CHAPTER 7: UNDERSTANDING SIMULATED ANNEALING

- What is Simulated Annealing?
- For What is Simulated Annealing Used?
- Implementing Simulated Annealing in Java
- Applying Simulated Annealing to the Traveling Salesman Problem

In chapter 6, "Understanding Genetic Algorithms," you were introduced to genetic algorithms and how they can be used to train a neural network. In this chapter you will learn about another popular algorithm you can use, simulated annealing. As you will see, it can also be applied to other situations.

The sample program that will be presented in this chapter solves the traveling salesman problem, as did the genetic algorithm in chapter 6. However, in this program, simulated annealing will be used in place of the genetic algorithm. This will allow you to see some of the advantages that simulated annealing offers over a genetic algorithm.

We will begin with a general background of the simulated annealing process. We will then construct a class that is capable of using simulated annealing to solve the traveling salesman problem. Finally, we will explore how simulated annealing can be used to train a neural network.

Simulated Annealing Background

Simulated annealing was developed in the mid 1970s by Scott Kirkpatrick and several other researchers. It was originally developed to better optimize the design of integrated circuit (IC) chips by simulating the actual process of annealing.

Annealing is the metallurgical process of heating up a solid and then cooling it slowly until it crystallizes. The atoms of such materials have high-energy values at very high temperatures. This gives the atoms a great deal of freedom in their ability to restructure themselves. As the temperature is reduced, the energy levels of the atoms decrease. If the cooling process is carried out too quickly, many irregularities and defects will be seen in the crystal structure. The process of cooling too rapidly is known as rapid quenching. Ideally, the temperature should be reduced slowly to allow a more consistent and stable crystal structure to form, which will increase the metal's durability.

Simulated annealing seeks to emulate this process. It begins at a very high temperature, at which the input values are allowed to assume a wide range of random values. As the training progresses, the temperature is allowed to fall, thus restricting the degree to which the inputs are allowed to vary. This often leads the simulated annealing algorithm to a better solution, just as a metal achieves a better crystal structure through the actual annealing process.

Simulated Annealing Applications

Given a specified number of inputs for an arbitrary equation, simulated annealing can be used to determine those inputs that will produce the minimum result for the equation. In the case of the traveling salesman, this equation is the calculation of the total distance the salesman must travel. As we will learn later in this chapter, this equation is the error function of a neural network.

When simulated annealing was first introduced, the algorithm was very popular for integrated circuit (IC) chip design. Most IC chips are composed of many internal logic gates. These gates allow the chip to accomplish the tasks that it was designed to perform. Just as algebraic equations can often be simplified, so too can IC chip layouts. Simulated annealing is often used to find an IC chip design that has fewer logic gates than the original. The result is a chip that generates less heat and runs faster.

The weight matrix of a neural network provides an excellent set of inputs for the simulated annealing algorithm to minimize. Different sets of weights are used for the neural network until one is found that produces a sufficiently low return from the error function.

