CHAPTER 9

Simulated Annealing

> We conjecture that the analogy with thermodynamics can offer a new insight into optimization problems and can suggest efficient algorithms for solving them.
>
> —V. Černý [Černý, 1985]

Simulated annealing (SA) is an optimization algorithm that is based on the cooling and crystallizing behavior of chemical substances. The literature often distinguishes SA from EAs because SA does not involve a population of candidate solutions. SA is a single-individual stochastic algorithm. However, the (1+1)-ES is actually a special case of an SA algorithm [Droste et al., 2002], so we can reasonably consider SA as an EA.

SA was first presented in its current form by Scott Kirkpatrick, Charles Gelatt, and Mario Vecchi in 1983 for the optimal solution of problems related to computer design, such as component placement and wire routing [Kirkpatrick et al., 1983]. SA was independently derived by Vlado Černý in 1985, who used it to solve the traveling salesman problem [Černý, 1985]. An optimization algorithm very similar to SA was developed in by Martin Pincus in the late 1960s [Pincus, 1968a], [Pincus, 1968b]. SA is sometimes called the Metropolis algorithm because it is closely related to the work of Nicholas Metropolis [Metropolis et al., 1953], whose development of an algorithm for investigating the properties of interacting particles formed the foundation for SA. Finally, SA is also sometimes called the Metropolis-Hastings

algorithm due to the work of W. Keith Hastings [Hastings, 1970], who generalized the results of Metropolis et al.

Overview of the Chapter

Section 9.1 gives a brief discussion of statistical mechanics, which is the foundational principle of SA. Section 9.2 presents a simple SA algorithm. Section 9.3 discusses various cooling schedules, which is the primary tuning parameter for SA, and which has the strongest effect on its performance. Section 9.4 briefly discusses a few implementation issues, including ways that we can generate new candidate solutions in SA, when to reinitialize the cooling temperature, and why we need to keep track of the best candidate solution.

9.1 ANNEALING IN NATURE

Crystalline lattices are fascinating examples of the optimization ability of nature. A crystalline lattice is an arrangment of atoms or molecules in a liquid or solid. Some familiar examples that are common to most people's everyday experiences are the crystalline structures of quartz, ice, and salt. At high temperatures, crystalline materials don't exhibit much structure; high temperatures give the materials a lot of energy, which contributes to a lot of vibration and disorder. However, as the temperature decreases, the crystalline materials settle into a more ordered state. The particular state into which they settle is not always the same. A material that is heated and then cooled multiple times will settle into a different equilibrium state every time, but every equilibrium state tends to have low energy. Figure 9.1 compares a crystalline structure with a high entropy (a high level of disorder) at a high temperature, and one with a low entropy (a high level of order) at a low temperature. The process of heating and cooling a material to recrystallize it is called *annealing*.

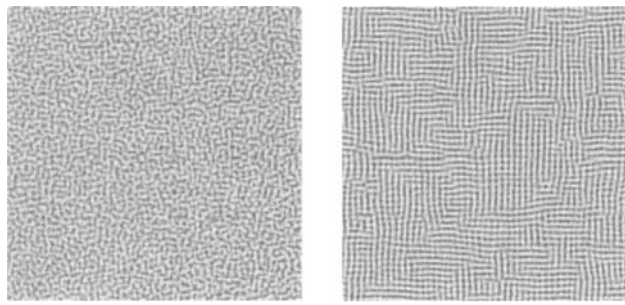

Figure 9.1 This figure gives a conceptual view of the annealing process. The figure on the left shows the disordered, high-energy state of a crystalline structure at a high temperature. The figure on the right shows the ordered, low-energy state of the same structure after it has cooled. (This figure was copied from http://en.wikipedia.org/wiki/Simulated_annealing and is distributed under the provisions of the GNU Free Documentation License.)

SA is based on statistical mechanics, which is the study of the behavior of large numbers of interacting particles, such as atoms in a gas. The number atoms in any material is on the order of 10^{23} per cubic centimeter, so when we examine the properties of a material, we only observe properties that are highly likely to occur. We also notice that equilibrium-energy configurations, and configurations that are similar to them, are observed very often, even though those configurations comprise only a tiny fraction of the possible configurations. This is because materials tend to converge to minimum-energy states; that is, nature is an optimizer.

