Table 11.1 Basic evolutionary notions in biology and computer science

Notion	Biology	Computer science
Individual	Living organism	Solution candidate
Chromosome	DNA histone protein strand	Sequence of computational objects
	describes the "construction plan" and thus (some of the) traits of an individual in encoded form	
	usually multiple chromosomes per individual	usually only one chromosome per individual
Gene	Part of a chromosome	Computational object (e.g., a bit, character, number etc.)
	is the fundamental unit of inheritance, which determines a (partial) characteristic of an individual	
Allele (allelomorph)	Form or "value" of gene	Value of a computational object
	in each chromosome there is at most one form/value of a gene	
Locus	Position of a gene	Position of a computational object
	at each position in a chromosome there is exactly one gene	
Phenotype	Physical appearance of a living organism	Implementation/application of a solution candidate
Genotype	Genetic constitution of a living organism	Encoding of a solution candidate
Population	Set of living organisms	Bag/multiset of chromosomes
Generation	Population at a point in time	Population at a point in time
Reproduction	Creating offspring of one or multiple (usually two) (parent) organisms	creating (child) chromosomes from one or multiple (parent) chromosomes
Fitness	Aptitude/conformity of a living organism	Aptitude/quality of a solution candidate
	determines chances of survival and reproduction	

An **individual**, which is a living organism in biology, corresponds to a candidate solution in computer science. Individuals are the entities to which a fitness is assigned and which are subject to the (natural) selection process. In both domains, an individual is described by a **chromosome** (from the Greek $\chi\rho\omega\mu\alpha$: color and $\sigma\omega\mu\alpha$: body, thus "colored body," because they are the colorable substance in a cell nucleus), which is the carrier of the genetic information. In biology, a chromosome

consists of deoxyribonucleic acid (DNA) and many histone proteins, while in computer science the genetic information is encoded as a sequence of computational objects like bits, characters, numbers, etc. A chromosome represents the "genetic blueprint" and encodes (parts of) traits of an individual. Most living organisms have several chromosomes, for example, humans have 46 chromosomes, which come in 23 so-called *homologous* pairs. In computer science, however, this complication is ignored and all genetic information is combined in a single chromosome.

A **gene** is the fundamental unit of inheritance as it determines (a part of) a trait or characteristic of an individual. An **allele** (from the Greek $\alpha\lambda\lambda\eta\lambda\omega\nu$: "each other," "mutual," because initially mainly two-valued genes were considered) refers to a possible form of a gene in biology. For example, a gene may represent the color of the iris in the eye of a human. This gene has alleles that code for blue, brown, green, gray, etc., irises. In computer science, an allele is simply the value of a computational object, which selects one of several possible properties of a solution candidate that the gene stands for. Note that in a given chromosome there is exactly one allele per gene. That is, the iris color gene may code for blue or brown or green or gray eyes, but only one of these possibilities, as specified by the corresponding allele, is present in a concrete chromosome. The **locus** is the position of a gene in its chromosome. At any locus in a chromosome there is exactly one gene. Usually a gene can be identified by its locus. That is, a specific position (or a specific section) of a chromosome codes for a specific trait of the individual.

In biology, **phenotype** refers to the physical appearance of an organism, that is, the shape, structure and organization of its body. Note that the phenotype is what interacts with the environment and hence that it is the phenotype that actually determines the fitness of the individual. Likewise, in computer science, the phenotype is the implementation or application of a candidate solution, from which the fitness of the corresponding individual can be read. In contrast to this, the **genotype** is the genetic configuration of an organism or the encoding of a candidate solution, respectively. Note that the genotype determines the fitness of an individual only indirectly through the phenotype it encodes and that, at least in biology, the phenotype also comprises acquired traits that are not represented in the genotype (for example, learned behavior and bodily changes like a limb lost due to an accident).

A **population** is a simple set of organisms, usually of the same species. Due to the complexity of biological genomes it is usually safe to assume that no two individuals from a population share exactly the same genetic configuration—homozygous twins being the only exception. In addition, even genetically identical individuals differ due to acquired traits, which are never perfectly identical (even for homozygous twins) and thus lead to different phenotypes. In computer science, however, due to the usually much more limited variability of a chromosome as they are used in evolutionary algorithms and the lack of acquired traits, we must allow for the possibility of identical individuals. As a consequence, a population of an evolutionary algorithm is a *bag* or a *multiset* of individuals. In both biological and simulated evolution **generation** refers to the population at a certain point in time.

