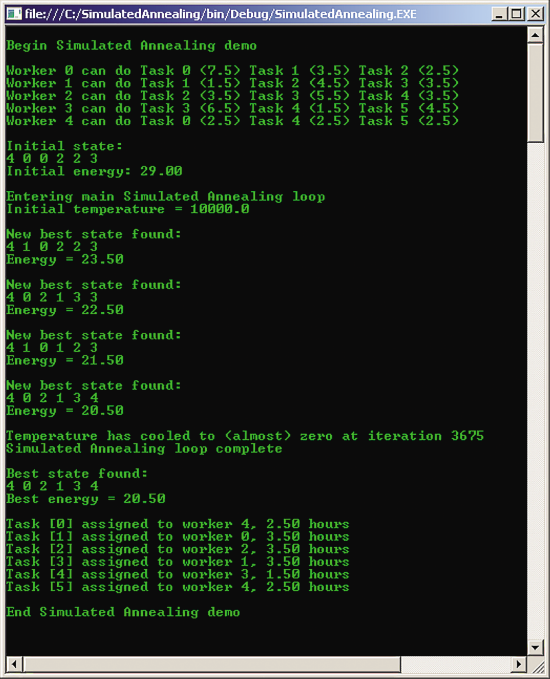
In this month’s column I present C# code that implements a Simulated Annealing (SA) algorithm to solve a scheduling problem. An SA algorithm is an artificial intelligence technique based on the behavior of cooling metal. It can be used to find solutions to difficult or impossible combinatorial optimization problems.

The best way to see where I’m headed is to take a look at **Figure 1** and **Figure 2**. The table in **Figure 1** describes a scheduling problem: five workers and six tasks. Each entry in the table represents how long it takes for a worker to complete a task. For example, worker w2 needs 5.5 hours to complete task t3. An empty entry in a row indicates that the corresponding worker can’t perform a task. The problem is to assign each of the six tasks to one of the workers in such a way that the total time to complete all tasks is minimized. Additionally, we assume that every time a worker switches to a new task, there’s a 3.5-hour retooling penalty.

**Figure 1 Time for Worker to Complete Task**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | t0 | t1 | t2 | t3 | t4 | t5 |
| w0 | 7.5 | 3.5 | 2.5 |  |  |  |
| w1 |  | 1.5 | 4.5 | 3.5 |  |  |
| w2 |  |  | 3.5 | 5.5 | 3.5 |  |
| w3 |  |  |  | 6.5 | 1.5 | 4.5 |
| w4 | 2.5 |  |  |  | 2.5 | 2.5 |

  
**Figure 2 SimulatedAnnealing in Action**

**Figure 2** shows a program that uses an SA algorithm to find a solution to the scheduling problem. The program begins by generating a random solution. In SA terminology, a potential solution is called the state of the system. The initial state is [4 0 0 2 2 3], which means task 0 has been assigned to worker 4, task 1 has been assigned to worker 0, task 2 has been assigned to worker 0, task 3 has been assigned to worker 2, task 4 has been assigned to worker 2 and task 5 has been assigned to worker 3. The total time for the initial random state is 2.5 + 3.5 + 2.5 + 5.5 + 3.5 + 4.5 = 22 hours plus a 3.5-hour retooling penalty for worker 0 and a 3.5-hour penalty for worker 2, for a total of 29 hours. In SA terminology, the quantity you’re trying to minimize (or less frequently maximize) is called the energy of the state.

The program enters a loop. In each iteration of the loop, the SA algorithm generates an adjacent state and evaluates that adjacent state to see if it’s better than the current state. Behind the scenes, the SA algorithm uses a temperature variable that slowly decreases, as I’ll explain shortly. The SA loop ends when the temperature cools sufficiently close to zero. The program concludes by displaying the best solution found.

This example is artificially simple because with six tasks where each task can be performed by one of three workers, there are only 36 possible combinations, which equals 729, so you could just evaluate every one. But suppose you had a problem with 20 tasks where each task can be performed by one of 12 workers. There would be 1220 combinations, which equals a whopping 3,833,759,992,447,475,122,176. Even if you could evaluate 1 million possible solutions per second, you’d need about 121 million years to evaluate every possibility.

SA is a metaheuristic—that is, a general conceptual framework that can be used to create a specific algorithm to solve a specific problem. It’s best used for combinatorial optimization problems where there’s no practical deterministic solution algorithm. First described in a 1983 paper by S. Kirkpatrick, C. Gelatt and M. Vecchi, SA is loosely based on the annealing behavior of cooling metal. When some metals are heated to a high temperature, the atoms and molecules break their bonds. When the metal is slowly cooled, the atoms and molecules reform their bonds in such a way that the energy of the system is minimized.

