7 Genetic algorithms

This chapter covers

- Introducing population-based optimization algorithms
- Understanding evolutionary computation
- Understanding the different components of genetic algorithms
- Implementing genetic algorithms in Python

Suppose you're on a treasure-hunting mission and you don't want the risk of searching alone and returning empty-handed. You might decide to collaborate with a group of friends and share information. This approach follows a population-based search strategy, where multiple agents are involved in the search process.

During this collaborative effort, you may notice that some hunters perform better than others. In this case, you may choose to retain only the best-performing hunters and replace the less competent ones with new recruits. This process resembles the workings of evolutionary algorithms such as genetic algorithms, where the fittest individuals survive and pass on their traits to the next generation.

In this chapter, the binary-coded genetic algorithm is presented and discussed as an evolutionary computing algorithm. We'll look at different elements of this algorithm and at the implementation details. Other variants of genetic algorithms, such as the gray-coded genetic algorithm, real-valued genetic algorithm, and permutation-based genetic algorithm will be discussed in the next chapter.

7.1 Population-based metaheuristic algorithms

Population-based metaheuristic algorithms (P-metaheuristics), such as genetic algorithms, particle swarm optimization, and ant colony optimization, utilize multiple agents to search for an optimal or near-optimal global solution. As these algorithms begin with a diverse set of initial populations, they are naturally more exploration-based, allowing for the possibility of finding better solutions that might be missed by trajectory-based (S-metaheuristic) algorithms, which are more exploitation-based.

Population-based metaheuristic algorithms can be classified into two main categories based on their source of inspiration: *evolutionary computation algorithms* and *swarm intelligence algorithms*, as shown in figure 7.1.

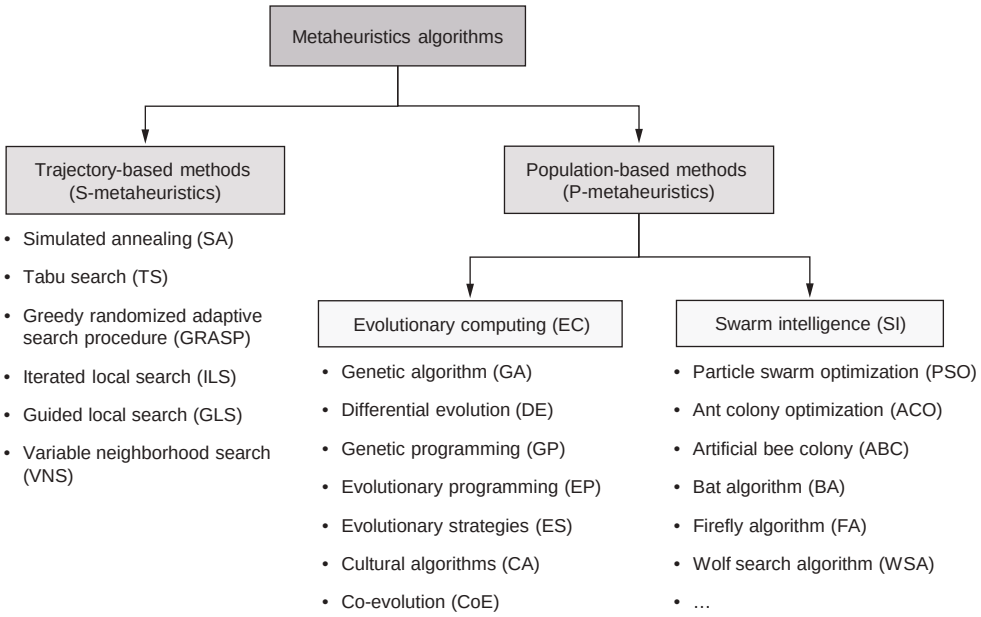


Figure 7.1 Metaheuristic algorithms

Evolutionary computation (EC) algorithms, as the name suggests, are inspired by the process of biological evolution. These algorithms use a population of potential solutions, which undergo genetic operations, such as mutation and crossover, to create new offspring that may have better fitness values. The process of selection determines which individuals in the population are selected to reproduce and create the next generation. Genetic algorithm (GA), differential evolution (DE), genetic programming (GP), evolutionary programming (EP), evolutionary strategies (ES), cultural algorithms (CA), and co-evolution (CoE) are examples of evolutionary computation algorithms.

Swarm intelligence (SI) algorithms, on the other hand, are inspired by the collective behavior of social organisms such as ants, bees, and birds, and they'll be discussed in part 4 of this book. These algorithms use a population of agents that interact with each

other to find a solution. They use a variety of mechanisms, such as communication, cooperation, and self-organization, to optimize the search process. Examples of swarm intelligence algorithms include particle swarm optimization (PSO), ant colony optimization (ACO), artificial bee colony (ABC), the firefly algorithm (FA), the bat algorithm (BA), and the wolf search algorithm (WSA).

Both evolutionary computation and swarm intelligence algorithms are population-based algorithms that begin their search for the optimal or near-optimal solution from an initial population of candidate solutions. The quality and diversity of the initial population significantly influences the performance and efficiency of the algorithm. A well-constructed initial population provides a good starting point for the search process and can help the algorithm quickly converge toward a promising region of the search space. In contrast, a poorly constructed initial population may result in a premature convergence to a suboptimal solution, may get the algorithm stuck in a suboptimal region, or may take longer to converge toward a solution. To ensure a good balance between exploration and exploitation, the initial population should be diverse and cover a wide range of potential solutions.

A comparison between different initialization strategies for population-based metaheuristics is provided in El-Ghazali Talbi's *Metaheuristics: From Design to Implementation* [1], based on three key aspects: diversity, computational cost, and the quality of initial solutions. Initial solutions can be generated using a pseudo-random process or a quasi-random search. Initial solutions can also be generated sequentially (sequential diversification) or concurrently (parallel diversification) to achieve very high diversity. Heuristics involve using local search or greedy methods to generate initial solutions.

As shown in table 7.1, a pseudo-random strategy provides moderate diversity, low computational cost, and low-quality initial solutions. A quasi-random strategy exhibits higher diversity with comparable computational cost and low-quality initial solutions. Sequential diversification and parallel diversification both stand out with very high diversity, but the former incurs moderate computational cost, while the latter has low computational cost; both methods result in low-quality initial solutions. In contrast, the use of heuristics, such as local search or a greedy heuristic, yields high-quality initial solutions but with low diversity and high computational cost.

Table 7.1 Initialization strategies for population-based metaheuristics

Initialization strategy	Diversity	Computational cost	Quality of initial solution
Pseudo-random	Moderate	Low	Low
Quasi-random	High	Low	Low
Sequential diversification	Very high	Moderate	Low
Parallel diversification	Very high	Low	Low
Heuristics (e.g., local search or greedy heuristic)	Low	High	High

It is often beneficial to use a randomized approach to generate the initial population, where the candidates are samples from different regions of the search space to maximize the chances of finding the optimal solution. The next listing shows how we can sample initial solutions using Python. Let's start by generating 200 pseudo-random numbers.

Listing 7.1 Generating initial populations in Python

```
import math
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(6345245)

N=200
P_random_pseudo=np.random.rand(N,N)
```

Set a seed for the random number generator.

Number of samples

Pseudo-random sampling

NOTE Random numbers are inherently unpredictable, pseudo-random numbers are deterministic but appear random, and quasi-random numbers are deterministic with evenly distributed patterns.

The generalized Halton number generator in the ghalton library can be used to generate quasi-random numbers. This method is based on the Halton sequence, which uses coprime numbers as its bases. You can use the generalized Halton number generator as follows:

```
!pip install ghalton
import ghalton

sequencer = ghalton.GeneralizedHalton(7,23)
P_random_quasi = np.array(sequencer.get(N))
```

The Box-Muller transform is used to generate pairs of independent, standard, normally distributed random numbers from pairs of uniformly distributed random numbers. Box-Muller is a 2D Gaussian sampling method that can be used as follows:

```
u1 = np.random.uniform(size=(N))
u2 = np.random.uniform(size=(N))

P_BM_x = np.sqrt(-2*np.log(u1))*np.cos(2*math.pi*u2)
P_BM_y = np.sqrt(-2*np.log(u1))*np.sin(2*math.pi*u2)
```

Generate uniformly distributed values between 0 and 1.

Calculate x and y values using Box-Muller.

One of the drawbacks of the Box-Muller transform is its tendency to cluster values around the mean due to its dependency on uniform distribution. Additionally, calculating the square root can be costly.

Central limit theorem (CLT) sampling is another sampling method where the distribution of the sample means approximates a normal distribution as the sample size gets larger, regardless of the population's distribution. The following code snippet shows how to implement this method:

```
import random

P_CLT_x=[2.0 * math.sqrt(N) * (sum(random.randint(0,1) for x in range(N)) /
N - 0.5)
➥ for x in range(N)]
P_CLT_y=[2.0 * math.sqrt(N) * (sum(random.randint(0,1) for x in range(N)) /
N - 0.5)
➥ for x in range(N)]
```

The Sobol low-discrepancy sequence (LDS) is a quasi-random sampling method available in the sobol_seq package. This method generates a sequence of points that are evenly spaced and distributed throughout the sample space, such that the gaps between adjacent points are as small as possible. It can be used as follows:

```
!pip install sobol_seq
import sobol_seq
P_sobel=sobol_seq.i4_sobol_generate(2,N)
```

Latin hypercube sampling is a parallel diversification method where the search space is decomposed into 25 blocks, and a solution is generated pseudo-randomly in each block. An example of using the Latin hybercube sampling method in the pyDOE (Design of Experiments) Python package is shown here:

```
!pip install pyDOE
from pyDOE import *
P_LHS=lhs(2, samples=N, criterion='center')
```

Let's visualize all of these sampling methods so we can get a good sense of the differences between them:

```
f, (ax1, ax2) = plt.subplots(ncols=2, figsize=(18,8))
f, (ax3,ax4) = plt.subplots(ncols=2, figsize=(18,8))
f, (ax5, ax6) = plt.subplots(ncols=2, figsize=(18,8))
ax1.scatter(P_random_pseudo[:,0], P_random_pseudo[:,1], color="gray")
ax2.scatter(P_random_quasi[:100], P_random_quasi[100:], color="red")
ax3.scatter(P_BM_x, P_BM_y, color="green")
ax4.scatter(P_CLT_x, P_CLT_y, color="cyan")
ax5.scatter(P_sobel[:,0], P_sobel[:,1], color="magenta")
ax6.plot(P_LHS[:,0], P_LHS[:,1], "o")

ax1.set_title("Pseudo-random")
ax2.set_title("Quasi-random")
ax3.set_title("Box-Muller")
ax4.set_title("Central Limit Theorem")
ax5.set_title("Sobol")
ax6.set_title("Latin Hypercube")
plt.show()
```

Running this code generates the plots shown in figure 7.2. In this figure, candidate solutions have been sampled from a feasible search space using various sampling methods, with each point representing a different solution. The level of diversity achieved

by each sampling method can be evaluated by observing the gaps between the points and their dispersion.