Suppose that we use $E(s)$ to denote the energy of a specific configuration s of the atoms in some material. The probability that the system of atoms is in configuration s is given as

$$P(s) = \frac{\exp[-E(s)/(kT)]}{\sum_w \exp[-E(w)/(kT)]} \tag{9.1}$$

where k is Boltzmann's constant, T is the temperature of the system at equilibrium, and the sum in the denominator is taken over all posible configurations w [Davis and Steenstrup, 1987]. Now suppose that we have a system that is in configuration q, and we randomly select a configuration r that is a candidate for the system configuration at the next time step. If $E(r) < E(q)$, then we accept r as the configuration at the next time step with probability one:

$$P(r|q) = 1 \text{ if } E(r) < E(q). \tag{9.2}$$

That is, if our candidate configuration r has an energy that is less than that of s, we automatically move to r at the next time step. However, if $E(r) \geq E(q)$, then we move to r at the next time step with a probability that is proportional to the relative energy of q and r:

$$P(r|q) = \exp[(E(q) - E(r))/(kT)] \text{ if } E(r) \geq E(q). \tag{9.3}$$

That is, there is a nonzero probability $P(r|q)$ that the system moves to a configuration with higher energy. If $E(r) > E(q)$, then Equation (9.3) shows that the probability $P(r|q)$ that the system transitions from state q to state r is less than 1, but it increases as T increases. If we use the transition rules of Equations (9.2) and (9.3), then as time $\rightarrow \infty$, the probability that the system is in some configuration s converges to the Boltzmann distribution of Equation (9.1).

9.2 A SIMPLE SIMULATED ANNEALING ALGORITHM

Since annealing in nature results in low-energy configurations of crystals, we can simulate it in an algorithm to minimize cost functions. We start with a candidate solution s to some minimization problem. We also start with a high "temperature" so that the candidate solution is likely to change to some other configuration. We randomly generate an alternative candidate solution r and measure its cost, which is analogous to the energy of a crystalline structure. If the cost of r is less than that of s, then we update the candidate solution accordingly, as indicated by Equation (9.2). If the cost of r is greater than or equal to that of s, then we update the candidate solution with some probability less than or equal to one, as indicated by Equation (9.3). SA is sometimes called Boltzmann annealing because of its use of

Equations (9.2) and (9.3). As time progresses (that is, as the the iteration number increases), we decrease the temperature. This results in a tendency of the candidate solution to settle in a low-cost state. The analogies between annealing in nature and the SA algorithm are summarized as follows.

Annealing in Nature		Simulated Annealing
atomic configuration	$\longleftrightarrow$	candidate solution
temperature	$\longleftrightarrow$	tendency to explore search space
cooling	$\longleftrightarrow$	decreasing tendency to explore
changes to atomic configurations	$\longleftrightarrow$	changes to candidate solutions

We see that SA includes many standard EA behaviors. Although we have used annealing in nature to motivate SA in this chapter, SA can in fact be developed without any motivation from nature [Michiels et al., 2007]. There are advantages to both approaches. Appealing to nature may open up new avenues of SA research, but it may also limit the possible extensions to SA. A simple SA algorithm is shown in Figure 9.2.

T = initial temperature > 0
$\alpha(T)$ = cooling function: $\alpha(T) \in [0, T]$ for all T
Initialize a candidate solution x_0 to minimization problem $f(x)$
While not(termination criterion)
    Generate a candidate solution x
    If $f(x) < f(x_0)$
        $x_0 \leftarrow x$
    else
        $r \leftarrow U[0, 1]$
        If $r < \exp[(f(x_0) - f(x))/T]$ then
            $x_0 \leftarrow x$
        End if
    End if
    $T \leftarrow \alpha(T)$
Next iteration

Figure 9.2 A basic simulated annealing algorithm for the minimization of $f(x)$. The function $U[0, 1]$ returns a random number uniformly distributed on [0, 1].

Figure 9.2 shows that the basic SA algorithm has features in common with most other EAs. First, it is simple and intuitively appealing. Second, it is based on an optimization process in nature. Third, it has several tuning parameters that can each have a significant impact on its performance.

- The initial temperature T provides an upper bound for the relative importance of exploration versus exploitation. If the initial temperature is too low, then the algorithm will not effectively explore the search space. If the initial temperature is too high, then the algorithm will take too long to converge.

- The cooling schedule $\alpha(T)$ controls the rate of convergence. At the beginning of the algorithm, exploration is high and exploitation is low. At the end of the algorithm, the converse is true: exploitation is high and exploration is low. The cooling schedule controls the transition from exploration to exploitation. If $\alpha(T)$ is too drastic, then, like a crystalline structure that cools too rapidly, the annealing process will converge to a disordered (high cost) state. If $\alpha(T)$ is too gradual, then the annealing process will take too long to converge. We discuss cooling schedules in Section 9.3.

- The strategy used to generate a candidate solution x at each iteration can have a significant impact on SA performance. Random generation of x may work, but a more intelligent method for trying to generate an x that is better than x_0 will probably give better performance. We discuss candidate generation strategies in Section 9.4.1.

