5 Simulated annealing

This chapter covers

- Introducing trajectory-based optimization algorithms
- Understanding the simulated annealing algorithm
- Solving function optimization as an example of continuous optimization problems
- Solving puzzle game problems like Sudoku as an example of constraint-satisfaction problems
- Solving permutation problems like TSP as an example of discrete problems
- Solving a real-world delivery semi-truck routing problem

In this chapter, we'll look at simulated annealing as a trajectory-based metaheuristic optimization technique. We'll discuss different elements of this algorithm and its adaptation aspects. A number of case studies will be presented to show the ability of this metaheuristic algorithm to solve continuous and discrete optimization problems.

5.1 Introducing trajectory-based optimization

Imagine yourself on a hiking trip looking for the lowest valley in a rugged landscape that has many valleys and hills. You don't have access to global information or a map that shows the location of the lowest valley. You start your hiking journey by randomly choosing a direction. You keep moving step by step until you get stuck in a local valley surrounded by hills. You are not highly satisfied with the location, as you believe that there is a lower valley in the area that may be behind the hills. Your curiosity drives you to move up one of the hills to find the lowest valley.

This is exactly what simulated annealing (SA) does. The basic idea of the SA algorithm is to use a stochastic search that follows a trial-and-error approach, accepting changes that improve the objective function and also keeping some changes that are not ideal. In a minimization problem, for example, any better moves or changes that decrease the value of the objective function will be accepted. However, some moves that increase the objective function will also be accepted with a certain probability. SA is a trajectory-based metaheuristic algorithm that can be used to find the global optimum solution for complex optimization problems.

Generally speaking, *metaheuristic algorithms* can be classified into *trajectory-based* and *population-based* algorithms, as shown in figure 5.1.

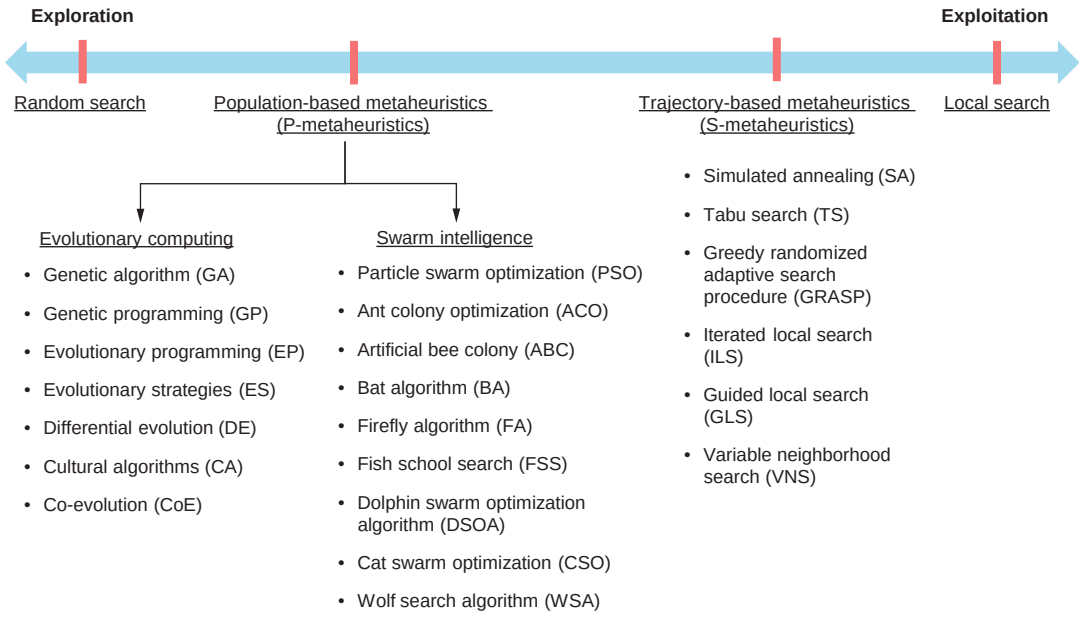


Figure 5.1 Exploration and exploitation of optimization algorithms

Trajectory-based metaheuristic algorithms or *S-metaheuristics*, such as SA or tabu search, use a single search agent that moves through the search space in a piecewise style. A better move or solution is always accepted, while a not-so-good move can be accepted with a certain probability. The steps, or moves, trace a trajectory in the search space, with a nonzero probability that this trajectory can reach the global optimum.

In contrast, *population-based algorithms* or *P-metaheuristics*, such as genetic algorithms, particle swarm optimization, and ant colony optimization, use multiple agents to search for an optimal or near-optimal global solution.

Due to the large diversity of initial populations, population-based algorithms are naturally more exploration-based, whereas single or trajectory-based algorithms are more exploitation-based. The following section explains the SA algorithm in more detail.

5.2 The simulated annealing algorithm

Whether you need to solve a complex nonlinear nondifferential function optimization problem, a puzzle game like Sudoku, an academic course scheduling problem, a travelling salesman problem (TSP), a network design problem, a task allocation problem, a circuit partitioning and placement problem, a production planning and scheduling problem, or even a tennis tournament planning problem, SA can be used as a generic solver for these different continuous and discrete optimization problems.

Let's first look at the details of this solver before we use it to solve different problems. We'll start by shedding some light on the physical annealing process, which was the inspiration for SA.

5.2.1 Physical annealing

Annealing, as a heat treatment process, has been used for centuries across various industries, including metallurgy, glassmaking, and ceramics. For example, in the context of making glass bottles, annealing removes the stresses and strains in the glass resulting from shaping. This is an important step, and if it's not done, the glass may shatter due to the buildup of tension caused by uneven cooling. After the bottles have cooled to room temperature, they are inspected and finally packaged.

Annealing alters a material, causing changes in its properties, such as strength and hardness. This process heats the material to above the recrystallization temperature, maintains a suitable temperature, and then cools the material. As the temperature reduces, the mobility of molecules reduces, with the tendency that molecules will align themselves in a crystalline structure (figure 5.2).

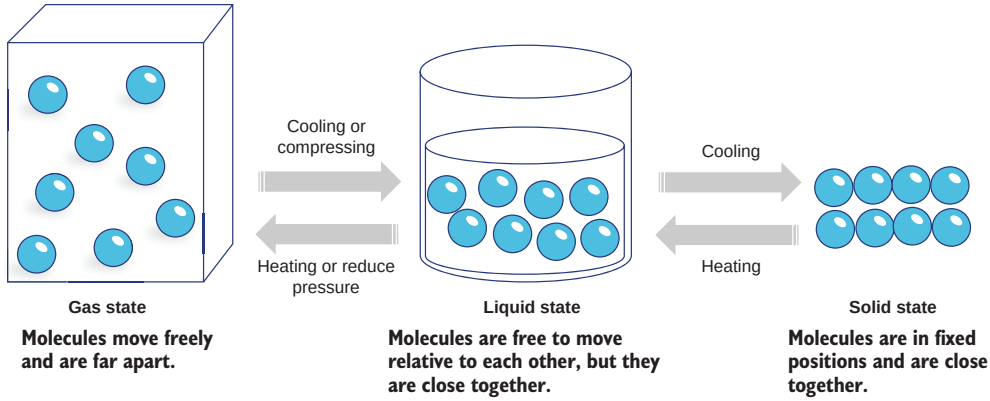


Figure 5.2 The effect of temperature on the mobility of the molecules

The aligned structure is the minimum energy state for the system. To ensure that this alignment is obtained, cooling must occur at a sufficiently slow rate. If the substance is cooled too rapidly, a noncrystalline state with irregular three-dimensional patterns may be reached, as illustrated in figure 5.3. Quartz, sodium chloride, diamond, and sugar are examples of crystalline solids that have a regular order for the arrangement of constituent particles, atoms, ions, or molecules. Glass, rubber, pitch, and many plastics are examples of noncrystalline amorphous solids. As you may know, quartz crystals are harder than glass thanks to their symmetrical molecular structure.

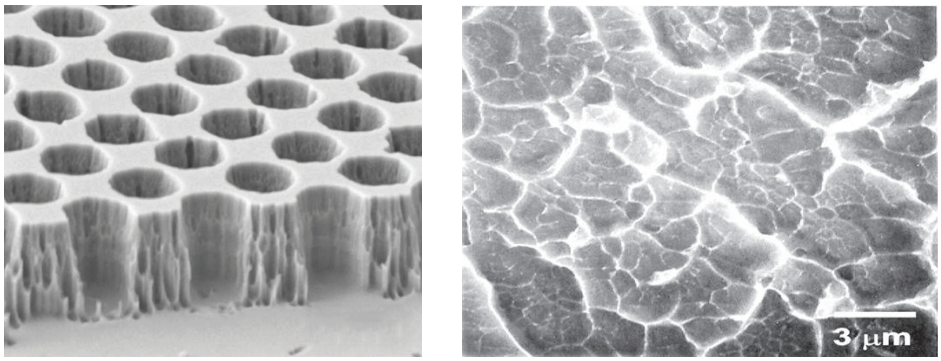


Figure 5.3 Physical annealing. Left: a metal with a crystalline structure. Right: an amorphous metal with a disordered atomic-scale structure.

NOTE The annealing process involves the careful control of the temperature and cooling rate, often called the annealing or cooling schedule. The annealing time should be long enough for the material to undergo the required transformation. If the difference in the temperature rate of change between the outside and inside of a material is too big, this may cause defects and cracks.

The fact that the aligned structure represents the minimum energy state for the system inspired scientists to think about mimicking this process to solve optimization problems. *Simulated* annealing is a computational model that mimics the physical annealing process. In the context of mathematical optimization, the minimum of an objective function represents the minimum energy of the system. SA is an algorithmic implementation of the cooling process, used to find the optimum of an objective function. Table 5.1 outlines the analogy between SA and the physical annealing process.

Table 5.1 The physical annealing and simulated annealing analogy

Physical annealing	Simulated annealing
State of a material	Solution of an optimization problem
The energy of a state	The cost of a solution
Temperature	Control parameter (temperature)
High temperature makes molecules move freely	High temperature favors search space exploration
Low temperature restricts molecules' motion	Low temperature leads to exploiting the search space
Gradual cooling helps to reduce stress and increase homogeneity and structural stability.	Gradual cooling helps to avoid getting stuck in suboptimal local minima and to find the globally optimal or near-optimal solution.

In 1953, the first computational model that replicated the physical process of annealing was introduced. This model was presented as a universal method for computing the properties of substances that can be considered collections of individual molecules interacting with each other. S. Kirkpatrick et al. were trailblazers in utilizing SA for optimization, as described in their paper "Optimization by Simulated Annealing" [1]. The following subsection explains the steps involved in the SA algorithm.

5.2.2 SA pseudocode

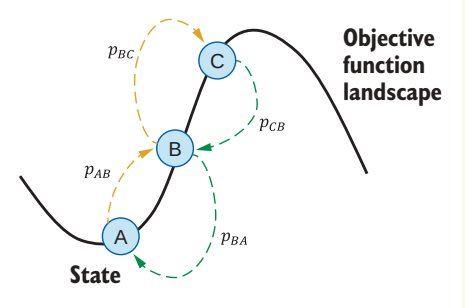
SA employs a Markov chain-based random search approach, which not only accepts new solutions that decrease the objective function (assuming a minimization problem) but can also accept probabilistic solutions that increase objective function values.