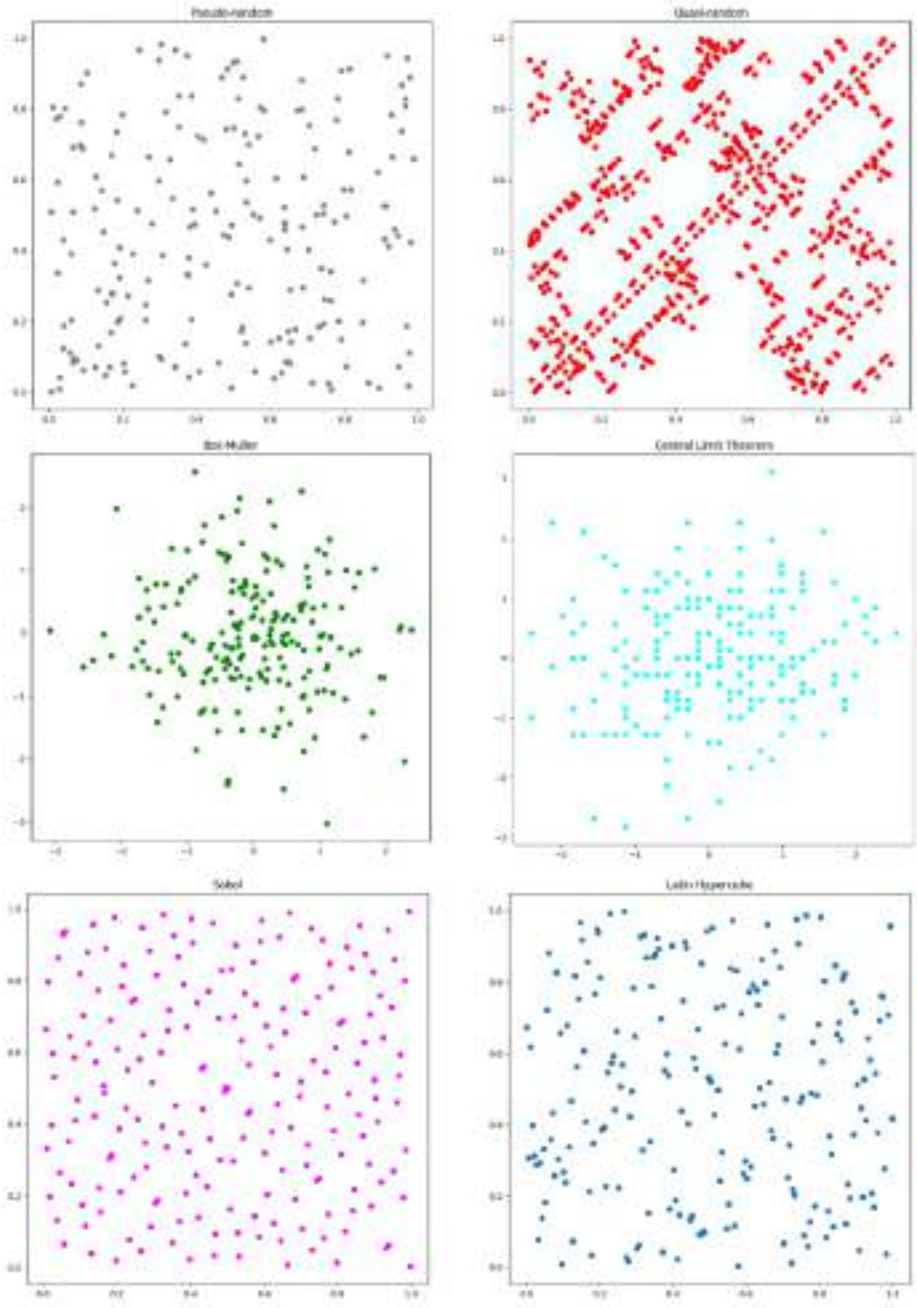


Figure 7.2 Sampling methods for generating an initial population

As mentioned in appendix A (see liveBook), there are several Python packages for evolutionary computation. In this chapter, we will focus on using pymoo: multi-objective optimization in Python. Pymoo provides different sampling methods for creating an initial population or an initial search point. Examples include random sampling and Latin hypercube sampling. As a continuation of listing 7.1, the following code snippet shows random sampling in pymoo:

```
!pip install -U pymoo
from pymoo.core.problem import Problem
from pymoo.operators.sampling.rnd import FloatRandomSampling
from pymoo.util import plotting

problem = Problem(n_var=2, xl=0, xu=1)

sampling = FloatRandomSampling()

X = sampling(problem, 200).get("X")
plotting.plot(X, no_fill=True)
```

Import an instance of the problem class.

Import the random sampling method.

Import the visualization method.

Create a problem with two variables, and specify the lower and upper bounds.

Create an instance of the random sampler.

Generate 200 random solutions/individuals.

Visualize the generated individuals.

The following code generates and visualizes 200 initial solutions using Latin hypercube sampling:

```
from pymoo.operators.sampling.lhs import LHS

sampling = LHS()

X = sampling(problem, 200).get("X")
plotting.plot(X, no_fill=True)
```

Import the Latin hypercube sampling module.

If the solutions take the form of permutations, random permutations can be generated as follows:

```
per1=np.random.permutation(10)
print(per1)

per2 = np.array([5, 4, 9, 0, 1, 2, 6, 8, 7, 3])
np.random.shuffle(per2)
print(per2)

pop_init = np.arange(50).reshape((10,5))
np.random.permutation(pop_init)

from itertools import combinations
size=5
ones=2

for pos in map(set, combinations(range(size), ones)):
     print([int(i in pos) for i in range(size)], sep='\n')
```

Randomly permute a sequence, or return a permuted range.

Randomly shuffle a sequence.

Population of the initial solution as real-value permutations

Population of the initial solution as binary permutations with the number of bits in the binary string and the number of ones in each binary string

You can also generate a random route between two points using the following code:

```
import osmnx as ox
import random
from collections import deque
from optalgotools.structures import Node
```

```
G = ox.graph_from_place("University of Toronto")
fig, ax = ox.plot_graph(G)

def randomized_search(G, source, destination):
    origin = Node(graph = G, osmid = source)
    destination = Node(graph = G, osmid = destination)

    route = []
    frontier = deque([origin])
    explored = set()
    while frontier:
        node = random.choice(frontier)
        frontier.remove(node)
        explored.add(node.osmid)

        for child in node.expand():
            if child not in explored and child not in frontier:
                if child == destination:
                    route = child.path()
                    return route
                frontier.append(child)

    raise Exception("destination and source are not on same component")

random_route = randomized_search(G, 24959528, 1480794706)

fig, ax = ox.plot_graph_route(G, random_route)
```

This is a typical graph search with a shuffled frontier.

This is the randomization part.

Generate random routes between two nodes.

Visualize the random routes.

The preceding code modifies a typical graph search algorithm by scrambling the frontier nodes. This means that candidates for expansion are "random," which means different routes are yielded when it's called repeatedly. Some generated random routes are shown in figure 7.3.

Figure 7.3 Generating random initial routes

In the next section, I'll introduce evolutionary computation as population-based metaheuristics.

7.2 Introducing evolutionary computation

Evolution can be considered an optimization process in the sense that it involves the gradual improvement of the characteristics of living organisms over time, resulting in adaptation to dynamically changing and competitive environments and an enhanced ability to survive in these environments. In this section, I'll provide an overview of the fundamental concepts of biological evolution. Understanding these principles is important for gaining insight into evolutionary computation.

7.2.1 A brief recap of biology fundamentals

The *nucleus* is the central part of any living cell that contains the genetic information. This genetic information is stored in the *chromosomes*, each of which is built of deoxyribonucleic acid (DNA), which carries the genetic information used in the growth, development, functioning, and reproduction of all living organisms. Humans have a total of 23 pairs of chromosomes, or 46 chromosomes in total, in each of their cells. Each chromosome is made up of many different sections called *genes*, which are responsible for coding specific properties of an individual. The variant form of a gene that determines these properties, found at a specific location on a chromosome, is called an *allele*. Every gene has a unique position on the chromosome called a *locus*. The entire combination of genes is called a *genotype*, and it's the genotype that provides the genetic blueprint for an organism, determining the potential for an individual's traits and characteristics. The term *phenotype* refers to the observable physical, behavioral, and physiological characteristics of an organism, which result from the interaction between its genotype and the environment.

To illustrate the concept of genes and their role in determining the characteristics of a living organism, let's consider an example where a DNA molecule consists of four genes that are responsible for different traits: appetite, movement, feet, and skin type. The appetite gene may have different values that reflect the diet of the organism, such as herbivore (H), carnivore (C), or insectivore (I). The movement gene may determine the organism's mode of movement, such as climbing (CL), flying (FL), running (R), or swimming (SW). The feet gene may determine the type of feet or limbs that the organism has, such as claws (CLW), flippers (FLP), hooves (HV), or wings (WNG). Finally, the skin gene may determine the skin covering of the organism, such as fur (F), scales (S), or feathers (FTH), as illustrated in figure 7.4.

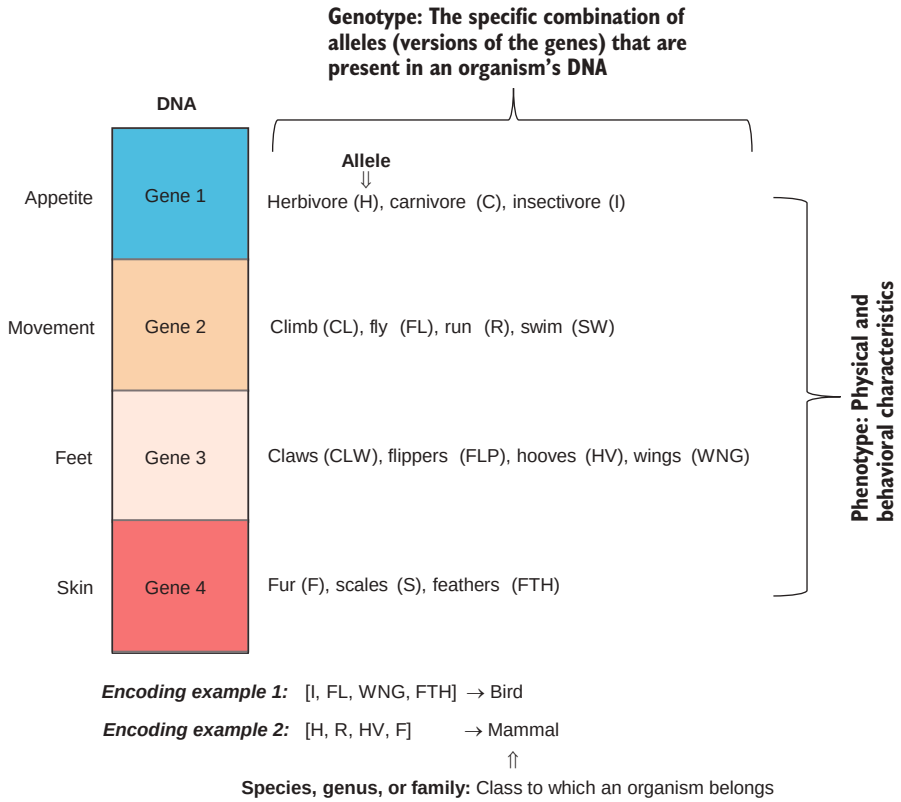


Figure 7.4 Genotype, phenotype, and taxonomic classification

In this example, the *genotype* refers to the specific genetic makeup of an organism, which is determined by the specific combination of alleles that an individual inherits from its parents. The specific values of these genes will determine the *phenotype*, or observable characteristics, of the organism. For example, an organism with an insectivore appetite gene, a flying movement gene, a wings feet gene, and a feather skin gene would likely be a bird. On the other hand, an organism with an herbivorous appetite gene, a running movement gene, a hooves feet gene, and a fur skin gene would likely be a mammal, such as a white-tailed deer.