A new generation is created by **reproduction**, that is, by the generation of offspring from one or more (in biology: if more than one, then usually two) organisms, in

which genetic material of the parent individuals may be recombined. The same holds for computer science, only that the child creation process works directly on the chromosomes and that the number of parents may exceed two.

Finally, the **fitness** of an individual measures how high its chances of survival and reproduction are due to its adaptation to its environment. Since the quality of a biological organism w.r.t. its environment is difficult to assess objectively and simply defining fitness as the ability to survive can lead to a tautological "survival of the survivor," a formally more precise notion defines the fitness of an organism as the number of its offspring organisms that procreate themselves, thus linking (biological) fitness directly to the concept of differential reproduction. In computer science, the situation is simpler, because we are given an optimization problem that directly provides a fitness function with which solution candidates are to be evaluated.

It should be noted that, even though there are many parallels, simulated evolution is (usually) much simpler than biological evolution. For example, there are principles of biological evolution, e.g., speciation, that are usually not implemented in an evolutionary algorithm. On the genetic level, we already pointed out that in most life forms the genetic information is spread over multiple chromosomes, which often even come in so-called *homologous* pairs. These are pairs of chromosomes comprising the same genes, but possibly with different alleles, of which both or only one determine the corresponding phenotypical trait. Although such complications have their purpose in biological evolution, they are usually not simulated in a computer.

11.3.3 Building Blocks of an Evolutionary Algorithm

The general idea of an evolutionary algorithm is to employ evolution principles to generate increasingly better solution candidates for the optimization problem to solve. Essentially, this is achieved by evolving a population of solution candidates with the help of random variation and fitness-based selection of the next generation.

An evolutionary algorithm requires the following ingredients:

- an **encoding** for the solution candidates,
- a method to create an **initial population**,
- an **fitness function** to evaluate the individuals,
- a **selection method** on the basis of the fitness function,
- a set of **genetic operators** to modify chromosomes,
- a **termination criterion** for the search, and
- values for various **parameters**.

Since we want to evolve a population of solution candidates, we need a way of representing them as chromosomes, that is, we have to encode them, essentially as sequences of computational objects (like bits, characters, numbers etc.). Such an encoding may be so direct that the distinction between the genotype, as it is represented by the chromosome, and the phenotype, which is the actual solution candidate, becomes blurred. For example, for the problem of finding the side lengths

of a box with given surface area that has maximum volume (which we considered above), we may use the triples (x, y, z) of the side lengths, which are the solution candidates, directly as the chromosomes. In other cases there is a clear distinction between the solution candidate and its encoding, for example, if we have to turn the chromosome into some other structure (the phenotype) before we can evaluate its fitness. We will see several examples of such cases in later chapters.

Generally, the encoding of the solution candidates is highly problem-specific and there are no general rules. However, in Sect. 12.1 we discuss several aspects that attention should be paid to when choosing an encoding for a given problem. An inappropriate choice can severely reduce the effectiveness of the evolutionary algorithm or may even make it impossible to find a sufficiently good solution. Depending on the problem to solve, it is therefore highly recommended to spent considerable effort on finding a good encoding of the solution candidates.

Once we have decided on an encoding, we can create an initial population of solution candidates in the form of chromosomes representing them. Since chromosomes are simple sequences of computational objects, an initial population is commonly created by simply generating random sequences. However, depending on the problem to solve and the chosen encoding, more complex methods may be needed, especially, if the solution candidates have to satisfy certain constraints.

In order to mimic the influence of the environment in biological evolution, we need a fitness function with which we can evaluate the individuals of the created population. In many cases this fitness function is simply identical to the function to optimize, which is given by the optimization problem to solve. However, the fitness function may also contain additional elements that represent constraints that have to be satisfied in order for a solution candidate to be acceptable or that introduce a tendency toward certain additionally desired properties of a solution.

The (natural) selection process of biological evolution is simulated by a method to select candidate solutions according to their fitness. This method is used to choose the parents of offspring we want to create or to select those individuals that are transferred to the next generation without change. Such a selection method may simply transform the fitness values into a selection probability, such that better individuals have higher chances of getting chosen for the next generation.