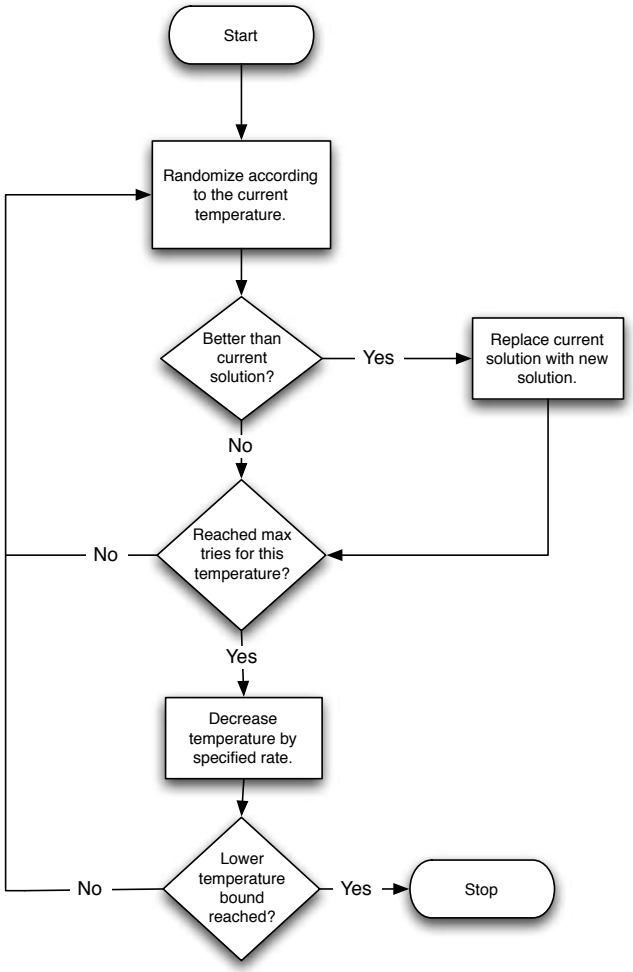
Understanding Simulated Annealing

The previous sections discussed the background of the simulated annealing algorithm and presented various applications for which it is used. In this section, you will be shown how to implement the simulated annealing algorithm. We will first examine the algorithm and then we will develop a simulated annealing algorithm class that can be used to solve the traveling salesman problem, which was introduced in chapter 6.

The Structure of a Simulated Annealing Algorithm

There are several distinct steps that the simulated annealing process must go through as the temperature is reduced and randomness is applied to the input values. Figure 7.1 presents a flowchart of this process.

Figure 7.1: Overview of the simulated annealing process.



As you can see in Figure 7.1, there are two major processes that take place in the simulated annealing algorithm. First, for each temperature, the simulated annealing algorithm runs through a number of cycles. The number of cycles is predetermined by the programmer. As a cycle runs, the inputs are randomized. In the case of the traveling salesman problem, these inputs are the order of the cities that the traveling salesman will visit. Only randomizations which produce a better-suited set of inputs will be retained.

Once the specified number of training cycles have been completed, the temperature can be lowered. Once the temperature is lowered, it is determined whether or not the temperature has reached the lowest temperature allowed. If the temperature is not lower than the lowest temperature allowed, then the temperature is lowered and another cycle of randomizations will take place. If the temperature is lower than the lowest temperature allowed, the simulated annealing algorithm terminates.

At the core of the simulated annealing algorithm is the randomization of the input values. This randomization is ultimately what causes simulated annealing to alter the input values that the algorithm is seeking to minimize. The randomization process must often be customized for different problems. In this chapter we will discuss randomization methods that can be used for both the traveling salesman problem and neural network training. In the next section, we will examine how this randomization occurs.

How Are the Inputs Randomized?

An important part of the simulated annealing process is how the inputs are randomized. The randomization process takes the previous input values and the current temperature as inputs. The input values are then randomized according to the temperature. A higher temperature will result in more randomization; a lower temperature will result in less randomization.

There is no specific method defined by the simulated annealing algorithm for how to randomize the inputs. The exact nature by which this is done often depends upon the nature of the problem being solved. When comparing the methods used in the simulated annealing examples for the neural network weight optimization and the traveling salesman problem, we can see some of the differences.

Simulated Annealing and Neural Networks

The method used to randomize the weights of a neural network is somewhat simpler than the traveling salesman's simulated annealing algorithm, which we will discuss next. A neural network's weight matrix can be thought of as a linear array of floating point numbers. Each weight is independent of the others. It does not matter if two weights contain the same value. The only major constraint is that there are ranges that all weights must fall within.

Thus, the process generally used to randomize the weight matrix of a neural network is relatively simple. Using the temperature, a random ratio is applied to all of the weights in the matrix. This ratio is calculated using the temperature and a random number. The higher the temperature, the more likely it is that the ratio will cause a larger change in the weight matrix. A lower temperature will most likely produce a smaller change. This is the method that is used for the simulated annealing algorithm that will be presented later in this chapter.