This column assumes you have intermediate-level programming skills. I implement the SA program using C#, but you shouldn’t have too much trouble refactoring my code to a different language such as Visual Basic or Python. In the sections that follow, I’ll walk you through the program that generated **Figure 2**. All of the code is available as a download from [archive.msdn.microsoft.com/mag201201TestRun](http://archive.msdn.microsoft.com/mag201201TestRun). I think you’ll find the ability to understand and use SA an interesting and possibly useful addition to your personal skill set.

## Program Structure

I implemented the SA demo program as a single C# console application. The overall structure of the program is shown in **Figure 3**.

**Figure 3 SimulatedAnnealing Program Structure**

1. **using** System;
2. **namespace** SimulatedAnnealing
3. {
4. **class** SimulatedAnnealingProgram
5. {
6. **static** Random random;
7. **static** **void** Main(**string**[] args)
8. {
9. **try**
10. {
11. // Set up problem data.
12. // Create random state.
13. // Set up SA variables for temperature and cooling rate.
14. **while** (iteration < maxIteration && currTemp > 0.0001)
15. {
16. // Create adjacent state.
17. // Compute energy of adjacent state.
18. // Check if adjacent state is new best.
19. // If adjacent state better, accept state with varying probability.
20. // Decrease temperature and increase iteration counter.
21. }
22. // Display best state found.
23. }
24. **catch** (Exception ex)
25. {
26. Console.WriteLine(ex.Message);
27. Console.ReadLine();
28. }
29. } // Main
30. **static** **double**[][] MakeProblemData(**int** numWorkers, **int** numTasks) { ... }
31. **static** **int**[] RandomState(**double**[][] problemData) { ... }
32. **static** **int**[] AdjacentState(**int**[] currState,
33. **double**[][] problemData) { ... }
34. **static** **double** Energy(**int**[] state, **double**[][] problemData) { ... }
35. **static** **double** AcceptanceProb(**double** energy, **double** adjEnergy,
36. **double** currTemp) { ... }
37. **static** **void** Display(**double**[][] matrix) { ... }
38. **static** **void** Display(**int**[] vector) { ... }
39. **static** **void** Interpret(**int**[] state, **double**[][] problemData) { ... }
40. } // Program
41. } // ns

I used Visual Studio to create a console application program named SimulatedAnnealing. In the Solution Explorer window, I renamed the default Program.cs file to SimulatedAnnealingProgram.cs, which automatically renamed the single class in the project. I deleted all the template-generated using statements except for the System namespace—SA is quite simple and typically doesn’t need much library support. I declared a class-scope Random object named random. SA algorithms are probabilistic, as you’ll see shortly.

The heart of the SA algorithm processing is a while loop. The loop is controlled by a loop counter variable named “iteration” and by a variable that represents the temperature of the system. In practice, the temperature variable almost always reaches near-zero and terminates the loop before the loop counter reaches its maximum value and terminates the loop.

SA algorithms must have three problem-specific methods as suggested in **Figure3**. The SA algorithm must have a method that generates an initial (usually random) state/solution. The SA algorithm must have a method that generates an adjacent state relative to a given state. And the SA algorithm must have a method that computes the energy/cost of a state—the value you’re trying to minimize or maximize. In**Figure 3** these are methods RandomState, Adjacent­State and Energy, respectively. Method AcceptanceProb generates a value used to determine if the current state of the system transitions to the adjacent state even when the adjacent state is worse than the current state. Method MakeProblemData is a helper and creates a data structure matrix that corresponds with **Figure 1**. The overloaded Display methods and the Interpret method are just helpers to display information.

## Program Initialization

The Main method begins like so:

1. **try**
2. {
3. Console.WriteLine("\nBegin Simulated Annealing demo\n");
4. Console.WriteLine("Worker 0 can do Task 0 (7.5) Task 1 (3.5) Task 2 (2.5)");
5. Console.WriteLine("Worker 1 can do Task 1 (1.5) Task 2 (4.5) Task 3 (3.5)");
6. Console.WriteLine("Worker 2 can do Task 2 (3.5) Task 3 (5.5) Task 4 (3.5)");
7. Console.WriteLine("Worker 3 can do Task 3 (6.5) Task 4 (1.5) Task 5 (4.5)");
8. Console.WriteLine("Worker 4 can do Task 0 (2.5) Task 4 (2.5) Task 5 (2.5)");
9. ...