Markov chain

The Markov property, named after Russian mathematician Andrey Markov (1856–1922), is a memoryless random process. This means that the next state depends only on the current state and not on the sequence of events that preceded it. A Markov chain (MC) is a stochastic or probabilistic model that describes a sequence of possible moves in which the probability of each move depends only on the state attained in the previous move. This means that the transition from one state to another depends only on the current fully observable state and a transition probability.

(continued)

Following this memoryless random process, the transition between the current known state A, for example, to a next neighboring state B is governed by a transition probability as illustrated in the following figure. Markov chains are used in different domains such as stochastic optimization, economy, speech recognition, weather prediction, and control systems. It's also worth mentioning that Google's PageRank algorithm uses a Markov chain to model the behavior of users navigating the web. SymPy provides a Python implementation for a finite discrete time-homogeneous Markov chain through the class `sympy.stats.DiscreteMarkovChain`.



Markov chain—p_{AB}, p_{BA}, p_{BC}, and p_{CB} are transition probabilities between the states A, B, and C.

As illustrated in figure 5.4, a new neighboring solution or state x_k is always accepted if it's an improving solution (i.e., $f(x_k) < f(x_i)$). An improving solution is a solution that gives a lower value for the objective function if we're dealing with a minimization problem or gives a higher value in the case of a maximization problem. In the case of non-improving solutions, such as x_j, the solution can still be probabilistically accepted as a way to avoid the risk of getting trapped in a local minimum. This contrasts with a greedy algorithm's tendency to accept only improving solutions, making greedy algorithms more susceptible to getting stuck in local minima.

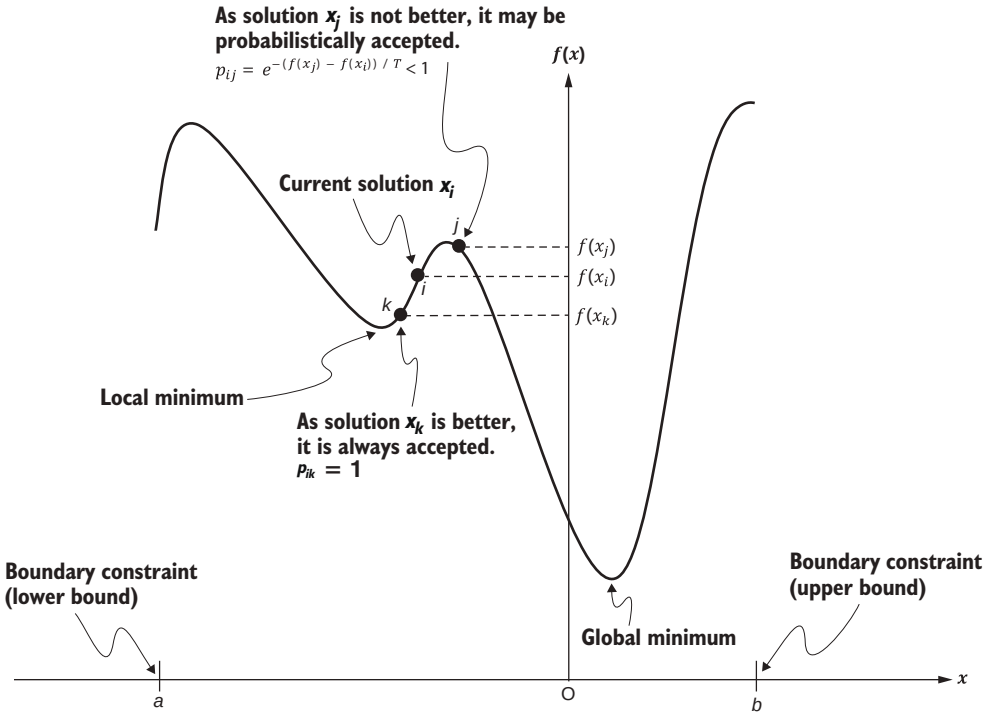


Figure 5.4 Transition probability, assuming a minimization problem. As solution x_k is an improving move, it is always accepted, and as solution x_j is non-improving, it may be probabilistically accepted based on the transition probability.

Temperature T appears in the transition probability and controls the exploration and exploitation in the search space. At high temperatures, non-improving moves will have a good chance of being accepted, but as the temperature decreases, the probability of accepting worse moves decreases. We'll discuss this in more detail in the following subsections.

The steps in SA can be summarized in the following pseudocode.

Algorithm 5.1 The SA algorithm

```
Objective function f(x), x = (x_1, . . . , x_p)^T
Initialize initial temperature T_o, initial guess x_o, iteration counter n=0
and iteration per temperature counter k=0
Set final temperature T_f, kmax maximum number of iterations per temperature
and max number of iterations N
```

```
Define cooling schedule
Begin
While T > T_f and n < N do
      While k<k_{max}
             Move randomly to a new location/state x_n + 1
             Calculate Δf = f_{n+1}(x_{n+1}) - f_n(x_n)
             If the new solution if better then
                 Accept the new solution
             Else
                 Generate a random number r
                 Accept if exp(-Δf/T)>r
             k=k+1
        End
        Update T according to the cooling schedule
        n = n + 1
End
Return the final solution
```

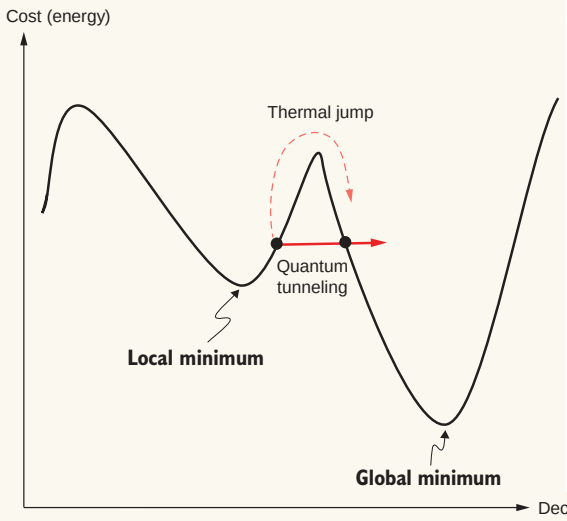
SA has the advantages of ease of use and the ability to provide optimal or near-optimal solutions for a wide range of continuous and discrete problems. The main drawbacks of this algorithm are the need to tune many parameters and the occasional slow convergence of the algorithm to the optimal or near-optimal solutions.

Aside from this original SA algorithm—classical simulated annealing (CSA)—various variants have been proposed to improve the algorithm's performance. For example, fast simulated annealing (FSA) is a semi-local search and consists of occasional long jumps. Dual annealing is a stochastic global optimization algorithm that is useful for dealing with complex nonlinear optimization problems. It is based on the combined classical simulated annealing and fast simulated annealing algorithms. The generalized simulated annealing (GSA) algorithm uses a distorted Cauchy-Lorentz visiting distribution [2].

Quantum annealing (QA)

In quantum mechanics, a quantum particle is treated as an electromagnetic wave that can penetrate with a certain probability through a potential barrier. Due to the wave nature of matter on the quantum level, there is indeed some probability that a quantum particle can traverse such a barrier if the barrier is thin enough. This phenomenon is known as quantum tunneling. The quantum tunneling effect is a phenomenon whereby wave functions or particles can penetrate through a supposedly impassable barrier even if the total energy of the particle is less than the barrier height.

As illustrated in the following figure, SA uses a thermal jump to push the search particle out of the local valley to avoid getting trapped in local minima. QA, on the other hand, searches an energy landscape to find an optimal or near-optimal solution by applying quantum effects. Instead of just walking through the landscape of the function, quantum annealing can tunnel through. This allows the algorithm to escape from local minima using quantum tunneling (tunnel effect) instead of the thermal jumps used in SA.



Feature	SA	QA
Method	Classical mechanics	Quantum mechanics
Initial state	Starts with a random solution	Starts in a superposition of all possible solutions
Evolution	Small random changes	Evolves over time using the Schrödinger equation
Acceptance process	Thermal jump	Quantum tunneling
Acceptance criteria	Transition probability	Quantum-mechanical probability
Pros/cons	Simpler and slower	Faster and difficult to implement

Simulated annealing versus quantum annealing

In QA, a number of candidate states are initialized with equal weights. Quantum-mechanical probability is used to change adiabatically and gradually the amplitudes of all states in parallel. For more information and an example of quantum annealers, see the D-Wave implementation: https://docs.dwavesys.com/docs/latest/c_gs_2.html.

The following subsections explain the different components of the SA algorithm, starting with the transition probability that allows SA to accept or reject non-improving moves.

5.2.3 Acceptance probability

Unlike hill climbing (see section 4.3.1), SA probabilistically allows downward steps, controlled by the current temperature and how bad the move is. In SA, better moves are always accepted. As shown in figure 5.4, non-improving moves can be probabilistically accepted based on the Boltzmann-Gibbs distribution.

In thermodynamics, a state at a temperature t has a probability of an increase in the energy magnitude ΔE given by the Boltzmann-Gibbs distribution as in equation 5.1:

$$p(\Delta E) = e^{-\frac{\Delta E}{k \times t}} \quad \textbf{5.1}$$

where k is the Boltzmann constant, which is the proportionality factor that relates the average relative kinetic energy of particles in a gas with the thermodynamic temperature of the gas, and which has the value of $1.380{,}649 \times 10^{-23}$ m^2 kg s^{-2} K^{-1}. However, there's no need to use this constant in a computational model that mimics the physical annealing process, so it's replaced by 1.

Moreover, the change in the energy can be replaced by the change in the objective function as a way to quantify the search progress toward the optimal or near-optimal state. So ΔE can be linked with the change of the objective function using equation 5.2:

$$\Delta E = \gamma \Delta f \tag{5.2}$$

where γ is a real constant. For simplicity, and without altering the core meaning, we can use $k = 1$ and $\gamma = 1$. Thus, the transition probability p simply becomes

$$p(\Delta f, T) = e^{-\frac{\Delta f}{T}} \tag{5.3}$$

where T is the temperature of the system. To determine whether or not we accept a change, we usually use a random number r in the interval [0,1] as a threshold. Thus, if $p > r$, or $p = e^{(-\Delta f / T)} > r$, the move is accepted. Otherwise, the move is rejected.

If P_{ij} is the probability of moving from point x_i to x_j, then P_{ij} is calculated using

$$P_{ij} = e^{-\Delta f / T} > r \tag{5.4}$$

The probability P_{ij} is called transition or acceptance probability. Accepting non-improving moves probabilistically makes the algorithm able to avoid getting trapped in some local minima. If the acceptance probability is set to 0, SA behaves similarly to hill climbing, as it will only accept solutions that are better than the current one. Conversely, if the acceptance probability is set to 1, SA becomes more exploratory, as it will always accept worse solutions, making it more akin to a random search.

The probability of accepting a worse state is a function of both the temperature of the system and the change in the cost function. As the temperature decreases, the probability of accepting worse moves decreases. Temperature can be seen as a parameter to balance the exploration and the exploitation in the search space. At high temperatures, the acceptance probability is high, which means that the algorithm accepts most of the moves to explore the parameter space. On the other hand, when the temperature is low, the acceptance probability is low, meaning that the algorithm restricts exploration. As shown in figure 5.5, if $T = 0$, no non-improving moves are accepted. In this case, SA is converted into hill climbing. As can be seen, the cooling process has an important effect on the search progress. The next section will present the different components of the cooling schedules used in SA.

