**What is Simulated Annealing**

In the world of optimization, finding the best solution to complex problems can be challenging, especially when the solution space is vast and filled with local optima. One powerful method for overcoming this challenge is ***Simulated Annealing (SA)***. Inspired by the physical process of annealing in metallurgy, Simulated Annealing is a probabilistic technique used for solving both combinatorial and continuous optimization problems.

**What is Simulated Annealing?**

Simulated Annealing is an optimization algorithm designed to search for an optimal or near-optimal solution in a large solution space. The name and concept are derived from the process of annealing in metallurgy, where a material is heated and then slowly cooled to remove defects and achieve a stable crystalline structure. In Simulated Annealing, the "heat" corresponds to the degree of randomness in the search process, which decreases over time (cooling schedule) to refine the solution. The method is widely used in combinatorial optimization, where problems often have numerous local optima that standard techniques like gradient descent might get stuck in. Simulated Annealing excels in escaping these local minima by introducing controlled randomness in its search, allowing for a more thorough exploration of the solution space.

**How Simulated Annealing Works**

The algorithm starts with an initial solution and a high "temperature," which gradually decreases over time. Here’s a step-by-step breakdown of how the algorithm works:

* **Initialization:**Begin with an initial solution Sο*Sο* and an initial temperature Tο*Tο*. he temperature controls how likely the algorithm is to accept worse solutions as it explores the search space.
* **Neighborhood Search**: At each step, a new solution S′*S*′ is generated by making a small change (or perturbation) to the current solution S.
* **Objective Function Evaluation:** The new solution S' is evaluated using the objective function. If S' provides a better solution than S, it is accepted as the new solution.
* **Acceptance Probability**: If S' is worse than S, it may still be accepted with a probability based on the temperature and the difference in objective function values.

The acceptance probability is given by:

*A mathematical equation with black text

AI-generated content may be incorrect.*

* Cooling Schedule: After each iteration, the temperature is decreased according to a predefined cooling schedule, which determines how quickly the algorithm converges. Common cooling schedules include linear, exponential, or logarithmic cooling.
* Termination: The algorithm continues until the system reaches a low temperature (i.e., no more significant improvements are found), or a predetermined number of iterations is reached.

**Cooling Schedule and Its Importance**

The cooling schedule plays a crucial role in the performance of Simulated Annealing. If the temperature decreases too quickly, the algorithm might converge prematurely to a suboptimal solution (local optimum). On the other hand, if the cooling is too slow, the algorithm may take an excessively long time to find the optimal solution. Hence, finding the right balance between exploration (high temperature) and exploitation (low temperature) is essential.

**Advantages of Simulated Annealing**

* **Ability to Escape Local Minima**: One of the most significant advantages of Simulated Annealing is its ability to escape local minima. The probabilistic acceptance of worse solutions allows the algorithm to explore a broader solution space.
* **Simple Implementation**: The algorithm is relatively easy to implement and can be adapted to a wide range of optimization problems.
* **Global Optimization**: Simulated Annealing can approach a global optimum, especially when paired with a well-designed cooling schedule.
* **Flexibility:** The algorithm is flexible and can be applied to both continuous and discrete optimization problems.

**Limitations of Simulated Annealing**

* **Parameter Sensitivity:** The performance of Simulated Annealing is highly dependent on the choice of parameters, particularly the initial temperature and cooling schedule.
* **Computational Time:** Since Simulated Annealing requires many iterations, it can be computationally expensive, especially for large problems.
* **Slow Convergence:** The convergence rate is generally slower than more deterministic methods like gradient-based optimization.

**Applications of Simulated Annealing**

* Simulated Annealing has found widespread use in various fields due to its versatility and effectiveness in solving complex optimization problems. Some notable applications include:
* Traveling Salesman Problem (TSP): In combinatorial optimization, SA is often used to find near-optimal solutions for the TSP, where a salesman must visit a set of cities and return to the origin, minimizing the total travel distance.
* VLSI Design: SA is used in the physical design of integrated circuits, optimizing the layout of components on a chip to minimize area and delay.
* Machine Learning: In machine learning, SA can be used for hyperparameter tuning, where the search space for hyperparameters is large and non-convex.
* Scheduling Problems: SA has been applied to job scheduling, minimizing delays and optimizing resource allocation.
* Protein Folding: In computational biology, SA has been used to predict protein folding by optimizing the conformation of molecules to achieve the lowest energy state.

