10 Other swarm intelligence algorithms to explore

This chapter covers

- Getting familiar with ant colony optimization metaheuristics
- Understanding different variants of ant colony optimization
- Understanding artificial bee colony
- Applying these swarm intelligence algorithms to solve continuous and discrete optimization problems

In the previous chapter, we looked at the particle swarm optimization (PSO) algorithm, but ant colony optimization (ACO) and artificial bee colony (ABC) are other widely used swarm intelligence algorithms, drawing inspiration from ants and bees to tackle diverse optimization problems. Let's revisit the treasure hunting mission and assume you still want to follow a cooperative and iterative approach to find the treasure (which is the best solution in the case of an optimization problem). You and your friends are divided into two groups: the ant group and the bee group. Each group has its own unique way of finding the treasure, using ant colony optimization or the artificial bee colony algorithm. You can join either of these two groups.

As treasure-hunting ants, you and some of your friends will start at the base camp and explore different paths to find the treasure. As you explore, each of you leaves a trail of special chalk (pheromones) behind. The more promising the path, the more chalk you leave on that path. When your friends find your chalk trail, they can decide to follow it or to explore a new path. Over time, the most promising paths will have the strongest chalk trails, and eventually the whole group will converge on the path that leads to the treasure.

As treasure-hunting bees, you'll use a different approach. You have forager bees and scout bees. Forager bees concentrate on searching nearby areas, while scout bees fly out and randomly explore the island, searching for clues leading to the treasure. When a bee finds a promising clue, it returns to the base camp and performs a "waggle dance" to communicate the location and quality of the clue to the other friends (onlooker bees). This process continues until the group finds the best path to the treasure.

This chapter presents ant colony optimization and artificial bee colony as swarm intelligence algorithms. The open traveling salesman problem, function optimization, routing problem, pump design, and a supply-demand problem are discussed in this chapter and its supplementary exercises in appendix C.

10.1 Nature's tiny problem-solvers

Ants are tiny creatures that can solve complex problems better than some humans. Ants may be small, but when they work together in a colony, they can accomplish some incredible feats. During foraging, they can find the shortest path to a food source, build intricate tunnels, and even take down prey much larger than themselves! During nest construction, some ants cut leaves from plants and trees, others forage for leaves hundreds of meters away from their nest to construct highways to and from their foraging sites, and yet other ants form chains of their own bodies, allowing them to cross wide gaps and pull stiff leaves together to form a nest. In the latter case, the worker ants form a chain along the edge of the leaf and pull the edges together by shortening the chain one ant at a time. Once the leaf edges are in place, weaver ants hold one larva each in their mandibles and gently squeeze the larva to produce silk, which is used to glue the leaf edges together.

Fascinating facts about the mighty ant

- Ants appeared on earth some 100 million years ago, making them one of the oldest groups of insects on the planet.
- Ants have a current total population estimated at 10^{16} individuals. It is estimated that the total weight of ants is in the same order of magnitude as the total weight of human beings.
- Ants are incredibly strong for their size. Some species can carry objects that are 50 times their body weight! To put that in perspective, this is like a human carrying a car!
- About 2% of all insects are social. There are around 12,000 different types of ants, and most ants are social insects.

(continued)

- Ants are considered the densest population in the world. They live in colonies of 30 to millions of individuals. Some colonies like Formica Yesensis have approximately 1,080,000 queens and 306,000,000 workers and live in 45,000 nests connected to each other over an area of 2.7 square kilometers.
- Ants use pheromones as their primary medium of stigmergic communication. However, ants also use other forms of communication, including visual, auditory, and tactile communication. For example, some species of ants use sound to communicate with each other. These sounds can range from simple clicks and pops to more complex signals that convey information about food sources, nest locations, and other important information. Some species of ants produce sounds in the audible range of humans (20Hz to 20kHz). For example, leafcutter ants are known to produce a clicking sound when they communicate with each other. The frequency of these clicks can range from 1 to 10 kilohertz. Other species of ants produce sounds that are beyond the range of human hearing. For example, some species of army ants produce ultrasonic sounds that can be used to locate prey or communicate with each other. If you're interested, take a look at the "What Sound Does an Ant Make?" video on YouTube (http://mng.bz/aEKo).

An ant is a simple stimulus-response creature that is incapable of achieving complex tasks alone. However, as a colony, ants show an amazing capability to perform complex tasks without any planning, a central controller, centralized supervision, or direct communication. Ants employ an indirect communication mechanism known as stigmergic communication. *Stigmergy* is a concept introduced by the French biologist Pierre-Paul Grassé in 1959 as an indirect method of communication among social insects involving environmental modifications. These environmental modifications serve as external or shared memory between the insects.

Ants use pheromones as their primary medium of stigmergic communication. As they travel to and from a food source, they deposit pheromones along their path. Other ants can detect these pheromones, which influences their decision-making when choosing a path. This allows ants to work together as a cohesive unit and accomplish complex tasks such as finding the shortest path from the nest to a food source and vice versa. The absence of direct communication or a central controller makes the actions of ants seem almost as if they are coordinated by some form of collective intelligence. In essence, the phenomenon of stigmergic communication allows social insects like ants to use their collective knowledge and behavior to achieve tasks beyond their individual abilities.

Ant colony optimization (ACO) is inspired by the foraging behavior of ants. As they forage for food, ants initially explore randomly around the nest area. Once an ant discovers a food source, it carries some of the food back to the nest while laying a pheromone trail along its path. Other ants then follow the pheromone trail to the food source, as illustrated in figure 10.1. As more and more ants follow the pheromone trail to the

food source, the intensity of the pheromone trail increases, making it more attractive to other ants. In contrast, because pheromone trails are not fixed and will gradually evaporate over time, the pheromone trail on the longer path will evaporate. Eventually, a single pheromone trail becomes dominant, and most of the ants follow this trail to and from the food source. In this way, ants can find the shortest path between the nest and the food source through a process of collective or swarm intelligence.

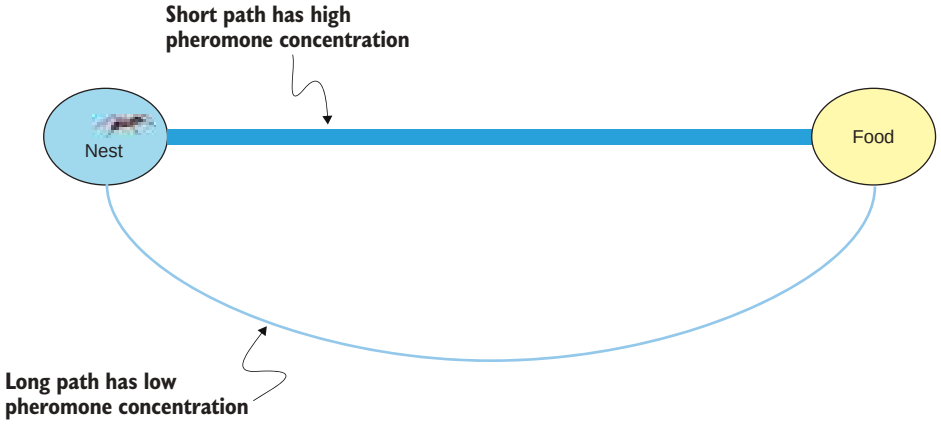

Figure 10.1 Ant foregoing process. A foraging ant deposits a pheromone trail along the path it takes on its way back to the nest. Other ants will likely follow the path with a stronger pheromone trail to reach the discovered food source.

As I explained in the previous chapter, the majority of the research carried out on swarm intelligence algorithms was initially based on experimental observations. To understand the collective behavior of ants during food foraging and to derive heuristics for the ACO algorithm, two famous experiments were conducted: the binary bridge experiment and the bridges with non-equal lengths experiment.

The binary bridge experiment was designed to observe the behavior of ants when presented with two equal-length bridges connecting their nest to a food source (figure 10.2a). The experiment aimed to investigate how ants determine the best path to use and how they adapt their behavior over time. Initially, the ants chose one of the two bridges randomly. As the ants traveled back and forth between the nest and the food source, they deposited pheromones along the path they took. As time progressed, more ants followed the path with the higher concentration of pheromones, which made the path even more attractive to other ants. Eventually, one of the two bridges became dominant, and most of the ants used it to travel between the nest and the food source. The ants' decision-making process was based on the principle of positive feedback, where the ants reinforced the path with the highest pheromone concentration, making it even more attractive to other ants.

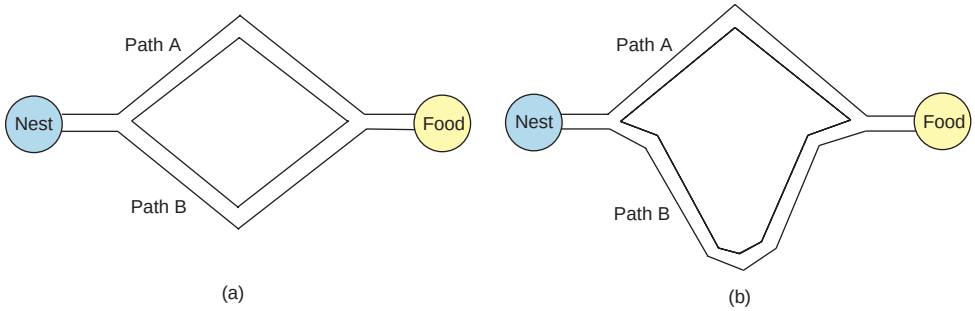

Figure 10.2 a) Binary bridge experiment; b) bridges with non-equal lengths experiment

The bridges with non-equal lengths experiment (figure 10.2b) is an extension of the binary bridge experiment with one branch of the bridge being longer than the other. The goal of this experiment was to observe how ants adapted their behavior when presented with two paths of different lengths. The experiment showed that ants tended to select the shorter path over the longer one. This was because ants traveling on the shorter path returned to the nest earlier than those on the longer path. As a result, the pheromone trail on the shorter path was reinforced sooner than that on the longer path, making it more attractive to other ants. This reinforcement behavior is called *autocatalytic behavior*.