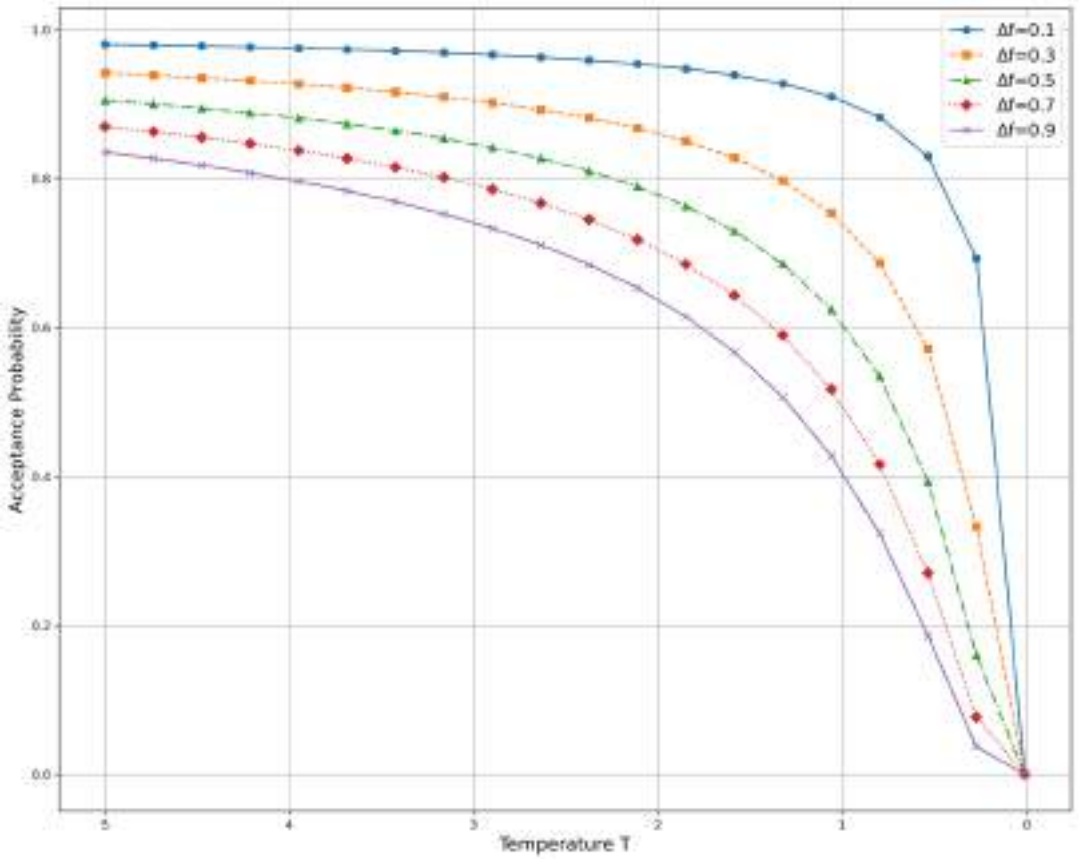


Figure 5.5 Change of acceptance probability with the temperature and the change in the objective function. The objective function change is the difference in the objective function's value between the current solution and a candidate solution. In minimization problems, a positive objective function change indicates that the candidate solution is worse than the current solution. The acceptance probability gets lower as the objective function change increases. At high temperatures, SA tends to explore more by accepting non-improving moves. As the temperature gets lower, the algorithm restricts exploration, favoring exploitation.

Given that the Boltzmann-based acceptance probability takes significant computational time (~1/3 of the SA computations), lookup tables or non-exponential probability formulas can be used instead. A lookup table can be generated by performing the exponential calculations offline only once for a range of values for changes in f and T. Other non-exponential probability formulas, such as $p(\Delta f) = 1 - \Delta f/T$, can be used as an acceptance probability. This formula should be normalized to make sure that the maximum value is 1 and the minimum value is 0.

In a computational model like SA, there is no need to strictly mimic the thermodynamic models that govern the physical annealing process. Figure 5.6 shows the difference between exponential and non-exponential acceptance probability functions. The code is available in the book's GitHub repo. The difference between exponential and non-exponential acceptance probability functions is small for small changes in the objective function—you can experiment with this using the provided code.

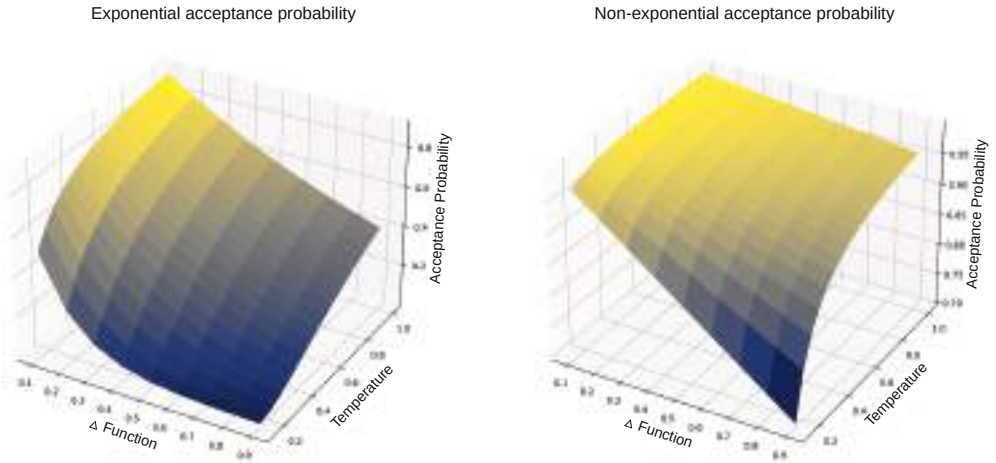


Figure 5.6 Exponential versus non-exponential acceptance probability

As temperature is part of the acceptance probability, it plays an important role in controlling the behavior of SA. The following subsection looks at how we can control the temperature to achieve a trade-off between exploration and exploitation.

5.2.4 The annealing process

The annealing process in SA involves the careful control of temperature and the cooling rate, often called the *annealing schedule*. This process involves defining the following parameters:

- Starting temperature
- Temperature decrement following a cooling schedule
- Number of iterations at each temperature
- Final temperature

This is shown in figure 5.7.

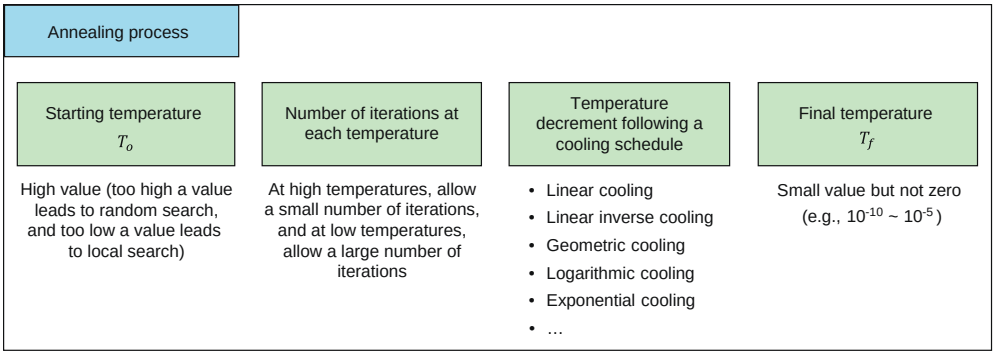


Figure 5.7 Annealing process parameters

The following subsections provide in-depth information about each of these parameters.

INITIAL TEMPERATURE

The choice of the right initial temperature is crucially important. As shown in equation 5.4, for a given change Δf

- If T is too high ($T \to \infty$), then $p \to 1$, which means almost all the changes will be accepted and the algorithm will behave like a random search algorithm.
- If T is too low ($T \to 0$), then any $\Delta f > 0$ (worse solution assuming a minimization problem) will rarely be accepted as $p \to 0$, and thus the diversity of the solution is limited, but any improvement (i.e., any $\Delta f < 0$ in the case of a minimization problem) will almost always be accepted. In this case, SA behaves like a local search and may easily become trapped in local minima.

To find a suitable starting temperature, we can use any available information about the objective function. If we know the maximum change $\max(\Delta f)$ of the objective function, we can use this to estimate an initial temperature T_o for a given acceptance probability p_o using equation 5.5:

$$T_o \approx -\frac{\max(\Delta f)}{\ln(p_o)} \qquad \textbf{5.5}$$

If the potential maximum alteration of the objective function is unknown, we can use the following heuristic approach:

1. Initiate evaluations at a very high temperature to allow for nearly all changes to be accepted.

2 Reduce the temperature quickly until roughly 50% to 60% of the inferior moves are accepted.

3 Use this temperature as the new initial temperature T_o for proper and relatively slow cooling processing.

TEMPERATURE DECREMENT

The cooling schedule is the rate at which the temperature is systematically decreased as the algorithm proceeds. This schedule is among the tunable parameters of SA. The following cooling schedules are commonly used:

- *Linear cooling schedule*—The temperature is decremented linearly using equation 5.6:

$$T = T_o - \beta_i \tag{5.6}$$

where T_o is the initial temperature, i is the pseudo time for iterations, and β is the cooling rate, which should be chosen in such a way that $T \to 0$ when $i \to i_f$ (or the maximum number N of iterations). This usually gives

$$\beta = \frac{T_o - T_f}{i_f} \tag{5.7}$$

This cooling schedule is simple and easy to implement, but may not be the best choice for all types of problems. Moreover, it requires prior knowledge or assumptions about the maximum number of iterations.

- *Linear-inverse cooling schedule*—In linear-inverse cooling, the temperature decreases quickly at high temperatures and more gradually at low temperatures, as per equation 5.8. In this equation, α is the cooling factor and should be between 0 and 1:

$$T(i) = \frac{T_o}{1 + \alpha_i} \tag{5.8}$$

- *Geometric cooling schedule*—A geometric cooling schedule essentially decreases the temperature by a cooling factor $0 < \alpha < 1$ following equation 5.9:

$$T(i) = T_o \alpha^i \tag{5.9}$$

The cooling process should be slow enough to allow the system to stabilize easily. In practice, $\alpha = 0.7 \sim 0.95$ is commonly used. The higher the value of α, the longer it will take to reach the final (low) temperature. The main advantage of the geometric method is that $T \to 0$ when $i \to \infty$, and thus there is no need to specify the maximum number of iterations. Moreover, the geometric annealing schedule provides more gradual cooling, as shown in figure 5.8.

- *Logarithmic cooling schedule*—In this cooling schedule, the temperature is decreased logarithmically according to equation 5.10:

$$T(i) = \frac{T_o}{1 + \alpha \log(1 + i)} \quad \textbf{5.10}$$

where $\alpha > 1$. Theoretically, this cooling process asymptotically converges toward the global minimum. However, it requires prohibitive computing time.

- *Exponential cooling schedule*—In this cooling schedule, the temperature is decreased exponentially according to equation 5.11:

$$T(i) = T_o e^{-\alpha i^{\frac{1}{n}}} \quad \textbf{5.11}$$

where α is the cooling factor and n is the dimensionality of the model space. In this cooling process, the temperature is decreased very quickly during the first iterations, but the speed of the exponential decay is slowed down later and can be controlled using the cooling factor.

