Chapter 8

Introduction to Evolutionary Computation

Evolution is an optimization process where the aim is to improve the ability of an organism (or system) to survive in dynamically changing and competitive environments. Evolution is a concept that has been hotly debated over centuries, and still causes active debates.[1] When talking about evolution, it is important to first identify the area in which evolution can be defined, for example, cosmic, chemical, stellar and planetary, organic or man-made systems of evolution. For these different areas, evolution may be interpreted differently. For the purpose of this part of the book, the focus is on biological evolution. Even for this specific area, attempts to define the term *biological evolution* still cause numerous debates, with the Lamarckian and Darwinian views being the most popular and accepted. While Darwin (1809–1882) is generally considered as the founder of both the theory of evolution and the principle of common descent, Lamarck (1744–1829) was possibly the first to theorize about biological evolution.

Jean-Baptiste Lamarck's theory of evolution was that of *heredity*, i.e. the inheritance of acquired traits. The main idea is that individuals adapt during their lifetimes, and transmit their traits to their offspring. The offspring then continue to adapt. According to Lamarckism, the method of adaptation rests on the concept of use and disuse: over time, individuals lose characteristics they do not require, and develop those which are useful by "exercising" them.

It was Charles Darwin's theory of *natural selection* that became the foundation of biological evolution (Alfred Wallace developed a similar theory at the same time, but independently of Darwin). The Darwinian theory of evolution [173] can be summarized as: In a world with limited resources and stable populations, each individual competes with others for survival. Those individuals with the "best" characteristics (traits) are more likely to survive and to reproduce, and those characteristics will be passed on to their offspring. These desirable characteristics are inherited by the following generations, and (over time) become dominant among the population.

A second part of Darwin's theory states that, during production of a child organism,

[1] Refer to http:www.johmann.net/book/ciy7-1.html
http://www.talkorigins.org/faqs/evolution-definition.html
http://www.evolutionfairytale.com/articles_debates/evolution-definition.html
http://www.creationdesign.org/ (accessed 05/08/2004).

Computational Intelligence: An Introduction, Second Edition A.P. Engelbrecht

random events cause random changes to the child organism's characteristics. If these new characteristics are a benefit to the organism, then the chances of survival for that organism are increased.

Evolutionary computation (EC) refers to computer-based problem solving systems that use computational models of evolutionary processes, such as natural selection, survival of the fittest and reproduction, as the fundamental components of such computational systems.

This chapter gives an overview of the evolution processes modeled in EC. Section 8.1 presents a generic evolutionary algorithm (EA) and reviews the main components of EAs. Section 8.2 discusses ways in which the computational individuals are represented, and Section 8.3 discusses aspects about the initial population. The importance of fitness functions, and different types of fitness functions are discussed in Section 8.4. Selection and reproduction operators are respectively discussed in Sections 8.5 and 8.6. Algorithm stopping conditions are considered in Section 8.7. A short discussion on the differences between EC and classical optimization is given in Section 8.8.

8.1 Generic Evolutionary Algorithm

Evolution via natural selection of a randomly chosen population of individuals can be thought of as a search through the space of possible chromosome values. In that sense, an evolutionary algorithm (EA) is a stochastic search for an optimal solution to a given problem. The evolutionary search process is influenced by the following main components of an EA:

- an **encoding** of solutions to the problem as a chromosome;
- a **function** to evaluate the **fitness**, or survival strength of individuals;
- **initialization** of the initial population;
- **selection** operators; and
- **reproduction** operators.

Algorithm 8.1 shows how these components are combined to form a generic EA.

Algorithm 8.1 Generic Evolutionary Algorithm

Let $t = 0$ be the generation counter;
Create and initialize an n_x-dimensional population, $\mathcal{C}(0)$, to consist of n_s individuals;
while *stopping condition(s) not true* **do**
    Evaluate the fitness, $f(\mathbf{x}_i(t))$, of each individual, $\mathbf{x}_i(t)$;
    Perform reproduction to create offspring;
    Select the new population, $\mathcal{C}(t+1)$;
    Advance to the new generation, i.e. $t = t + 1$;
end

The steps of an EA are applied iteratively until some stopping condition is satisfied (refer to Section 8.7). Each iteration of an EA is referred to as a generation.

The different ways in which the EA components are implemented result in diffferent EC paradigms:

- **Genetic algorithms** (GAs), which model genetic evolution.
- **Genetic programming** (GP), which is based on genetic algorithms, but individuals are programs (represented as trees).
- **Evolutionary programming** (EP), which is derived from the simulation of adaptive behavior in evolution (i.e. *phenotypic* evolution).
- **Evolution strategies** (ESs), which are geared toward modeling the strategic parameters that control variation in evolution, i.e. the evolution of evolution.
- **Differential evolution** (DE), which is similar to genetic algorithms, differing in the reproduction mechanism used.
- **Cultural evolution** (CE), which models the evolution of culture of a population and how the culture influences the genetic and phenotypic evolution of individuals.
- **Co-evolution** (CoE), where initially "dumb" individuals evolve through cooperation, or in competition with one another, acquiring the necessary characteristics to survive.

These paradigms are discussed in detail in the chapters that follow in this part of the book.

With reference to Algorithm 8.1, both parts of Darwin's theory are encapsulated within this algorithm:

- Natural selection occurs within the reproduction operation where the "best" parents have a better chance of being selected to produce offspring, and to be selected for the new population.
- Random changes are effected through the mutation operator.

8.2 Representation – The Chromosome

In nature, organisms have certain characteristics that influence their ability to survive and to reproduce. These characteristics are represented by long strings of information contained in the chromosomes of the organism. Chromosomes are structures of compact intertwined molecules of DNA, found in the nucleus of organic cells. Each chromosome contains a large number of genes, where a gene is the unit of heredity. Genes determine many aspects of anatomy and physiology through control of protein production. Each individual has a unique sequence of genes. An alternative form of a gene is referred to as an *allele*.

In the context of EC, each individual represents a candidate solution to an optimization problem. The characteristics of an individual is represented by a *chromosome*, also

referred to as a *genome*. These characteristics refer to the variables of the optimization problem, for which an optimal assignment is sought. Each variable that needs to be optimized is referred to as a *gene*, the smallest unit of information. An assignment of a value from the allowed domain of the corresponding variable is referred to as an *allele*. Characteristics of an individual can be divided into two classes of evolutionary information: genotypes and phenotypes. A *genotype* describes the genetic composition of an individual, as inherited from its parents; it represents which allele the individual possesses. A *phenotype* is the expressed behavioral traits of an individual in a specific environment; it defines what an individual looks like. Complex relationships exist between the genotype and phenotype [570]:

- *pleiotropy*, where random modification of genes causes unexpected variations in the phenotypic traits, and
- *polygeny*, where several genes interact to produce a specific phenotypic trait.

An important step in the design of an EA is to find an appropriate representation of candidate solutions (i.e. chromosomes). The efficiency and complexity of the search algorithm greatly depends on the representation scheme. Different EAs from the different paradigms use different representation schemes. Most EAs represent solutions as vectors of a specific data type. An exception is genetic programming (GP) where individuals are represented in a tree format.

The classical representation scheme for GAs is binary vectors of fixed length. In the case of an n_x-dimensional search space, each individual consists of n_x variables with each variable encoded as a bit string. If variables have binary values, the length of each chromosome is n_x bits. In the case of nominal-valued variables, each nominal value can be encoded as an n_d-dimensional bit vector where 2^{n_d} is the total number of discrete nominal values for that variable. To solve optimization problems with continuous-valued variables, the continuous search space problem can be mapped into a discrete programming problem. For this purpose mapping functions are needed to convert the space $\{0,1\}^{n_b}$ to the space $\mathbb{R}^{n_x}$. For such mapping, each continuous-valued variable is mapped to an n_d-dimensional bit vector, i.e.

$$\phi : \mathbb{R} \rightarrow (0,1)^{n_d} \tag{8.1}$$

The domain of the continuous space needs to be restricted to a finite range, $[\mathbf{x}_{min}, \mathbf{x}_{max}]$. A standard binary encoding scheme can be used to transform the individual $\mathbf{x} = (x_1, \ldots, x_j, \ldots, x_{n_x})$, with $x_j \in \mathbb{R}$ to the binary-valued individual, $\mathbf{b} = (\mathbf{b}_1, \ldots, \mathbf{b}_j, \ldots, \mathbf{b}_{n_x})$, where $\mathbf{b}_j = (b_{(j-1)n_d+1}, \ldots, b_{jn_d})$, with $b_l \in \{0,1\}$ and the total number of bits, $n_b = n_x n_d$. Decoding each $\mathbf{b}_j$ back to a floating-point representation can be done using the function, $\Phi_j : \{0,1\}^{n_d} \rightarrow [x_{min,j}, x_{max,j}]$, where [39]

$$\Phi_j(\mathbf{b}) = x_{min,j} + \frac{x_{max,j} - x_{min,j}}{2^{n_d} - 1} \left(\sum_{l=1}^{n_d-1} b_{j(n_d-l)} 2^l \right) \tag{8.2}$$

Holland [376] and De Jong [191] provided the first applications of genetic algorithms to solve continuous-valued problems using such a mapping scheme. It should be noted

that if a bitstring representation is used, a grid search is done in a discrete search space. The EA may therefore fail to obtain a precise optimum. In fact, for a conversion form a floating-point value to a bitstring of n_d bits, the maximum attainable accuracy is

$$\frac{x_{max,j} - x_{min,j}}{2^{n_d} - 1} \tag{8.3}$$

for each vector component, $j = 1, \ldots, n_x$.

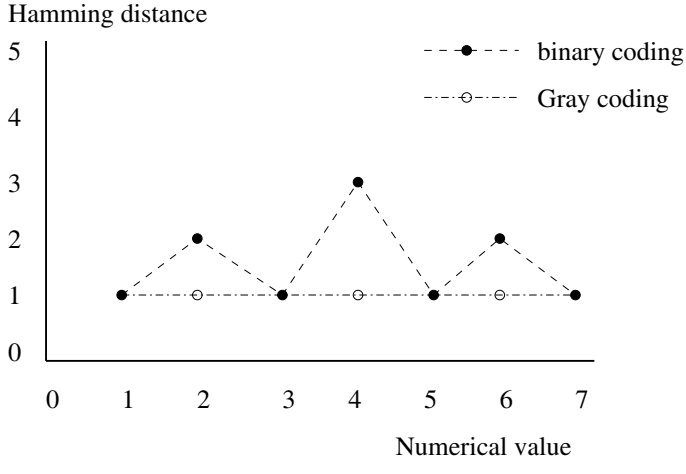


Figure 8.1 Hamming Distance for Binary and Gray Coding

While binary coding is frequently used, it has the disadvantage of introducing Hamming cliffs as illustrated in Figure 8.1. A Hamming cliff is formed when two numerically adjacent values have bit representations that are far apart. For example, consider the decimal numbers 7 and 8. The corresponding binary representations are (using a 4-bit representation) $7 = 0111$ and $8 = 1000$, with a Hamming distance of 4 (the Hamming distance is the number of corresponding bits that differ). This presents a problem when a small change in variables should result in a small change in fitness. If, for example, 7 represents the optimal solution, and the current best solution has a fitness of 8, many bits need to be changed to cause a small change in fitness value.

An alternative bit representation is to use Gray coding, where the Hamming distance between the representation of successive numerical values is one (as illustrated in Figure 8.1). Table 8.1 compares binary and Gray coding for a 3-bit representation.

Binary numbers can easily be converted to Gray coding using the conversion

$$\begin{aligned} g_1 &= b_1 \\ g_l &= b_{l-1}\overline{b}_l + \overline{b}_{l-1}b_l \end{aligned} \tag{8.4}$$

where b_l is bit l of the binary number $b_1 b_2 \cdots b_{n_b}$, with b_1 the most significant bit; $\overline{b}_l$ denotes *not* b_l, + means logical OR, and multiplication implies logical AND.

A Gray code representation, $\mathbf{b}_j$ can be converted to a floating-point representation

Table 8.1 Binary and Gray Coding

	Binary	*Gray*
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

using

$$\Phi_j(\mathbf{b}) = x_{min,j} + \frac{x_{max,j} - x_{min,j}}{2^{n_d} - 1} \left(\sum_{l=1}^{n_d-1} \left(\sum_{q=1}^{n_d-l} b_{(j-1)n_d+q} \right) \bmod 2 \right) 2^l \tag{8.5}$$

Real-valued representations have been used for a number of EAs, including GAs. Although EP (refer to Chapter 11) was originally developed for finite-state machine representations, it is now mostly applied to real-valued representations where each vector component is a floating-point number, i.e. $x_j \in \mathbb{R}, j = 1, \ldots, n_x$. ESs and DE, on the other hand, have been developed for floating-point representation (refer to Chapters 12 and 13). Real-valued representations have also been used for GAs [115, 178, 251, 411, 918]. Michalewicz [583] indicated that the original floating-point representation outperforms an equivalent binary representation, leading to more accurate, faster obtained solutions.

Other representation schemes that have been used include integer representations [778], permutations [778, 829, 905, 906], finite-state representations [265, 275], tree representations (refer to Chapter 10), and mixed-integer representations [44].

8.3 Initial Population

Evolutionary algorithms are stochastic, population-based search algorithms. Each EA therefore maintains a population of candidate solutions. The first step in applying an EA to solve an optimization problem is to generate an initial population. The standard way of generating an initial population is to assign a random value from the allowed domain to each of the genes of each chromosome. The goal of random selection is to ensure that the initial population is a uniform representation of the entire search space. If regions of the search space are not covered by the initial population, chances are that those parts will be neglected by the search process.

The size of the initial population has consequences in terms of computational complexity and exploration abilities. Large numbers of individuals increase diversity, thereby improving the exploration abilities of the population. However, the more the individuals, the higher the computational complexity per generation. While the execution time per generation increases, it may be the case that fewer generations are needed to locate an acceptable solution. A small population, on the other hand will represent a small part of the search space. While the time complexity per generation is low, the EA may need more generations to converge than for a large population.

In the case of a small population, the EA can be forced to explore more of the search space by increasing the rate of mutation.