I wrap all SA code in a single, high-level try-catch block and display the dummy problem that I intend to set up. Here, I’m using an artificially simple example—but one that’s representative of the kinds of combinatorial problems that are suited for a solution by SA algorithms. Next comes:

1. random = **new** Random(0);
2. **int** numWorkers = 5;
3. **int** numTasks = 6;
4. **double**[][] problemData = MakeProblemData(numWorkers, numTasks);

I initialize the Random object using an arbitrary seed value of 0. Then I call helper method MakeProblemData to construct the data structure shown in **Figure 1**. I’ll present MakeProblemData and the other helper methods after I finish presenting all the code in the Main method. Next comes:

1. **int**[] state = RandomState(problemData);
2. **double** energy = Energy(state, problemData);
3. **int**[] bestState = state;
4. **double** bestEnergy = energy;
5. **int**[] adjState;
6. **double** adjEnergy;

I call helper method RandomState to generate a random state/­solution for the problem. State is represented as an int array where the array index represents a task and the value in the array at the index represents the worker assigned to the task. Helper method Energy computes the total time required by its state parameter, taking into account the 3.5-hour penalty for retooling every time a worker does an additional task. I’ll track the best state found and its corresponding energy, so I declare variables bestState and bestEngery. Variables adjState and adjEnergy are used to hold a state that’s adjacent to the current state, and the corresponding energy. Next comes:

1. **int** iteration = 0;
2. **int** maxIteration = 1000000;
3. **double** currTemp = 10000.0;
4. **double** alpha = 0.995;

The primary SA processing loop terminates on one of two conditions: when a counter exceeds a maximum value or when the temperature variable decreases to a value close to zero. I name the loop counter “iteration,” but I could’ve called it “counter” or “time” or “tick” or something similar. I name the temperature variable currTemp rather than temp so there’s less chance someone reviewing the code might interpret it as a temporary variable. Variable alpha represents the cooling rate, or a factor that determines how the temperature variable decreases, or cools, each time through the processing loop. Next comes:

1. Console.WriteLine("\nInitial state:");
2. Display(state);
3. Console.WriteLine("Initial energy: " + energy.ToString("F2"));
4. Console.WriteLine("\nEntering main Simulated Annealing loop");
5. Console.WriteLine("Initial temperature = " + currTemp.ToString("F1") + "\n");

Before entering the main processing loop, I display some informational messages about the initial state, energy and temperature. You might want to display additional information such as the cooling rate. Here’s the loop:

1. **while** (iteration < maxIteration && currTemp > 0.0001)
2. {
3. adjState = AdjacentState(state, problemData);
4. adjEnergy = Energy(adjState, problemData);
5. **if** (adjEnergy < bestEnergy)
6. {
7. bestState = adjState;
8. bestEnergy = adjEnergy;
9. Console.WriteLine("New best state found:");
10. Display(bestState);
11. Console.WriteLine("Energy = " + bestEnergy.ToString("F2") + "\n");
12. }
13. ...

Notice the loop control exits when the temperature variable drops below 0.0001 rather than when it hits 0.0. You might want to parameterize that minimum temperature value. After creating an adjacent state and computing its energy, I check to see if that adjacent state is a new global best solution, and if so I save that information. I can copy the best state by reference because method AdjacentState allocates a new array, but I could’ve made an explicit copy. Whenever a new global best state is found, I display it and its energy. The main processing loop ends like this:

1. **double** p = random.NextDouble();
2. **if** (AcceptanceProb(energy, adjEnergy, currTemp) > p)
3. {
4. state = adjState;
5. energy = adjEnergy;
6. }
7. currTemp = currTemp \* alpha;
8. ++iteration;
9. } // While

The loop finishes up by first generating a random value p greater than or equal to 0.0 and strictly less than 1.0 and comparing that value against the return from helper method AcceptanceProb. If the acceptance probability exceeds the random value, the current state transitions to the adjacent state. Next, the current temperature is decreased slightly by multiplying by the cooling factor, and the loop counter variable is incremented. Next comes:

1. Console.Write("Temperature has cooled to (almost) zero ");
2. Console.WriteLine("at iteration " + iteration);
3. Console.WriteLine("Simulated Annealing loop complete");
4. Console.WriteLine("\nBest state found:");
5. Display(bestState);
6. Console.WriteLine("Best energy = " + bestEnergy.ToString("F2") + "\n");
7. Interpret(bestState, problemData);
8. Console.WriteLine("\nEnd Simulated Annealing demo\n");
9. Console.ReadLine();

After the main SA processing loop completes, I display the best state (solution) found and its corresponding energy (hours). The Main method ends like this:

1. ...
2. } // Try
3. **catch** (Exception ex)
4. {
5. Console.WriteLine(ex.Message);
6. Console.ReadLine();
7. }
8. } // Main

The method wraps up by handling any exceptions simply by displaying the exception’s message.