A simplified SA algorithm can be implemented by replacing the acceptance test in Figure 9.2 as follows:

$$\text{Replace "If } r < \exp[(f(x_0) - f(x))/T]\text{" with "If } r < \exp[-c/T]\text{"} \tag{9.4}$$

where c is called the *acceptance probability constant.* This indicates that if the candidate solution x has a higher cost than x_0, then the probability of replacing x_0 with x is independent of its cost. The acceptance probability constant c controls exploration versus exploitation. If c is too large, then the algorithm will not explore the search space aggressively enough. If c is too small, then the algorithm will explore too aggressively without exploiting good solutions that it has previously discovered.

9.3 COOLING SCHEDULES

This section discusses different cooling schedules $\alpha(T)$ that can be used in the SA algorithm of Figure 9.2. The cooling schedule can have a significant impact on SA performance. If an SA implementation does not work on some problem, it may be because the cooling schedule is not appropriate for the problem. Some commonly use cooling schedules include linear cooling, exponential cooling, inverse cooling, logarithmic cooling, and inverse linear cooling, which we discuss in the following sections. We also note that optimization problems can have different scales along different dimensions, and so we discuss in dimension-dependent cooling in Section 9.3.6.

9.3.1 Linear Cooling

Linear cooling is the simplest type of cooling, and follows the schedule

$$\alpha(T) = T_0 - \eta k \tag{9.5}$$

where T_0 is the initial temperature, k is the SA iteration number, and η is a constant. We need to make sure that $T > 0$ for all k, so we should choose η such that the temperature at the maximum iteration number is positive. Alternatively, we could use the following modified form of linear cooling:

$$\alpha(T) = \max(T_0 - \eta k, T_{\min}) \tag{9.6}$$

where $T_{\min}$ is a user-specified minimum temperature.

9.3.2 Exponential Cooling

Exponential cooling follows the schedule

$$\alpha(T) = aT \tag{9.7}$$

where typically $a \in (0.8, 1)$. A larger value of a will give a slower cooling schedule. Figure 9.3 shows normalized temperature for the exponential cooling schedule for different values of a. We see that a should be quite close to 1, otherwise the cooling rate will be too drastic.

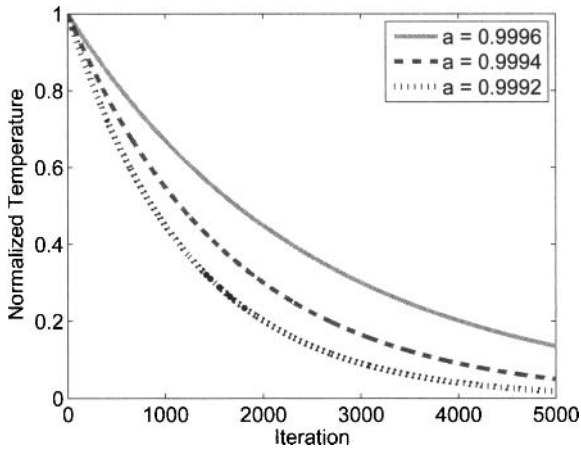

Figure 9.3 Normalized temperature as a function of a for the exponential cooling schedule. The parameter a is usually very close to 1 for this cooling schedule. The cooling rate is very sensitive to changes in a.

9.3.3 Inverse Cooling

Inverse cooling follows the schedule

$$\alpha(T) = T/(1 + \beta T) \tag{9.8}$$

where β is a small constant, typically on the order of 0.001. A smaller value of β will give a slower cooling schedule. This cooling schedule was first suggested in [Lundy and Mees, 1986]. Figure 9.4 shows normalized temperature for the inverse cooling schedule for different values of β. We see that β should be quite small, otherwise the cooling rate will be too drastic.

Comparing Figures 9.3 and 9.4, we see that the exponential cooling and inverse cooling schedules can be made to be very similar to each other by choosing appropriate values for a and β.

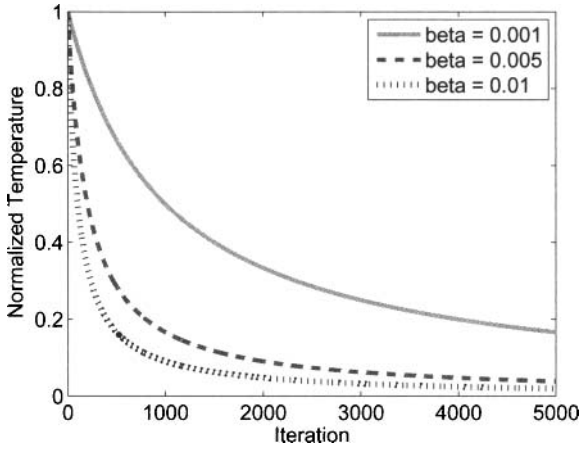

Figure 9.4 Normalized temperature as a function of β for the inverse cooling schedule. The parameter β is usually very small for this cooling schedule. The cooling rate is very sensitive to changes in β.