The class to which an organism belongs, such as the species, genus, or family, is determined by its taxonomic classification based on shared characteristics with other organisms.

7.2.2 The theory of evolution

The theory of evolution explains how species of living organisms have changed over time and diversified into the forms we see today. This theory, developed by Charles Darwin, offers an explanation of biological diversity and its underlying mechanisms.

According to the theory, *natural selection* is a major mechanism that drives evolution. Over the course of numerous generations, adaptations arise from the cumulative effects of successive, minor, stochastic alterations in traits, and natural selection favors those variants that are best suited to their environment. This phenomenon is known as survival of the fittest: selected individuals reproduce, passing their properties to their offspring. Other individuals die without mating, and their properties are thus discarded. Over time, natural selection plays a significant role in shaping the characteristics and adaptations of populations, promoting the transmission of advantageous traits and eliminating less beneficial ones.

The theory of evolution

The theory of evolution by natural selection can be summarized as follows:

- 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 subsequent generations, and (over time) become dominant among the population.
- During production of a child organism, random events cause random changes to the child organism's characteristics.
- If these new characteristics are a benefit to the organism, the chances of survival for that organism are increased.

Evolutionary computation techniques mimic biological evolution and process a sequence of operations, such as creating an initial population (a collection of chromosomes), evaluating the population, and then evolving the population through multiple generations.

7.2.3 Evolutionary computation

Computational intelligence (CI) is a subfield of artificial intelligence (AI) that emphasizes the design, application, and development of algorithms that can learn and adapt to solve complex problems. It focuses on soft computing methods such as fuzzy logic, neural networks, evolutionary computation, and swarm intelligence. *Evolutionary computation* (EC), as a branch of CI, employs various computational methods inspired by biological evolution. These methods have computational mechanisms of natural selection, survival of the fittest, and reproduction as the core elements of their computational systems.

Generally speaking, EC algorithms consist of the following main components:

- *Population of individuals*—This is a set of candidate solutions that are initially generated randomly or by some heuristic methods and are then improved over time. The population size is usually large in order to explore a wide range of possible

solutions to the problem. However, the optimal population size depends on various factors, such as the complexity of the problem, the number of variables in the problem, the required accuracy of the solution, and the computational resources available. In practice, the optimal population size is often determined through experimentation, with the performance of the algorithm being evaluated for different population sizes and the best performing size being selected.

- *Fitness function*—This function evaluates the quality of candidate solutions. It determines how well each solution solves the given problem by assigning a fitness value to each individual in the population. The higher the fitness value, the better the solution.
- *Parent selection method*—This method is used to select the most promising individuals from the population in order to create new offspring for the next generation.
- *Genetic operators*—These operators include *crossover* and *mutation*, and they are used to create new offspring from selected parents. The crossover operator exchanges genetic material between two selected individuals to create new offspring with a combination of traits from both parents. The mutation operator introduces random changes to the offspring's genetic makeup to add diversity to the population and prevent stagnation.
- *Survival methods*—These methods determine which individuals in a population will survive to the next generation.

Together, these five components form the basis of EC algorithms, which can effectively solve various optimization problems. As illustrated in figure 7.5, there are several EC paradigms.

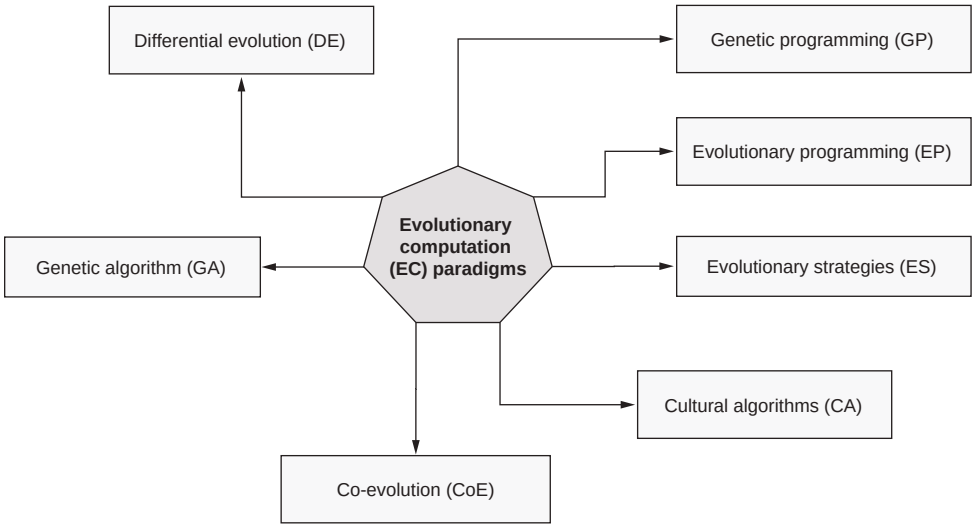


Figure 7.5 EC paradigms

These paradigms mainly vary in their approaches to representing individuals, parents, survival selection methods, and genetic operators:

- *Genetic algorithm (GA)*—This search algorithm mimics natural evolution, where each individual is a candidate solution encoded as a binary, real-valued, or permutation vector. We will discuss genetic algorithms in detail in this part of the book.
- *Differential evolution (DE)*—This algorithm uses real-valued vectors as individuals and generates new solutions by adding weighted differences between pairs of existing solutions. It is similar to GA, differing in the reproduction mechanism used.
- *Genetic programming (GP)*—This is a special case of GA, where each individual is a computer program encoded as a variable-length tree. This tree structure is used to represent functions and operators, such as `if-else` statements and mathematical operations.
- *Evolutionary programming (EP)*—This is similar to GP, but it focuses on evolving behavioral traits rather than program structure. It is an open framework where any representation and mutation operation can be applied, but there is no recombination.
- *Evolutionary strategies (ES)*—This algorithm uses real-valued vectors as individuals and adapts mutation and recombination parameters during evolution. Plus-selection, comma-selection, greedy selection, and distance-based selection are used as selection methods.
- *Cultural algorithm (CA)*—This approach incorporates social learning from a shared belief space into the traditional population-based evolution process. CA models the evolution of a population's culture and how it influences the genetic and phenotypic evolution of individuals.
- *Co-evolution (CoE)*—This is based on the reciprocal evolutionary change that occurs between interacting populations, where each represents a given species, together optimizing coupled objectives.

EC is a powerful approach to optimization that has several advantages, as well as a few drawbacks. The advantages include the following:

- EC algorithms do not make any presumptions about the problem space, making them applicable to a wide range of problems.
- They are widely applicable across different domains and can be used to solve continuous and discrete problems in various fields.
- The solutions produced by EC algorithms are more interpretable than those of neural networks or other black-box optimization techniques. This is mainly because EC algorithms use a more transparent process of selection, mutation, and recombination that can be tracked and understood step by step, whereas neural networks are often considered "black boxes" due to their complex, layered structures and nonlinear operations.

- EC algorithms provide multiple alternative solutions, which can be useful in cases where there is no single best solution.
- EC algorithms exhibit inherent parallelism, making them well-suited for simple parallel implementations on modern hardware.

The disadvantages of EC include the following:

- EC algorithms can be computationally expensive, meaning that they may be slow to converge or require a significant amount of computational resources to run.
- While EC algorithms, like many metaheuristic algorithms, cannot guarantee finding an optimal solution, they often converge to a near-optimal solution within a finite time frame.
- EC algorithms often require parameter tuning to achieve good performance, which can be time-consuming and challenging.

This chapter primarily focuses on genetic algorithms. The following section will look at the various components of genetic algorithms.

7.3 Genetic algorithm building blocks

Genetic algorithms are the most widely used form of EC. They are adaptive heuristic search algorithms that are designed to simulate processes in natural systems necessary for evolution, as proposed by Charles Darwin in his theory of evolution. These algorithms represent an intelligent exploitation of a random search within a defined search space.

The first genetic algorithm, named simple genetic algorithm (SGA) and also known as the classical or canonical GA, was developed by John Holland in 1975. Through his research, Holland provided insights into the design of artificial systems that are robust, adaptive, and capable of evolving to meet new challenges. By studying the processes of natural systems, he sought to create algorithms and computational models that could solve complex problems much like natural systems can. Holland defined GA as a computer program that evolves in ways that resemble natural selection and that can solve complex problems that even their creators do not fully understand. GA is based on the principles of evolution via natural selection, employing a population of individuals that undergo selection in the presence of variation-inducing operators such as mutation and crossover (recombination). A fitness function is used to evaluate individuals, and their reproductive success varies with their fitness. Figure 7.6 shows an analogy between GA and natural evolution.

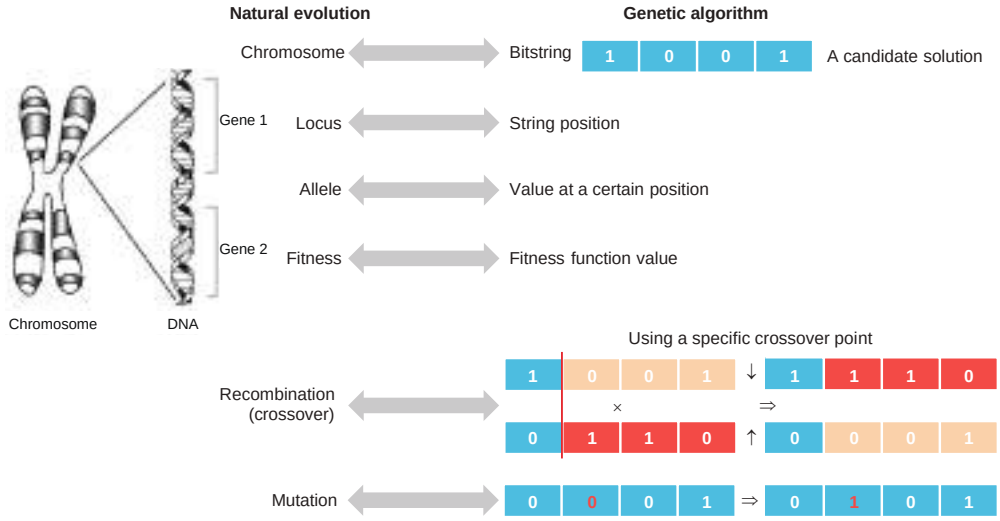


Figure 7.6 GA versus natural evolution

GA starts by initializing a population of individuals or candidate solutions. The fitness of all the individuals in the population is evaluated based on a defined fitness function, and then a new population is created by performing crossover and mutation, which generate children or new solutions. In constrained optimization problems, a feasibility check and repair should be applied after the offspring are produced.