Simulated Annealing and the Traveling Salesman Problem

The method used to randomize the path of the traveling salesman is somewhat more complex than the method used to randomize the weights of a neural network. This is because there are constraints that exist on the path of the traveling salesman problem that do not exist when optimizing the weight matrix of the neural network. The most significant constraint is that the randomization of the path must be controlled enough to prevent the traveling salesman from visiting the same city more than once, and at the same time ensure that he does visit each city once; no cities may be skipped.

You can think of the traveling salesman randomization as the reordering of elements in a fixed-size list. This fixed-size list is the path that the traveling salesman must follow. Since the traveling salesman can neither skip nor revisit cities, his path will always have the same number of "stops" as there are cities.

As a result of the constraints imposed by the traveling salesman problem, most randomization methods used for this problem change the order of the previous path through the cities. By simply rearranging the data, and not modifying original values, we can be assured that the final result of this reorganization will neither skip, nor revisit cities.

This is the method that is used to randomize the traveling salesman's path in the example in this chapter. Using a combination of the temperature and distance between two cities, the simulated annealing algorithm determines if the positions of the two cities should be changed. You will see the actual Java implementation of this method later in this chapter.

Temperature Reduction

There are several different methods that can be used for temperature reduction; we will examine two. The most common is to simply reduce the temperature by a fixed amount through each cycle. This is the method that is used in this chapter for the traveling salesman problem.

Another method is to specify a beginning and ending temperature. To use this method, we must calculate a ratio at each step in the simulated annealing process. This is done by using an equation that guarantees that the step amount will cause the temperature to fall to the ending temperature in the number of cycles specified. Equation 7.1 describes how to logarithmically decrease the temperature between a beginning and ending temperature. It calculates the ratio and ensures that the temperature naturally decreases for each cycle.

Equation 7.1: Scaling the Temperature

$$step = e^{\frac{\ln(\frac{s}{e})}{c-1}}$$

The variables are **s** for starting temperature, **e** for ending temperature, and **c** for cycle count. The equation can be implemented in Java as follows:

```
    double ratio = Math.exp(Math.log(stopTemperature/
startTemperature)/(cycles-1));
```

The above line calculates a ratio that should be multiplied against the current temperature. This will produce a change that will cause the temperature to reach the ending temperature in the specified number of cycles. This method is used later in this chapter when simulated annealing is applied to neural network training.

Implementing Simulated Annealing

The source code accompanying this book provides a generic simulated annealing class. This abstract class is named **SimulatedAnnealing** and can be used to implement a simulated annealing solution for a variety of problems. We will use this simulated annealing class for both the neural network example and the traveling salesman problem.

This section will describe how the generic **SimulatedAnnealing** class works. The application of simulated annealing to neural networks and the traveling salesman problem will be covered later in this chapter.

Inputs to the Simulated Annealing Algorithm

There are several variables that must be set on the **SimulatedAnnealing** class for it to function properly. These variables are usually set by the constructor of one of the classes that subclass the **SimulatedAnnealing** class. Table 7.1 summarizes these inputs.

Table 7.1: Simulated Annealing Inputs

Variable	Purpose
startTemperature	The temperature at which to start.
stopTemperature	The temperature at which to stop.
cycles	The number of cycles to be used.

The simulated annealing training algorithm works very much like every other training algorithm in this book. Once it is set up, it progresses through a series of iterations.

Processing Iterations

The **SimulatedAnnealing** class contains a method named **iteration** that is called to process each iteration of the training process.

```
public void iteration() throws NeuralNetworkError {
```

First, an array is created to hold the best solution.

```
UNIT_TYPE bestArray[];
```

Next, the starting error is determined.

```
setError(determineError());
bestArray = this.getArrayCopy();
```

The training process is then cycled through a specified number of times. For each training pass, the **randomize** method is called. This method is abstract and must be implemented for any problem that is to be solved by simulated annealing.

```
for (int i = 0; i < this.cycles; i++) {
  double curError;
  randomize();
```

The error is determined after **randomize** has been called.

```
  curError = determineError();
```

