8 Genetic algorithm variants

This chapter covers

- Introducing the Gray-coded genetic algorithm
- Understanding real-valued GA and its genetic operators
- Understanding permutation-based GA and its genetic operators
- Understanding multi-objective optimization
- Adapting genetic algorithms to strike a balance between exploration and exploitation
- Solving continuous and discrete problems using GA

This chapter continues with the topic of chapter 7: we will look at various forms of genetic algorithms (GAs) and delve deeper into their real-world applications. We'll also look at a number of case studies and exercises, such as the traveling salesman problem (TSP), proportional integral derivative (PID) controller design, political districting, the cargo bike loading problem, manufacturing planning, facility allocation, and the opencast mining problem in this chapter and its supplementary exercises included in the online appendix C.

8.1 Gray-coded GA

The Hamming cliff effect refers to the fact that small changes in a chromosome can result in large changes in a solution's fitness, which can lead to a sharp drop-off in the fitness landscape and cause the algorithm to converge prematurely. In binary genetic algorithms, the crossover and mutation operations can significantly affect the solution due to this Hamming cliff effect, especially when the bits that are to be changed are among the most significant bits in the binary string. To mitigate the Hamming cliff effect, Gray-coded GA uses a Gray-code encoding scheme for the chromosomes.

The reflected binary code, commonly referred to as *Gray code* after its inventor Frank Gray, is a unique binary numbering system characterized by adjacent numerical values differing by only one bit, as shown in table 8.1. In this numeral system, each value has a unique representation that is close to the representations of its neighboring values, which helps minimize the effect of crossover and mutation operations on the solution. This coding ensures a smooth transition between values and minimizes the risk of errors during conversions or when used in various applications, such as rotary encoders and digital-to-analog converters. Table 8.1 shows the decimal numbers 1 to 15 and their corresponding binary and Gray equivalents.

Table 8.1 Decimal, binary, and Gray coding

Decimal number	Binary code	Gray code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Exclusive OR (XOR) gates are used to convert 4-bit binary numbers to Gray codes, as illustrated in figure 8.1 for the ticket pricing example discussed in the previous chapter. These XOR gates result in 1 only if the inputs are different and 0 if the inputs are the same.

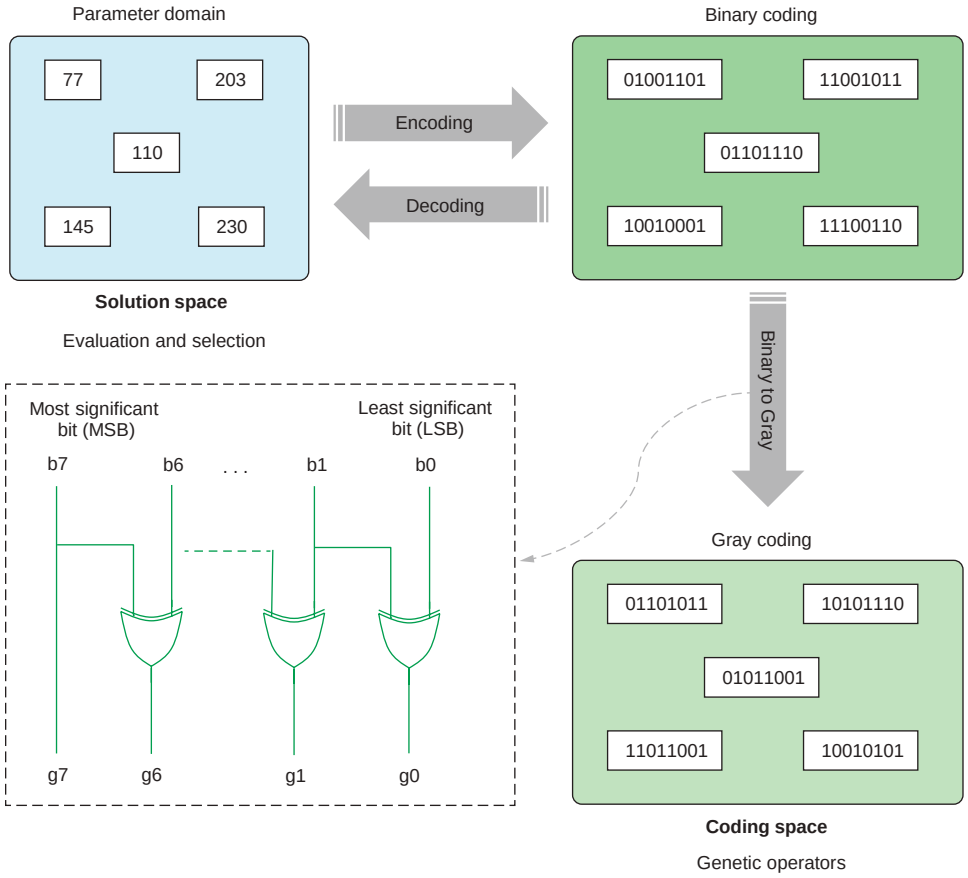


Figure 8.1 Gray coding and binary-to-Gray conversion

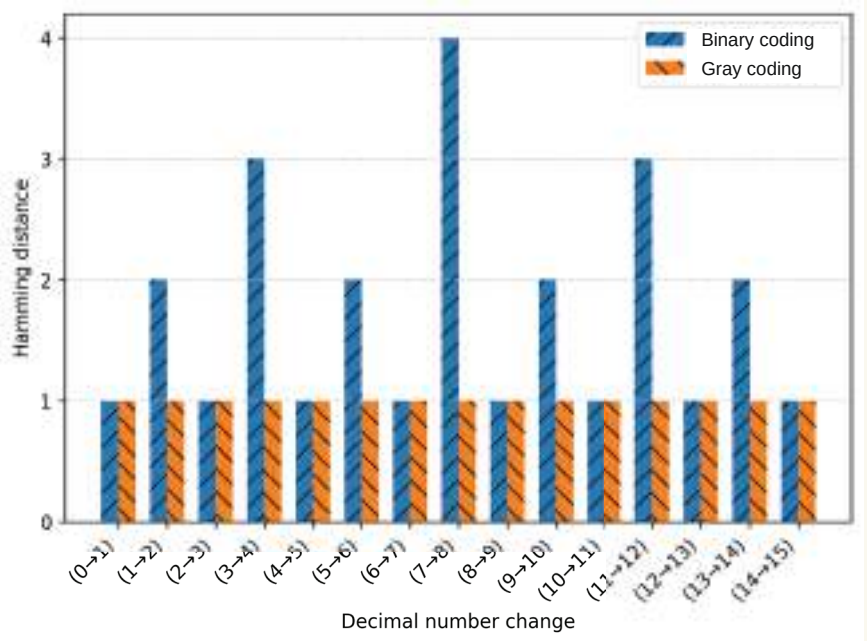
In Gray code, two successive values differ by only one bit. This property reduces the Hamming distance between adjacent numbers, leading to a smoother search space or smoother genotype-to-phenotype mapping.

Hamming cliff problem

One of the drawbacks of encoding variables as binary strings is the presence of Hamming cliffs. In binary-coded GAs, a small change in the encoded value (e.g., flipping a single bit) can lead to a significant change in the decoded value, especially if the flipped bit is located toward the most significant bit position. This abrupt change between two adjacent numbers in the search space is referred to as a Hamming cliff. This problem negatively affects binary-coded GAs by disrupting the search space's smoothness, causing poor convergence and leading to inefficient exploration and exploitation. To address the Hamming cliff problem, alternative representations like Gray code or real-valued encoding can be used, as they offer better locality and smoother search spaces, minimizing the disruptive effects of small changes on decoded values.

(continued)

For example, assume that we have a decision variable in the range [0, 15], as shown in the following figure. In binary-coded GA, we would use 4-bit binary representation to encode the candidate solutions. Let's assume we have two adjacent solutions in the search space: 7 and 8, or 0111 and 1000 in binary representation. The Hamming distance is the number of bit-wise differences, so the Hamming distance between 1000 and 0111 is 4. These two solutions (7 and 8) are neighbors in the search space, but when you look at their binary representations, you can see that they differ in all 4 bits. Flipping the most significant bit causes a significant change in the decoded value. In the case of Gray code, the Hamming distance between the Gray code representation 0100 (7 in decimal) and 1100 (8 in decimal) is only 1. This means that these Gray code representations for the two adjacent solutions differ by only 1 bit, providing a smoother search space and potentially improving the performance of the GA.



Hamming distances for decimal number change from 0 to 15 for binary and Gray coding

Gray code representations provide better locality, meaning that small changes in the encoded value result in small changes in the decoded value. This property can improve the convergence of the GA by reducing the likelihood of disruptive changes during crossover and mutation operations. However, it is worth noting that the performance improvements offered by Gray coding are problem-dependent, and this representation method is not commonly used compared to binary-coded GA.

8.2 Real-valued GA

Real-valued GA is a variation on the standard GA that uses real numbers for encoding chromosomes instead of binary or Gray code representations. Many optimization problems involve continuous variables or real-valued parameters, such as curve fitting, function optimization with real-valued inputs, proportional integral derivative (PID) controller parameter tuning, or optimizing the weights of a neural network. To handle these continuous problems, it's recommended that we use real-value GA directly for the following reasons:

- *Precision*—Real-valued GAs can achieve a higher level of precision in the search space than binary GAs. Binary encoding requires the discretization of the search space into a finite number of possible solutions, which can limit the accuracy of the search. Real-valued encoding, on the other hand, allows for a continuous search space, which can provide a more precise search.
- *Efficiency*—Real-valued GAs can require fewer bits to encode a solution compared to binary GAs. For example, assume that the decision variable to be represented has a lower bound (LB) of 0 and an upper bound (UB) of 10, and we need to represent the solution with a precision (P) of 0.0001. As explained in the previous chapter, the number of bits required to represent a range between LB and UB with a desired precision P is $number\_of\_bits = \text{ceil}(\log2((UB - LB)/P)) = \text{ceil}(\log2(\text{ceil}(10/0.0001))) = \text{ceil}(\log2(100000)) = 17$ bits. A real-valued encoding can use floating-point numbers with a smaller number of bits to represent a wider range of values than a binary encoding. This can result in a more efficient use of the available memory and computation resources.
- *Smoothness*—Real-valued GAs can maintain the continuity and smoothness of the search space, which can be important in some applications. In contrast, binary GAs can suffer from the Hamming cliff effect, as discussed in the previous section.
- *Adaptability*—Real-valued GAs can adapt more easily to changes in the search space or the fitness landscape. For example, if the fitness landscape changes abruptly, real-valued GAs can adjust the step size or mutation rate to explore the new landscape more effectively. Binary GAs, on the other hand, can require a more extensive redesign of the encoding or operator parameters to adapt to changes in the search space.

In the following subsections, we'll look at the crossover and mutation methods used in real-valued GA.

8.2.1 Crossover methods

Some popular crossover methods for real-valued GAs are single arithmetic crossover, simple arithmetic crossover, and whole arithmetic crossover.

SINGLE ARITHMETIC CROSSOVER

The *single arithmetic crossover method* involves picking a gene (k) at random and generating a random weight α, which lies in the range [0, 1]. Genes with indices i before and after the crossover point ($i < k$ or $i > k$) will inherit the genes from the corresponding parent chromosome. For genes at the crossover point ($i = k$), we create the offspring genes by taking a weighted average of the corresponding genes in the parent chromosomes:

$$Child_1Gene_i = \alpha \times Parent_1Gene_i + (1 - \alpha) \times Parent_2Gene_i$$
$$Child_2Gene_i = \alpha \times Parent_2Gene_i + (1 - \alpha) \times Parent_1Gene_i$$

Figure 8.2 illustrates the single arithmetic crossover in real-valued GA.

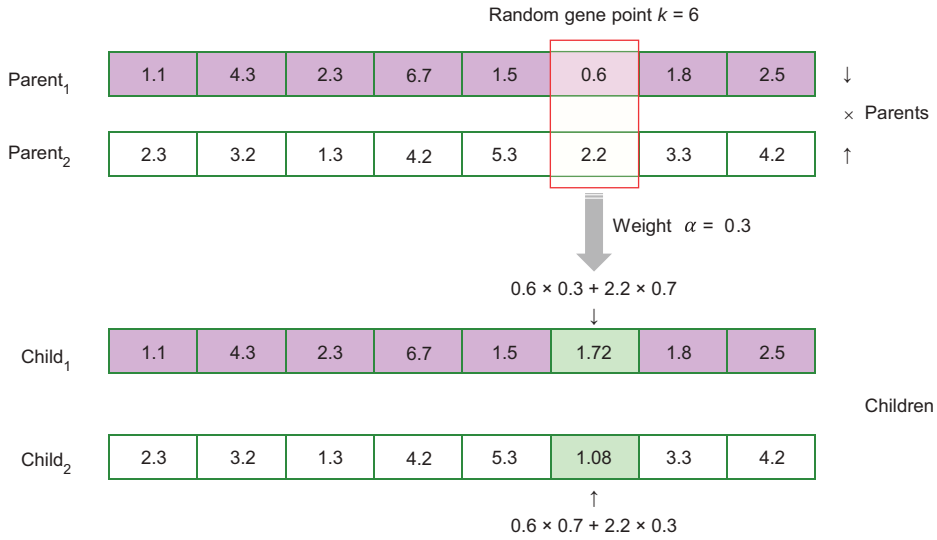


Figure 8.2 Single arithmetic crossover in real-valued GA

SIMPLE ARITHMETIC CROSSOVER

Simple arithmetic crossover is similar to single arithmetic crossover. Before a randomly picked crossover point ($i < k$), the genes are inherited from the corresponding parent chromosome. After the crossover point ($i >= k$), we create the offspring genes by taking a weighted average of the corresponding genes in the parent chromosomes. Figure 8.3 illustrates the simple arithmetic crossover in real-valued GA.