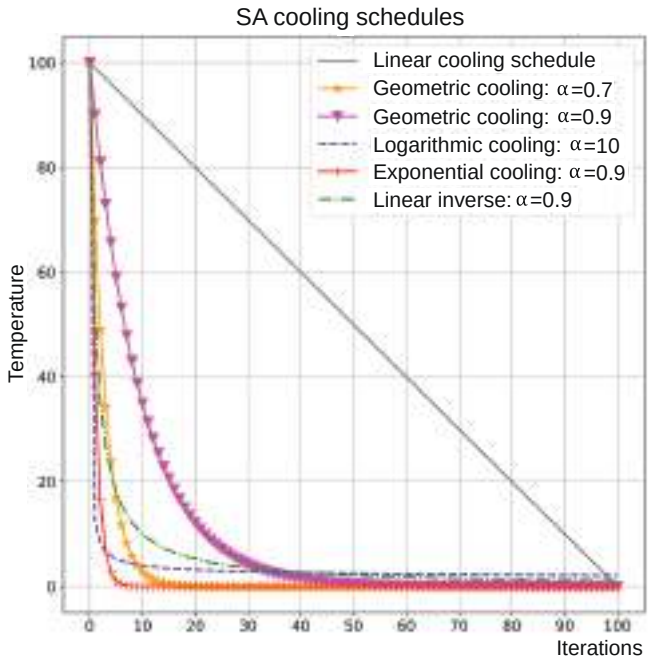


Figure 5.8 Different SA cooling schedules

As you can see, these cooling schedules are all monotonically decreasing functions that don't explicitly take into consideration how the search is progressing. In section 5.2.5, we'll look at a nonmonotonic adaptive cooling schedule.

ITERATIONS AT EACH TEMPERATURE

Before applying the cooling schedule (i.e., decreasing the temperature), it is important to allow a sufficient number of iterations at each temperature level to stabilize the system at that temperature. Typically, this is achieved by using a constant value. For example, the number of iterations at each temperature might be exponential to the problem size (e.g., the number of cities in TSP as a discrete problem or the dimensionality of a mathematical function in the case of continuous problems). However, this value can be altered dynamically.

One way to accomplish this is by limiting the number of iterations during the exploration phase of the search at the beginning, when the temperature is high. For example, when the temperature is high, we could perform a small number of iterations at each temperature and then implement the cooling process. As the search continues and the temperature decreases, we can shift toward exploitation by conducting a larger number of iterations at lower temperatures.

FINAL TEMPERATURE

It is usual to let the temperature decrease until it reaches zero. However, this can make the algorithm run a lot longer, especially when certain cooling schedules, such as geometric cooling, are used. In reality, it is not necessary to let the temperature reach zero if the chances of accepting a non-improving move at the current temperature are almost the same as if the temperature were zero. Therefore, the stopping criteria can be either of the following:

- A suitably low temperature ($T_f = 10^{-10} \sim 10^{-5}$)
- When the system reaches a "frozen" or minimum energy state (assuming a minimization problem), where neither better nor worse moves are accepted

5.2.5 Adaptation in SA

Several parameters in SA can be used to make the algorithm more adaptive to the search's progress. The initial temperature, the cooling schedule, and the number of iterations per temperature are the most critical of these parameters. Other components include the cost function, the method of generating neighborhood solutions, and the acceptance probability.

As illustrated in figure 5.9, the initial temperature can be used to control the exploration and exploitation behavior of SA. A high temperature leads to a high level of exploration, and a low temperature results in exploitative behavior (i.e., restricting the search around neighbors).

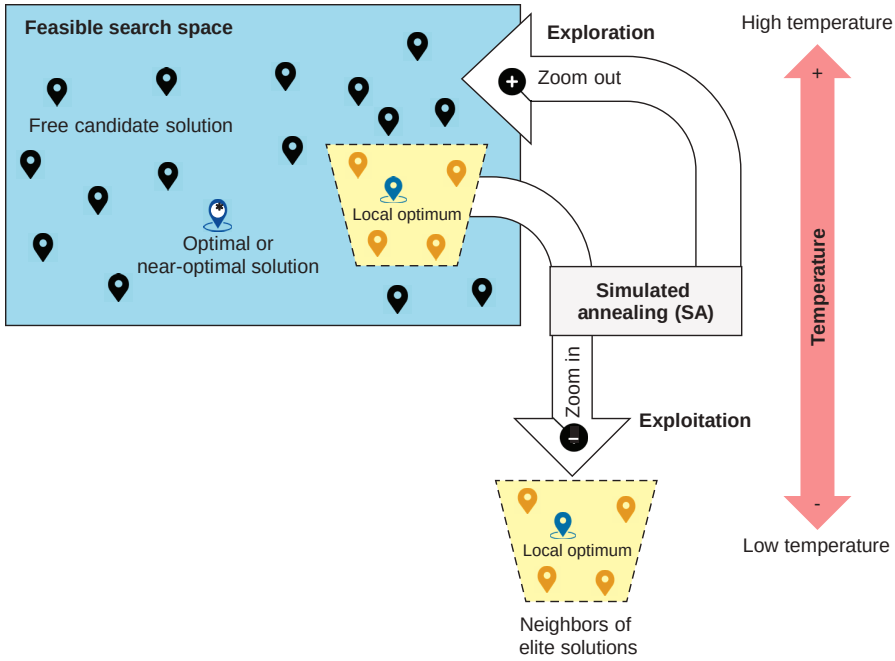


Figure 5.9 Effect of temperature in SA. High temperatures result in more exploration, whereas a low temperature restricts the exploration and leads to more exploitation in the search space.

You can think about it in terms of the movement of molecules. Assume that the molecule is the search agent. At high temperatures, the molecule moves freely in the search space, exploring different solutions. At low temperatures, the movement of the molecule becomes limited, so the exploration is restricted, and the search agent focuses on a specific part of the search space. With high temperatures at the beginning of the search, SA oscillates due to the exploration behavior that makes the algorithm accept non-improving moves with high probability. As the search progresses and the temperature gets lower, the algorithm starts to stabilize due to the exploitation behavior that makes the algorithm accept fewer non-improving moves and instead focus on the elite improving solutions.

It is always recommended that you start with a high temperature and gradually decrease it as the search progresses. However, the right initial temperature is problem-dependent. You can try different values and see which leads to better solutions. Some researchers suggest doing this adaptively, using other search methods or metaheuristics, such as a genetic algorithm.

Cooling schedules can also be used to make the algorithm more adaptive. Different cooling schedules can be used in different phases of the search, taking into consideration that most useful work is done in the middle of the schedule. Reheating can also be tried if no progress is observed. Cooling may take place every time a move (or a specific number of moves) is accepted. A nonmonotonic adaptive cooling schedule can be tried, where an adaptive factor is used, based on the difference between the current solution objective and the best objective achieved by the algorithm up to that moment, according to the following formula:

$$T = \left(\frac{1 + (f_i - f^*)}{f_i} \right) T(i) \qquad \textbf{5.12}$$

where T is the system temperature at each state transition, $T(i)$ is the current temperature, f_i is the value of the objective function at iteration i, and f^* is the best value of the objective function obtained so far.

Another adaptation parameter is the number of iterations per temperature. This number can be adaptively changed by allowing a small number of iterations at high temperatures and allowing a large number of iterations at low temperatures to fully explore the local optimum.

The adaptation ability of the SA algorithm can be also influenced by the objective function and the representation of problem constraints utilized. As a general rule, it is advisable to steer clear of cost functions that yield the same result for multiple states (e.g., the number of edges incorporated in a TSP route). This type of function does not guide the search because it may not change in the objective function from one state to another.

For example, imagine two feasible routes with the same number of edges (i.e., $\Delta f = 0$). Counting on the number of edges as a cost function wouldn't be a good idea. However, many problems have constraints that can be represented in the cost function using reward or penalty terms. One way to make the algorithm more adaptive is to dynamically change the weighting of the reward and penalty terms. In the initial phase, the constraints can be relaxed more than in the advanced phases of the search.

There have been numerous efforts to make the selection and control of SA parameters totally adaptive. One example of such an effort was proposed by Ingber in "Adaptive simulated annealing (ASA): Lessons learned" [3]. ASA automatically adjusts the algorithm parameters that control the temperature schedule, requiring the user to only specify the cooling rate. The method uses a linear random combination of previously accepted steps and parameters to estimate new steps and parameters.

An ASA algorithm with greedy search (ASA-GS) is proposed by Geng et al. to solve the TSP [4]. ASA-GS is based on the classical SA algorithm and utilizes a greedy search technique to speed up the convergence rate. ASA utilizes dynamic adjustments in parameters like the temperature cooling coefficient, greedy search iterations, compulsive accept instances, and probability of accepting a new solution. These adaptive parameter controls aim to enhance the trade-off between quality and time efficiency.

SA finds extensive application across various domains. Its utility extends to solving diverse optimization problems encompassing nonlinear function optimization, TSP, academic course scheduling, network design, task allocation, circuit partitioning and placement, robot motion planning, and vehicle routing, as well as resource allocation and scheduling. The following sections show how you can use SA to solve continuous and discrete optimization problems in different domains.

5.3 Function optimization

As an example of continuous optimization problems, let's consider the following simple function optimization problem: find x which minimizes $f(x) = (x - 6)^2$, subject to the constraint $0 \le x \le 31$.

We can start with an initial random solution from the range of x. Different neighboring solutions can be generated by adding a random floating value chosen from a Gaussian or normal distribution with a given mean and standard deviation. The `random.gauss()` function in Python can be used as follows:

```
import random
mu, sigma = 0, 1 # mean and standard deviation
print(random.gauss(mu, sigma))
```

Assume that the initial temperature $T_0 = 5$, the number of iterations per temperature is 2, and the geometric cooling factor $\alpha = 0.85$. Let's carry out a few hand iterations to show how SA can solve this problem:

- *Initialization*—An initial solution is randomly generated, and its cost is evaluated as follows: $x = 2$ and $f(2) = 16$.
- *Iteration 1*—A new solution $x = 2.25$ is generated by adding a random value from a Gaussian distribution using the following code:

  ```
  import numpy as np
  x=x+np.random.normal(mu, sigma, 1)
  ```

  $f(2.25) = 14.06$ is an improving solution and so is accepted.
- *Iteration 2*—A new solution is generated by adding a random value to the last accepted solution from the previous iteration. The new solution $x = 2.25 - 1.07 = 1.18$, $f(1.18) = 23.23$ is a non-improving solution, so the acceptance probability must be calculated: $p = e^{-\Delta f / T} = e^{-(23.23 - 14.06)/5} = 0.1597$. We generate a random number r between (0,1), and let's say it was $r = 0.37$. As $p \not> r$, we reject this solution.
- *Iteration 3*—We update the temperature because we have used the initial temperature $T_0 = 5$ for two iterations so far. Following geometric cooling, the new temperature is $T_1 = T_o \alpha^i = 5*0.85^1 = 4.25$. We'll use this value for two iterations starting with this iteration. We'll now generate a new solution based on the last accepted solution by adding a random value from a Gaussian distribution. The new solution is $x = 2.25 + 1.57 = 3.82$ with $f(3.82) = 4.75$. This is an improving solution, so it's accepted, and the search continues.

SciPy provides Python implementations for SA algorithms and other algorithms to handle mathematical optimization problems. `scipy.optimize.anneal` is deprecated in SciPy, and a `dual_annealing()` function is available instead. The following listing shows the Bohachevsky function (which has the formula $f(x_1,x_2) = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1) - 0.4\cos(3\pi x_2) + 0.7$) solution using the SciPy dual annealing algorithm. The complete listing is available in the book's GitHub repo.