The population keeps evolving until certain stopping criteria are met, as illustrated in figure 7.7. These termination criteria could be

- A specified number of generations or fitness evaluations (100 or 150 generations)
- An adequate solution that reaches a minimum threshold
- When there is no improvement in the best individual for a specified number of generations
- When memory or time constraints are reached
- Any combination of the preceding points

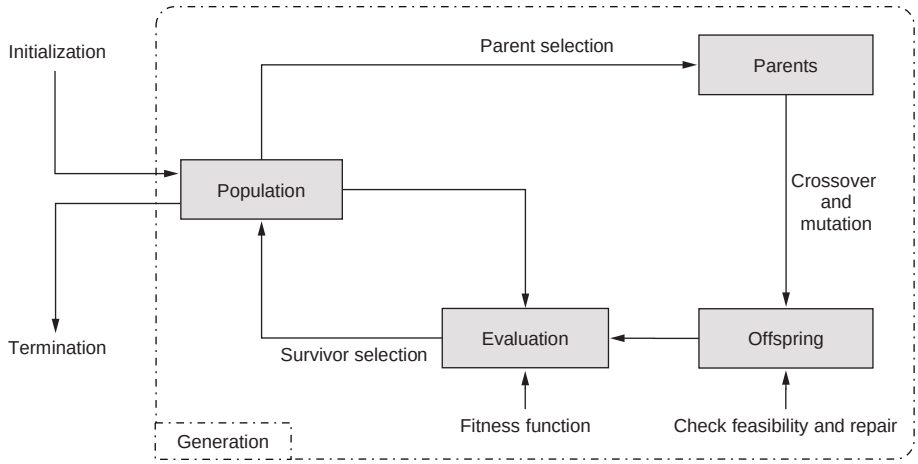


Figure 7.7 GA steps

Algorithm 7.1 summarizes the main steps of genetic algorithms.

Algorithm 7.1 Genetic algorithm

```
Initialization: Randomly generate an initial population M(0)

Evaluate all individuals: Compute and save the fitness f(m) for each
individual in the current population M(t)

While termination criteria are not met
    Select parents: Define selection probabilities p(m)for each individual p
in M(t)
    Apply crossover: Generate M(t+1) by probabilistically selecting
individuals from M(t) to produce offspring via genetic operators
    Apply mutation: Introduce random changes to individuals
    Evaluate: evaluate the fitness of the new individuals
    Select survivors: select individuals to form the next generation
```

The concept of GA is straightforward and easy to understand, as it emulates the process of natural evolution. It is a modular algorithm that can operate in parallel and can be easily distributed. GA is versatile and can handle multi-objective optimization problems effectively. It is particularly effective in noisy environments. GA is widely employed for tackling complex continuous and discrete optimization problems, and it excels in scenarios featuring numerous combinatorial parameters and nonlinear interdependencies among variables. Notably, as of the publication of this book in 2024, a search for "genetic algorithm" as a composite keyword returns approximately 100,000 results on Google Patent Search, while Google Scholar presents a staggering 1,940,000 results. This volume reflects the substantial interest in and diverse applications of genetic algorithms across academic and industrial domains.

7.3.1 Fitness function

As mentioned earlier, GA mimics nature's survival-of-the-fittest principle in a search process. Therefore, genetic algorithms are naturally suitable for solving maximization problems. However, various mathematical transformations can be used to convert minimization problems into maximization problems, such as these:

- *Negation transformation*—The simplest transformation is to negate the objective function. For example, maximizing a fitness function $f(x) = -O(x)$ is the same as minimizing the original objective function $O(x)$.
- *Reciprocal transformation*—Another way to convert a minimization problem into a maximization problem is to take the reciprocal of the objective function. This works only if the objective function is always non-negative. Equation 7.1 shows an example:

$$f(x) = \frac{1}{1 + O(x)} \tag{7.1}$$

- *Other mathematical transformations*—Equation 7.2 shows another transformation that converts an objective function in a minimization problem $O(x)$ into a fitness function in a maximization problem $f(x)$. In this equation, O_i is the objective function value of individual i, N is the population size, and V is a large value to ensure non-negative fitness values. The value of V can be the maximum value of the second term of the equation, so that the fitness value corresponding to the maximum value of the objective function is zero:

$$f(x) = V - \frac{O_i(x) \times N}{\sum_{i=1}^{N} O_i(x)} \tag{7.2}$$

According to the duality principal introduced in section 1.3.2, these transformations do not alter the location of the minima but convert a minimization problem to an equivalent maximization problem.

7.3.2 Representation schemes

An *encoding* is a data structure for representing candidate solutions, and a good encoding is probably the most important factor for the performance of GA. In GA, the parameters of a candidate solution (the genes) are concatenated to form a string (a chromosome). Binary encoding, real-value encoding, and permutation encoding can be used to encode the solution. Binary encoding is used in binary-coded GA (BGA) where the solution is represented as a binary string, as illustrated in figure 7.8.

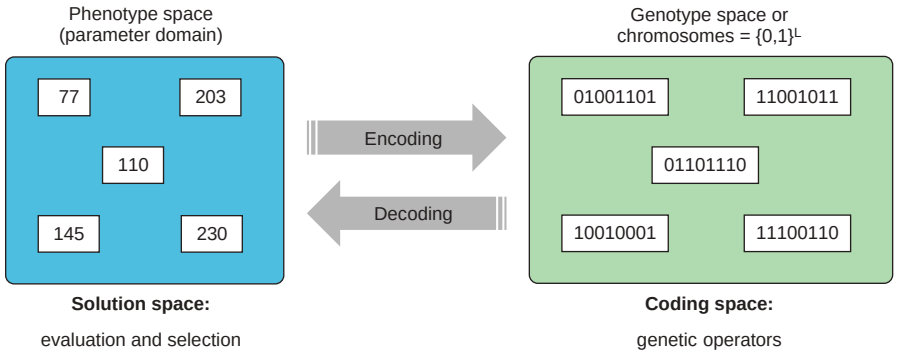


Figure 7.8 Binary encoding

Let's look again at the ticket pricing example introduced in section 1.3.1, where an event organizer is planning a conference and wants to determine the optimal ticket price to maximize the profit. The expected profit is given by the following equation:

$$\text{Profit} f(x) = -20x^2 + 6,200x - 350,000 \quad \textbf{7.3}$$

where x is the ticket price. The binary genetic algorithm (BGA) can be used to find the optimal ticket price to maximize the profit, subject to the boundary constraint $75.0 \leq x \leq 235.0$, which will make sure that profit is positive. BGA features a simple binary encoding. The boundary constraint on the preceding function requires us to use an 8-bit binary encoding, as explained in the following sidebar. Hence, the chromosomes are represented by bit strings of length 8.

Calculating the minimum number of bits for a solution

To calculate the number of bits required to represent a range between the lower bound (*LB*) and upper bound (*UB*) with a desired precision p, follow these steps:

1. Calculate the range size: $R = (UB - LB)$.
2. Divide the range size by the desired precision: R / P.
3. Round up to the nearest whole number: *number_of_steps* = ceil(R / P), where ceil is the ceiling function that rounds up to the nearest integer.
4. Calculate the number of bits: *number_of_bits* = ceil($\log_2$(*number_of_steps*)), where $\log_2$ is the logarithm to the base 2.

Let's calculate the number of bits we'll need for the ticket pricing problem: $75.0 \leq x \leq 235.0$, assuming a precision of 0.1:

1. Calculate the range size: (235.0 – 75.0) = 160
2. Divide the range size by the desired precision: 160 / 0.1 = 1600
3. Round up to the nearest whole number: 1600. Now you have 1600 steps (values) to represent the numbers from 75.0 to 235.0 with a precision of 0.1.

4 To find the minimum number of bits required, you can use the formula *number_of_bits* = ceil($\log_2$(*number_of_steps*)):
number_of_bits = ceil($\log_2$(1600)) ≈ ceil(10.64) = 11

So you'll need 11 bits to represent the numbers from 75.0 to 235.0 with a precision of 0.1. If you want to consider integer values only (i.e., a precision of 1), you would need ceil(logs(160)) = ceil(7.32) = 8 bits.

As mentioned previously, GA starts with an initial population of candidate solutions. Population size has to be carefully selected, as very big population size usually does not improve performance of GA. Some research also shows that the best population size depends on the size of encoded string (chromosomes). It means that if you have chromosomes with 32 bits, the population should be higher than for chromosomes with 16 bits.

In the ticket pricing problem, assume that we start with a population of size 5. Table 7.2 shows examples of random solutions that can be generated to form the initial population.

Table 7.2 Initial population

Candidate solutions *x*	Values of *x* in the solution space	Candidate solutions in the binary coding space	Objective function *f*(*x*)
x_1	77	01001101	8,820
x_2	203	11001011	84,420
x_3	110	01101110	90,000
x_4	145	10010001	128,500
x_5	230	11100110	18,000

Once we have an initial population, we can proceed to select the parents that will be subjected to genetic operators (crossover and mutation). We'll look at the selection operators next.

7.3.3 Selection operators

There are different methods (operators) for parent selection, and they have different levels of selective pressure. *Selective pressure* refers to the probability of the best individual being selected compared to the average probability of selection for all individuals. When using an operator with a high selective pressure in a genetic algorithm, the diversity within the population decreases at a faster rate than it would using operators with a lower selective pressure. This may sound good, but it can result in the population converging prematurely towards suboptimal solutions, thus limiting the exploration abilities of the population and eliminating individuals that do not fit the specific

criteria determined by the selective pressure. This can lead to a lack of diversity in the population, which reduces the chances of finding better solutions.

It is important to balance selective pressure with the exploration capabilities of the population to avoid premature convergence and to encourage the discovery of a diverse range of optimal solutions. Figure 7.9 illustrates some selection methods.

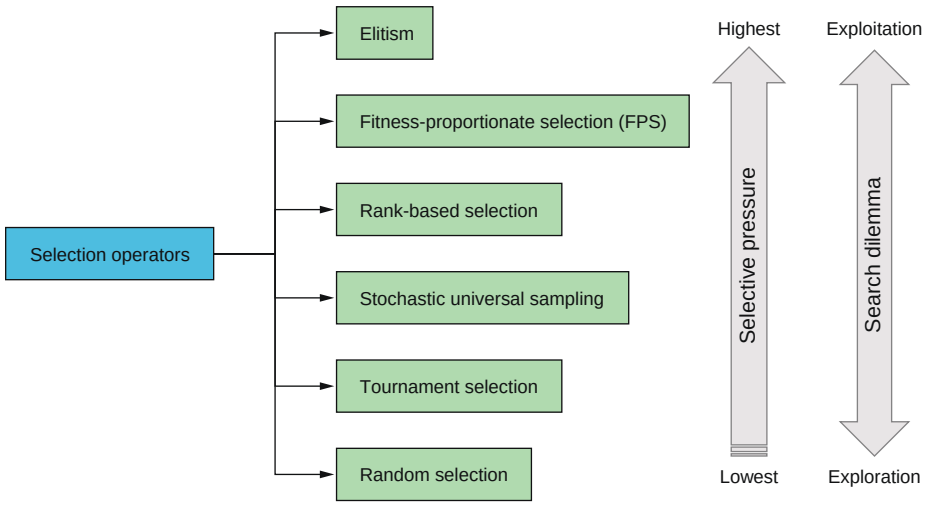


Figure 7.9 Selection methods with their selective pressure

ELITISM

Elitism in genetic algorithms involves selecting the fittest individuals for crossover and mutation and preserving the top-performing individuals of the current population to propagate into the next generation. The greater the number of individuals that are preserved, the lower the diversity of the succeeding population. This selection method has the highest selective pressure, as illustrated in figure 7.9.