If this was an improvement, then the newly created array is saved.

```
  if (curError < getError()) {
    bestArray = this.getArrayCopy();
    setError(curError);
  }
}
```

Once the cycle is complete, the best array is stored.

```
this.putArray(bestArray);
```

A ratio is calculated that will decrease the temperature to the desired level. This is the Java implementation of Equation 7.1, which was shown earlier.

```
final double ratio = Math.exp(Math.log(getStopTemperature()
  / getStartTemperature())
  / (getCycles() - 1));
```

The temperature is scaled by this amount.

```
this.temperature *= ratio;
```

This simulated annealing class is, of course, abstract; thus it only implements the simulated annealing algorithm at a primitive level. The examples in this chapter that actually put the **SimulatedAnnealing** class to use must implement the **randomize** function for their unique situations.

Simulated Annealing for the Traveling Salesman Problem

Simulated annealing can provide potential solutions to the traveling salesman problem. The traveling salesman problem was introduced in chapter 6, "Understanding Genetic Algorithms." Aside from the fact that the traveling salesman problem from this chapter uses simulated annealing, it is the same as the program presented in chapter 6.

The simulated annealing traveling salesman problem implements a special version of the **SimulatedAnnealing** class. The class is named **TSPSimulatedAnnealing**. The most important method is the **randomize** method. The signature for the **randomize** method is shown here:

```
public void randomize()
```

First, the **length** of the path is determined.

```
final int length = this.path.length;
```

Next, we iterate through the loop a number of times equal to the temperature. The higher the temperature, the more iterations. The more iterations, the more "excited" the underlying path becomes and the more changes made.

```
for (int i = 0; i < this.temperature; i++) {
```

Two random index locations are chosen inside the path.

```
  int index1 = (int) Math.floor(length * Math.random());
  int index2 = (int) Math.floor(length * Math.random());
```

A basic distance number is calculated based on the two random points.

```
  final double d = distance(index1, index1 + 1)
  + distance(index2, index2 + 1)
  - distance(index1, index2)
  - distance(index1 + 1, index2 + 1);
```

If the distance calculation is greater than zero, then the array elements in the path are excited.

```
  if (d > 0) {
```

The index locations, **index1** and **index2**, are sorted if necessary.

```
if (index2 < index1) {
```

```
  final int temp = index1;
  index1 = index2;
  index2 = temp;
}
```

All cities between the two random index points are sorted.

```
for (; index2 > index1; index2--) {
  final int temp = this.path[index1 + 1];
  this.path[index1 + 1] = this.path[index2];
  this.path[index2] = temp;
  index1++;
}
```

When the new path is returned to the **`iteration`** function, it will be evaluated. If it is an improvement over the last path, it will be retained.

Simulated Annealing for Neural Networks

Simulated annealing can also be applied to neural networks. This book provides a class named **`NeuralSimulatedAnnealing`**. By making use of the generic **`SimulatedAnnealing`** class introduced earlier in this chapter, this class implements a training solution for neural networks. Listing 7.1 shows a simple program that uses simulated annealing to train a neural network for the XOR operator.