■ EXAMPLE 9.1

In this example, we optimize the 20-dimensional Ackley function, which is defined in Appendix C.1.2, with the SA algorithm of Figure 9.2. We use the inverse cooling function described in Equation (9.8): $T_{k+1} = T_k/(1 + \beta T_k)$, where k is the iteration number, β is the cooling schedule parameter, T_k is the temperature at the k-th iteration, and $T_0 = 100$. We use a Gaussian random number centered at x_0 to generate a new candidate solution at each iteration:

$$x \leftarrow x_0 + N(0, T_k I) \tag{9.9}$$

where $N(0, T_k I)$ is a Gaussian random vector with a mean of 0 and a covariance of $T_k I$, and I is the 20×20 identity matrix. We use the simple acceptance test of Equation (9.4) with $c = 1$.

Figure 9.5 shows the best solution found as a function of the SA iteration number, averaged over 20 Monte Carlo simulations, and for three different values of β. We see that if β is too small (0.0002), then cooling occurs too slowly and the SA algorithm jumps around too aggressively in the search space without exploiting good solutions that it has already obtained. If β is too large (0.001), then cooling occurs too quickly and the SA algorithm tends to get stuck in local minima. If β is just right (0.0005), then cooling occurs at a rate that results in the best convergence. However, we note that toward the end of the plot, the $\beta = 0.0002$ trace appears to be rapidly overtaking the $\beta = 0.0005$ trace. This indicates that although $\beta = 0.0002$ is too small to give good convergence within the iteration number limit that we have used, it will eventually result in enough cooling if the iteration number continues to increase, and will eventually converge to a good result.

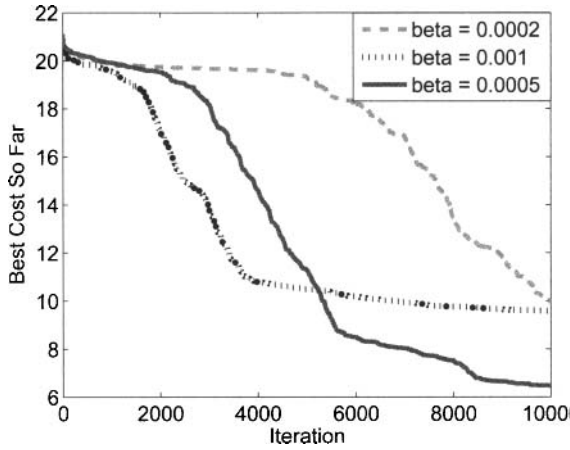

Figure 9.5 Example 9.1 simulation results of the SA algorithm for optimizing the 20-dimensional Ackley function. Results are averaged over 20 Monte Carlo simulations. The inverse cooling schedule parameter β has a significant impact on SA performance.

□

Example 9.1 shows that if cooling is too fast or too slow, then SA performance might not be good. The same conclusion can be drawn about the initial temperature T_0. In Example 9.1 we arbitrarily used $T_0 = 100$. Unfortunately, there are not any good guidelines for the selection of T_0; it depends entirely on the particular optimization problem.

9.3.4 Logarithmic Cooling

Logarithmic cooling follows the schedule

$$\alpha(T) = c/\ln k \tag{9.10}$$

where c is a constant, and k is the SA iteration number. This was first suggested in [Geman and Geman, 1984]. It is sometimes generalized to

$$\alpha(T) = c/\ln(k + d) \tag{9.11}$$

where d is a constant that is often set equal to 1 [Nourani and Andresen, 1998]. Logarithmic cooling is qualitatively different than exponential and inverse cooling, as seen from Figure 9.6. The temperature decreases very rapidly for the first few iterations, and then decreases extremely slowly. This slow decrease means that SA convergence is usually poor with the logarithmic cooling schedule. Therefore, the logarithmic cooling schedule is not recommended for practical applications.

