# 2 A deeper look at search and optimization

***This chapter covers***

- Classifying optimization problems based on different criteria
- Classifying search and optimization algorithms based on the way the search space is explored and how deterministic the algorithm is
- Introducing heuristics, metaheuristics, and heuristic search strategies
- A first look at nature-inspired search and optimization algorithms

Before we dive into the problems and algorithms that I hinted at in chapter 1, it will be useful to be clear about how we talk about these problems and algorithms. Classifying problems allows us to group similar problems together and potentially exploit existing solutions. For example, a traveling salesman problem involving geographic values (i.e., cities and roads) may be used as a model to find the minimum length of wires connecting pins in a very large-scale integration (VLSI) design. The same can be said for classifying the algorithms themselves, as grouping algorithms with similar properties can allow us to easily identify the right algorithm to solve a problem and meet expectations, such as the quality of the solution and the permissible search time.

Throughout this chapter, we'll discuss common classifications of optimization problems and algorithms. Heuristics and metaheuristics will also be introduced as general algorithmic frameworks or high-level strategies that guide the search process. Many of these strategies are inspired by nature, so we'll shed some light on nature-inspired algorithms. Let's start by discussing how we can classify optimization problems based on different criteria.

## 2.1 Classifying optimization problems

Optimization is everywhere! In everyday life, you'll face different kinds of optimization problems. For example, you may like to set the thermostat to a certain temperature to stay comfortable and at the same time save energy. You may select light fixtures and adjust the light levels to reduce energy costs. When you start driving your electric vehicle (EV), you may search for the fastest or most energy-efficient route to your destination. Before arriving at your destination, you may look for a parking spot that is affordable, provides the shortest walking distance to your destination, offers EV charging, and is preferably underground. These optimization problems have different levels of complexity that mainly depend on the type of problem. As mentioned in the previous chapter, the process of optimization involves selecting decision variables from a given feasible search space in such a way as to optimize (minimize or maximize) a given objective function or, in some cases, multiple objective functions.

Optimization problems are characterized by three main components: decision variables or design vectors, objective functions or criteria to be optimized, and a set of hard and soft constraints to be satisfied. The nature of these three components, the permissible time allowed for solving the problem, and the expected quality of the solutions lead to different types of optimization problems, as shown in figure 2.1.

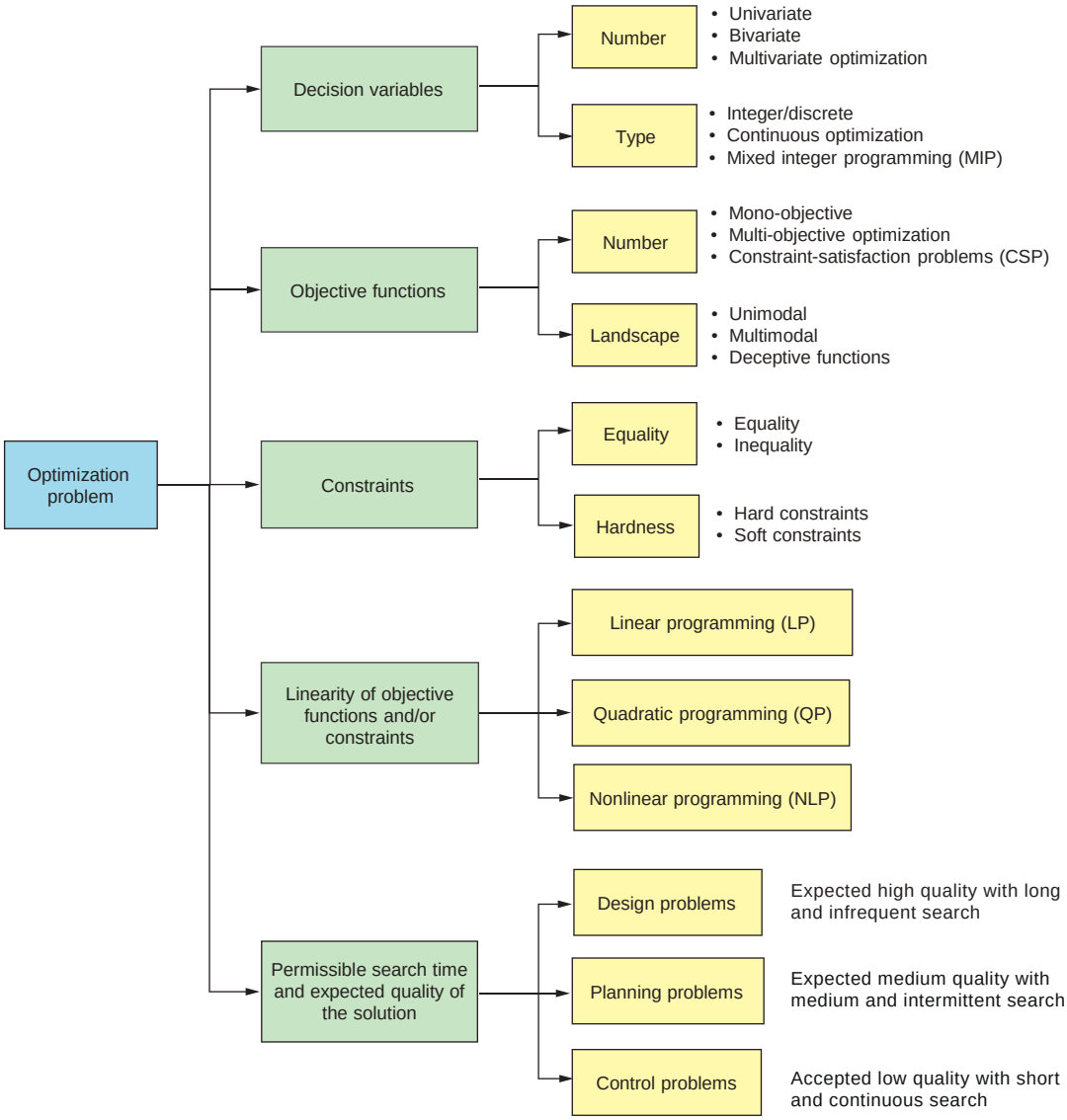



**Figure 2.1 Optimization problem classification—an optimization problem can be broken down into its constituent parts, which form the basis for classifying such problems.**

The following subsections explain these types in greater detail and provide examples of each type of optimization problem.

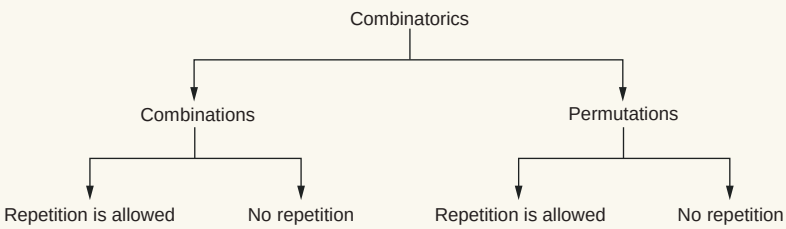
### 2.1.1 Number and type of decision variables

Based on the number of decision variables, optimization problems can be broadly grouped into univariate (single variable) or multivariate (multiple variable) problems. For example, vehicle speed, acceleration, and tire pressure are among the parameters that effect a vehicle's fuel economy, where fuel economy refers to how far a vehicle can travel on a specific amount of fuel. According to the US Department of Energy, controlling the speed and acceleration of a vehicle can improve its fuel economy by 15% to 30% at highway speeds and 10% to 40% in stop-and-go traffic. A study by the US National Highway Traffic Safety Administration (NHTSA) found that a 1% decrease in tire pressure correlated to a 0.3% reduction in fuel economy. If we are only looking for the optimal vehicle speed for maximum fuel economy, the problem is a univariate optimization problem. Finding the optimal speed and acceleration for maximum fuel economy is a bivariate optimization problem, whereas finding optimal speed, acceleration, and tire pressure is a multivariate problem.

Problem classification also varies according to the type of decision variables. A continuous problem involves continuous-valued variables, where $x_j \in \mathrm{R}$. In contrast, if $x_j \in \mathrm{Z}$, the problem is an integer or discrete optimization problem. A mixed-integer problem has both continuous-valued and integer-valued variables. For example, optimizing elevator speed and acceleration (continuous variables) and the sequence of picking up passengers (a discrete variable) is a mixed-integer problem. Problems where the solutions are sets, combinations, or permutations of integer-valued variables are referred to as combinatorial optimization problems.

#### Combination vs. permutation

Combinatorics is the branch of mathematics studying both the combination and permutation of a set of elements. The main difference between combination and permutation is the order. If the order of the elements doesn't matter, it is a combination, and if the order does matter, it is a permutation. Thus, permutations are ordered combinations. Depending on whether repetition of the elements is allowed or not, we can have different forms of combinations and permutations.




**Combinations and permutations—permutations respect order and are thus ordered combinations. Both combinations and permutations have variants with and without repetition.**

For example, assume we are designing a fitness plan that includes multiple fitness activities. Five types of exercises can be included in the fitness plan: jogging, swimming, biking, yoga, and aerobics. In a weekly plan, if we choose only three of these five exercises, and repetition is allowed, the number of possible combinations will be $(n + r - 1)! / r!(n - 1)!$ = (5 + 3 – 1)! / 3!(5 – 1)! = 7! / (3! × 4!) = 35. This means we can generate 35 different fitness plans by selecting three of the available five exercises and by allowing repetition.

However, if repetition is not allowed, the number of possible combinations will be $C(n,r) = n! / r!(n - r)!$ = 5! / (3! × 2!) = 10. This formula is often called "$n$ choose $r$" (such as "5 choose 3"), and it's also known as the *binomial coefficient*. This means that we can generate only 10 plans if we don't want to repeat any of the exercises.

In both combination with and without repetition, the fitness plan doesn't include the order of performing the included exercises. If we respect specific order, the plan will take the form of a permutation. If repeating exercises is allowed, the number of possible permutations when selecting three of the five available exercises will be $n^r = 5^3 = 125$. However, if repetition is not allowed, the number of possible permutations will be $P(n,r) = n! / (n - r)!$ = 5! / (5 – 3)! = 60.

Combinatorics can be implemented fairly easily in Python when coding from scratch, but there are excellent libraries available, such as SymPy, an open source Python library for symbolic mathematics. Its capabilities include, but are not limited to, statistics, physics, geometry, calculus, equation solving, combinatorics, discrete math, cryptography, and parsing. For example, the binomial coefficient can be calculated in SymPy using the following simple code:

```
from sympy import binomial
print(binomial(5,3))
```

See appendix A and the documentation for SymPy for more on implementing combinatorics in Python.

The traveling salesman problem (TSP) is a common example of a combinational problem whose solution is a permutation—a sequence of cities to be visited. In TSP, given $n$ cities, a traveling salesman must visit all the cities and then return home, making a loop (a round trip). The salesman would like to travel in the most efficient way (such as the fastest, cheapest, or shortest route).

TSP can be subdivided into *symmetric TSP* (STSP) and *asymmetric TSP* (ATSP). In STSP, the distance between two cities is the same in both directions, forming an undirected graph. This symmetry halves the number of possible solutions. ATSP is a strict generalization of the symmetric version. In ATSP, paths may not exist in both directions, or the distances might be different, forming a directed graph. Traffic collisions, one-way streets, bridges, and airfares for cities with different departure and arrival fees are examples of how this symmetry could break down.

The search space in TSP is very large. For example, let's assume the salesman is to visit the 13 major cities in the Greater Toronto Area (GTA), as illustrated in figure 2.2. The

naive solution's complexity is $O(n!)$. This means that there are $n! = 13! = 6{,}227{,}020{,}800$ possible tours in the case of ATSP. This is a huge search space in both STSP and ATSP. However, dynamic programming (DP) algorithms enable reduced complexity.