**Comparison to Other Optimization Techniques**

Simulated Annealing is often compared to other global optimization techniques like [Genetic Algorithms (GA)](https://www.geeksforgeeks.org/dsa/genetic-algorithms/) and[Particle Swarm Optimization (PSO).](https://www.geeksforgeeks.org/machine-learning/particle-swarm-optimization-pso-an-overview/)

* Simulated Annealing vs. Genetic Algorithms: While both methods are probabilistic and capable of escaping local minima, GAs use a population of solutions and evolve them over generations, whereas SA works with a single solution that is iteratively improved.
* Simulated Annealing vs. Gradient Descent: Gradient descent is faster but can easily get stuck in local minima. Simulated Annealing, on the other hand, can escape local minima but is generally slower.

**Conclusion**

Simulated Annealing is a robust optimization technique that mimics the physical process of annealing to find optimal or near-optimal solutions in large and complex search spaces. Its ability to escape local minima, combined with its simple implementation, makes it a valuable tool in various applications, from combinatorial optimization to machine learning and beyond. However, its reliance on a well-designed cooling schedule and its relatively slow convergence can limit its efficiency. Despite these challenges, Simulated Annealing remains a popular choice for solving optimization problems where traditional methods struggle.

**Implement Simulated Annealing in Python**

Simulated Annealing (SA) is a probabilistic technique used for finding an approximate solution to an optimization problem. It is particularly useful for large search spaces where finding the exact solution is impractical. The algorithm is inspired by the annealing process in metallurgy.

**Step-by-Step Simulated Annealing in Python**

**Step 1: Understanding Simulated Annealing**

Simulated Annealing is inspired by the annealing process in metallurgy. The key idea is to use a "temperature" parameter that gradually decreases over time, allowing the algorithm to explore the solution space more freely at high temperatures and refine the search as the temperature cools down.

**Step 2: Defining the Objective Function**

The objective function is the function we want to optimize (minimize or maximize). For this example, we'll use the Rastrigin function, which is a common benchmark function for optimization algorithms.

import random

​

def get\_neighbor(x, step\_size=0.1):

neighbor = x[:]

index = random.randint(0, len(x) - 1)

neighbor[index] += random.uniform(-step\_size, step\_size)

return neighbor

**Step 3: Creating the Neighbor Function**

The neighbor function generates a new candidate solution by making a small random change to the current solution.

import random

​

def get\_neighbor(x, step\_size=0.1):

neighbor = x[:]

index = random.randint(0, len(x) - 1)

neighbor[index] += random.uniform(-step\_size, step\_size)

return neighbor

**Step 4: Implementing the Simulated Annealing Algorithm**

Now we'll implement the Simulated Annealing algorithm. The main components are initializing the solution, updating the temperature, generating new candidates, and deciding whether to accept them.

def simulated\_annealing(objective, bounds, n\_iterations, step\_size, temp):

# Initial solution

best = [random.uniform(bound[0], bound[1]) for bound in bounds]

best\_eval = objective(best)

current, current\_eval = best, best\_eval

scores = [best\_eval]

​

for i in range(n\_iterations):

# Decrease temperature

t = temp / float(i + 1)

# Generate candidate solution

candidate = get\_neighbor(current, step\_size)

candidate\_eval = objective(candidate)

# Check if we should keep the new solution

if candidate\_eval < best\_eval or random.random() < math.exp((current\_eval - candidate\_eval) / t):

current, current\_eval = candidate, candidate\_eval

if candidate\_eval < best\_eval:

best, best\_eval = candidate, candidate\_eval

scores.append(best\_eval)

​

# Optional: print progress

if i % 100 == 0:

print(f"Iteration {i}, Temperature {t:.3f}, Best Evaluation {best\_eval:.5f}")

​

return best, best\_eval, scores

**Step 5: Running the Algorithm**

Define the problem domain, set the parameters, and run the Simulated Annealing algorithm.

# Define problem domain

bounds = [(-5.0, 5.0) for \_ in range(2)] # for a 2-dimensional Rastrigin function

n\_iterations = 1000

step\_size = 0.1

temp = 10

​

# Perform the simulated annealing search

best, score, scores = simulated\_annealing(objective\_function, bounds, n\_iterations, step\_size, temp)

​

print(f'Best Solution: {best}')

print(f'Best Score: {score}')

**Implementation of Simulated Annealing in Python**

Here's the complete code with all the steps combined:

import math

import random

​

# Objective function: Rastrigin function

def objective\_function(x):

return 10 \* len(x) + sum([(xi\*\*2 - 10 \* math.cos(2 \* math.pi \* xi)) for xi in x])

