.424720
0.703899	0.389647	0.277226	0.368071	0.983437
0.005398	0.765682	0.646473	0.767139	0.780237

The first number $r = 0.513870$ is greater than q_{10} and smaller than q_{11}, meaning the chromosome $\mathbf{v}_{11}$ is selected for the new population; the second number $r = 0.175741$ is greater than q_3 and smaller than q_4, meaning the chromosome $\mathbf{v}_4$ is selected for the new population, etc.

Finally, the new population consists of the following chromosomes:

After reproduction.

$$
\begin{aligned}
\mathbf{v}'_1 &= (01100111111011010110000110111100)\ (\mathbf{v}_{11}) \\
\mathbf{v}'_2 &= (10001100010110100111100000111001 0)\ (\mathbf{v}_4) \\
\mathbf{v}'_3 &= (00100010000011010111101101111101 1)\ (\mathbf{v}_7) \\
\mathbf{v}'_4 &= (01100111111011010110000110111100 0)\ (\mathbf{v}_{11}) \\
\mathbf{v}'_5 &= (00010101001111111111000011000110 0)\ (\mathbf{v}_{19}) \\
\mathbf{v}'_6 &= (10001100010110100111100000111001 0)\ (\mathbf{v}_4) \\
\mathbf{v}'_7 &= (11101110110111000010001111101111 0)\ (\mathbf{v}_{15}) \\
\mathbf{v}'_8 &= (00011101100101001101011111100010 1)\ (\mathbf{v}_5) \\
\mathbf{v}'_9 &= (01100111111011010110000110111100 0)\ (\mathbf{v}_{11}) \\
\mathbf{v}'_{10} &= (00001000001100100000101011101110 1)\ (\mathbf{v}_3) \\
\mathbf{v}'_{11} &= (11101110110111000010001111101111 0)\ (\mathbf{v}_{15}) \\
\mathbf{v}'_{12} &= (01000000101100010110000001111001 0)\ (\mathbf{v}_9) \\
\mathbf{v}'_{13} &= (00010100001001010100101011111101 1)\ (\mathbf{v}_6) \\
\mathbf{v}'_{14} &= (10000110000111010001011010110011 1)\ (\mathbf{v}_8) \\
\mathbf{v}'_{15} &= (10111001011001111001100010111111 0)\ (\mathbf{v}_{20}) \\
\mathbf{v}'_{16} &= (10011010000000011111110100110111 1)\ (\mathbf{v}_1) \\
\mathbf{v}'_{17} &= (00000111100011000001101000011101 1)\ (\mathbf{v}_{10}) \\
\mathbf{v}'_{18} &= (11101111010001000110000001000110)\ (\mathbf{v}_{13}) \\
\mathbf{v}'_{19} &= (11101110110111000010001111101111 0)\ (\mathbf{v}_{15}) \\
\mathbf{v}'_{20} &= (11001111000001111110000110100101 1)\ (\mathbf{v}_{16})
\end{aligned}
$$

Now we are ready to apply the first recombination operator, crossover, to the individuals in the new population (vectors $\mathbf{v}'_i$). The probability of crossover $p_c = 0.25$, so we expect that (on average) 25% of chromosomes (i.e., 5 out of 20) undergo crossover. We proceed in the following way: for each chromosome

in the (new) population we generate a random number r from the range [0..1]; if $r < 0.25$, we select a given chromosome for crossover.

Let us assume that the sequence of random numbers is:

0.822951	0.151932	0.625477	0.314685	0.346901
0.917204	0.519760	0.401154	0.606758	0.785402
0.031523	0.869921	0.166525	0.674520	0.758400
0.581893	0.389248	0.200232	0.355635	0.826927