The role of pheromones in the collective behavior of ants can be summarized in the following key points:

- The pheromone trail acts as a collective memory for the ants to communicate through by sensing and recording their foraging experience.
- The pheromone trail evaporates over time, introducing changes in the environment that can influence the ants' behavior.
- The concentration of pheromones on the trail represents a feedback signal that influences the ants' decision-making process.

Let's now dive deep into ACO metaheuristics.

10.2 ACO metaheuristics

Ant colony optimization (ACO) mimics the behavior of real ant colonies by having a group of "artificial ants" search for the best solution to a problem. These artificial ants leave "pheromone trails" to communicate with each other, just like real ants do, and eventually converge on the best solution.

To simulate the behavior of ants, let's assume we have a nest and a food source connected through two paths with different lengths L_1 and L_2, as in the case of the bridges

with non-equal lengths. Let's now assign a computational parameter τ to represent the pheromone deposited by the ants. We'll initially assign equal values of pheromones to each path: $\tau_1 = \tau_2$ as shown in figure 10.3. We then start by placing m ants at the nest. Let's assume that these artificial ants exactly mimic the real ants and take decisions based on the pheromone concentration, but without any knowledge of the lengths of the paths. For each ant k, this ant traverses path 1 with probability

$$p_1 = \frac{\tau_1}{\tau_1 + \tau_2} \qquad \textbf{10.1}$$

This ant thus traverses path 2 with a probability $p_2 = 1 - p_1$.

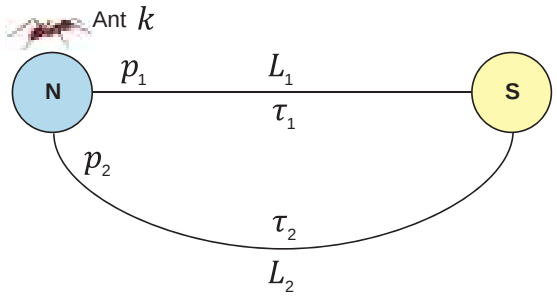

Figure 10.3 ACO simulation

As $\tau_1 = \tau_2$, the ant k will randomly pick one of the two paths, as both have same probability to of being traversed. After traversing the selected paths, pheromone concentration on each path needs to be updated. This pheromone update includes two phases: evaporation and deposit. During the evaporation phase, the pheromone concentration τ is decremented as follows:

$$\tau(t+1) = (1-\rho) \times \tau(t),\ \rho \in (0, 1) \qquad \textbf{10.2}$$

where ρ specifies the rate of evaporation. Figure 10.4 shows the effect of the pheromone evaporation rate during the foraging process based on a NetLogo simulation. NetLogo is a multi-agent programmable modeling environment used to simulate natural and social phenomena. It allows users to create, experiment with, and analyze simulations of complex systems, such as ecosystems, economies, and social networks. The foraging behavior of ants is shown in NetLogo's Ants model (https://ccl.northwestern.edu/netlogo/models/Ants).

Figure 10.4 Effect of evaporation rate during the food foraging process. In the simulation, the ants initiate their search for food from the central nest, which is surrounded by three food sources shown as blobs. The pheromone trails are shown in white. Upon discovering a food item, an ant transports it back to the nest, leaving behind a chemical trail. This trail is then followed by other ants that pick up the scent, directing them toward the food source. As more ants continue to retrieve food, they strengthen the chemical trail.

As you can see, if the evaporation rate is set to 0, the pheromone trail will never evaporate, and the ants will follow the same path repeatedly. This will cause the ants to become trapped in a local optimum, and they will not be able to explore other paths or find a better solution. On the other hand, if the evaporation rate is set to 1, the pheromone trail will evaporate at the maximum rate, which means the ants will not be able to follow any trail, and they will be forced to explore the environment randomly. This can result in slow convergence to the optimal solution.

During the deposit phase, each ant leaves more pheromones on its traversed path. Figure 10.5 shows the different methods used for pheromone updates:

- *Online step-by-step pheromone update*—Each ant deposits a certain amount of pheromones on the path it has traversed. This will increase the probability of another ant choosing the same edge:

$$\tau(t+1) = \tau(t) + \Delta\tau \qquad \textbf{10.3}$$

  There are different approaches for choosing the value of $\Delta\tau$. Following the *ant density model*, the ant adds a constant amount Q to each traversed edge. This means that the final pheromone added to the edge will be proportional to the number of ants choosing it. The higher the density of the traffic on the edge, the more desirable that edge becomes as a component of the final solution. This method does not take the quality of the solution (i.e., the edge length) into account. In the *ant quantity model*, the amount of pheromones deposited is proportional to the quality of the solution obtained by the ant. For example, an ant traversing between node i and j will deposit a quantity Q/d_{ij}, where d_{ij} is the distance between i and j. In this case, only local information, d_{ij}, is used to update pheromone concentrations. Lower cost edges are made more desirable.

- *Online delayed pheromone update (or ant cycle model)*–Once the ant constructs the solution, it retraces its steps and updates the pheromones trails on the edges it has traversed based on the quality of the solution. The amount of pheromones deposited is determined by the quality of the solution obtained by the ant as follows:

$$\Delta\tau_{ij}^{k}(t) = \frac{Q}{L^{k}(t)} \tag{10.4}$$

where Q is a constant and L^k is the length of the path constructed by ant k. For each edge (i,j) of the corresponding path, and after all the ants have completed their tours, the total amount of pheromones deposited will be

$$\tau_{ij}(t+1) = \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t) \tag{10.5}$$

where m is the number of ants.

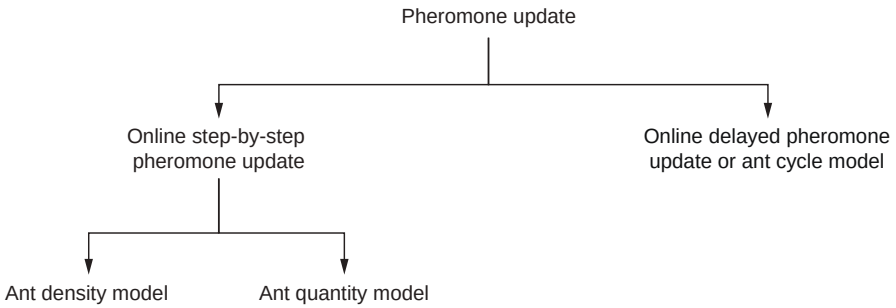

Figure 10.5 Pheromone update methods

In summary, with online step-by-step pheromone update, the ant updates the pheromone trail τ_{ij} on the edge (i,j) when moving from node i to node j. In online delayed pheromone update, once a path is constructed, the ant can retrace the same path backward and update the pheromone trails on the traversed edges. Which method you choose depends on the specific problem being solved. Any combination of online step-by-step pheromone updates and online delayed pheromone updates is also possible.

10.3 ACO variants

ACO has been used to solve a wide range of optimization problems, such as vehicle routing problems, scheduling problems, and optimal assignment problems. Over the years, several variants of the algorithm have been developed, as shown in figure 10.6.

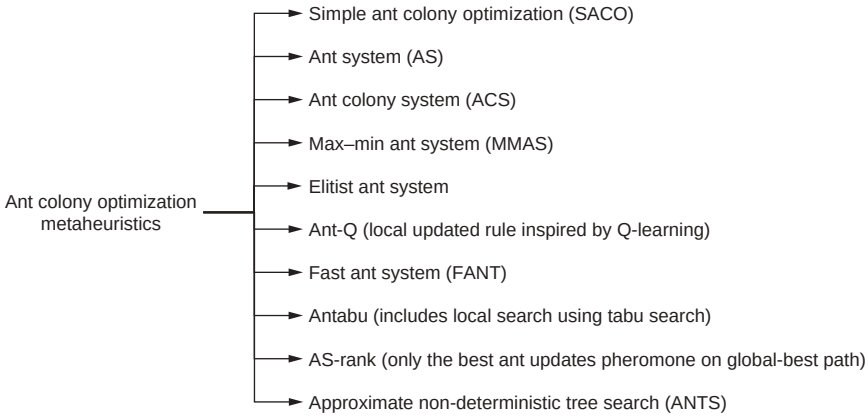

Figure 10.6 Examples of ACO variants

These variants have different strengths and weaknesses, and the choice of variant depends on the specific problem being solved. In the following subsections, we will discuss some of these variants.

10.3.1 Simple ACO

Simple ACO (SACO) is an algorithmic implementation of the double bridge experiment. Consider the problem of finding the shortest path between two nodes on a graph, as shown in figure 10.7.

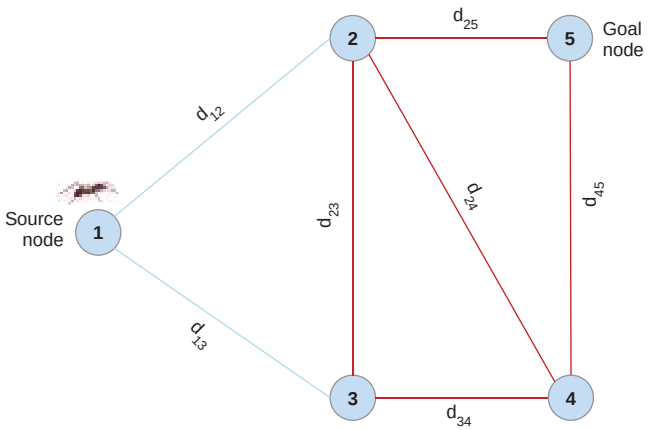

Figure 10.7 Shortest path problem

Let's solve this problem using SACO. On each edge, we'll assign a small random value to indicate the initial pheromone concentration, $\tau_{ij}(0)$. Then we'll place a number of ants, $k = 1,\ldots,m$ on the source node.