In the ticket pricing example, the best solutions (x_4 and x_3) will be selected parents to generate offspring, as shown in table 7.3.

Table 7.3 Solution ranking

Candidate solutions x	Values of x in the solution space	Candidate solutions in the binary coding space	Objective function $f(x)$	Ranking
x_1	77	01001101	8,820	5
x_2	203	11001011	84,420	3
x_3	**110**	**01101110**	**90,000**	**2 (second-best individual)**
x_4	**145**	**10010001**	**128,500**	**1 (best individual)**
x_5	230	11100110	18,000	4

FITNESS-PROPORTIONATE SELECTION

Fitness-proportionate selection (FPS) is a selection method that favors the selection of the fittest individuals in a population. This method creates a probability distribution where the probability of an individual being selected is directly proportional to its fitness value. Individuals are chosen from this distribution by sampling it randomly. The individual fitness assignment relative to the whole population can be calculated as follows:

$$F(x_i) = \frac{f(x_i)}{\sum_{i=1}^{N} f(x_i)} \qquad 7.4$$

where f is the solution represented by an individual chromosome and N is the population size. Roulette wheel selection is an example of an FPS operator.

In our ticket pricing example, the roulette wheel can be constructed by implementing the following steps:

1. Calculate the total fitness for the population: F = 8,820 + 84,420 + 90,000 + 128,500 + 18,000 = 329,740.
2. Calculate the selection probability p_k for each chromosome x_k where $p_k = f(x_k) / F$. Table 7.4 shows the calculated selection probabilities.

Table 7.4 Selection probabilities

Candidate solutions x	Values of x in the solution space	Candidate solutions in the binary coding space	Objective function $f(x)$	Selection probability p_k
x_1	77	01001101	8,820	0.03
x_2	203	11001011	84,420	0.26
x_3	110	01101110	90,000	0.27
x_4	145	10010001	128,500	0.39
x_5	230	11100110	18,000	0.05

3. Calculate the cumulative probability q_k for each chromosome x_k where $q_k = \text{sum}(p_j)$, $j = \{1, k\}$. Table 7.5 shows the calculated cumulative probabilities.

Table 7.5 Cumulative probabilities

Candidate solutions x	Values of x in the solution space	Candidate solutions in the binary coding space	Objective function $f(x)$	Selection probability p_k	Cumulative probability q_k
x_1	77	01001101	8,820	0.03	0.03
x_2	203	11001011	84,420	0.26	0.28
x_3	110	01101110	90,000	0.27	0.56
x_4	145	10010001	128,500	0.39	0.95
x_5	230	11100110	18,000	0.05	1.00

4 Generate a random number r from the range [0,1].

5 If $q_1 >= r$, then select the first chromosome x_1; otherwise, select the k^{th} chromosome x_k ($2 \le k \le N$) such that $q_{k-1} < r \le q_k$. If we assume that the randomly generated number $r = 0.25$, then x_2 with $q_2 = 0.28$ is selected because $q_2 > 0.25$, and if $r = 0.58$, $x4$ will be selected because $q_4 > 0.58$. Figure 7.10 illustrates the roulette wheel for the ticket pricing example.

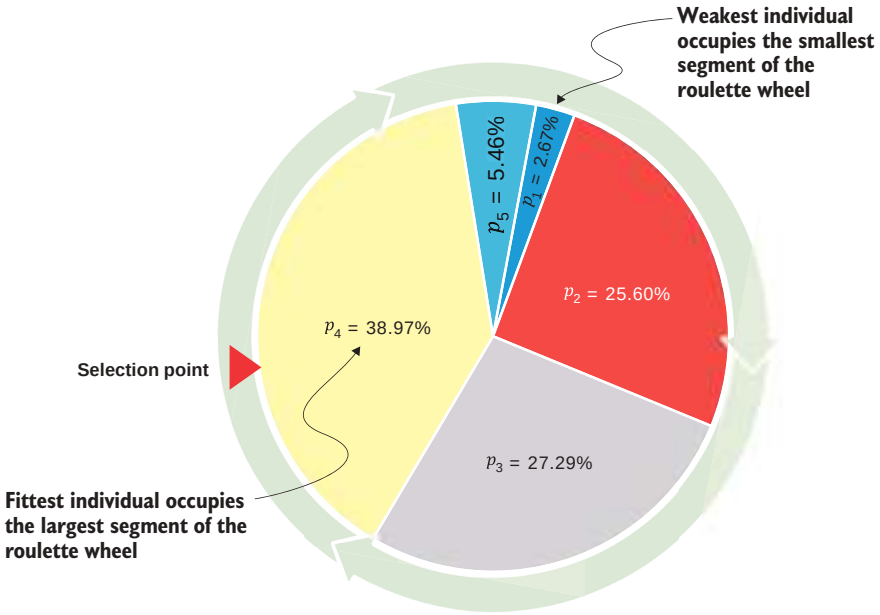


Figure 7.10 Roulette wheel for the ticket pricing example

As you can see, the fittest individual occupies the largest segment of the roulette wheel, and the weakest individual occupies the smallest segment of the wheel. Due to the direct correlation between fitness and selection in proportional selection, there is a potential for dominant individuals to disproportionately contribute to the next generation's offspring, leading to a reduction in the diversity of the population. This implies that proportional selection results in a high selective pressure.

RANK-BASED SELECTION

One way to address the limitations of FPS in genetic algorithms is to use relative fitness instead of absolute fitness to determine selection probabilities—individuals are selected based on their fitness relative to the fitness of other individuals in the population. This approach ensures that the selection process is not dominated by the best individual in the population.

Linear ranking and nonlinear ranking can be used. In *linear ranking*, the rank-based probability of an individual i being selected is calculated using the following equation:

$$p(i) = (2 - SP) + (SP - 1)\frac{r(i) - 1}{(N - 1)} \quad \textbf{7.5}$$

where N is the size of the population, SP is the selection pressure $(1.0 < SP \leq 2.0)$, and $r(i)$ is the rank associated with individual i (a higher rank is better). In the ticket pricing example, where $N = 5$, and assuming that $SP = 1.5$, the rank-based selection probability of each individual in the population is shown in table 7.6.

Table 7.6 Rank-based selection probabilities

Candidate solutions x	Values of x in the solution space	Candidate solutions in the binary coding space	Objective function $f(x)$	Rank r_i	FPS cumulative probability q_k	Rank-based selection probability
x_1	77	01001101	8,820	1	0.03	0.50
x_2	203	11001011	84,420	3	0.28	0.75
x_3	110	01101110	90,000	4	0.56	0.88
x_4	145	10010001	128,500	5	0.95	1.00
x_5	230	11100110	18,000	2	1.00	0.63

As you can see, rank-based selection reduces the bias of FPS by assigning greater probabilities of selection to less-fit individuals.

Nonlinear ranking permits higher selective pressures than linear ranking does. The selection probability is calculated using the following equation:

$$p(i) = \frac{N.X^{i-1}}{\sum_{i=1}^{N} X^{i-1}} \quad \textbf{7.6}$$

where X is computed as the root of the polynomial $(SP - N).X^{N-1}) + SP.X^{N-2} + \ldots + SP.X + SP = 0$. This nonlinear ranking allows values of selective pressure in the interval $[1, N - 2]$.

STOCHASTIC UNIVERSAL SAMPLING

Stochastic universal sampling (SUS) is another approach to mitigating the potential bias in the roulette-wheel selection approach. This method involves placing an outer roulette wheel around the pie with m evenly spaced pointers. With SUS, a single spin of the roulette wheel is used to simultaneously select all m individuals for reproduction. Figure 7.11 shows SUS for the ticket pricing problem using four selection points.

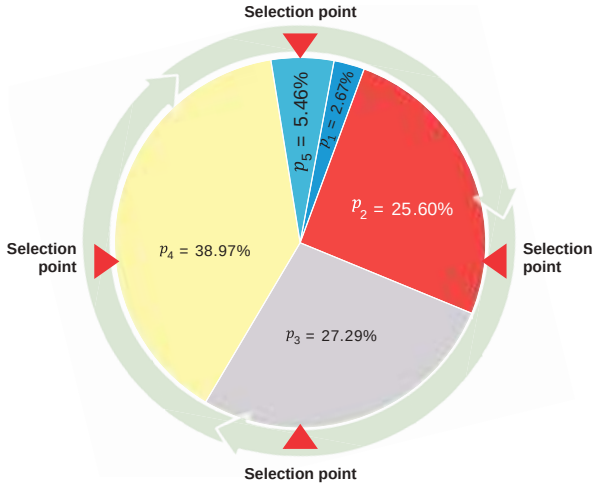


Figure 7.11 Stochastic universal sampling (SUS) strategy

TOURNAMENT SELECTION

Tournament selection involves randomly selecting a group of k individuals from the current population, where k is the size of the tournament group. Once the group is formed, a tournament is held among its members to identify the best-performing individual based on their fitness values. The individual with the highest fitness score is the winner and advances to the next stage of the genetic algorithm. Figure 7.12 shows the tournament selection process.

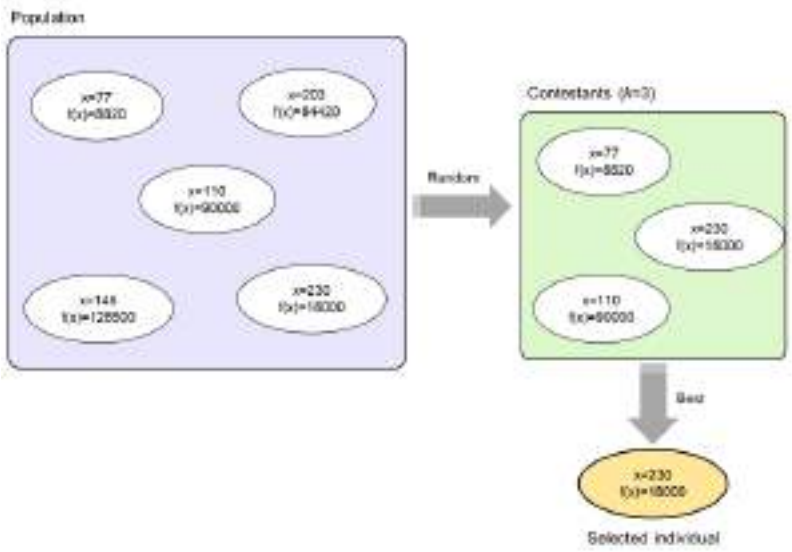


Figure 7.12 Tournament selection

To select m individuals for reproduction, the tournament procedure is carried out m times. In each iteration, a new tournament group is randomly chosen from the population, and the individuals in the group compete against each other until the best-performing individual is identified. The winners from each tournament are then selected for reproduction, which involves applying genetic operators such as crossover and mutation to create new offspring.