**Figure 2.2 TSP in the Greater Toronto Area (GTA). The traveling salesman must visit all 13 cities and wishes to select the "best" path, whether that be based on distance, time, or some other criterion.**

Dynamic programming is a method of solving optimization problems by breaking them down into smaller subproblems and solving each subproblem independently. For example, the complexity of the Bellman-Held-Karp algorithm [1] is $O(2^n \times n^2)$. There are other solvers and algorithms with different levels of computational complexity and approximation ratios such as the Concorde TSP solver, the 2-opt and 3-opt algorithms, branch and bound algorithms, the Christofides algorithm (or Christofides–Serdyukov algorithm), the Lin-Kernighan algorithm, metaheuristics-based algorithms, graph neural networks, and deep reinforcement learning methods. For example, the Christofides algorithm [2] is a polynomial-time approximation algorithm that produces a solution to TSP that is guaranteed to be no more than 50% longer than the optimal solution with a time complexity of $O(n^3)$. See appendix A for the solution of TSP using the Christofides algorithm implemented with the NetworkX package. We will discuss how to solve TSP using a number of these algorithms throughout this book.

A wide range of discrete optimization problems can be modeled as TSP. These problems include, but are not limited to, microchip manufacturing, permutation flow shop scheduling, arranging school bus routes for children in a school district, assigning routes for airplanes, transporting farming equipment, scheduling of service calls, meal delivery, and routing trucks for parcel delivery and pickup. For example, the

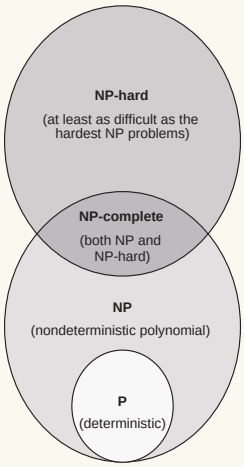
capacitated vehicle routing problem (CVRP) is a generalization of TSP where one has to serve a set of customers using a fleet of vehicles based at a common depot. Each customer has a certain demand for goods that are initially located at the depot. The task is to design vehicle routes starting and ending at the depot such that all customer demands are fulfilled. Later in this book, we'll look at several examples of solving TSP and its variants using stochastic approaches.

### Problem types

Decision problems are foundational in the study of algorithmic complexity. Generally speaking, a decision problem is a type of problem that requires determining whether a given input satisfies a certain property or condition. This problem can be answered with a simple "yes" or "no."

Decision problems are commonly classified based on their levels of complexity. These classes can also be applied to optimization problems, given that optimization problems can be converted into decision-making problems. For example, an optimization problem whose objective is to find an optimal or near-optimal solution within a feasible search space can be paraphrased as a decision-making problem that answers the question "Is there an optimal or a near-optimal solution within the feasible search space?" The answer will be "yes" or "no," or "true" or "false".

A generally accepted notion of an algorithm's efficiency is that its running time is polynomial. This means that the time or the computational cost to solve the problem can be described by a polynomial function of the size of the input for the algorithm. For example, in the context of TSP, the size of the input would typically be the number of cities that the salesperson needs to visit. Problems that can be solved in polynomial time are known as *tractable*. The following figure shows different types of problems and gives examples of commonly used benchmarks (toy problems) and real-life applications of each type.




| Problem class | Description | Examples of benchmarks (toy problems) | Examples of real-world problems |
|---|---|---|---|
| NP-hard | NP-hard problems may not have a solution verifiable in polynomial time or they may not even have a solution at all. They are at least as hard as the hardest problems in NP. | • Travelling salesman problem (TSP)<br>• Graph coloring<br>• Set covering and partitioning | • Vehicle routing<br>• Microchips manufacturing<br>• Mobile Radio Frequency Assignment<br>• Map Coloring<br>• Register Allocation<br>• Political/school/healthcare districting |
| NP-complete | NP-complete problems are NP-hard problems but the solutions can be verified in polynomial time. | • Bin packing problem (BBP)<br>• Knapsack problem<br>• Subset Sum Problem (SSP)<br>• Hamiltonian circuit<br>• Vertex cover problem<br>• *n*-puzzle problem<br>• Boolean satisfiability (SAT) | • Container/truck loading<br>• Telecommunication traffic overload control<br>• Design of queuing network models in manufacturing<br>• Combinational equivalence checking<br>• Job shop scheduling |
| NP/co-NP | Decision problems whose "yes" answer (or "no" in the case of co-NP) can be verified in polynomial time if we already have the proof (or witness). | • Integer Factorization<br>• Graph Isomorphism<br>• Maximum flow problem | • Image matching<br>• Direct or indirect interaction in social media<br>• Protein structure analysis |
| P | P problems are deterministic in nature and can be solved in polynomial time. | • Minimum spanning tree (MST) problem<br>• Shortest path problem | • Network designs<br>• Some single-machine scheduling problems |

**Problem classes based on hardness and completeness. Problems can be categorized into NP-hard, NP-complete, NP, or P.**

***(continued)***

For example, a complexity class P represents all decision problems that can be solved in polynomial time by deterministic algorithms (i.e., algorithms that do not guess at a solution). The NP or nondeterministic polynomial problems are those whose solutions are hard to find but easy to verify and are solved by a nondeterministic algorithm in polynomial time. NP-complete problems are those that are both NP-hard and verifiable in polynomial time. Finally, a problem is NP-hard if it is at least as hard as the hardest problem in NP-complete. NP-hard problems are usually solved by approximation or heuristic solvers, as it is hard to find efficient exact algorithms to solve such problems.

Clustering is a type of combinatorial problem whose solution takes the form of a combination where the order doesn't matter. In clustering, given $n$ objects, we need to group them in $k$ groups (clusters) such that all objects in a single group or cluster have a "natural" relation to one another, and objects not in the same group are somehow different. This means that the objects will be grouped based on some similarity or dissimilarity metric.

*Stirling numbers* can be used for counting partitions and permutations in combinatorial problems. Stirling numbers of the *first kind* count permutations according to their number of cycles, while Stirling numbers of the *second kind* represent the number of ways we can partition a set of objects into non-empty subsets. The following formula is for a Stirling number of the second kind (a *Stirling partition number*), and it gives the number of ways you can partition a set of $n$ objects into $k$ non-empty subsets in the context of our clustering problem:

$$S(n,k) = \left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n \qquad \textbf{2.1}$$

Let's consider smart cart clustering as an example. Shopping and luggage carts are commonly found in shopping malls and large airports. Shoppers or travelers pick up these carts at designated points and leave them in arbitrary places. It is a considerable task to re-collect them, and it is therefore beneficial if a "smarter" version of these carts could draw themselves together automatically to the nearest assembly points, as illustrated in figure 2.3.

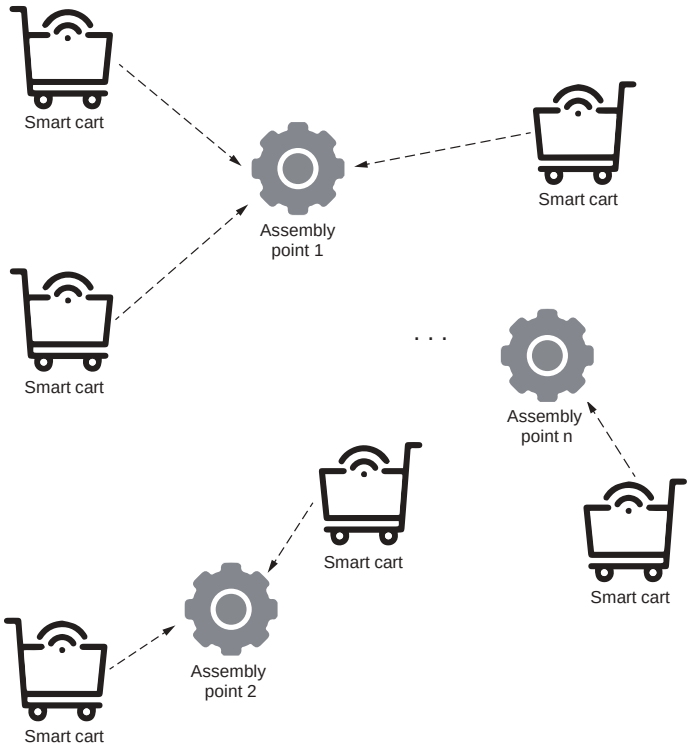



**Figure 2.3 Smart cart clustering. Unused shopping or luggage carts congregate near designated assembly points to make collection and redistribution easier.**

In practice, this problem is considered an NP-hard problem, as the search space can be very large based on the numbers of available carts and assembly points. To cluster these carts effectively, the centers of clustering (the *centroids*) must be found. The carts in each cluster will then be directed to the assembly point closest to the centroids.

For example, assume that 50 carts are to be clustered around four assembly points. This means that $n = 50$ and $k = 4$. Stirling numbers can be generated using the SymPy library. To do so, simply call the `stirling` function on two numbers, $n$ and $k$:

```
from sympy.functions.combinatorial.numbers import stirling
print(stirling(50,4))
print(stirling(100,4))
```

The result is $5.3 \times 10^{28}$, and if $n$ is increased to 100, the number becomes $6.7 \times 10^{58}$. Enumerating all possible partitions for large problems is not feasible.

### 2.1.2 Landscape and number of objective functions

An objective function's *landscape* represents the distribution of the function's values in the feasible search space. In this landscape, you'll find the optimal solution or the global minima in the lowest valley, assuming you are dealing with a minimization problem, or at the highest peak in the case of a maximization problem. According to the landscape of the objective function, if there is only one clear global optimal solution, the problem is *unimodal* (e.g., convex and concave functions). In a *multimodal* problem, more than one optimum exists. The objective function is called *deceptive* when the global minimum lies in a very narrow valley and there is also a strong local minimum with a wide basin of attraction, such that the value of this objective function is close to the value of an objective function at the global minimum [3]. Figure 2.4 is a 3D visualization of the landscapes of unimodal, multimodal, and deceptive functions generated using Python in the next listing. The complete listing is available in the GitHub repo for the book.

**Listing 2.1 Examples of objective functions**

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

def objective_unimodal(x, y):                          <-- Unimodal function
    return x**2.0 + y**2.0

def objective_multimodal(x, y):                        <-- Multimodal function
    return np.sin(x) * np.cos(y)

def objective_deceptive(x, y):                         <-- Deceptive function
    return (1-(abs((np.sin(math.pi*(x-2))*np.sin(math.pi*(y-2)))/
➥   (math.pi*math.pi*(x-2)*(y-2))))**5)*(2+(x-7)**2+2*(y-7)**2)

fig = plt.figure(figsize = (25,25))
ax = fig.add_subplot(1,3,1, projection='3d')

x = np.arange(-3, 3, 0.01)
y = np.arange(-3, 3, 0.01)

X, Y = np.meshgrid(x, y)
Z = objective_unimodal(X, Y)
surf = ax.plot_surface(X, Y, Z, cmap=plt.cm.cividis)
ax.set_xlabel('x', fontsize=15)
ax.set_ylabel('y', fontsize=15)
ax.set_zlabel('Z', fontsize=15)
ax.set_title("Unimodal/Convex function", fontsize=18)
```

```
ax = fig.add_subplot(1,3,2, projection='3d')
Z = objective_multimodal(X, Y)
surf = ax.plot_surface(X, Y, Z, cmap=plt.cm.cividis)
ax.set_xlabel('x', fontsize=15)
ax.set_ylabel('y', fontsize=15)
ax.set_zlabel('Z', fontsize=15)
ax.set_title("Multimodal function", fontsize=18)