However, the logarithmic cooling schedule is theoretically attractive and widely known in the SA community because it has been proven to give a global minimum under certain conditions [Geman and Geman, 1984]. As a simple demonstration

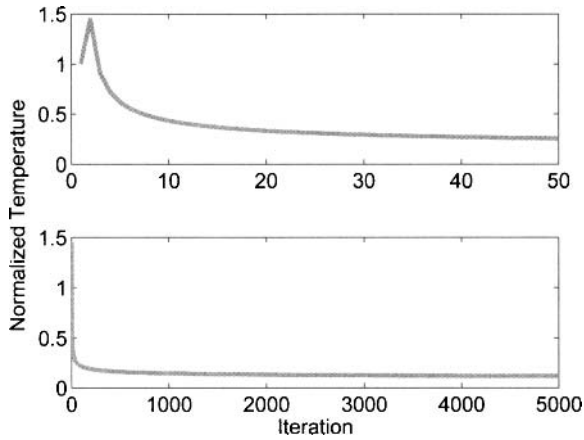

Figure 9.6 Normalized temperature for the logarithmic cooling schedule. The top figure shows the temperature for the first 50 iterations, and the bottom figure shows the temperature for the first 5,000 iterations. The temperature decreases very rapidly for the first few iterations, and then decreases so slowly that it is impractical for SA implementations.

of the proof [Ingber, 1996], suppose that we have a discrete problem so that the size of the search space is finite. We generate the candidate x from a Gaussian distribution so that the probability of generating x, given that x_k is the current candidate solution at the k-th iteration, is

$$g_k \equiv P(x|x_k) = (2\pi T_k)^{D/2} \exp\left[-||x - x_k||_2^2/(2T_k)\right] \tag{9.12}$$

where D is the problem dimension. In other words, the conditional probability of generating x given that x_k is the current candidate, is Gaussian with a mean of x_k and a covariance of $T_k I$, where T_k is the temperature at the k-th iteration, and I is the identity matrix. In order to visit every possible candidate solution in the search space, it suffices to show that as the iteration count approaches infinity, the probability of *not* visiting x approaches zero; that is,

$$\lim_{N\to\infty} \prod_{k=1}^{N} (1 - g_k) = 0. \tag{9.13}$$

Taking the natural log of the above equation gives

$$\ln\left[\lim_{N\to\infty} \prod_{k=1}^{N} (1 - g_k)\right] = \lim_{N\to\infty}\left[\ln \prod_{k=1}^{N} (1 - g_k)\right] = -\infty. \tag{9.14}$$

A Taylor series expansion of the logarithm about $g_1 = g_2 = \cdots = 0$ gives

$$\ln[(1 - g_1)(1 - g_2)\cdots] = \ln 1 - g_1 - g_2 - \cdots. \tag{9.15}$$

Combining the two previous equations gives the following sufficient condition for obtaining a 100% probability of visiting x as the iteration count approaches infinity:

$$\lim_{N\to\infty} \sum_{k=1}^{N} g_k = \infty. \tag{9.16}$$

If g_k is given by Equation (9.12), and if $T_k = T_0/\ln k$, then the left side of the above equation becomes

$$\lim_{N\to\infty} \sum_{k=1}^{N} (2\pi T_0/\ln k)^{D/2} \exp\left[-||x - x_k||_2^2/(2T_0/\ln k)\right] \geq$$

$$\sum_{k=1}^{\infty} \exp(-\ln k) = \sum_{k=1}^{\infty} 1/k = \infty \tag{9.17}$$

where the inequality is true if T_0 is large enough (see Problem 9.5).

9.3.5 Inverse Linear Cooling

Inverse linear cooling follows the schedule

$$\alpha(T) = T_0/k \tag{9.18}$$

where T_0 is the initial temperature, and k is the SA iteration number. The inverse linear cooling schedule exhibits the fast cooling of the logarithmic schedule during the first few iterations, but it avoids the nonzero temperatures and slow cooling of later iterations, as seen from Figure 9.7. The temperature decreases very rapidly and quickly reaches zero. This means that inverse linear cooling is not effective for problems that require a lot of exploration, but is more suitable for problems that can be initialized with a candidate solution that is known to be close to the optimal solution.

Inverse linear cooling, like logarithmic cooling, is theoretically attractive and widely known in the SA community because it has been proven to result in a global optimum under certain conditions [Szu and Hartley, 1987]. As a simple demonstration similar to that used in the previous section for logarithmic cooling [Ingber, 1996], suppose that we have a discrete problem so that the size of the search space is finite. We generate the candidate x from a Cauchy distribution so that the probability of obtaining x, given that x_k is the current candidate solution at the k-th iteration, is

$$g_k \equiv P(x|x_k) = \frac{T_k}{(||x - x_k||_2^2 + T_k^2)^{(D+1)/2}} \tag{9.19}$$

where D is the problem dimension. Note that in the previous section we used a Gaussian distribution to generate candidate solutions, while in this section we use a Cauchy distribution. Figure 9.8 compares a Cauchy PDF with a Gaussian PDF. We see that the Cauchy PDF has much fatter tails, which means that we are more likely to generate candidate solutions that are farther from the current candidate solution.