​

# Neighbor function: small random change

def get\_neighbor(x, step\_size=0.1):

neighbor = x[:]

index = random.randint(0, len(x) - 1)

neighbor[index] += random.uniform(-step\_size, step\_size)

return neighbor

​

# Simulated Annealing function

def simulated\_annealing(objective, bounds, n\_iterations, step\_size, temp):

# Initial solution

best = [random.uniform(bound[0], bound[1]) for bound in bounds]

best\_eval = objective(best)

current, current\_eval = best, best\_eval

scores = [best\_eval]

​

for i in range(n\_iterations):

# Decrease temperature

t = temp / float(i + 1)

# Generate candidate solution

candidate = get\_neighbor(current, step\_size)

candidate\_eval = objective(candidate)

# Check if we should keep the new solution

if candidate\_eval < best\_eval or random.random() < math.exp((current\_eval - candidate\_eval) / t):

current, current\_eval = candidate, candidate\_eval

if candidate\_eval < best\_eval:

best, best\_eval = candidate, candidate\_eval

scores.append(best\_eval)

​

# Optional: print progress

if i % 100 == 0:

print(f"Iteration {i}, Temperature {t:.3f}, Best Evaluation {best\_eval:.5f}")

​

return best, best\_eval, scores

​

# Define problem domain

bounds = [(-5.0, 5.0) for \_ in range(2)] # for a 2-dimensional Rastrigin function

n\_iterations = 1000

step\_size = 0.1

temp = 10

​

# Perform the simulated annealing search

best, score, scores = simulated\_annealing(objective\_function, bounds, n\_iterations, step\_size, temp)

​

print(f'Best Solution: {best}')

print(f'Best Score: {score}')

**Output**

Iteration 0, Temperature 10.000, Best Evaluation 53.95166

Iteration 100, Temperature 0.099, Best Evaluation 49.75767

Iteration 200, Temperature 0.050, Best Evaluation 49.74828

Iteration 300, Temperatu...

**1. Overview**

In this tutorial, we’ll review the Simulated Annealing (SA), a metaheuristic algorithm commonly used for optimization problems with large search spaces. Additionally, we illustrate the SA optimization procedure and show how to minimize a function.

**2. The Physics Behind the Algorithm**

In general, SA is a metaheuristic optimization technique introduced by [Kirkpatrick et al.](https://www.dcs.gla.ac.uk/~pat/ads2/java/TxSxP/papers/sa.pdf) in 1983 to solve the [Travelling Salesman Problem](https://www.baeldung.com/java-simulated-annealing-for-traveling-salesman#overview-2) (TSP).

The SA algorithm is based on the [annealing process](https://en.wikipedia.org/wiki/Annealing_(materials_science)) used in metallurgy, where a metal is heated to a high temperature quickly and then gradually cooled. At high temperatures, the atoms move fast. Moreover, when the temperature is reduced, their kinetic energy decreases as well. At the end of the annealing process, the atoms fall into a more ordered state. Additionally, the material becomes more ductile and easier to work with.

**Similarly, in SA, a search process starts with a high-energy state (an initial solution) and gradually lowers the temperature (a control parameter) until it reaches a state of minimum energy (the optimal solution).**

We can apply SA in a wide range of optimization problems, such as TSP, [protein folding](https://en.wikipedia.org/wiki/Protein_structure_prediction), graph partitioning, and [job-shop scheduling](https://en.wikipedia.org/wiki/Job-shop_scheduling). In particular, **the main advantage of SA is its ability to escape from local minima and converge to a global minimum**. Moreover, SA is relatively easy to implement and doesn’t require prior knowledge of the search space.

**3. Algorithm**

At first, the simulated annealing process starts with an initial solution. Furthermore, SA iteratively improves the current solution by randomly perturbing it and accepting the perturbation with a certain probability.

The probability of accepting a worse solution is initially high. However, as the number of iterations increases, the probability of accepting a worse solution gradually decreases. Therefore, the accuracy of the solution depends on the number of iterations SA performs.

**The SA algorithm is quite simple, and we implement it straightforwardly, as described below.**

**3.1. Define the Problem**

First, we need to define the problem to optimize.**It involves defining the energy function, i.e., the function to minimize or maximize**. For example, if we want to minimize a real-valued function of two variables, e.g., , the energy corresponds to the function  itself. In the case of the TSP, the energy related to a sequence of cities is represented by the total length of the travel.