RANDOM SELECTION

Random selection is the simplest selection operator, where each individual has the same selection probability of $1/N$ (where N is the population size). No fitness information is used, which means that the best and the worst individuals have exactly the same probability of being selected. Random selection has the lowest selective pressure among the selection operators, as all individuals within the population have the same chance of being selected.

OTHER SELECTION METHODS

Other selection methods include, but are not limited to, Boltzmann Selection, (μ, λ)- and $(\mu + \lambda)$-selection, and hall of fame. The random selection and tournament selection methods are implemented as part of the `pymoo.operators.selection` class in pymoo.

After we select the parents, we need to produce offspring by applying reproduction operators.

7.3.4 Reproduction operators

Genetic algorithms employ two primary genetic operators, namely crossover and mutation, to generate offspring. Let's look at these two reproduction operators in detail.

CROSSOVER

Crossover is inspired by the biological process of recombination, where a portion of the genetic information is exchanged between two chromosomes. This exchange of genetic material results in the production of offspring, so two parents can thus give rise to two offspring. In order to ensure that the best individuals are able to contribute their genetic material, superior individuals are typically given more opportunities to reproduce through crossover. This mechanism promotes the effective combination of schemata, which are subsolutions located on different chromosomes. 1-point crossover, n-point crossover, and uniform crossover are commonly used crossover methods.

In *1-point crossover*, we start by choosing a random point on the two parents and splitting parents at this crossover point. Two children are then created by exchanging tails, as illustrated in figure 7.13. This crossover operation produces two new children (candidate solutions), which in the figure are 01010001 and 01001101 (or 81 and 77 in decimal, respectively) as potential ticket prices. These solutions result in total profits of $20,980 and $8,820 respectively, based on equation 7.3.

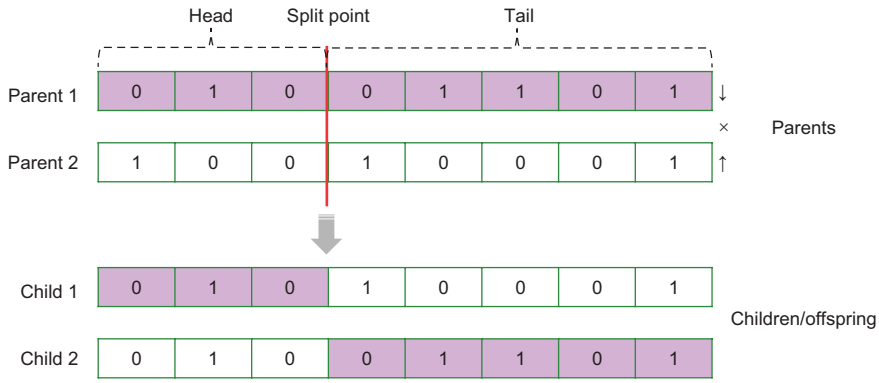


Figure 7.13 1-point crossover

In *n-point crossover*, which is a generalization of 1-point crossover, we choose n random crossover points and split along those points. The children are generated by gluing parts together and alternating between parents, as illustrated in figure 7.14. Following the 2-point crossover illustrated in figure 7.14, two candidate solutions are generated, which are 141 and 81 with fitness values of $126,580 and $20,980 respectively.

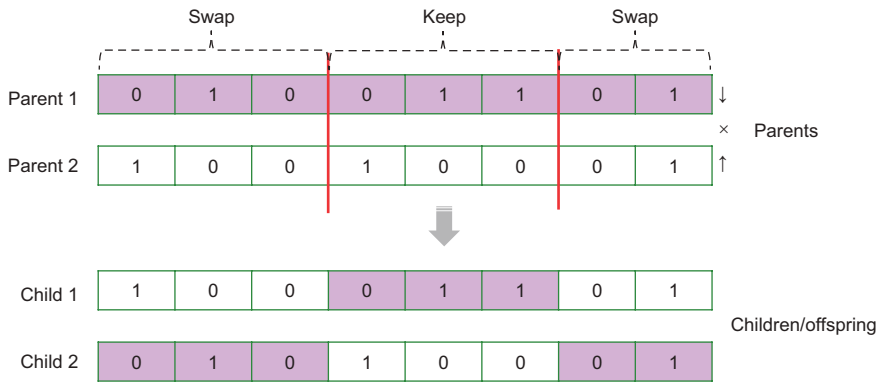


Figure 7.14 *n*-point crossover

In *uniform crossover*, random bit positions from two parents are swapped to create two offspring. One parent is assigned the label "heads" and the other "tails." For the first child, a coin is flipped for each gene to determine whether it should come from the "heads" or "tails" parent. The second child is created by taking the inverse of each gene in the first child, as shown in figure 7.15. In this example, applying uniform crossover results in 217 and 5 with fitness values of $53,620 and $–319,500. As you can see, 5 is not a feasible solution because it is not within the boundary constraints of {75.0,235.0}. This solution is rejected.

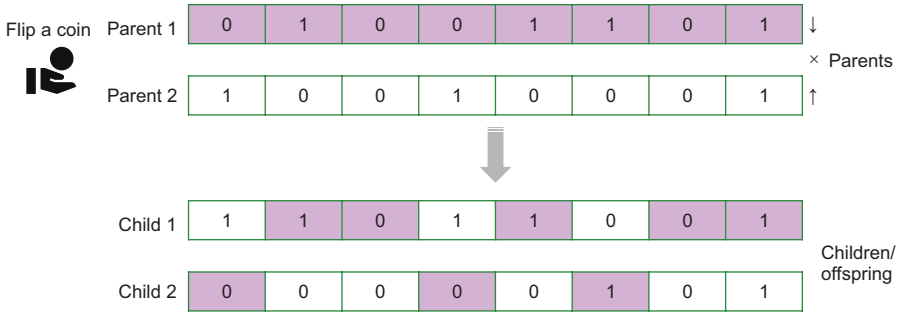


Figure 7.15 Uniform crossover

In pymoo, the repair operator can be used make sure the algorithm only searches in the feasible space. It is applied after the offspring have been produced.

Mutation

Mutation is a process that introduces new genetic material into an individual, which helps to increase the diversity of the population. This diversity is important because it allows the population to explore a wider range of possible solutions to the problem at hand. Mutation is often used in combination with crossover to ensure that the full range of alleles is accessible for each gene. In the case of mutation, selection mechanisms could focus on "weak" individuals in the hope that mutation will introduce better traits to those individuals, increasing their chances of survival.

In binary genetic algorithms, mutation is performed by altering each gene independently with a probability p_m. For each gene, we generate a random number r between 0 and 1. If $p_m > r$, we alter the gene. Figure 7.16 illustrates mutating one of the individuals of the ticket pricing problem.

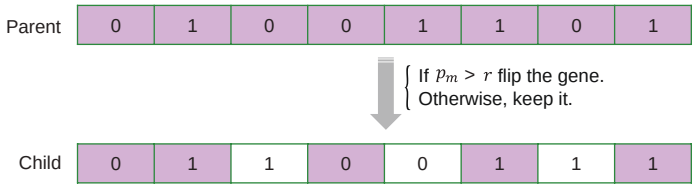


Figure 7.16 Mutation

New population

After applying crossover and mutation, we will have new offspring that represent new candidate solutions. To start a new generation, we need to create a new population by selecting individuals from the old population and from the newly generated offspring. The size of the new population will remain the same as the old population.

Generational GA and steady-state GA are two models used in genetic algorithms. As shown in figure 7.17, in *generational GA* models, the whole population is replaced by its offspring to start a "next generation." In *steady-state GA*, the number of generated offspring is less than the population size. Old individuals may be replaced by new ones. The process of selecting individuals for the new population is known as *survivor selection*. We'll look at survivor selection methods next.

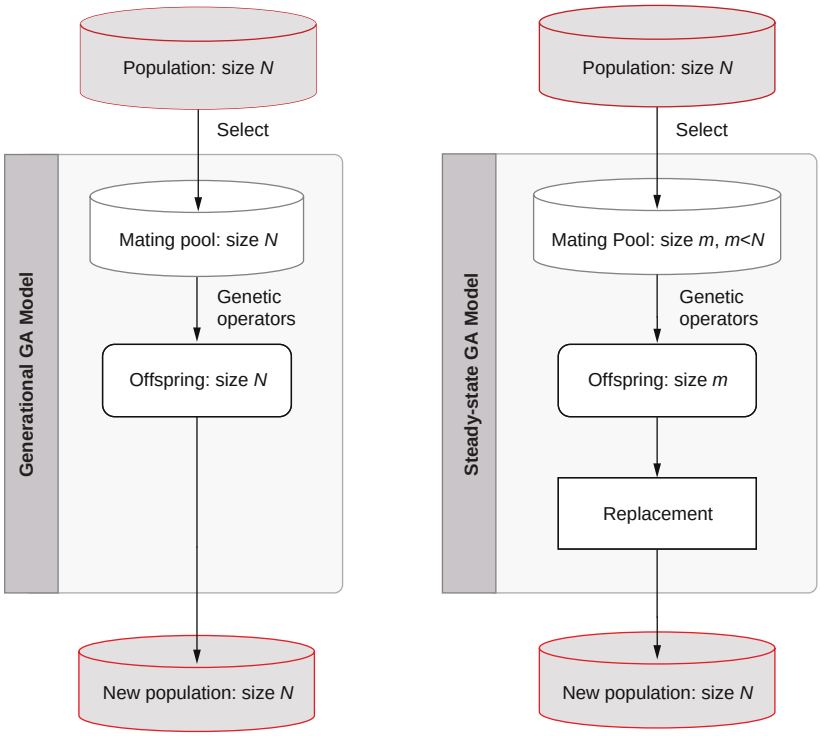


Figure 7.17 GA generational and steady-state models

7.3.5 Survivor selection

Random selection, age-based selection, fitness-proportionate selection, and tournament selection are examples of survivor selection methods that can preserve the best individuals while also introducing diversity to a population by making use of the newly generated offspring:

- In *random selection*, the new population is formed by random selection of N individuals.
- With *age-based selection* (or first in, first out), the oldest individuals will be deleted.

- *Fitness-proportionate selection* (FPS) takes into consideration the fitness of each individual—we can delete or replace individuals based on the inverse of fitness, always keeping the best individuals or deleting the worst individuals. For example, *elitist selection* involves simply selecting the best individuals from both the old population and the new offspring to create the new population. This method ensures that the best solutions are preserved from generation to generation.
- Tournament selection involves selecting individuals from both the old population and the new offspring at random and then selecting the best individuals from each group to create the new population. This method can be more effective at preserving diversity in the population.

In the ticket pricing example, if we apply elitist selection, the new population will be formed by the selected solutions shown in table 7.7.

Table 7.7 Elitist selection

Source	Candidate solutions x in the solution space	Candidate solutions in the binary coding space	Objective function f(x)	Ranking	Selected
Old individuals	77	01001101	8,820	7	No
	203	11001011	84,420	3	Yes
	110	01101110	90,000	2	Yes
	145	10010001	128,500	1	Yes
	230	11100110	18,000	6	No
New individuals generated by 1-point crossover	81	01010001	20,980	5	Yes
	77	01001101	8,820	7	No
New individuals generated by mutating individual 77	103	01100111	76,420	4	Yes

You may have noticed that 1-point crossover generated a solution that already exists in the initial population. This phenomenon is not necessarily a problem, as it is an expected outcome when applying genetic operators in a search process. Crossover and mutation can result in both explorative and exploitative behaviors. For example, in 1-point or *n*-point crossover and based on the random split point position, a new solution can be the same as or close to the parents or can generate more diverse offspring.

