Implementing Simulated Annealing for a Logical OR Gate

Jesse Gempel
Introduction to Artificial Intelligence

February 13, 2026

1 Abstract

The intent of this report is to trace the Simulated Annealing algorithm to try to accurately represent a Boolean logical OR gate. For each iteration, a new configuration of three different weights (*Wx*, *Wy*, and *Wb*) needs to be generated by generating a random number for each weight, calculated by:

$$\text{config}_{\text{next}} = \text{config}_{\text{current}} + \text{RAND}(-2, 2)$$

Each configuration of those three weights is intended to act as a solution towards the OR gate's correct implementation. It is ideal for the algorithm to correctly recognize that one or two true values yield a true result, and that no true values yield a false result. To determine whether the weights should be accepted or rejected for an iteration, the value ΔE of the *change in energy* is analyzed. The configuration is accepted if:

$$\Delta\text{E} > 0,\ \textit{or}\ \Delta\text{E} \leq 0 \text{ and P} > \text{RAND(0, 1)}$$

If $\Delta E \leq 0$, then the value of P is determined by dividing ΔE by a temperature schedule value. The value is then compared with a random number between 0 and 1. If:

$$\text{P} \leq \text{RAND(0, 1)}$$

then a configuration must be rejected, and a new one must be obtained for an iteration. An iteration will not be complete until the algorithm generates NOR gate weights that satisfy one of the first two conditions above.

2 Background

The Simulated Annealing Algorithm focuses on the idea that finding a robust configuration or move is *not* always important. The algorithm begins by attempting to find a configuration without focusing on its quality. As it continues to execute, the algorithm becomes more selective with the configurations it chooses to accept. Lower-quality moves experience a higher rate of discarding due to its increasing focus on discovering an optimal solution.

Simulated Annealing acts as a modification of the Hill Climbing algorithm. Those two algorithms are centered on three different types of optima: *local maxima*, *local minima*, and *global maxima*. Hill Climbing *always* focuses on obtaining the best solution–that is why it struggles to discover configurations that lie among the *global maxima*. Fortunately, Simulated Annealing does not encounter this obstacle because it never remains stuck on global maxima. The dynamic nature of the algorithm's selectivity makes it a suitable alternative to Hill Climbing.

3 Data and Results

3.1 Temperature Schedules

To analyze the effectiveness of the Simulated Annealing algorithm, I utilized three different temperature schedules calculated for 10 iterations, as mentioned in Figure 1.

The graph illustrates an *inversely proportional* relationship between each temperature schedule and the number of iterations that finished. As the ratio in the equation's denominator increases, the temperature monotonically decreases at a faster rate.

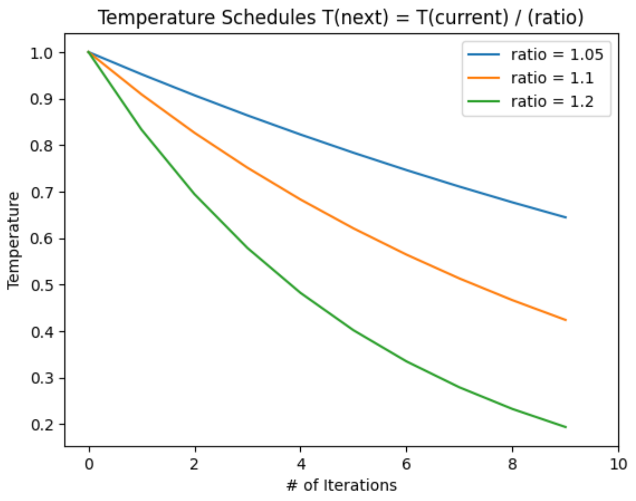

Figure 1: The three temperature schedules used for the Simulated Annealing Algorithm are calculated by the equation in the graph title. The ratios for the equation's denominator (1.05, 1.1, and 1.2) are compared to analyze the effectiveness of the algorithm.

3.2 Change in Energy

To analyze the performance of the Simulated Annealing algorithm, it is also important to observe the change in energy ΔE observed for the three temperature schedules, as shown in Figure 2. Each graph appears to behave quite differently. The temperature schedule with a ratio of 1.05 appears to fluctuate at a moderate pace. The same can be said about ΔE running a schedule with a ratio of 1.2, although the graph pattern fluctuates more frequently. In both cases, the numbers appeared to alternate between positive and negative numbers, which means that weight configurations have a higher chance of being rejected.

Running a temperature schedule with a ratio of 1.1, the algorithm appeared to generate highly consistent ΔE values until the final iteration. The values also appeared to maintain positive values for the first 9 iterations. Consistently positive ΔE values demonstrate that the NOR gate weight configurations are accepted more frequently with this temperature schedule.