For each iteration of SACO, each ant will incrementally construct a path (solution) to the destination node. Initially, an ant will randomly select which edge to follow next. Later, each ant will execute a decision policy to determine the next edge of the path. At each node i, the ant has a choice to move to any of the j nodes connected to it, based on the following transition probability:

$$p_{ij}^{k}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}(t)]^{\beta}}{\sum_{l \in N_i^k} [\tau_{il}(t)]^{\alpha} \cdot [\eta_{il}(t)]^{\beta}} & \text{if } j \in N_i^k \\ 0 & \text{if } j \notin N_i^k \end{cases} \quad \textbf{10.6}$$

where

- N_i^k is the set of feasible nodes connected to node i, with respect to ant k.
- τ_{ij} is the amount of pheromones deposited for transition from state i to j.
- η_{ij} is a heuristic value that represents the desirability of state transition ij (a priori knowledge, typically $1/d_{ij}$, where d is the distance).
- $\alpha \geq 0$ is a parameter that controls the influence of τ_{ij}. α is used to amplify the influence of the pheromone. Large values of α give excessive importance to the pheromone, especially the initial random pheromones, which may lead to rapid convergence to suboptimal paths.
- $\beta \leq 1$ is a parameter that controls the influence of the desirability of the edge η_{ij}.

In the shortest path problem, assume that we use five ants, an initial pheromone value of 0.5, and $\alpha = \beta = 1$. The first ant ($k = 1$), placed at the source node, has two neighboring nodes {2,3}, as shown in figure 10.8.

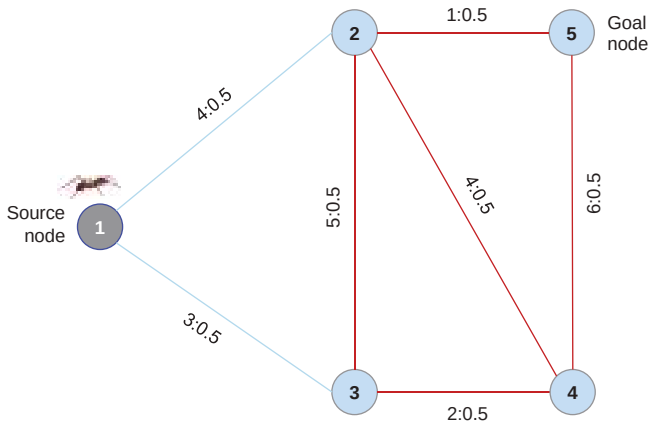

Figure 10.8 The first ant is at source node 1 with neighbors 2 and 3. There are two numbers on each edge separated by a colon. The first number represents the length of the edge, and the second represents the current pheromone concentration on the edge.

Considering the inverse of the edge length as edge desirability, this ant needs to choose between nodes 2 and 3 by applying the transition probability as follows:

$$p_{12}^{1} = \frac{\tau_{12}^{\alpha}\eta_{12}^{\beta}}{\sum_{l\in N_1^1}\tau_{1l}^{\alpha}\eta_{1l}^{\beta}} = \frac{\tau_{12}^{\alpha}/d_{12}^{\beta}}{\sum_{l\in N_1^1}\tau_{1l}^{\alpha}/d_{1l}^{\beta}} = \frac{0.5/4}{0.5/4+0.5/3} = 0.43 \quad \textbf{10.7}$$

$$p_{13}^{1} = \frac{\tau_{13}^{\alpha}\eta_{13}^{\beta}}{\sum_{l\in N_1^1}\tau_{1l}^{\alpha}\eta_{1l}^{\beta}} = \frac{\tau_{13}^{\alpha}/d_{13}^{\beta}}{\sum_{l\in N_1^1}\tau_{1l}^{\alpha}/d_{1l}^{\beta}} = \frac{0.5/3}{0.5/4+0.5/3} = 0.57 \quad \textbf{10.8}$$

where p_{12}^{1} is the probability of node 2 being selected by ant 1 at node 1, and $p_{13}{}^{1}$ is the probability of node 3 being selected. We then generate a random number r between 0 and 1. If $p_{13}^{1} \geq r$, we select node 3; otherwise, we select 2. As node 3 has the highest probability of being selected, it will most likely be selected.

Moving forward, the first ant is now at node 3 and needs to decide between the adjacent nodes 2 and 4 following the same transition probability, which results in $p_{32}^{1} = 0.29$ and $p_{34}^{1} = 0.71$. Let's assume that node 4 is selected. The ant is now at node 4 and needs to decide between the adjacent nodes 2 and 5 following the same transition probability, which results in $p_{42}^{1} = 0.6$ and $p_{45}^{1} = 0.4$. Let's assume that node 5 is selected, based on the generated random number. Figure 10.9 shows the path completed by the first ant in the first iteration with length $L^{1}(t = 1) = 3 + 2 + 6 = 11$. Each ant will generate its own path following the same steps.

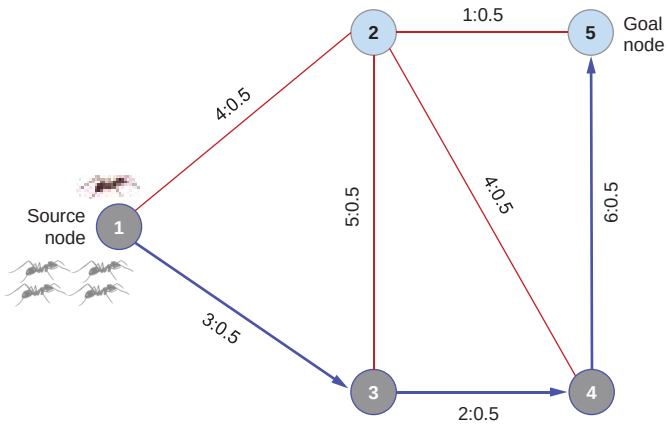

Figure 10.9 The path constructed by the first ant. Each of the other four ants will similarly construct a path.

Before starting a new iteration, the pheromones need to be updated. Following equation 10.2 and assuming that the evaporation rate ρ is 0.7, the new pheromone value will be

$$\tau(t+1) = (1-\rho) \times \tau(t) = (1-0.7) \times 0.5 = 0.15 \quad \textbf{10.9}$$

Pheromones are also deposited. If the first ant $k = 1$ is selected to deposit pheromones based on the costs of the paths found by each ant, it enforces the edges {1,3), {3,4}, and {4,5} with the value $Q/L^1 = 1/11$ following the online delayed pheromone update model. Figure 10.10 shows the updated pheromone values on each edge.

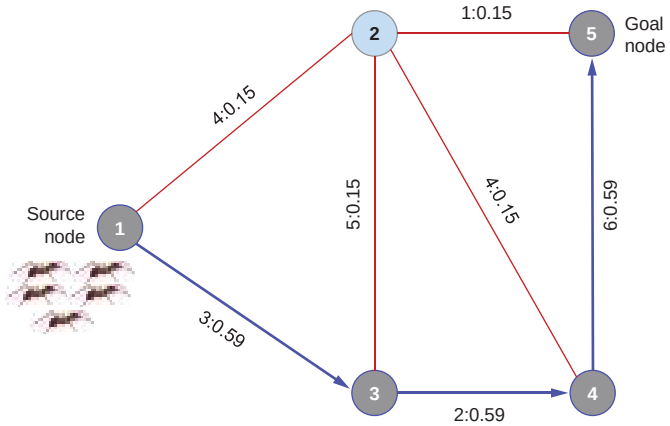

Figure 10.10 Updated pheromone concentrations

In this simple example, over three iterations, the ants find the shortest path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$. In the following sections, we'll discuss the ant system (AS) algorithm, ant colony system (ACS) algorithm, and max–min ant system (MMAS) algorithm as ACO variants proposed to deal with SACO limitations.

10.3.2 Ant system

The ant system (AS) algorithm improves on SACO by adding a memory capability via a tabu list. This list, or ant memory, identifies the already-visited nodes. The transition probability used in AS is the same as in equation 10.6. As an ant visits a new node, that node is added to the ant's tabu list for a predefined number of iterations. And as in SACO, after an ant completes a path, the pheromone on each edge is updated. The ant density, ant quantity, and ant cycle models can be used to update the pheromones. As previously explained, in the ant density and ant quantity models, ants deposit pheromones while building, whereas in the ant cycle model, ants deposit pheromones after they have built a complete path.

10.3.3 Ant colony system

The ant colony system (ACS) algorithm is an extension of the AS algorithm with a modified transition rule that utilizes an elitist strategy. This strategy, known as the *pseudo-random proportional action rule*, is designed to improve the efficiency and effectiveness of the algorithm. The pseudo-random proportional action rule used in ACS is based on the idea that the best solutions found by the ants should be given more weight in the decision-making process. In ACS, a random number r is generated, and the parameter $r_o \in [0,1]$ is predefined. An ant k, located at node i, selects the next node j to move to using the following decision rule with a double function:

- If $r \leq r_o$, the ant selects node j

$$j = \arg\max_{l \in N_i^k} \tau_{il}(t) \cdot [\eta_{il}(t)]^{\beta} \qquad \textbf{10.10}$$

- Else, a node is probabilistically selected (using a roulette wheel method, for example, which you learned about in chapter 7) according to the following transition probability:

$$p_{ij}^k(t) = \begin{cases} \frac{\tau_{ij}(t) \cdot [\eta_{ij}(t)]^{\beta}}{\sum_{l \in N_i^k} \tau_{il}(t) \cdot [\eta_{il}(t)]^{\beta}} & \text{if } j \in N_i^k \\ 0 & \text{if } j \notin N_i^k \end{cases}$$

Notice that compared to the transition probability of SACO (equation 10.6), the parameter that controls the influence of the pheromone concentration is $\alpha = 1$ in ACS. The parameter r_0 is used to balance the exploration–exploitation trade-off. When $r \leq r_0$, the decision rule exploits the knowledge available about the problem by favoring the best edge, and when $r > r_0$, the algorithm explores. Properly tuning r_0 allows us to strike a balance between exploration and exploitation.