7.4 Implementing genetic algorithms in Python

A genetic algorithm is an easy algorithm to implement. Let's see how we can solve the ticket pricing problem using GA in Python.

We'll start by importing the necessary packages and defining the problem.

Listing 7.2 Solving the ticket pricing problem using binary GA

```
import numpy as np
import random
from tqdm.notebook import tqdm
from copy import copy
import matplotlib.pyplot as plt

def profit(x):
    return -20*x*x+6200*x-350000
```

Because we're solving this problem using a binary GA, we need to generate an initial random population. As a continuation of listing 7.2, the following `init_pop` function takes two arguments as input—`pop_size`, which represents the population size, and `chromosome_length`, which represents the length of each chromosome:

```
def init_pop(pop_size, chromosome_length):
    ints = [random.randint(75,235) for i in range(pop_size)]
    strs = [bin(n)[2:].zfill(chromosome_length) for n in ints]
    bins = [[int(x) for x in n] for n in strs]
    return bins
```

Generate a list of random integers.

Convert the integers to binary strings.

Convert the binary strings to lists of binary digits.

Return the final list of binary chromosomes.

The `init_pop` function starts by generating a list of random integers from 75 to 235 (inclusive) with a length equal to `pop_size`. This list will later be converted to binary representations. The integers in the `ints` list are then converted to binary strings using the `bin()` function, which returns a binary string representation of a given number with the prefix `0b`. To remove this prefix, we use slicing with `[2:]`. Then we use the `zfill()` method to pad the binary string with leading zeros to ensure it has the same length as `chromosome_length`. The binary strings in the `strs` list are converted to lists of binary digits (0 or 1). This is done using a nested list comprehension that iterates through each character in the binary strings and converts it to an integer. The function finally returns a list of binary chromosomes, where each chromosome is a list of binary digits.

For a given population, we can calculate the fitness of each element in the population using the following `fitness_score` function. This fitness function essentially determines how "good" a particular offspring is. It converts each unit in the population to a binary number (the genotype), evaluates the function to optimize profit, and then returns the "best" offspring. The function mainly takes a population as input and returns a tuple containing two lists, one of the sorted fitness values and another of the sorted population:

```
def fitness_score(population):
    fitness_values = []
    num = []
    for i in range(len(population)):
```

```
        num.append(int("".join(str(x) for x in population[i]), base=2))
        fitness_values.append(profit(num[i]))
    tuples = zip(*sorted(zip(fitness_values, population),reverse=True))
    fitness_values, population = [list(t) for t in tuples]
    return fitness_values, population
```

Convert binary to decimal.

Evaluate the fitness of each chromosome and append the fitness value to the fitness_values list.

Create tuples of fitness values and their corresponding chromosomes, and then sort them in descending order based on fitness values.

Unzip the sorted tuples back into separate lists for fitness values and the population.

Return the sorted fitness values and population.

Let's now select two parents using the random selection method implemented in the following `select_parent` function. This function takes two arguments as input: `population`, which is a list of individuals in the population, and `num_parents`, which represents the number of parents to select. It returns a list of selected parents:

```
def select_parent(population, num_parents):
    parents=random.sample(population, num_parents)
    return parents
```

Randomly select a specified number of unique parents from the given population.

Return the list of selected parents.

The `select_parent` function implements a simple random sampling selection method, which gives each individual in the population an equal chance of being selected as a parent. Other selection methods, such as FPS or roulette wheel selection, can also be used to give higher chances to individuals with better fitness values.

The following `roulette_wheel_selection` function shows the steps of roulette wheel selection. The function takes two arguments as input—`population`, which is a list of individuals in the population, and `num_parents`, which represents the number of parents to select:

```
def roulette_wheel_selection(population, num_parents):
    fitness_values, population = fitness_score(population)
    total_fitness = sum(fitness_values)
    probabilities = [fitness / total_fitness for fitness in fitness_values]

    selected_parents = []
    for i in range(num_parents):
        r = random.random()

        cumulative_probability = 0
        for j in range(len(population)):
            cumulative_probability += probabilities[j]
            if cumulative_probability > r:
                selected_parents.append(population[j])
                break

    return selected_parents
```

Calculate total fitness.

Calculate selection probabilities for each individual.

Select only two parents.

Generate a random number *r* between 0 and 1.

Find the individual whose cumulative probability includes *r*.

After selecting the parents, it's time to apply genetic operators to produce the offspring. The following `crossover` function implements 1-point crossover. The function takes two arguments as input: `parents`, which is a list of two parent chromosomes, and `crossover_prob`, which represents the probability of crossover occurring between the parents. It returns a list of parents and offspring. The first offspring is generated by taking the first part (up to and including the crossover point) of the first parent and the second part (from the crossover point + 1 to the end of the chromosome) of the

second parent. Similarly, the second offspring is generated by taking the first part (up to and including the crossover point) of the second parent and the second part (from the crossover point + 1 to the end of the chromosome) of the first parent:

```
def crossover(parents, crossover_prob):
    chromosome_length = len(parents[0])
    if crossover_prob > random.random():
        cross_point = random.randint(0,chromosome_length)
        parents+=tuple([(parents[0][0:cross_point+1] +parents[1][cross_
point+1])])
        parents+=tuple([(parents[1][0:cross_point+1] +parents[0][cross_
point+1])])
    return parents
```

Choose a random crossover point within the range of chromosome indices.

Apply crossover if, and only if, crossover probability is greater than a randomly generated number.

Create the first offspring.

Create the second offspring.

Return the original parents and the new offspring generated by the crossover operation.

Let's now apply the mutation process. The following `mutation` function performs mutation operations on a given population of chromosomes. It takes two arguments as input: `population`, which is a list of binary chromosomes, and `mutation_prob`, which represents the probability of mutation occurring at each gene in the chromosomes. It returns the mutated population:

```
def mutation(population, mutation_prob) :
    chromosome_length = len(population[0])
    for i in range(len(population)):
        for j in range(chromosome_length-1):
            if mutation_prob > random.random():
                if population[i][j]==1:
                    population[i][j]=0
                else:
                    population[i][j]=1
    return population
```

Iterate through each chromosome in the population.

Iterate through each gene in the chromosome, except the last one.

Apply mutation if, and only if, the mutation probability is greater than a randomly generated number.

Flip the value of the gene.

Return the mutated population.

Let's now put everything together and define the binary genetic algorithm (`BGA`) function. This function takes the following arguments as input:

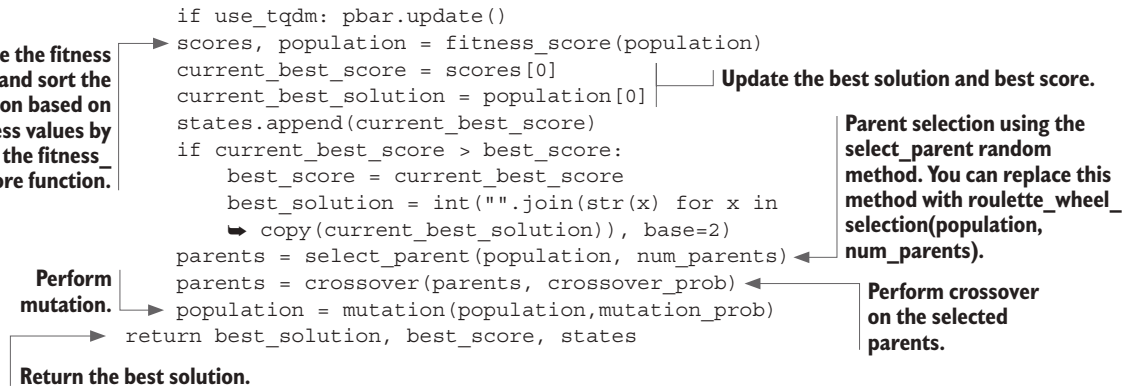
- `population`—The initial population of binary chromosomes
- `num_gen`—The number of generations the algorithm will run for
- `num_parents`—The number of parents to select for crossover
- `crossover_prob`—The probability of crossover occurring between parents
- `mutation_prob`—The probability of mutation occurring at each gene
- `use_tqdm` (optional, default=`False`)—A Boolean flag to enable or disable a progress bar using the tqdm library

This is the BGA function:

```
def BGA(population, num_gen, num_parents, crossover_prob, mutation_prob, use_
tqdm =
➥ False):
    states = []
    best_solution = []
    best_score = 0
    if use_tqdm: pbar = tqdm(total=num_gen)
    for _ in range(num_gen):
```

Initialization

Run the genetic algorithm for num_gen generations using a for loop.




```
        if use_tqdm: pbar.update()
        scores, population = fitness_score(population)
        current_best_score = scores[0]
        current_best_solution = population[0]
        states.append(current_best_score)
        if current_best_score > best_score:
            best_score = current_best_score
            best_solution = int("".join(str(x) for x in
            ➥ copy(current_best_solution)), base=2)
        parents = select_parent(population, num_parents)
        parents = crossover(parents, crossover_prob)
        population = mutation(population,mutation_prob)
    return best_solution, best_score, states
```

This function returns the best solution, the best score, and the list of best scores at each generation.