Listing 5.1 Function optimization using `scipy.optimize.dual_annealing`

```
#!pip install scipy
import numpy as np
from scipy.optimize import dual_annealing

def objective_function(solution):
    return solution[0]**2 +2*(solution[1]**2) - 0.3*np.cos(3*np.
pi*solution[0]) -
➥ 0.4*np.cos(4*np.pi*solution[1]) + 0.7

bounds = np.asarray([[-100, 100], [-100, 100]])

res_dual = dual_annealing(objective_function, bounds=bounds, maxiter = 100)

print('Dual Annealing Solution: f(%s) = %.5f' % (res_dual['x'], res_
dual['fun']))
```

Define the objective function or functions (e.g., Bohachevsky function).

Define the boundary constraints of the decision variables.

Perform the dual annealing search.

Print the dual annealing solution.

MEALPY is another Python library that provides implementations of different nature-inspired metaheuristic algorithms (see appendix A for more details). As a continuation, the following code shows the Bohachevsky function solution using MEALPY SA (the complete version of listing 5.1 is available in the book's GitHub repo):

```
#!pip install mealpy
from numpy import exp, arange
import matplotlib.pyplot as plt
from pylab import meshgrid,cm,imshow,contour,clabel,colorbar,axis,title,show
from mealpy.physics_based.SA import OriginalSA

problem = {"fit_func": objective_function,"lb": [bounds[0][0], bounds[1][0]],
"ub":
➥ [bounds[0][1], bounds[1][1]], "minmax": "min", "obj_weights": [1, 1]}

epoch = 100
pop_size = 10
max_sub_iter = 2
t0 = 1000
t1 = 1
move_count = 5
mutation_rate = 0.1
mutation_step_size = 0.1
mutation_step_size_damp = 0.99
```

Define the problem.

Define the MEALPY algorithm parameters to perform an SA search using MEALPY.

```
model = OriginalSA(epoch, pop_size, max_sub_iter, t0, t1, move_count,
mutation_rate,
➥ mutation_step_size, mutation_step_size_damp)

mealpy_solution, mealpy_value = model.solve(problem)

print('MEALPY SA Solution: f(%s) = %.5f' % (mealpy_solution, mealpy_value))
```

Define a MEALPY SA solver.

Solve the problem using a defined solver.

Print the MEALPY SA solution.

Figure 5.10 shows the Bohachevsky function's solution. The performance of the algorithm depends mainly on its parameter tuning and stopping criteria. MEALPY runs a parallel version of SA and exposes many parameters to be tuned.

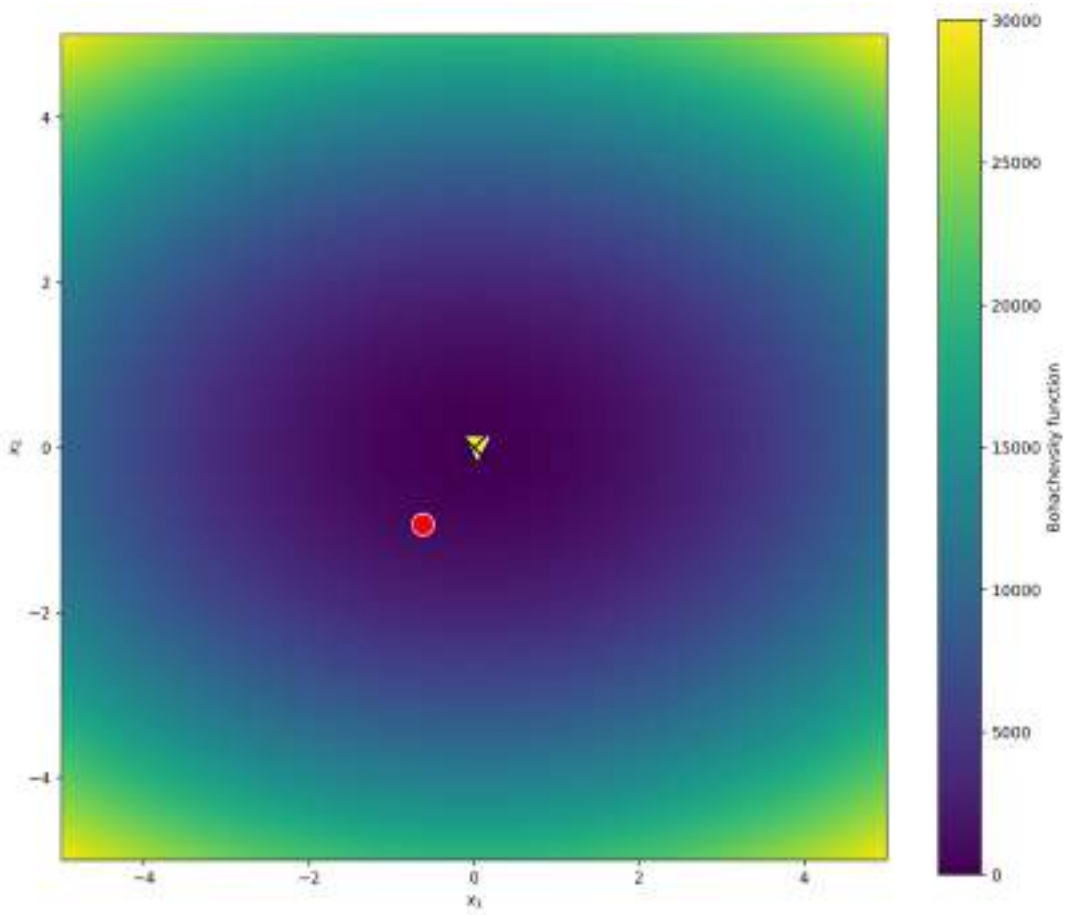


Figure 5.10 Solution of a continuous function optimization problem using SA. The cross in the center is the optimal solution. The triangle is the solution obtained by MEALPY SA. The dot is the SciPy dual annealing solution.

NOTE Appendix A shows how to use SA in other Python packages to solve mathematical optimization problems.

Let's implement an SA algorithm from scratch so we can gain more control and better handle different types of continuous and discrete optimization problems. In our implementation in the optalgotools package, we decouple the problem definition from the solver so we can use the solvers to handle different problems.

Let's apply our implementation to find the global minimum of the aforementioned simple function optimization problem and of more complex function optimization problems as well. There are several complex mathematical functions in multidimensional space, such as Rosenbrock's function, the Ackley function, the Rastrigin function, the Schaffer function, the Schwefel function, the Langermann function, the Levy function, the Bukin function, the Eggholder function, the cross-in-tray function, the drop wave function, and the Griewank function. Examples of function optimization test problems and datasets can be found in appendix B.

Listing 5.2 shows how SA can be used to solve the following mathematical functions:

- *A simple quadratic equation*—This is what we used in the hand iterations.
- *The Bohachevsky function (global minimum 0)*—This is a 2D unimodal function with a bowl shape. This function is known to be continuous, convex, separable, differentiable, nonmultimodal, nonrandom, and nonparametric, so derivate-based solvers can efficiently handle it. Note that a function whose variables can be separated is known as a *separable function*. Nonrandom functions contain no random variables. Nonparametric functions assume that the data distribution cannot be defined in terms of a finite set of parameters.
- *The Bukin function*—This function has many local minima, all of which lie on a ridge, and one global minimum $f(x_0) = 0$ at $x_0 = f(-10,1)$. This function is continuous, convex, nonseparable, nondifferentiable, multimodal, nonrandom, and nonparametric. This requires a derivative-free solver (also known as a black-box solver) such as SA.
- *The Gramacy & Lee function*—This is a 1D function with multiple local minima and local and global trends. This function is continuous, nonconvex, separable, differentiable, nonmultimodal, nonrandom, and nonparametric.
- *The Griewank 1D, 2D, and 3D functions*—These have many widespread local minima. These functions are continuous, nonconvex, separable, differentiable, multimodal, nonrandom, and nonparametric.

In our implementation, these are the SA parameters:

- A maximum number of iterations: `max_iter=1000`
- Maximum iterations per temperature: `max_iter_per_temp=100`
- An initial temperature: `initial_temp=1000`
- A final temperature: `final_temp=0.0001`

- A cooling schedule: `cooling_schedule='geometric'` (available options: `'linear'`, `'geometric'`, `'logarithmic'`, `'exponential'`, `'linear_inverse'`)
- A cooling factor: `cooling_alpha=0.9`
- A debug option: `debug=1` (`debug=1` prints the initial and final solution; `debug=2` provides hand-iteration tracing)

Feel free to change these settings and observe their effect on the performance of the algorithm.

Listing 5.2 Continuous function optimization using SA

```
import random
import math
import numpy as np
from optalgotools.algorithms import SimulatedAnnealing
from optalgotools.problems import ProblemBase, ContinuousFunctionBase
```

Quadratic function SA-based solution

```
def simple_example(x):
    return (x-6)**2

simple_example_bounds = np.asarray([[0, 31]])
simple_example_obj = ContinuousFunctionBase(simple_example, simple_example_
➥ simple_example_bounds)
sa = SimulatedAnnealing(max_iter=1000, max_iter_per_temp=100,
➥ initial_ temp=1000, final_temp=0.0001, cooling_schedule='geometric',
➥ cooling_alpha=0.9, debug=1)
sa.run(simple_example_obj)
```

Bohachevsky SA-based solution

```
def Bohachevsky(x_1, x_2):
    return x_1**2 +2*(x_2**2)-0.3*np.cos(3*np.pi*x_1)-0.4*np.cos(4*np.
pi*x_2)+0.7

Bohachevsky_bounds = np.asarray([[-100, 100], [-100, 100]])
Bohachevsky_obj = ContinuousFunctionBase(Bohachevsky, Bohachevsky_bounds, 5)
sa.run(Bohachevsky_obj)
```

Bukin SA-based solution

```
def bukin(x_1, x_2):
    return 100*math.sqrt(abs(x_2-0.01*x_1**2)) + 0.01 * abs(x_1 + 10)

bukin_bounds = np.asarray([[-15, -5], [-3, 3]])
bukin_obj = ContinuousFunctionBase(bukin, bukin_bounds, 5)
sa.run(bukin_obj)
```

Gramacy & Lee SA-based solution

```
def gramacy_and_lee(x):
    return math.sin(10*pi*x)/(2*x) + (x-1)**4

gramacy_and_lee_bounds = np.asarray([[0.5, 2.5]])
gramacy_and_lee_obj = ContinuousFunctionBase(gramacy_and_lee, gramacy_and_
➥ lee_bounds, .1)
sa.run(gramacy_and_lee_obj)
```

Griewank 1D SA-based solution

```
def griewank(*x):
    x = np.asarray(x)
    return np.sum(x**2/4000) - np.prod(np.cos(x/np.sqrt(np.asarray(range(1,
    ➥ len(x)+1))))) + 1

griewank_bounds = np.asarray([[-600, 600]])
griewank_1d=ContinuousFunctionBase(griewank, griewank_bounds, 10)
sa.run(griewank_1d)
```

Griewank 2D SA-based solution

```
griewank_bounds_2d = np.asarray([[-600, 600]]*2)
griewank_2d=ContinuousFunctionBase(griewank, griewank_bounds_2d,
➥ (griewank_bounds_2d[:, 1] - griewank_bounds_2d[:, 0])/10)
sa.run(griewank_2d)
```

Griewank 3D SA-based solution

```
griewank_bounds_3d = np.asarray([[-600, 600]]*3)
griewank_3d=ContinuousFunctionBase(griewank, griewank_bounds_3d,
➥ (griewank_bounds_3d[:, 1] - griewank_bounds_3d[:, 0])/10)
sa.run(griewank_3d)
```

This is an example of the output for the Bukin function:

```
Simulated annealing is initialized:
current value = 60.73784664253138, current temp=1000
Simulated Annealing is done:
curr iter: 154, curr best value: 0.6093437608551259, curr
temp:9.97938882337113e-05, curr best: sol: [-14.63282848   2.14122839]
global minimum: x = -14.6328, 2.1412, f(x) = 0.6093
```

As can be seen, SA is able to handle different multidimensional, nonlinear function optimization problems. This stochastic global optimization algorithm is able to adapt to the landscape of the objective function and avoid getting trapped in local minima. However, in the case of multidimensional functions such as Griewank 2D and 3D, SA takes time to converge. The following sections show how SA can handle discrete problems such as Sudoku and TSP.