The random variation of chromosomes is simulated by so-called genetic operators that modify and recombine chromosomes, for example, **mutation**, which randomly changes individual genes, and **crossover**, which exchanges parts of the chromosomes of parent individuals to produce offspring. Depending on the problem and the chosen encoding, the genetic operators can be very generic or highly problem-specific. The choice of the genetic operators is another element that effort should be spent on, especially in connection with the chosen encoding.

The ingredients described up to now allow us to generate a sequence of populations of (hopefully) increasingly better quality. However, while biological evolution is unbounded, we need a criterion when to stop the process in order to retrieve a final solution. Such a criterion may be, for example, that the algorithm is terminated (1) after a user-specified number of generations have been created, (2) there has

been no improvement (of the best solution candidate) for a user-specified number of generations, or (3) a user-specified minimum solution quality has been obtained.

To complete the specification of an evolutionary algorithm, we have to choose the values of several parameters, which include, for example, the size of the population to evolve, the fraction of individuals that is chosen from each population to produce offspring, the probability of a mutation occurring in an individual etc.

More formally, the procedure of an evolutionary algorithm looks like this:

Algorithm 11.1 (*General Scheme of an Evolutionary Algorithm*)

```
procedure evoalg;
begin
  t ← 0;                              (* initialize the generation counter *)
  initialize pop(t);                  (* create the initial population *)
  evaluate pop(t);                    (* and evaluate it (compute fitness) *)
  while not termination criterion do  (* loop until termination *)
    t ← t + 1;                        (* count the created generation *)
    select pop(t) from pop(t − 1);    (* select individuals based on fitness *)
    alter pop(t);                     (* apply genetic operators *)
    evaluate pop(t);                  (* evaluate the new population *)
  end                                 (* (compute new fitness) *)
end
```

That is, after having created and evaluated an initial population of solution candidates (in the form of chromosomes), a sequence of generations of solution candidates is computed. Each new generation is created by selecting individuals based on their fitness (with a higher fitness meaning a higher chance of getting selected). Then genetic operators (like mutation and crossover) are applied to the selected individuals. Next, the modified population (or at least the new individuals in it, which have been created by the genetic operators) is evaluated and the cycle starts over. This process continues until the chosen termination criterion is fulfilled.

11.4 The n-Queens Problem

The n-queens problem consists in the task to place n queens (a piece in the game of chess) of the same color onto an $n \times n$ chessboard in such a way that no rank (chess term for row), no file (chess term for column) and no diagonal contains more than one queen. Drawing on the rules of how a queen may move in the game of chess (namely horizontally, vertically or diagonally by any number of squares, but not onto a square that is occupied by a piece of the same color or beyond a square that is occupied by a piece of either color, see Fig. 11.1), we may say that the queens

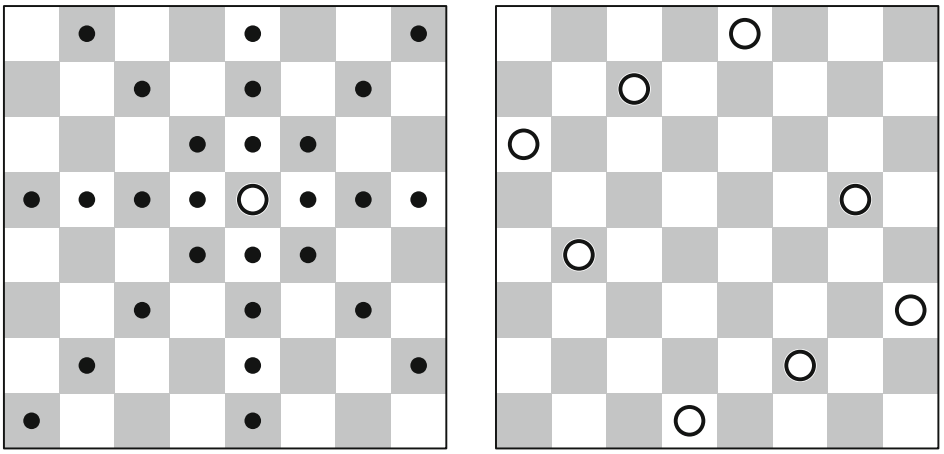


Fig. 11.1 Possible moves of a chess queen (*left*) and a solution of the 8-queens problem (*right*)

must be placed in such a way that none of them obstructs the possible moves of any other. As an example, Fig. 11.1 shows a solution of the 8-queens problem.