Now we can solve the ticket pricing problem, starting with generating an initial population with the following parameters:

```
num_gen = 1000
pop_size = 5
crossover_prob = 0.7
mutation_prob = 0.3
num_parents = 2

chromosome_length = 8
best_score = -100000

population = init_pop(pop_size, chromosome_length)
print("Initial population: \n", population)
```

Running this code produced the following initial population:

```
Initial population: [[1, 1, 1, 0, 1, 0, 0, 0], [1, 0, 0, 0, 0, 1, 1, 0], [1,
1, 0, 1, 1, 1, 0, 1], [1, 1, 0, 0, 0, 0, 1, 1], [1, 0, 1, 0, 0, 0, 0, 0]]
```

We can now run the binary GA solver to get the solutions, as follows:

```
best_solution, best_score, states = BGA(population, num_gen, num_parents,
➥ crossover_prob, mutation_prob, use_tqdm=True)
```

Running this code produces the same solution obtained by the SciPy optimizer (see listing 2.4):

```
Optimal ticket price ($): 155
Profit ($): 130500
```

Rather than writing your own genetic algorithm code from scratch, you can take advantage of existing Python packages that offer GA implementations. Numerous open source Python libraries can help streamline the development process and save time. These libraries often include genetic operators, selection methods, and other features that make it easier to adapt a genetic algorithm to different optimization problems. Examples of these libraries include, but are not limited to, the following:

- *Pymoo* (Multi-objective Optimization in Python; https://pymoo.org/algorithms/moo/nsga2.html)—A Python library for multi-objective optimization using evolutionary algorithms and other metaheuristic techniques. Pymoo offers a variety of algorithms such as GA, differential evolution, evolutionary strategy, nondominated sorting genetic algorithm (NSGA-II), NSGA-III, and particle swarm optimization (PSO).
- *DEAP* (Distributed Evolutionary Algorithms in Python; https://deap.readthedocs.io/en/master/—A Python library for implementing genetic algorithms in Python. It provides tools for defining, training, and evaluating genetic algorithm models, as well as for visualizing the optimization process. DEAP provides a variety of built-in genetic operators, including mutation, crossover, and selection, as well as support for custom operators tailored to specific optimization problems.
- *PyGAD* (Python Genetic Algorithm; https://pygad.readthedocs.io/en/latest/)—A Python library for implementing genetic algorithms and differential evolution (DE) algorithms. PyGAD is suitable for both single-objective and multi-objective optimization tasks and can be used in a wide range of applications, including machine learning, and other problem domains.
- *jMetalPy* (https://github.com/jMetal/jMetalPy)—A Python library designed for developing and experimenting with metaheuristic algorithms for solving multi-objective optimization problems. It provides support for a variety of metaheuristic algorithms, including popular evolutionary algorithms like non-dominated sorting genetic algorithm (NSGA-II), NSGA-III, strength Pareto evolutionary algorithm (SPEA2), and multi-objective evolutionary algorithm based on decomposition (MOEA/D), as well as other optimization techniques such as simulated annealing and particle swarm optimization.
- *PyGMO* (Python Parallel Global Multi-objective Optimizer; https://esa.github.io/pygmo/)—A scientific library providing a large number of optimization problems and algorithms such as NSGA-II, SPEA2, non-dominated sorting particle swarm optimization (NS-PSO), and parameter adaptive differential evolution (PaDE). It uses the generalized island-model paradigm for the coarse grained parallelization of optimization algorithms and, therefore, allows users to develop asynchronous and distributed algorithms.
- *Inspyred* (Bio-inspired Algorithms in Python; https://pythonhosted.org/inspyred/)—A library for creating and working with bio-inspired computational intelligence algorithms. It supports a variety of bio-inspired optimization algorithms, such as GA, evolution strategy, simulated annealing, differential evolution algorithm, estimation of distribution algorithm, Pareto archived evolution strategy (PAES), nondominated sorting genetic algorithm (NSGA-II), particle swarm optimization (PSO), and ant colony optimization (ACO).
- *Platypus* (https://platypus.readthedocs.io/en/latest/)—A framework for evolutionary computing in Python with a focus on multi-objective evolutionary

algorithms (MOEAs). It provides tools for analyzing and visualizing algorithm performance and solution sets.

- *MEALPY* (https://mealpy.readthedocs.io/en/latest/index.html)—A Python library that provides implementations for population-based meta-heuristic algorithms such as evolutionary computing algorithms, swarm inspired computing, physics inspired computing, human inspired computing, and biology inspired computing.
- *Mlrose* (Machine Learning, Randomized Optimization and Search; https://mlrose.readthedocs.io/en/stable/index.html)—An open source Python library that provides an implementation of standard GA to find the optimum for a given optimization problem.
- *Pyevolve* (https://pyevolve.sourceforge.net/)—An open source Python library designed for working with genetic algorithms and other EC techniques
- *EasyGA* (https://github.com/danielwilczak101/EasyGA)—A Python package designed to provide an easy-to-use GA. It's worth noting that EasyGA and Pyevolve are simple libraries with less functionality and predefined problems than other libraries such as DEAP and Pymoo.

Listing A.3, available in the book's GitHub repo, shows how to use some of these libraries.

In this book, we will focus on utilizing the pymoo library, as it is a comprehensive framework that offers several optimization algorithms, visualization tools, and decision-making capabilities. This library is particularly well-suited for multi-objective optimization, which we'll explore in more detail in the next chapter. Pymoo's extensive features make it an excellent choice for implementing and analyzing genetic algorithms in various problem domains. Table 7.8 summarizes a comparative study of selected evolutionary computing frameworks, including pymoo [2].

Table 7.8 Comparing selected evolutionary computing frameworks in Python

Library	License	Pure Python	Visualization	Focus on multi-objective	Decision making
jMetalPy	MIT	Yes	Yes	Yes	No
PyGMO	GPL-3.0	No (C++ with Python wrappers)	No	Yes	No
Platypus	GPL-3.0	Yes	No	Yes	No
DEAP	LGPL-3.0	Yes	No	No	No
inspyred	MIT	Yes	No	No	No
pymoo	Apache 2.0	Yes	Yes	Yes	Yes

The following listing shows the steps for solving the ticket pricing problem using GA implemented in pymoo. We'll start by importing various classes and functions from the pymoo library.

Listing 7.3 Solving the ticket pricing problem using GA in pymoo

```
from pymoo.algorithms.soo.nonconvex.ga import GA
from pymoo.operators.crossover.pntx import PointCrossover,
➥ SinglePointCrossover,
➥ TwoPointCrossover
from pymoo.operators.mutation.pm import PolynomialMutation
from pymoo.operators.repair.rounding import RoundingRepair
from pymoo.operators.sampling.rnd import FloatRandomSampling
from pymoo.core.problem import Problem
from pymoo.optimize import minimize
```

The `GA` class represents a single-objective genetic algorithm in the pymoo library. The `PointCrossover`, `SinglePointCrossover`, and `TwoPointCrossover` classes represent different crossover operators for combining the genetic material of parent chromosomes to create offspring. The `PolynomialMutation` class represents a mutation operator that introduces small, random changes in the chromosomes' genes. The `RoundingRepair` class represents a repair operator that rounds the variable values of the chromosomes, ensuring that they stay within a specific range or meet certain constraints. The `FloatRandomSampling` class represents a random sampling operator that generates an initial population of chromosomes with random float values. The `Problem` class is used to define optimization problems by specifying objectives, constraints, and variable bounds. Finally, the `minimize` function is used to perform the optimization process. It is worth noting that pymoo can only handle minimization problems, so if you need to use it with a maximization problem, you'll have to convert the problem into a minimization problem, as discussed in section 7.3.1.

After importing the necessary classes and functions from the pymoo library, we can define the `TicketPrice` problem by subclassing the `Problem` class from the pymoo library as follows:

```
class TicketPrice(Problem):
    def __init__(self):
        super().__init__(n_var=1,
                         n_obj=1,
                         n_constr=0,
                         xl=75.0,
                         xu=235.0, vtype=float)

    def _evaluate(self, x, out, *args, **kwargs):
        out["F"]= 20*x*x-6200*x+350000
```

Define the constructor for the TicketPrice class.

Call the constructor of the parent Problem class.

Define the evaluation function for the TicketPrice class.

Evaluate the value of the objective function using the given formula.

As can be seen, the constructor of the parent `Problem` class contains the following components with customized values applied to the ticket pricing problem:

- `n_var=1`—The number of decision variables in the problem, which is set to 1, indicating a single decision variable for the ticket price.
- `n_obj=1`—The number of objectives in the problem, which is set to 1, indicating a single-objective optimization problem.

- `n_constr=0`—The number of constraints in the problem, which is set to 0, indicating that there are no constraints in this optimization problem.
- `xl=75.0`—The lower bound for the decision variable, which is set to 75.0.
- `xu=235.0`—The upper bound for the decision variable, which is set to 235.0.
- `vtype=float`—The variable type for the decision variables, which is set to float. Other types include `int` and `bool`.

Now we can apply GA to solve the problem as follows:

```
problem = TicketPrice()
algorithm = GA(
    pop_size=100,
    sampling=FloatRandomSampling(),
    crossover=PointCrossover(prob=0.8, n_points=2),
    mutation = PolynomialMutation(prob=0.3, repair=RoundingRepair()),
    eliminate_duplicates=True
)

res = minimize(problem, algorithm, ('n_gen', 100), seed=1, verbose=True)

print(f"Optimal ticket price ($): {res.X}")
print(f"Profit ($): {-res.F}")
```

Create an instance of the TicketPrice problem.

Instantiate a GA object.

Run the solver.

Print the optimal ticket price.

Print the profit. Negate the objective value when printing the result.

GA parameters include the following:

- `pop_size=100`—Set the population size to 100 individuals.
- `sampling=FloatRandomSampling()`—Use the `FloatRandomSampling` class to generate an initial population of chromosomes with random float values.
- `crossover=PointCrossover(prob=0.8, n_points=2)`—Use the `PointCrossover` class as the crossover operator with a probability of 0.8 and two crossover points.
- `mutation=PolynomialMutation(prob=0.3, repair=RoundingRepair())`—Use the `PolynomialMutation` class as the mutation operator with a probability of 0.3, and apply the `RoundingRepair` class to repair mutated solutions if needed. The repair makes sure every solution that is evaluated is, in fact, feasible.
- `eliminate_duplicates=True`—Set the flag to eliminate duplicate individuals in the population.
- `res = minimize(...)`—Call the minimize function from pymoo to run the optimization process.

Running listing 7.3 produces the following output:

```
Optimal ticket price ($): [155]
Profit ($): [130500.]
```

So far, we've only scratched the surface of genetic algorithms. We'll dive into the details, study different variants of genetic algorithms, and address more practical use cases in the next chapter.

Summary

- Metaheuristic algorithms that are population-based, often referred to as P-metaheuristics, employ multiple agents to find an optimal or near-optimal global solution. These algorithms can be divided into two main categories, depending on their source of inspiration: evolutionary computation (EC) algorithms and swarm intelligence (SI) algorithms.
- EC algorithms draw inspiration from the process of biological evolution. Examples of EC algorithms include the genetic algorithm (GA), differential evolution (DE), genetic programming (GP), evolutionary programming (EP), evolutionary strategies (ES), cultural algorithms (CA), and co-evolution (CoE).
- The genetic algorithm is the most widely used form of EC. It is an adaptive heuristic search method designed to mimic the natural system's processes required for evolution, as outlined in Charles Darwin's theory of evolution.
- Pseudo-random strategies, quasi-random strategies, sequential diversification, parallel diversification, and heuristics represent various initialization strategies for P-metaheuristics like genetic algorithms. Each strategy offers distinct levels of diversity, computational cost, and initial solution quality.
- In genetic algorithms, the crossover and mutation operators play essential roles in searching the solution space and maintaining diversity within the population. The primary purpose of these operators is to handle the search dilemma by balancing exploration (searching new areas of the solution space) and exploitation (refining the existing solutions).
- A high crossover rate and a low mutation rate are recommended to balance exploration and exploitation. The high crossover rate facilitates the sharing of good traits between individuals, while the low mutation rate introduces small, random changes to maintain diversity and prevent premature convergence. This combination allows the algorithm to efficiently search the solution space and find high-quality solutions.
- In the generational model of genetic algorithms, the entire population is replaced, whereas in the steady-state model of genetic algorithms, a small fraction of the population is replaced. The steady-state model has lower computation costs than the generational model in genetic algorithms, but the generational model improves diversity preservation compared to the steady-state models.
- A wide range of open source Python libraries exist for working with genetic algorithms. One such library, pymoo (Multi-objective Optimization in Python), includes popular algorithms such as genetic algorithms, differential evolution, evolutionary strategies, the non-dominated sorting genetic algorithm (NSGA-II), NSGA-III, and particle swarm optimization (PSO).