5.4 Solving Sudoku

Sudoku, also known as Su Doku, is one of the most popular number puzzles of all time. This game is adapted from a mathematical concept called a *Latin square*. The first version of the Sudoku puzzle was created by a retired architect named Howard Garns and appeared in the late 1970s as a puzzle in *Dell Pencil Puzzles and Word Games*. The game subsequently showed up in Japan in 1984 under the name "Sudoku," which is abbreviated from the Japanese "Sūji wa dokushin ni kagiru," which means the numbers (or digits) must remain single. Nowadays, Sudoku games are popular around the globe and are published in game websites, puzzle booklets, and newspapers.

The Sudoku game can be seen as a constraint-satisfaction problem (CSP) that is solved by correctly filling a 9×9 grid with digits so that each column, each row, and each

of the nine 3 × 3 subgrids (aka "boxes," "blocks," or "regions") contain all of the digits from 1 to 9. Any row or column or 3 × 3 subgrid shouldn't contain more than one of the same number from 1 to 9.

Aside from entertainment, Sudoku is used in real-life applications such as developmental psychology and steganography. For example, several studies have showed that solving Sudoku or crosswords or other brain games may help in keeping your brain 10 years younger and can slow down the progression of conditions such as Alzheimer's. Sudoku can also be used as a tool to improve problem-solving skills, critical thinking, and attention. Finally, steganography is the technique of hiding images, messages, files, or other secret data within something that isn't a secret. In secret data delivery applications, digital images can be used to conceal secret data. The Sudoku puzzle is then used to modify selected pixel pairs in the cover image, based on a specially designed reference matrix, to insert secret digits.

Latin square

Latin squares were devised in the 10th century by Arabic numerologists who dealt with the mystical power of numbers. Islamic amulets, known as wafq majazi, from the 13th century have been found, and they were sketched in the margins of a 16th century Arabic medical text. The name "Latin" was inspired by the famous Swiss mathematician Leonhard Euler (1707–1783) who used Latin letters as symbols in the squares.

A *Latin square* is an $n \times n$ array filled with n different numbers, symbols, or colors arranged in such a way that no orthogonal (row or column) contains the same number, symbol, or color twice. An example of a 4 × 4 Latin square is shown here:

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

Latin squares are different from *magic squares*. A magic square is a square array of positive integers 1, 2, ..., n^2 arranged such that the sum of the n numbers in any horizontal, vertical, or main diagonal line is always the same number. Sudoku is based on Latin squares. In fact, any solution to a Sudoku puzzle is a Latin square. KenKen and KenDoku are other number puzzles based on an enhanced version of Latin squares and require some degree of arithmetic skills.

Generally speaking, the search space of Sudoku is huge. There are 6.671×10^{21} possible solvable Sudoku grids that yield a unique result [5]. According to Encyclopedia Britannica, if each human on earth solves one Sudoku puzzle every second, they wouldn't get through all of them until about the year 30,992. However, taking out symmetries, such as rotations, reflections, permuting columns and rows, and swapping digits, the number of essentially different Sudoku grids is reduced to $5{,}472{,}730{,}538 \approx 5.473 \times 10^9$ [6]. The generalized $n \times n$ Sudoku problem is an NP-complete problem. However, some instances, such as standard 9×9 Sudoku, are not NP-complete. Constant-time algorithms exist to solve some instances of 9×9 Sudoku, in $O(1)$ time, as each and every 9×9 Sudoku can be listed, enumerated, and indexed in a finite dictionary or a lookup table used to find a solution. However, these algorithms cannot handle arbitrary generalized $n \times n$ Sudoku problems.

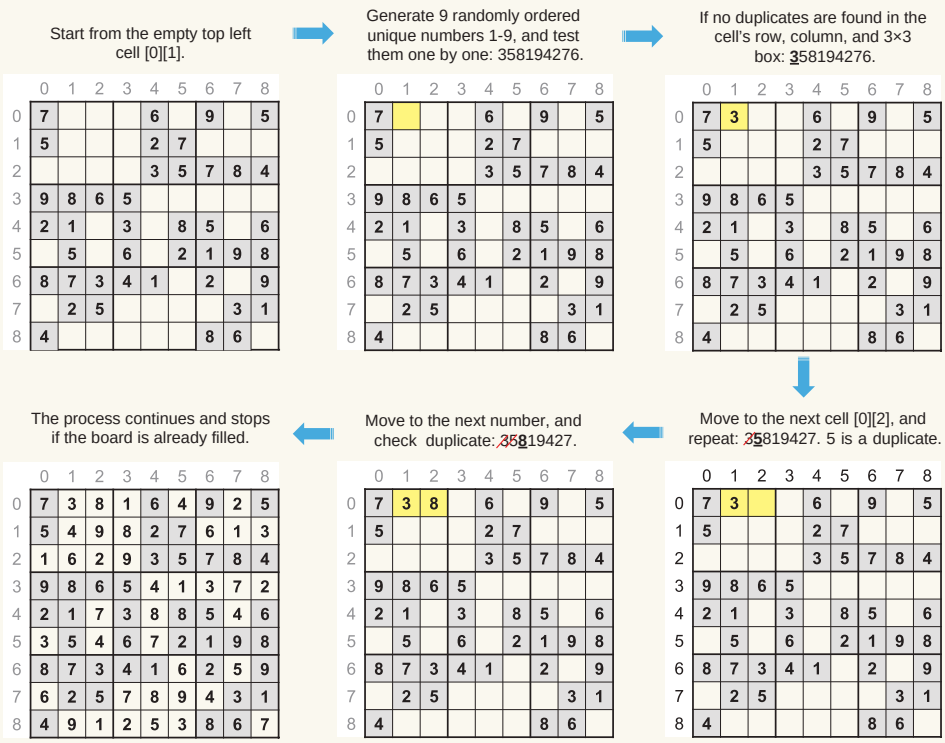
Backtracking, dancing links, and Crook's pencil-and-paper are common algorithms for solving Sudoku, especially if the size of the problem is small. *Backtracking* is mainly a classical depth-first search that tests a whole branch until that branch violates the rules or returns a solution.

Dancing links (DLX), invented by Donald Knuth in 2000, uses algorithm X to solve Sudoku puzzles, handled as exact cover problems. In the exact cover problem, given a binary matrix (i.e., a matrix composed only of 0 and 1), it is necessary to find a set of rows containing exactly one 1 in each column. Algorithm X, a recursive search algorithm, is applied to solve the exact cover problem using the backtracking method.

In *Crook's pencil-and-paper algorithm*, all possible numbers in each cell are listed. This list of numbers is called marking-up of the cell. We then try to find out if there is a row, column, or block with only one possible value throughout the row, column, or block. Once found, we fill in this cell with this number and update the markups in any affected row, column, or box. The next step is to find preemptive sets. As described in Crook's paper, a preemptive set is composed of numbers from the set [1,2,…,9] and is a set of size m, $2 \leq m \leq 9$, whose numbers are potential occupants of m cells exclusively, where exclusively means that no other numbers in the set [1,2,…,9], other than the members of the preemptive set, are potential occupants of those m cells. The last step is to eliminate possible numbers outside preemptive sets.

Backtracking

Backtracking algorithms are commonly used to solve search and optimization problems by recursion. The backtracking algorithm builds a feasible solution or a set of feasible solutions incrementally. Given a 9 × 9 Sudoku board, the algorithm visits all the empty cells following depth-first traversal order, filling the digits incrementally, and it backtracks when a number is not found to be valid. The following figure illustrates the backtracking algorithm steps for a 9 × 9 Sudoku puzzle.



Backtracking steps for a 9 × 9 Sudoku puzzle

The next listing shows how a 9 × 9 Sudoku is solved using the SA algorithm.

Listing 5.3 Solving Sudoku using SA

```
from optalgotools.algorithms import SimulatedAnnealing
from optalgotools.problems import Sudoku

sa = SimulatedAnnealing(max_iter=100000, max_iter_per_temp=1000,
➥ initial_temp=500, final_temp=0.001, cooling_schedule='geometric',
➥ cooling_alpha=0.9, debug=1)

Sudoku_hard = [
    [9, 0, 0, 1, 0, 0, 0, 5, 4],
    [0, 0, 0, 0, 8, 0, 0, 0, 0],
    [0, 0, 5, 0, 0, 9, 0, 0, 3],
```

Import the SA solver.

Import a Sudoku problem.

Create a SA solver with the selected parameters.

```
    [0, 9, 0, 0, 3, 5, 0, 4, 1],
    [0, 0, 0, 0, 1, 0, 0, 0, 0],
    [4, 1, 0, 2, 6, 0, 0, 8, 0],
    [7, 0, 0, 3, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 4, 0, 0, 0, 0],
    [3, 5, 0, 0, 0, 1, 0, 0, 6],
]
sudoku_prob = Sudoku(Sudoku_hard)
sudoku_prob.print()

sudoku_prob.solve_backtrack()
sa.run(sudoku_prob, 0)
```

Create a hard 9 × 9 Sudoku (available variants include trivial, easy, medium, hard, and evil).

Solve the Sudoku using the backtracking algorithm.

Solve the Sudoku using SA.

You can try different variants of Sudoku by changing the puzzle configuration. In easy Sudoku problems, cells contain more prefilled numbers than medium or hard ones. Evil Sudoku is the highest level of puzzle difficulty. Table 5.2 compares the time it takes for SA, backtracking, and the Python Linear Programming (PuLP) library to solve different instances of 9 × 9 Sudoku puzzles. PuLP provides linear and mixed programming solvers. The default solver used in PuLP is Cbc (COIN-OR branch and cut), which is an open source solver for mixed integer linear programming problems. More information about PuLP is available in appendix A.

Table 5.2 SA versus backtracking versus PuLP in solving a 9 × 9 Sudoku puzzle

Time to find the solution(s)	Trivial	Easy	Medium	Hard	Evil
Backtracking	0.01	0.01	0.11	0.69	1.58
PuLP	0.69	0.12	0.11	0.13	0.12
Classical SA	0.10	0.07	0.01	3:17	Suboptimal in 3:16

As you can see, classical SA does not outperform the backtracking approach and is much slower in the case of hard and evil instances of Sudoku problems. PuLP is efficiently able to handle different variants of Sudoku in a consistent time, compared to backtracking and SA.