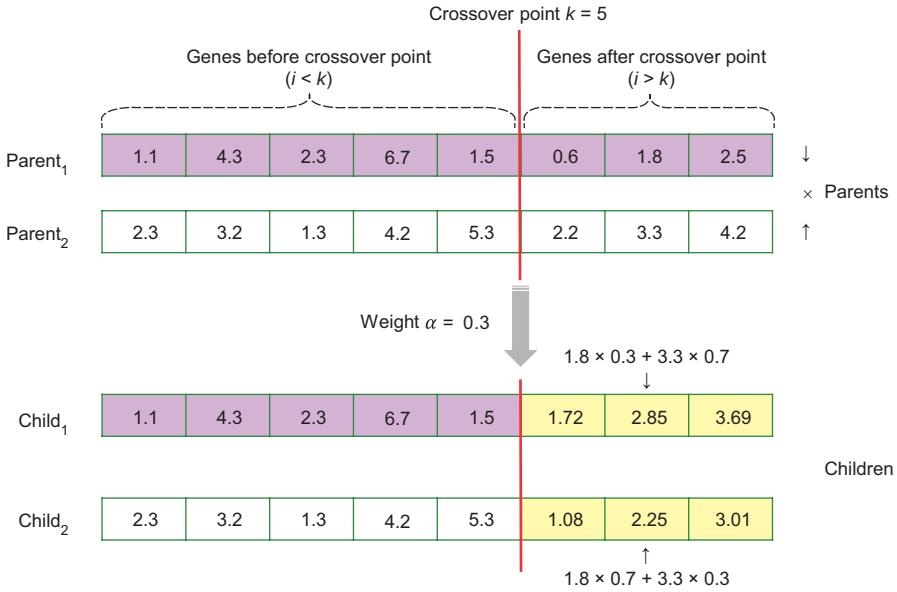


Figure 8.3 Simple arithmetic crossover in real-valued GA

WHOLE ARITHMETIC CROSSOVER

In the *whole arithmetic crossover* method, we take a weighted average of the entire parent chromosomes to create the offspring. Figure 8.4 illustrates this method in real-valued GA.

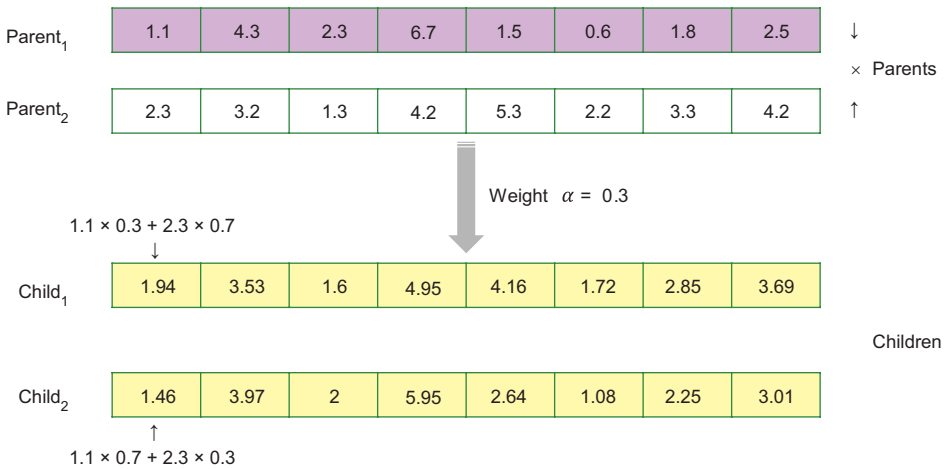


Figure 8.4 Whole arithmetic crossover in real-valued GA

SIMULATED BINARY CROSSOVER

Simulated binary crossover (SBX) [1] is another crossover method in real-valued GA. In SBX, real values can be represented by a binary notation, and then the point crossovers can be performed. SBX is designed to generate offspring close to the parent chromosomes by creating a probability distribution function, thus maintaining a balance between exploration and exploitation in the search space. SBX is implemented in pymoo.

8.2.2 Mutation methods

The simplest way to mutate a continuous variable is to introduce small, random perturbations to the genes of an individual to maintain diversity in the population and help the search process escape from local optima. There are several common mutation methods used in real-valued GAs:

- *Gaussian mutation*—Gaussian mutation adds a random value to the gene, where the random value is sampled from a Gaussian distribution with 0 mean and a specified standard deviation σ. The standard deviation controls the magnitude of the mutation (aka the *mutation step*).
- *Cauchy mutation*—Similar to Gaussian mutation, Cauchy mutation adds a random value to the gene, but the random value is sampled from a *Cauchy distribution* (aka a *Lorentz distribution* or *Cauchy–Lorentz distribution*) instead of a Gaussian distribution. The Cauchy distribution has heavier tails than the Gaussian distribution, leading to a higher probability of larger mutations.
- *Boundary mutation*—In boundary mutation, the mutated gene is randomly drawn from the uniform distribution within the variable's range, defined by the lower bound (*LB*) and upper bound (*UB*). This method is analogous to the bit-flipping mutation in a binary-coded GA, and it helps explore the boundaries of the search space. It may be useful when optimal solutions are located near the variable limits.
- *Polynomial mutation*—Polynomial mutation is a method that generates offspring close to the parent by creating a probability distribution function [2]. A distribution index (η) controls the shape of the probability distribution function, with higher values resulting in offspring closer to their parents (exploitation) and lower values leading to offspring more spread out in the search space (exploration).

To illustrate these genetic operators, let's consider a curve-fitting example. Assume we have the data points shown in table 8.2 and that we want to fit a third-order polynomial to these data points using real-valued GA.

Table 8.2 Curve-fitting problem data

x	0	1.25	2.5	3.75	5
y	1	5.22	23.5	79.28	196

The third-order polynomial takes the form $y = ax^3 + bx^2 + cx + d$. A real-valued GA can be used to find the four coefficients of the polynomial: a, b, c, and d. This problem is treated as a minimization problem where the objective is to minimize the mean squared error (MSE) that measures how close a fitted polynomial is to given data points. MSE is calculated using the following formula:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i')^2 \tag{8.1}$$

where n is the number of data points, y is the y-coordinate value of each data point, and y' is the desired value that sits on the line we created.

In real-valued GA, a candidate solution is represented by a vector of parameters a, b, c, and d that can be represented by real values. Let's start with the following initial random solution: $Parent_1 = [1\ 2\ 3\ 4]$. We calculate its fitness by substituting these values in the function $(y = x^3 + 2x^2 + 3x + 4)$, calculating y' for each corresponding x and calculating the MSE as in table 8.3.

Table 8.3 MSE calculation for parent 1

x	0	1.25	2.5	3.75	5
y	1	5.22	23.5	79.28	196
y′	4	12.83	39.63	96.11	194
Square of error	9	57.88	260	283.23	4
MSE	122.83				

Let's generate another random solution: $Parent_2 = [2\ 2\ 2\ 2]$, which gives the formula $2x^3 + 2x^2 + 2x + 2$ and the MSE in table 8.4.

Table 8.4 MSE calculation for parent 2

x	0	1.25	2.5	3.75	5
y	1	5.22	23.5	79.28	196
y′	2	11.53	50.75	143.09	312
Square of error	1	39.83	742.56	4,072.2	13,456
MSE	3,662.32				

Applying whole arithmetic crossover on the two parents $P_1 = [1\ 2\ 3\ 4]$ and $P_2 = [2\ 2\ 2\ 2]$ with weight $\alpha = 0.2$ results in the following offspring:

$$\begin{aligned} Child_1 &= \alpha P_1 + (1-\alpha)P_2 \\ &= [0.2\ 0.4\ 0.6\ 0.8] + [1.6\ 1.6\ 1.6\ 1.6] \\ &= [1.8\ 2\ 2.2\ 2.4], MSE = 2434.07 \end{aligned}$$

$$\begin{aligned} Child_2 &= \alpha P_2 + (1-\alpha)P_1 \\ &= [0.4\ 0.4\ 0.4\ 0.4] + [0.8\ 1.6\ 2.4\ 3.2] \\ &= [1.2\ 2\ 2.8\ 3.6], MSE = 310.37 \end{aligned}$$

Let's assume $Child_1$ is subject to Gaussian mutation. This mutation process results in another child as follows: $Child_3 = Child_1 + N(0, \sigma)$, where $N(0, \sigma)$ is a random number from a normal distribution with a mean of 0 and a standard deviation of σ. Assuming that $\sigma = 1.2$, a random value of 0.43 is generated by `numpy.random.normal(0, 1.2)`, so $Child_3 = [1.8\ 2\ 2.2\ 2.4] + 0.43 = [2.23\ 2.43\ 2.63\ 2.83]$.

Listing 8.1 shows how to perform this curve-fitting using real-valued GA implemented in pymoo. We'll start by generating a dataset driven by a third-order polynomial, to be used later as a ground truth. Feel free to replace this synthetically generated data with any experimental data you may have.

Listing 8.1 Curve fitting using real-valued GA

```
import numpy as np

def third_order_polynomial(x, a, b, c, d):
    return a * x**3 + b * x**2 + c * x + d

a, b, c, d = 2, -3, 4, 1

x = np.linspace(0, 5, 5)

y = third_order_polynomial(x, a, b, c, d)

data_samples = np.column_stack((x, y))
```

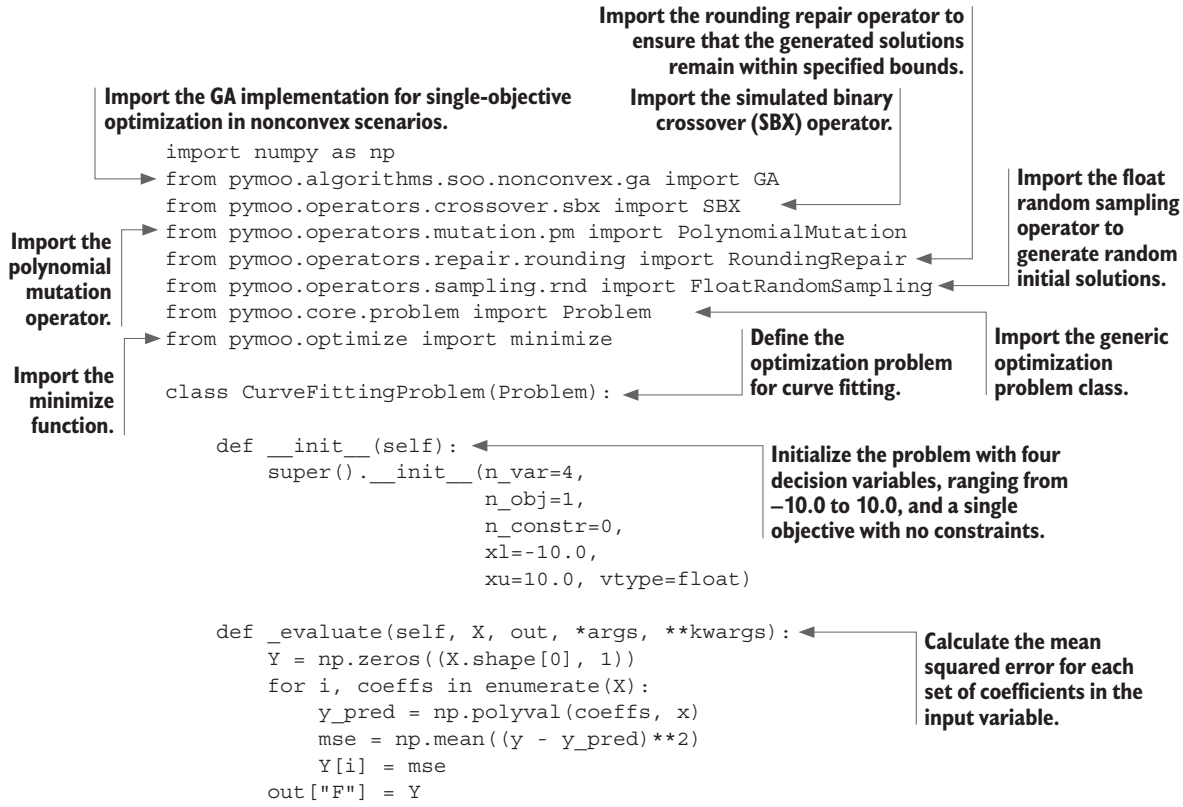
Define coefficients for the third-order polynomial.

Generate five values as in the hand-iteration example.

Calculate y values using the third-order polynomial function.

Combine x and y values into an array of data samples.

As a continuation of listing 8.1, we can define a problem for curve fitting by subclassing pymoo's `Problem` class, ensuring we pass the parameters to the superclass and provide an `_evaluate` function. The `CurveFittingProblem` class has an initializer method that sets the number of decision variables to 4, the number of objectives to 1, the number of constraints to 0, the lower bound of the decision variables to –10.0, and the upper bound of the decision variables to 10.0. The `vtype` parameter specifies the data type of the decision variables, which is set to `float`. This initializer method creates an instance of the problem to be solved using the genetic algorithm. The `_evaluate` method takes as input a set of candidate solutions (`X`) and an output dictionary (`out`) and returns the fitness of each candidate solution in the `F` field of the `out` dictionary:




```
import numpy as np
from pymoo.algorithms.soo.nonconvex.ga import GA
from pymoo.operators.crossover.sbx import SBX
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

class CurveFittingProblem(Problem):

    def __init__(self):
        super().__init__(n_var=4,
                         n_obj=1,
                         n_constr=0,
                         xl=-10.0,
                         xu=10.0, vtype=float)

    def _evaluate(self, X, out, *args, **kwargs):
        Y = np.zeros((X.shape[0], 1))
        for i, coeffs in enumerate(X):
            y_pred = np.polyval(coeffs, x)
            mse = np.mean((y - y_pred)**2)
            Y[i] = mse
        out["F"] = Y
```

Now we can instantiate the `CurveFittingProblem` class to create an instance of the problem to be solved. We can then define the genetic algorithm to be used for the optimization. The `GA` class is used to define the algorithm, and the `pop_size` parameter sets the population size to 50. The `sampling` parameter uses the `FloatRandomSampling` operator to generate the initial population of candidate solutions randomly. The `crossover` parameter uses the `SBX` operator with a crossover probability of 0.8. The `mutation` parameter uses the `PolynomialMutation` operator with a mutation probability of 0.3 and a rounding repair operator to ensure that the decision variables remain within the specified bounds. The `eliminate_duplicates` parameter is set to `True` to remove duplicate candidate solutions from the population.