In the previous shortest path example, assume that the ant is at node 4 and needs to choose node 2 or 5 following the ACS decision rule (figure 10.11). Let's assume that we have the values $r_o = 0.5$, $\beta = 1$, and $\eta_{ij} = 1 / d_{ij}$. Let's now generate a random number r.

If $r \leq r_o$, the ant will select node

$$\begin{aligned} j &= \arg\max_{l \in N_i^k} \tau_{il}(t) \cdot [\eta_{il}(t)]^{\beta} \\ &= \arg\max_{l \in N_i^k} \left\{ 0.5 \times \frac{1}{4}, 0.5 \times \frac{1}{3} \right\} \\ &= \arg\max\{0.125, 0.1666\} \Rightarrow node5 \end{aligned} \qquad \textbf{10.11}$$

If $r > r_o$, the ant will select a node with maximum transition probability: $p_{45}^1 = 0.6$ and $p_{42}^1 = 0.4$ as calculated before in section 10.3.1 with $\alpha = 1$. Using the roulette wheel method, node 2 or node 5 may be selected.

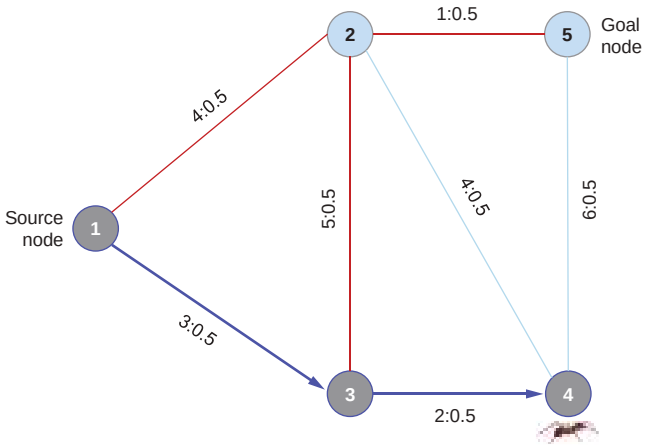

Figure 10.11 The ant at node 4 selects the next node (2 or 5) following the ACS elitist strategy.

Unlike in AS, the pheromone reinforcement process in ACS is exclusively performed by the ant with the global-best solution, which corresponds to the best path found so far. However, relying solely on the global-best solution to dictate pheromone deposition may cause the search to converge too rapidly around the global-best solution to date, hindering exploration of other potentially better solutions. The max–min ant system (MMAS) algorithm was developed to address this issue.

10.3.4 Max–min ant system

ACS can experience premature stagnation, which occurs when all ants follow the same path and little exploration is done. This issue is especially prevalent in complex problems, where the search space is large and the optimal solution is difficult to find. To overcome this problem, the max–min ant system (MMAS) was proposed.

MMAS employs the iteration-best path instead of the global-best path for pheromone updates. Pheromone trails are only updated using the online delayed pheromone update model, where the edges that were traversed by the best ant in the current iteration receive additional pheromones. Since the best paths can vary significantly between iterations, this approach promotes a higher degree of exploration throughout the search space compared to ACS. Hybrid strategies can also be implemented, in which the iteration-best path is primarily utilized to encourage exploration, while the global-best path is incorporated periodically.

In MMAS, the pheromone concentrations are constrained within an upper bound (τ_{max}) and lower bound (τ_{min}), ensuring that the search remains focused yet flexible. The pheromone trails are initialized to their maximum value τ_{max}, and if the algorithm

reaches a stagnation point, all pheromone concentrations are reset to the maximum value. Following this reset, the iteration-best path is exclusively used for a limited number of iterations. The values for τ_{min} and τ_{max} are typically determined through experimentation, although they could also be computed analytically if the optimal solution is known.

10.3.5 Solving open TSP with ACO

Let's now implement the ACO algorithm to solve open TSP, considering the 20 major US cities. Our objective is to find the shortest route that a salesperson can follow to visit each of these 20 cities once, starting from New York City and without returning to the home city.

We'll start by defining a `cities` dictionary that contains the names of the 20 US cities and their latitude and longitude coordinates. We'll then use a nested loop to calculate the distance between each pair of cities using the `haversine` distance formula, storing the results in the `distance_matrix` dictionary. The `haversine` distance is used because it takes into account the earth's curvature, providing accurate distance measurements between two points on the earth's surface (see the "Haversine distance" sidebar in section 4.3.3 for more details). The `cost_function` is defined to calculate the total distance of a path. It takes a list of city indices (`path`) and a distance matrix (`distances`) as input arguments. The function then iterates through the path, summing the distance between each consecutive pair of cities. The total path distance is then returned. This code is shown in the next listing.

Listing 10.1 Solving shortest path problem using ACO

```
import numpy as np
import pandas as pd
from collections import defaultdict
from haversine import haversine
import networkx as nx
import matplotlib.pyplot as plt
import random
from tqdm import tqdm

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)
city_indices = {city: idx for idx, city in enumerate(city_names)}
city_count = len(city_names)

def cost_function(path):
    distance = 0
    for i in range(len(path) - 1):
        city1, city2 = city_names[path[i]], city_names[path[i + 1]]
        distance += haversine(cities[city1], cities[city2])
    return distance
```

Define latitude and longitude for 20 major US cities.

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

Inter-city values

City names

Define the cost function that represents the path length.

As a continuation of listing 10.1, the next code snippet presents a function called `ant_tour` that takes two arguments: `pheromones`, representing the pheromone levels between cities, and `distances`, representing the distances between cities. It initializes a `paths` array to store paths for each ant, and it iterates over each ant in the specified range of ants. For each ant, it initializes a path starting from New York City. It enters a `while` loop that continues until all cities are visited. Within the `while` loop, it selects the current city as the last city in the path. It then calculates the probabilities for choosing the next city based on the pheromone levels and the inverse of the distances between the current city and unvisited cities. The probabilities are calculated using equation 10.6. The next city is chosen using the `random.choices` function, based on the normalized probabilities. The chosen next city is removed from the list of unvisited cities, and it is appended to the path:

```
def ant_tour(pheromones):
    paths = np.empty((ants, city_count), dtype=int)
    for ant in range(ants):
        path = [city_indices['New York City']]
        unvisited_cities = set(range(city_count))
        unvisited_cities.remove(path[0])

        while unvisited_cities:
            current_city = path[-1]
```

Initialize an array to store paths for each ant.

Start each ant's path from New York City.

Initialize a set of unvisited cities.

Remove New York City from unvisited cities.

Continue building the path until all cities are visited.

```
            probabilities = []

            for city in unvisited_cities:
                tau = pheromones[current_city, city]
                eta = (1 / distances[current_city, city])
                probabilities.append((tau** alpha)*(eta ** beta))

            probabilities /= sum(probabilities)
            next_city = np.random.choice(list(unvisited_cities),
p=probabilities)
            unvisited_cities.remove(next_city)
            unvisited_cities.remove(next_city)
            path.append(next_city)

        paths[ant] = path

    return paths
```

Calculate the probabilities for moving to each unvisited city.

Choose the next city based on probabilities.

Normalize the probabilities.

Remove the chosen city from the set of unvisited cities.

Add the chosen city to the path.

Store the completed path for the current ant.

Once all the cities have been visited, the path for the current ant is stored in the `paths` array. After all the ants have completed their paths, the function returns the `paths` array containing the optimal tours found by each ant.

The following `update_pheromones` function is used to update the pheromone levels based on the distances and paths of the ants:

```
def update_pheromones(paths, pheromones):
    delta_pheromones = np.zeros_like(pheromones)

    for i in range(ants):
        for j in range(city_count - 1):
            city1_idx, city2_idx = paths[i, j], paths[i, j + 1]
            delta_pheromones[city1_idx, city2_idx] += Q / cost_
function(paths[i])

    return (1 - evaporation_rate) * pheromones + delta_pheromones
```

Initialize a matrix to store the changes in pheromone levels.

Get the indices of the cities in the current path.

Update pheromones based on the paths taken by the ants.

Update the pheromone level between the current and next city.

Evaporate existing pheromones, add the changes in pheromones, and return the updated pheromones.

This function takes two arguments: `paths`, representing the paths taken by ants, and `pheromones`, representing the current pheromone levels on edges between cities. It initializes a matrix `delta_pheromones` to store the changes in pheromone levels. This matrix has the same shape as the `pheromones` matrix. It iterates over each ant in the specified range of ants. Within the loop, it iterates over each city in the ant's path (except the last city). For each pair of consecutive cities, it updates the `delta_pheromones` matrix by adding a value based on the inverse of the cost of the ant's path. After the inner loop, it calculates the updated pheromones by combining the existing pheromones, considering evaporation, and adding the changes stored in `delta_pheromones`. Finally, it returns the matrix of updated pheromones.

As a continuation, the following code snippet shows the `run_ACO` function, which takes the following inputs:

- `distances`—A 2D array (matrix) that stores the distances between cities
- `ants`—The number of ants to use in the algorithm
- `iterations`—The number of iterations
- `alpha`—A parameter that controls the influence of the pheromone trail on the ant's decision
- `beta`—A parameter that controls the influence of the distance to the next city on the ant's decision
- `evaporation_rate`—The rate at which pheromones evaporate from the paths
- `Q`—A constant used in the calculation of the amount of pheromones deposited by the ants

This function returns the `best_path` and `best_distance`, representing the optimal solution found by the ACO algorithm:

```
def run_ACO(distances, ants, iterations, alpha, beta, evaporation_rate, Q):
    pheromones = np.ones((city_count, city_count))
    best_path = None
    best_distance = float('inf')

    for _ in tqdm(range(iterations), desc="Running ACO", unit="iteration"):
        paths =ant_tour(pheromones, distances)
        distances_paths = np.array([cost_function(path) for path in paths])
        min_idx = distances_paths.argmin()
        min_distance = distances_paths[min_idx]

        if min_distance < best_distance:
            best_distance = min_distance
            best_path = paths[min_idx]

        pheromones = update_pheromones(paths, pheromones)

    return best_path, best_distance
```

Initialize the pheromones array.