8.4 Fitness Function

In the Darwinian model of evolution, individuals with the best characteristics have the best chance to survive and to reproduce. In order to determine the ability of an individual of an EA to survive, a mathematical function is used to quantify how good the solution represented by a chromosome is. The *fitness function*, f, maps a chromosome representation into a scalar value:

$$f : \Gamma^{n_x} \rightarrow \mathbb{R} \tag{8.6}$$

where Γ represents the data type of the elements of an n_x-dimensional chromosome.

The fitness function represents the objective function, Ψ, which describes the optimization problem. It is not necessarily the case that the chromosome representation corresponds to the representation expected by the objective function. In such cases, a more detailed description of the fitness function is

$$f : \mathcal{S}_C \stackrel{\Phi}{\rightarrow} \mathcal{S}_X \stackrel{\Psi}{\rightarrow} \mathbb{R} \stackrel{\Upsilon}{\rightarrow} \mathbb{R}_+ \tag{8.7}$$

where $\mathcal{S}_C$ represents the search space of the objective function, and Φ, Ψ and Υ respectively represent the chromosome decoding function, the objective function, and the scaling function. The (optional) scaling function is used in proportional selection to ensure positive fitness values (refer to Section 8.5). As an example,

$$f : \{0,1\}^{n_b} \stackrel{\Phi}{\rightarrow} \mathbb{R}^{n_x} \stackrel{\Psi}{\rightarrow} \mathbb{R} \stackrel{\Upsilon}{\rightarrow} \mathbb{R}_+ \tag{8.8}$$

where an n_b-bitstring representation is converted to a floating-point representation using either equation (8.2) or (8.5).

For the purposes of the remainder of this part on EC, it is assumed that $\mathcal{S}_C = \mathcal{S}_X$ for which $f = \Psi$.

Usually, the fitness function provides an absolute measure of fitness. That is, the solution represented by a chromosome is directly evaluated using the objective function. For some applications, for example game learning (refer to Chapter 11) it is not possible to find an absolute fitness function. Instead, a relative fitness measure is used

to quantify the performance of an individual in relation to that of other individuals in the population or a competing population. Relative fitness measures are used in coevolutionary algorithms (refer to Chapter 15).

It is important to realize at this point that different types of optimization problems exist (refer to Section A.3), which have an influence on the formulation of the fitness function:

- **Unconstrained** optimization problems as defined in Definition A.4, where, assuming that $\mathcal{S}_C = \mathcal{S}_X$, the fitness function is simply the objective function.
- **Constrained** optimization problems as defined in Definition A.5. To solve constrained problems, some EAs change the fitness function to contain two objectives: one is the original objective function, and the other is a constraint penalty function (refer to Section A.6).
- **Multi-objective** optimization problems (MOP) as defined in Definition A.10. MOPs can be solved by using a weighted aggregation approach (refer to Section A.8), where the fitness function is a weighted sum of all the sub-objectives (refer to equation (A.44)), or by using a Pareto-based optimization algorithm.
- **Dynamic** and **noisy** problems, where function values of solutions change over time. Dynamic fitness functions are time-dependent whereas noisy functions usually have an added Gaussian noise component. Dynamic problems are defined in Definition A.16. Equation (A.58) gives a noisy function with an additive Gaussian noise component.

As a final comment on the fitness function, it is important to emphasize its role in an EA. The evolutionary operators, e.g. selection, crossover, mutation and elitism, usually make use of the fitness evaluation of chromosomes. For example, selection operators are inclined towards the most-fit individuals when selecting parents for crossover, while mutation leans towards the least-fit individuals.

8.5 Selection

Selection is one of the main operators in EAs, and relates directly to the Darwinian concept of survival of the fittest. The main objective of selection operators is to emphasize better solutions. This is achieved in two of the main steps of an EA:

- **Selection of the new population**: A new population of candidate solutions is selected at the end of each generation to serve as the population of the next generation. The new population can be selected from only the offspring, or from both the parents and the offspring. The selection operator should ensure that good individuals do survive to next generations.
- **Reproduction**: Offspring are created through the application of crossover and/or mutation operators. In terms of crossover, "superior" individuals should have more opportunities to reproduce to ensure that offspring contain genetic material of the best individuals. In the case of mutation, selection mechanisms

should focus on "weak" individuals. The hope is that mutation of weak individuals will result in introducing better traits to weak individuals, thereby increasing their chances of survival.

Many selection operators have been developed. A summary of the most frequently used operators is given in this section. Preceding this summary is a discussion of selective pressure in Section 8.5.1.

8.5.1 Selective Pressure

Selection operators are characterized by their *selective pressure*, also referred to as the *takeover time*, which relates to the time it requires to produce a uniform population. It is defined as the speed at which the best solution will occupy the entire population by repeated application of the selection operator alone [38, 320]. An operator with a high selective pressure decreases diversity in the population more rapidly than operators with a low selective pressure, which may lead to premature convergence to suboptimal solutions. A high selective pressure limits the exploration abilities of the population.

8.5.2 Random Selection

Random selection is the simplest selection operator, where each individual has the same probability of $\frac{1}{n_s}$ (where n_s is the population size) to be selected. No fitness information is used, which means that the best and the worst individuals have exactly the same probability of surviving to the next generation. Random selection has the lowest selective pressure among the selection operators discussed in this section.

8.5.3 Proportional Selection

Proportional selection, proposed by Holland [376], biases selection towards the most-fit individuals. A probability distribution proportional to the fitness is created, and individuals are selected by sampling the distribution,

$$\varphi_s(\mathbf{x}_i(t)) = \frac{f_\Upsilon(\mathbf{x}_i(t))}{\sum_{l=1}^{n_s} f_\Upsilon(\mathbf{x}_l(t))} \tag{8.9}$$

where n_s is the total number of individuals in the population, and $\varphi_s(\mathbf{x}_i)$ is the probability that $\mathbf{x}_i$ will be selected. $f_\Upsilon(\mathbf{x}_i)$ is the scaled fitness of $\mathbf{x}_i$, to produce a positive floating-point value. For minimization problems, possible choices of scaling function, Υ, are

- $f_\Upsilon(\mathbf{x}_i(t)) = \Upsilon(\mathbf{x}_i(t)) = f_{max} - f_\Psi(\mathbf{x}_i(t))$ where $f_\Psi(\mathbf{x}_i(t)) = \Psi(\mathbf{x}_i(t))$ is the raw fitness value of $\mathbf{x}_i(t)$. However, knowledge of f_{max} (the maximum possible fitness) is usually not available. An alternative is to use $f_{max}(t)$, which is the maximum fitness observed up to time step t.

- $f_\Upsilon(\mathbf{x}_i(t)) = \Upsilon(\mathbf{x}_i(t)) = \frac{1}{1+f_\Psi(\mathbf{x}_i(t))-f_{min}(t)}$, where $f_{min}(t)$ is the minimum observed fitness up to time step t. Here, $f_\Upsilon(\mathbf{x}_i(t)) \in (0, 1]$.

In the case of a maximization problem, the fitness values can be scaled to the range (0,1] using

$$f_\Upsilon(\mathbf{x}_i(t)) = \Upsilon(\mathbf{x}_i(t)) = \frac{1}{1 + f_{max}(t) - f(\mathbf{x}_i(t))} \tag{8.10}$$

Two popular sampling methods used in proportional selection is roulette wheel sampling and stochastic universal sampling.

Assuming maximization, and normalized fitness values, roulette wheel selection is summarized in Algorithm 8.2. Roulette wheel selection is an example proportional selection operator where fitness values are normalized (e.g. by dividing each fitness by the maximum fitness value). The probability distribution can then be seen as a roulette wheel, where the size of each slice is proportional to the normalized selection probability of an individual. Selection can be likened to the spinning of a roulette wheel and recording which slice ends up at the top; the corresponding individual is then selected.

Algorithm 8.2 Roulette Wheel Selection

Let $i = 1$, where i denotes the chromosome index;
Calculate $\varphi_s(\mathbf{x}_i)$ using equation (8.9);
$sum = \varphi_s(\mathbf{x}_i)$;
Choose $r \sim U(0, 1)$;
while $sum < r$ **do**
    $i = i + 1$, i.e. advance to the next chromosome;
    $sum = sum + \varphi_s(\mathbf{x}_i)$;
end
Return $\mathbf{x}_i$ as the selected individual;

When roulette wheel selection is used to create offspring to replace the entire population, n_s independent calls are made to Algorithm 8.2. It was found that this results in a high variance in the number of offspring created by each individual. It may happen that the best individual is not selected to produce offspring during a given generation. To prevent this problem, Baker [46] proposed stochastic universal sampling (refer to Algorithm 8.3), used to determine for each individual the number of offspring, λ_i, to be produced by the individual with only one call to the algorithm.

Because selection is directly proportional to fitness, it is possible that strong individuals may dominate in producing offspring, thereby limiting the diversity of the new population. In other words, proportional selection has a high selective pressure.

Algorithm 8.3 Stochastic Universal Sampling

for $i = 1, \ldots, n_s$ **do**
    $\lambda_i(t) = 0$;
end
$r \sim U(0, \frac{1}{\lambda})$, where λ is the total number of offspring;
$sum = 0.0$;
for $i = 1, \ldots, n_s$ **do**
    $sum = sum + \varphi_s(\mathbf{x}_i(t))$;
    while $r < sum$ **do**
        $\lambda_i + +$;
        $r = r + \frac{1}{\lambda}$;
    end
end
return $\lambda = (\lambda_1, \ldots, \lambda_{n_s})$;

8.5.4 Tournament Selection

Tournament selection selects a group of n_{ts} individuals randomly from the population, where $n_{ts} < n_s$ (n_s is the total number of individuals in the population). The performance of the selected n_{ts} individuals is compared and the best individual from this group is selected and returned by the operator. For crossover with two parents, tournament selection is done twice, once for the selection of each parent.

Provided that the tournament size, n_{ts}, is not too large, tournament selection prevents the best individual from dominating, thus having a lower selection pressure. On the other hand, if n_{ts} is too small, the chances that bad individuals are selected increase.

Even though tournament selection uses fitness information to select the best individual of a tournament, random selection of the individuals that make up the tournament reduces selective pressure compared to proportional selection. However, note that the selective pressure is directly related to n_{ts}. If $n_{ts} = n_s$, the best individual will always be selected, resulting in a very high selective pressure. On the other hand, if $n_{ts} = 1$, random selection is obtained.

8.5.5 Rank-Based Selection

Rank-based selection uses the rank ordering of fitness values to determine the probability of selection, and not the absolute fitness values. Selection is therefore independent of actual fitness values, with the advantage that the best individual will not dominate in the selection process.

Non-deterministic linear sampling selects an individual, $\mathbf{x}_i$, such that $i \sim U(0, U(0, n_s - 1))$, where the individuals are sorted in decreasing order of fitness value. It is also assumed that the rank of the best individual is 0, and that of the worst individual is $n_s - 1$.

Linear ranking assumes that the best individual creates $\hat{\lambda}$ offspring, and the worst individual $\tilde{\lambda}$, where $1 \leq \hat{\lambda} \leq 2$ and $\tilde{\lambda} = 2 - \hat{\lambda}$. The selection probability of each individual is calculated as

$$\varphi_s(\mathbf{x}_i(t)) = \frac{\tilde{\lambda} + (f_r(\mathbf{x}_i(t))/(n_s - 1))(\hat{\lambda} - \tilde{\lambda})}{n_s} \tag{8.11}$$

where $f_r(\mathbf{x}_i(t))$ is the rank of $\mathbf{x}_i(t)$.

Nonlinear ranking techniques calculate the selection probabilities, for example, as follows:

$$\varphi_s(\mathbf{x}_i(t)) = \frac{1 - e^{-f_r(\mathbf{x}_i(t))}}{\beta} \tag{8.12}$$

or

$$\varphi_s(\mathbf{x}_i) = \nu(1 - \nu)^{n_p - 1 - f_r(\mathbf{x}_i)} \tag{8.13}$$

where $f_r(\mathbf{x}_i)$ is the rank of $\mathbf{x}_i$ (i.e. the individual's position in the ordered sequence of individuals), β is a normalization constant, and ν indicates the probability of selecting the next individual.

Rank-based selection operators may use any sampling method to select individuals, e.g. roulette wheel selection (Algorithm 8.2) or stochastic universal sampling (Algorithm 8.3).

8.5.6 Boltzmann Selection

Boltzmann selection is based on the thermodynamical principles of simulated annealing (refer to Section A.5.2). It has been used in different ways, one of which computes selection probabilities as follows:

$$\varphi(\mathbf{x}_i(t)) = \frac{1}{1 + e^{f(\mathbf{x}_i(t))/T(t)}} \tag{8.14}$$

where $T(t)$ is the temperature parameter. A temperature schedule is used to reduce $T(t)$ from its initial large value to a small value.

The initial large value ensures that all individuals have an equal probability of being selected. As $T(t)$ becomes smaller, selection focuses more on the good individuals. The sampling methods discussed in Section 8.5.3 can be used to select individuals.

Alternatively, Boltzmann selection can be used to select between two individuals, for example, to decide if a parent, $\mathbf{x}_i(t)$, should be replaced by its offspring, $\mathbf{x}_i'(t)$. If

$$U(0,1) > \frac{1}{1 + e^{(f(\mathbf{x}_i(t)) - f(\mathbf{x}_i'(t)))/T(t)}} \tag{8.15}$$

then $\mathbf{x}_i'(t)$ is selected; otherwise, $\mathbf{x}_i(t)$ is selected.

8.5.7 $(\mu \dot{+} \lambda)$-Selection

The (μ, λ)- and $(\mu+\lambda)$-selection methods are deterministic rank-based selection methods used in evolutionary strategies (refer to Chapter 12). For both methods μ indicates the number of parents (which is the size of the population), and λ is the number of offspring produced from each parent. After production of the λ offspring, (μ, λ)-selection selects the best μ offspring for the next population. This process of selection is very similar to beam search (refer to Section A.5.2). $(\mu + \lambda)$-selection, on the other hand, selects the best μ individuals from both the parents and the offspring.

8.5.8 Elitism

Elitism refers to the process of ensuring that the best individuals of the current population survive to the next generation. The best individuals are copied to the new population without being mutated. The more individuals that survive to the next generation, the less the diversity of the new population.

8.5.9 Hall of Fame

The hall of fame is a selection scheme similar to the list of best players of an arcade game. For each generation, the best individual is selected to be inserted into the hall of fame. The hall of fame will therefore contain an archive of the best individuals found from the first generation. The hall of fame can be used as a parent pool for the crossover operator, or, at the last generation, the best individual is selected as the best one in the hall of fame.