X, Y = np.meshgrid(x, y)
Z = objective_unimodal(X, Y)
ax = fig.add_subplot(1,3,3, projection='3d')
Z = objective_deceptive(X, Y)
surf = ax.plot_surface(X, Y, Z, cmap=plt.cm.cividis, antialiased=False)
ax.set_xlabel('x', fontsize=15)
ax.set_ylabel('y', fontsize=15)
ax.set_zlabel('Z', fontsize=15)
ax.set_title("Deceptive function", fontsize=18)

plt.show()
```

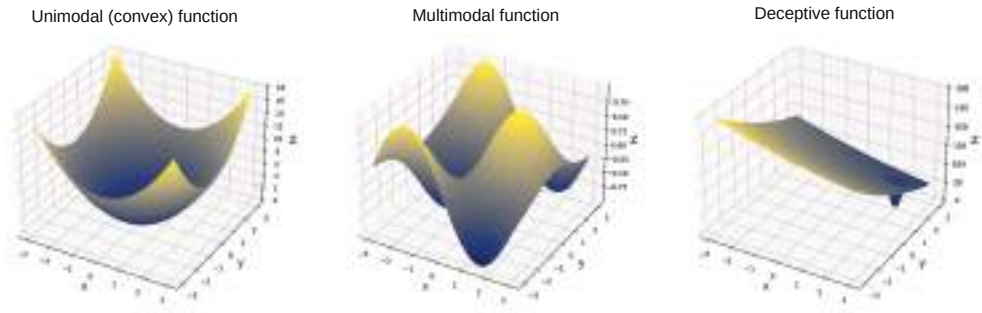



**Figure 2.4 Unimodal, multimodal, and deceptive functions. Unimodal functions have one global optimum, whereas multimodal functions can have many. Deceptive functions contain false optima close to the value of an objective function at a global minimum, which can cause some algorithms to get stuck.**

If the quantity to be optimized is expressed using only one objective function, the problem is referred to as a mono-objective or single-objective optimization problem (such as convex or concave functions). A multi-objective optimization problem specifies multiple objectives to be simultaneously optimized. Problems without an explicit objective function are called constraint-satisfaction problems (CSPs). The goal in this case is to find a solution that satisfies a given set of constraints.

The *n*-queen problem is an example of a CSP. In this problem, the aim is to put $n$ queens on an $n \times n$ board with no two queens on the same row, column, or diagonal, as illustrated in figure 2.5. In this 4-queen problem, there are 5 conflicts in the first state ({Q1,Q2}, {Q1,Q3}, {Q2,Q3}, {Q2,Q4}, and {Q3,Q4}). After moving Q4, the number of conflicts reduces by 2, and after moving Q3, the number of conflicts is only 1, which is between Q1 and Q2.

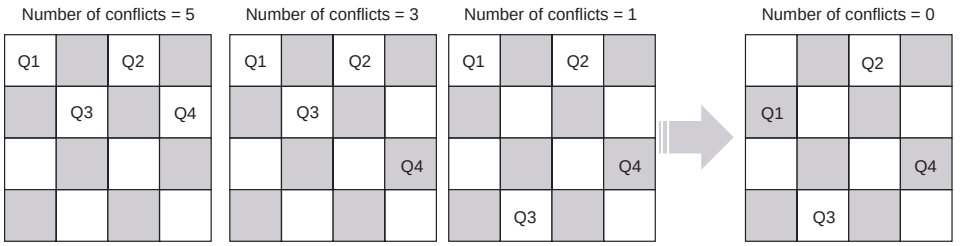



**Figure 2.5 The *n*-queen problem. This problem has no objective function, only a set of constraints that must be satisfied.**

If we keep moving or placing the pieces, we can reach the goal state where the number of conflicts is 0, which means that no queen could attack any other queen horizontally, vertically, or diagonally. The next listing is a Python implementation of the 4-queen problem.

**Listing 2.2 *n*-queen CSP**

```
from copy import deepcopy
import math
import matplotlib.pyplot as plt
import numpy as np

board_size = 4
board = np.full((board_size, board_size), False)

def can_attack(board, row, col):
    if any(board[row]):
        return True

    offset = col - row
    if any(np.diagonal(board, offset)):
        return True
    offset = (len(board) - 1 - col) - row
    if any(np.diagonal(np.fliplr(board), offset)):
        return True

    return False

board[0][0] = True
col = 1
states = [deepcopy(board)]
while col < board_size:
    row = 0
    while row < board_size:
        if not can_attack(board, row, col):
            board[row][col] = True
            col += 1
            states.append(deepcopy(board))
```

**Create an n x n board.**

**Check for a queen on the same row.**

**Check for queens on the diagonals.**

**The piece can be placed in this column.**

```
        break
    row += 1
    if row == board_size:
        board = np.delete(board, 0, 1)
        new_col = [[False]] * board_size
        board = np.append(board, new_col, 1)
        states.append(deepcopy(board))
        col -= 1
        continue
```

The piece cannot be placed in this column.

In the preceding listing, the `can_attack` function detects if a newly placed piece can attack a previously placed piece. A piece can attack another piece if it is in the same row, column, or diagonal. Figure 2.6 shows the solution obtained after six steps.

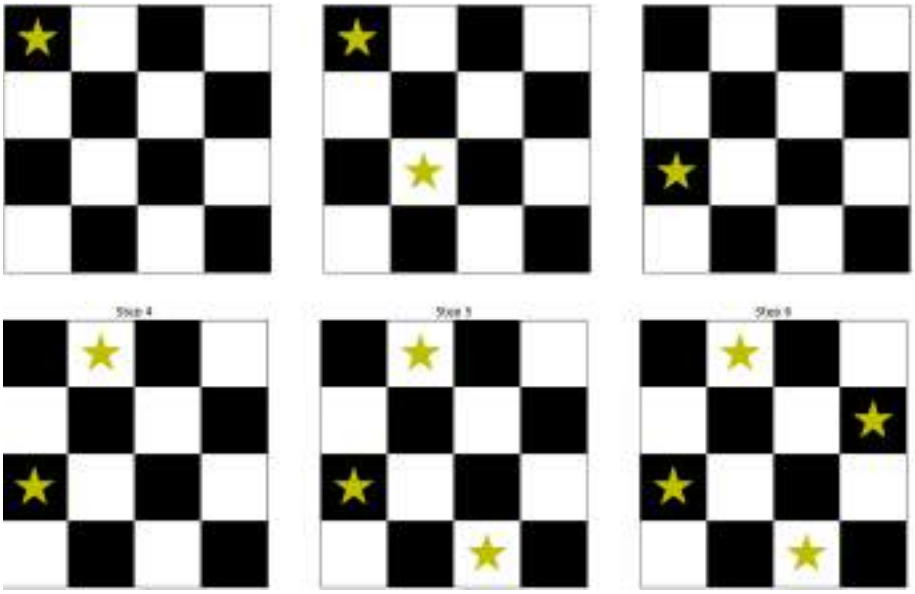


**Figure 2.6** *n*-queen solution

The first piece is trivially placed in the first position. The second piece must be placed either in the third or fourth position, as the first two can be attacked. By placing it in the third position, however, the third piece cannot be placed. Thus, the first piece is removed (the board is "slid" one column over), and we try again. This continues until a solution is found.

The full code for this problem, including the code used to generate visualizations, can be found in the code file for listing 2.2, available in the book's GitHub repo. The solution algorithm is as follows:

1. Moving from top to bottom in a column, the algorithm attempts to place the piece while avoiding conflicts. For the first column, this will default to Q1 = 0.
2. Moving to the next column, if a piece cannot be placed at row 0, it will be placed at row 1, and so on.

3 When a piece has been placed, the algorithm moves to the next column.
4 If it is impossible to place a piece in a given column, the first column of the entire board is removed, and the current column is reattempted.

Constraint programming solvers available in Google OR-Tools can also be used to solve this $n \times n$ queen problem. The next listing shows the steps of the solution using OR-Tools.

**Listing 2.3 Solving the $n$-queen problem using OR-Tools**

```
import numpy as np
import matplotlib.pyplot as plt
import math
from ortools.sat.python import cp_model

board_size = 4

model = cp_model.CpModel()

queens = [model.NewIntVar(0, board_size - 1, 'x%i' % i)
➥for i in range(board_size)]

model.AddAllDifferent(queens)

model.AddAllDifferent(queens[i] + i for i in range(board_size))
model.AddAllDifferent(queens[i] - i for i in range(board_size))

solver = cp_model.CpSolver()
solver.parameters.enumerate_all_solutions = True
solver.Solve(model)


all_queens = range(board_size)
state=[]
for i in all_queens:
    for j in all_queens:
        if solver.Value(queens[j]) == i:
            # There is a queen in column j, row i.
            state.append(True)
        else:
            state.append(None)

states=np.array(state).reshape(-1, board_size)
fig = plt.figure(figsize=(5,5))
markers = [
    x.tolist().index(True) if True in x.tolist() else None
    for x in np.transpose(states)
]
res = np.add.outer(range(board_size), range(board_size)) % 2
plt.imshow(res, cmap="binary_r")
plt.xticks([])
plt.yticks([])
plt.plot(markers, marker="*", linestyle="None",
➥markersize=100/board_size, color="y")H
```

**Import a constraint programming solver that uses SAT (satisfiability) methods.**

**Set the board size for the n x n queen problem.**

**Define a solver.**

**Define the variables. The array index represents the column, and the value is the row.**

**Define the constraint: all rows must be different.**

**Solve the model.**

**Define the constraint: no two queens can be on the same diagonal.**

**Visualize the solution.**

Running this code produces the output in figure 2.7. More information about Google OR-Tools is available in appendix A.

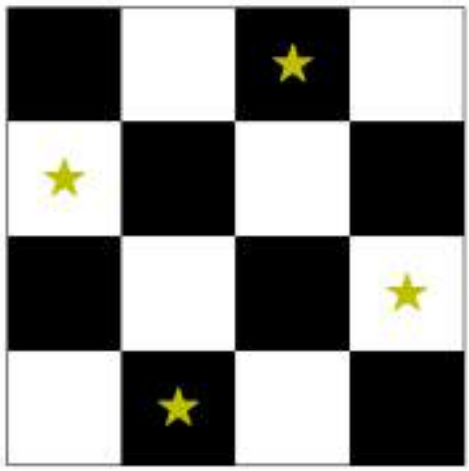


**Figure 2.7 The *n*-queen solution using OR-Tools**

### 2.1.3 Constraints

Constrained problems have hard or soft constraints for equality, inequality, or both. Hard constraints must be satisfied, while soft constraints are nice to satisfy (but are not mandatory). If there are no constraints to be considered, aside from the boundary constraints, the problem is an unconstrained optimization problem.

Let's revisit the ticket pricing problem introduced in section 1.3.1. There is a wide range of derivative-based solvers in Python that can handle such kinds of differentiable mathematical optimization problems (see appendix A). The next listing shows how you can solve this simple ticket pricing problem using SciPy. SciPy is a library containing valuable tools for all things computation.

**Listing 2.4 Optimal ticket pricing**

```
import numpy as np
import scipy.optimize as opt
import matplotlib.pyplot as plt

def f(x):
    return -(-20*x**2+6200*x-350000)/1000

res=opt.minimize_scalar(f, method='bounded', bounds=[0, 250])

print("Optimal Ticket Price ($): %.2f" % res.x)
print("Profit f(x) in K$: %.2f" % -res.fun)
```

**The objective function, required by minimize_scalar to be a minimization function**

**The bounded method is the constrained minimization procedure that finds the solution.**

Running this code produces the following output:

```
Optimal Ticket Price ($): 155.00
Profit f(x) in K$: 130.50
```

This code finds the optimal ticket price in the range between $0 and $250 that maximizes the profit. As you may have noticed, the profit formula is converted into a minimization problem by adding a negative sign in the objective function to match with the `minimize` function in `scipy.optimize`. A minus sign is added in the `print` function to convert it back into profit.

What if we imposed an equality constraint on this problem? Let's assume that due to incredible international demand for our event, we are now considering using a different event planning company and opening up virtual attendance for our conference so that international guests can also participate. Interested participants can now choose between attending the event in person or joining via a live stream. All participants, whether in-person or virtual, will receive a physical welcome package, which is limited to 10,000 units. Thus, in order to ensure a "full" event, we must either sell 10,000 in-person tickets, 10,000 virtual tickets, or some combination thereof. The new event company is charging us a $1,000,000 flat rate for the event, so we want to sell as many tickets as possible (exactly 10,000). The following equation is associated with this problem:

Let $x$ be the number of physical ticket sales, and let $y$ be the number of virtual ticket sales. Additionally, let $f(x,y)$ be the function for profits generated from the event, where

$$f(x, y) = 155x + \left(0.001x^{\frac{3}{2}} + 70\right)y - 1000000 \qquad \textbf{2.2}$$

Essentially, we earn $155 profit on in-person attendance, and the profit for online attendance is $70, but it increases by some amount with the more physical attendance we have (let's say that as the event looks "more crowded," we can charge more for online attendees).

Suppose we add a constraint function, $x + y \leq 10000$, which shows that the combined ticket sales cannot exceed 10,000. The problem is now a bivariate mono-objective constrained optimization problem. It is possible to convert this constrained optimization problem to an unconstrained optimization using the Lagrange multiplier, $\lambda$. We can use SymPy to implement Lagrange multipliers and solve for the optimal mix of virtual and physical ticket sales. The idea is to convert the constrained optimization problem defined by the objective function $f(x,y)$ with an equality constraint $g(x,y)$ into an unconstrained optimization problem using the Lagrangian function $L(x,y,\lambda) = f(x,y) + \lambda g(x,y)$. This function combines an objective function and constraints, enabling constrained optimization problems to be formulated as unconstrained problems through the use of Lagrange multipliers. To do so, we take the partial derivatives of the objective functions and the constraints, with respect to the decision variables $x$ and $y$, to form the unconstrained optimization equations to be used by the SymPy solver, as illustrated in figure 2.8.

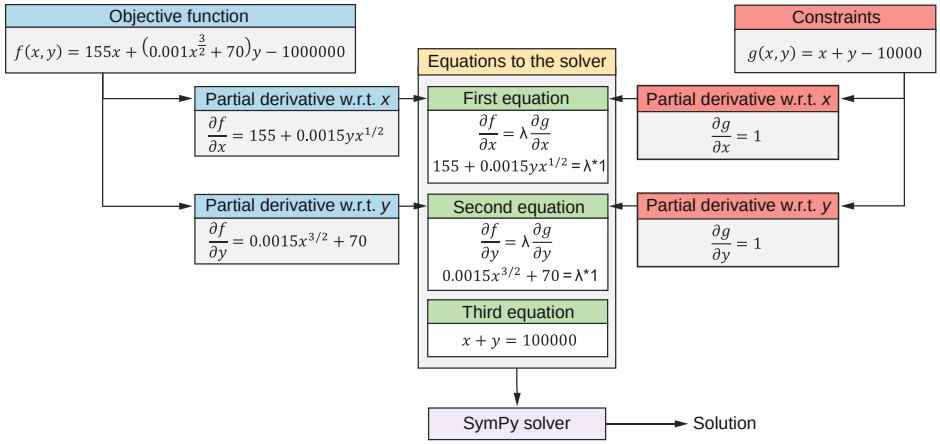



**Figure 2.8 Steps for solving the ticket pricing problem using the Lagrange method**

The next listing shows the Python implementation using SymPy.

**Listing 2.5 Maximizing profits using Lagrange multipliers**

```
import sympy as sym

x,y=sym.var('x, y', positive=True)

f=155*x+(0.001*x**sym.Rational(3,2)+70)*y-1000000

g=x+y-10000

lamda=sym.symbols('lambda')
Lagr=f-lamda*g

eqs = [sym.diff(Lagr, x), sym.diff(Lagr, y), g]

sol=sym.solve(eqs,[x,y,lamda], dict=True)

def getValueOf(k, L):
    for d in L:
        if k in d:
            return d[k]

profit=[f.subs(p) for p in sol]

print("optimal number of physical ticket sales: x = %.0f" % getValueOf(x, sol))
print("optimal number of online ticket sales: y = %.0f" % getValueOf(y, sol))
print("Expected profil: f(x,y) = $%.4f" % profit[0])
```

**Define the decision variables.**

**Define the ticket pricing objective function.**

**Define the equality constraint.**

**Lagrange multiplier**

**Lagrangian function**

**Equations to the solver**

**Solve these three equations in three variables (x,y,lambda) using SymPy.**

By solving the preceding three equations, we get $x$ and $y$ values that correspond to the optimized quantities for virtual and physical ticket sales. With the code in listing 2.5, we can see that the best result is to sell 6,424 in-person tickets and 3,576 online tickets. This results in a maximum profit of $2,087,260.

### 2.1.4 Linearity of objective functions and constraints

If all the objective functions and associated constraint conditions are linear, the optimization problem is categorized as a *linear optimization problem* or *linear programming problem* (LPP or LP), where the goal is to find the optimal value of a linear function subject to linear constraints. Blending problems are a typical application of mixed integer linear programming (MILP), where a number of ingredients are to be blended or mixed to obtain a product with certain characteristics or properties. In the animal feed mix problem described in Paul Jensen's *Operations Research Models and Methods* [4], the optimum amounts of three ingredients in an animal feed mix need to be determined. The possible ingredients, their nutritive contents (in kilograms of nutrient per kilograms of ingredient), and the unit costs are shown in table 2.1.

**Table 2.1 Animal feed mix problem**

| Ingredients | Nutritive content and price of ingredients | | | |
|---|---|---|---|---|
| | Calcium (kg/kg) | Protein (kg/kg) | Fiber (kg/kg) | Unit cost (cents/kg) |
| Corn | 0.001 | 0.09 | 0.02 | 30.5 |
| Limestone | 0.38 | 0.0 | 0.0 | 10.0 |
| Soybean meal | 0.002 | 0.50 | 0.08 | 90.0 |

The mixture must meet the following restrictions:

- Calcium—At least 0.8% but not more than 1.2%
- Protein—At least 22%
- Fiber—At most 5%

The problem is to find the mixture that satisfies these constraints while minimizing cost. The decision variables are $x_1$, $x_2$, and $x_3$, which are proportions of limestone, corn, and soybean meal respectively.

The objective function $f = 30.5x_1 + 10x_2 + 90x_3$ needs to be minimized, subject to the following constraints:

- Calcium limits: $0.008 \le 0.001x_1 + 0.38x_2 + 0.002x_3 \le 0.012$
- Protein constraint: $0.09x_1 + 0.5x_3 \ge 0.22$
- Fiber constraint: $0.02x_1 + 0.08x_3 <= 0.05$
- Non-negativity restriction: $x_1, x_2, x_3 \ge 0$
- Conservation: $x_1 + x_2 + x_3 = 1$

In this problem, both the objective function and the constraints are linear, so it is an LPP. There are several Python libraries that can be used to solve mathematical optimization problems.

We'll try solving the animal feed mix problem using PuLP. PuLP is a Python linear programming library that allows users to define linear programming problems and solve them using optimization algorithms such as COIN-OR's linear and integer programming solvers. See appendix A for more information about PuLP and other mathematical programming solvers. The next listing shows the steps for solving the animal feed mix problem using PuLP.

**Listing 2.6 Solving a linear programming problem using PuLP**

```
from pulp import *

model = LpProblem("Animal_Feed_Mix_Problem", LpMinimize)

x1 = LpVariable('Corn', lowBound = 0, upBound = 1, cat='Continous')
x2 = LpVariable('Limestone', lowBound = 0, upBound = 1, cat='Continous')
x3 = LpVariable('Soybean meal', lowBound = 0, upBound = 1, cat='Continous')

model += 30.5*x1 + 10.0*x2 + 90*x3, 'Cost'

model +=0.008 <= 0.001*x1 + 0.38*x2 + 0.002*x3 <= 0.012, 'Calcium limits'
model += 0.09*x1 + 0.5*x3 >=0.22, 'Minimum protein'
model += 0.02*x1 + 0.08*x3 <=0.05, 'Maximum fiber'
model += x1+x2+x3 == 1, 'Conservation'

model.solve()

for v in model.variables():
    print(v.name, '=', round(v.varValue,2)*100, '%')

print('Total cost of the mixture per kg = ',
➥    round(value(model.objective)/100, 2), '$')
```

**Create a linear programming model.**

**Define three variables that represent the percentages of corn, limestone, and soybean meal in the mixture.**

**Define the total cost as the objective function to be minimized.**

**Add the constraints.**

**Solve the problem using PuLP's choice of solver.**

**Print the results (the optimal percentages of the ingredients and the cost of the mixture per kg).**

As you can see in this listing, we start by importing PuLP and creating a model as a linear programming problem. We then define LP variables with the associated parameters, such as name, lower bound, and upper bound for each variable's range and the type of variable (e.g., integer, binary, or continuous). A solver is then used to solve the problem. PuLP supports several solvers, such as GLPK, GUROBI, CPLEX, and MOSEK. The default solver in PuLP is Cbc (COIN-OR branch and cut). Running this code gives the following output:

```
Corn = 65.0%
Limestone = 3.0%
Soybean_meal = 32.0%
Total cost of the mixture per kg = 0.4916$
```

If one of the objective functions, or at least one of the constraints, is nonlinear, the problem is considered a nonlinear optimization problem or nonlinear programming problem (NLP), and it's harder to solve than a linear problem. A special case of NLP, when the objective function is quadratic, is called quadratic programming (QP). For

example, the plant layout problem (PLP) or facility location problem (FLP) is a quadratic assignment problem (QAP) that aims at assigning different facilities (departments) $F$ to different locations $L$ in order to minimize a given function cost, such as the total material handling cost, as shown in figure 2.9.

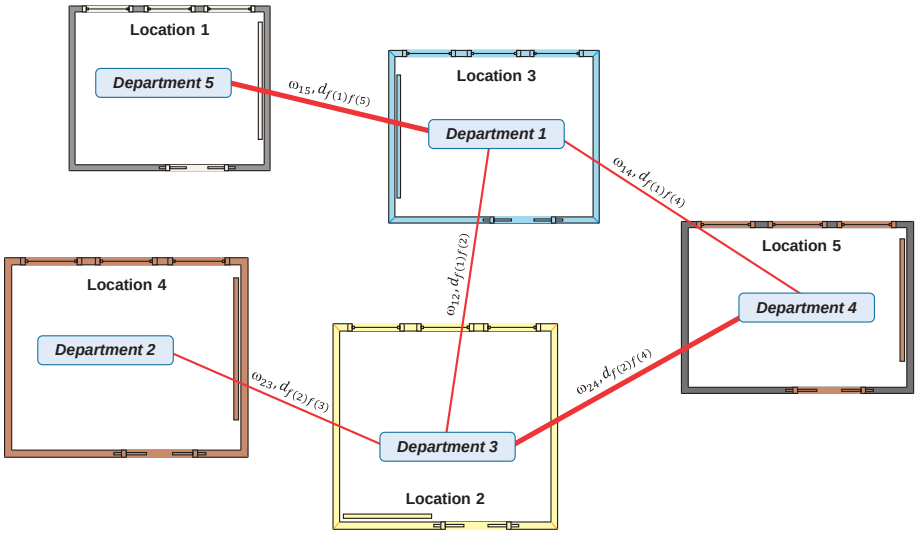



**Figure 2.9 Plant layout problem—what is the optimal location for each department that minimizes the overall material handling costs?**

Assume that $\omega_{ij}$ is the frequency of interaction or the flow of products between these facilities and $d_{f(i)f(j)}$ is the distance between facilities $i$ and $j$. The material handling cost (MHC) is

$$\text{MHC}_{ij} = \text{flow} \times \text{distance} = \omega_{ij} \times d_{f(i)f(j)} \qquad \textbf{2.3}$$

and the total material handling cost (TMHC) is the summation of all the material handling costs inside the material handling cost matrix. In matrix notation, the problem can be formulated as

$$\text{Find } X \text{ which minimizes } trace(WXDX^T)$$

where $X$ represents the assignment vector, $W$ is the flow matrix, and $D$ is the distance matrix. Trace is the sum of elements on the main diagonal (from the upper left to the lower right) of the resultant material handling cost matrix.

In a more general case, NLP includes nonlinear objective functions, or at least nonlinear constraints, of any form. For example, imagine you're designing a landmine

detection and disposal unmanned ground vehicle (UGV) [5]. In outdoor applications like humanitarian demining, UGVs should be able to navigate through rough terrain. Sandy soils, rocky terrain with obstacles, steep inclines, ditches, and culverts can be difficult for vehicles to negotiate. The locomotion systems of such vehicles need to carefully designed to guarantee motion fluidity.

Assume that you are in charge of finding optimal values for wheel parameters (e.g., diameter, width, and loading) that will

- Minimize the wheel sinkage, which is the maximum amount the wheel sinks in the soil that it is moving on
- Minimize motion resistance, which is the overall resistance faced by the UGV unit due to the different components of resistance (compaction, gravitational, etc.)
- Minimize drive torque, which is the driving torque required from the actuating motors for each wheel
- Minimize drive power, which is the driving power required from the actuating motors for each wheel
- Maximize the slope negotiability, which represents the maximum slope that can be climbed by the UGV unit considering its weight and the soil parameters.

Due to availability in the market or manufacturing concerns and costs, the wheel diameter should be in the range of 4 to 8.2 inches, wheel width should be in the range of 3 to 5 inches, and wheel loading should be in the range of 22 to 24 pounds per wheel. This wheel design problem (figure 2.10) can be stated as follows:

Find $X$ which optimizes $f$, subject to a possible set of boundary constraints, where $X$ is a vector that is composed of a number of decision variables such as

- $x_1$ = wheel diameter, $x_1 \in [4, 8.2]$
- $x_2$ = wheel width, $x_2 \in [3, 5]$
- $x_3$ = wheel loading, $x_3 \in [22, 24]$

We can also consider the objective functions $f=\{f_1, f_2,\ldots\}$. For example, the function for wheel sinkage might look like this:

$$f_1 = \left( \frac{3x_3}{(3-n)(k_c + x_2 k_\phi \sqrt{x_1})} \right)^{\frac{2}{2n+1}} \qquad \textbf{2.4}$$

where $n$ is the exponent of sinkage, $k_c$ is the cohesive modulus of soil deformation, and $k_\varphi$ is the frictional modulus of soil deformation. This problem is considered to be nonlinear because the objective function is nonlinear.

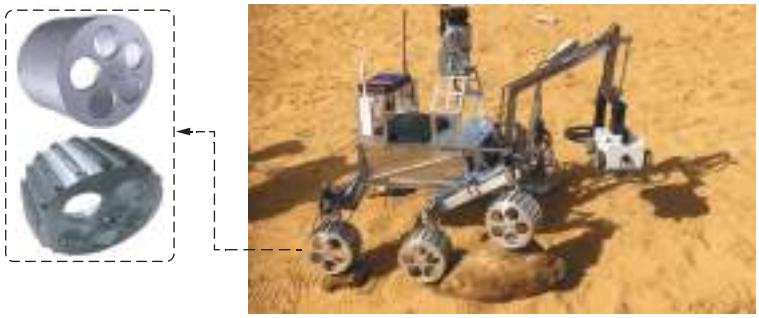


**Figure 2.10 The MineProbe wheel design problem [5]**

The catenary problem discussed in Veselić's "Finite catenary and the method of Lagrange" article [6] is another example of a nonlinear optimization problem. A catenary is a flexible hanging object composed of multiple parts, such as a chain or telephone cable (figure 2.11). In this problem, we are provided with $n$ homogenous beams, with lengths $d_1, d_2, \ldots d_n > 0$ and masses $m_1, m_2, \ldots m_n > 0$, which are connected by $n + 1$ joints $G_0, G_2, \ldots G_{n+1}$. The location of each joint is represented by the Cartesian coordinates $(x_i, y_i, z_i)$. The ends of the catenary are $G_0$ and $G_{n+1}$, which both have the same $y$ and $z$ values (they are at the same height and in line with each other).

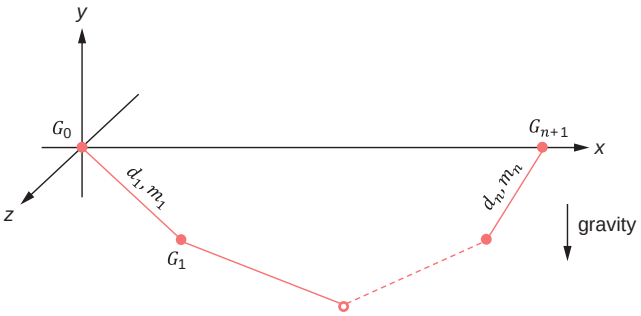



**Figure 2.11 Finite catenary problem—the catenary (or chain) is suspended from two points, $G_0$ and $G_{n+1}$.**

Assuming that the beam lengths and masses are predefined parameters, our goal is to look for stable equilibrium positions in the field of gravity—those positions where the potential energy is minimized. The potential energy to be minimized is defined as follows:

$$E = \sum_{i=1}^{n+1} m_i \gamma \frac{y_i + y_{i-1}}{2}, \gamma > 0 \qquad \textbf{2.5}$$

subject to the following constraints:

$$g_i = (x_i - x_{i-1})^2 + (y_i - y_{i-1})^2 + (z_i - z_{i-1})^2 - d_i^2 = 0, i = 0, 2, \ldots, n \quad \textbf{2.6}$$

where $\gamma$ is the gravitational constant. The nonlinearity of the constraints makes this problem nonlinear, despite having a linear objective function.

### 2.1.5 Expected quality and permissible time for the solution

Optimization problems can also be categorized according to the expected quality of the solutions and the time allowed to find the solutions. Figure 2.12 shows three main types of problems: design problems (strategic functions), planning problems (tactical functions), and control problems (operational functions).

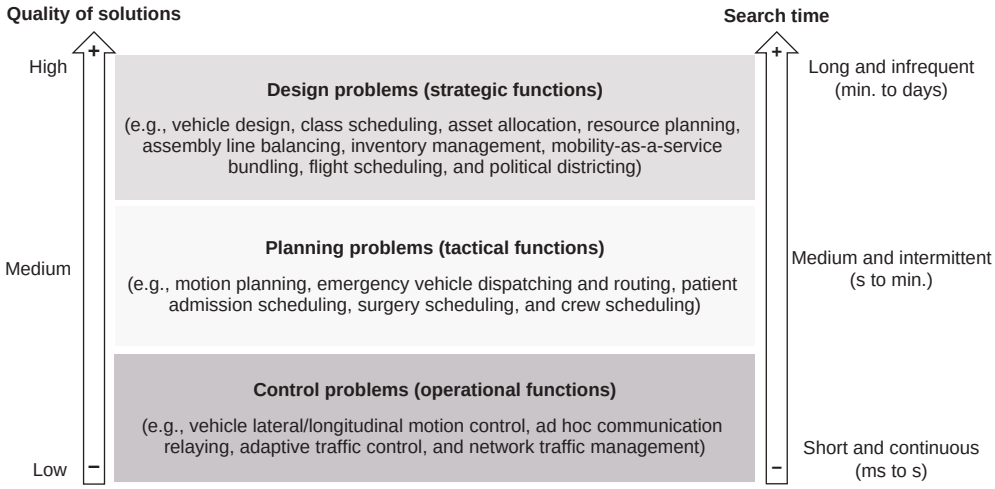



**Figure 2.12 Qualities of solutions vs. search time. Some types of problems require fast computations but do not require incredibly accurate results, while others (such as design problems) allow more processing time in return for higher accuracy.**

In *design problems*, time is not as important as the quality of the solution, and users are willing to wait (sometimes even a few days) to get an optimal, or near-optimal, result. These problems can be solved offline, and the optimization process is usually carried out only once in a long time. Examples of design problems include vehicle design, class scheduling, asset allocation, resource planning, assembly line balancing, inventory management, flight scheduling, and political districting.

Let's discuss political districting as a design problem in more detail. Districting is the problem of grouping small geographic areas, called *basic units*, into larger geographic clusters, called *districts*, in such a way that the latter are acceptable according to relevant planning criteria [7]. Typical examples of basic units are customers, streets, or zip code areas. The planning criteria may include the following:

- Balance or equity in terms of demographic background, equitable size, balanced workload, equal sales potential, or the number of customers
- Contiguity to enable traveling between the basic units of the district without having to leave the district
- Compactness to allow for round- or square-shaped undistorted districts without holes
- Respect of boundaries, such as administrative boundaries, railroads, rivers, or mountains
- Socio-economic heterogeneity, to allow for better representation of residents with different incomes, ethnicities, concerns, or views

Political districting, school districting, districting for health services, districting for EV charging stations, districting for micro-mobility stations (e.g., for e-bikes and e-scooters), and districting for sales or delivery are all examples of districting problems.

Political districting is a problem that has plagued societies since the advent of representative democracy in the Roman Republic. In a representative democracy, officials are nominated and elected to represent the interests of the people who elected them. In order to have a greater say when deciding on matters that concern the entire state, the party system came about, which defines political platforms that nominees use to differentiate themselves from their competitors. Manipulating the shapes of electoral districts to determine the outcome of elections is called *gerrymandering* (named after the early nineteenth century Massachusetts governor Elbridge Gerry who redrew the map of the Senate's districts in 1810 in order to weaken the opposing federalist party). Figure 2.13 shows how manipulating the shapes of the districts can sway the vote in favor of a decision that otherwise wouldn't have won.

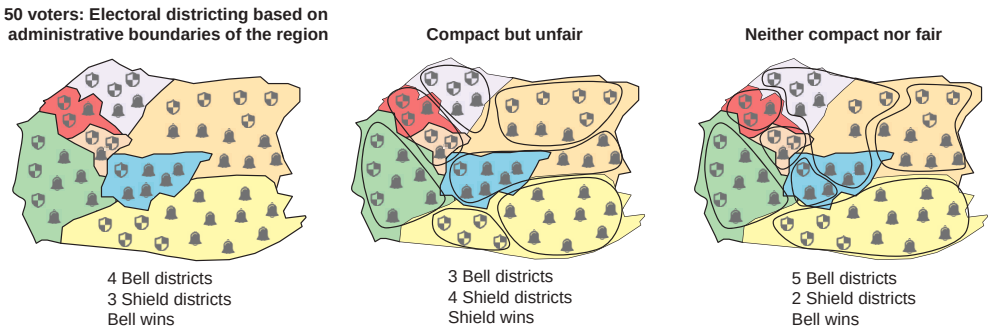



**Figure 2.13 Example of gerrymandering. The two major political parties, Shield and Bell, try to gain an advantage by manipulating the district boundaries to suppress undesired interests and promote their own.**

An effective and transparent political districting strategy is needed to avoid gerrymandering and generate a solution that preserves the integrity of individual subdistricts and divides the population into almost equal voting populations in a reproducible way.

In many countries, electoral districts are reviewed from time to time to reflect changes and movements in the country's population. For example, the Constitution of Canada requires that federal electoral districts be reviewed after each 10-year census.