Generate paths for each ant.

Calculate the total distance for each path.

Find the index of the path with the minimum distance.

Return the best path and distance found during the iterations.

Update the pheromones.

Let's now apply ACO to solve the shortest path problem using the following parameters:

```
ants = 30
iterations = 100
alpha = 1
beta = 0.9
evaporation_rate = 0.5
Q = 100

best_path, best_distance = run_ACO(distances.values, ants, iterations, alpha,
beta,
➥ evaporation_rate, Q)
```

Set ACO parameters.

Run ACO with the defined parameters.

Given the randomness included in the algorithm, your solution may vary. The following path is what was generated when I ran the solver:

```
Route: New York City → Philadelphia → Baltimore → Detroit → Chicago →
Indianapolis → Columbus → Charlotte → Jacksonville → Memphis → Fort Worth →
Dallas → Houston → Austin → San Antonio → Phoenix → San Diego → Los Angeles →
San Jose → San Francisco
Route length: 7937.115
```

The preceding path is shown in figure 10.12. The complete version of listing 10.1 is available in the book's GitHub repo, which also contains the code to generate this visualization.

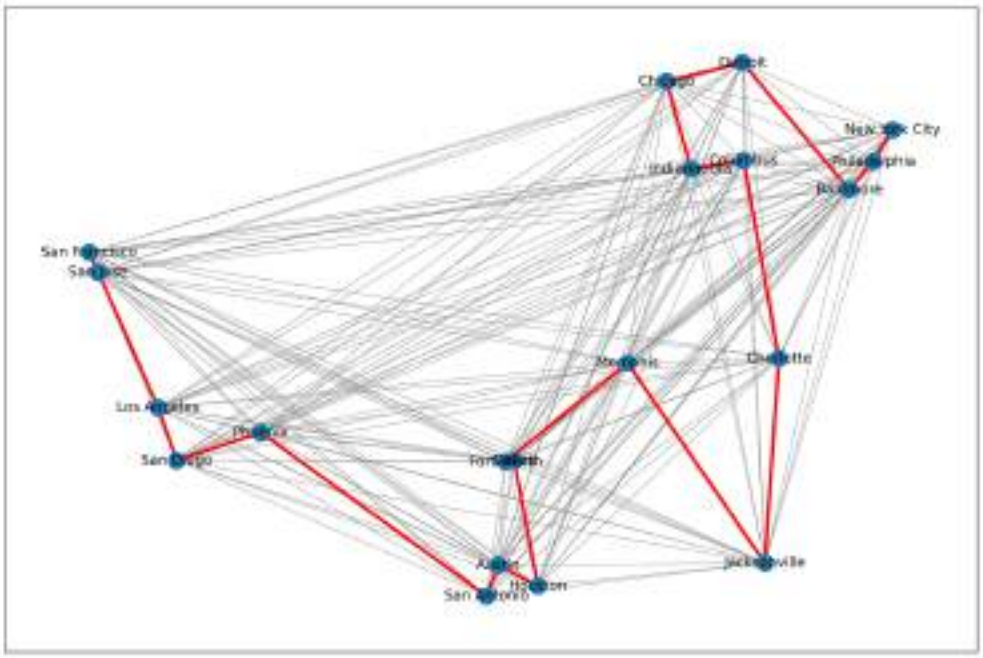

Figure 10.12 Shortest path obtained by ACO

Unlike genetic algorithms and particle swarm optimization algorithms, there are no well-developed and comprehensive Python packages for ACO metaheuristics. The ACOpy project (https://acopy.readthedocs.io/en/latest/index.html) provides an implementation of ACO and can be installed using pip as follows: `pip install acopy`. As a continuation of listing 10.1, let's use ACOpy to solve the shortest path problem.

We'll start by importing the `acopy` and `networkx` libraries. A graph, `G`, is created where the nodes represent cities and the edges represent the distances between them. The `distance_matrix` contains the distances between each pair of cities. The loops iterate over all pairs of cities, adding an edge between each pair of cities to the graph, with the weight of the edge being the distance between the cities. Self-loop edges (edges that connect a node to itself) are then removed from the graph:

```
import acopy
import networkx as nx

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))
```

The parameters for the ACO algorithm are then defined: `evaporation_rate`, `iterations`, and `Q`, as explained previously. An ACO solver is created with the specified `evaporation_rate` and `Q`. The `acopy.Colony` object is initialized with `alpha` and `beta` parameters. The algorithm then iterates for the specified number of iterations. In each iteration, the solver's `solve` method is used to find a tour whose path is a list of edges. For each edge in the path, the code determines the city that hasn't been added to the `path_indices` list yet and adds it. Finally, the path of the tour is updated to be the `path_indices` list, which is a list of city names instead of edges:

```
evaporation_rate = 0.5
iterations = 100
Q = 100

solver = acopy.Solver(rho=evaporation_rate, q=Q)

colony = acopy.Colony(alpha=1, beta=0.9)

for n_iter in range(iterations):
    tour = solver.solve(G, colony, limit=4)
    path_indices = ['New York City']
    for edge in tour.path:
        next_city = edge[0] if edge[1] == path_indices[-1] else edge[1]
        if next_city not in path_indices:
            path_indices.append(next_city)

tour.path=path_indices
```

ACO parameters

Set up the ACO solver.

Set up the ACO colony with alpha and beta parameters.

Run the ACO algorithm.

Start with city 0 (New York City).

Add the other node from the edge that is not already in the path.

Return the ordered list of city names included in the path.

Let's now print the obtained path and its length as follows:

```
best_path = tour.path
best_distance = tour.cost
Route = " → ".join(best_path)
print("Route:", Route)
print("Route length:", np.round(best_distance, 3))
```

The `best_path` variable is set to the `path` property of the `tour` object obtained by the `acopy` solver. This path is a list of cities that represents the shortest route found. The `best_distance` variable is set to the `cost` property of the `tour` object, which is the total distance (or cost) of the best path. The `Route` variable is a string that joins all the cities in `best_path` with an arrow (→) in between, representing the sequence of cities to visit

in the optimal tour. Finally, the `print` statements display the best route and its total distance. A path like the following will be generated after running the solver:

```
Route: New York City → Columbus → Detroit → Philadelphia → Baltimore →
Charlotte → Jacksonville → Memphis → Houston → Dallas → Fort Worth → Austin →
San Antonio → Phoenix → San Diego → Los Angeles → San Jose → San Francisco →
Chicago → Indianapolis
Route length: 11058.541
```

The obtained path is shown in figure 10.13. The complete version of listing 10.1 available in the book's GitHub repo contains the code to generate this visualization.

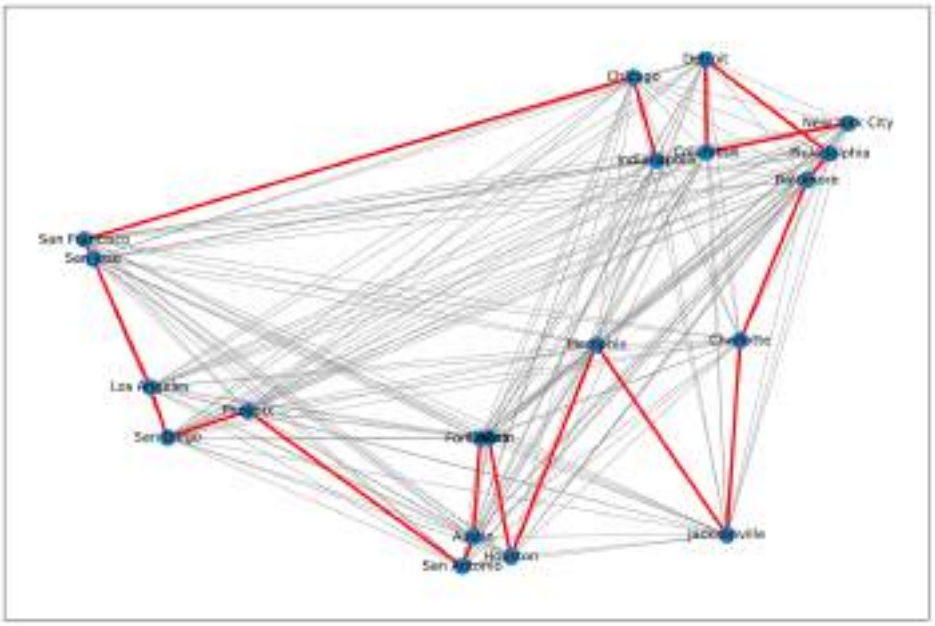

Figure 10.13 Shortest path obtained by ACOpy

It's worth mentioning that ACO, like many other stochastic optimization algorithms, contains elements of randomness. The randomness in ACO comes from two main sources:

- *Initial conditions*—At the start of the algorithm, the ants are usually placed at random positions unless the start position is predefined, such as in the case of TSP where ants start from a predetermined home city. This means that in scenarios where random positions are used, each ant starts exploring from a different city, leading to diverse paths.
- *Path selection*—As the ants move from city to city, they probabilistically choose which city to visit next. This choice is influenced by the amount of pheromones on the path to a city and the distance to the city. Even if two ants are in the same city and have the same information, they might still choose different cities to visit next due to this probabilistic choice.

This inherent randomness means that each run of the ACO algorithm can produce different results. However, over multiple runs, ACO should consistently find near-optimal solutions, even if they are not always the exact same solution.

In the following section, we will delve into another fascinating algorithm that is a product of swarm intelligence. This algorithm again takes its inspiration from the natural world, specifically the food-seeking behavior of honeybees. You'll soon understand how this bee-inspired algorithm operates and how it can be applied in a computational context.