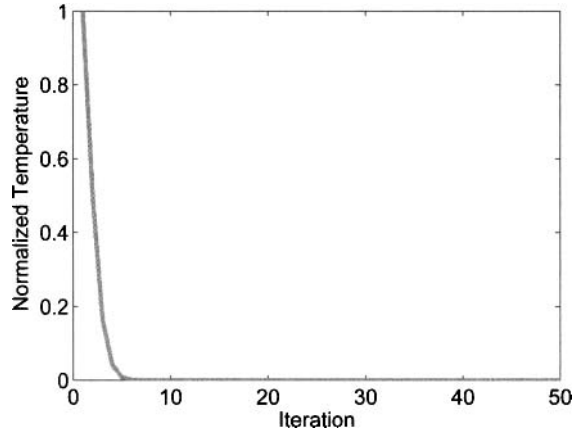

Figure 9.7 Normalized temperature for the inverse linear cooling schedule. The temperature reaches zero very quickly.

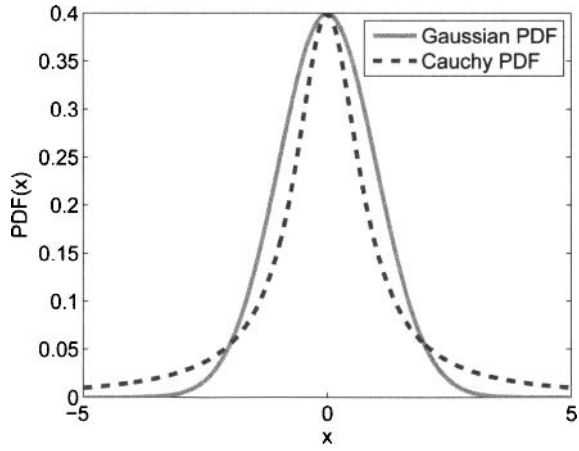

Figure 9.8 A comparison of the one-dimensional Cauchy and Gaussian probability density functions.

If g_k is given by Equation (9.19), and if $T_k = T_0/k$, then the left side of Equation (9.16) becomes

$$\lim_{N\to\infty} \sum_{k=1}^{N} \frac{T_0/k}{\left(||x - x_k||_2^2 + T_0^2/k^2\right)^{(D+1)/2}} \geq \sum_{k=1}^{\infty} 1/k = \infty \tag{9.20}$$

where the inequality is true for appropriate values of T_0 (see Problem 9.6).

Now we compare the convergence results obtained with the logarithmic and inverse linear cooling schedules. Recall that the Cauchy PDF has much fatter tails than the Gaussian PDF. Also recall that the candidate solution generation function of Equation (9.19), which uses the Cauchy PDF, is combined with the inverse linear cooling schedule of Figure 9.7. Finally, recall that the candidate solution generation function of Equation (9.12), which uses the Gaussian PDF, uses the logarithmic cooling schedule of Figure 9.6. Since the Cauchy generation function has fatter tails than the Gaussian generation function, it is guaranteed to converge with a much faster cooling schedule than the one used in conjunction with the Gaussian generation function.

9.3.6 Dimension-Dependent Cooling

In real-world applications, and even in some benchmark problems, the landscape of the cost function can look very different when viewed along different dimensions. For example, consider the function

$$\begin{aligned} f(x) &= 20 + e - 20\exp\left(-0.2\sum_{i=1}^{n} y_i^2/n\right) - \exp\left(\sum_{i=1}^{n}(\cos 2\pi y_i)/n\right) \\ y_i &= \begin{cases} x_i & \text{for odd } i \\ x_i/4 & \text{for even } i. \end{cases} \end{aligned} \tag{9.21}$$

This is simply a scaled version of the Ackley function, which is defined in Appendix C.1.2. The fact that x_i is scaled for even values of i means that the function is "stretched out" along those dimensions. Figure 9.9 shows a two-dimensional plot of this function. Because of the scaling of even dimensions, the function is much smoother along the x_2 dimension than along the x_1 dimension.

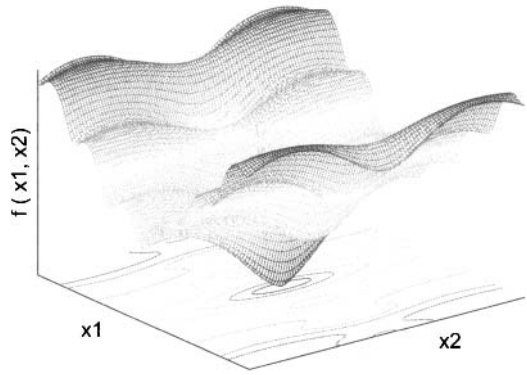

Figure 9.9 Scaled version of the two-dimensional Ackley function. The topology is much smoother along the x_2 direction, which indicates that the SA algorithm should use a slower cooling schedule along that dimension.