Political districting is defined as aggregating $n$ subregions of a territory into $m$ electoral districts subject to constraints such as

- The districts should have near-equal voting population.
- The socioeconomic homogeneity inside each district, as well as the integrity of different communities, should be maximized.
- The districts have to be compact, and the subregions of each district have to be contiguous.
- Subregions should be considered as indivisible political units, and their boundaries should be respected.

The problem can be formulated as an optimization problem in which a function that quantifies the preceding factors is maximized. Here is an example of this function:

$$F(x) = \alpha_{\text{pop}} f_{\text{pop}}(x) + \alpha_{\text{comp}} f_{\text{comp}}(x) + \alpha_{\text{soc}} f_{\text{soc}}(x) + \alpha_{\text{sim}} f_{\text{sim}}(x) \qquad \textbf{2.7}$$

where $x$ is a solution to the problem or the electoral districts, $\alpha_i$ are user-specified multipliers $0 \le \alpha_i \le 1$, and $f_{\text{pop}}$, $f_{\text{comp}}$, $f_{\text{soc}}$, $f_{\text{int}}$, and $f_{\text{sim}}$ are functions that quantify the population equality, compactness of districts, socioeconomic homogeneity, integrity of different communities, and similarity to existing districts respectively. In the upcoming chapters, I will show you how we can use offline optimization algorithms to handle optimal multicriteria assignment design problems.

*Planning problems* need to be solved faster than design problems, in a time span from a few seconds to a few minutes. To find a solution in such a short time, optimality is usually traded for speed. Examples of planning problems include vehicle motion planning, emergency vehicle dispatching and routing, patient admission scheduling, surgery scheduling, and crew scheduling. Let's consider the ride-sharing problem as an example of a planning problem.

Ride-sharing involves a fleet of pay-per-use vehicles and a set of passengers with predefined pick-up and drop-off points (figure 2.14). The dispatch service needs to assign a set of passengers in a specific order to each driver to achieve a set of objectives. This ride-sharing problem is a multi-objective constrained optimization problem. A noncomprehensive list of optimization goals for ride-sharing includes

- Minimizing the total travel distance or time of drivers' trips
- Minimizing the total travel time of passengers' trips
- Maximizing the number of matched (served) requests
- Minimizing the cost of the drivers' trips
- Minimizing the cost of the passengers' trips

- Maximizing the drivers' earnings
- Minimizing passengers' waiting time
- Minimizing the total number of drivers required

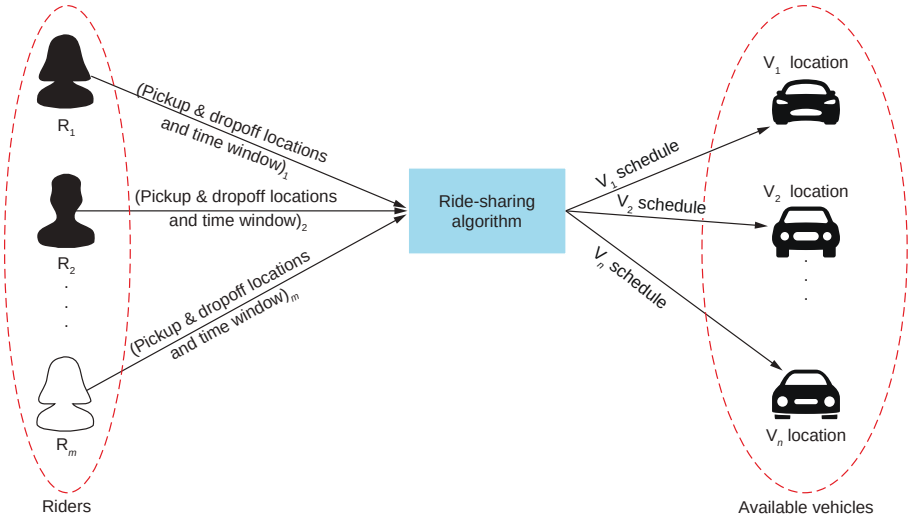



**Figure 2.14 Ride-sharing problem—this planning problem needs to be solved in a shorter amount of time, as delays could mean lost trips and a bad user experience.**

For the ride-sharing problem, both the search time and the quality of the solutions are important. On many popular ride-sharing platforms, dozens if not hundreds of users may simultaneously be searching for rides at the same place in a given district. Overly costly and time-consuming solutions would lead to higher operating costs (i.e., employing more drivers than necessary or calling in drivers from other districts) as well as the potential for lost business (bad user experiences may dissuade passengers from using the platform a second time) and high driver turnover.

In practice, the assignment of drivers to passengers goes well beyond the distance between passenger and driver—it may also include factors such as driver reliability, passenger rating, vehicle type, and pickup and destination location types. For example, a customer going to the airport may request a larger vehicle to accommodate luggage. In the upcoming chapters, we will discuss how to solve planning problems using different search and optimization algorithms.

*Control problems* require very fast solutions in real time. In most cases, this means a time span from a millisecond to a few seconds. Vehicle lateral or longitudinal motion control, surgical robot motion control, disruptions management, and ad hoc communication relaying are examples of control problems. Online optimization algorithms are required to handle these kinds of problems. Optimization tasks in both planning and control problems are often carried out repetitively—new orders will, for instance, continuously arrive in a production facility and need to be scheduled to machines in a way that minimizes the waiting time for all jobs.

Imagine a real-world situation where a swarm of unmanned aerial vehicles (UAVs) or micro aerial vehicles (MAVs) is deployed to search for victims trapped on untraversable terrain after a natural disaster, like an earthquake, avalanche, tsunami, tornado, wildfire, etc. The mission consists of two phases: a search phase and a relay phase. During the search phase, the MAVs will conduct a search according to the deployment algorithm. When a target is found, the swarm of MAVs will self-organize to utilize their range-limited communication capabilities and set up an ad hoc communication relay network between the victim and the base station, as illustrated in figure 2.15.

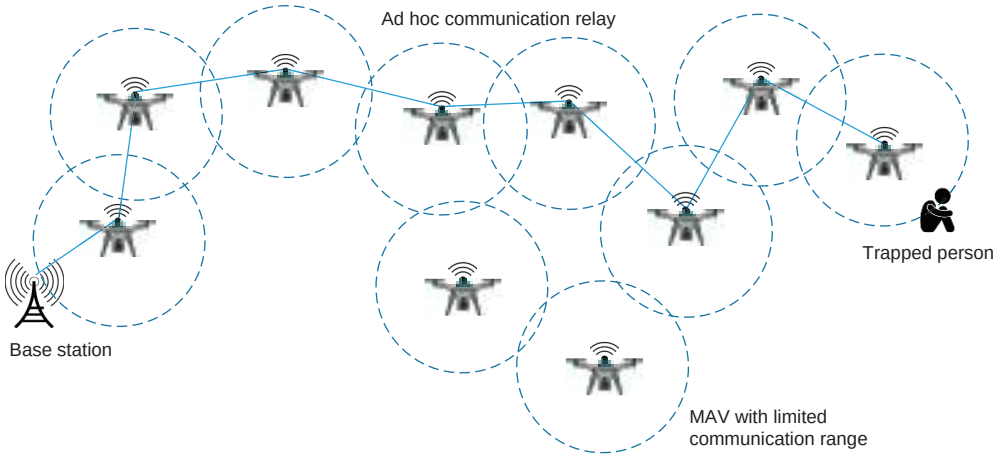



**Figure 2.15 Communication relaying problem—a swarm of MAVs must form an ad hoc communication relay between a base station and a trapped victim. The movement of the MAVs is a control problem that must be solved repeatedly, multiple times per second. In this case, speed is more important than accuracy, as minor errors can be immediately corrected during the next cycle.**

During the search phase, MAVs can be deployed to maximize the area covered. After they detect a victim, the MAVs can be repositioned to maximize the victim's visibility. The ad hoc communication relay network is then established to maximize the radio coverage in the swarm and find the shortest path between the MAV that detected the victim and the base station, given the following assumptions:

- MAVs are capable of situational awareness by combining data from three noise-prone sensors: a magnetic compass for direction, a speedometer for speed, and an altimeter for altitude.
- MAVs are capable of communicating via a standard protocol such as IEEE 802.11b with a limited range of 100 m.
- MAVs are capable of relaying ground signals as well as controlling signals sent among MAVs.
- MAVs have enough onboard power to sustain 30 minutes of continuous flight, at which point they must return to the base to recharge. However, the amount of flight time varies depending on the amount of signaling completed during flight.

- MAVs are capable of quickly accelerating to a constant flight speed of 10 m/s.
- MAVs are not capable of hovering and have a minimum turn radius of approximately 10 m.

For control problems such as MAV repositioning, search time is of paramount importance. As the MAVs cannot hover and thus must remain in constant motion, delayed decisions may lead to unexpected situations, such as mid-air collisions or a loss of signal. As instructions are sent (or repeated) every few milliseconds, each MAV must be able to decide its next move within that span of time. A MAV must account not only for its current position, target position, and velocity, but must also consider obstacles, communications signal strength, wind, and other environmental effects. Minor errors are acceptable, as they can be corrected in subsequent searches. In the upcoming chapters, we will discuss how to solve control problems like this.

This book will largely focus on complex, ill-structured problems that cannot be handled by traditional mathematical optimization or derivative-based solvers. We'll look at examples of design, planning and control problems in various domains. Next, let's take a look at how search and optimization algorithms are classified.

## 2.2 Classifying search and optimization algorithms

When we search, we try to examine different states to find a path from the start (initial) state to the goal state. Often, an optimization algorithm searches for an optimum solution by iteratively transforming a current state or a candidate solution into a new, hopefully better, solution. Search algorithms can be classified based on the way the search space is explored:

- *Local search* uses only local information about the search space surrounding the current solution to produce new solutions. Since only local information is used, local search algorithms (also known as local optimizers) locate local optima (which may or may not be global optima).
- *Global search* uses more information about the search space to locate global optima.

In other words, global search algorithms explore the entire search space, while local search algorithms only exploit neighborhoods.

Yet another classification distinguishes between deterministic and stochastic algorithms, as illustrated in figure 2.16:

- *Deterministic algorithms* follow a rigorous procedure in their path, and both the values of their design variables and their functions are repeatable. From the same starting point, they will follow the same path, whether you run the program today or tomorrow. Examples include, but are not limited to, graphical methods, gradient and Hessian-based methods, penalty methods, gradient projection methods, and graph search methods. Graph search methods can be further subdivided into blind search methods (e.g., depth-first, breadth-first, or Dijkstra) and informed search methods (e.g., hill climbing, beam search, best-first, A*, or contraction hierarchies). Deterministic methods are covered in part 1 of this book.