This means that the chromosomes $\mathbf{v}'_2$, $\mathbf{v}'_{11}$, $\mathbf{v}'_{13}$, and $\mathbf{v}'_{18}$ were selected for crossover. (We were lucky: the number of selected chromosomes is even, so we can pair them easily. If the number of selected chromosomes were odd, we would either add one extra chromosome or remove one selected chromosome — this choice is made randomly as well.) Now we mate selected chromosomes randomly: say, the first two (i.e., $\mathbf{v}'_2$ and $\mathbf{v}'_{11}$) and the next two (i.e., $\mathbf{v}'_{13}$ and $\mathbf{v}'_{18}$) are coupled together. For each of these two pairs, we generate a random integer number *pos* from the range [1..32] (33 is the total length — number of bits — in a chromosome). The number *pos* indicates the position of the crossing point. The first pair of chromosomes is

$$\mathbf{v}'_2 = (100011000|101101001111000001110010)$$
$$\mathbf{v}'_{11} = (111011101|101110000100011111011110)$$

and the generated number $pos = 9$. These chromosomes are cut after the 9th bit and replaced by a pair of their offspring:

$$\mathbf{v}''_2 = (100011000|101110000100011111011110)$$
$$\mathbf{v}''_{11} = (111011101|101101001111000001110010).$$

The second pair of chromosomes is

$$\mathbf{v}'_{13} = (000101000010010101 00|1010111111011)$$
$$\mathbf{v}'_{18} = (111011111010001000 11|0000001000110)$$

and the generated number $pos = 20$. These chromosomes are replaced by a pair of their offspring:

$$\mathbf{v}''_{13} = (00010100001001010100|0000001000110)$$
$$\mathbf{v}''_{18} = (11101111101000100011|1010111111011).$$

The current version of the population is:

$$\mathbf{v}'_1 = (01100111111011010110000110111 1000)$$ After crossover
$$\mathbf{v}''_2 = (10001100010111000010001111011110)$$
$$\mathbf{v}'_3 = (00100010000011010111101101111 1011)$$
$$\mathbf{v}'_4 = (01100111111011010110000110111 1000)$$
$$\mathbf{v}'_5 = (00010101001111111111000011000 1100)$$
$$\mathbf{v}'_6 = (10001100010110100111100000111 0010)$$
$$\mathbf{v}'_7 = (11101110110111000010001111011110)$$

$\mathbf{v}'_8 = (00011101100101001101011111000101)$
$\mathbf{v}'_9 = (01100111111011010110000110111000)$
$\mathbf{v}'_{10} = (00001000001100100000101011011101)$
$\mathbf{v}''_{11} = (11101110110110100111100000111001 0)$
$\mathbf{v}'_{12} = (01000000010110001011000000111100)$
$\mathbf{v}''_{13} = (00010100001001010100000001000110)$
$\mathbf{v}'_{14} = (10000110000111010001011010110011 1)$
$\mathbf{v}'_{15} = (10111001011001111001100010111111 0)$
$\mathbf{v}'_{16} = (10011010000000111111101001101111 1)$
$\mathbf{v}'_{17} = (00000111100011000001101000011101 1)$
$\mathbf{v}''_{18} = (11101111010001000111010111111011)$
$\mathbf{v}'_{19} = (11101110110111000010001111101111 0)$
$\mathbf{v}'_{20} = (11001111000001111110000110100101 1)$

The next recombination operator, mutation, is performed on a bit-by-bit basis. The probability of mutation $p_m = 0.01$, so we expect that (on average) 1% of bits would undergo mutation. There are $m \times pop\_size = 33 \times 20 = 660$ bits in the whole population; we expect (on average) 6.6 mutations per generation. Every bit has an equal chance to be mutated, so, for every bit in the population, we generate a random number r from the range [0..1]; if $r < 0.01$, we mutate the bit.

This means that we have to generate 660 random numbers. In a sample run, 5 of these numbers were smaller than 0.01; the bit number and the random number are listed below:

Bit position	Random number
112	0.000213
349	0.009945
418	0.008809
429	0.005425
602	0.002836

The following table translates the bit position into chromosome number and the bit number within the chromosome:

Bit position	Chromosome number	Bit number within chromosome
112	4	13
349	11	19
418	13	22
429	13	33
602	19	8

This means that four chromosomes are affected by the mutation operator; one of the chromosomes (the 13th) has two bits changed.