Next, we can run the genetic algorithm to solve the curve fitting problem using the `minimize` function. This function takes three arguments: the instance of the problem to be solved (`problem`), the instance of the algorithm to be used (`algorithm`), and a tuple specifying the stopping criterion for the algorithm `('n_gen', 100)`, which specifies that the algorithm should run for 100 generations. The `seed` parameter is set to 1 to ensure the reproducibility of the results. The `verbose` parameter is set to `True` to display the progress of the algorithm during the optimization:

```
problem = CurveFittingProblem()

algorithm = GA(
    pop_size=50,
    sampling=FloatRandomSampling(),
    crossover= SBX(prob=0.8),
    mutation = PolynomialMutation(prob=0.3, repair=RoundingRepair()),
    eliminate_duplicates=True
)

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

Initialize an instance of the CurveFittingProblem class.

Create a GA solver.

Perform the optimization for 100 generations.

You can print the four coefficients obtained by GA as follows:

```
best_coeffs = res.X
print("Coefficients of the best-fit third-order polynomial:")
print("a =", best_coeffs[0])
print("b =", best_coeffs[1])
print("c =", best_coeffs[2])
print("d =", best_coeffs[3])
```

This results in the following output:

```
Coefficients of the best-fit third-order polynomial:
a = 2, b = -3, c = 4, d = 1
```

As you can see, the estimated values of the four coefficients are same as the coefficients of the ground truth polynomial (a, b, c, d = 2, –3, 4, 1). You can experiment with the code by changing the polynomial coefficients, using your own data, and using different crossover and mutation methods.

Next, we'll look at permutation-based GA.

8.3 Permutation-based GA

Permutation-based GAs are designed to solve optimization problems where the solutions are permutations of a set of elements. Examples of such problems include the traveling salesman problem (TSP), vehicle routing problem, sports tournament scheduling, and job scheduling problem. In these problems, the solutions are represented as optimal orders or permutations of a set of elements or events.

There are typically two main types of problems where the goal is to determine the optimal order of events:

- *Resource or time-constrained problems*—In these problems, the events rely on limited resources or time, making the order of events crucial for optimal solutions. One example of this type of problem is ride-sharing scheduling, where the goal is to efficiently allocate resources like vehicles and drivers to serve the maximum number of passengers in the shortest possible time.

- *Adjacency-based problems*—In these problems, the proximity or adjacency of elements plays a significant role in finding the best solution. An example of such a problem is the traveling salesman problem (TSP), where the aim is to visit a set of cities while minimizing the total travel distance, taking into account the distances between adjacent cities in the tour.

These problems are often formulated as permutation problems. In a permutation representation, if there are n variables, the solution is a list of n distinct integers, each occurring exactly once. This representation ensures that the order or adjacency of the elements in the solution is explicitly encoded, which is essential for finding the optimal sequence of events in these types of problems. For example, let's consider the following TSP for 8 cities. A candidate solution for this TSP is represented by a permutation such as [1, 2, 3, 4, 5, 6, 7, 8]. In permutation-based GA, specialized crossover and mutation operators are employed to preserve the constraints of the permutation problem, such as maintaining a valid sequence of cities in the TSP, where each city appears only once.

The following subsections describe commonly used crossover and mutation methods in permutation-based GAs. The choice of crossover and mutation methods in GAs depends on the problem being solved, the type of solutions being sought, and the objectives of the optimization problem. By carefully selecting and designing these operators, GAs can effectively explore and exploit the search space to find high-quality solutions.

8.3.1 Crossover methods

Several crossover methods are commonly used in permutation-based GAs, such as partially mapped crossover (PMX), edge crossover (EC), order 1 crossover (OX1), and cycle crossover (CX).

PARTIALLY MAPPED CROSSOVER

The *partially mapped crossover* (PMX) method creates offspring by combining the genetic information from two parent chromosomes while preserving the feasibility of the resulting offspring with the procedure shown in algorithm 8.1.

Algorithm 8.1 Partially mapped crossover (PMX)

```
Input: two parents P1 and P2
Output: two children C1 and C2
1. Initialize: Choose two random crossover points and copy the segment
between these two points from parent P1 to child C1 and from the second
parent P2 to the second child C2.
2. For each element in the copied segment of C1:
3.   Find the corresponding element in P2's segment.
4.   If the corresponding element is not already in C1:
5.      Replace the element in C1 at the same position as in P2 with the
        corresponding element from P2.
```

```
6. Fill the remaining positions in the offspring with the elements from the
other parent, ensuring that no duplicates are introduced.
7. Repeat steps 2-6 for the second offspring, using P1's segment as the
reference.
8. Return C1 and C2.
```

Figure 8.5 illustrates these steps for an 8-city TSP. In step 1, two random crossover points are chosen, and the cities between these two points are copied from parent P1 to child C1 and from the second parent P2 to the second child C2. Then we follow steps 2 to 5 for the cities that were not included in step 1. For the first city in C1, which is 3, we need to find the corresponding city in the P2 segment, which is 7. City 7 is not already included in C1, so we need to place city 7 in the place where city 3 appears in P2, which is the last position on the right, as shown by the solid black arrow in figure 8.5.

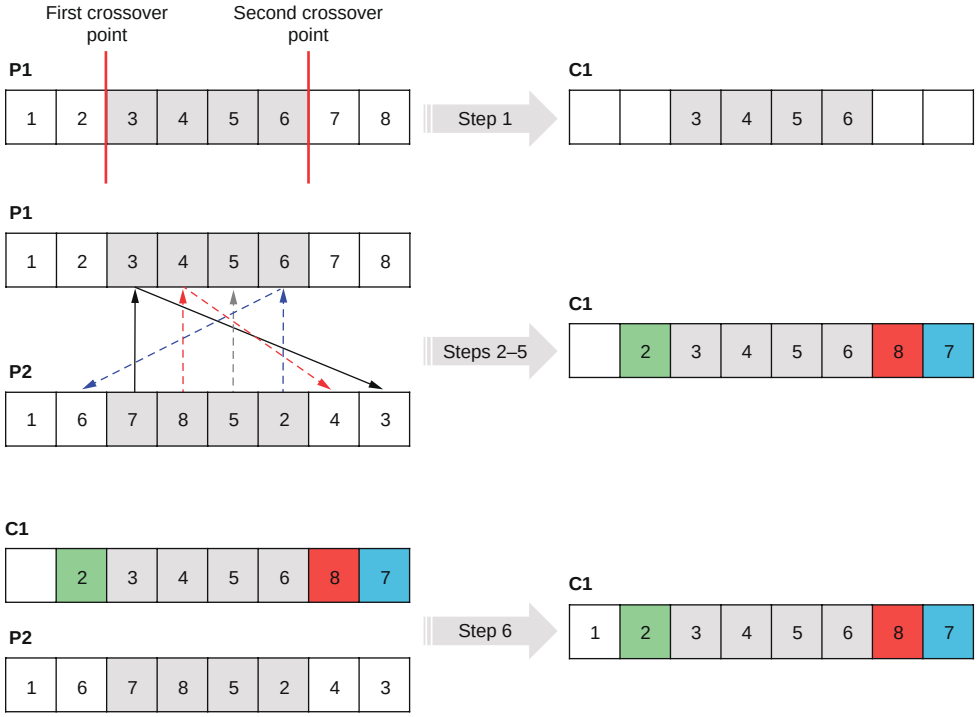


Figure 8.5 Partially mapped crossover (PMX)

The following listing shows code that performs a partially mapped crossover on two parents to generate two offspring.

Listing 8.2 Partially mapped crossover (PMX)

```
import random

def partially_mapped_crossover(parent1, parent2):

    n = len(parent1)

    point1, point2 = sorted(random.sample(range(n), 2))

    child1 = [None] * n
    child2 = [None] * n
    child1[point1:point2+1] = parent1[point1:point2+1]
    child2[point1:point2+1] = parent2[point1:point2+1]

    for i in range(n):
        if child1[i] is None:
            value = parent2[i]
            while value in child1:
                value = parent2[parent1.index(value)]
            child1[i] = value

        if child2[i] is None:
            value = parent1[i]
            while value in child2:
                value = parent1[parent2.index(value)]
            child2[i] = value

    return child1, child2
```

Select two random crossover points.

Copy the segment between crossover points from a parent to a child.

Map the remaining elements from the other parent.

Return the generated offspring.

Running this code will produce output like that shown in figure 8.6, depending on the generated random sample.

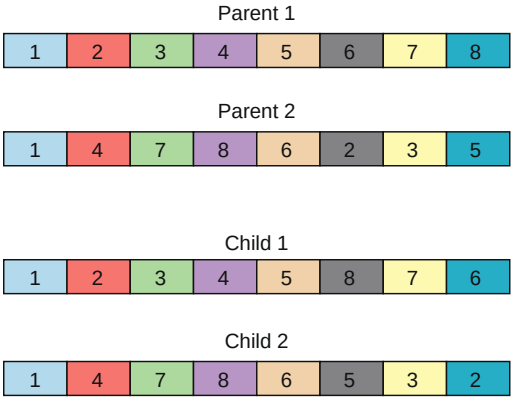


Figure 8.6 PMX results

The complete version of listing 8.2 is available in the book's GitHub repository.

EDGE CROSSOVER

The *edge crossover* (EC) method preserves the connectivity and adjacency information between elements from the parent chromosomes. In order to achieve this, an *edge table* (or *adjacency list*) is constructed. For example, in the 8-cities TSP, the edge table for two parents P1 = [1, 2, 3, 4, 5, 6, 7, 8] and P2 = [1, 6, 7, 8, 5, 2, 4, 3] is created by counting the adjacent elements in both parents, as in table 8.5. The "+" signs in the table denote a common edge between the two parents.

Table 8.5 An edge table (or adjacency list)

City	1	2	3	4	5	6	7	8
Edges	2,8,6,3	1,3,5,4	2,4+,1	3+,5,2	4,6,8,2	5,7+,1	6+,8+	7+,1,5

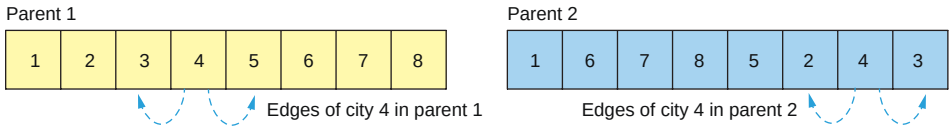
Algorithm 8.2 shows the steps involved in edge crossover.

Algorithm 8.2 Edge crossover (EC)

```
Input: two parent P1 and P2
Output: offspring C
1. Construct an edge table.
2. Start by selecting an arbitrary element from one of the parents as the
starting point for the offspring.
3. For the current element in the offspring, compare its edges.
4. If an edge is common in both parents, choose that as the next element in
the offspring.
5. If there is no common edge or the common edge is already in the offspring,
choose the next element from the parent with the shortest edge list.
6. Repeat until the offspring is complete.
7. Return C
```

Figure 8.7 illustrates these steps for the 8-cities TSP. This figure illustrates how to construct the edge table or adjacency list of each city. The process of counting the edges is shown in this figure. For example, cities 3 and 5 are adjacent cities or edges to city 4 in the first parent. In the second parent, cities 2 and 3 are edges for city 4. This means that city 3 is a common edge.

Creating a child starts by selecting city 1 randomly or as a home city. In the second row of the table, we list the adjacent cities of city 1, which are 2, 8, 6 and 3. Note that cities loop around, meaning city 1 is adjacent to city 8 in the first parent, and city 1 is adjacent to city 3 in the second parent. Discarding the already visited city 1, these cities have the following adjacent cities {3,5,4} for city 2, {7,5} for city 8, {5,7} for city 6, and {2,4} for city 3. We discard city 2, as it has three adjacent cities, and we select city 3 arbitrarily from 8, 6, and 3, as they have the same number of edges. We keep adding cities to the child following algorithm 8.2 until all the cities are added.



City	1	2	3	4	5	6	7	8
Edges	2,8,6,3	1,3,5,4	2,4+,1	3+,5,2	4,6,8,2	5,7+,1	6+,8+	7+,1,5

Choices	Element selected	Reason	Partial results - offspring
All	1	Random choice/home city	[1]
2,8,6,3	3	Arbitrarily chosen from 8, 6, and 3 as they have shortest lists	[1 3]
2,4+,1	4	Common edge	[1 3 4]
3+,5,2	2	Shortest list	[1 3 4 2]
1,3,5,4	5	Only item in the list	[1 3 4 2 5]
4,6,8,2	6	Arbitrarily chosen (both 6 and 8 have the same items)	[1 3 4 2 5 6]
5,7+,1	7	Only item and common edge	[1 3 4 2 5 6 7]
6+,8+	8	Last element	[1 3 4 2 5 6 7 8]

Figure 8.7 Edge crossover

The complete version of listing 8.2 is available in the book's GitHub repository. It shows the Python implementation of edge crossover with a TSP example.

ORDER 1 CROSSOVER

Order 1 crossover (OX1) creates offspring by combining the genetic information from two parent chromosomes while preserving the relative order of the elements in the resulting solutions. Algorithm 8.3 shows the steps involved in order 1 crossover.

Algorithm 8.3 Order 1 crossover (OX1)

```
Input: two parents P1 and P2
Output: offspring C1 and C2
1. Choose two random crossover points within the chromosomes and copy the
segment between the crossover points from P1 to C1 and from P2 to C2.
2. Starting from the second crossover point, go through the remaining
elements in P2.
3. If an element is not already present in C1, append it to the end in the
same order as it appears in P2.
4. Wrap around P2 and continue appending the elements until C1 is complete.
Repeat steps 2-4 for C2, using P1 as the reference.
5. Return C1 and C2
```

Figure 8.8 illustrates these steps for the 8-cities TSP. Starting from the second crossover point, cities 4 and 3 cannot be added, as they are already included in C1. The next element in P2 is city 1, so it is added to C1 after the second crossover point, followed by city 7, as city 6 is already included. City 8 is added next, followed by city 2, as city 5 is already included in C1.

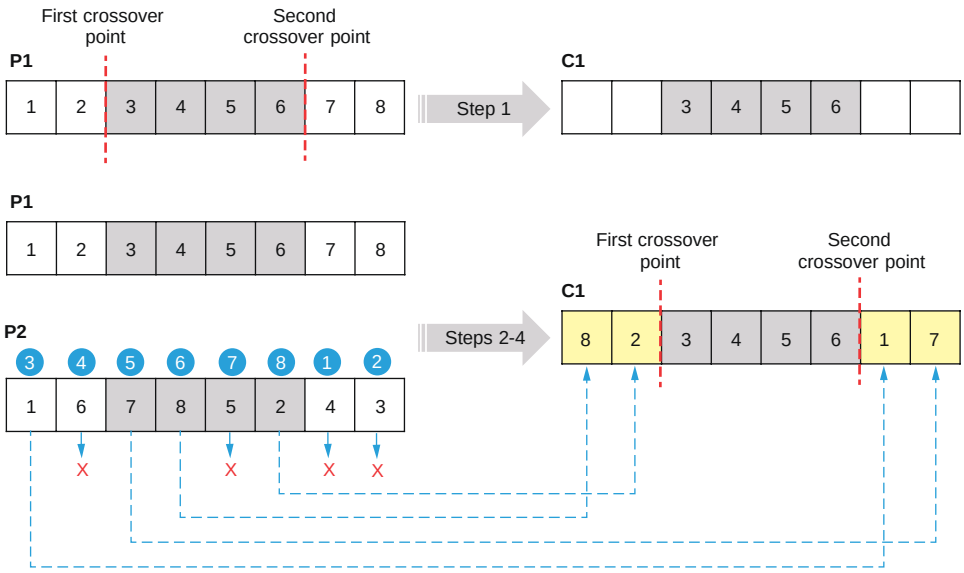


Figure 8.8 Order 1 crossover (OX1)—The numbers in circles show the sequence in which elements are added from parent 2 to child 1.

The complete version of listing 8.2 is available in the book's GitHub repository, and it shows the Python implementation of OX1 with the TSP example.

CYCLE CROSSOVER

Cycle crossover (CX) operates by dividing the elements into cycles, where a *cycle* is a subset of elements that consistently appear together in pairs when the two parent chromosomes are aligned. Given two parents, a cycle is formed by selecting an element from the first parent, finding its corresponding position in the second parent, and then repeating this process with the element at that position until returning to the starting element. The CX operator effectively combines the genetic information from both parents while preserving the order and adjacency relationships among the elements in the resulting offspring and maintaining the feasibility and diversity of the offspring solutions. Algorithm 8.4 shows the steps of this crossover method.

Algorithm 8.4 Cycle crossover (CX)