- *Stochastic algorithms* explicitly use randomness in their parameters or decision-making process or both. For example, genetic algorithms use some random or pseudo-random numbers, resulting in individual paths that are not exactly repeatable. With stochastic algorithms, the time taken to obtain an optimal solution cannot be accurately foretold. Solutions do not always get better, and stochastic algorithms sometimes miss the opportunity to find optimal solutions. This behavior can be advantageous, however, because it can prevent them from becoming trapped in local optima. Examples of stochastic algorithms include tabu search, simulated annealing, genetic algorithms, differential evolution algorithms, particle swarm optimization, ant colony optimization, artificial bee colony, firefly algorithm, etc. Most statistical machine learning algorithms are stochastic because they make use of randomness during the learning stage and they make predictions during the inference stage with a certain level of uncertainty. Moreover, some machine learning models are, like people, unpredictable. Models trained using human behavior-based data as independent variables are more likely to be unpredictable than those trained using independent variables that strictly follow physical laws. For example, the human intent recognition model is less predictable than a model that predicts the stress-strain curve of a material. Due to the uncertainty associated with machine learning predictions, machine learning–based algorithms used to solve optimization problems can be considered stochastic methods. Stochastic algorithms are covered in parts 2 to 5 of this book.

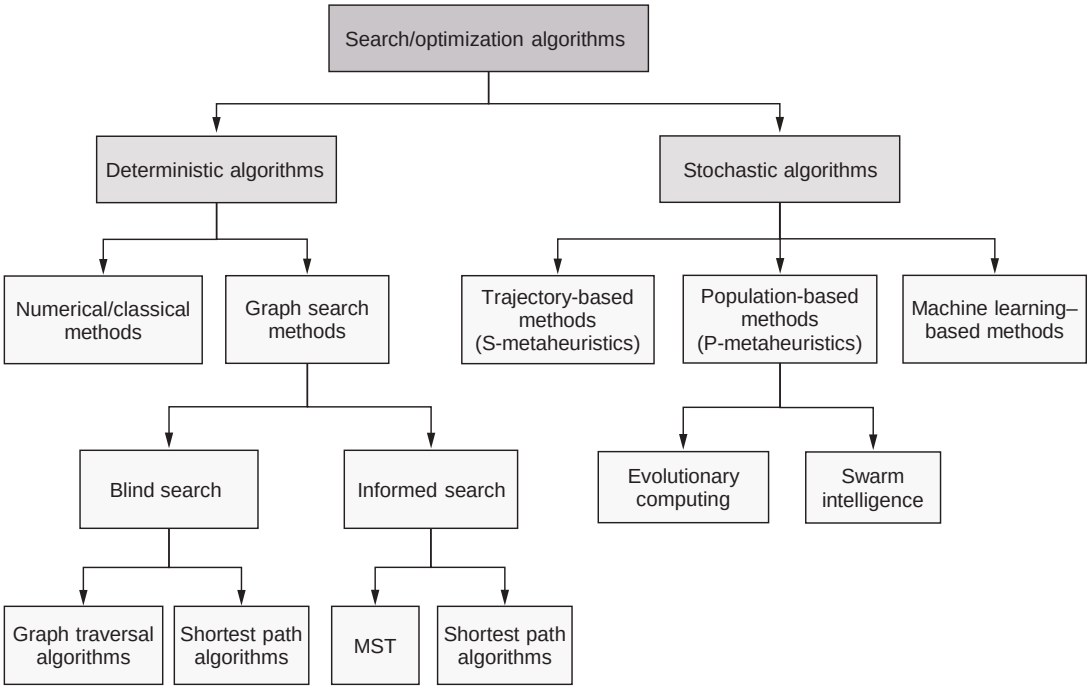



**Figure 2.16 Deterministic vs. stochastic algorithms. Deterministic algorithms follow a set procedure, and the results are repeatable, while stochastic searches have elements of randomness built into the algorithms.**

### Treasure-hunting mission

The search for an optimal solution in a given search space can be likened to a treasure-hunting mission. Imagine you and a group of friends decided to visit an island looking for pirate treasure.

All the areas on the island (except the active volcano area) correspond to the feasible search space of the optimization problem. The treasure corresponds to the optimal solution in this feasible space. You and your friends are the "search agents" launched to search for the solution, each following different search approaches. If you don't have any information that can guide you while searching, you are following a blind (uninformed) search approach, which is usually inefficient and time-consuming. If you know that the pirates used to hide the treasure in elevated spots, you could then directly climb up the steepest cliff and try to reach the highest peak. This scenario corresponds to the classic hill-climbing technique (informed search). Uninformed and informed search algorithms are presented in the next two chapters. You could also follow a trial-and-error approach, looking for hints and repeatedly moving from one place to another plausible place until you find the treasure. This corresponds to trajectory-based search, which we'll discuss in part 2 of the book.

If you do not want to take the risk of getting nothing and decide to share information with your friends instead of treasure-hunting alone, you will be following a population-based search approach. While working in a team, you may notice that some treasure hunters show better performance than others. In this case, only better-performing hunters can be kept, and new ones can be recruited to replace the lesser-performing hunters. This is akin to evolutionary algorithms, such as genetic algorithms, where the fittest hunters survive. Genetic algorithms are covered in part 3 of the book. Alternatively, you and other friends can try to emulate the success of the outperforming hunters in each area of the treasure island without getting rid of any team members and without recruiting new ones. This scenario uses the so-called swarm intelligence and corresponds to population-based optimization algorithms such as particle swarm optimization, ant colony optimization, and artificial bee colony algorithm. These algorithms will be discussed in part 4 of the book.

You alone, or with the help of your friends, can build a mental model based on historical data of previous and similar treasure-hunting missions, or you can train a reward predictor based on trial-and-error interaction with the treasure island (search space), taking the strength of the metal detector signal as a reward indicator. After a few iterations, you will learn to maximize the reward from the predictor and improve your behavior until you fulfill the desired goal and find the treasure. This corresponds to a machine learning–based approach, which we'll discuss in part 5 of this book.

## 2.3 Heuristics and metaheuristics

*Heuristics* (also known as *mental shortcuts* or *rules of thumb*) are solution strategies, seeking methods, or rules that can facilitate finding acceptable (optimal or near-optimal) solutions to a complex problem in a practical time. Despite the fact that heuristics can seek near-optimal solutions at a reasonable computational cost, they cannot guarantee either feasibility or degree of optimality.

### "Eureka! Eureka!"

The word *heuristic* comes from the Greek word *heuriskein*, which means "to find or discover." The past tense of this verb, *eureka*, was used by the Greek mathematician, physicist, engineer, astronomer, and inventor Archimedes. Archimedes was contracted to detect fraud in the manufacture of a golden crown, and he accepted the challenge. During a subsequent visit to the public baths, he had a revelation. As his body submerged in the water, he observed that the more he sank, the more water was displaced, offering an exact measure of his volume. Realizing the principle at play, he deduced that a crown containing silver, being less dense than pure gold, would need to have greater volume to match the weight of a pure gold crown. Consequently, it would displace more water. Recognizing the solution, Archimedes leaped out of the bath and hurried home, exclaiming "Eureka! Eureka!" which translates to "I've found it! I've found it!"

The term metaheuristic is a combination of two Greek words: *meta*, which means "beyond, on a higher level," and *heuristics*. It's a term coined by Fred Glover, inventor of the tabu search (discussed in chapter 6) to refer to high-level strategies used to guide and modify other heuristics to enhance their performance. The goal of metaheuristics is to efficiently explore the search space in order to find optimal or near-optimal solutions. Metaheuristics may incorporate mechanisms to achieve a trade-off between exploration (diversification) and exploitation (intensification) of the search space to avoid getting trapped in confined areas of the search space while also finding optimal or near-optimal solutions in a reasonable amount of time. Finding this balance of exploration and exploitation is crucial in heuristics, as discussed in section 1.5. Metaheuristic algorithms are often global optimizers that can be applied to different linear and nonlinear optimization problems with relatively few modifications for specific problems. These algorithms are often robust and can handle different problem sizes, problem instances, and random variables.

Let's assume that we have 6 objects with different sizes (2, 4, 3, 6, 5, and 1) and we need to pack them into a minimum number of bins. Each bin has a limited size of 7, so the total size of the objects in the bin should be 7 or less. If we have $n$ objects, there are $n!$ possible ways of packing the objects. The minimum number of bins we need is the *lower bound*. To calculate this lower bound, we need to find the total number of object sizes ($2 + 4 + 3 + 6 + 5 + 1 = 21$). The lower bound is $21 / 7 = 3$ bins. This means that we need at least 3 bins to pack these objects. Figure 2.17 illustrates two heuristics that can be used to solve this bin packing problem.

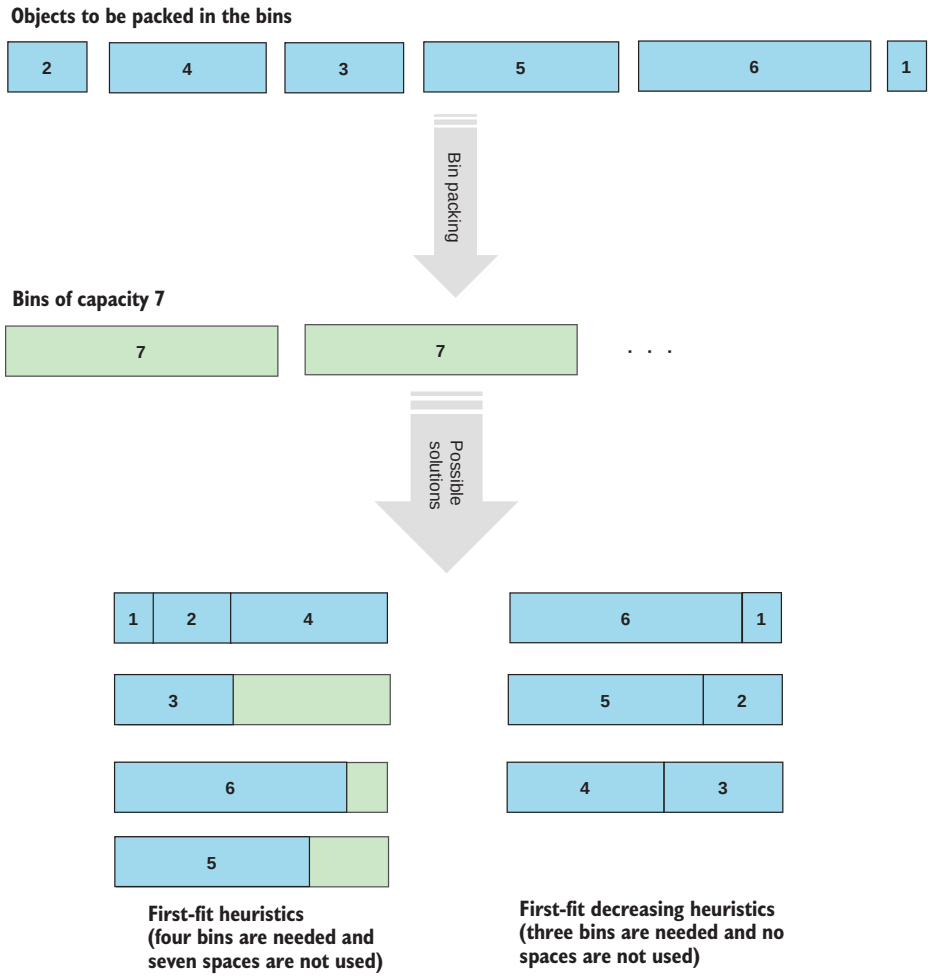



**Figure 2.17 Handling the bin packing problem using first-fit and first-fit decreasing heuristics**

First-fit heuristics pack the objects following their order without taking into consideration their sizes. This results in the need for four bins that are not fully utilized, as there are seven spaces left in three of these bins. If we apply the first-fit decreasing heuristic, we will order the objects based on their sizes and pack them following this order. This heuristic allows us to pack all the objects in three fully utilized bins, which is the lower bound.

In the previous example, all the objects have the same height. However, in a more generalized version, let's consider objects with different widths and heights, as illustrated in figure 2.18. Applying heuristics such as smallest-first can allow us to load the container much faster. Some heuristics do not guarantee optimality; for example, the largest-first heuristic gives a suboptimal solution, as one object is left out. This can be considered an infeasible solution if we need to load all the objects into the container, or it will be a suboptimal solution if the objective is to load as many objects as possible.

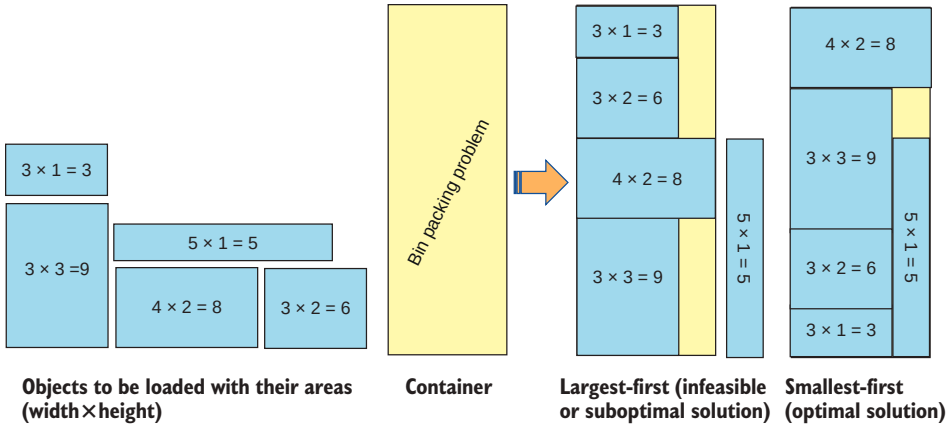



**Figure 2.18 Bin packing problem. Using heuristics allows us to solve the problem much faster than with a brute-force approach. However, some heuristic functions may result in infeasible or suboptimal solutions, and they do not guarantee optimality.**

To solve this problem in Python, let's first define the objects, the containers, and what it means to place an object inside a container. For the sake of simplicity, the following listing avoids custom classes and uses `numpy` arrays instead.

**Listing 2.7 Bin packing problem**