In the case of evil Sudoku, SA converges to a suboptimal solution despite trying different parameter settings. Given that this is a constraint-satisfaction problem, the notion of suboptimality is not valid, as a suboptimal solution is an invalid solution. This means that SA doesn't manage to solve the evil instance of Sudoku. Being a well-structured problem, 9 × 9 Sudoku with different levels of difficulty can be easily solved using a backtracking algorithm. Generally speaking, if the problem is well-structured with a well-known algorithm solution, metaheuristics approaches don't usually outperform these classical and more deterministic approaches.

5.5 Solving TSP

As described in section 2.1.1, the traveling salesman problem (TSP) is used as a platform for the study of general methods that can be applied to a wide range of discrete optimization problems. Consider solving the instance of TSP shown in figure 5.11 using SA. In this TSP, a traveling salesman must visit five cities and return home, making a loop (a round trip).

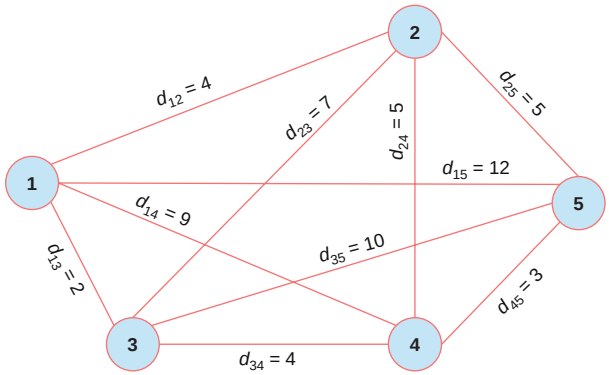


Figure 5.11 TSP for five cities—there are 5!/2 = 60 possible tours, assuming symmetric TSP. The weights on the edges of the graph represent the travel distances between the cities.

Assume the following values: initial temperature = 500, final temperature = 50, a linear decrement rate of 50, and one iteration at each temperature. A TSP solution takes the form of a permutation as follows: Solution = [1, 3, 4, 2, 5]. The objective function is the total distance of the route. Swapping is a suitable operator that can be used to generate neighboring solutions:

- *Iteration 0*—The initial solution is Solution = [1, 3, 4, 2, 5], cost = 2 + 4 + 5 + 5 + 12 = 28, as shown in figure 5.12.

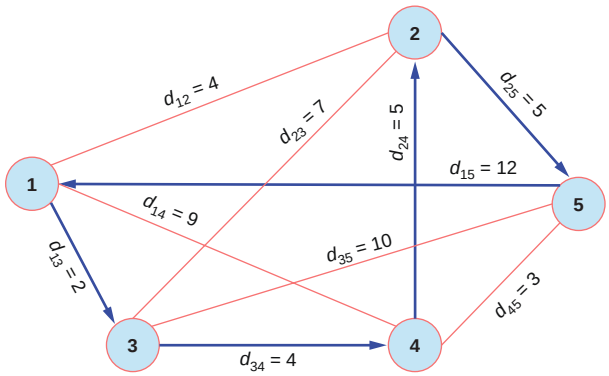


Figure 5.12 SA iteration 0 of a 5-city TSP

- *Iteration 1*—To generate a candidate solution, select two random cities (e.g., 2 and 3), and swap them. This results in a new solution [1, 2, 4, 3, 5] with a cost of 35 (figure 5.13).

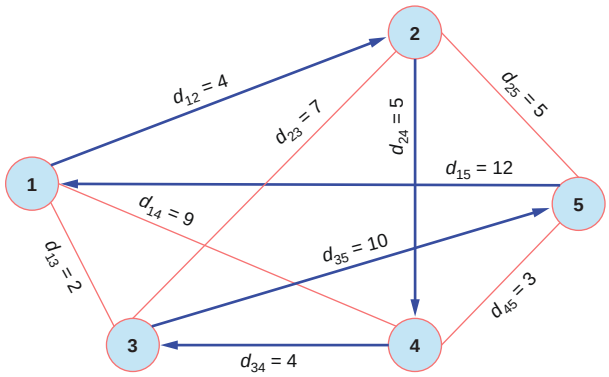


Figure 5.13 SA iteration 1 of a 5-city TSP

Since the new solution has a longer tour length, it will be conditionally accepted, according to a probability of $p = e^{-\Delta f/T} = e^{-(35-28)/T} = e^{-7/T}$ (at higher temperatures, there's a higher probability of acceptance). We pick a random value r within 0 and 1. If $P > r$, we accept this solution. Otherwise, we reject this solution. Assuming the new solution was not accepted, we generate a different one, starting from the initial solution:

- *Iteration 2*—A solution is generated by swapping cities 2 and 5 in the initial solution. The tour length of the candidate solution is 18 (figure 5.14).

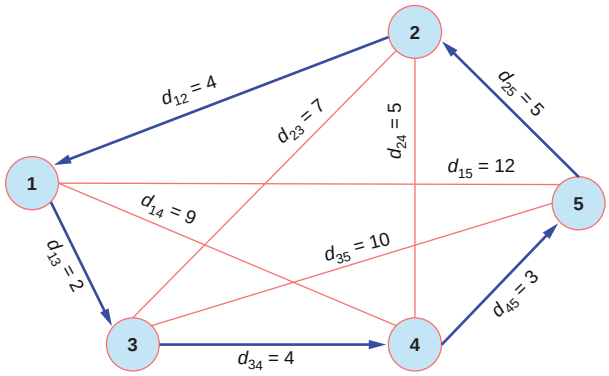


Figure 5.14 SA iteration 2 of a 5-city TSP

Since this solution has a shorter tour length, it will be accepted, and the search continues until the termination criteria are met. The following listing shows an SA solution for this simple TSP problem.

Listing 5.4 Solving TSP using SA

```
from optalgotools.algorithms import SimulatedAnnealing
from optalgotools.problems import TSP

dists = [ [0] * 5 for _ in range(5)]
dists[0][1] = dists[1][0] = 4
dists[0][2] = dists[2][0] = 2
dists[0][3] = dists[3][0] = 9
dists[0][4] = dists[4][0] = 12
dists[1][2] = dists[2][1] = 7
dists[1][3] = dists[3][1] = 5
dists[1][4] = dists[4][1] = 5
dists[2][3] = dists[3][2] = 4
dists[2][4] = dists[4][2] = 10
dists[3][4] = dists[4][3] = 3

tsp_sample = TSP(dists, 'random_swap')

sa = SimulatedAnnealing(max_iter=10000, max_iter_per_temp=1, initial_
temp=500,
➥ final_temp=50, cooling_schedule='linear_inverse', cooling_alpha=0.9,
debug=2)
sa.run(tsp_sample)
```

Create an instance of a TSP problem.

Run the SA solver, and show the results in each iteration.

Create an instance of the SA solver.

Let's now consider some benchmark instances of TSP, such as Berlin52 from TSPLIB (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/). This dataset contains 52 locations in the city of Berlin. The shortest route obtained for the Berlin52 dataset is 7,542. The next listing shows how we can solve this TSP instance using our SA implementation.

Listing 5.5 Solving the Berlin52 TSP using SA

```
from optalgotools.problems import TSP
from optalgotools.algorithms import SimulatedAnnealing
import matplotlib.pyplot as plt

berlin52_tsp_url = 'https://raw.githubusercontent.com/coin-or/jorlib/
b3a41ce773e9b3b5b73c149d4c06097ea1511680/jorlib-core/src/test/resources/
tspLib/tsp/berlin52.tsp'

berlin52_tsp = TSP(load_tsp_url=berlin52_tsp_url, gen_method='mutate',
➥ rand_len=True, init_method='random')

sa = SimulatedAnnealing(max_iter=1200, max_iter_per_temp=500, initial_
temp=150,
➥ final_temp=0.01, cooling_schedule='linear', debug=1)

sa.run(berlin52_tsp, repetition=1)
print(sa.val_allbest)

berlin52_tsp.plot(sa.s_best)
```

Permanent URL for the Berlin52 dataset

Create a TSP object for Berlin52.

Create an SA model.

Run SA, and evaluate the best solution distance.

Plot the route.

Here is an example of the output:

```
sol: [0, 48, 34, 35, 33, 43, 45, 36, 37, 47, 23, 4, 14, 5, 3, 24, 11, 50, 10,
51, 13, 12, 26, 27, 25, 46, 28, 15, 49, 19, 22, 29, 1, 6, 41, 20, 30, 17, 16,
2, 44, 18, 40, 7, 8, 9, 32, 42, 39, 38, 31, 21, 0]
8106.88
```

Figure 5.15 shows the route generated by SA for Belin52 TSP.

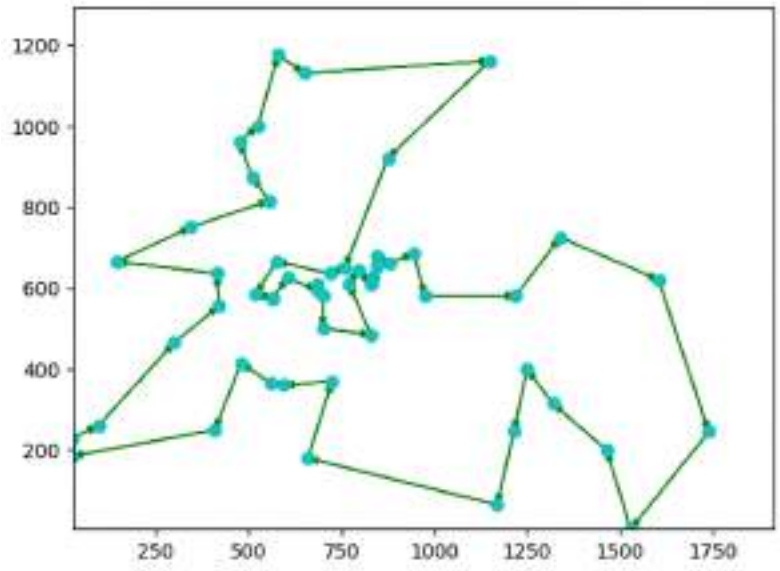


Figure 5.15 Solution of Belin52 using SA

As you can see, the near-optimal solution found by SA is 8,106.88. This value is a bit higher than the best-known solution for the Berlin52 TSP, which is 7,542. Parameter tuning and algorithm adaptation can help improve the results. For example, Geng et al.'s "Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search" paper discusses using three kinds of mutations (vertex insert mutation, block insert mutation, and block reverse mutation) with different probabilities during the search to improve the accuracy of SA in solving TSP problems. Moreover, parameters such as the cooling coefficient of the temperature, the times of greedy search used to speed up the convergence rate, the times of compulsive accept, and the probability of accepting a new solution, can be adapted according to the size of the TSP instances. An implementation of this adaptive algorithm is available from this GitHub repo: https://github.com/ildoonet/simulated-annealing-for-tsp.