```
Input: two parents P1 and P2
Output: offspring C1 and C2
```

```
1. Identify cycles between the two parents. A cycle of elements from a parent
P1 is created following these steps:
    a) Begin with the starting element of P1.
    b) Look at the element at the corresponding position in P2.
    c) Move to the position with the same element in P1.
    d) Include this element in the cycle.
    e) Iterate through steps b to d until you reach the starting element of
P1.
2. Create offspring by placing the elements of the identified cycles,
preserving their positions from the corresponding parents.
3. Fill in the remaining positions in C1 with elements from P2 and the
remaining positions of C2 with elements from P1 that were not included in the
identified cycles. Maintain the order of the elements as they appear in the
parents.
4. Return C1 and C2.
```

Figure 8.9 illustrates these steps for the 10-cities TSP.

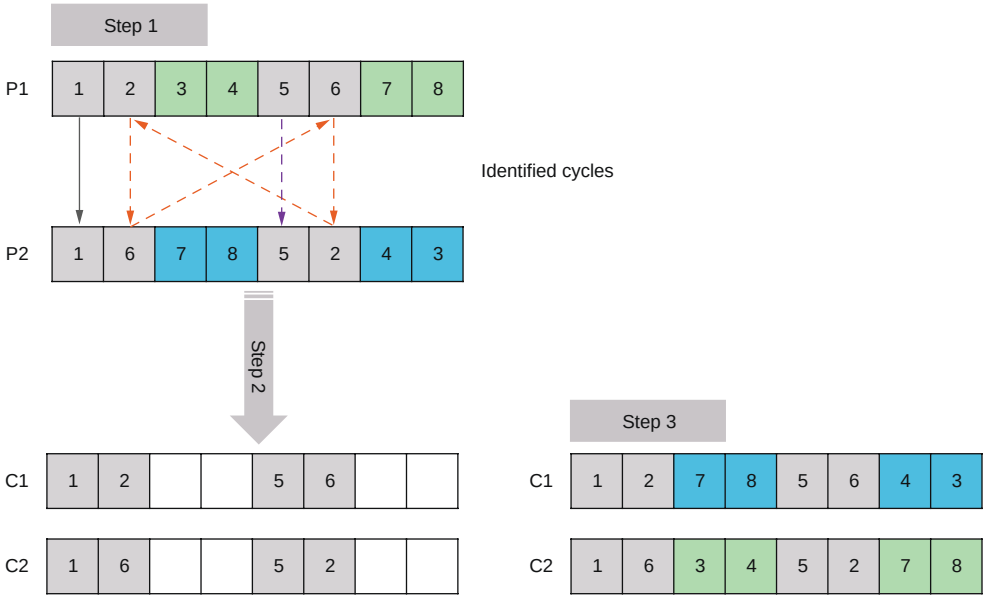


Figure 8.9 Cycle crossover (CX)

A Python implementation of CX with a TSP example is included in the complete version of listing 8.2, available in the book's GitHub repository. It's important to note that the performance of crossover operators is often problem-dependent and may also be influenced by the specific parameter settings of the genetic algorithm, such as population size, mutation rate, and selection pressure. Therefore, it is recommended that you experiment with different crossover operators and fine-tune the genetic algorithm's parameters to suit the problem being addressed.

8.3.2 Mutation methods

Insert, swap, inversion, and scramble are commonly used mutation methods in permutation-based GA. These methods are designed to introduce small perturbations to the solution while still preserving its feasibility:

- *Insert mutation*—Pick two gene values at random, and move the second to follow the first, shifting the rest along to accommodate them. This method primarily maintains the order and adjacency information of the genes.
- *Swap mutation*—Pick two genes at random, and swap their positions. This method mainly retains adjacency information while causing some disruption to the original order.
- *Inversion mutation*—Randomly select two genes, and invert the substring between them. This method largely maintains adjacency information but is disruptive to the order information.
- *Scramble mutation*—Randomly select two gene values, and rearrange the genes between the chosen positions non-contiguously, applying a random order.

Figure 8.10 illustrates these methods on the first parent as a selected individual in the 8-cities TSP.

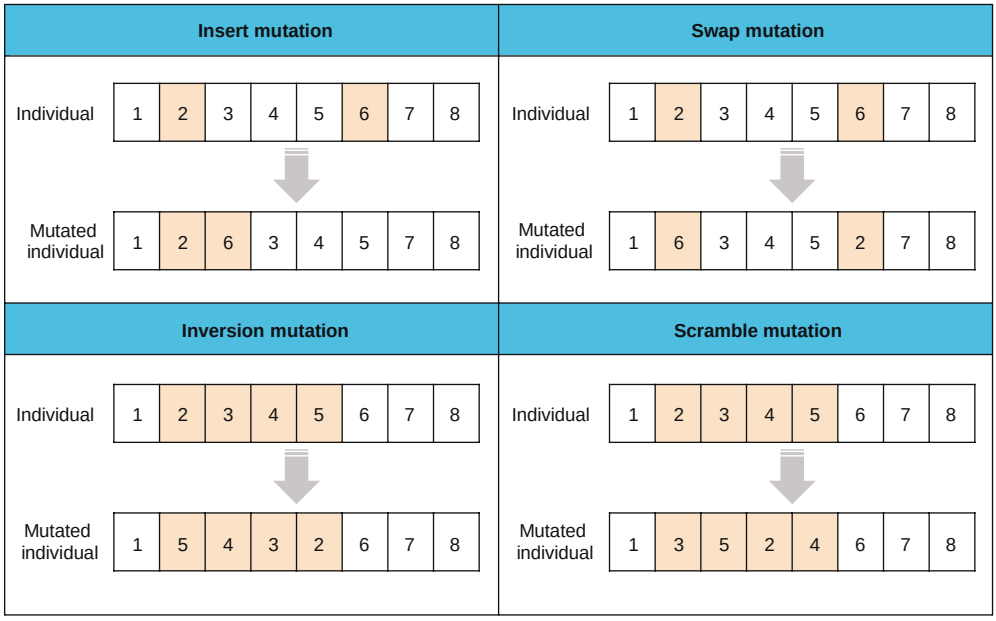


Figure 8.10 Mutation methods in permutation-based GA

As a continuation of listing 8.2, the following code snippet shows how you can implement inversion mutation in Python:

```
def inversion_mutation(individual, mutation_rate):
    n = len(individual)
    mutated_individual = individual.copy()

    if random.random() < mutation_rate:
        i, j = sorted(random.sample(range(n), 2))
        mutated_individual[i:j+1] = reversed(mutated_individual[i:j+1])

    return mutated_individual
```

Running this code will produce output like that in figure 8.11.

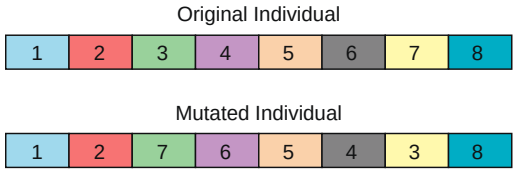


Figure 8.11 Inversion mutation result

The complete version of listing 8.2, available in the book's GitHub repository, includes implementations of different crossover and mutation methods commonly used in permutation-based genetic algorithms.

8.4 Multi-objective optimization

As mentioned earlier in section 1.3.2, optimization problems with multiple objective functions are known as multi-objective optimization problems (MOPs). These problems can be handled using a preference-based multi-objective optimization procedure or by using a Pareto optimization approach. In the former approach, the multiple objectives are combined into a single or overall objective function by using a relative preference vector or a weighting scheme to scalarize the multiple objectives. However, finding this preference vector or weight is subjective and sometimes is not straightforward.

Pareto optimization, named after Italian economist and sociologist Vilfredo Pareto (1848–1923), relies on finding multiple trade-off optimal solutions and choosing one using higher-level information. This procedure tries to find the best trade-off by reducing the number of alternatives to an optimal set of nondominated solutions known as the Pareto front (or Pareto frontier), which can be used to make strategic decisions in multi-objective space. A solution is Pareto optimal if there is no other solution that improves one objective without worsening another objective, in the case of conflicting objective functions. Thus, the optimal solution for MOPs is not a single solution, as for mono-objective or single optimization problems (SOPs), but a set of solutions defined as *Pareto optimal solutions*. These Pareto optimal solutions are also known as acceptable, efficient, nondominated, or non-inferior solutions. *Nondominated solutions* in Pareto

optimization represent the best compromises that are not outperformed by any other solution across multiple conflicting objectives.

In chapter 1, we looked at an electric vehicles example: acceleration time and driving range are conflicting objective functions, as we need to minimize the acceleration time and maximize the driving range of the vehicle. There is no universal best vehicle that achieves both, as shown in figure 8.12, which is based on real data retrieved from the *Inside EVs* website (https://insideevs.com/). For example, the Lucid Air Dream Edition has the highest driving range but not the lowest acceleration time. The dotted line shows the Pareto front—the vehicles that achieve the best trade-off between the acceleration time and the driving range.

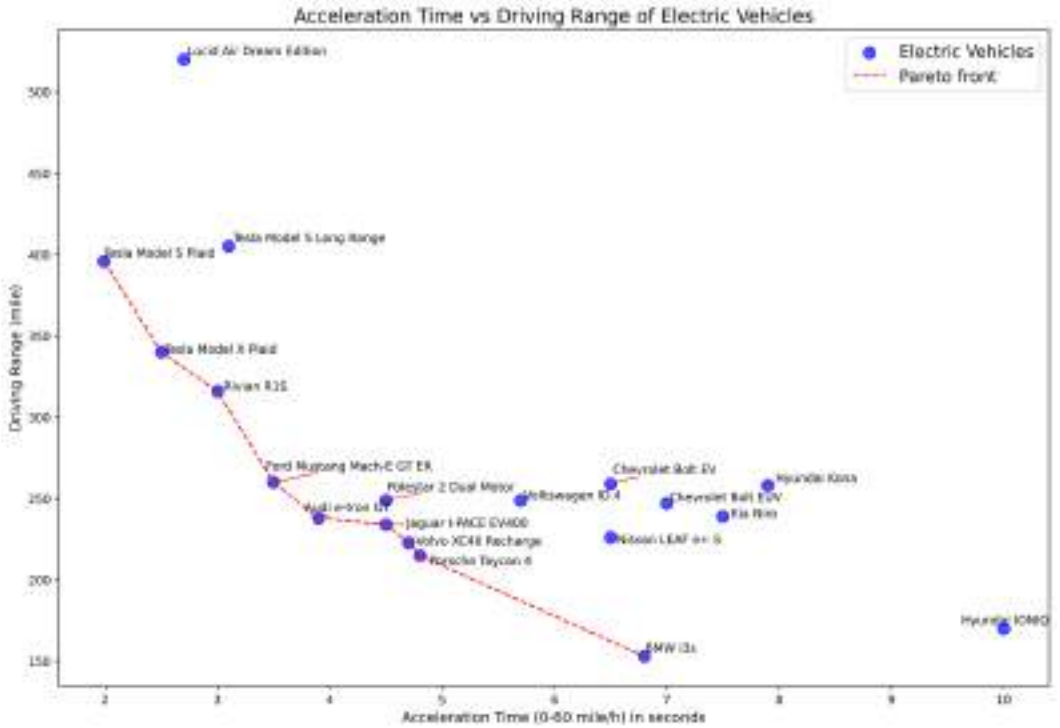


Figure 8.12 Acceleration time vs. driving range of 19 electric vehicles, as per September 2021

Multi-objective optimization algorithms

There are several algorithms for solving multi-objective optimization problems. The non-dominated sorting genetic algorithm (NSGA-II) is one of the most commonly used. Other algorithms include, but are not limited to, the strength Pareto evolutionary algorithm 2 (SPEA2), the Pareto-archived evolution strategy (PAES), the niched-Pareto genetic algorithm (NPGA), multi-objective selection based on dominated hypervolume (SMS-EMOA), and multi-objective evolutionary algorithm based on decomposition (MOEA/D).

These algorithms have their own strengths and weaknesses, and your choice of algorithm will depend on the specific problem being solved and your preferences. NSGA-II has several advantages, such as diversity maintenance, non-dominated sorting, and fast convergence. For more details about multi-objective optimization, see Deb's "Multi-objective optimization using evolutionary algorithms" [3] and Zitzler's "Evolutionary algorithms for multiobjective optimization" [4].

Let's solve a MOP using NSGA-II in an example. Assume that a manufacturer produces two products, P1 and P2, involving two different machines, M1 and M2. Each machine can only produce one product at a time, and each product has a different production time and cost on each machine:

- P1 requires 2 hours on M1 and 3 hours on M2, with production costs of $100 and $150 respectively.
- P2 requires 4 hours on M1 and 1 hour on M2, with production costs of $200 and $50 respectively.

In each shift, the two machines, M1 and M2, have the capacity of producing 100 units of P1 and 500 units of P2. The manufacturer wants to produce at least 80 units of P1 and 300 units of P2 while minimizing production costs and minimizing the difference in production times between the two machines.

We'll let x_1 and x_2 be the number of units of P1 produced on M1 and M2, respectively, and y_1 and y_2 be the number of units of P2 produced on M1 and M2, respectively. The problem can be formulated as follows:

$$\text{Minimize} f_1(x_1, x_2, y_1, y_2) = 100x_1 + 150x_2 + 200y_1 + 50y_2$$
$$\text{Minimize} f_2(x_1, x_2, y_1, y_2) = |(2x_1 + 4y_1) - (3x_2 + y_2)|$$

Subject to:

$$x_1 + x_2 \geq 80$$
$$y_1 + y_2 \geq 300$$
$$x_1, x_2, y_1, y_2 \geq 0$$

The first objective function (f_1) represents the total production costs, and the second objective function (f_2) represents the difference in production times between the two machines. Listing 8.3 shows the code for finding the optimal number of units to be produced in a shift using NSGA-II.

We'll start by inheriting from `ElementwiseProblem`, which allows us to define the optimization problem in an element-wise manner. `n_var` specifies the number of variables (4 in this case), `n_obj` defines the number of objectives (2), and `n_ieq_constr`

indicates the number of inequality constraints (2). The `xl` and `xu` parameters define the lower and upper bounds for each variable respectively. The `_evaluate` method takes an input `x` (a solution candidate) and computes the objective values `f1` and `f2`, as well as the inequality constraints `g1` and `g2`. The third constraint is boundary constraint represented by the lower and upper bounds of the decision variables.

Listing 8.3 Solving a manufacturing problem using NSGA-II