8.6 Reproduction Operators

Reproduction is the process of producing offspring from selected parents by applying crossover and/or mutation operators. Crossover is the process of creating one or more new individuals through the combination of genetic material randomly selected from two or more parents. If selection focuses on the most-fit individuals, the selection pressure may cause premature convergence due to reduced diversity of the new populations.

Mutation is the process of randomly changing the values of genes in a chromosome. The main objective of mutation is to introduce new genetic material into the population, thereby increasing genetic diversity. Mutation should be applied with care not to distort the good genetic material in highly fit individuals. For this reason, mutation is usually applied at a low probability. Alternatively, the mutation probability can be made proportional to the fitness of individuals: the less fit the individual, the more it is mutated. To promote exploration in the first generations, the mutation probability can be initialized to a large value, which is then reduced over time to allow for

exploitation during the final generations.

Reproduction can be applied with replacement, in which case newly generated individuals replace parent individuals only if the fitness of the new offspring is better than that of the corresponding parents.

Since crossover and mutation operators are representation and EC paradigm dependent, the different implementations of these operators are covered in chapters that follow.

8.7 Stopping Conditions

The evolutionary operators are iteratively applied in an EA until a stopping condition is satisfied. The simplest stopping condition is to limit the number of generations that the EA is allowed to execute, or alternatively, a limit is placed on the number of fitness function evaluations. This limit should not be too small, otherwise the EA will not have sufficient time to explore the search space.

In addition to a limit on execution time, a convergence criterion is usually used to detect if the population has converged. Convergence is loosely defined as the event when the population becomes stagnant. In other words, when there is no genotypic or phenotypic change in the population. The following convergence criteria can be used:

- **Terminate when no improvement is observed over a number of consecutive generations**. This can be detected by monitoring the fitness of the best individual. If there is no significant improvement over a given time window, the EA can be stopped. Alternatively, if the solution is not satisfactory, mechanisms can be applied to increase diversity in order to force further exploration. For example, the mutation probability and mutational step sizes can be increased.
- **Terminate when there is no change in the population**. If, over a number of consecutive generations, the average change in genotypic information is too small, the EA can be stopped.
- **Terminate when an acceptable solution has been found**. If $\mathbf{x}^*(t)$ represents the optimum of the objective function, then if the best individual, $\mathbf{x}_i$, is such that $f(\mathbf{x}_i) \leq |f(\mathbf{x}) - \epsilon|$, an acceptable solution is found; ϵ is the error threshold. If ϵ is too large, solutions may be bad. Too small values of ϵ may cause the EA never to terminate if a time limit is not imposed.
- **Terminate when the objective function slope is approximately zero**, as defined in equation (16.16) of Chapter 16.

8.8 Evolutionary Computation versus Classical Optimization

While classical optimization algorithms have been shown to be very successful (and more efficient than EAs) in linear, quadratic, strongly convex, unimodal and other specialized problems, EAs have been shown to be more efficient for discontinuous, non-differentiable, multimodal and noisy problems.

EC and classical optimization (CO) differ mainly in the search process and information about the search space used to guide the search process:

- **The search process:** CO uses deterministic rules to move from one point in the search space to the next point. EC, on the other hand, uses probabilistic transition rules. Also, EC applies a parallel search of the search space, while CO uses a sequential search. An EA search starts from a diverse set of initial points, which allows parallel search of a large area of the search space. CO starts from one point, successively adjusting this point to move toward the optimum.
- **Search surface information:** CO uses derivative information, usually first-order or second-order, of the search space to guide the path to the optimum. EC, on the other hand, uses no derivative information. The fitness values of individuals are used to guide the search.

8.9 Assignments

1. Discuss the importance of the fitness function in EC.
2. Discuss the difference between genetic and phenotypic evolution.
3. In the case of a small population size, how can we ensure that a large part of the search space is covered?
4. How can premature convergence be prevented?
5. In what situations will a high mutation rate be of advantage?
6. Is the following statement valid? *"A genetic algorithm is assumed to have converged to a local or global solution when the ratio $\overline{f}/f_{max}$ is close to 1, where f_{max} and $\overline{f}$ are the maximum and average fitness of the evolving population respectively."*
7. How can an EA be used to train a NN? In answering this question, focus on
   (a) the representation scheme, and
   (b) fitness function.
8. Show how an EA can be used to solve systems of equations, by illustrating how
   (a) solutions are represented, and
   (b) the fitness is calculated.

   What problem can be identified in using an EA to solve systems of equations?
9. How can the effect of a high selective pressure be countered?

10. Under which condition will stochastic universal sampling behave like tournament selection?
11. Identify disadvantages of fitness-based selection operators.
12. For the nonlinear ranking methods given in equations (8.12) and (8.13), indicate if these assume a minimization or maximization problem.
13. Critisize the following stopping condition: Stop execution of the EA when there is no significant change in the average fitness of the population over a number of consecutive generations.

Chapter 9

Genetic Algorithms

Genetic algorithms (GA) are possibly the first algorithmic models developed to simulate genetic systems. First proposed by Fraser [288, 289], and later by Bremermann [86] and Reed *et al.* [711], it was the extensive work done by Holland [376] that popularized GAs. It is then also due to his work that Holland is generally considered the father of GAs.

GAs model genetic evolution, where the characteristics of individuals are expressed using genotypes. The main driving operators of a GA is selection (to model survival of the fittest) and recombination through application of a crossover operator (to model reproduction). This section discusses in detail GAs and their evolution operators, organized as follows: Section 9.1 reviews the canonical GA as proposed by Holland. Crossover operators for binary and floating-point representations are discussed in Section 9.2. Mutation operators are covered in Section 9.3. GA control parameters are discussed in Section 9.4. Different GA implementations are reviewed in Section 9.5, while advanced topics are considered in Section 9.6. A summary of GA applications is given in Section 9.7.

9.1 Canonical Genetic Algorithm

The canonical GA (CGA) as proposed by Holland [376] follows the general algorithm as given in Algorithm 8.1, with the following implementation specifics:

- A bitstring representation was used.
- Proportional selection was used to select parents for recombination.
- One-point crossover (refer to Section 9.2) was used as the primary method to produce offspring.
- Uniform mutation (refer to Section 9.3) was proposed as a background operator of little importance.

It is valuable to note that mutation was not considered as an important operator in the original GA implementations. It was only in later implementations that the explorative power of mutation was used to improve the search capabilities of GAs.

Computational Intelligence: An Introduction, Second Edition A.P. Engelbrecht

Since the CGA, several variations of the GA have been developed that differ in representation scheme, selection operator, crossover operator, and mutation operator. Some implementations introduce other concepts from nature such as mass extinction, culling, population islands, amongst others. While it is impossible to provide a complete review of these alternatives, this chapter provides a good flavor of these approaches to illustrate the richness of GAs.

9.2 Crossover

Crossover operators can be divided into three main categories based on the arity (i.e. the number of parents used) of the operator. This results in three main classes of crossover operators:

- **asexual**, where an offspring is generated from one parent.
- **sexual**, where two parents are used to produce one or two offspring.
- **multi-recombination**, where more than two parents are used to produce one or more offspring.

Crossover operators are further categorized based on the representation scheme used. For example, binary-specific operators have been developed for binary string representations (refer to Section 9.2.1), and operators specific to floating-point representations (refer to Section 9.2.2).

Parents are selected using any of the selection schemes discussed in Section 8.5. It is, however, not a given that selected parents will mate. Recombination is applied probabilistically. Each pair (or group) of parents have a probability, p_c, of producing offspring. Usually, a high crossover probability (also referred to as the crossover rate) is used.

In selection of parents, the following issues need to be considered:

- Due to probabilistic selection of parents, it may happen that the same individual is selected as both parents, in which case the generated offspring will be a copy of the parent. The parent selection process should therefore incorporate a test to prevent such unnecessary operations.
- It is also possible that the same individual takes part in more than one application of the crossover operator. This becomes a problem when fitness-proportional selection schemes are used.

In addition to parent selection and the recombination process, the crossover operator considers a replacement policy. If one offspring is generated, the offspring may replace the worst parent. Such replacement can be based on the restriction that the offspring must be more fit than the worst parent, or it may be forced. Alternatively, Boltzmann selection (refer to Section 8.5.6) can be used to decide if the offspring should replace the worst parent. Crossover operators have also been implemented where the offspring replaces the worst individual of the population. In the case of two offspring, similar replacement strategies can be used.

9.2.1 Binary Representations

Most of the crossover operators for binary representations are sexual, being applied to two selected parents. If $\mathbf{x}_1(t)$ and $\mathbf{x}_2(t)$ denote the two selected parents, then the recombination process is summarized in Algorithm 9.1. In this algorithm, $\mathbf{m}(t)$ is a mask that specifies which bits of the parents should be swapped to generate the offspring, $\tilde{\mathbf{x}}_1(t)$ and $\tilde{\mathbf{x}}_2(t)$. Several crossover operators have been developed to compute the mask:

- **One-point crossover:** Holland [376] suggested that segments of genes be swapped between the parents to create their offspring, and not single genes. A one-point crossover operator was developed that randomly selects a crossover point, and the bitstrings after that point are swapped between the two parents. One-point crossover is illustrated in Figure 9.1(a). The mask is computed using Algorithm 9.2.
- **Two-point crossover:** In this case two bit positions are randomly selected, and the bitstrings between these points are swapped as illustrated in Figure 9.1(b). The mask is calculated using Algorithm 9.3. This operator can be generalized to an n-point crossover [85, 191, 250, 711].
- **Uniform crossover:** The n_x-dimensional mask is created randomly [10, 828] as summarized in Algorithm 9.4. Here, p_x is the bit-swapping probability. If $p_x = 0.5$, then each bit has an equal chance to be swapped. Uniform crossover is illustrated in Figure 9.1(c).

Algorithm 9.1 Generic Algorithm for Bitstring Crossover

Let $\tilde{\mathbf{x}}_1(t) = \mathbf{x}_1(t)$ and $\tilde{\mathbf{x}}_2(t) = \mathbf{x}_2(t)$;
if $U(0,1) \leq p_c$ **then**
    Compute the binary mask, $\mathbf{m}(t)$;
    for $j = 1, \ldots, n_x$ **do**
        if $m_j = 1$ **then**
            //swap the bits
            $\tilde{\mathbf{x}}_{1j}(t) = \mathbf{x}_{2j}(t)$;
            $\tilde{\mathbf{x}}_{2j}(t) = \mathbf{x}_{1j}(t)$;
        end
    end
end

Algorithm 9.2 One-Point Crossover Mask Calculation

Select the crossover point, $\xi \sim U(1, n_x - 1)$;
Initialize the mask: $m_j(t) = 0$, for all $j = 1, \ldots, n_x$;
for $j = \xi + 1$ *to* n_x **do**
    $m_j(t) = 1$;
end

Algorithm 9.3 Two-Point Crossover Mask Calculation

Select the two crossover points, $\xi_1, \xi_2 \sim U(1, n_x)$;
Initialize the mask: $m_j(t) = 0$, for all $j = 1, \ldots, n_x$;
for $j = \xi_1 + 1$ *to* ξ_2 **do**
    $m_j(t) = 1$;
end

Algorithm 9.4 Uniform Crossover Mask Calculation

Initialize the mask: $m_j(t) = 0$, for all $j = 1, \ldots, n_x$;
for $j = 1$ *to* n_x **do**
    if $U(0, 1) \leq p_x$ **then**
        $m_j(t) = 1$;
    end
end

Bremermann *et al.* [85] proposed the first multi-parent crossover operators for binary representations. Given n_μ parent vectors, $\mathbf{x}_1(t), \ldots, \mathbf{x}_{n_\mu}(t)$, majority mating generates one offspring using

$$\tilde{\mathbf{x}}_{ij}(t) = \begin{cases} 0 & \text{if } n_\mu^{'} \geq n_\mu/2,\ l = 1, \ldots, n_\mu \\ 1 & \text{otherwise} \end{cases} \tag{9.1}$$

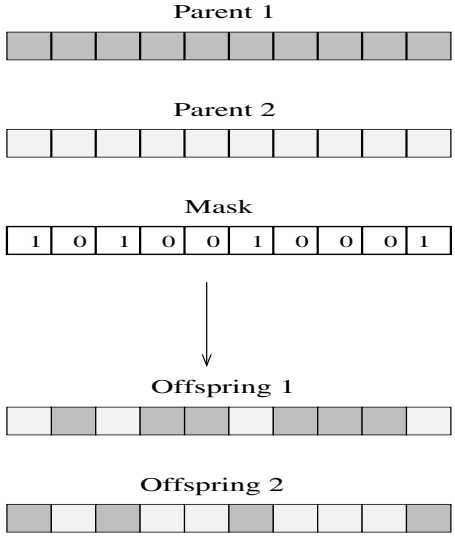
where $n_\mu^{'}$ is the number of parents with $x_{lj}(t) = 0$.

A multiparent version of n-point crossover was also proposed by Bremermann *et al.* [85], where $n_\mu - 1$ identical crossover points are selected in the n_μ parents. One offspring is generated by selecting one segment from each parent.

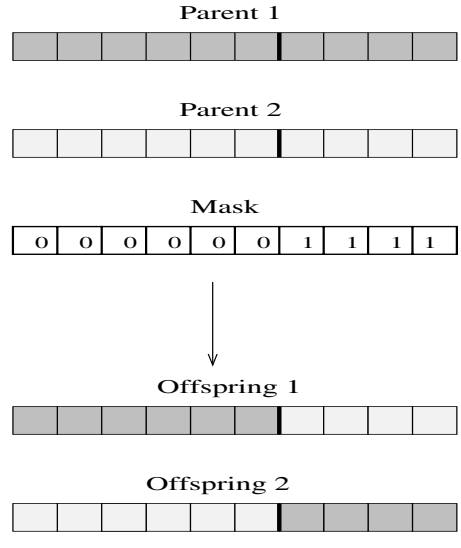
Jones [427] developed a crossover hillclimbing operator that can be applied to any representation. Crossover hillclimbing starts with two parents, and continues to produce offspring from this pair of parents until either a maximum number of crossover attempts has been exceeded, or a pair of offspring is found where one of the offspring has a better fitness than the best parent. Crossover hillclimbing then continues reproduction using these two offspring as the new parent pair. If a better parent pair cannot be found within the specified time limit, the worst parent is replaced by a randomly selected parent.