Once we define the energy function, we need to set the initial temperature value and the initial candidate solution. Moreover, we can set the initial temperature value and the initial candidate solution either randomly or using some heuristic method. Furthermore, the next step is to compute the energy of the initial candidate solution.

**3.2. Define the Perturbation Function**

**Next, we define a perturbation function to generate new candidate solutions.** Thus, this function should generate solutions that are close to the current solution but not too similar. For example, if we want to minimize a function , we can randomly perturb the current solution by adding a random value between -0.1 and 0.1 to both  and . In the case of the TSP, we can generate a new candidate solution by swapping two cities in the travelling order of the current solution.

**3.3. Acceptance Criterion**

The acceptance criterion determines whether a new solution is accepted or rejected. Moreover, the acceptance depends on the energy difference between the new solution and the current solution, as well as the current temperature.

**The classic acceptance criterion of SA comes from statistical mechanics, and it is based on the**[Boltzmann probability distribution](https://en.wikipedia.org/wiki/Boltzmann_distribution). A system in thermal equilibrium at temperature  can be found in a state with energy  with a probability proportional to

(1)

where  is the [Boltzmann constant](https://en.wikipedia.org/wiki/Boltzmann_constant). Hence, at low temperatures, there is a small chance that the system is in a high-energy state. It plays a crucial role in SA because an increase in energy allows escape from local minima and finds the global minimum.

**Based on the Boltzmann distribution, the following algorithm defines the criterion for accepting an energy variation  at temperature :**

algorithm **AcceptanceFunction**(T, ΔE):

// INPUT

T = the temperature

ΔE = the energy variation between the **new** **candidate** solution and the current one

// OUTPUT

Returns true **if** the **new** **solution** is accepted. Otherwise, returns false.

**if** ΔE < 0:

**return** true

**else**:

r <- generate a random value in the range [0, 1)

**if** r < exp(-ΔE / T):

**return** true

**else**:

**return** falseCopy

Thus, a candidate solution with lower energy is always accepted. Conversely, a candidate solution with higher energy is accepted randomly with probability  (for our purpose, we can set ). Moreover, we can implement the latter cases by comparing the probability with a random value generated in the range .

**3.4. Temperature Schedule**

The temperature schedule determines how the temperature of the system changes over time. In the beginning, the temperature is high so that the algorithm can explore a wide range of solutions, even if they are worse than the current solution. As the iterations increase, the temperature gradually decreases. Hence, the algorithm becomes more selective and accepts better solutions with higher probability.

Here, we can obtain a simple scheduling method by multiplying the current temperature by a factor , where :

Hence, it ensures the temperature decreases gradually over time as the value of  is always less than 1.

**3.5. Run the SA Algorithm**

Finally, we run the algorithm iteratively by applying the perturbation function and acceptance criterion to the current solution. Moreover, the algorithm terminates when the temperature has cooled to a certain level  or when the energy of the current solution is lower than a fixed threshold .

**Here’s the pseudocode of SA:**

algorithm **SimulatedAnnealingOptimizer**(T\_max, T\_min, E\_th, α):

// INPUT

T\_max = the maximum **temperature**

T\_min = the minimum temperature **for** stopping the **algorithm**

E\_th = the energy threshold to stop the **algorithm**

alpha = the cooling factor

// OUTPUT

The best found solution

T <- T\_max

x <- generate the initial candidate solution

E <- E(x) // compute the energy of the initial solution

**while** T > T\_min and E > E\_th:

x\_new <- generate a **new** **candidate** solution

E\_new <- compute the energy of the **new** **candidate** x\_new

ΔE <- E\_new - E

**if** **Accept**(ΔE, T):

x <- x\_new

E <- E\_new

T <- T \* alpha // cool the temperature

**return** xCopy

**4. SA Flowchart**

**Here, we provide a detailed flowchart representing all steps of SA:**

**5. Example**

To better understand the algorithm, we use SA to illustrate the minimization of the function . First, we used as search space a grid of size 101  101 placed in the square area defined by . Furthermore, we set the cooling rate  and the initial solution . Additionally, at each step, we generate a new solution by randomly shifting the current solution by  in  and  direction.

**Here’s an animation showing the candidate solution, its energy, and the temperature at each step:**

**Thus, we can observe that SA accepts the worse solutions when the temperature is high. Conversely, when the temperature is low (e.g., ), the algorithm is more selective, and better solutions are accepted with higher probability.**

**6. Conclusion**

In this article, we provided an overview of the SA algorithm. Furthermore, we illustrated the optimization procedure and provided a practical example of its application.