```
import numpy as np
import matplotlib.pyplot as plt
from pymoo.core.problem import ElementwiseProblem

class ManufacturingProblem(ElementwiseProblem):

    def __init__(self):
        super().__init__(n_var=4,
                         n_obj=2,
                         n_ieq_constr=2,
                         xl=np.array([0,0,0,0]),
                         xu=np.array([100,100,500,500]))

    def _evaluate(self, x, out, *args, **kwargs):
        f1 = 100*x[0] + 150*x[1] + 200*x[2] + 50*x[3]
        f2 = np.abs((2*x[0] + 4*x[2]) - (3*x[1] + x[3]))

        g1 = -x[0] - x[1] + 80
        g2 = -x[2] - x[3] + 300

        out["F"] = [f1, f2]
        out["G"] = [g1, g2]

problem = ManufacturingProblem()
```

Import an instance of the problem class.

Define the number of variables, objective functions, constraints, and lower and upper bounds.

Difference in production times between the two machines as a second-objective function

Total production costs as a first-objective function

Define the constraints.

We can now set up an instance of the NSGA-II algorithm as the solver:

```
from pymoo.algorithms.moo.nsga2 import NSGA2
from pymoo.operators.crossover.sbx import SBX
from pymoo.operators.mutation.pm import PM
from pymoo.operators.sampling.rnd import FloatRandomSampling

algorithm = NSGA2(
    pop_size=40,
    n_offsprings=10,
    sampling=FloatRandomSampling(),
    crossover=SBX(prob=0.9, eta=15),
    mutation=PM(eta=20),
    eliminate_duplicates=True
)
```

Import simulated binary crossover (SBX) as a crossover operator.

Import the NSGA-II class.

Import polynomial mutation (PM) as a mutation operator.

Import the FloatRandomSampling method to generate random floating-point values for each variable within a specified range.

Set up an instance of NSGA-II.

The solver has a population size (`pop_size`) of 40 individuals, generates 10 offspring using `FloatRandomSampling`, employs SBX crossover with a probability of 0.9, and fine-tunes the exponential distribution with an `eta` parameter of 15. PM mutation is used with an `eta` parameter of 20. This `eta` parameter controls the spread of the mutation distribution. `eliminate_duplicates` is set to `True` so that duplicate candidate solutions will be removed from the population at each generation.

We define the termination criterion by specifying the number of generations as follows:

```
from pymoo.termination import get_termination
termination = get_termination("n_gen", 40)
```

We can now run the solver to minimize both objective functions simultaneously:

```
from pymoo.optimize import minimize

res = minimize(problem,
               algorithm,
               termination,
               seed=1,
               save_history=True,
               verbose=True)
X = res.X
F = res.F
```

Finally, we print the best 10 solutions as follows:

```
print("Solutions found: ")
print("Number of units of product P1 produced on machines M1 and M2\n and
➥ Number of units of product P2 produced on machines M1 and M2 are:\n",
➥ np.asarray(X, dtype = 'int'))
np.set_printoptions(suppress=True, precision=3)
print("The total production costs and \n difference in production times
➥ between the two machines are:\n",F)
```

This code will produce output representing the best 10 non-dominated solutions obtained by NSGA-II and will look like this:

```
Solutions found:
Number of units of product P1 produced on machines M1 and M2
and Number of units of product P2 produced on machines M1 and M2 are:
[[ 90  18  39 300]
[ 91  19  39 297]
[ 91  16  12 300]
[ 90  12  30 310]
[ 90  14  21 300]
[ 90  14  47 328]
[ 34  48   1 305]
[ 87  13   3 299]
[ 91  11   7 297]
```

```
[ 30  51   0 300]]
The total production costs and
difference in production times between the two machines are:
[[34757.953    16.105]
[34935.538    13.813]
[29235.912   114.763]
[32498.687    43.463]
[30481.316    79.233]
[37228.051     0.652]
[26307.998   378.004]
[26388.316   150.968]
[27199.394   118.385]
[25980.561   392.176]]
```

As there is no universal best solution for these two objective functions, multi-criteria decision-making can be applied to select the best trade-off—the Pareto optimal. In pymoo, the decision-making procedure starts by defining boundary points called *ideal* and *nadir* points:

- *The ideal point*—This refers to the best possible values for each objective function that can be achieved in the entire feasible region of the problem. This point represents the scenario where all the objective functions are minimized simultaneously.
- *The nadir point*—This is the point where each objective function is maximized while satisfying all the constraints of the problem. It is the opposite of the ideal point and represents the worst possible values for each objective function in the entire feasible region of the problem.

These points are used in multi-objective optimization problems to normalize the objective functions and convert them to a common scale, allowing for a fair comparison of different solutions. The two points are calculated as follows:

```
approx_ideal = F.min(axis=0)
approx_nadir = F.max(axis=0)
nF = (F - approx_ideal) / (approx_nadir - approx_ideal)
```

We then define weights, which are required by the decomposition functions, based on the level of importance of each objective function from the developer's perspective:

```
weights = np.array([0.2, 0.8])
```

Weights for f1 and f2 respectively

A decomposition method is defined using the augmented scalarization function (ASF), discussed in Wierzbicki's "The use of reference objectives in multiobjective optimization" [5]:

```
from pymoo.decomposition.asf import ASF
decomp = ASF()
```

To find the best solutions, we choose the minimum ASF values calculated from all the solutions and use the inverse of the weights as required by ASF:

```
i = decomp.do(nF, 1/weights).argmin()

print("Best regarding ASF: Point \ni = %s\nF = %s" % (i, F[i]))

plt.figure(figsize=(7, 5))
plt.scatter(F[:, 0], F[:, 1], s=30, facecolors='none', edgecolors='blue')
plt.scatter(F[i, 0], F[i, 1], marker="x", color="red", s=200)
plt.title("Objective Space")
plt.xlabel("Total production costs")
plt.ylabel("Difference in production times")
plt.show()
```

The output is shown in figure 8.13.

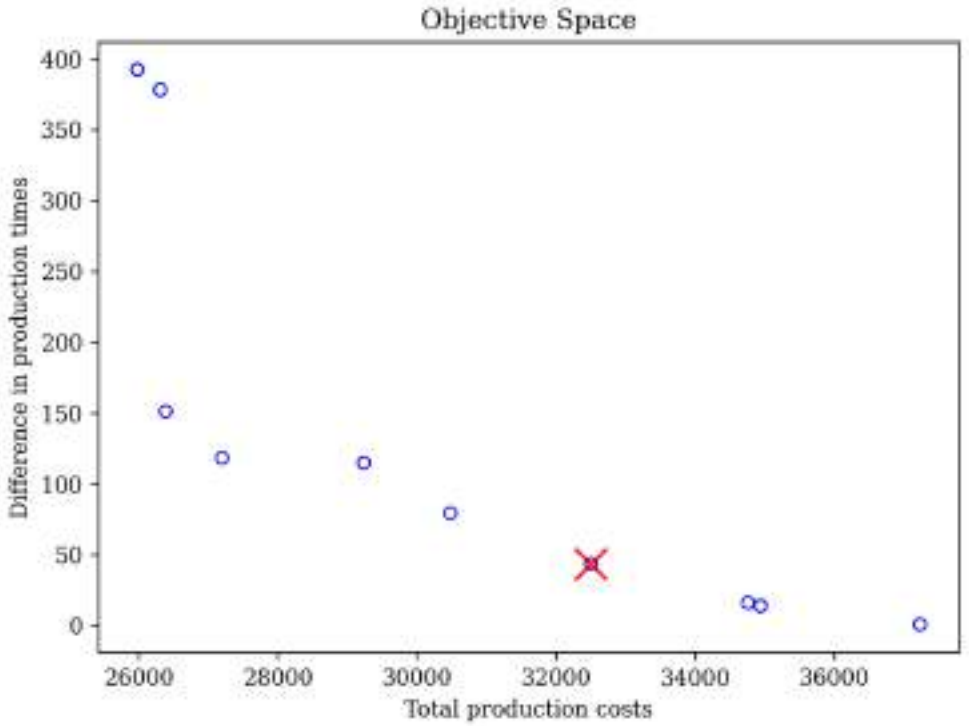


Figure 8.13 Manufacturing problem solution—the point marked with the X represents the selected Pareto optimal or best trade-off.

Running the code produces the following output:

```
The best solution found:
Number of units of product P1 produced on machines M1 and M2 are 90 and 12
respectively
Number of units of product P2 produced on machines M1 and M2 are 30 and 310
respectively
The total production costs are 32498.69
The difference in production times between the two machines is 43
```

The complete version of listing 8.3 is available in the book's GitHub repository. It includes another method using pseudo-weights to choose a solution from a solution set in the context of multi-objective optimization.

8.5 Adaptive GA

Adaptation methods help genetic algorithms strike a balance between exploration and exploitation, using different parameters such as initialization population size, crossover operators, and mutation operators. These parameters can be deterministically or dynamically adapted based on the search progress, allowing the algorithm to converge on high-quality solutions for complex optimization problems.

For example, population size can be adaptive. A larger population size promotes diversity and exploration, while a smaller size allows for faster convergence. The population size can be increased if the algorithm is struggling to find better solutions or decreased if the population has become too diverse.

Mutation operator parameters can be used to adapt the genetic algorithm and balance its exploration and exploitation aspects. For example, in the case of Gaussian mutation, we can adaptively set the value of the standard deviation σ of the Gaussian distribution during the run. The standard deviation of the Gaussian distribution can be changed following a deterministic approach, an adaptive approach, or a self-adaptive approach. If you're using a deterministic approach, the value of σ can be calculated in each generation using this formula: $\sigma(i) = 1 - 0.9 * i / N$ where i is the generation number, ranging from 0 to N (the maximum generation number). In this case, the value of σ is 1 at the beginning of the optimization process and gradually reduces to 0.1 toward the end to move the search algorithm's behavior from exploration to exploitation.

The adaptive approach incorporates feedback from the search process to adjust the variance and improve the search performance. Rechenberg's *1/5 success rule* is a well-known method that adjusts the step size of the search by monitoring the success rate of the search. This rule involves increasing the variance if a certain percentage of the previous mutations were successful in finding better solutions (i.e., if there was more than one successful mutation out of five tries), favoring exploration in order to avoid getting trapped in local optima. Otherwise, if there was a lower success rate, the variance should be decreased to favor exploitation. This allows the search to fine-tune its parameters based on its progress, leading to better performance and faster convergence to optimal solutions.

Figure 8.14 shows the steps of applying Rechenberg's 1/5 success rule. This update rule is applied in every generation, and a constant $0.82 <= c <= 1$ is used to update the standard deviation of the Gaussian distribution. As you can see, the higher the standard deviation, the higher the value of x, and the higher the deviation from the current solution (more exploration), and vice versa.

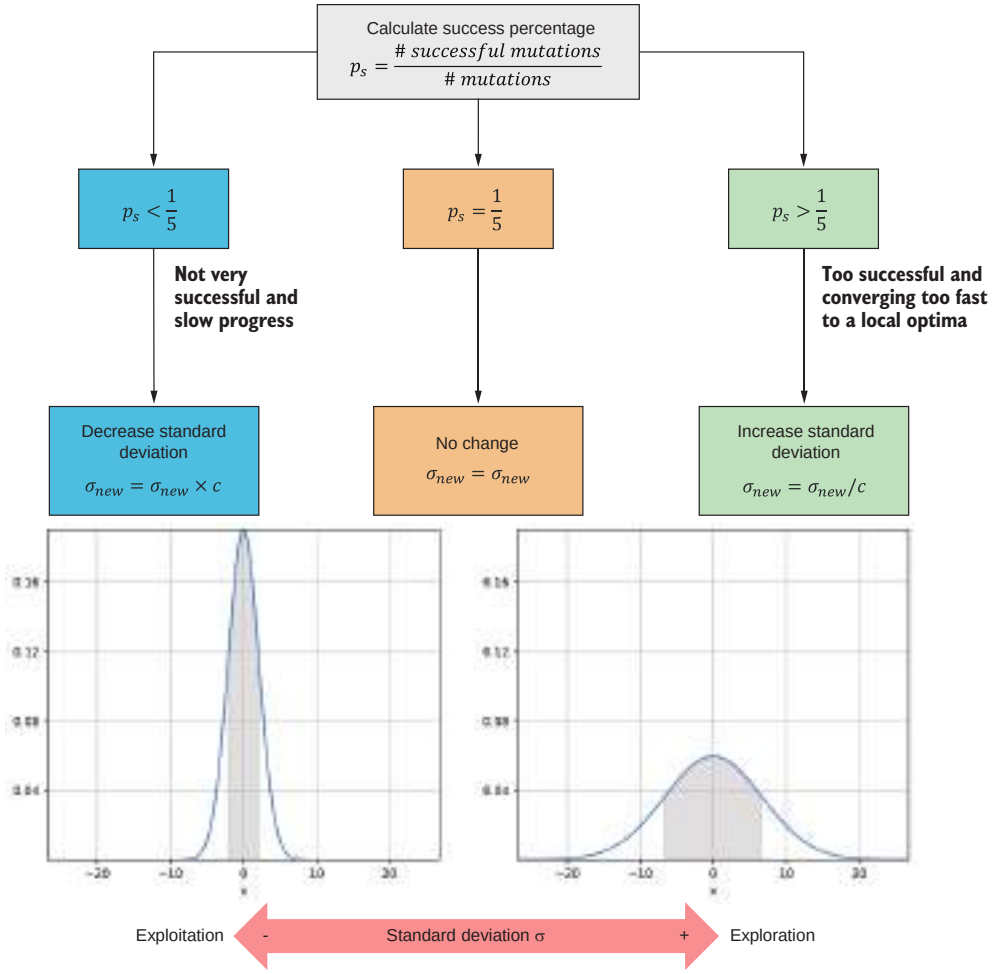


Figure 8.14 Rechenberg's 1/5 success rule. Following this rule, the Gaussian distribution's standard deviation is updated by a constant. The higher the standard deviation, the higher the value of *x* (i.e., larger step size), the higher the deviation from the current solution (more exploration), and vice versa.

The self-adaptive approach incorporates the mutation step size into each individual—a technique originally employed in evolution strategies (ES). In this method, the value of σ (the standard deviation or the mutation step size) evolves alongside the individual, resulting in distinct mutation step sizes for each individual in the population. The following equations are used in this self-adaptive approach:

$$\sigma' = \sigma e^{N(0,\tau_0)} \quad \textbf{8.2}$$

$$x_i' = x_i + N(0, \sigma') \tag{8.3}$$

where τ_0 is the learning rate.

Now that you have a solid understanding of the various components of GAs, we can apply this powerful optimization technique to real-world problems. In the following sections, we will use GAs to solve three distinct problems: the traveling salesman problem, tuning the parameters of a PID controller, and the political districting problem.

8.6 Solving the traveling salesman problem

Let's consider the following traveling salesman problem (TSP) for 20 major cities in the USA, starting from New York City, as illustrated in figure 8.15.

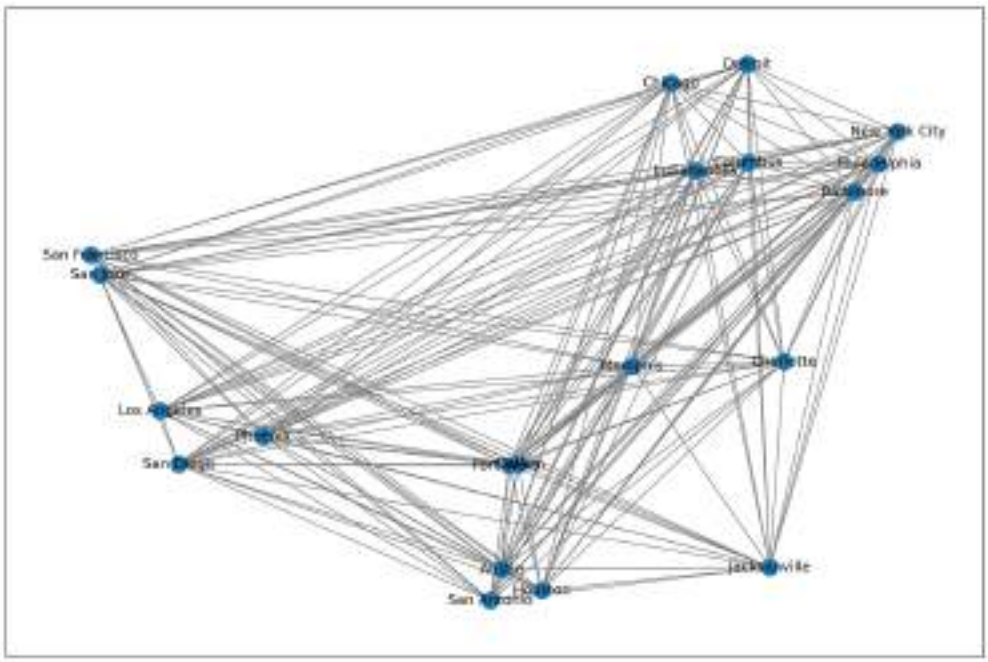


Figure 8.15 The 20 major US cities TSP

In listing 8.4, we start by importing the libraries we'll use and defining the TSP. First, we define the city names and their latitudes and longitudes. We then use those coordinates to create a haversine distance matrix and then convert the data dictionary into a dataframe.

Listing 8.4 Solving TSP using GA

```
import numpy as np
import pandas as pd
import networkx as nx
```

```
from collections import defaultdict
from haversine import haversine
import matplotlib.pyplot as plt
from pymoo.core.problem import ElementwiseProblem
from pymoo.core.repair import Repair
from pymoo.algorithms.soo.nonconvex.ga import GA
from pymoo.optimize import minimize
from pymoo.operators.sampling.rnd import PermutationRandomSampling
from pymoo.operators.crossover.ox import OrderCrossover
from pymoo.operators.mutation.inversion import InversionMutation
from pymoo.termination.default import DefaultSingleObjectiveTermination
from pymoo.optimize import minimize

cities = {
    'New York City': (40.72, -74.00),
    'Philadelphia': (39.95, -75.17),
    'Baltimore': (39.28, -76.62),
    'Charlotte': (35.23, -80.85),
    'Memphis': (35.12, -89.97),
    'Jacksonville': (30.32, -81.70),
    'Houston': (29.77, -95.38),
    'Austin': (30.27, -97.77),
    'San Antonio': (29.53, -98.47),
    'Fort Worth': (32.75, -97.33),
    'Dallas': (32.78, -96.80),
    'San Diego': (32.78, -117.15),
    'Los Angeles': (34.05, -118.25),
    'San Jose': (37.30, -121.87),
    'San Francisco': (37.78, -122.42),
    'Indianapolis': (39.78, -86.15),
    'Phoenix': (33.45, -112.07),
    'Columbus': (39.98, -82.98),
    'Chicago': (41.88, -87.63),
    'Detroit': (42.33, -83.05)
}

distance_matrix = defaultdict(dict)
for ka, va in cities.items():
    for kb, vb in cities.items():
        distance_matrix[ka][kb] = 0.0 if kb == ka else
haversine((va[0], va[1]),
➥ (vb[0], vb[1]))

distances = pd.DataFrame(distance_matrix)
city_names=list(distances.columns)
distances=distances.values

G=nx.Graph()
for ka, va in cities.items():
    for kb, vb in cities.items():
        G.add_weighted_edges_from({(ka,kb, distance_matrix[ka][kb])})
        G.remove_edges_from(nx.selfloop_edges(G))
```

Define city names, latitudes, and longitudes for 20 major US cities.

Create a haversine distance matrix based on latitude-longitude coordinates.

Convert the distance dictionary into a dataframe.

Create a networkx graph.

We can then create `TravelingSalesman` as a subclass of the `ElementwiseProblem` class available in pymoo. This class defines the number of cities and the intercity distances as problem parameters, and it evaluates the total path length as an objective function to be minimized:

```
class TravelingSalesman(ElementwiseProblem):

    def __init__(self, cities, distances, **kwargs):
        self.cities = cities
        n_cities = len(cities)
        self.distances = distances

        super().__init__(
            n_var=n_cities,
            n_obj=1,
            xl=0,
            xu=n_cities,
            vtype=int,
            **kwargs
        )

    def _evaluate(self, x, out, *args, **kwargs):
        f = 0
        for i in range(len(x) - 1):
            f += distances[x[i], x[i + 1]]
        f += distances[x[-1], x[0]]
        out["F"] = f
```

The following function is a subclass of the `Repair` class, and it provides a method to repair solutions for the TSP, ensuring that each solution starts with the city indexed as 0 (New York City, in this example). The repair operator in pymoo is used to make sure the algorithm is only searching the feasible space. It is applied after the offspring have been reproduced:

```
class StartFromZeroRepair(Repair):

    def _do(self, problem, X, **kwargs):
        I = np.where(X == 0)[1]

        for k in range(len(X)):
            i = I[k]
            X[k] = np.concatenate([X[k, i:], X[k, :i]])

        return X
```

It's time now to define a GA solver and apply it to solve the problem.

```
problem = TravelingSalesman(cities,distance_matrix)

algorithm = GA(
    pop_size=20,
    sampling=PermutationRandomSampling(),
    mutation=InversionMutation(),
    crossover=OrderCrossover(),
```

Create a TSP instance for the given cities and intercity distances.

```
    repair=StartFromZeroRepair(),
    eliminate_duplicates=True
)

termination = DefaultSingleObjectiveTermination(period=300, n_max_gen=np.inf)

res = minimize(
    problem,
    algorithm,
    termination,
    seed=1,
    verbose=False
)
```

Define the GA solver.

Terminate (and disable the max generations) if the algorithm did not improve in the last 300 generations.

Find the shortest path.

We can print the found route and its length as follows:

```
Order = res.X
Route = [city_names[i] for i in Order]
arrow_route = ' → '.join(Route)
print("Route:", arrow_route)
print("Route length:", np.round(res.F[0], 3))
print("Function Evaluations:", res.algorithm.evaluator.n_eval)
```

This results in the following output:

```
Route: New York City → Detroit → Columbus → Indianapolis → Chicago → San
Francisco → San Jose → Los Angeles → San Diego → Phoenix → San Antonio →
Austin → Houston → Fort Worth → Dallas → Memphis → Jacksonville → Charlotte →
Baltimore → Philadelphia
Route length: 10934.796
Function Evaluations: 6020
```

The following code is used to visualize the obtained route using NetworkX:

```
fig, ax = plt.subplots(figsize=(15,10))

H = G.copy()

reversed_dict = {key: value[::-1] for key, value in cities.items()}

keys_list = list(cities.keys())

included_cities = {keys_list[index]: cities[keys_list[index]] for index in
➥ list(res.X)}
included_cities_keys=list(included_cities.keys())

edge_list =list(nx.utils.pairwise(included_cities_keys))

nx.draw_networkx_edges(H, pos=reversed_dict, edge_color="gray", width=0.5)

ax=nx.draw_networkx(
    H,
    pos=reversed_dict,
    with_labels=True,
    edgelist=edge_list,
    edge_color="red",
```

Create an independent shallow copy of the problem graph and attributes.

Reverse latitude and longitude for correct visualization.

Create a list of keys in the original dictionary.

Create a new dictionary with the keys in the desired order.

Create an edge list.

Draw the closest edges on each node only.