9.2.2 Floating-Point Representation

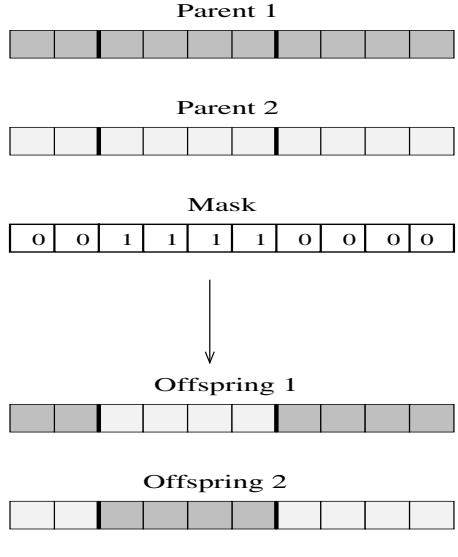
The crossover operators discussed above (excluding majority mating) can also be applied to floating-point representations as discrete recombination strategies. In contrast



(a) Uniform Crossover



(b) One-point Crossover



(c) Two-point Crossover

Figure 9.1 Crossover Operators for Binary Representations

to these discrete operators where information is swapped between parents, intermediate recombination operators, developed specifically for floating-point representations, blend components across the selected parents.

One of the first floating-point crossover operators is the linear operator proposed by Wright [918]. From the parents, $\mathbf{x}_1(t)$ and $\mathbf{x}_2(t)$, three candidate offspring are generated as $(\mathbf{x}_1(t) + \mathbf{x}_2(t)), (1.5\mathbf{x}_1(t) - 0.5\mathbf{x}_2(t))$ and $(-0.5\mathbf{x}_1(t) + 1.5\mathbf{x}_2(t))$. The two best solutions are selected as the offspring. Wright [918] also proposed a directional heuristic crossover operator where one offspring is created from two parents using

$$\tilde{x}_{ij}(t) = U(0,1)(\mathbf{x}_{2j}(t) - \mathbf{x}_{1j}(t)) + \mathbf{x}_{2j}(t) \tag{9.2}$$

subject to the constraint that parent $\mathbf{x}_2(t)$ cannot be worse than parent $\mathbf{x}_1(t)$.

Michalewicz [586] coined the arithmetic crossover, which is a multiparent recombination strategy that takes a weighted average over two or more parents. One offspring is generated using

$$\tilde{x}_{ij}(t) = \sum_{l=1}^{n_\mu} \gamma_l x_{lj}(t) \tag{9.3}$$

with $\sum_{l=1}^{n_\mu} \gamma_l = 1$. A specialization of the arithmetic crossover operator is obtained for $n_\mu = 2$, in which case

$$\tilde{x}_{ij}(t) = (1-\gamma)x_{1j}(t) + \gamma x_{2j}(t) \tag{9.4}$$

with $\gamma \in [0,1]$. If $\gamma = 0.5$, the effect is that each component of the offspring is simply the average of the corresponding components of the parents.

Eshelman and Schaffer [251] developed a variation of the weighted average given in equation (9.4), referred to as the blend crossover (BLX-α), where

$$\tilde{x}_{ij}(t) = (1-\gamma_j)x_{1j}(t) + \gamma_j x_{2j}(t) \tag{9.5}$$

with $\gamma_j = (1+2\alpha)U(0,1) - \alpha$. The BLX-$\alpha$ operator randomly picks, for each component, a random value in the range

$$[x_{1j}(t) - \alpha(x_{2j}(t) - x_{1j}(t)), x_{2j}(t) + \alpha(x_{2j}(t) - x_{1j}(t))] \tag{9.6}$$

BLX-α assumes that $x_{1j}(t) < x_{2j}(t)$. Eshelman and Schaffer found that $\alpha = 0.5$ works well.

The BLX-α has the property that the location of the offspring depends on the distance that the parents are from one another. If this distance is large, then the distance between the offpsring and its parents will be large. The BLX-α allows a bit more exploration than the weighted average of equation (9.3), due to the stochastic component in producing the offspring.

Michalewicz *et al.* [590] developed the two-parent geometrical crossover to produce a single offspring as follows:

$$\tilde{x}_{ij}(t) = (x_{1j}x_{2j})^{0.5} \tag{9.7}$$

The geometrical crossover can be generalized to multi-parent recombination as follows:

$$\tilde{x}_{ij}(t) = (x_{1j}^{\alpha_1} x_{2j}^{\alpha_2} \dots x_{n_\mu j}^{\alpha_{n_\mu}}) \tag{9.8}$$

where n_μ is the number of parents, and $\sum_{l=1}^{n_\mu} \alpha_l = 1$.

Deb and Agrawal [196] developed the simulated binary crossover (SBX) to simulate the behavior of the one-point crossover operator for binary representations. Two parents, $\mathbf{x}_1(t)$ and $\mathbf{x}_2(t)$ are used to produce two offspring, where for $j = 1, \dots, n_x$

$$\tilde{x}_{1j}(t) = 0.5[(1+\gamma_j)x_{1j}(t) + (1-\gamma_j)x_{2j}(t)] \tag{9.9}$$

$$\tilde{x}_{2j}(t) = 0.5[(1-\gamma_j)x_{1j}(t) + (1+\gamma_j)x_{2j}(t)] \tag{9.10}$$

where

$$\gamma_j = \begin{cases} (2r_j)^{\frac{1}{\eta+1}} & \text{if } r_j \leq 0.5 \\ \left(\frac{1}{2(1-r_j)}\right)^{\frac{1}{\eta+1}} & \text{otherwise} \end{cases} \tag{9.11}$$

where $r_j \sim U(0,1)$, and $\eta > 0$ is the distribution index. Deb and Agrawal suggested that $\eta = 1$.

The SBX operator generates offspring symmetrically about the parents, which prevents bias towards any of the parents. For large values of η there is a higher probability that offspring will be created near the parents. For small η values, offspring will be more distant from the parents.

While the above focused on sexual crossover operators (some of which can also be extended to multiparent operators), the remainder of this section considers a number of multiparent crossover operators. The main objective of these multiparent operators is to intensify the explorative capabilities compared to two-parent operators. By aggregating information from multiple parents, more disruption is achieved with the resemblance between offspring and parents on average smaller compared to two-parent operators.

Ono and Kobayashi [642] developed the unimodal distributed (UNDX) operator where two or more offspring are generated using three parents. The offspring are created from an ellipsoidal probability distribution, with one of the axes formed along the line that connects two of the parents. The extent of the orthogonal direction is determined from the perpendicular distance of the third parent from the axis. The UNDX operator can be generalized to work with any number of parents, with $3 \leq n_\mu \leq n_s$. For the generalization, $n_\mu - 1$ parents are randomly selected and their center of mass (mean), $\overline{\mathbf{x}}(t)$, is calculated, where

$$\overline{x}_j(t) = \sum_{l=1}^{n_\mu - 1} x_{lj}(t) \tag{9.12}$$

From the mean, $n_\mu - 1$ direction vectors, $\mathbf{d}_l(t) = \mathbf{x}_l(t) - \overline{\mathbf{x}}(t)$ are computed, for $l = 1, \dots, n_\mu - 1$. Using the direction vectors, the direction cosines are computed as $\mathbf{e}_l(t) = \mathbf{d}_l(t)/|\mathbf{d}_l(t)|$, where $|\mathbf{d}_l(t)|$ is the length of vector $\mathbf{d}_l(t)$. A random parent, with index n_μ is selected. Let $\mathbf{x}_{n_\mu}(t) - \overline{\mathbf{x}}(t)$ be the vector orthogonal to all $\mathbf{e}_l(t)$, and

$\delta = |\mathbf{x}_{n_\mu}(t) - \overline{\mathbf{x}}(t)|$. Let $\mathbf{e}_l(t), l = n_\mu, \ldots, n_s$ be the orthonormal basis of the subspace orthogonal to the subspace spanned by the direction cosines, $\mathbf{e}_l(t),\ l = 1, \ldots, n_\mu - 1$. Offspring are then generated using

$$\tilde{\mathbf{x}}_i(t) = \overline{\mathbf{x}}(t) + \sum_{l=1}^{n_\mu - 1} N(0, \sigma_1^2)|\mathbf{d}_l|\mathbf{e}_l + \sum_{l=n_\mu}^{n_s} N(0, \sigma_2^2)\delta\mathbf{e}_l(t) \tag{9.13}$$

where $\sigma_1 = \frac{1}{\sqrt{n_\mu - 2}}$ and $\sigma_2 = \frac{0.35}{\sqrt{n_s - n_\mu - 2}}$.

Using equation (9.13) any number of offspring can be created, sampled around the center of mass of the selected parents. A higher probability is assigned to create offspring near the center rather than near the parents. The effect of the UNDX operator is illustrated in Figure 9.2(a) for $n_\mu = 4$.

Tsutsui and Goldberg [857] and Renders and Bersini [714] proposed the simplex crossover (SPX) operator as another center of mass approach to recombination. Renders and Bersini selects $n_\mu > 2$ parents, and determines the best and worst parent, say $\mathbf{x}_1(t)$ and $\mathbf{x}_2(t)$ respectively. The center of mass, $\overline{\mathbf{x}}(t)$ is computed over the selected parents, but with $\mathbf{x}_2(t)$ excluded. One offspring is generated using

$$\tilde{\mathbf{x}}(t) = \overline{\mathbf{x}}(t) + (\mathbf{x}_1(t) - \mathbf{x}_2(t)) \tag{9.14}$$

Tsutsui and Goldberg followed a similar approach, selecting $n_\mu = n_x + 1$ parents independent from one another for an n_x-dimensional search space. These n_μ parents form a simplex. The simplex is expanded in each of the n_μ directions, and offspring sampled from the expanded simplex as illustrated in Figure 9.2(b). For $n_x = 2$, $n_\mu = 3$, and

$$\overline{\mathbf{x}}(t) = \sum_{l=1}^{n_\mu} \mathbf{x}_l(t) \tag{9.15}$$

the expanded simplex is defined by the points

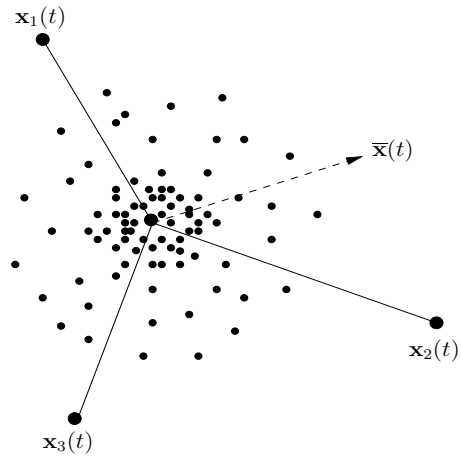
$$(1 + \gamma)(\mathbf{x}_l(t) - \overline{\mathbf{x}}(t)) \tag{9.16}$$

for $l = 1, \ldots, n_\mu = 3$ and $\gamma \geq 0$. Offspring are obtained by uniform sampling of the expanded simplex.

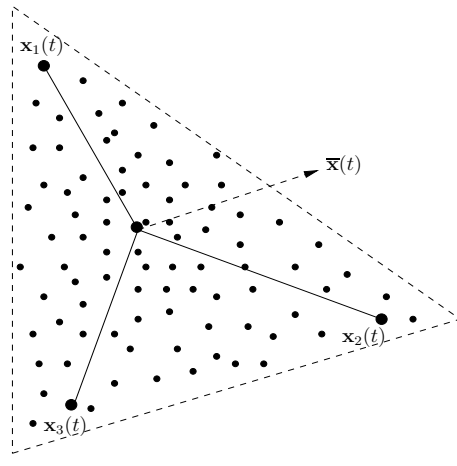
Deb *et al.* [198] proposed a variation of the UNDX operator, which they refer to as parent-centric crossover (PCX). Instead of generating offspring around the center of mass of the selected parents, offspring are generated around selected parents. PCX selects n_μ parents and computes their center of mass, $\overline{\mathbf{x}}(t)$. For each offspring to be generated one parent is selected uniformly from the n_μ parents. A direction vector is calculated for each offspring as

$$\mathbf{d}_i(t) = \mathbf{x}_i(t) - \overline{\mathbf{x}}(t)$$

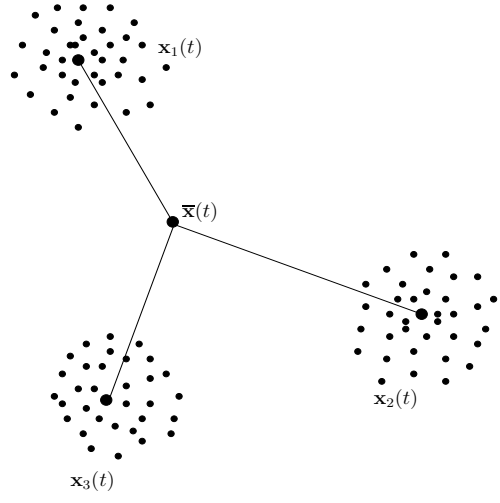
where $\mathbf{x}_i(t)$ is the randomly selected parent. From the other $n_\mu - 1$ parents perpendicular distances, δ_l, for $i \neq l = 1, \ldots, n_\mu$, are calculated to the line $\mathbf{d}_i(t)$. The average



(a) UNDX Operator



(b) SPX Operator



(c) PCX Operator

Figure 9.2 Illustration of Multi-parent Center of Mass Crossover Operators (dots represent potential offpsring)

over these distances is calculated, i.e.

$$\overline{\delta} = \frac{\sum_{l=1,l\neq i}^{n_\mu} \delta_l}{n_\mu - 1} \tag{9.17}$$

Offspring is generated using

$$\tilde{\mathbf{x}}_i(t) = \mathbf{x}_i(t) + N(0,\sigma_1^2)|\mathbf{d}_i(t)| + \sum_{l=1,i\neq l}^{n_\mu} N(0,\sigma_2^2)\overline{\delta}\mathbf{e}_l(t) \tag{9.18}$$

where $\mathbf{x}_i(t)$ is the randomly selected parent of offspring $\tilde{\mathbf{x}}_i(t)$, and $\mathbf{e}_l(t)$ are the $n_\mu - 1$

orthonormal bases that span the subspace perpendicular to $\mathbf{d}_i(t)$.