A well-known approach to solve the n-queens problem is a backtracking algorithm, which can be seen as an essentially exhaustive exploration of the space of candidate solutions with a depth-first search. Such an algorithm exploits the obvious fact that each rank (row) of the chessboard must contain exactly one queen. Hence it proceeds by placing the queens rank by rank. For each rank, it is checked for each possible square whether a queen placed on it obstructs the moves of any queens placed earlier (that is, whether there is already a queen on the same file (column) or the same diagonal). If this is not the case, the algorithm proceeds recursively to the next rank. However, if the newly placed queen obstructs any of the queens placed earlier or if the recursion to the next rank returns with the result that no solution could be found because obstructions could not be avoided, the queen is removed again and the algorithm continues with the next square. More formally, this backtracking algorithm can be described, for example, by the following function:

Algorithm 11.2 (*Backtracking Solution of the* ***n****-Queens Problem*)

```
function queens (n: int, k: int, board: array of array of boolean) : boolean;
begin                                   (* recursively solve n-queens problem *)
  if k ≥ n then return true;            (* if all ranks filled, abort with success *)
  for i = 0 up to n − 1 do begin        (* traverse the squares of rank k *)
    board[i][k] ← true;                 (* place a queen on square (i, k) *)
    if    not board[i][j] ∀j : 0 < j < k (* if no other queen is obstructed *)
    and not board[i − j][k − j] ∀j : 0 < j ≤ min(k, i)
    and not board[i + j][k − j] ∀j : 0 < j ≤ min(k, n − i − 1)
    and queens (n, k + 1, board)        (* and the recursion succeeds, *)
    then return true;                   (* a solution has been found *)
    board[i][k] ← false;                (* remove queen from square (i, k) *)
  end       (* for i = 0 ...*)
```

```
  return false;                    (* if no queen could be placed, *)
end                                (* abort with failure *)
```

This function is called with the number n of queens that defines the problem size, $k = 0$ indicating that the board should be filled starting from rank 0, and *board* being an $n \times n$ Boolean matrix that is initialized to *false* in all elements. If the function returns *true*, the problem can be solved. In this case one possible placement of the queens is indicated by the *true* entries in *board*. If the function returns *false*, the problem cannot be solved (the 3-queens problem, for example, has no solution). In this case the variable *board* is in its initial state of all *false* entries.

Note that the above algorithm can easily be modified to yield all possible solutions of an n-queens problem. In this case, the first if-statement, which checks whether all ranks have been filled, must be extended by a procedure that reports the found solution. In addition, the recursion must not be terminated if the recursion succeeds (and thus a solution has been found), but the loop over the squares of the current rank must be continued to find possibly existing other solutions.

Although a backtracking approach is very effective for sufficiently small n (up to, say, $n \approx 30$), it can take a long time to find a solution if n is larger. If we are interested in only one solution (i.e., one placement of the queens), there exists a better method, namely an analytical solution (which is slightly less well known than the backtracking approach). We compute the positions of the queens as follows:

Algorithm 11.3 (*Analytical Solution of the **n**-Queens Problem*)

- If $n = 2$ or $n = 3$, the n-queens problem does not have a solution.
- If n is odd (that is, if $n \bmod 2 = 1$),
  then we place a queen onto the square $(n - 1, n - 1)$ and decrement n by 1.
- If $n \bmod 6 \neq 2$, then we place
  the queens in the rows $y = 0, \ldots, \frac{n}{2} - 1$ in the columns $x = 2y + 1$ and
  the queens in the rows $y = \frac{n}{2}, \ldots, n - 1$ in the columns $x = 2y - n$.
- If $n \bmod 6 = 2$, then we place
  the queens in the rows $y = 0, \ldots, \frac{n}{2} - 1$ in the columns $x = (2y + \frac{n}{2}) \bmod n$ and
  the queens in the rows $y = \frac{n}{2}, \ldots, n - 1$ in the columns $x = (2y - \frac{n}{2} + 2) \bmod n$.

Due to this analytical solution, it is not quite appropriate to approach the n-queens problem with an evolutionary algorithm. Here we do so nevertheless, because this problem allows us to illustrate certain aspects of evolutionary algorithms very well.