10.4 From hive to optimization

Honeybees are remarkable social insects known for their extraordinary cooperation. They build hives capable of accommodating approximately 30,000 bees, all working together harmoniously. Each bee has a designated task, such as producing wax, creating honey, making bee-bread, forming combs, or bringing water to the cells and mixing it with honey. Young bees typically handle tasks outside the hive, while elder bees focus on indoor duties.

Honeybee colonies operate as goal-oriented decision-making systems, with their functions directed by the decentralized control and actions of individual bees. The cooperation between honeybees during the foraging process leads to advantageous behaviors that optimize the hive's overall fitness. By using individual foragers, honeybee colonies aim to minimize the cost/benefit ratio, rather than expending energy searching in all directions indiscriminately. They concentrate their foraging efforts on the most rewarding patches, while disregarding those of lesser quality.

Observations have shown that when colony food resources are scarce, foragers exhibit increased recruitment behaviors, characterized by changes in their dance patterns upon returning to the hive. This enhanced recruitment serves to mobilize more nestmates to exploit available food sources. In addition to foraging, honeybees also cooperate in various other tasks, such as hive construction, hive thermoregulation, and colony defense, showcasing their exceptional teamwork skills.

Discover the fascinating world of honeybees

- Honeybees are the most well-known and important insects that produce food consumed by humans.
- Honeybee colonies consist of a single queen, hundreds of male drones, and 20,000 to 80,000 female worker bees.
- A single worker bee may visit 50 to 1,000 flowers per day. Bees from the same hive can visit up to 225,000 flowers in one day. Honeybees can fly at speeds of 21 to 28 km/h (13–17 mph) and can have a foraging area up to 70 km^2 (27 mi^2).
- Honeybees can maintain a constant temperature of about 33°C (91°F) in their hive, regardless of the outside temperature.
- Honeybees choose the hexagonal shape for their honeycomb cells to hold the queen bee's eggs and store the pollen and honey the worker bees bring to the hive.

(continued)

- The hexagonal structure has several advantages, such as efficient use of space (creating the maximum number of cells that can be built in a given area), structural strength (it's strong and stable), material efficiency (it uses less beeswax), and optimal angle (a slight tilt, ~13 degrees from horizontal, to prevent honey from spilling out of the cells while still allowing bees to move around easily).
- Honeybees communicate with each other through complex dance moves called "waggle dances," explained in "The Waggle Dance of the Honeybee," a video from Georgia Tech College of Computing (http://mng.bz/gvxx).

The artificial bee colony (ABC) algorithm is a swarm intelligence algorithm based on the foraging behavior of honeybees. Specifically, it is inspired by the way honeybees search for food sources and communicate their findings to optimize the gathering of resources. Let's first look at how honeybees forage for food. Figure 10.14 illustrates the steps of their foraging behavior.

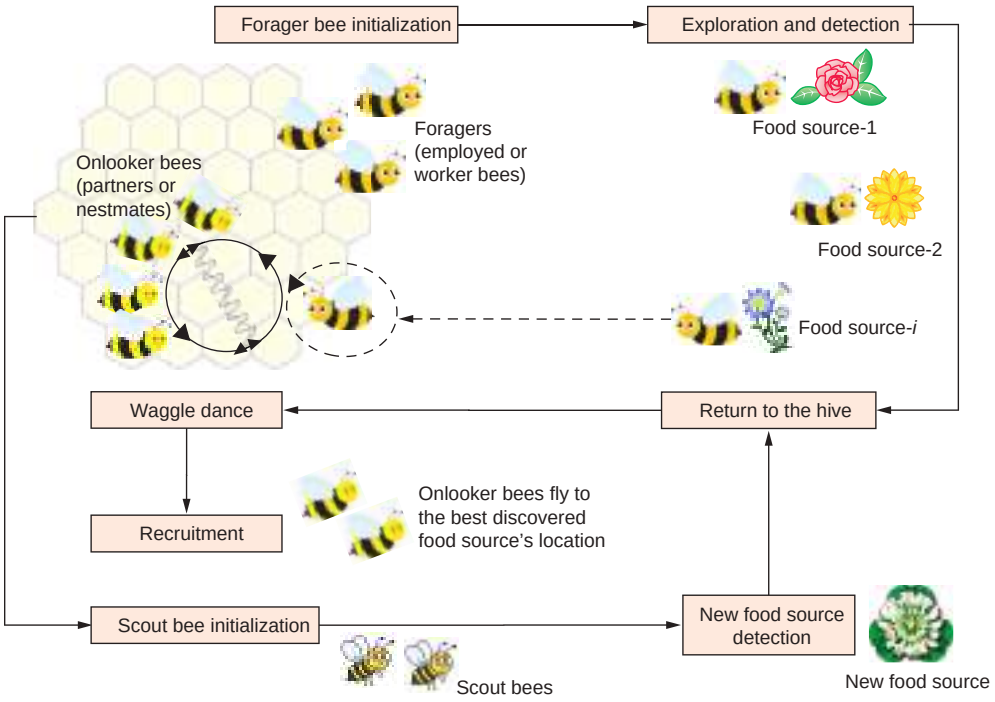

Figure 10.14 Foraging behavior of honeybees

The foraging behavior can be summarized in the following steps:

1. *Initialization*—Forager bees (employed bees) and scout bees begin their foraging for food sources. Forager bees usually gather resources from known sources around the hive to meet the colony's immediate needs. Scout bees locate new food sources to ensure the colony's long-term survival, especially if the food sources around the hive start to deplete. Scout bees only represent a small percentage of colony members, but they save the colony many wasted miles of flight trying to locate abundant new food sources. It is worth noting that forager bees and scout bees are both worker bees (female bees). A worker bee can switch roles from being a forager to a scout, depending on the colony's needs and food source availability. In summary, forager bees concentrate on exploiting the available resources while scout bees focus on exploring to discover new resources.
2. *Exploration*—Forager bees leave the hive and start searching for food sources, such as flowers with nectar and pollen, in the surrounding area. Scout bees explore areas farther away to discover new food sources.
3. *Detection*—When a suitable food source is found, the worker bee lands on the flower and begins to collect nectar in her honey stomach or gathers pollen on her hind legs.
4. *Memorization*—The bee takes note of the food source's location, including its distance and direction from the hive, as well as the flower type and quality.
5. *Return to the hive*—Once the worker bee has collected enough resources or her honey stomach is full, she flies back to the hive. Upon reaching the hive, the forager bee transfers the nectar to a house bee, who then processes and stores it as honey. Pollen is similarly offloaded to other bees for storage and later use as food.
6. *Communication*—The worker bee performs a waggle dance on the hive's dance floor to share the location information with her nestmates (aka *onlooker bees*). The dance communicates the direction, distance, and quality of the food source.
7. *Recruitment*—Onlooker bees observe the waggle dance and decode the information about the food source's location. These bees then fly out to collect the resources.
8. *Repeat*—The worker bee continues to visit the same food source until it is depleted or another bee recruits her to a more promising source. In either case, she repeats the foraging process to ensure the colony's needs are met.

Now let's look at the ABC algorithm in more detail.

10.5 Exploring the artificial bee colony algorithm

The artificial bee colony (ABC) algorithm, proposed by Dervis Karaboga in 2005 [1], simulates the roles of three types of bees: employed bees (foragers), onlooker bees, and scout bees. Algorithm 10.1 shows the steps of the ABC algorithm.

Algorithm 10.1 Artificial bee colony algorithm

```
Initialization Phase: population of candidate solutions (food sources) are
initialized
REPEAT
   Forager Bee Phase: Each forager bee goes to a food source in her memory
and determines a closest source, then evaluates its nectar amount and dances
in the hive
   Onlooker Bee Phase: Each onlooker bee watches the dance of forager bees
and chooses one of their sources depending on the dances, and then goes to
that source. After choosing a neighbor around that, she evaluates its nectar
amount.
   Scout Bee Phase: Abandoned food sources are determined and are replaced
with the new food sources discovered by scout bees.
   Memorize the best food source (solution) achieved so far.
UNTIL (termination criteria are met)
```

As you can see, the ABC algorithm simulates the honeybee foraging behaviors to explore and exploit the search space, balancing global exploration (diversity) and local exploitation (convergence) to efficiently solve optimization problems. In the ABC algorithm, the three types of bees have the following complementary roles:

- *Employed bees (foragers)*—These bees exploit the current food sources, meaning they search around their current position (searching the neighborhood) to find better solutions. These bees perform a local search (intensification), which refines the current best solutions.
- *Onlooker bees*—These bees also contribute to exploitation. They probabilistically choose food sources depending on the fitness of the solutions found by the employed bees. They are more likely to choose better solutions (food sources with more nectar) for further exploitation.
- *Scout bees*—These bees perform the exploration. If a food source is exhausted (if the solution cannot be improved after a certain number of iterations), the employed bee associated with that food source becomes a scout bee. Scout bees perform a global search (diversification) by abandoning the exhausted food source and randomly searching for new food sources in the problem space. This process prevents the algorithm from getting stuck in local optima by exploring new regions of the search space.

In the ABC algorithm, communication between the bees is simulated by sharing the fitness values of solutions among employed and onlooker bees, guiding them toward better solutions. The ABC algorithm adopts a fitness-proportionate selection process inspired by how bees choose food sources based on their quality. In the algorithm, employed bees and onlooker bees select solutions with a probability proportional to their fitness, promoting better solutions to be explored more frequently.

To understand how we can use ABC to solve optimization problems, let's consider minimizing the Rosenbrock function using ABC. The Rosenbrock function, also

referred to as the *valley* or *banana function*, is a popular test problem for gradient-based optimization algorithms. This function has n dimensions and takes the following general form:

$$f(x) = \sum_{i=1}^{n-1} \left(100 \times \left(x_{i+1} - x_i^2 \right)^2 + (x_i - 1)^2 \right) \quad \textbf{10.12}$$

The function is usually evaluated on the hypercube $x_i \in [-5, 10]$ for all $i = 1,...,n$, but the domain may be restricted to $x_i \in [-2.048, 2.048]$ for all $i = 1,...,n$. This function has a global minimum at $f(x^*) = 0.0$ located at $(1,...,1)$.