The effect of the PCX operator is illustrated in Figure 9.2(c).

Eiben *et al.* [231, 232, 233] developed a number of gene scanning techniques as multi-parent generalizations of n-point crossover. For each offspring to be created, the gene scanning operator is applied as summarized in Algorithm 9.5. The algorithm contains two main procedures:

- A scanning strategy, which assigns to each selected parent a probability that the offspring will inherit the next component from that parent. The component under consideration is indicated by a marker.
- A marker update strategy, which updates the markers of parents to point to the next component of each parent.

Marker initialization and updates depend on the representation method. For binary representations the marker of each parent is set to its first gene. The marker update strategy simply advances the marker to the next gene.

Eiben *et al.* proposed three scanning strategies:

- **Uniform scanning** creates only one offspring. The probability, $p_s(\mathbf{x}_l(t))$, of inheriting the gene from parent $\mathbf{x}_l(t), l = 1, \ldots, n_\mu$, as indicated by the marker of that parent is computed as

$$p_s(\mathbf{x}_l(t+1)) = \frac{1}{n_\mu} \tag{9.19}$$

  Each parent has an equal probability of contributing to the creation of the offspring.
- **Occurrence-based scanning** bases inheritance on the premise that the allele that occur most in the parents for a particular gene is the best possible allele to inherit by the offspring (similar to the majority mating operator). Occurrence-based scanning assumes that fitness-proportional selection is used to select the n_μ parents that take part in recombination.
- **Fitness-based scanning**, where the allele to be inherited is selected proportional to the fitness of the parents. Considering maximization, the probability to inherit from parent $\mathbf{x}_l(t)$ is

$$p_s(\mathbf{x}_l(t)) = \frac{f(\mathbf{x}_l(t))}{\sum_{i=1}^{n_\mu} f(\mathbf{x}_i(t))} \tag{9.20}$$

  Roulette-wheel selection is used to select the parent to inherit from.

For each of these scanning strategies, the offspring inherits $p_s(\mathbf{x}_l(t+1))n_x$ genes from parent $\mathbf{x}_l(t)$.

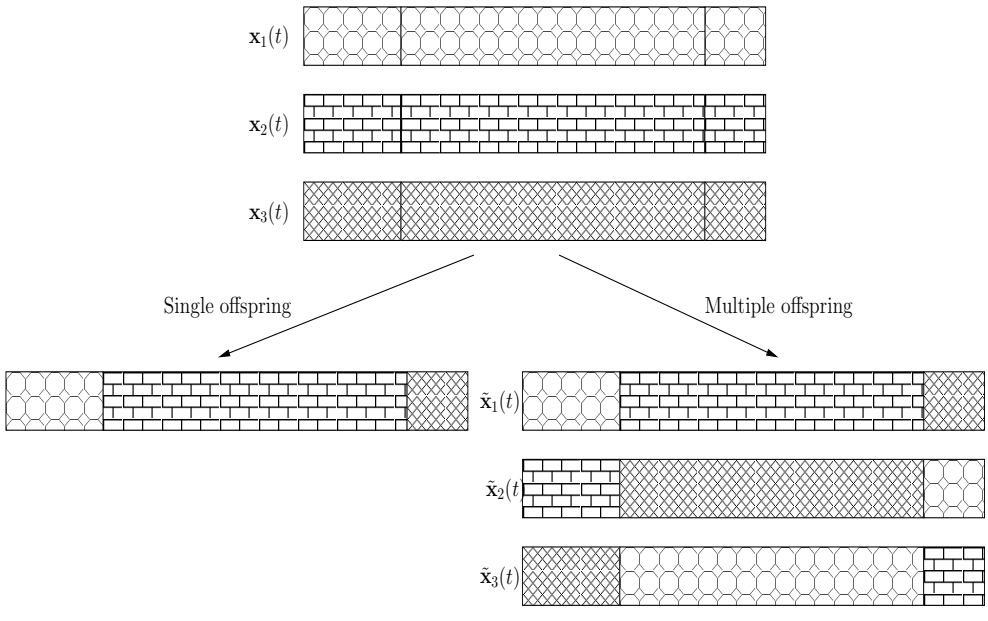


Figure 9.3 Diagonal Crossover

Algorithm 9.5 Gene Scanning Crossover Operator

Initialize parent markers;
for $j = 1, \ldots, n_x$ **do**
    Select the parent, $\mathbf{x}_l(t)$, to inherit from;
    $\tilde{x}_j(t) = x_{lj}(t)$;
    Update parent markers;
end

The diagonal crossover operator developed by Eiben *et al.* [232] is a generalization of n-point crossover for more than two parents: $n \geq 1$ crossover points are selected and applied to all of the $n_\mu = n + 1$ parents. One or $n + 1$ offspring can be generated by selecting segments from the parents along the diagonals as illustrated in Figure 9.3, for $n = 2, n_\mu = 3$.

9.3 Mutation

The aim of mutation is to introduce new genetic material into an existing individual; that is, to add diversity to the genetic characteristics of the population. Mutation is used in support of crossover to ensure that the full range of allele is accessible for each gene. Mutation is applied at a certain probability, p_m, to each gene of the offspring, $\tilde{\mathbf{x}}_i(t)$, to produce the mutated offspring, $\mathbf{x}_i^{'}(t)$. The mutation probability, also referred

to as the mutation rate, is usually a small value, $p_m \in [0, 1]$, to ensure that good solutions are not distorted too much.

Given that each gene is mutated at probability p_m, the probability that an individual will be mutated is given by

$$Prob(\tilde{\mathbf{x}}_i(t) \text{ is mutated}) = 1 - (1 - p_m)^{n_x} \tag{9.21}$$

where the individual contains n_x genes.

Assuming binary representations, if $H(\tilde{\mathbf{x}}_i(t), \mathbf{x}_i^{'}(t))$ is the Hamming distance between offspring, $\tilde{\mathbf{x}}_i(t)$, and its mutated version, $\mathbf{x}_i^{'}(t)$, then the probability that the mutated version resembles the original offspring is given by

$$Prob(\mathbf{x}_i^{'}(t)) \approx \tilde{\mathbf{x}}_i(t)) = p_m^{H(\tilde{\mathbf{x}}_i(t), \mathbf{x}_i^{'}(t))}(1 - p_m)^{n_x - H(\tilde{\mathbf{x}}_i(t), \mathbf{x}_i^{'}(t))} \tag{9.22}$$

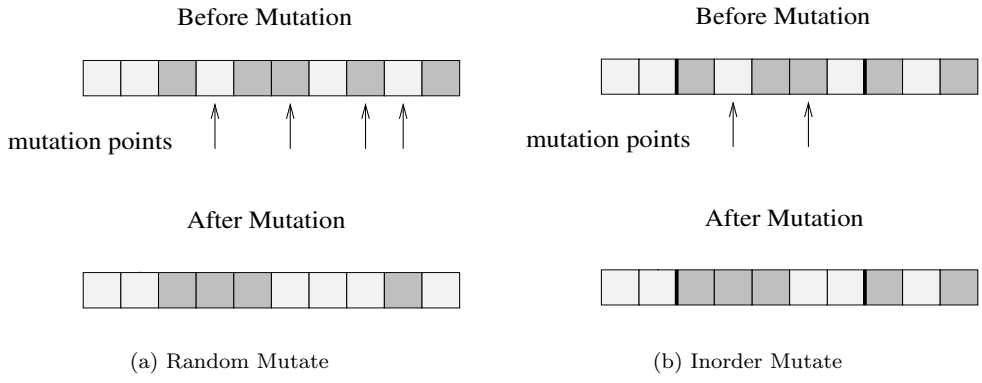
This section describes mutation operators for binary and floating-point representations in Sections 9.3.1 and 9.3.2 respectively. A macromutation operator is described in Section 9.3.3.

9.3.1 Binary Representations

For binary representations, the following mutation operators have been developed:

- **Uniform (random) mutation** [376], where bit positions are chosen randomly and the corresponding bit values negated as illustrated in Figure 9.4(a). Uniform mutation is summarized in Algorithm 9.6.
- **Inorder mutation**, where two mutation points are randomly selected and only the bits between these mutation points undergo random mutation. Inorder mutation is illustrated in Figure 9.4(b) and summarized in Algorithm 9.7.
- **Gaussian mutation:** For binary representations of floating-point decision variables, Hinterding [366] proposed that the bitstring that represents a decision variable be converted back to a floating-point value and mutated with Gaussian noise. For each chromosome random numbers are drawn from a Poisson distribution to determine the genes to be mutated. The bitstrings representing these genes are then converted. To each of the floating-point values is added the stepsize $N(0, \sigma_j)$, where σ_j is 0.1 of the range of that decision variable. The mutated floating-point value is then converted back to a bitstring. Hinterding showed that Gaussian mutation on the floating-point representation of decision variables provided superior results to bit flipping.

For large dimensional bitstrings, mutation may significantly add to the computational cost of the GA. In a bid to reduce computational complexity, Birru [69] divided the bitstring of each individual into a number of bins. The mutation probability is applied to the bins, and if a bin is to be mutated, one of its bits are randomly selected and flipped.



(a) Random Mutate (b) Inorder Mutate

Figure 9.4 Mutation Operators for Binary Representations

Algorithm 9.6 Uniform/Random Mutation

for $j = 1, \ldots, n_x$ **do**
  if $U(0,1) \leq p_m$ **then**
    $x_{ij}^{'}(t) = \neg \tilde{x}_{ij}(t)$, where $\neg$ denotes the boolean NOT operator;
  end
end

Algorithm 9.7 Inorder Mutation

Select mutation points, $\xi_1, \xi_2 \sim U(1, \ldots, n_x)$;
for $j = \xi_1, \ldots, \xi_2$ **do**
  if $U(0,1) \leq p_m$ **then**
    $x_{ij}^{'}(t) = \neg \tilde{x}_{ij}(t)$;
  end
end

9.3.2 Floating-Point Representations

As indicated by Hinterding [366] and Michalewicz [586], better performance is obtained by using a floating-point representation when decision variables are floating-point values and by applying appropriate operators to these representations, than to convert to a binary representation. This resulted in the development of mutation operators for floating-point representations. One of the first proposals was a uniform mutation, where [586]

$$x_{ij}^{'}(t) = \begin{cases} \tilde{x}_{ij}(t) + \Delta(t, x_{max,j} - \tilde{x}_{ij}(t)) & \text{if a random digit is 0} \\ \tilde{x}_{ij}(t) + \Delta(t, \tilde{x}_{ij}(t) - x_{min,j}(t)) & \text{if a random digit is 1} \end{cases} \tag{9.23}$$

where $\Delta(t, x)$ returns random values from the range $[0, x]$.

Any of the mutation operators discussed in Sections 11.2.1 (for EP) and 12.4.3 (for ES) can be applied to GAs.

9.3.3 Macromutation Operator – Headless Chicken

Jones [427] proposed a macromutation operator, referred to as the headless chicken operator. This operator creates an offspring by recombining a parent individual with a randomly generated individual using any of the previously discussed crossover operators. Although crossover is used to combine an individual with a randomly generated individual, the process cannot be referred to as a crossover operator, as the concept of inheritence does not exist. The operator is rather considered as mutation due to the introduction of new randomly generated genetic material.

9.4 Control Parameters

In addition to the population size, the performance of a GA is influenced by the mutation rate, p_m, and the crossover rate, p_c. In early GA studies very low values for p_m and relatively high values for p_c are propagated. Usually, the values for p_m and p_c are kept static. It is, however, widely accepted that these parameters have a significant influence on performance, and that optimal settings for p_m and p_c can significantly improve performance. To obtain such optimal settings through empirical parameter tuning is a time consuming process. A solution to the problem of finding best values for these control parameters is to use dynanically changing parameters.

Although dynamic, and self-adjusting parameters have been used for EP and ES (refer to Sections 11.3 and 12.3) as early as the 1960s, Fogarty [264] provided one of the first studies of dynamically changing mutation rates for GAs. In this study, Fogarty concluded that performance can be significantly improved using dynamic mutation rates. Fogarty used the following schedules where the mutation rate exponentially decreases with generation number:

$$p_m(t) = \frac{1}{240} + \frac{0.11375}{2^t} \tag{9.24}$$

As an alternative, Fogarty also proposed for binary representations a mutation rate per bit, $j = 1, \ldots, n_b$, where n_b indicates the least significant bit:

$$p_m(j) = \frac{0.3528}{2^{j-1}} \tag{9.25}$$

The two schedules above were combined to give

$$p_m(j,t) = \frac{28}{1905 \times 2^{j-1}} = \frac{0.4026}{2^{t+j-1}} \tag{9.26}$$

A large initial mutation rate favors exploration in the initial steps of the search, and with a decrease in mutation rate as the generation number increases, exploitation

is facilitated. Different schedules can be used to reduce the mutation rate. The schedule above results in an exponential decrease. An alternative may be to use a linear schedule, which will result in a slower decrease in p_m, allowing more exploration. However, a slower decrease may be too disruptive for already found good solutions. A good strategy is to base the probability of being mutated on the fitness of the individual: the more fit the individual is, the lower the probability that its genes will be mutated; the more unfit the individual, the higher the probability of mutation.

Annealing schedules similar to those used for the learning rate of NNs (refer to equation (4.40)), and to adjust control parameters for PSO and ACO can be applied to p_m (also refer to Section A.5.2).

For floating-point representations, performance is also influenced by the mutational step sizes. An ideal strategy is to start with large mutational step sizes to allow larger, stochastic jumps within the search space. The step sizes are then reduced over time, so that very small changes result near the end of the search process. Step sizes can also be proportional to the fitness of an individual, with unfit individuals having larger step sizes than fit individuals. As an alternative to deterministic schedules to adapt step sizes, self-adaptation strategies as for EP and ES can be used (refer to Sections 11.3.3 and 12.3.3).

The crossover rate, p_c, also bears significant influence on performance. With its optimal value being problem dependent, the same adaptive strategies as for p_m can be used to dynamically adjust p_c.