For functions that have different topologies along different dimensions, we might want to use a different cooling schedule for different dimensions. For the scaled Ackley function of Equation (9.21), we would want to use slower cooling along the even dimensions, and faster cooling along the odd dimensions. This will allow the SA algorithm to gradually converge to optimal values along the gradually-changing dimensions of the function. A fast cooling schedule along gradually-changing dimensions would prevent the SA algorithm from moving down gentle slopes. However, along the more dynamic dimensions of the function, a faster cooling schedule is needed. The SA algorithm will move downhill along dynamic dimensions even with a fast cooling rate, but a slow cooling rate will result in too much jumping around. As another way of looking at it, we can say that we need a more aggressive search (high temperatures) along low-sensitivity dimensions, and a less aggressive search (low temperatures) along high-sensitivity dimensions.

If we use the inverse cooling schedule of Equation (9.8), the above discussion implies a smaller value of β for the even dimensions and a larger value of β for the odd dimensions. This means that each dimension of the problem will have its own temperature. This results in a modification of the basic SA algorithm of Figure 9.2 to obtain the dimension-dependent SA algorithm of Figure 9.10.

T_i = initial temperature > 0, $i \in [1, n]$
$\alpha_i(T_i)$ = cooling function for i-th dimension, $i \in [1, n]$: $\alpha(T_i) \in [0, T_i]$ for all T_i
Initialize a candidate solution x_0 to minimization problem $f(x)$:
    $x_0 = [x_{01}, x_{02}, \cdots, x_{0n}]$
While not(termination criterion)
    Generate a candidate solution $x_1 = [x_{11}, x_{12}, \cdots, x_{1n}]$
    If $f(x_1) < f(x_0)$
        $x_0 \leftarrow x_1$
    else
        For $i = 1, \cdots, n$
            $r \leftarrow U[0, 1]$
            If $r < \exp[(f(x_0) - f(x_1))/T_i]$ then
                $x_{0i} \leftarrow x_{1i}$
            End if
        Next dimension i
    End if
    $T_i \leftarrow \alpha_i(T_i)$, $i \in [1, n]$
Next iteration

Figure 9.10 A dimension-dependent simulated annealing algorithm for the minimization of the n-dimensional function $f(x)$. The function $U[0, 1]$ returns a random number uniformly distributed on [0, 1]. This algorithm is a generalization of the basic SA algorithm of Figure 9.2; here we have allowed each dimension to have its own temperature and its own cooling schedule.

■ EXAMPLE 9.2

In this example we optimize the 20-dimensional scaled Ackley function, which is given by Equation (9.21), with the dimension-dependent SA algorithm of Figure 9.10. We use the inverse cooling function described in Equation (9.8) for each dimension: $T_{k+1,i} = T_{ki}/(1+\beta_i T_{ik})$, where i is the dimension number, k is the SA iteration number, β_i is the cooling schedule parameter for the i-th dimension, T_{ki} is the temperature at the k-th iteration of the i-th dimension, and $T_{0i} = 100$ for all i. We use a Gaussian random number centered at x_0 to generate a new candidate solution at each iteration:

$$x_{1i} \leftarrow x_{0i} + N(0, T_{ki}) \tag{9.22}$$

where $N(0, T_{ki})$ is a Gaussian random number with a mean of 0 and a variance of T_{ki}. We use the simple acceptance test of Equation (9.4) with $c = 1$.

Figure 9.11 shows the best solution found as a function of the SA iteration number, averaged over 20 Monte Carlo simulations, and for four different combinations of β. We see that if β is too small (0.001), then cooling occurs too slowly and the SA algorithm jumps around too aggressively in the search space without exploiting good solutions that it has already obtained. If β is too large (0.005), then cooling occurs too quickly and the SA algorithm tends to get stuck in local minima. However, if β is large for odd dimensions and small for even dimensions, then cooling occurs at a rate that results in the best convergence. This combination gives fast cooling for the highly dynamic odd dimensions, and slow cooling for the even dimensions.

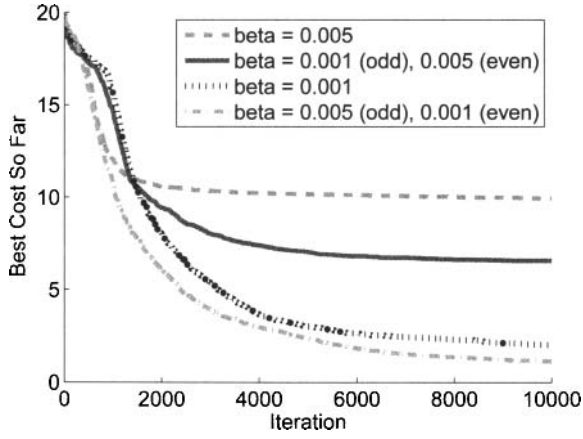

Figure 9.11 Example 9.2 simulation results of the dimension-dependent SA algorithm optimizing the 20-dimensional scaled Ackley function. Results are averaged over 20 Monte Carlo simulations. The cooling schedule parameters $\{\beta_i\}$ can be adjusted individually for each dimension to give the best results.