```
import numpy
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.colors import rgb2hex

width = 4
height = 8
container = numpy.full((height,width), 0)

objects = [[3,1],[3,3],[5,1],[4,2],[3,2]]

def fit(container, object, obj_index, rotate=True):
    obj_w = object[0]
    obj_h = object[1]
    for i in range(height - obj_h + 1): C
        for j in range(width - obj_w + 1):
            placement = container[i : i + obj_h, j : j + obj_w]
            if placement.sum() == 0:
                container[i : i + obj_h, j : j + obj_w] = obj_index
                return True
        return fit(container, object[::-1], obj_index, rotate=False)
```

**Define the dimensions of the container, and initialize the numpy array to 0s.**

**Represent objects to be placed as [width, height].**

**The fit function places objects into the container, either through direct placement, shifting, or rotation.**

The `fit` function attempts to write a value to a 2D slice of the container, provided there are no values in that slice already (the sum is 0). If that fails, it shifts along the container from top to bottom, from left to right, and tries again. As a last resort, it tries the same thing but with the object rotated by 90 degrees.

The first heuristic prioritizes fitting by object area in descending order:

```
def largest_first(container, objects):
    excluded = []
    assigned = []
    objects.sort(key=lambda obj: obj[0] * obj[1], reverse=True)
    for obj in objects:
        if not fit(container, obj, objects.index(obj) + 1):
            excluded.append(objects.index(obj) + 1)
        else:
            assigned.append(objects.index(obj) + 1)
    if excluded: print(f"Items excluded: {len(excluded)}")
    visualize(numpy.flip(container, axis=0), assigned)
```

**Sort elements by area in descending order.**

**Some objects may not fit; we can keep track of them using a list.**

**Visualize the filled container.**

The output of this code is shown in figure 2.19. The code for visualizing this result is included in the full code files for listing 2.7, available in the book's GitHub repo.

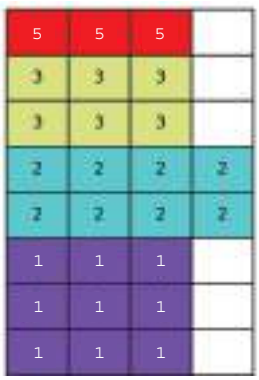



**Figure 2.19 Bin packing using the largest-first heuristic—one object has been excluded, as it does not fit in the remaining space.**

The second heuristic sorts first by width and then by total area, in ascending order:

```
def smallest_width_first(container, objects):
    excluded = []
    assigned = []
    objects.sort(key=lambda obj: (obj[0], obj[0] * obj[1]))
    for obj in objects:
        if not fit(container, obj, objects.index(obj) + 1):
            excluded.append(objects.index(obj) + 1)
        else:
            assigned.append(objects.index(obj) + 1)
    if excluded: print(f"Items excluded: {len(excluded)}")
    visualize(numpy.flip(container, axis=0), assigned)
```

**Sort by width as primary key, and then by area in ascending order.**

**Visualize the solution.**

The `smallest_width_first` heuristic manages to successfully fit all the objects into the container, as shown in figure 2.20.

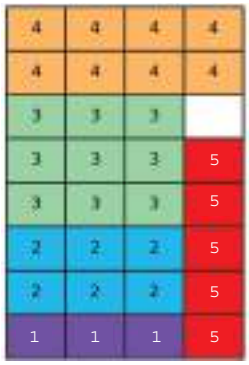



**Figure 2.20 Bin packing problem using the smallest-first heuristic—all five objects have been successfully placed in the container.**

Different heuristic search strategies can be used to generate candidate solutions. These strategies include, but are not limited to, search by repeated solution construction (e.g., graph search and ant colony optimization), search by repeated solution modification (e.g., tabu search, simulated annealing, genetic algorithm, and particle swarm optimization), and search by repeated solution recombination (e.g., genetic algorithm and differential evolution).

Let's reconsider the cargo bike loading problem discussed in section 1.3.3. We can order the items to be delivered based on their efficiency (profit per kg), as shown in table 2.2.

**Table 2.2 Packages ranked by efficiency. The efficiency of a package is defined as the profit per kilogram.**

| Item | Weight (kg) | Profit ($) | Efficiency ($/kg) |
|---|---|---|---|
| 10 | 7.8 | 20.9 | 2.68 |
| 7 | 4.9 | 10.3 | 2.10 |
| 4 | 10 | 12.12 | 1.21 |
| 1 | 14.6 | 14.54 | 1 |
| 8 | 16.5 | 13.5 | 0.82 |
| 6 | 9.6 | 7.4 | 0.77 |
| 2 | 20 | 15.26 | 0.76 |
| 9 | 8.77 | 6.6 | 0.75 |
| 3 | 8.5 | 5.8 | 0.68 |
| 5 | 13 | 8.2 | 0.63 |

Using a search strategy based on the *repeated solution construction* heuristic, we can start by applying a greedy principle and pick items based on their efficiency until we reach the maximum payload of the cargo bike (100 kg) as a hard constraint. The steps for this are shown in table 2.3.

**Table 2.3 Repeated solution construction—packages are added to the bike until the maximum capacity is reached.**

| Step | Item | Add? | Total weight (kg) | Total profit ($) |
|---|---|---|---|---|
| 1 | 10 | Yes | 7.8 | 20.9 |
| 2 | 7 | Yes | 12.7 | 31.2 |
| 3 | 4 | Yes | 22.7 | 43.32 |
| 4 | 1 | Yes | 37.3 | 57.86 |
| 5 | 8 | Yes | 53.8 | 71.36 |
| 6 | 6 | Yes | 63.4 | 78.76 |
| 7 | 2 | Yes | 83.4 | 94.02 |
| 8 | 9 | Yes | 92.17 | 100.62 |
| 9 | 3 | No | (100.67) | - |
| 10 | 5 | No | (113.67) | - |

We obtain the following subset of items: 10, 7, 4, 1, 8, 6, 2, and 9. This can also be written as (1,1,0,1,0,1,1,1,1,1), which when read from left to right shows that we include items 1, 2, 4, 6, 7, 8, 9, and 10 (and exclude items 3 and 5). This results in a total profit of $100.62 and a weight of 92.17 kg. We can generate more solutions by repeating the process of adding objects, starting with an empty container.

Instead of creating one or more solutions completely from scratch, we could also think about ways of modifying an existing feasible solution—this is a *repeated solution modification-based* heuristic search strategy. Consider the previous solution generated for the cargo-bike problem: (1,1,0,1,0,1,1,1,1,1). We know that this feasible solution is not optimal, but how can we improve it? We could do so by removing item 9 from the cargo bike and adding item 5. This process of removing and adding results in a new solution, (1,1,0,1,1,1,1,1,0,1), with a total profit of $102.22 and a weight of 96.4 kg.

Another approach is to combine existing solutions to generate new solutions to progress in the search space—this is *repeated solution recombination*. Suppose the following two solutions are given:

- $S_1$ = (1,1,1,1,1,0,0,1,0,1) with a weight of 75.8 kg and a profit of $75.78
- $S_2$ = (0,1,0,1,1,0,1,1,1,1) with a weight of 80.97 kg and a profit of $86.88

As illustrated in figure 2.21, we can take the configuration of the first two items of $S_1$ and the last eight items of $S_2$ to get a new solution. This means that we include items 1, 2, 4, 5, 7, 8, 9, and 10 in the new solution and exclude items 3 and 6. This yields a new solution: $S_3$ = (1,1,0,1,1,0,1,1,1,1) with a weight of 95.57 kg and a higher profit of $101.42.

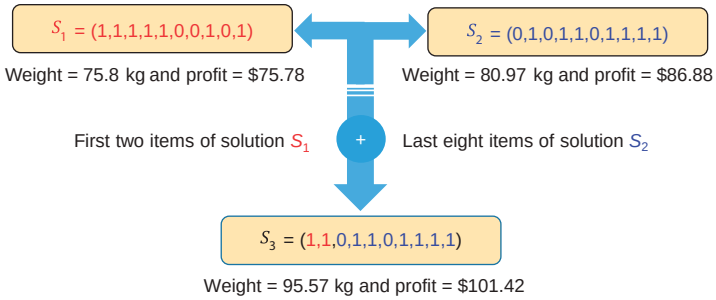



**Figure 2.21 Repeated solution recombination—taking the first two elements of $S_1$ and adding the last eight elements of $S_2$ yields a new, better solution.**

## 2.4 Nature-inspired algorithms

Nature is the ultimate source of inspiration. Problems in nature are usually ill-structured, dynamic, partially observable, nonlinear, multimodal, and multi-objective with hard and soft constraints and with no or limited access to global information. Nature-inspired algorithms are computational models that mimic or reverse engineer the intelligent behaviors observed in nature. Examples include molecular dynamics, cooperative foraging, division of labor, self-replication, immunity, biological evolution, learning, flocking, schooling, and self-organization, just to name just a few.

Molecular dynamics (the science of simulating the motions of a system of particles) and thermal annealing inspired scientists to create an optimization algorithm called *simulated annealing*, which we'll discuss in chapter 5. Evolutionary computing algorithms such as genetic algorithm (GA), genetic programming (GP), evolutionary programming (EP), evolutionary strategies (ES), differential evolution (DE), cultural algorithms (CA), and co-evolution (CoE) are inspired by evolutionary biology (the study of the evolutionary processes) and biological evolution. Part 3 of this book will cover a number of evolutionary computing algorithms.

Ethology (the study of animal behavior) is the main source of inspiration for swarm intelligence algorithms such as particle swarm optimization (PSO), ant colony optimization (ACO), artificial bee colony (ABC), firefly algorithm (FA), bat algorithm (BA), social spider optimization (SSO), butterfly optimization algorithm (BOA), dragonfly algorithm (DA), krill herd (KH), shuffled frog leaping algorithm (SFLA), fish school search (FSS), dolphin partner optimization (DPO), dolphin swarm optimization algorithm (DSOA), cat swarm optimization (CSO), monkey search algorithm (MSA), lion optimization algorithm (LOA), cuckoo search (CS), cuckoo optimization algorithm (COA), wolf search algorithm (WSA), and grey wolf optimizer (GWO). Swarm intelligence-based optimization algorithms are covered in part 4 of this book.

Neural networks (NNs) are computational models inspired by the structure and functioning of biological neural networks. How NNs can be used to solve search and optimization problems is described in part 5 of this book. Tabu search (explained in

chapter 6) is based on evolving memory (adaptive memory and responsive exploration), which is studied in behavioral psychology (the science of behavior and mind). Reinforcement learning is a branch of machine learning that draws inspiration from several sources such as psychology, neuroscience, and control theory, and it can be used to solve search and optimization problems, as described in the last chapter of the book.

Other nature-inspired search and optimization algorithms include, but are not limited to, bacterial foraging optimization algorithm (BFO), bacterial swarming algorithm (BSA), biogeography-based optimization (BBO), invasive weed optimization (IWO), flower pollination algorithm (FPA), forest optimization algorithm (FOA), water flow-like algorithm (WFA), water cycle algorithm (WCA), brainstorm optimization algorithm (BSO), stochastic diffusion search (SDS), alliance algorithm (AA), black hole algorithm (BH), black hole mechanics optimization (BHMO), adaptive black hole algorithm (BHA), improved black hole algorithm (IBH), levy flight black hole (LBH), multiple population levy black hole (MLBH), spiral galaxy-based search algorithm (GbSA), galaxy-based search algorithm (GSA), big-bang big-crunch (BBBC), ray optimization (RO), quantum annealing (QA), quantum-inspired genetic algorithm (QGA), quantum-inspired evolutionary algorithm (QEA), quantum swarm evolutionary algorithm (QSE), and quantum-inspired particle swarm optimization (QPSO). For a comprehensive list of metaheuristic algorithms, see S.M. Almufti's "Historical survey on metaheuristics algorithms" [8].

In the five parts of this book, we'll explore five primary categories of search and optimization algorithms: graph search algorithms, trajectory-based optimization, evolutionary computing, swarm intelligence algorithms, and machine learning methods. The following algorithms are covered within these categories:

- Graph search methods (blind or uninformed search and informed search algorithms)
- Simulated annealing (SA)
- Tab search (TS)
- Genetic algorithm (GA)
- Particle swarm optimization (PSO)
- Ant colony optimization (ACO)
- Artificial bee colony (ABC)
- Graph convolutional network (GCN)
- Graph Attention Network (GAT)
- Self-organizing map (SOM)
- Actor-Critic (A2C) architecture
- Proximal policy optimization (PPO)
- Multi-armed bandit (MAB)
- Contextual multi-armed bandit (CMAB)

Throughout this book, we'll look at several real-world problems and see how these algorithms can be applied.

## Summary

- Search and optimization problems can be classified based on the number of decision variables (univariate and multivariate problems), the types of decision variables (continuous, discrete, or mixed-integer), the number of objective functions (mono-objective, multi-objective, or constraint-satisfaction problems), the landscape of the objective function (unimodal, multimodal, or deceptive), the number of constraints (unconstrained and constrained problems), and the linearity of the objective functions and constraints (linear problems and nonlinear problems).
- Based on the expected quality of the solutions and the search time permitted to find the solutions, optimization problems can also be categorized as design problems (strategic functions), planning problems (tactical functions), or control problems (operational functions).
- Search and optimization algorithms can be classified based on the way the search space is explored (local versus global search), on their optimization speeds (online versus offline optimization), and the determinism of the algorithm (deterministic versus stochastic).
- Heuristics (also known as *mental shortcuts* or *rules of thumb*) facilitate finding acceptable (optimal or near-optimal) solutions to complex problems in a reasonably practical time.
- Metaheuristics are high-level strategies used to guide and modify other heuristics to enhance their performance.
- Nature-inspired algorithms are computational models that mimic or reverse engineer the intelligent behaviors observed in nature to solve complex ill-structured problems.