The final population is listed below; the mutated bits are typed in boldface. We drop *primes* for modified chromosomes: the population is listed as new vectors $\mathbf{v}_i$:

After mutation.

$$\begin{aligned}
\mathbf{v}_1 &= (01100111111011010110000110111 1000)\\
\mathbf{v}_2 &= (10001100010111000010001111011110)\\
\mathbf{v}_3 &= (00100010000011010111011011111011)\\
\mathbf{v}_4 &= (01100111111\mathbf{0}10101100001101111000)\\
\mathbf{v}_5 &= (00010101001111111110000110001100)\\
\mathbf{v}_6 &= (10001100010110100111100000111 0010)\\
\mathbf{v}_7 &= (11101110110111000010001111011110)\\
\mathbf{v}_8 &= (00011101100101001101011111 1000101)\\
\mathbf{v}_9 &= (01100111111011010110000110111 1000)\\
\mathbf{v}_{10} &= (00001000001100100000101011101 1101)\\
\mathbf{v}_{11} &= (11101110110110100\mathbf{0}11000001110010)\\
\mathbf{v}_{12} &= (01000000010110001011000000111 1100)\\
\mathbf{v}_{13} &= (00010100001001010100\mathbf{1}00001000 11\mathbf{1})\\
\mathbf{v}_{14} &= (10000110000111010001011010 1100111)\\
\mathbf{v}_{15} &= (10111001011001111001100010 1111110)\\
\mathbf{v}_{16} &= (10011010000000111111010011011111)\\
\mathbf{v}_{17} &= (00000111100011000001101000 0111011)\\
\mathbf{v}_{18} &= (11101111010001000111010111 1111011)\\
\mathbf{v}_{19} &= (11101110\mathbf{0}10111000010001111011110)\\
\mathbf{v}_{20} &= (11001111000001111110000110 1001011)
\end{aligned}$$

We have just completed one iteration (i.e., one generation) of the **while** loop in the genetic procedure (Figure 0.1 from the introduction). It is interesting to examine the results of the evaluation process of the new population. During the evaluation phase we decode each chromosome and calculate the fitness function values from (x_1, x_2) values just decoded. We get:

$$\begin{aligned}
eval(\mathbf{v}_1) &= f(3.130078, 4.996097) = 23.410669\\
eval(\mathbf{v}_2) &= f(5.279042, 5.054515) = 18.201083\\
eval(\mathbf{v}_3) &= f(-0.991471, 5.680258) = 16.020812\\
eval(\mathbf{v}_4) &= f(3.128235, 4.996097) = 23.412613\\
eval(\mathbf{v}_5) &= f(-1.746635, 5.395584) = 20.095903\\
eval(\mathbf{v}_6) &= f(5.278638, 5.593460) = 17.406725\\
eval(\mathbf{v}_7) &= f(11.089025, 5.054515) = 30.060205\\
eval(\mathbf{v}_8) &= f(-1.255173, 4.734458) = 25.341160\\
eval(\mathbf{v}_9) &= f(3.130078, 4.996097) = 23.410669\\
eval(\mathbf{v}_{10}) &= f(-2.516603, 4.390381) = 19.526329\\
eval(\mathbf{v}_{11}) &= f(11.088621, 4.743434) = 33.351874\\
eval(\mathbf{v}_{12}) &= f(0.795406, 5.381472) = 16.127799\\
eval(\mathbf{v}_{13}) &= f(-1.811725, 4.209937) = 22.692462\\
eval(\mathbf{v}_{14}) &= f(4.910618, 4.703018) = 17.959701\\
eval(\mathbf{v}_{15}) &= f(7.935998, 4.757338) = 13.666916\\
eval(\mathbf{v}_{16}) &= f(6.084492, 5.652242) = 26.019600
\end{aligned}$$

$$eval(\mathbf{v}_{17}) = f(-2.554851, 4.793707) = 21.278435$$
$$eval(\mathbf{v}_{18}) = f(11.134646, 5.666976) = 27.591064$$
$$eval(\mathbf{v}_{19}) = f(11.059532, 5.054515) = 27.608441$$
$$eval(\mathbf{v}_{20}) = f(9.211598, 4.993762) = 23.867227$$

Note that the total fitness of the new population F is 447.049688, much higher than total fitness of the previous population, 387.776822. Also, the best chromosome now ($\mathbf{v}_{11}$) has a better evaluation (33.351874) than the best chromosome ($\mathbf{v}_{15}$) from the previous population (30.060205).