In order to solve the n-queens problem with an evolutionary algorithm, we first need an encoding of the solution candidates. For this we draw on the same obvious fact that was already exploited for the backtracking algorithm, namely that each rank (row) of the chessboard must contain exactly one queen. Therefore we describe a candidate solution by a chromosome with n genes, each of which refers to one rank of the chessboard and has n possible alleles, namely the possible file (column)

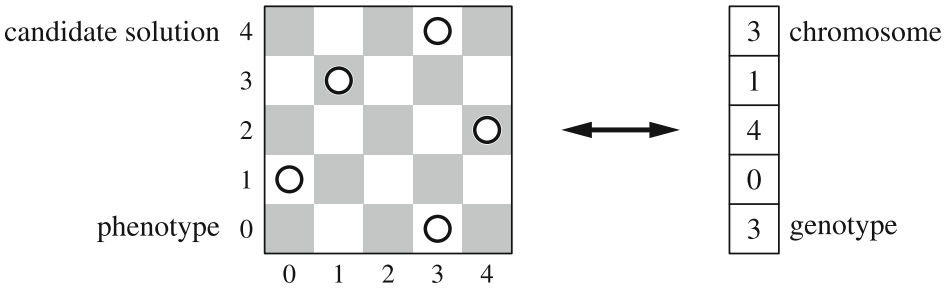
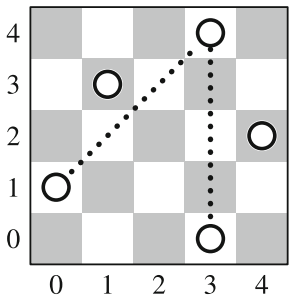


Fig. 11.2 Encoding of candidate solutions in the n-queens problem (here: $n = 5$)

Fig. 11.3 A solution candidate for the 5-queen problem with four obstructions and thus fitness value -2



numbers 0 to $n - 1$. Such a chromosome is interpreted as demonstrated in Fig. 11.2 (for $n = 5$): the allele of each gene indicates the file in which the queen is placed in the rank to which the gene refers. Note that with this encoding we can clearly distinguish between the genotype, which is an array of numbers, and the phenotype, which is the actual placement of the queens on the chessboard.

Note that this way of encoding the solution candidates has the advantage that we already exclude candidate solutions with more than one queen per rank. As a consequence, the search space becomes much smaller and thus can be explored more quickly and more effectively by an evolutionary algorithm. In order to reduce the search space even further, we may even consider restricting the chromosomes to permutations of the file (column) numbers. That is, each file number must occur for exactly one gene. However, although this clearly shrinks the search space even further, it introduces complications w.r.t. the genetic operators and thus we refrain from introducing this requirement here (however, cf. the discussion in Sect. 12.3).

In order to create an initial population we simple generate a random sequence of n numbers in $\{0, 1, \ldots, n - 1\}$ for each individual, because there are no special conditions that such a sequence has to satisfy to be an element of the search space.

The fitness function is derived directly from the defining characteristics of a solution: we compute for each queen the number of obstructions, that is, the number of other queens that obstruct its moves. Then we sum these numbers over the queens, divide by 2 and negate the result (see Fig. 11.3 for an example). Clearly, for an actual solution the fitness computed in this manner is zero, whereas it is negative for all other candidates. Note that we divide by two, because each obstruction is counted twice with the above procedure as obstruction is symmetric: if queen 1 obstructs queen 2, then queen 2 also obstructs queen 1. Note also that we negate the result,

because we want a fitness function that has to be maximized. For the example shown in Fig. 11.3 we have four (pairwise symmetric) obstructions (we may also say: two collisions between queens) and thus the fitness value is -2.

The fitness function immediately fixes the termination criterion: since a solution has the (maximally possible) fitness value of 0, we stop the algorithm as soon as a solution candidate with fitness 0 has been generated. However, to be on the safe side, we should also introduce a limit for the number of generations, so that the algorithm is guaranteed to terminate. Note, though, that with such an additional criterion the evolutionary algorithm may stop without having found a solution.

For the selection operation we choose a simple, but often very effective form of so-called **tournament selection**. That is, from the individuals of the current population a (small) sample of individuals is drawn that carry out a tournament with each other. This tournament is won by the individual with the highest fitness (ties are broken randomly). A copy of the winning individual is then added to the next generation and the participants of the tournament are replaced into the current population. The process is repeated until the next generation is complete, which usually means that it has reached the same size as the current population. Alternative selection methods as well as variants of tournament selection are discussed in Sect. 12.2.