□

9.4 IMPLEMENTATION ISSUES

This section discusses a couple of implementation issues, including how to generate candidate solutions, when to reinitialize the cooling temperature, and why we need to keep track of the best candidate solution.

9.4.1 Candidate Solution Generation

The statement "Generate a candidate solution" in the SA algorithms of Figures 9.2 and 9.10 is deceptively simple. There are many different ways that this statement can be implemented, and the implementation choice can have a large impact on SA performance. One method of generating candidate solutions is to simply choose a random point in the search space. However, after the SA algorithm has begun converging to a good solution, we would expect that the current solution candidate x_0 is much better than most other points in the search space. Therefore, generating a random solution candidate will probably not be very effective. As a general rule, we should bias candidate solution generation toward the current candidate solution x_0. This is the reason that Equations (9.9) and (9.22) use a Gaussian random variable centered at x_0 for candidate solution generation. Furthermore, the variance of Gaussian random variable is equal to the temperature, which decreases with time, and so the search tends to narrow as the SA iteration count increases. Equation (9.19), the Cauchy distribution, can be used as a more aggressive method for generating candidate solutions while still being centered at x_0. Biasing candidate solution generation toward x_0 tends to exclude not only very poor candidates, but also very good candidates. However, very poor points in the search space are generally more common than very good candidates, so biasing the search toward x_0 is usually effective.

9.4.2 Reinitialization

As we discussed earlier in this chapter, the cooling schedule is an important contributor to SA performance. If we cool the temperature too quickly, then the SA will get stuck in a local optimum and performance will be poor. However, we usually do not know ahead of time what the appropriate cooling schedule is. Therefore, we often monitor the improvement of the SA algorithm, and if we do not find a better candidate solution within L iterations, we reinitialize the temperature to T_0 to increase exploration.

9.4.3 Keeping Track of the Best Candidate Solution

Recall from Figure (9.2) that a new candidate solution x might replace a current candidate solution x_0, even if x is worse than x_0. This is a necessary risk to sufficiently explore the search space, but it might result in the loss of a good candidate solution. Therefore, we usually want to implement an archive in SA so that we keep track of the best candidate solution obtained so far. This is similar to elitism in Section 8.4; however, in that section we actually retained the best candidate solutions in the population. We cannot do that directly in SA unless we increase the population size beyond 1, which is a possibility that we have not discussed in this chapter. However, regardless of the population size, we can always maintain

an archive that contains the best candidate found so far. So even if we replace a good candidate solution with a poor candidate due to the exploratory nature of SA, we will still keep track of the best candidate found so far. The best candidate solution in the archive will never be replaced with a worse candidate. Then we can return the best candidate found when the SA algorithm is complete.

9.5 CONCLUSION

Simulated annealing is one of the older EAs, originating in 1983, but we have discussed in this part of the book because it is not always considered to be a classic EA. It is not population based, but it is clear that some of the classic EAs are not population based either, and so that is not a sufficient reason to remove it from the EA category. Since SA is based on a natural process, and since it is an iterative optimization algorithm, we generally consider it to be an EA. Its maturity and scientific roots have resulted in many papers, books, and applications. Readers who want a more comprehensive coverage of SA are recommended to the books [van Laarhoven and Aarts, 2010], [Otten and van Ginneken, 1989], and [Aarts and Korst, 1989]. Tutorial chapters are available at [Aarts et al., 2003] and [Henderson et al., 2003].

Like all of the EAs discussed in this book, SA can be useful for a wide variety of optimization problems, including both continuous-domain and discrete-domain problems. Current research directions in the area of SA mirrors current emphases in general EA research: SA for multi-objective problems [Bandyopadhyay et al., 2008], hybridizations of SA with other EAs [Cakir et al., 2011], parallelization [Zimmerman and Lynch, 2009], and constrained optimization [Singh et al., 2010].

This chapter has presented the background and implementation of SA, but there are other important aspects of SA that we have not had time to discuss. For instance, a Markov model and some theoretical convergence proofs are presented in [Michiels et al., 2007], although there is still much room for additional modeling and theoretical results. Practitioners are interested not only in convergence, but also in performance over finite time intervals, and this issue is discussed in [Henderson et al., 2003], [Vorwerk et al., 2009].