```
    node_size=200,
    width=3,
)
plt.show()
```

Draw and show the route.

Figure 8.16 shows the obtained route for this traveling salesman problem.

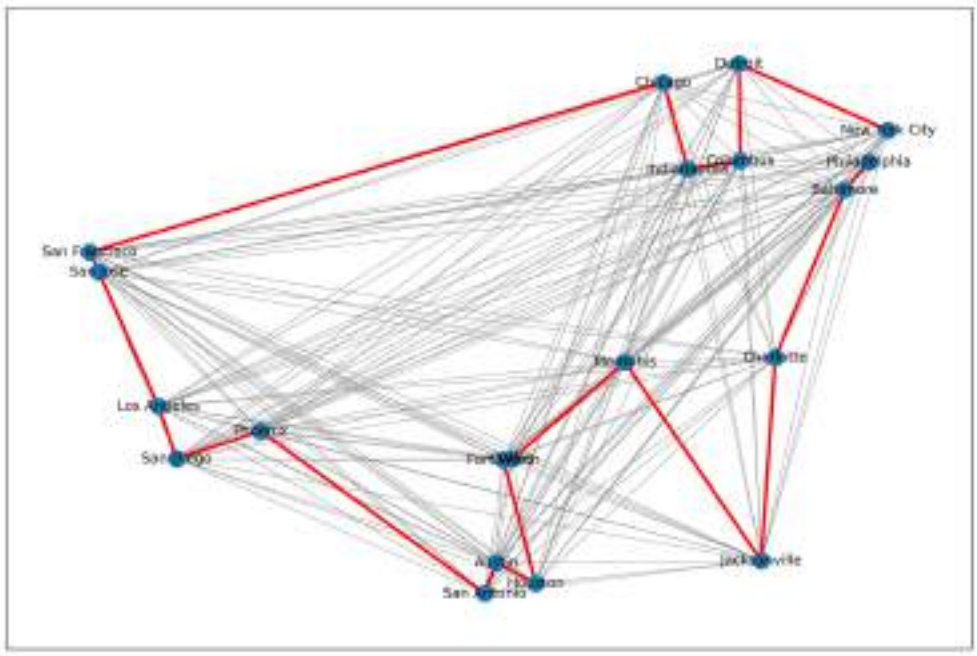


Figure 8.16 The 20 major US cities TSP solution

You can experiment with the full code in the book's GitHub repository by changing the problem data and the genetic algorithm parameters, such as population size, sampling, crossover, and mutation methods.

8.7 PID tuning problem

Have you ever wondered how your room stays at a comfortable temperature? Have you ever thought about how the heating or cooling system knows when to turn on and off automatically to maintain the temperature set on the thermostat? This is where control systems come into the picture. Control systems are like behind-the-scenes wizards that ensure things work smoothly and efficiently. They are sets of rules and mechanisms that guide devices or processes to achieve specific goals.

One type of control system is a *closed-loop system*. Picture this: you've set your room's thermostat to a cozy 22°C (72°F), and the heating or cooling system kicks in to reach that temperature. But what happens if it becomes too chilly or too warm? That's when the closed-loop system starts to take action. It's continually tracking the room's current

temperature, comparing it to the desired temperature, and making the necessary heating or cooling tweaks.

The proportional integral derivative (PID) controller is the most commonly used algorithm in control systems engineering. This controller is designed to compensate for any error between the measured state (e.g., the current room temperature) and the desired state (e.g., the desired temperature value). Let's consider room temperature control using a PID controller as an example.

As shown in figure 8.17, the controller takes the error signal $e(t)$ (the difference between the desired state and the feedback signal) and produces the appropriate control signal $u(t)$ to turn on or off the heater in order to minimize the difference between the current room temperature and the desired value. The control signal is calculated using equation 8.4:

$$u(t) = K_p e(t) + K_i \int_o^t e(t)dt + K_d \frac{d}{dt} e(t) \qquad \textbf{8.4}$$

As shown in this equation, the *proportional term* $K_p e(t)$ tends to produce a control signal that is proportional to the error and aims to rectify it. The *integral term* (the second term on the right side of the equation) tends to produce a control signal that is proportional to the magnitude of the error and its duration, or the area under the error curve. The *derivative term* (the third term on the right side of the equation) tends to produce a control signal that is proportional to the rate of error change, thus providing an anticipatory control signal.

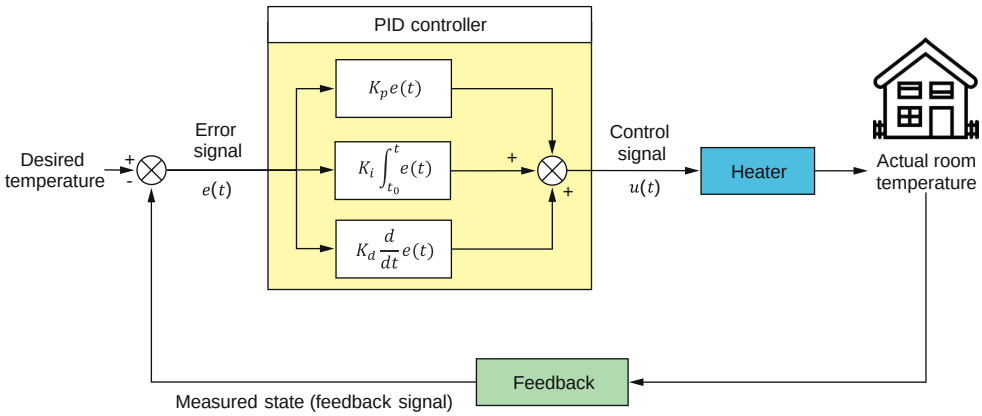


Figure 8.17 PID-based closed-loop control system—the PID controller takes the error signal and produces a control signal to reduce the error to zero.

Utilizing a PID controller allows the system (e.g., an air conditioner or heater) to follow the specified input and attain a desired or optimal steady-state error, rise time, settling time, and overshoot:

- *Rise time*—The rise time is the time required for the response to rise from 10% to 90% of its final value.
- *Peak overshoot*—The peak overshoot (aka maximum overshoot) is the deviation of the response at peak time from the final value of the response.
- *Settling time*—The settling time is the time required for the response to reach the steady state and stay within the specified tolerance bands (e.g., 2–5% of the final value) after the transient response has settled.
- *Steady-state error*—The steady-state error is the difference between the desired value and the actual value of the system output when the system has reached a stable condition.

As shown in figure 8.18, the heater is turned on (i.e., energized) when the current room temperature is lower than the set point or the desired value. The heater is turned off (i.e., de-energized) when the temperature is above the set point.

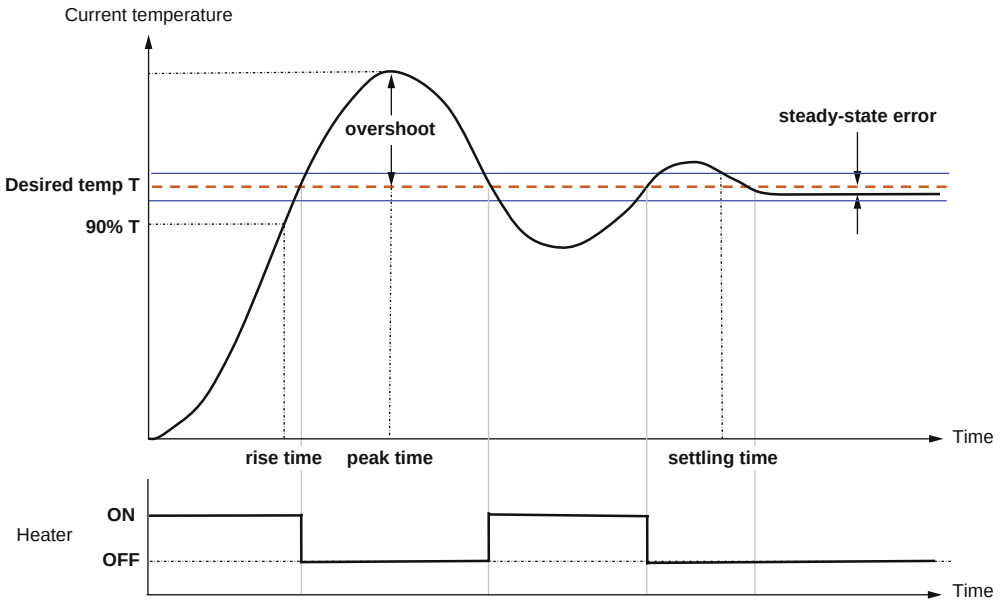


Figure 8.18 Step response of a system. The heater is turned on or off according to the difference between the actual temperature and the desired value.

Table 8.6 shows the effect of PID controller parameters on the time response of the system. Note that these correlations may not be exactly accurate, because K_p, K_i, and K_d are dependent on each other. In fact, changing one of these variables can change the effect of the other two. For this reason, the table should only be used as a reference when you are determining the values for K_p, K_i, and K_d.

Table 8.6 Effects of adding PID controller parameters on the system's response

Parameter	Rise time	Overshoot	Settling time	Steady-state error
K_p	Decreases	Increases	Small change	Decreases
K_i	Decreases	Increases	Increases	Decreases significantly
K_d	Small change	Decreases	Decreases	Small change

Finding the optimal values of the PID controller parameters for an optimal controller response is a multivariate optimization problem commonly referred to as the *PID tuning problem*. The following four performance metrics are commonly used to evaluate the quality of a control system such as a PID controller:

- *ITAE (integral time absolute error)*—This metric penalizes errors that persist over time, making it suitable for systems where transient response and settling time are important. It is calculated using this formula: ITAE = $\int (t|e(t)|)\ dt$, where t is the time, $e(t)$ is the error at time t defined as $e(t) = r(t) - y(t)$, $r(t)$ is the reference signal (desired output) at time t (for step response $r(t) = 1$), and $y(t)$ is the actual output of the system at time t.
- *ITSE (integral time square error)*—Like ITAE, this metric also penalizes errors that last for a long time but places more emphasis on larger errors due to the squared term. It is calculated using this formula: ITSE = $\int (te(t)^2)\ dt$.
- *IAE (integral absolute error)*—This metric measures the overall magnitude of the error without considering the duration of the error. This is a simple and widely used performance metric, and it's calculated using this formula: IAE = $\int |e(t)|\ dt$.
- *ISE (integral squared error)*—This metric emphasizes larger errors due to the squared term, making it useful for systems where minimizing large errors is a priority. It is calculated using this formula: ISE = $\int e(t)^2\ dt$. It penalizes errors more heavily if they occur later in the evolution of the response. It also penalizes an error E for time dt more heavily than E/α for time αdt, where $\alpha > 1$. This expected response may have a slow rise time but with a more oscillatory behavior.
- *Combined criteria*—This metric combines overshoot, rise time, settling time, and steady-state error [6]. It is calculated using this formula: $W = (1 - e^{-\beta})(M_p + \text{error}_{ss}) + e^{-\beta}(t_s - t_r)$, where M_p is the overshoot, error_{ss} is the steady-state error, t_s is the settling time, t_r is the rise time, and β is a balancing factor in the range of 0.8 to 1.5. You can set β to be larger than 0.7 to reduce the overshoot and steady-state error. On the other hand, you can set β to be smaller than 0.7 to reduce the rise time and settling time.

Each of these metrics quantifies the error between the desired output and the actual output of the system in different ways, emphasizing different aspects of the control system's performance. Note that performance metrics are not strictly confined to the aforementioned metrics. Engineers have the flexibility to devise custom performance

metrics tailored to the specific goals and characteristics of the control system under consideration.

Figure 8.19 shows a closed-loop control system where a transfer function is used to describe the relationship between the input and output of the system in a Laplace domain. This domain is a generalization of a frequency domain, providing a more comprehensive representation that includes transient behavior and initial conditions. Assume that T_{sp} is the set point or the desired output and G represents the transfer functions indicated in the block diagram.

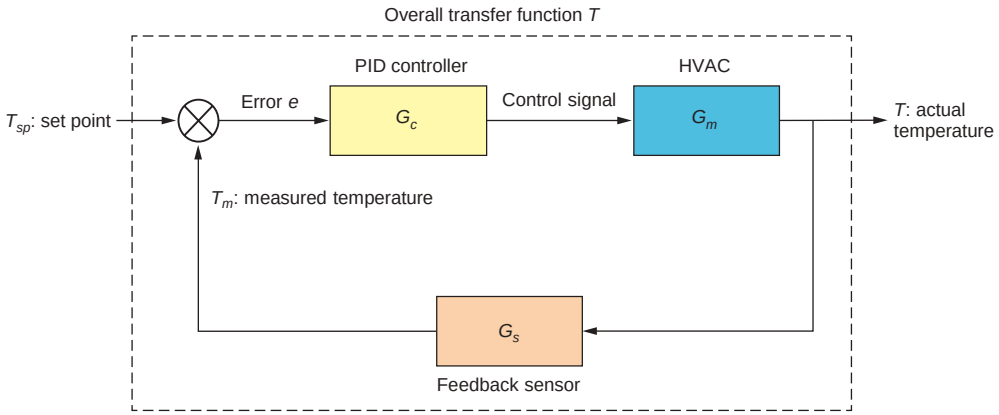


Figure 8.19 Closed-loop control system

All variables are a function of s, which is the output variable from a Laplace transform. The transfer function of a PID controller is given by this equation:

$$G_c(s) = \left(K_p + \frac{K_i}{s} + K_d s\right) = \frac{K_d s^2 + K_p s + K_i}{s} \quad \textbf{8.5}$$

where K_p is the proportional gain, K_i is the integral gain, and K_d is the derivative gain. Assume that the transfer function of the HVAC system is given by this equation:

$$G_m(s) = \frac{1}{s^2 + 10s + 20} \quad \textbf{8.6}$$

Assuming that $G_s = 1$ (unity feedback) and using block diagram reduction, we can find the overall transfer function $T(s)$ of the closed loop system:

$$T(s) = \frac{K_d s^2 K_p s + K_i}{s^3 + (K_d + 10)\, s^2 + (K_p + 20)\, s + K_i} \quad \textbf{8.7}$$

Let's now look at how we can find the optimal values for the PID parameters with Python. In the next listing, we start by importing the libraries we'll use and defining the overall transfer function of the control system.

Listing 8.5 Solving the PID tuning problem using GA

```
import numpy as np
import control
import math
import matplotlib.pyplot as plt
from pymoo.algorithms.soo.nonconvex.ga import GA
from pymoo.operators.crossover.pntx import PointCrossover
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

def transfer_function(Kp,Ki,Kd):
    num = np.array([Kd,Kp,Ki])
    den = np.array([1,(Kd+10),(Kp+20),Ki])
    T = control.tf(num, den)
    t, y = control.step_response(T)
    return T, t, y
#A Import the control module.
```

Import the control module.

Take PID parameters as input.

Define the numerator of the transfer function.

Define the denominator of the transfer function.

Create a transfer function.

Get time response output using a step function as a system input.

Next, we can define the objective functions or performance criteria:

```
def objective_function(t, error, Kp,Ki,Kd, criterion):

    if criterion == 1:
        ITAE = np.trapz(t, t*error)
        objfnc= ITAE
    elif criterion == 2:
        ITSE = np.trapz(t, t*error**2)
        objfnc= ITSE
    elif criterion == 3:
        IAE =  np.trapz(t, error)
        objfnc= IAE
    elif criterion == 4:
        ISE = np.trapz(t, error**2)
        objfnc= ISE
    elif criterion == 5:
        T, _, _ =transfer_function(Kp,Ki,Kd)
        info = control.step_info(T)
        beta = 1
        Mp = info['Overshoot']
        tr = info['RiseTime']
        ts = info['SettlingTime']
        ess = abs(1-info['SteadyStateValue'])
        W = ((1-math.exp(-beta))*(Mp+ess))+((math.exp(-beta))*(ts-tr))
        objfnc=W;

    return objfnc
```

ITAE (integral time absolute error)

ITSE (integral time square error)

IAE (integral absolute error)

ISE (integral squared error)

W (combined criteria)

We can now define the optimization problem for the PID controller:

```
class PIDProblem(Problem):

    def __init__(self):
        super().__init__(n_var=3,
                         n_obj=1,
                         n_constr=0,
                         xl=0,
                         xu=100,
                         vtype=float)

    def _evaluate(self, X, out, *args, **kwargs):
        f = np.zeros((X.shape[0], 1))
        for i, params in enumerate(X):
            Kp, Ki, Kd = params
            T, t, y =transfer_function(Kp,Ki,Kd)
            error = 1 - y
            f[i]=objective_function(t, np.abs(error), Kp,Ki,Kd, 5)
        out["F"] = f
```

Three decision variables, representing the PID controller's Kp, Ki, and Kd gains

Number of objective functions

No constraints

Lower and upper bounds for the decision variables

Evaluate the objective function.

Next, we can set up and solve the PID tuning problem using GA. The previously defined `PIDProblem` class is used to model the optimization problem. The GA solver is configured with a population size of 50. Initial solutions are sampled using `FloatRandomSampling`, and the crossover operation employs a two-point crossover with a probability of 0.8. Additionally, polynomial mutation is applied with a probability of 0.3, and the algorithm runs for 60 generations:

```
problem = PIDProblem()

algorithm = GA(
    pop_size=50,
    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', 60), seed=1, verbose=True)
```

Let's now print the results:

```
best_params = res.X
print("Optimal PID controller parameters:")
print("Kp =", best_params[0])
print("Ki =", best_params[1])
print("Kd =", best_params[2])
```

And we'll visualize the time response:

```
Kp = best_params[0]
Ki = best_params[1]
Kd = best_params[2]
```

```
T, t, y =transfer_function(Kp,Ki,Kd)

plt.plot(t,y)
plt.title("Step Response")
plt.xlabel("Time (s)")
plt.grid()
```

Figure 8.20 depicts the step response of the system, demonstrating how its outputs change over time when the inputs swiftly transition from 0 to 1.

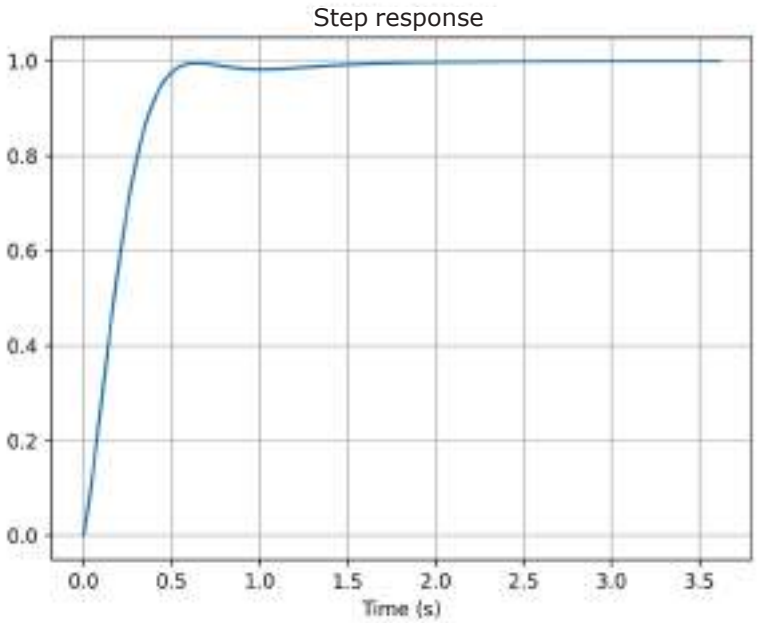


Figure 8.20 Step response

To show the step response characteristics (rise time, settling time, peak, and others), you can use the following function:

```
control.step_info(T)
```

This results in the following output:

```
{'RiseTime': 0.353,
 'SettlingTime': 0.52,
 'SettlingMin': 0.92,
 'SettlingMax': 1.0,
 'Overshoot': 0,
 'Undershoot': 0,
 'Peak': 0.99,
 'PeakTime': 3.62,
 'SteadyStateValue': 1.0}
```

You can experiment with adjusting the algorithm's parameters (such as population size, crossover method and probability, mutation method and probability, number of generations, etc.) and altering the performance metric to observe the effects on the system's performance.

8.8 Political districting problem

I introduced political districting in section 2.1.5—it can be defined as the process of grouping n subregions within a territory into m electoral districts while adhering to certain constraints. Suppose we need to merge n neighborhoods in the City of Toronto into m electoral districts while ensuring a sufficient level of population equality. Figure 8.21 shows a sample dataset that contains population and median household income for 16 neighborhoods in East Toronto.

Figure 8.21 The 16 neighborhoods in East Toronto with their population and median household income

In addressing the political districting problem, a viable solution must ensure that there is a satisfactory degree of population equilibrium (i.e., a fair and balanced distribution) in every electoral district. For example, we can evaluate a district's population balance by calculating the deviation from an ideal population size within an upper bound (pop_{UB}) and lower bound (pop_{LB}) as follows:

$$pop_{UB} = \text{ceil}\left(\left(pop_{av} + pop_{margin}\right) \times \frac{n}{m}\right)$$

$$pop_{LB} = \text{floor}\left(\left(pop_{av} - pop_{margin}\right) \times \frac{n}{m}\right)$$

where pop_{av} represents the target population size that can be considered the average of all the neighborhoods and pop_{margin} indicates the acceptable degree of deviation from the ideal population size. n is the number of the neighborhoods, and m is the number of districts.

A district will be regarded as overpopulated if its total population exceeds the upper bound, and conversely, a district will be deemed underpopulated if its total population falls below the lower bound. A district whose population falls within the upper and lower bounds will be regarded as having an appropriate population size. The objective function is to minimize the total number of overpopulated and underpopulated districts. The search process will persist until the objective function's minimum value is obtained, ideally zero. This indicates that no districts are either overpopulated or underpopulated.

The next listing shows how to find the political districts using GA. We'll start by reading the data from a local folder or using a URL.

Listing 8.6 Solving a political districting problem using GA

```
import geopandas as gpd
import pandas as pd
import folium

data_url="https://raw.githubusercontent.com/Optimization-Algorithms-Book/
Code-Listings/main/Appendix%20B/data/PoliticalDistricting/"

toronto = gpd.read_file(data_url+"toronto.geojson")
neighborhoods = pd.read_csv(data_url+"Toronto_Neighborhoods.csv")

range_limit = 16
toronto_sample = toronto.tail(range_limit)
values = neighborhoods.tail(range_limit)
values = values.join(toronto_sample["cartodb_id"])
```

Read the neighborhood information (e.g., names, populations, and median household incomes).

Read the Toronto region administration boundaries.

URL for the data folder

Pick 16 neighborhoods as a subset to represent neighborhoods in East Toronto.

After reading the dataset, we'll do the following data preprocessing to get the population of each neighborhood, and the adjacency relationship among every possible pair of neighborhoods, in a Boolean matrix.

```
import numpy as np

def get_population(lst, table):
    return table["population"].iloc[lst].to_numpy()

eval = get_population(range(range_limit), values)

def get_neighboors(database):
    result = []
    for i in range(database['name'].size):
        tmp = np.zeros(database['name'].size)
        geo1 = database.iloc[i]
        for j in range(database['name'].size):
            if i != j:
```

Get the population of each neighborhood.

Prepare the population dataset.

Represent the adjacency relationship among every possible pair of neighborhoods.

```
                geo2 = database.iloc[j]
                if geo1["geometry"].intersects(geo2["geometry"]):
                    tmp[j] = 1
        result.append(tmp)
    return np.stack(result)

neighbor = get_neighboors(toronto_sample)
```

We'll now define the political districting class with a single objective function, three constraints, a given number of districts, a given population margin, and an adjacency matrix between the neighborhoods. `PoliticalDistricting` is a custom problem class that extends the `Problem` class from pymoo. The `Problem` class implements a method that evaluates a set of solutions instead of a single solution at a time, like in the case of the `ElementwiseProblem` class. In the `PoliticalDistricting` class, the following parameters are defined:

- `num_dist`—The number of districts to divide the region into
- `neighbor`—A matrix representing the neighborhood relationships between locations in the region
- `populations`—The population of each neighborhood
- `margin`—The acceptable degree of deviation from the ideal population size
- `average`—The average population
- `n_var`—The number of decision variables, which is equal to the number of neighborhoods
- `n_obj=1`—The number of objectives, which is 1 for this problem
- `n_eq_constr=3`—The number of equality constraints, which is 3 for this problem
- `xl=0`—The lower bound for the decision variables, which is 0 for this problem
- `xu=num_dist-1`—The upper bound for the decision variables, which is `num_dist-1` for this problem
- `vtype=int`—The type of decision variables, which is integer for this problem

The following code shows how to define a `PoliticalDistricting` class with different parameters, such as the number of districts, neighbor information, populations, and margin:

```
from pymoo.core.problem import Problem

class PoliticalDistricting(Problem):
    def __init__(self,
                 num_dist,
                 neighbor,
                 populations,
                 margin
                 ):

        self.populations = populations
        self.average = np.mean(populations)
```

Define a constructor with specific parameters.

Hold the population data.

Store the mean population of all districts.

```
        super().__init__(n_var=len(self.populations), n_obj=1, n_eq_constr=3,
➥ xl=0, xu=num_dist-1, vtype=int)

        self.n_var = len(self.populations)
        self.n_dist = num_dist
        self.margin = margin
        self.neighbor = neighbor
        self.func = self._evaluate
```

Call the constructor of the parent class with specific parameters.

Evaluate the solution against the objective function and constraints.

As a continuation and as part of the `problem` class, we'll extract the neighborhoods that belong to a specific district using the following function:

```
def _gather(self, x, district):
    return np.where(x==district, 1, 0)
```

We'll then calculate the upper and lower bounds based on the given population values and margin as follows:

```
def _get_bounds(self):
    ub = np.ceil(self.average + self.margin) *
➥ (len(self.populations)/self.n_dist)
    lb = np.ceil(self.average - self.margin) *
➥ (len(self.populations)/self.n_dist)
    return ub, lb
```

The following function is used to decide whether an electoral district is overpopulated or underpopulated:

```
def _get_result(self, gathered, ub, lb):
    product = gathered * self.populations
    summed_product = np.sum(product, axis=1)
    return np.where((summed_product > ub), 1, 0) + np.where((summed_product <
➥ lb), 1, 0)
```

As all the constraints are equality constraints, the following function returns true if the constraint is satisfied:

```
def _get_constraint(self, constraint):
    constraint = np.stack(constraint)
    return np.any(constraint==0, axis=0)
```

To make sure that there is no isolated neighborhood far from other neighbors within a district, unless the district only has one neighborhood, the following function is used:

```
def _get_neighbor(self, gathered):
    singleton = np.sum(gathered, axis=1)
    singleton = np.where(singleton==1, True, False)
    tmp_neighbor = np.dot(gathered, self.neighbor)
    tmp_neighbor = np.where(tmp_neighbor > 0, 1, 0)
    product = gathered * tmp_neighbor
    return np.all(np.equal(product, gathered), axis=1) + singleton
```

The following function determines the best approximation to make an electoral district a contiguous block:

```
def cap_district(self, gathered):
    result = np.zeros(gathered.shape[0])
    for i in range(len(gathered)):
        nonzeros = np.nonzero(gathered[i])[0]
        if nonzeros.size != 0:
            mx = np.max(nonzeros)
            mn = np.min(nonzeros)
            result[i] = self.neighbor[mx][mn] or (mx == mn)
    return result
```

The last function in the `problem` class is used to evaluate the solution against the objective function, including checking the constraints:

```
    def _evaluate(self, x, out, *args, **kwargs):
        x=np.round(x).astype(int) # Ensure X is binary
        pop_count = []
        constraint1 = []
        constraint2 = []
        constraint3 = []
        for i in range(self.n_dist):
            gathered = self._gather(x, i)
            ub, lb = self._get_bounds()
            result = self._get_result(gathered, ub, lb)
            pop_count.append(result)
            constraint1.append(np.sum(gathered, axis=1))
            constraint2.append((self._get_neighbor(gathered)))
            constraint3.append(self.cap_district(gathered))

        holder = np.sum(np.stack(pop_count), axis=0)
        out["F"] = np.expand_dims(holder, axis=1)
        out["H"] = [self._get_constraint(constraint1),
                    self._get_constraint(constraint2),
                    self._get_constraint(constraint3)]
```

Constraint 1: make sure that there is no empty district,

Constraint 2: make sure there is no lone neighborhood within a district, unless the district only has one neighborhood.

Constraint 3: ensure the electoral district is a contiguous block by achieving the best possible approximation.

```
def create_districting_problem(number_of_districts, neighborlist, population_
➥ list, margin, seed=1):
    np.random.seed(seed)
    problem = PoliticalDistricting(number_of_districts, neighborlist,
➥ population_list, margin)
    return problem
```

We can now define the GA solver and apply it to solve the problem as follows:

```
from pymoo.algorithms.soo.nonconvex.ga import GA
from pymoo.operators.sampling.rnd import FloatRandomSampling
from pymoo.operators.crossover.pntx import PointCrossover
from pymoo.operators.mutation.pm import PolynomialMutation
from pymoo.operators.repair.rounding import RoundingRepair
from pymoo.termination import get_termination
from pymoo.optimize import minimize

num_districts = 3
margin=6000
```

```
problem = create_districting_problem(num_districts, neighbor, eval, margin,
seed=1)

algorithm = GA(
    pop_size=2000,
    sampling=FloatRandomSampling(),
    crossover=PointCrossover(prob=0.8, n_points=2),
    mutation = PolynomialMutation(prob=0.3, repair=RoundingRepair()),
    eliminate_duplicates=True
)

termination = get_termination("n_gen", 100)

res = minimize(problem,
               algorithm,
               termination,
               seed=1,
               save_history=True,
               verbose=True)
```

The resultant political districts are listed here and visualized in figure 8.22:

```
Political District- 1 :  ['Woburn', 'Highland Creek', 'Malvern']
Political District- 2 :  ['Bendale', 'Scarborough Village', 'Guildwood',
'Morningside', 'West Hill', 'Centennial Scarborough', 'Agincourt South-
Malvern West']
Political District- 3 :  ['Rouge', 'Hillcrest Village', 'Steeles',
"L'Amoreaux", 'Milliken', 'Agincourt North']
```

Figure 8.22 The three political districts that combine the 16 neighborhoods

The problem is treated as a single objective optimization problem where the objective is to minimize the total number of overpopulated and underpopulated districts. The dataset contains the median household income of each neighborhood, so you can replace the objective function to focus on the heterogeneity of the median household income. You can also treat the problem as a multi-objective optimization problem by considering both criteria.

This chapter marks the end of the third part of the book, which focused on genetic algorithms and their applications in solving complex optimization problems. The fourth part of the book will delve into the fascinating realm of swarm intelligence algorithms.

Summary

- The Hamming cliff problem, which results from the inherent nature of binary representation, negatively affects binary-coded GAs by disrupting the search space's smoothness, causing poor convergence and leading to inefficient exploration and exploitation. To address this problem, alternative representations like Gray code or real-valued encoding can be used, as they offer better locality and smoother search spaces, minimizing the disruptive effects of small changes on decoded values.
- Real-valued GA is well suited for optimization problems involving continuous variables or real-valued parameters. It offers benefits such as better representation precision, faster convergence, diverse crossover and mutation operations, and reduced complexity, making it an attractive choice for many continuous optimization problems.
- Permutation-based GA is a class of genetic algorithms specifically designed to handle combinatorial optimization problems where the solutions can be represented as ordered sequences, or permutations, of elements.
- Multi-objective optimization problems can be tackled using either a preference-based multi-objective optimization method or a Pareto optimization approach. In the preference-based method, the multiple objectives are combined into a single or overall objective function by using a weighting scheme. The Pareto optimization approach focuses on identifying multiple trade-off optimal solutions known as Pareto-optimal solutions. These solutions can be further refined using higher-level information or decision-making processes.
- Crossover is primarily exploitative, as it combines the genetic material of two parent individuals to produce offspring, promoting the exchange of beneficial traits between solutions. However, depending on the implementation, crossover can also have some explorative properties, as it can produce offspring with new combinations of genes, leading to the discovery of new solutions.
- Mutation can act as an explorative or exploitative operator depending on influencing factors such as the mutation rate and the mutation step size.
- In general, the crossover rate should be relatively high, as it promotes the exchange of genetic information between parent chromosomes. On the other hand, mutation is typically applied with a low probability, as its main purpose is to introduce random variations into the population.