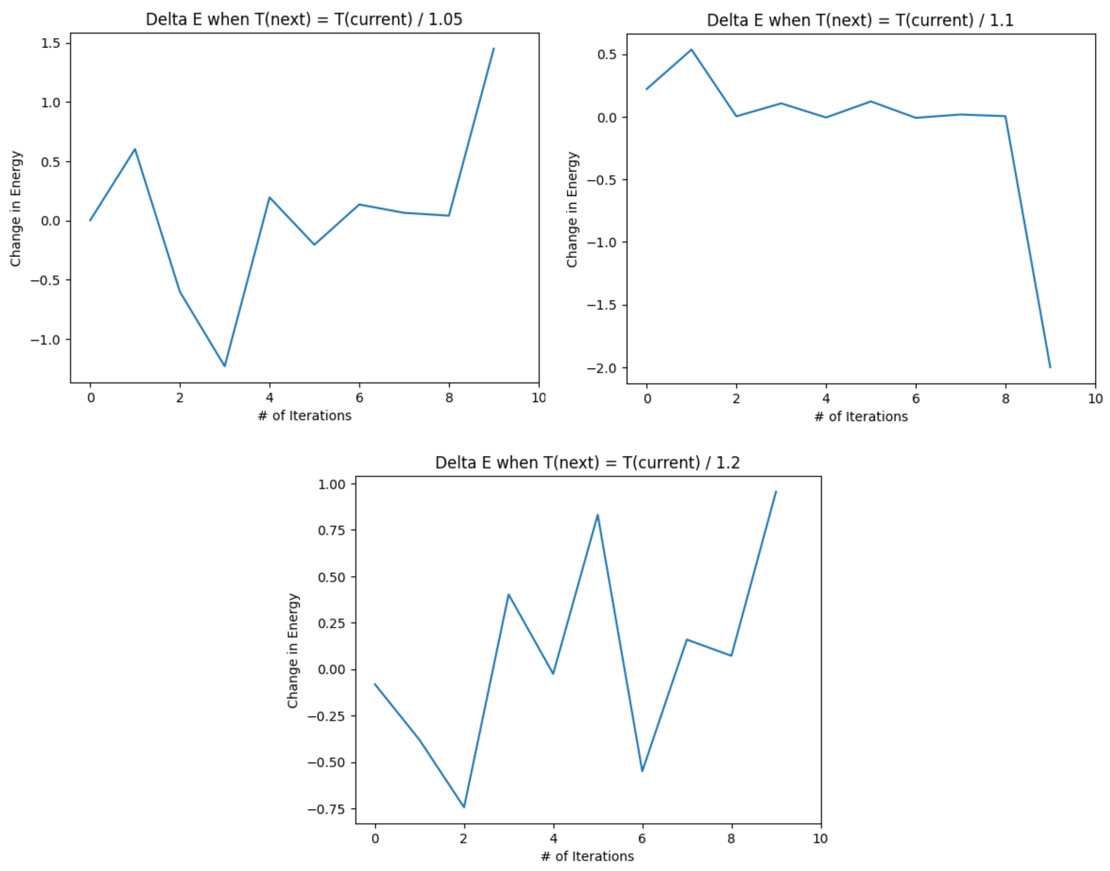

Figure 2: The change in energy ΔE is illustrated across the three examined temperature schedules.

3.3 The Objective Function

The objective value function is an error that is used to determine the effectiveness of the algorithm. The goal is for each iteration to generate an objective function value as close to 0 as possible. To determine the objective function value, the values of the four X/Y pairs (0,0), (0,1), (1,0), and (1,1) are first calculated by the weight configuration in this way:

$$value = x * Wx + y * Wy + Wb$$

where x and y are the x and y values in the OR gate truth table. In addition, Wx, Wy, and Wb serve as the weight configurations in an iteration. Next, a Sigmoid function:

$$error = \left| truthTableResult - \frac{1}{1 + e^{-value}} \right|$$

determines the error generated from each of the x-y pairings in the truth table. Finally, the 4 error values are summed to determine the error value of the objective function for an iteration like this:

$$value_{total} = error_{0,0} + error_{0,1} + error_{1,0} + error_{1,1}$$

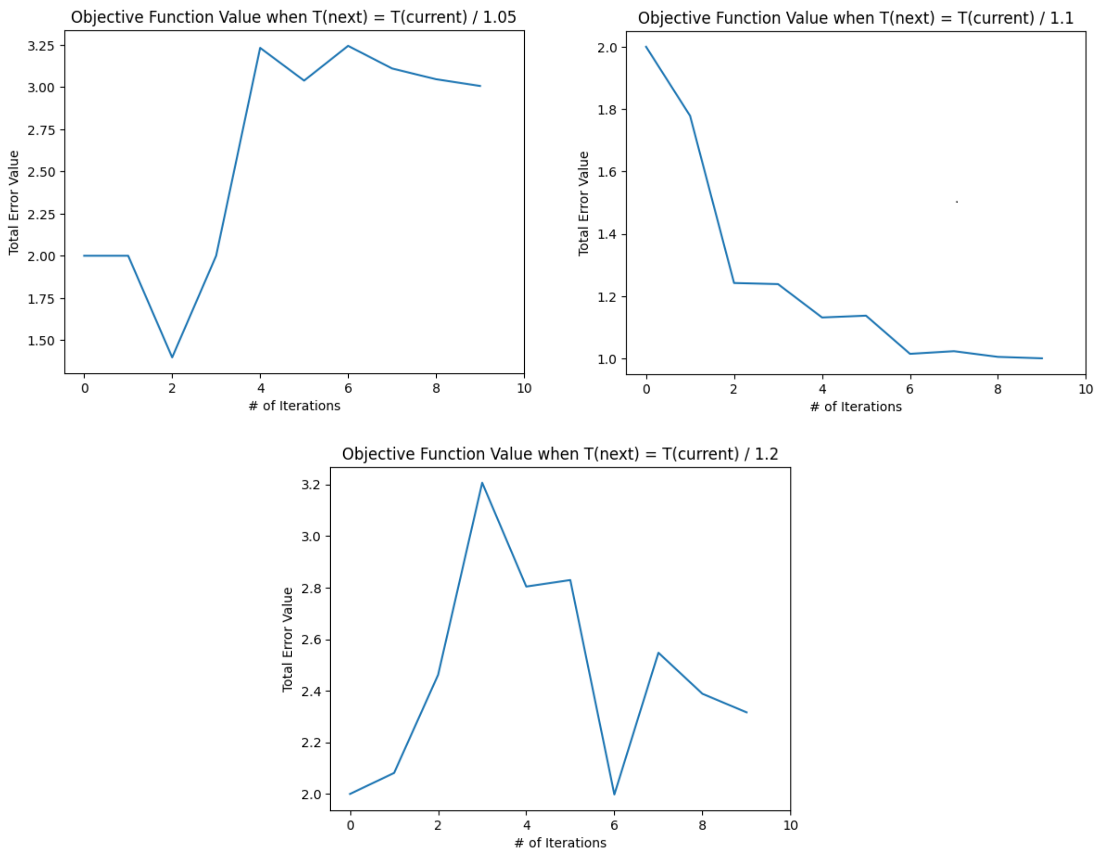

Figure 3: The objective function error is illustrated across the three examined temperature schedules.

When the Simulated Annealing algorithm begins, each x/y error value starts at 0.5. As the four error values are summed, the total objective function error value always initializes to 2.0. The ideal outcome over 10 iterations of the algorithm is the objective function value to approach 0 as much as possible.

The first algorithmic run began with a temperature schedule that is divided by 1.05 for every iteration. The total error value generally increased, then remained constant throughout the algorithmic run. Because these values never decreased monotonically, a schedule with a 1.05 ratio is undesirable.

When the algorithm performed 10 iterations with a temperature schedule of ratio 1.1, the total error value decreased monotonically. However, the objective function value never approaches below 1.0. This situation demonstrates that the Simulated Annealing algorithm performs well, then stops improving when it should.

The final algorithmic run dealt with a schedule of 1.2. The schedule setup caused the total error values to fluctuate, which means that the algorithm accepts poor configurations too frequently. The results are undesirable for this algorithmic run for that reason.

Discussion

The time required to learn about the mechanisms of Simulated Annealing served as a challenge. Before creating a Python program on Google Colab, I needed to practice and work with a complex Google Spreadsheet. The spreadsheet in question closely simulated the algorithm, and thus the Python program that I eventually created. Once the challenge was overcome, I created a Python program on Google Colab and executed the Simulated Annealing algorithm through the code.

When analyzing the three temperature schedules for this algorithm, the schedule that divided each temperature by 1.1 allowed Simulated Annealing to perform the most proficiently. Although the values did monotonically decrease, the total objective function values never decreased to less than 1.0, The issue of total error values not approaching 0 acts as another challenge. The Python code performed the algorithm accurately and correctly. However, the change in energy ΔE would generate extremely low values close to 0, so the configurations were rejected more often.

The frequent rejections of new configurations presented two additional challenges: employing a sufficient temperature schedule and running the algorithm over a certain number of iterations. Using a temperature schedule that divides the temperature by an overly high value would eventually cause every configuration to be rejected. An iteration does not finish until a robust NOR gate weight configuration is generated. However, constant configuration rejections sometimes caused an infinite loop in the Python program. This issue occurred most frequently when running the program over a large number of iterations.