Now we are ready to run the selection process again and apply the genetic operators, evaluate the next generation, etc. After 1000 generations the population is:

$$\mathbf{v}_1 = (11101111011001101110010101011011)$$
$$\mathbf{v}_2 = (11100110011000010001010101011000)$$
$$\mathbf{v}_3 = (11101111011101101110010101011011)$$
$$\mathbf{v}_4 = (11100110001000011000010101011001)$$
$$\mathbf{v}_5 = (11101111011101101110010101011011)$$
$$\mathbf{v}_6 = (11100110011000010000010010100001)$$
$$\mathbf{v}_7 = (11010110001001001000110010110000)$$
$$\mathbf{v}_8 = (11110110001000101000110101001001)$$
$$\mathbf{v}_9 = (11100110001001001000110010110001)$$
$$\mathbf{v}_{10} = (11101111011101101110010101011011)$$
$$\mathbf{v}_{11} = (11010110000001001000110010110000)$$
$$\mathbf{v}_{12} = (11010110001001001000110010110001)$$
$$\mathbf{v}_{13} = (11101111011101101110010101011011)$$
$$\mathbf{v}_{14} = (11100110011000010000010101011011)$$
$$\mathbf{v}_{15} = (11100110101011100101010110110001)$$
$$\mathbf{v}_{16} = (11100110011000010100010010100001)$$
$$\mathbf{v}_{17} = (11100110011000010000010101011011)$$
$$\mathbf{v}_{18} = (11100110011000010000010101011001)$$
$$\mathbf{v}_{19} = (11110110001000101000111000010001)$$
$$\mathbf{v}_{20} = (11100110011000010000010101011001)$$

The fitness values are:

$$eval(\mathbf{v}_1) = f(11.120940, 5.092514) = 30.298543$$
$$eval(\mathbf{v}_2) = f(10.588756, 4.667358) = 26.869724$$
$$eval(\mathbf{v}_3) = f(11.124627, 5.092514) = 30.316575$$
$$eval(\mathbf{v}_4) = f(10.574125, 4.242410) = 31.933120$$
$$eval(\mathbf{v}_5) = f(11.124627, 5.092514) = 30.316575$$
$$eval(\mathbf{v}_6) = f(10.588756, 4.214603) = 34.356125$$
$$eval(\mathbf{v}_7) = f(9.631066, 4.427881) = 35.458636$$
$$eval(\mathbf{v}_8) = f(11.518106, 4.452835) = 23.309078$$
$$eval(\mathbf{v}_9) = f(10.574816, 4.427933) = 34.393820$$
$$eval(\mathbf{v}_{10}) = f(11.124627, 5.092514) = 30.316575$$
$$eval(\mathbf{v}_{11}) = f(9.623693, 4.427881) = 35.477938$$

$$
\begin{aligned}
eval(\mathbf{v}_{12}) &= f(9.631066, 4.427933) = 35.456066 \\
eval(\mathbf{v}_{13}) &= f(11.124627, 5.092514) = 30.316575 \\
eval(\mathbf{v}_{14}) &= f(10.588756, 4.242514) = 32.932098 \\
eval(\mathbf{v}_{15}) &= f(10.606555, 4.653714) = 30.746768 \\
eval(\mathbf{v}_{16}) &= f(10.588814, 4.214603) = 34.359545 \\
eval(\mathbf{v}_{17}) &= f(10.588756, 4.242514) = 32.932098 \\
eval(\mathbf{v}_{18}) &= f(10.588756, 4.242410) = 32.956664 \\
eval(\mathbf{v}_{19}) &= f(11.518106, 4.472757) = 19.669670 \\
eval(\mathbf{v}_{20}) &= f(10.588756, 4.242410) = 32.956664
\end{aligned}
$$

However, if we look carefully at the progress during the run, we may discover that in earlier generations the fitness values of some chromosomes were better than the value 35.477938 of the best chromosome after 1000 generations. For example, the best chromosome in generation 396 had value of 38.827553. This is due to the stochastic errors of sampling — we discuss this issue in Chapter 4.

It is relatively easy to keep track of the best individual in the evolution process. It is customary (in genetic algorithm implementations) to store "the best ever" individual at a separate location; in that way, the algorithm would report the best value found during the whole process (as opposed to the best value in the final population).