In addition to p_m (and mutational step sizes in the case of floating-point representations) and p_c, the choice of the best evolutionary operators to use is also problem dependent. While a best combination of crossover, mutation, and selection operators together with best values for the control parameters can be obtained via empirical studies, a number of adaptive methods can be found as reviewed in [41]. These methods adaptively switch between different operators based on search progress. Ultimately, finding the best set of operators and control parameter values is a multi-objective optimization problem by itself.

9.5 Genetic Algorithm Variants

Based on the general GA, different implementations of a GA can be obtained by using different combinations of selection, crossover, and mutation operators. Although different operator combinations result in different behaviors, the same algorithmic flow as given in Algorithm 8.1 is followed. This section discusses a few GA implementations that deviate from the flow given in Algorithm 8.1. Section 9.5.1 discusses generation gap methods. The messy GA is described in Section 9.5.2. A short discussion on interactive evolution is given in Section 9.5.3. Island (or parallel) GAs are discussed in Section 9.5.4.

9.5.1 Generation Gap Methods

The GAs as discussed thus far differ from biological models of evolution in that population sizes are fixed. This allows the selection process to be described by two steps:

- Parent selection, and
- a replacement strategy that decides if offspring will replace parents, and which parents to replace.

Two main classes of GAs are identified based on the replacement strategy used, namely generational genetic algorithms (GGA) and steady state genetic algorithms (SSGA), also referred to as incremental GAs. For GGAs the replacement strategy replaces all parents with their offspring after all offpsring have been created and mutated. This results in no overlap between the current population and the new population (assuming that elitism is not used). For SSGAs, a decision is made immediately after an offspring is created and mutated as to whether the parent or the offspring survives to the next generation. Thus, there exists an overlap between the current and new populations.

The amount of overlap between the current and new populations is referred to as the generation gap [191]. GGAs have a zero generation gap, while SSGAs generally have large generation gaps.

A number of replacement strategies have been developed for SSGAs:

- **Replace worst** [192], where the offspring replaces the worst individual of the current population.
- **Replace random** [192, 829], where the offspring replaces a randomly selected individual of the current population.
- **Kill tournament** [798], where a group of individuals is randomly selected, and the worst individual of this group is replaced with the offspring. Alternatively, a tournament size of two is used, and the worst individual is replaced with a probability, $0.5 \leq p_r \leq 1$.
- **Replace oldest**, where a first-in-first-out strategy is followed by replacing the oldest individual of the current population. This strategy has a high probability of replacing one of the best individuals.
- **Conservative selection** [798] combines a first-in-first-out replacement strategy with a modified deterministic binary tournament selection. A tournament size of two individuals is used of which one is always the oldest individual of the current population. The worst of the two is replaced by the offspring. This approach ensures that the oldest individual will not be lost if it is the fittest.
- **Elitist** strategies of the above replacement strategies have also been developed, where the best individual is excluded from selection.
- **Parent-offspring** competition, where a selection strategy is used to decide if an offspring replaces one of its own parents.

Theoretical and empirical studies of steady state GAs can be found in [734, 797, 798, 872].

9.5.2 Messy Genetic Algorithms

Standard GAs use populations where all individuals are of the same fixed size. For an n_x-dimensional search space, a standard GA finds a solution through application of the evolutionary operators to the complete n_x-dimensional individuals. It may happen that good individuals are found, but some of the genes of a good individual are non-optimal. It may be difficult to find optimal allele for such genes through application of crossover and mutation on the entire individual. It may even happen that crossover looses optimized genes, or groups of optimized genes.

Goldberg *et al.* [321, 323, 324] developed the messy GA (mGA), which finds solutions by evolving optimal building blocks and combining building blocks. Here a building block refers to a group of genes. In a messy GA individuals are of variable length, and specified by a list of position-value pairs. The position specifies the gene index, and the value specifies the allele for that gene. These pairs are referred to as messy genes. As an example, if $n_x = 4$, then the individual, $((1,0)(3,1),(4,0)(1,1))$, represents the individual $0 * 10$.

The messy representation may result in individuals that are over-specified or under-specified. The example above illustrates both cases. The individual is over-specified because gene 1 occurs twice. It is under-specified because gene 2 does not occur, and has no value assigned. Fitness evaluation of messy individuals requires strategies to cope with such individuals. For over-specified individuals, a first-come-first-served approach is followed where the first specified value is assigned to the repeating gene. For under-specified individuals, a missing gene's allele is obtained from a competitive template. The competitive template is a locally optimal solution. As an example, if 1101 is the template, the fitness of $0 * 10$ is evaluated as the fitness of 0101.

The objective of mGAs is to evolve optimal building blocks, and to incrementally combine optimized building blocks to form an optimal solution. An mGA is implemented using two loops as shown in Algorithm 9.8. The inner loop consists of three steps:

- **Initialization** to create a population of building blocks of a specified length, n_m.
- **Primordial**, which aims to generate small, promising building blocks.
- **Juxtapositional**, to combine building blocks.

The outer loop specifies the size of the building blocks to be considered, starting with the smallest size of one, and incrementally increasing the size until a maximum size is reached, or an acceptable solution is found. The outer loop also sets the best solution obtained from the juxtaposition step as the competitive template for the next iteration.

Algorithm 9.8 Messy Genetic Algorithm

Initialize the competitive template;
for $n_m = 1$ *to* $n_{m,max}$ **do**
  Initialize the population to contain building blocks of size n_m;
  Apply the primordial step;
  Apply the juxtaposition step;
  Set the competitive template to the best solution from the juxtaposition step;
end

The initialization step creates all possible combinations of building blocks of length n_m. For n_x-dimensional solutions, this results in a population size of

$$n_s = 2^{n_m} \left(\begin{array}{c} n_x \\ n_m \end{array} \right) \tag{9.27}$$

where

$$\left(\begin{array}{c} n_x \\ n_m \end{array} \right) = \frac{n_x!}{n_m!(n_x - n_m)!} \tag{9.28}$$

This leads to one of the major disadvantages of mGAs, in that computational complexity explodes with increase in n_m (i.e. building block size). The fast mGA addresses this problem by starting with larger building block sizes and adding a gene deletion operator to the primordial step to prune building blocks [322].

The primordial step is executed for a specified number of generations, applying only selection to find the best building blocks. At regular intervals the population is halved, with the worst individuals (building blocks) discarded. No crossover or mutation is used. While any selection operator can be used, fitness proportional selection is usually used. Because individuals in an mGA may contain different sets of genes (as specified by the building blocks), thresholding selection has been proposed to apply selection to "similar" individuals. Thresholding selection applies tournament selection between two individuals that have in common a number of genes greater than a specified threshold. The effect achieved via the primordial step is that poor building blocks are eliminated, while good building blocks survive to the juxtaposition step.

The juxtaposition step applies cut and splice operators. The cut operator is applied to selected individuals at a probability proportional to the length of the individual (i.e. the size of the building block). The objective of the cut operator is to reduce the size of building blocks by splitting the individual at a randomly selected gene. The splicing operator combines two individuals to form a larger building block. Since the probability of cutting is proportional to the length of the individual, and the mGA starts with small building blocks, splicing occurs more in the beginning. As n_m increases, cutting occurs more. Cutting and splicing then resembles crossover.

9.5.3 Interactive Evolution

In standard GAs (and all EAs for that matter), the human user plays a passive role. Selection is based on an explicitly defined analytical function, used to quantify the quality of a candidate solution. It is, however, the case that such a function cannot be defined for certain application areas, for example, evolving art, music, animations, etc. For such application areas subjective judgment is needed, based on human intuition, aesthetical values or taste. This requires interaction of a human evaluator as the "fitness function".

Interactive evolution (IE) [48, 179, 792] involves a human user online into the selection and variation processes. The search process is now directed through interactive selection of solutions by the human user instead of an absolute fitness function. Dawkins [179] was the first to consider IE to evolve biomorphs, which are tree-like representations of two-dimensional graphical forms. Todd and Latham [849] used IE to evolve computer sculptures. Sims [792] provides further advances in the application of IE to evolve complex simulated structures, textures, and motions.

Algorithm 9.9 provides a summary of the standard IE algorithm. The main component of the IE algorithm is the interactive selection step. This step requires that the phenotype of individuals be generated from the genotype, and visualized. Based on the visual representations of candidate solutions, the user selects those individuals that will take part in reproduction, and that will survive to the next generation. Some kind of fitness function can be defined (if possible) to order candidate solutions and to perform a pre-selection to reduce the number of solutions to be evaluated by the human user.

In addition to act as the selection mechanism, the user can also interactively specify the reproduction operators and population parameters.

Instead of the human user performing selection, interaction may be of the form where the user assigns a fitness score to individuals. Automatic selection is then applied, using these user assigned quality measures.

Algorithm 9.9 Interactive Evolution Algorithm

Set the generation counter, $t = 0$;
Initialize the control parameters;
Create and initialize the population, $\mathcal{C}(0)$, of n_s individuals;
while *stopping condition(s) not true* **do**
    Determine reproduction operators, either automatically or via interaction;
    Select parents via interaction;
    Perform crossover to produce offspring;
    Mutate offspring;
    Select new population via interaction;
end

Although the section on IE is provided as part of the chapter on GAs, IE can be applied to any of the EAs.

9.5.4 Island Genetic Algorithms

GAs lend themselves to parallel implementation. Three main categories of parallel GA have been identified [100]:

- Single-population master-slave GAs, where the evaluation of fitness is distributed over several processors.
- Single-population fine-grained GAs, where each individual is assigned to one processor, and each processor is assigned only one individual. A small neighborhood is defined for each individual, and selection and reproduction are restricted to neighborhoods. Whitley [903] refers to these as cellular GAs.
- Multi-population, or island GAs, where multiple populations are used, each on a separate processor. Information is exchanged among populations via a migration policy. Although developed for parallel implementation, island GAs can be implemented on a single processor system.

The remainder of this section focuses on island GAs. In an island GA, a number of subpopulations are evolved in parallel, in a cooperative framework [335, 903, 100]. In this GA model, a number of islands occurs, where each island represents one population. Selection, crossover and mutation occur in each subpopulation independently from the other subpopulations. In addition, individuals are allowed to migrate between islands (or subpopulations), as illustrated in Figure 9.5.

An integral part of an island GA is the migration policy which governs the exchange of information between islands. A migration policy specifies [100, 102, 103, 104]:

- A **communications topology**, which determines the migration paths between islands. For example, a ring topology (such as illustrated in Figure 16.4(b)) allows exchange of information between neighboring islands. The communication topology determines how fast (or slow) good solutions disseminate to other subpopulations. For a sparsely connected structure (such as the ring topology), islands are more isolated from one another, and the spread of information about good solutions is slower. Sparse topologies also facilitate the appearance of multiple solutions. Densely connected structures have a faster spread of information, which may lead to premature convergence.
- A **migration rate**, which determines the frequency of migration. Tied with the migration rate is the question of when migration should occur. If migration occurs too early, the number of good building blocks in the migrants may be too small to have any influence at their destinations. Usually, migration occurs when each population has converged. After exchange of individuals, all populations are restarted.
- A **selection mechanism** to decide which individuals will migrate.

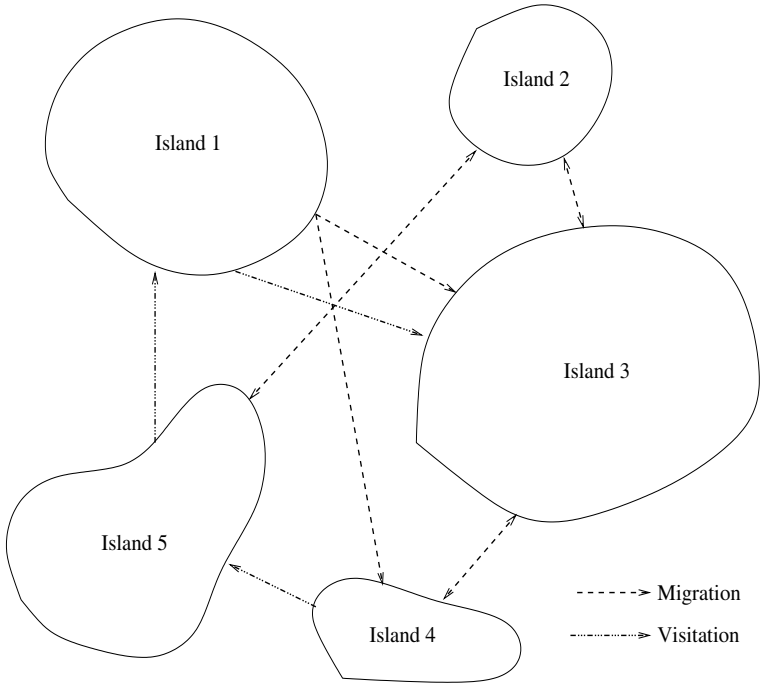


Figure 9.5 An Island GA Model

- A **replacement strategy** to decide which individual of the destination island will be replaced by the migrant.

Based on the selection and replacement strategies, island GAs can be grouped into two classes of algorithms, namely static island GAs and dynamic island GAs. For static island GAs, deterministic selection and replacement strategies are followed, for example [101],

- a good migrant replaces a bad individual,
- a good migrant replaces a randomly selected individual,
- a randomly selected migrant replaces a bad individual, or
- a randomly selected migrant replaces a randomly selected individual.

To select the good migrant, any of the fitness-proportional selection operators given in Section 8.5 can be used. For example, an elitist strategy will have the best individual of a population move to another population. Gordon [329] uses tournament selection, considering only two randomly selected individuals. The best of the two will migrate, while the worst one will be replaced by the winning individual from the neighboring population.

Dynamic models do not use a topology to determine migration paths. Instead, migration decisions are made probabilistically. Migration occurs at a specified probability.

If migration from an island does occur, the destination island is also decided probabilistically. Tournament selection may be used, based on the average fitness of the subpopulations. Additionally, an acceptance strategy can be used to decide if an immigrant should be accepted. For example, an immigrant is probabilistically accepted if its fitness is better than the average fitness of the island (using, e.g. Boltzmann selection).

Another interesting aspect to consider for island GAs is how subpopulations should be initialized. Of course a pure random approach can be used, which will cause different populations to share the same parts of the search space. A better approach would be to initialize subpopulations to cover different parts of the search space, thereby covering a larger search space and facilitating a kind of niching by individuals islands. Also, in multicriteria optimization, each subpopulation can be allocated the task to optimize one criterion. A meta-level step is then required to combine the solutions from each island (refer to Section 9.6.3).