Let's consider the 2D Rosenbrock function that takes the following form:

$$f(x, y) = 100 \times \left(y - x^2 \right)^2 + (1 + x)^2 \quad \textbf{10.13}$$

Figure 10.15 shows the 2D surface of Rosenbrock function.

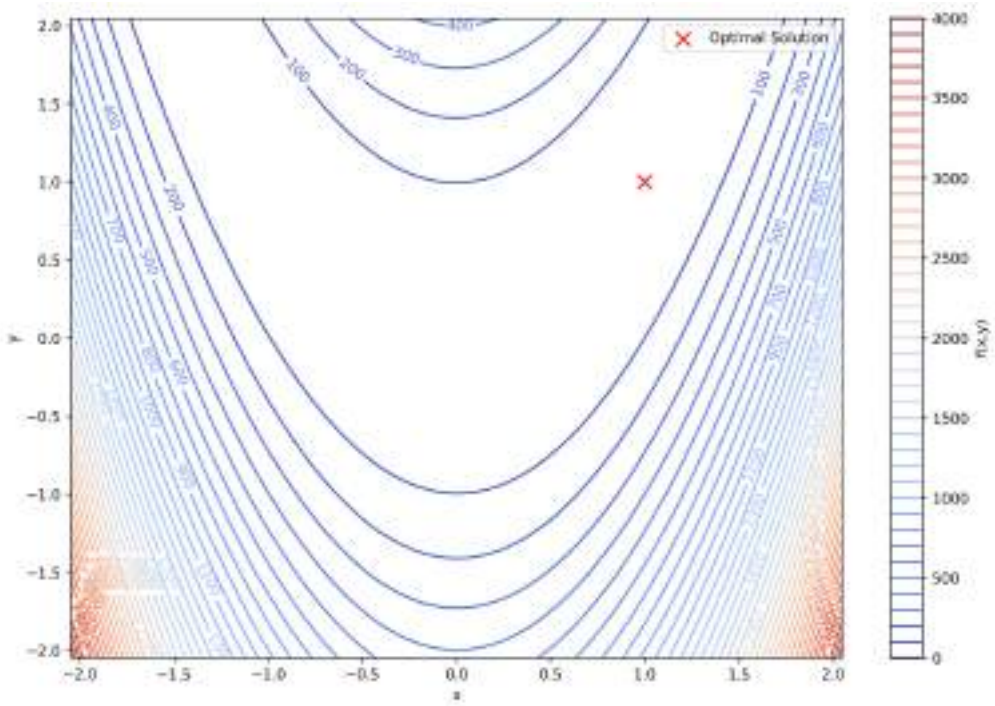

Figure 10.15 The 2D surface plot of the Rosenbrock function. The dot indicates the global minimum of this function.

Let's look at how we can minimize this function using ABC:

- *Initialization phase*—Let's assume that we have a swarm of $N = 6$ bees. Each bee tries to find a candidate solution, and each solution i in the population consists of a position vector $X_{mi} = \{x_{mi}, y_{mi}\}$ where $X_{mi} \in [-2.048, 2.048]$ and $m = 1,\ldots,N$. X_{mi} represents a potential solution to the optimization problem. The position of the employed bees is randomly determined within the boundaries. These initial solutions can be generated using the following formula:

$$x_{mi} = l_i + rand(0, 1) \times (u_i - l_i) \quad \textbf{10.14}$$

where l_i and u_i are the lower and upper bounds of the decision variables. Let's assume the initial positions (represented as (x, y)) shown in table 10.1.

Table 10.1 Initial food sources

Candidate solution X_m	Objective function $f_m(X_m)$
X_1 = (–1.04,0.11)	98.56
X_2 = (–1.61,–1.98)	2097.22
X_3 = (1.82,1.22)	438.49
X_4 = (–1.64,1.92)	66.20
X_5 = (0.77,0.04)	30.62
X_6 = (–0.66,1.59)	136.02

- *Employed bee phase*—In the employed bee phase, each bee generates a new solution in the neighborhood of its current solution using the following formula:

$$v_{mi} = x_{mi} + \phi_{mi} \times (x_{mi} - x_{ki}) \quad \textbf{10.15}$$

where v_{mj} is the new solution, x_{mi} is the current solution, ϕ_{mi} is a random number between –1 and 1, and x_{ki} is a randomly chosen solution different from the current solution. Let's assume that all ϕ_{mi} are –0.9 for simplicity, and for each bee, we choose the solution of bee 1 to calculate the new solutions. The best bee, in the initial population (table 10.1), bee 5, can also be used. We then calculate the new fitness values shown in table 10.2.

Table 10.2 New food sources

Candidate solution X_m	Objective function $f_m(X_m)$
X_1 = (–1.04,0.11)	98.56
X_2 = (–1.10,–0.10)	174.02
X_3 = (–0.75,0.22)	15.15
X_4 = (–1.10,0.29)	88.87
X_5 = (–0.86,0.10)	43.76
X_6 = (–1.00,0.26)	59.66

- *Onlooker bee phase*—The onlooker bees observe the dance of the employed bees and choose a food source depending on the nectar amount (the *fitness value*). If the new solution has a better fitness value, it is remembered as a global variable, and the position is updated. Otherwise, the old position is retained. The probability value p_m, with which X_m is chosen by an onlooker bee, can be calculated by using the following formula:

$$p_m = \frac{fit_m(X_m)}{\sum_{m=1}^{N} fit_m(X_m)} \qquad \textbf{10.16}$$

where $fit_m(X_m)$ is the fitness value of the solution, which can be calculated using the following expression:

$$fit_m(X_m) = \begin{cases} \frac{1}{1+f_m(X_m)} & \text{if } f_m(X_m) \geq 0 \\ 1 + \text{abs}(f_m(X_m)) & \text{otherwise} \end{cases} \qquad \textbf{10.17}$$

where $f_m(X_m)$ is the objective function of solution X_m. Table 10.3 shows the solution fitness calculations.

Table 10.3 Solution fitness calculations

Candidate solution X_m	Objective function $f_m(X_m)$	Fitness $fit_m(X_m)$	Probability of selection p_m
X_1 = (–1.04,0.11)	98.56	0.010	0.08
X_2 = (–1.10,–0.10)	174.02	0.006	0.04
X_3 = (–0.75,0.22)	15.15	0.062	0.49
X_4 = (–1.10,0.29)	88.87	0.011	0.09
X_5 = (–0.86,0.10)	43.76	0.022	0.18
X_6 = (–1.00,0.26)	59.66	0.016	0.13

In this example, the food source discovered by bee 3 is most likely to be chosen. After a food source X_m for an onlooker bee is probabilistically chosen, a neighborhood source v_m is determined by using equation 10.15, and its fitness value is computed.

- *Scout bee phase*—If a position cannot be improved further through a predetermined number of cycles or trials (called the *limit*), that position is abandoned and the bee becomes a scout, searching for a new random position, which can be generated by equation 10.14.

Let's now see how we can implement ABC in Python to solve this problem. In the next listing, we start by importing the libraries we'll use and defining the `rosenbrock_function`. This function takes as an argument a candidate solution (x, y) to the Rosenbrock function and returns its value.

Listing 10.2 Solving Rosenbrock function optimization using ABC

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

def rosenbrock_function(cand_soln):
    return (1 - cand_soln[0]) ** 2 + 100 * (cand_soln[1] - cand_soln[0] ** 2)
** 2
```

As a continuation of listing 10.2, we'll create a `Bee` that contains the following attributes:

- `position`—The position of the bee in the search space (solution)
- `fitness`—The fitness of the bee's current position (the value of the Rosenbrock function at the current position)
- `counter`—A counter to track the number of unsuccessful trials (iterations without improvement in the bee's fitness):

```
class Bee:
    def __init__(self, position, fitness):
        self.position = position
        self.fitness = fitness
        self.counter = 0
```

Now we need a function to generate a `Bee` with a random position and calculate its fitness using the Rosenbrock function:

```
def generate_bee(dimensions):
    position = np.array([random.uniform(-5, 5) for _ in range(dimensions)])
    fitness = rosenbrock_function(position)
    return Bee(position, fitness)
```

The following function will update the position of a given bee using the position of a partner bee. If the new position has a better fitness value, the bee's position, fitness, and counter are updated. Otherwise, the counter is incremented:

```
def update_position(bee, partner, dimensions):
    index = random.randrange(dimensions)
    phi = random.uniform(-1, 1)
    new_position = bee.position.copy()
    new_position[index] += phi * (bee.position[index] - partner.
position[index])
    new_position = np.clip(new_position, -5, 5)
    new_fitness = rosenbrock_function(new_position)
    if new_fitness < bee.fitness:
        bee.position = new_position
        bee.fitness = new_fitness
        bee.counter = 0
    else:
        bee.counter += 1
```

Determine which element of the bee's position will be updated.

Clip to ensure it stays within a specified range.

Next, we'll define an `abc_algorithm` function to implement the ABC algorithm with the following input parameters:

- `dimensions`—The number of dimensions of the problem, which is 2 for the Rosenbrock function
- `num_bees`—The total number of bees in the colony
- `max_iter`—The maximum number of iterations the algorithm should run
- `max_trials`—The maximum number of unsuccessful cycles or trials (iterations without improvement) allowed before a bee becomes a scout bee:

```
def abc_algorithm(dimensions, num_bees, max_iter, max_trials):

    bees = [generate_bee(dimensions) for _ in range(num_bees)]
    best_bee = min(bees, key=lambda bee: bee.fitness)

    for _ in range(max_iter):
        for i in range(num_bees // 2):
            employed_bee = bees[i]
            partner_bee = random.choice(bees)
            update_position(employed_bee, partner_bee, dimensions)

        total_fitness = sum(1 / (1 + bee.fitness) if bee.fitness >= 0 else 1
➥ + abs(bee.fitness) for bee in bees)
        probabilities = [(1 / (1 + bee.fitness)) / total_fitness if bee.
➥ fitness >= 0 else (1 + abs(bee.fitness)) / total_fitness for bee in bees]

        for i in range(num_bees // 2, num_bees):
            onlooker_bee = random.choices(bees, weights=probabilities)[0]
            partner_bee = min(bees[:num_bees // 2], key=lambda bee: bee.
fitness)
            update_position(onlooker_bee, partner_bee, dimensions)

        for bee in bees:
            if bee.counter > max_trials:
                new_bee = generate_bee(dimensions)
                bee.position = new_bee.position
                bee.fitness = new_bee.fitness
                bee.counter = 0

        best_iter_bee = min(bees, key=lambda bee: bee.fitness)
        if best_iter_bee.fitness < best_bee.fitness:
            best_bee = best_iter_bee

    return best_bee
```

Find the bee with the best fitness value.

Generate an initial population of bees.

Perform the employed bees phase.

Perform the onlooker bees phase.

Calculate the selection probability according equations 16 and 17.

Perform the scout bees phase.

Check if each bee's counter exceeds max_trials.

Update best_bee with the new best bee.

Return best_bee, which represents the optimal solution.

Now we can set up the parameters of the ABC algorithm and apply it to solve the problem:

```
dimensions = 2
num_bees = 50
max_iter = 1000
max_trials = 100
```

Define the problem dimensions.

Set the maximum number of iterations used as a stopping criterion.

Set the number of bees.

Set the maximum number of unsuccessful trials (iterations without improvement) allowed before a bee becomes a scout bee.

```
best_bee = abc_algorithm(dimensions, num_bees, max_iter, max_trials)
print(f"Best solution: {best_bee.position}")
print(f"Best fitness: {best_bee.fitness}")
```

Run the ABC algorithm with the parameters specified and store the best bee (the one with the minimum fitness value) in the best_bee variable.

Print the position of best_bee, which represents the solution and its fitness, which is the value of the Rosenbrock function at the best solution.

This code will produce output like the following:

```
Best solution: [0.99766117 0.99542949]
Best fitness: 6.50385257086524e-06
```

In contrast to genetic algorithms and particle swarm optimization algorithms, the availability of well-established and comprehensive Python packages specifically designed for the ABC algorithm is relatively limited. However, there is a Python library called MEALPY that offers implementations of population-based metaheuristic algorithms, including ABC. You can install MEALPY using `pip install mealpy`.

As a continuation of listing 10.2, the following code snippet demonstrates using the `OriginalABC` class from the MEALPY library to minimize the Rosenbrock function:

```
from mealpy.swarm_based.ABC import OriginalABC

problem_dict = {
    "fit_func": rosenbrock_function,
    "lb": [-5, -5],
    "ub": [5, 5],
    "minmax": "min",
}

epoch = 200
pop_size = 50
n_limits = 15

model = OriginalABC(epoch, pop_size, n_limits)

best_position_mealpy, best_fitness_mealpy = model.solve(problem_dict)

print(f"Best solution: {best_position_mealpy}")
print(f"Best fitness: {best_fitness_mealpy}")
```

Import the solver from MEALPY library.

Define the problem using dictionary.

Set the number of epochs (iterations).

Set the population size.

Set the limit on the number of unsuccessful trials before a scout bee is triggered.

Create an instance of the algorithm class.

Run the algorithm.

Print the results.

We start by importing the `OriginalABC` class from the `mealpy.swarm_based.ABC` module, which is the implementation of the ABC algorithm provided by the MEALPY library. We then define the problem dictionary, which contains the cost function (`fit_func`), lower bound (`lb`), upper bound (`ub`), and whether this is a minimization or maximization problem (`minmax`). The number of epochs (iterations), population size, and the limit on the number of unsuccessful trials before a scout bee is triggered are set. We then create an instance of the `OriginalABC` class, initialized with the specified parameters. The `solve()` method is called on the `model` object, passing the `problem_dict` as an argument. It performs the ABC algorithm optimization process on the defined problem and returns the best solution and fitness value.

Running this code will produce a solution like the following:

```
Best solution: [1.07313697 1.04914444]
Cost at best solution: 0.0009197449137428784
```

Figure 10.16 shows the solution obtained by the ABC solver, ABC MEALPLY, and the ACO solver implemented as part of the complete listing 10.2.

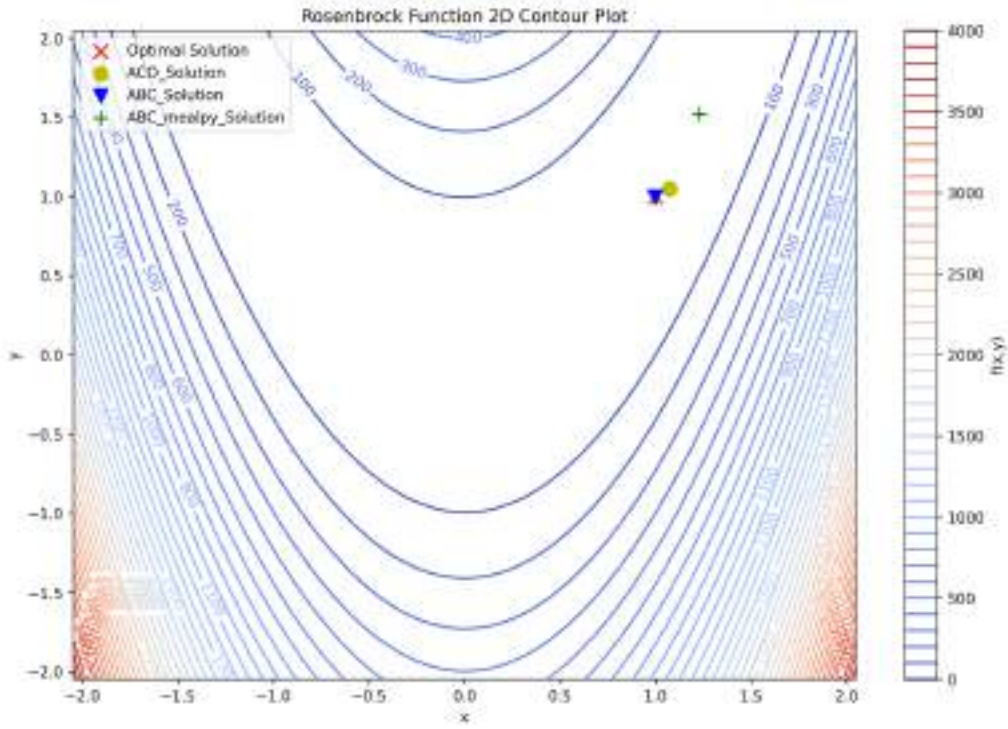

Figure 10.16 The Rosenbrock function contour and solutions using the ABC and ACO algorithms

As you can see, the ABC, ABC MEALPY, and ACO solutions are all close to the optimal solution of this function. With parameter tuning, an optimal solution can be reached by these algorithms. You can use the code in listing 10.2 to experiment with different algorithm parameter settings and different problem dimensions.

This chapter concludes the fourth part of this book. In this part, we've delved deep into the fascinating world of swarm intelligence, exploring how simple entities, like birds in particle swarm optimization (PSO), ants in ant colony optimization (ACO), and bees in the artificial bee colony (ABC) algorithm, can collectively perform complex tasks. These nature-inspired algorithms elegantly balance exploration and exploitation to find optimal or near-optimal solutions to complex optimization problems.

As we move forward, we'll transition to the domain of machine learning. In the last part of this book, we'll look at machine learning methods specifically tailored for search and optimization. We'll explore cutting-edge techniques, such as graph neural networks, attention mechanisms, self-organizing maps, and reinforcement learning, and investigate their applications in search and optimization.

Summary

- Ant colony optimization (ACO) is a population-based algorithm inspired by the foraging behavior of ants. Simple ACO (SACO), ant system (AS), ant colony system (ACS), and max–min ant system (MMAS) are examples of ACO metaheuristics algorithms.
- During foraging, ants discover good solutions, which influence the decisions of other ants. Over time, the pheromone trails intensify along the paths of better solutions, attracting more ants to explore those paths. This is called autocatalytic behavior.
- Pheromone updates include two phases: evaporation and deposit. During the evaporation phase, the pheromone concentration is decreased. Ants can deposit pheromones during the construction of a solution, using the online step-by-step pheromone update method, or after the solution has been built, by revisiting all the states visited during the construction process, using the online delayed pheromone update method. In some cases, both methods can be used together.
- Ant system (AS) improves on simple ACO by adding a memory capability in the form of a tabu list.
- The ant colony system (ACS) algorithm is an extension of the AS algorithm with a modified transition rule that utilizes an elitist strategy.
- The max–min ant system (MMAS) addresses the limitations of AS and ACS by using the iteration-best path for pheromone updates, encouraging exploration and constraining pheromone values between minimum and maximum values. This approach reduces the risk of premature stagnation and improves performance by balancing exploration and exploitation.
- The artificial bee colony (ABC) algorithm is a population-based search algorithm inspired by the foraging behavior of honeybees. The ABC algorithm manages the balance between exploration and exploitation through its three types of bees (employed bees, onlooker bees, and scout bees), each of which perform different complementary roles.
- The inherent randomness in stochastic optimization algorithms due to initial conditions and the probabilistic decision-making process is not necessarily a bad thing. It can help the algorithm avoid getting stuck in local optima—solutions that are the best in their immediate vicinity but are not the best overall. By occasionally taking less promising paths, the algorithm can explore more of the solution space and has a better chance of finding the global optimum—the best possible solution.