In order to alter the selected individuals, we need genetic operators for recombination and variation. For the former we use so-called **one-point crossover**, which chooses a random cut point on the chromosomes of two parent individuals and exchanges the part on one side of the cut point between these individuals to create two children. An example for $n = 5$ is shown in Fig. 11.4: the genes below the randomly chosen cut point (the second out of the four possible ones) are exchanged. This example demonstrates what one hopes a genetic recombination may achieve: by combining partial solutions that are present in two deficient individuals (that is, both parent individuals have a negative fitness) a complete solution is obtained (the left child has a fitness of 0 and thus is a solution of the 5-queens problem). Alternative crossover operators are discussed in Sect. 12.3.

As a variation operation we use a random replacement of the alleles of randomly selected genes (so-called **standard mutation**). An example is shown in Fig. 11.5, in which two genes of a chromosome representing a candidate solution for the 5-queens problem receive new values. This example is fairly typical in the respect that most mutations reduce the fitness of the individual affected by them. However, mutation is nevertheless important, because alleles that are not present in the initial population cannot be created by recombination, which only reorganizes existing alle-

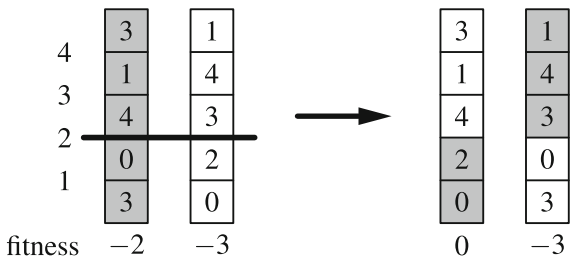


Fig. 11.4 One-point crossover of two chromosomes

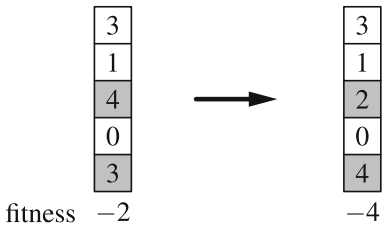


Fig. 11.5 Mutation of two genes of a chromosome

les. Generally, mutation can more easily introduce new alleles into chromosomes. Alternative mutation operators are discussed in Sect. 12.3.

Finally, we have to choose the values of several parameters. These include the size of the population to evolve (for example, $\mu = 1000$ individuals), the maximum number of generations to compute (for example, $g = 100$ generations), the size of the tournaments to carry out (for example, $\mu_t = 5$ individuals), the fraction of individuals that are subject to crossover (for example, $p_c = 0.5$), and the probability that a gene is subject to a mutation (for example, $p_m = 0.01$). Once all parameters have been chosen, the evolutionary algorithm is fully specified and can be executed according to the general scheme presented in Algorithm 11.1 on p. 197.

An implementation of this evolutionary algorithm, which can be found (as a command line program) on the web site for this book, shows that one can find solutions to the n-queens problem even for somewhat larger n than backtracking allows (also available as a command line program on the web site for this book), at least if a sufficiently large population is used and a sufficient number of generations is computed. However, as it is not guaranteed that a solution can be found, the program sometimes ends with a candidate solution that has a high fitness (like -1 or -2), but does not really solve the problem since obstructions remain.

By experimenting with the parameters, especially the fraction of individuals that are subject to crossover or the probability that a gene gets mutated, one can discover some interesting properties. For example, it turns out that mutations seem to be more important than crossover, since the speed with which a solution is found or the quality of solutions that are found in a given number of generations is not reduced if the fraction of individuals that are subject to crossover is reduced to zero. On the other hand, disallowing mutations causes the (average) solution quality to degrade significantly. Note, however, that these are not general characteristics of evolutionary algorithms, but are exhibited only for this particular problem and chosen encoding, selection and genetic operators etc. It cannot be transferred directly to other applications, where the crossover operation may contribute more to a solution being found and mutation, if its probability is chosen too large, rather degrades performance.

11.5 Related Optimization Techniques

In classical optimization (for example, in operations research) many techniques and algorithms have been developed that are fairly closely related to evolution-