The effect of the algorithm's parameters can be studied, such as the initial temperature, the cooling schedule, the number of iterations per temperature, and the final temperature. For example, SA was applied for the Berlin52 TSP instance with the following settings:

- Maximum number of iterations = 1200
- Number of iterations per each temperature T = 500
- $T_{initial} = 150$
- $T_{final} = 0.01$
- Linear cooling

Our implementation supports the following methods for mutating a new solution from an old one:

- `random_swap`—Swap two cities in the path. This can be done multiple times for the same solution by using `num_swaps`. Also, the swap can be done in a smaller window of the whole path using `swap_wind = [1 - n]`. For example, suppose the route is [A, B, C, D, F]. Swapping two random cities, like B and F, will result in a new route [A, F, C, D, B].
- `reverse`—Reverse the order of a subset of the cities with either a random length, using `rand_len`, or using `rev_len`, which has a default of 2. For example, starting with the solution [A, B, C, D, F], if we apply `reverse` with length 3, we can get a new solution [A, D, C, B, F].
- `insert`—Pick a random city, remove it from the path, and reinsert it before a different random city. For example, starting from the solution [A, B, C, D, F], we could pick city B and insert it before city F so we get a new solution [A, C, D, B, F].
- `mutate`—Randomly pick a number of consecutive cities from the current solution, and shuffle them. For example, starting from the solution [A, B, C, D, F], we may pick C, D, F and shuffle them so we get a new solution [A, B, F, C, D].

The implementation also supports two methods of initializing the path:

- `random`—This means the path is generated completely randomly.
- `greedy`—This tries to select a possibly suboptimal initial path by selecting the pairwise shortest distances between cities. This will not lead to the shortest path, but it may be better than the random initialization.

It is worth noting that the results of the SA algorithm may not be exactly repeatable. Due to the randomness included in the algorithm, each time you run the algorithm, you may get slightly different results. To avoid this, the `run` function included in the `SimulatedAnnealing` class contains a `repetition` argument that allows you to report the best solution generated out of multiple runs, as follows:

```
run(self, problem_obj=None, stoping_val=None, init=None, repetition=1)
```

You can set repetition to be 10, so the algorithm reports the best solution generated out of 10 runs.

5.6 Solving a delivery semi-truck routing problem

Let's consider a more real-life example of TSP. Assume that Walmart Supercenters are points of interest (POIs) to be visited by a delivery semi-truck. The vehicle will start from Walmart Supercenter number 3001, located at 270 Kingston Rd. E in Ajax, Ontario. It is required to find the shortest possible route the truck can follow to visit each POI only once and get back to the home location. There are 18 Walmart Supercenters in the selected part of the Greater Toronto Area (GTA), as shown in figure 5.16. This results in 18! possible routes to visit these stores located in Durham Region, York Region, and Toronto, Ontario.

Figure 5.16 Selected Walmart Supercenters in the Greater Toronto Area (GTA)

The GPS coordinates (longitude and latitude) of each POI and the addresses are available on the POI Factory website (www.poi-factory.com/node/25560) and are included in the Walmart_United States&Canada.csv file that can be downloaded for free (for noncommercial use) after registration. Google Places API, Here Places API, and SafeGraph on ArcGIS Marketplace can also be used to get data about points of interest such as hospitals, restaurants, retail stores, and grocery stores. Appendix B provides more details about open data sources.

The OSMnx library is used in this example to create a NetworkX graph that represents the supercenter locations. Pyrosm can also be used instead of OSMnx. The shortest distances between these locations are computed using the NetworkX built-in function `shortest_path`, which uses Dijkstra's algorithm as a default method (see section 3.4.1). Supercenter locations are rendered on OpenStreetMap based on their GPS

coordinates using the folium library. Appendix A provides more details about these libraries.

In our implementation, the problem solver is decoupled from the problem object. We start by creating a TSP object for this discrete problem. We then create an SA object to solve the TSP problem. An initial solution is generated using the `mutate` method. As shown in figure 5.17, this initial solution is far from optimal. The total length of the initial route is 593.88 km, and this route is not convenient or easy to follow in practice.

Figure 5.17 Initial solution for the Walmart delivery semi-truck route with a total distance of 593.88 km

Let's run SA with the following parameters:

- Maximum number of iterations = 10000
- Maximum interaction per temperature = 100
- Initial temperature = 85
- Final temperature = 0.0001
- Linear cooling schedule

Other cooling schedules could also be used. For example, geometric cooling can generate consistent, superior-quality, and timely solutions compared to other schemes. However, this is one of the algorithm parameters that can be tuned, as it sometimes depends on the nature of the problem. Figure 5.18 shows the shortest route with a total distance of 227.17 km.

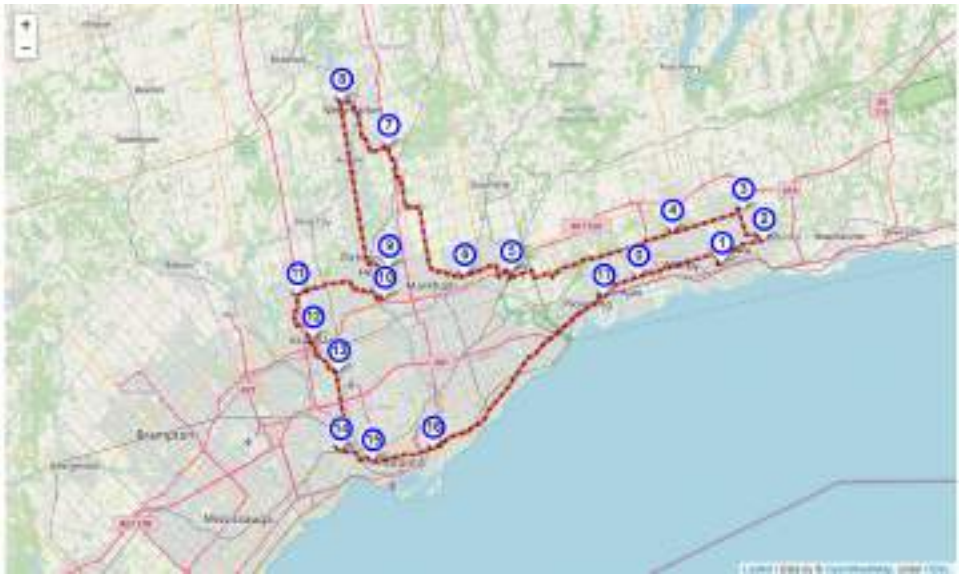


Figure 5.18 SA solution for the Walmart delivery semi-truck route with a total distance of 227.17 km

The next listing is a snippet of the code used to generate the shortest route for the delivery semi-truck using SA. The complete code is available in the book's GitHub repo.

Listing 5.6 Generating a Walmart delivery semi-truck route using SA

```
from optalgotools.algorithms import SimulatedAnnealing
from optalgotools.problems import TSP
import numpy as np
import pandas as pd
import osmnx as ox
import networkx as nx
import folium
import folium.plugins

wal_df = pd.read_csv("https://raw.githubusercontent.com/Optimization-
Algorithms-
➥Book/Code-Listings/main/Appendix%20B/data/TSP/Walmart_ON.csv")

cities_list = [city for city, region in cityToRegion.items() if city in
➥wal_df.city.unique() and region in ['Durham Region', 'York Region',
'Toronto']]

gta_part = wal_df[wal_df.store_number.str.startswith('Walmart Supercentre') &
➥ wal_df.city.isin(cities_list)].reset_index(drop=True)
wal_gta_count = gta_part.shape[0]
```

Load a list of all Walmart locations in Ontario.

Select cities that are in Durham Region, York Region, or Toronto.

Select Walmart stores that are in the preceding list and are Supercenters.

Get the lat. and long. locations of the preceding set of Walmarts, and create a graph of roads that connects them and that is within 42 km.

```
gta_part_loc = gta_part[['latitude', 'longitude']]

G = ox.graph_from_point(tuple(gta_part_loc.mean().to_list()), dist=42000,
➥ dist_type='network', network_type='drive', clean_periphery=True,
simplify=True,
➥ retain_all=True, truncate_by_edge=True)

gta_part['osmid'], gta_part['osmid_dist_m'] = zip(*gta_part.apply(lambda row:
➥ox.nearest_nodes(G, row.longitude, row.latitude, return_dist=True), axis =
1))
```

Calculate the distances between the Walmart locations using the graph.

```
gta_part_dists = np.zeros([wal_gta_count, wal_gta_count])
gta_part_pathes = [[[] for i in range(wal_gta_count)] for j in range(wal_gta_
count)]
for i in range(wal_gta_count):
    for j in range(wal_gta_count):
        if i==j:
            continue
        gta_part_pathes[i][j] = nx.shortest_path(G=G, source=gta_part.
osmid[i],
➥ target=gta_part.osmid[j], weight='length', method='dijkstra')
        gta_part_dists[i][j] = nx.shortest_path_length(G=G,
➥source=gta_part.osmid[i], target=gta_part.osmid[j], weight='length',
➥ method='dijkstra')/1000
```

Create a TSP object for the problem.

```
gta_part_tsp = TSP(dists=gta_part_dists, gen_method='mutate')

sa = SimulatedAnnealing(max_iter=1000, max_iter_per_temp=100, initial_
temp=85,
➥ final_temp=0.0001, cooling_schedule='linear')
```

Create an SA object to help solve the TSP problem.

```
sa.init_annealing(gta_part_tsp)
```

Get an initial random solution, and check its length.

```
sa.run(gta_part_tsp)
```

Run SA, and evaluate the best solution distance.

As you can see, we separated the solver class from the problem object in optalgotools. The solver is imported from `algorithms`, and the problem is an instance of the TSP problem in the `problems` class. This implementation allows you to change the problem instances and tune the parameters of the algorithm to reach an optimal or near-optimal solution. You may consider trying the adaptation aspects of SA explained in section 5.2.5 to figure out their effect on the algorithm's performance in terms of the length of the obtained route and the run time (CPU time and wall-clock time).

A metaheuristic algorithm like SA seeks optimal or near-optimal solutions at a reasonable computational cost, but it cannot guarantee either their feasibility or degree of optimality. With proper parameter tuning, the algorithm can provide acceptable solutions without further postprocessing. In the next chapter, we will discuss tabu search as another trajectory-based optimization algorithm.

Summary

- Metaheuristic algorithms can be broadly classified into trajectory-based algorithms and population-based algorithms. A trajectory-based metaheuristic algorithm, or S-metaheuristic, uses a single search agent that moves through the design or search space in a piecewise style. Population-based algorithms, or P-metaheuristics, use multiple agents to search for an optimal or near-optimal global solution. Simulated annealing is a trajectory-based metaheuristic algorithm.
- Simulated annealing mimics the annealing process in material processing, where a metal cools and freezes into a crystalline state with the minimum energy and larger crystal size so as to reduce the defects in metallic structures. The annealing process involves the careful control of temperature and cooling rate, often called the annealing schedule.
- Simulated annealing runs a series of moves under different thermodynamic conditions and always accepts improving moves and can probabilistically accept non-improving moves.
- The acceptance probability is proportional to the temperature. A high temperature increases the chance of accepting non-improving moves to favor exploration of the search space at the beginning of the search. As the search progresses, the temperature is decremented to restrict exploration and favor exploitation.
- As the temperature goes to zero, SA acts greedily like hill climbing, and as the temperature goes to infinity, SA behaves like a random walk. The temperature should decrease gradually to achieve the best trade-off between exploration and exploitation.
- Simulated annealing is a stochastic search algorithm and a derivative-free solver that can be used when derivative information is unavailable, unreliable, or prohibitively expensive. SA seeks optimal or near-optimal solutions at a reasonable computational cost but it cannot guarantee either the feasibility or degree of optimality.
- Adaptive simulated annealing can dynamically change its parameters with the search progress to control the exploration and exploitation behavior.
- Simulated annealing is an easy-to-implement probabilistic approximation algorithm that can be used to solve continuous and discrete problems in different domains.