A different kind of "island" GA is the cooperative coevolutionary GA (CCGA) of Potter [686, 687]. In this case, instead of distributing entire individuals over several subpopulations, each subpopulation is given one or a few genes (one decision variable) to optimize. The subpopulations are mutually exclusive, each having the task of evolving a single (or limited set of) gene(s). A subpopulation therefore optimizes one parameter (or a limited number of parameters) of the optimization problem. Thus, no single subpopulation has the necessary information to solve the problem itself. Rather, information of all the subpopulations must be combined to construct a solution.

Within the CCGA, a solution is constructed by adding together the best individual from each subpopulation. The main problem is how to determine the best individual of a subpopulation, since individuals do not represent complete solutions. A simple solution to this problem is to keep all other components (genes) within a complete chromosome fixed and to change just the gene that corresponds to the current subpopulation for which the best individual is sought. For each individual in the subpopulation, the value of the corresponding gene in the complete chromosome is replaced with that of the individual. Values of the other genes of the complete chromosome are usually kept fixed at the previously determined best values.

The constructed complete chromosome is then a candidate solution to the optimization problem.

It has been shown that such a cooperative approach substantially improves the accuracy of solutions, and the convergence speed compared to non-cooperative, non-coevolutionary GAs.

9.6 Advanced Topics

This section shows how GAs can be used to find multiple solutions (Section 9.6.1), to solve multi-objective optimization problems (Section 9.6.3), to cope with constraints (Section 9.6.2), and to track dynamically changing optima (Section 9.6.4). For each

of these problem types, only a few of the GA approaches are discussed.

9.6.1 Niching Genetic Algorithms

Section A.7 defines niching and different classes of niching methods. This section provides a short summary of GA implementations with the ability to locate multiple solutions to optimization problems.

Fitness Sharing

Fitness sharing is one of the earliest GA niching techniques, originally introduced as a population diversity maintenance technique [325]. It is a parallel, explicit niching approach. The algorithm regards each niche as a finite resource, and shares this resource among all individuals in the niche. Individuals are encouraged to populate a particular area of the search space by adapting their fitness based on the number of other individuals that populate the same area. The fitness $f(\mathbf{x}_i(t))$ of individual $\mathbf{x}_i$ is adapted to its shared fitness:

$$f_s(\mathbf{x}_i(t)) = \frac{f(\mathbf{x}_i(t))}{\sum_j sh(d_{ab})} \tag{9.29}$$

where $\sum_j sh(d_{ab})$ is an estimate of how crowded a niche is. A common sharing function is the triangular sharing function,

$$sh(d) = \begin{cases} 1 - (d/\sigma_{share})^\alpha & \text{if } d < \sigma_{share} \\ 0 & \text{otherwise.} \end{cases} \tag{9.30}$$

The symbol d_{ab} represents the distance between individuals $\mathbf{x}_a$ and $\mathbf{x}_b$. The distance measure may be genotypic or phenotypic, depending on the optimization problem. If the sharing function finds that d_{ab} is less than σ_{share}, it returns a value in the range $[0, 1]$, which increases as d_{ab} decreases. The more similar $\mathbf{x}_a$ and $\mathbf{x}_b$ are, the lower their individual fitnesses will become. Individuals within σ_{share} of one another will reduce each other's fitness. Sharing assumes that the number of niches can be estimated, i.e. it must be known prior to the application of the algorithm how many niches there are. It is also assumed that niches occur at least a minimum distance, $2\sigma_{share}$, from each other.

Dynamic Niche Sharing

Miller and Shaw [593] introduced dynamic niche sharing as an optimized version of fitness sharing. The same assumptions are made as with fitness sharing. Dynamic niche sharing attempts to classify individuals in a population as belonging to one of the emerging niches, or to a non-niche category. Fitness calculation for individuals belonging to the non-niche category is the same as in the standard fitness sharing technique above. The fitness of individuals found to belong to one of the developing

niches is diluted by dividing it by the size of the developing niche. Dynamically finding niches is a simple process of iterating through the population of individuals and constructing a set of non-overlapping areas in the search space. Dynamic sharing is computationally less expensive than 'normal' sharing. Miller and Shaw [593] presented results showing that dynamic sharing has improved performance when compared to fitness sharing.

Sequential Niching

Sequential niching (SN), introduced by Beasley *et al.* [55], identifies multiple solutions by adapting an optimization problem's objective function's fitness landscape through the application of a *derating* function at a position where a potential solution was found. A derating function is designed to lower the fitness appeal of previously located solutions. By repeatedly running the algorithm, all optima are removed from the fitness landscape. Sample derating functions, for a previous maximum $\mathbf{x}^*$, include:

$$G_1(\mathbf{x}, \mathbf{x}^*) = \begin{cases} \left(\frac{\|\mathbf{x}-\mathbf{x}^*\|}{R}\right)^{\alpha} & \text{if } \|\mathbf{x}-\mathbf{x}^*\| < R \\ 1 & \text{otherwise} \end{cases} \tag{9.31}$$

and

$$G_2(\mathbf{x}, \mathbf{x}^*) = \begin{cases} e^{\log m \frac{R-\|\mathbf{x}-\mathbf{x}^*\|}{R}} & \text{if } \|\mathbf{x}-\mathbf{x}^*\| < R \\ 1 & \text{otherwise} \end{cases} \tag{9.32}$$

where R is the radius of the derating function's effect. In G_1, α determines whether the derating function is concave ($\alpha > 1$) or convex ($\alpha < 1$). For $\alpha = 1$, G_1 is a linear function. For G_2, m determines 'concavity'. Noting that $\lim_{x\to 0} \log(x) = -\infty$, m must always be larger than 0. Smaller values for m result in a more concave derating function. The fitness function $f(\mathbf{x})$ is then redefined to be

$$M_{n+1}(\mathbf{x}) \equiv M_n(\mathbf{x}) \times G(\mathbf{x}, \hat{\mathbf{x}}_n) \tag{9.33}$$

where $M_0(\mathbf{x}) \equiv f(\mathbf{x})$ and $\hat{\mathbf{x}}_n$ is the best individual found during run n of the algorithm. G can be any derating function, such as G_1 and G_2.

Crowding

Crowding (or the crowding factor model), as introduced by De Jong [191], was originally devised as a diversity preservation technique. Crowding is inspired by a naturally occurring phenomenon in ecologies, namely competition amongst similar individuals for limited resources. Similar individuals compete to occupy the same ecological niche, while dissimilar individuals do not compete, as they do not occupy the same ecological niche. When a niche has reached its carrying capacity (i.e. being occupied by the maximum number of individuals that can exist within it) older individuals are replaced by newer (younger) individuals. The carrying capacity of the niche does not change, so the population size will remain constant.

For a genetic algorithm, crowding is performed as follows: It is assumed that a population of GA individuals evolve over several generational steps. At each step, the crowding algorithm selects only a portion of the current generation to reproduce. The selection strategy is fitness proportionate, i.e. more fit individuals are more likely to be chosen. After the selected individuals have reproduced, individuals in the current population are replaced by their offspring. For each offspring, a random sample is taken from the current generation, and the most similar individual is replaced by the offspring individual.

Deterministic crowding (DC) is based on De Jong's crowding technique, but with the following improvements as suggested by Mahfoud [553]:

- Phenotypic similarity measures are used instead of genotypic measures. Phenotypic metrics embody domain specific knowledge that is most useful in multimodal optimization, as several different spatial positions can contain equally optimal solutions.
- It was shown that there exists a high probability that the most similar individuals to an offspring are its parents. Therefore, DC compares an offspring only to its parents and not to a random sample of the population.
- Random selection is used to select individuals for reproduction. Offspring replace parents only if the offspring perform better than the parents.

Probabilistic crowding, , introduced by Mengshoel *et al.* [578], is based on Mahfoud's deterministic crowding, but employs a probabilistic replacement strategy. Where the original crowding and DC techniques replaced an individual $\mathbf{x}_a$ with $\mathbf{x}_b$ if $\mathbf{x}_b$ was more fit than $\mathbf{x}_a$, probabilistic crowding uses the following rule: If individuals $\mathbf{x}_a$ and $\mathbf{x}_b$ are competing against each other, the probability of $\mathbf{x}_a$ winning is given by

$$P(\mathbf{x}_a(t) \text{ wins}) = \frac{f(\mathbf{x}_a(t))}{f(\mathbf{x}_a(t)) + f(\mathbf{x}_b(t))} \tag{9.34}$$

where $f(\mathbf{x}_a(t))$ is the fitness of individual $\mathbf{x}_a(t)$. The core of the algorithm is therefore to use a probabilistic tournament replacement strategy. Experimental results have shown it to be both fast and effective.

Coevolutionary Shared Niching

Goldberg and Wang [326] introduced coevolutionary shared niching (CSN). CSN locates niches by co-evolving two different populations of individuals in the same search space, in parallel. Let the two parallel populations be designated by $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Population $\mathcal{C}_1$ can be thought of as a normal population of candidate solutions, and it evolves as a normal population of individuals. Individuals in population $\mathcal{C}_2$ are scattered throughout the search space. Each individual in population $\mathcal{C}_1$ associates with itself a member of $\mathcal{C}_2$ that lies the closest to it using a genotypic metric. The fitness calculation of the i^{th} individual in population $\mathcal{C}_1$, $\mathcal{C}_1.\mathbf{x}_i$, is then adapted to $f'(\mathcal{C}_1.\mathbf{x}_i) = \frac{f(\mathcal{C}_1.\mathbf{x}_i)}{\mathcal{C}_2.n_2}$, where $f(\cdot)$ is the fitness function; $\mathcal{C}_2.n_2$ designates the cardinality of the set of individuals associated with individual $\mathcal{C}_2.\mathbf{x}_i$ and $\mathcal{C}_2.n_2$ is the index of the

closest individual in population $\mathcal{C}_2$ to individual $\mathcal{C}_1.\mathbf{x}_i$ in population $\mathcal{C}_1$. The fitness of individuals in population $\mathcal{C}_2$ is simply the average fitness of all the individuals associated to it in population $\mathcal{C}_1$, multiplied by $\mathcal{C}_2.\mathbf{x}_i$. Goldberg and Wang also developed the *imprint* CSN technique, that allows for the transfer of good performing individuals from the $\mathcal{C}_1$ to the $\mathcal{C}_2$ population.

CSN overcomes the limitation imposed by fixed inter-niche distances assumed in the original fitness sharing algorithm [325] and its derivate, dynamic fitness sharing [593]. The concept of a niche radius is replaced by the association made between individuals from the different populations.

Dynamic Niche Clustering

Dynamic niche clustering (DNC) is a fitness sharing based, cluster driven niching technique [305, 306]. It is distinguished from all other niching techniques by the fact that it supports 'fuzzy' clusters, i.e. clusters may overlap. This property allows the algorithm to distinguish between different peaks in a multi-modal function that may lie extremely close together. In most other niching techniques, a more general inter-niche radius (such as the σ_{share} parameter in fitness sharing) would prohibit this.

The algorithm constructs a *nicheset*, which is a list of niches in a population. The nicheset persists over multiple generations. Initially, each individual in a population is regarded to be in its own niche. Similar niches are identified using Euclidean distance and merged. The population of individuals is then evolved over a pre-determined number of generational steps. Before selection takes place, the following process occurs:

- The midpoint of each niche in the nicheset is updated, using the formula

$$\overline{\mathbf{x}}_u = \overline{\mathbf{x}}_u + \frac{\sum_{i=1}^{n_u}(\mathbf{x}_i - \overline{\mathbf{x}}_u) \cdot f(\mathbf{x}_i)}{\sum_{i=1}^{n_u} f(\mathbf{x}_i)} \tag{9.35}$$

  where $\overline{\mathbf{x}}_u$ is the midpoint of niche u, initially set to be equal to the position of the individual from which it was constructed, as described above. n_u is the niche count, or the number of individuals in the niche, $f(\mathbf{x}_i)$ is the fitness of individual $\mathbf{x}_i$ in niche u.
- A list of inter-niche distances is calculated and sorted. Niches are then merged.
- Similar niches are merged. Each niche is associated with a minimum and maximum niche radius. If the midpoints of two niches lie within the minimum radii of each other, they are merged.
- If any niche has a population size greater than 10% of the total population, random checks are done on the niche population to ensure that all individuals are focusing on the same optima. If this is not the case, such a niche may be split into sub-niches, which will be optimized individually in further generational steps.

Using the above technique, Gan and Warwick [307] also suggested a *niche linkage* extension to model niches of arbitrary shape.

9.6.2 Constraint Handling

Section A.6 summarizes different classes of methods to solve constrained optimization problems, as defined in Definition A.5. Standard genetic algorithms cannot be applied as is to solve constrained optimization problems. Most GA approaches to solve constrained problems require a change in the fitness function, or in the behavior of the algorithm itself.

Penalty methods are possibly one of the first approaches to address constraints [726]. As shown in Definition A.7, unfeasible solutions are penalized by adding a penalty function. A popular approach to implement penalties is given in equations (A.25) and (A.26) [584]. This approach basically converts the constrained problem to a penalized unconstrained problem.

Homaifar *et al.* [380] proposed a multi-level penalty function, where the magnitude of a penalty is proportional to the severity of the constraint violation. The multi-level function assumes that a set of intervals (or penalty levels) are defined for each constraint. An appropriate penalty value, λ_{mq}, is assigned to each level, $q = 1, \ldots, n_q$ for each constraint, m. The penalty function then changes to

$$p(\mathbf{x}_i, t) = \sum_{m=1}^{n_g+n_h} \lambda_{mq}(t) p_m(\mathbf{x}_i) \tag{9.36}$$

As an example, the following penalties can be used:

$$\lambda_{mq}(t) = \begin{cases} 10\sqrt{t} & \text{if } p_m(\mathbf{x}_i) < 0.001 \\ 20\sqrt{t} & \text{if } p_m(\mathbf{x}_i) \leq 0.1 \\ 100\sqrt{t} & \text{if } p_m(\mathbf{x}_i) \leq 1.0 \\ 300\sqrt{t} & \text{otherwise} \end{cases} \tag{9.37}$$

The multi-level function approach has the weakness that the number of parameters that has to be maintained increases significantly with increase in the number of levels, n_q, and the number of constraints, $n_g + n_h$.