Listing 7.1: Simulated Annealing and the XOR Operator (AnnealXOR.java)

```
package com.heatonresearch.book.introneuralnet.ch7.xor;

import com.heatonresearch.book.introneuralnet.neural.feedforward.
FeedforwardLayer;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
FeedforwardNetwork;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
train.anneal.NeuralSimulatedAnnealing;

/**
 * Chapter 7: Training using Simulated Annealing
 *
 * XOR: Learn the XOR pattern with a feedforward neural
 * network that uses simulated annealing.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class AnnealXOR {
      public static double XOR_INPUT[][] =
            { { 0.0, 0.0 }, { 1.0, 0.0 },
```

```
                { 0.0, 1.0 }, { 1.0, 1.0 } };

      public static double XOR_IDEAL[][] =
            { { 0.0 }, { 1.0 }, { 1.0 }, { 0.0 } };

      public static void main(final String args[]) {
            final FeedforwardNetwork network =
                  new FeedforwardNetwork();
            network.addLayer(new FeedforwardLayer(2));
            network.addLayer(new FeedforwardLayer(3));
            network.addLayer(new FeedforwardLayer(1));
            network.reset();

            // train the neural network
            final NeuralSimulatedAnnealing train =
                  new NeuralSimulatedAnnealing(
                  network, XOR_INPUT, XOR_IDEAL, 10, 2, 100);

            int epoch = 1;

            do {
                  train.iteration();
                  System.out
                              .println("Epoch #" + epoch
                        + " Error:" + train.getError());
                  epoch++;
            } while ((epoch < 100) && (train.getError() > 0.001));

            // network = train.getNetwork();

            // test the neural network
            System.out.println("Neural Network Results:");
            for (int i = 0; i < XOR_IDEAL.length; i++) {
                  final double actual[] =
                  network.computeOutputs(XOR_INPUT[i]);
                  System.out.println(XOR_INPUT[i][0] + ","
                        + XOR_INPUT[i][1]
                        + ", actual=" + actual[0]
                        + ",ideal=" + XOR_IDEAL[i][0]);
            }
      }
}
```

The **NeuralSimulatedAnnealing** class implements the **Train** interface, and thus can be used just like backpropagation and genetic algorithms discussed in earlier chapters. The **NeuralSimulatedAnnealing** is instantiated as follows:

```
final NeuralSimulatedAnnealing train =
      new NeuralSimulatedAnnealing(
      network, XOR_INPUT, XOR_IDEAL, 10, 2, 100);
```

The **NeuralSimulatedAnnealing** class implements a special **randomize** method. This method excites the state of the neural network in a way that is very similar to how the traveling salesman implementation works. The signature for the **randomize** method is shown here:

```
public void randomize()
```

First, **MatrixCODEC** is used to serialize the neural network into an array of **Double** variables.

```
final Double array[] = MatrixCODEC.networkToArray(this.network);
```

We then loop through the array.

```
for (int i = 0; i < array.length; i++) {
```

Each array element is randomly excited based on the temperature.

```
  double add = 0.5 - (Math.random());
  add /= getStartTemperature();
  add *= this.temperature;
  array[i] = array[i] + add;
}
```

Finally, **MatrixCODEC** is used to turn the array back into a neural network.

```
MatrixCODEC.arrayToNetwork(array, this.network);
```

When this new neural network is returned to the **iteration** function, it will be evaluated. If it is an improvement, it will be retained.

Chapter Summary

In this chapter, you learned about the simulated annealing algorithm. The simulated annealing algorithm is based on the actual process of annealing. The key point behind the annealing process is that a metal that is allowed to cool slowly will form more consistent, and therefore stronger, crystal structures. The reason being that the higher temperatures result in higher energy levels for the atoms that make up the metal. At the higher energy levels, the atoms have greater freedom of movement. As the metal cools, this freedom of movement is curtailed. This allows the atoms to settle into consistent crystal patterns.

The process of simulated annealing is very similar to the actual annealing process. A series of input values are presented to the simulated annealing algorithm. The simulated annealing algorithm wants to optimize these input values so that an arbitrary equation can be minimized. Examples of equations to be minimized include the error

function for a neural network, or the distance that a traveling salesman travels. The input values, which drive the simulated annealing algorithm, can be the weight matrix of a neural network or the current route between cities that a traveling salesman is traveling.

To present a relatively simple example of how to use simulated annealing, this chapter once again turned to the traveling salesman problem. The traveling salesman problem was also used in chapter 6 in conjunction with genetic algorithms. Reusing the traveling salesman problem allows us to easily compare the performance of genetic algorithms with simulated annealing.

Vocabulary

Annealing

Annealing Cycles

Excite

Simulated Annealing

Temperature

Questions for Review

1. Describe the metallurgical annealing process.

2. How are the array elements randomized when finding solutions for the traveling salesman problem?

3. How are the array elements randomized when training a neural network?

4. What is the difference between a cycle and an iteration in the simulated annealing algorithm presented in this chapter?

5. How do we ensure that the temperature changes naturally between cycles?