Joines and Houck [425] proposed dynamic penalties, where

$$p(\mathbf{x}_i, t) = (\gamma \times t)^\alpha \sum_{m=1}^{n_g+n_h} p_m^\beta(\mathbf{x}_i) \tag{9.38}$$

where γ, α and β are constants. The longer the search continues, the higher the penalty for constraint violations. This allows for better exploration.

Other penalty methods can be found in [587, 588, 691].

Often referred to as the "death penalty" method, unfeasible solutions can be rejected. However, Michalewicz [585] shows that the method performs badly when the feasible region is small compared to the entire search space.

The interested reader is referred to [584] for a more complete survey of constraint handling methods.

9.6.3 Multi-Objective Optimization

Extensive research has been done to solve multi-objective optimization problems (MOP) as defined in Definition A.10 [149, 195]. This section summarizes only a few of these GA approaches to multi-objective optimization (MOO).

GA approaches for solving MOPs can be grouped into three main categories [421]:

- **Weighted aggregation** approaches where the objective is defined as a weighted sum of sub-objectives.
- **Population-based non-Pareto** approaches, which do not make use of the dominance relation as defined in Section A.8.
- **Pareto-based** approaches, which apply the dominance relation to find an approximation of the Pareto front.

Examples from the first and last classes are considered below.

Weighted Aggregation

One of the simplest approaches to deal with MOPs is to define an aggregate objective function as a weighted sum of sub-objectives:

$$f(\mathbf{x}) = \sum_{k=1}^{n_k} \omega_k f_k(\mathbf{x}) \tag{9.39}$$

where $n_k \geq 2$ is the total number of sub-objectives, and $\omega_k \in [0,1], k = 1, \ldots, n_k$ with $\sum_{k=1}^{n_k} \omega_k = 1$. While the aggregation approach above is very simple to implement and computationally efficient, it suffers from the following problems:

- It is difficult to get the best values for the weights, ω_k, since these are problem dependent.
- These methods have to be re-applied to find more than one solution, since only one solution can be obtained with a single run of an aggregation algorithm. However, even for repeated applications, there is no guarantee that different solutions will be found.
- The conventional weighted aggregation as given above cannot solve MOPs with a concave Pareto front [174].

To address these problems, Jin *et al.* [421, 422], proposed aggregation methods with dynamically changing weights (for $n_k = 2$) and an approach to maintain an archive of nondominated solutions. The following approaches have been used to dynamically adapt weights:

- **Random distribution of weights**, where for each individual,

$$\omega_{1,i}(t) = U(0, n_s)/n_s \tag{9.40}$$
$$\omega_{2,i}(t) = 1 - \omega_{1,i}(t) \tag{9.41}$$

- **Bang-bang weighted aggregation**, where

$$\omega_1(t) = \text{sign}(\sin(2\pi t/\tau)) \quad (9.42)$$
$$\omega_2(t) = 1 - \omega_1(t) \quad (9.43)$$

  where τ is the weights' change frequency. Weights change abruptly from 0 to 1 each τ generation.

- **Dynamic weighted aggregation**, where

$$\omega_1(t) = |\sin(2\pi t/\tau)| \quad (9.44)$$
$$\omega_2(t) = 1 - \omega_1(t) \quad (9.45)$$

  With this approach, weights change more gradually.

Jin *et al.* [421, 422] used Algorithm 9.10 to produce an archive of nondominated solutions. This algorithm is called after the reproduction (crossover and mutation) step.

Algorithm 9.10 Algorithm to Maintain an Archive of Nondominated Solutions

for *each offspring,* $\mathbf{x}_i'(t)$ **do**
  if $\mathbf{x}_i'(t)$ *dominates an individual in the current population,* $\mathcal{C}(t)$*, and* $\mathbf{x}_i'(t)$ *is not dominated by any solutions in the archive and* $\mathbf{x}_i'(t)$ *is not similar to any solutions in the archive* **then**
    if *archive is not full* **then**
      Add $\mathbf{x}_i'(t)$ to the archive;
    else if $\mathbf{x}_i'(t)$ *dominates any solution* $\mathbf{x}_a$ *in the archive* **then**
      Replace $\mathbf{x}_a$ with $\mathbf{x}_i'(t)$;
    else if *any* $\mathbf{x}_{a_1}$ *in the archive dominates another* $\mathbf{x}_{a_2}$ *in the archive* **then**
      Replace $\mathbf{x}_{a_2}$ with $\mathbf{x}_i'(t)$;
    else
      Discard $\mathbf{x}_i'(t)$;
    end
  end
  else
    Discard $\mathbf{x}_i'(t)$;
  end
  for *each solution* $\mathbf{x}_{a_1}$ *in the archive* **do**
    if $\mathbf{x}_{a_1}$ *dominates* $\mathbf{x}_{a_2}$ *in the archive* **then**
      Remove $\mathbf{x}_{a_2}$ from the archive;
    end
  end
end

Vector Evaluated Genetic Algorithm

The vector evaluated GA (VEGA) [760, 761] is one of the first algorithms to solve MOPs using multiple populations. One subpopulation is associated with each objective. Selection is applied to each subpopulation to construct a mating pool. The result of this selection process is that the best individuals with respect to each objective are included in the mating pool. Crossover then continues by selecting parents from the mating pool.

Niched Pareto Genetic Algorithm

Horn *et al.* [382] developed the niched Pareto GA (NPGA), where an adapted tournament selection operator is used to find nondominated solutions. The Pareto domination tournament selection operator randomly selects two candidate individuals, and a comparison set of randomly selected individuals. Each candidate is compared against each individual in the comparison set. If one candidate is dominated by an individual in the comparison set, and the other candidate is not dominated, then the latter is selected. If neither or both are dominated equivalence class sharing is used to select one individual: The individual with the lowest niche count is selected, where the niche count is the number of individuals within a niche radius, σ_{share}, from the candidate. This strategy will prefer a solution on a less populated part of the Pareto front.

Nondominated Sorting Genetic Algorithm

Srinivas and Deb [807] developed the nondominated sorting GA (NSGA), where only the selection operator is changed. Individuals are Pareto-ranked into different Pareto fronts as described in Section 12.6.2. Fitness proportionate selection is used based on the shared fitness assigned to each solution. The NSGA is summarized in Algorithm 9.11.

Deb *et al.* [197] pointed out that the NSGA has a very high computational complexity of $O(n_k n_s^3)$. Another issue with the NSGA is the reliance on a sharing parameter, σ_{share}. To address these problems, a fast nondominated sorting strategy was proposed and a crowding comparison operator defined. The fast nondominated sorting algorithm calculates for each solution, $\mathbf{x}_a$, the number of solutions, n_a, which dominates $\mathbf{x}_a$, and the set, $\mathcal{X}_a$, of solutions dominated by $\mathbf{x}_a$. All those solutions with $n_a = 0$ are added to a list, referred to as the current front. For each solution in the current front, each element, $\mathbf{x}_b$, of the set $\mathcal{X}_b$ has its counter, n_b, decremented. When $n_b = 0$, the corresponding solution is added to a temporary list. When all the elements of the current front have been processed, its elements form the first front, and the temporary list becomes the new current list. The process is repeated to form the other fronts.

To eliminate the need for a sharing parameter, solutions are sorted for each subobjective. For each subobjective, the average distance of the two points on either side of $\mathbf{x}_a$ is calculated. The sum of the average distances over all subobjectives gives the

Algorithm 9.11 Nondominated Sorting Genetic Algorithm

```
Set the generation counter, t = 0;
Initialize all control parameters;
Create and initialize the population, C(0), of n_s individuals;
while stopping condition(s) not true do
    Set the front counter, p = 1;
    while there are individuals not assigned to a front do
        Identify nondominated individuals;
        Assign fitness to each of these individuals;
        Apply fitness sharing to individuals in the front;
        Remove individuals;
        p = p + 1;
    end
    Apply reproduction using rank-based, shared fitness values;
    Select new population;
    t = t + 1;
end
```

crowding distance, d_a. If R_a indicates the nondomination rank of $\mathbf{x}_a$, then a crowding comparison operator is defined as: $a \leq_* b$ if $(R_a < R_b)$ or $((R_a = R_b)$ and $(d_a > d_b))$. For two solutions with differing nondomination ranks, the one with the lower rank is preferred. If both solutions have the same rank, the one located in the less populated region of the Pareto front is preferred.

9.6.4 Dynamic Environments

A very simple approach to track solutions in a dynamic environment as defined in Definition A.16 is to restart the GA when a change is detected. A restart approach can be quite inefficient if changes in the landscape are small. For small changes, the question arises if a changing optimum can be tracked by simply continuing with the search. This will be possible only if the population has some degree of diversity to enable further exploration. An ability to maintain diversity is therefore an important ingredient in tracking changing optima. This is even more so for large changes in the landscape, which may cause new optima to appear and existing ones to disappear.

A number of approaches have been developed to maintain population diversity. The hyper-mutation strategy of Cobb [141] drastically increases the rate of mutation for a number of generations when a change is detected. An increased rate of mutation as well as an increased mutational step size allow for further exploration. The variable local search strategy [873] gradually increases the mutational step sizes and rate of mutation after a change is detected. Mutational step sizes are increased if no improvement is obtained over a number of generations for the smaller step sizes.

Grefenstette [336] proposed the random immigrants strategy where, for each generation, part of the population is replaced by randomly generated individuals.

Mori *et al.* [607] proposed that a memory of every generation's best individual be stored (in a kind of hall-of-fame). Individuals stored in the memory serve as candidate parents to the reproduction operators. The memory has a fixed size, which requires some kind of replacement strategy. Branke [82] proposed that the best individual of the current generation replaces the individual that is most similar.

For a more detailed treatment of dynamic environments, the reader is referred to [83].

9.7 Applications

Genetic algorithms have been applied to solve many real-world optimization problems. The reader is referred to http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/tcw2/report.html for a good summary of and references to applications of GAs. The rest of this section describes how a GA can be applied to routing optimization in telecommunications networks [778]. Given a network of n_x switches, an origin switch and a destination switch, the objective is to find the best route to connect a call between the origin and destination switches. The design of the GA is done in the following steps:

1. **Chromosome representation**: A chromosome consists of a maximum of n_x switches. Chromosomes can be of variable length, since telecommunication routes can differ in length. Each gene represents one switch. Integer values representing switch numbers are used as gene values - no binary encoding is used. The first gene represents the origin switch and the last gene represents the destination switch. Example chromosomes are

$$(1 \quad 3 \quad 6 \quad 10)$$

$$(1 \quad 5 \quad 2 \quad 5 \quad 10) = (1 \quad 5 \quad 2 \quad 10)$$

   Duplicate switches are ignored. The first chromosome represents a route from switch 1 to switch 3 to switch 6 to switch 10.

2. **Initialization of population**: Individuals are generated randomly, with the restriction that the first gene represents the origin switch and the last gene represents the destination switch. For each gene, the value of that gene is selected as a uniform random value in the range $[1, n_x]$.

3. **Fitness function:** The multi-criteria objective function

$$f(\mathbf{x}_i) = \omega_1 f_{Switch}(\mathbf{x}_i) + \omega_2 f_{Block}(\mathbf{x}_i) + \omega_3 f_{Util}(\mathbf{x}_i) + \omega_4 f_{Cost}(\mathbf{x}_i) \tag{9.46}$$

   is used where

$$f_{Switch}(\mathbf{x}_i) = \frac{|\mathbf{x}_i|}{n_x} \tag{9.47}$$

   represents the minimization of route length, where $\mathbf{x}_i$ denotes the route and $|\mathbf{x}_i|$ is the total number of switches in the route,

$$f_{Block}(\mathbf{x}_i) = 1 - \prod_{ab \in \mathbf{x}_i}^{|\mathbf{x}_i|} (1 - B_{ab} + \alpha_{ab}) \tag{9.48}$$

with

$$\alpha_{ab} = \begin{cases} 1 & \text{if } ab \text{ does not exist} \\ 0 & \text{if } ab \text{ does exist} \end{cases} \tag{9.49}$$

represent the objective to select routes with minimum congestion, where B_{ab} denotes the blocking probability on the link between switches a and b,

$$f_{Util}(\mathbf{x}_i) = \min_{ab \in \mathbf{x}_i} \{1 - U_{ab}\} + \alpha_{ab} \tag{9.50}$$

maximizes utilization, where U_{ab} quantifies the level of utilization of the link between a and b, and

$$f_{Cost}(\mathbf{x}_i) = \sum_{ab \in \mathbf{x}_i}^{|\mathbf{x}_i|} C_{ab} + \alpha_{ab} \tag{9.51}$$

ensures that minimum cost routes are selected, where C_{ab} represents the financial cost of carrying a call on the link between a and b. The constants ω_1 to ω_4 control the influence of each criterion.

4. Use any **selection** operator.
5. Use any **crossover** operator.
6. **Mutation**: Mutation consists of replacing selected genes with a uniformly random selected switch in the range $[1, n_x]$.

This example is an illustration of a GA that uses a numeric representation, and variable length chromosomes with constraints placed on the structure of the initial individuals.

9.8 Assignments

1. Discuss the importance of the crossover rate, by considering the effect of different values in the range [0,1].
2. Compare the following replacement strategies for crossover operators that produce only one offspring:
   (a) The offspring always replaces the worst parent.
   (b) The offspring replaces the worst parent only when its fitness is better than the worst parent.
   (c) The offspring always replaces the worst individual in the population.
   (d) Boltzmann selection is used to decide if the offspring should replace the worst parent.
3. Show how the heuristic crossover operator incorporates search direction.
4. Propose a multiparent version of the geometrical crossover operator.
5. Propose a marker initialization and update strategy for gene scanning applied to order-based representations
6. Propose a random mutation operator for discrete-valued decision variables.
7. Show how a GA can be used to train a FFNN.

8. In the context of GAs, when is a high mutation rate an advantage?
9. Is the following strategy sensible? Explain your answer. "*Start evolution with a large mutation rate, and decrease the mutation rate with an increase in generation number.*"
10. Discuss how a GA can be used to cluster data.
11. For floating-point representations, devise a deterministic schedule to dynamically adjust mutational step sizes. Discuss the merits of your proposal.
12. Suggest ways in which the competitive template can be initialized for messy GAs.
13. Discuss the consequences of migrating the best individuals before islands have converged.
14. Discuss the influence that the size of the comparison set has on the performance of the niched Pareto GA.