## The Helper Methods

The code for helper method MakeProblemData is:

1. **static** **double**[][] MakeProblemData(**int** numWorkers, **int** numTasks)
2. {
3. **double**[][] result = **new** **double**[numWorkers][];
4. **for** (**int** w = 0; w < result.Length; ++w)
5. result[w] = **new** **double**[numTasks];
6. result[0][0] = 7.5; result[0][1] = 3.5; result[0][2] = 2.5;
7. result[1][1] = 1.5; result[1][2] = 4.5; result[1][3] = 3.5;
8. result[2][2] = 3.5; result[2][3] = 5.5; result[2][4] = 3.5;
9. result[3][3] = 6.5; result[3][4] = 1.5; result[3][5] = 4.5;
10. result[4][0] = 2.5; result[4][4] = 2.5; result[4][5] = 2.5;
11. **return** result;
12. }

I decided to use type double[][]—that is, an array of arrays—to hold my scheduling problem definition. The C# language, unlike many C-family languages, does support a built-in two-dimensional array, so I could’ve used type double[,] but an array of arrays is easier to refactor if you want to recode my example to a language that doesn’t support two-dimensional arrays. In this example I arbitrarily put the worker index first and the task index second (so result[1][3] is the time required by worker 1 to perform task 3), but I could’ve reversed the order. Notice that C# automatically initializes type double array cells to 0.0, so I don’t have to explicitly do so. I could’ve used some other value, such as -1.0 to indicate that a worker can’t perform a particular task.

Helper method RandomState is:

1. **static** **int**[] RandomState(**double**[][] problemData)
2. {
3. **int** numWorkers = problemData.Length;
4. **int** numTasks = problemData[0].Length;
5. **int**[] state = **new** **int**[numTasks];
6. **for** (**int** t = 0; t < numTasks; ++t) {
7. **int** w = random.Next(0, numWorkers);
8. **while** (problemData[w][t] == 0.0) {
9. ++w; **if** (w > numWorkers - 1) w = 0;
10. }
11. state[t] = w;
12. }
13. **return** state;
14. }

Recall that a state represents a solution and that a state is an int array where the index is the task and the value is the worker. For each cell in state, I generate a random worker w. But that worker might not be able to perform the task, so I check to see if the corresponding value in the problem data matrix is 0.0 (meaning the worker can’t do the task), and if so I try the next worker, being careful to wrap back to worker 0 if I exceed the index of the last worker.

Helper method AdjacentState is:

1. **static** **int**[] AdjacentState(**int**[] currState, **double**[][] problemData)
2. {
3. **int** numWorkers = problemData.Length;
4. **int** numTasks = problemData[0].Length;
5. **int**[] state = **new** **int**[numTasks];
6. **int** task = random.Next(0, numTasks);
7. **int** worker = random.Next(0, numWorkers);
8. **while** (problemData[worker][task] == 0.0) {
9. ++worker; **if** (worker > numWorkers - 1) worker = 0;
10. }
11. currState.CopyTo(state, 0);
12. state[task] = worker;
13. **return** state;
14. }

Method AdjacentState starts with a given state, then selects a random task and then selects a random worker who can do that task. Note that this is a pretty crude approach; it doesn’t check to see if the randomly generated new worker is the same as the current worker, so the return state might be the same as the current state. Depending on the nature of the problem being targeted by an SA algorithm, you might want to insert code logic that ensures an adjacent state is different from the current state.

The Energy method is shown in **Figure 4**.

**Figure 4 The Energy Method**

1. **static** **double** Energy(**int**[] state, **double**[][] problemData)
2. {
3. **double** result = 0.0;
4. **for** (**int** t = 0; t < state.Length; ++t) {
5. **int** worker = state[t];
6. **double** time = problemData[worker][t];
7. result += time;
8. }
9. **int** numWorkers = problemData.Length;
10. **int**[] numJobs = **new** **int**[numWorkers];
11. **for** (**int** t = 0; t < state.Length; ++t) {
12. **int** worker = state[t];
13. ++numJobs[worker];
14. **if** (numJobs[worker] > 1) result += 3.50;
15. }
16. **return** result;
17. }

The Energy method first walks through each task in the state array, grabs the assigned worker value, looks up the time required in the problem data matrix, and accumulates the result. Next, the method counts the number of times a worker does more than one task and adds a 3.5-hour retooling penalty for every additional task. In this example, computing the energy of a state is quick; however, in most realistic SA algorithms, the Energy method dominates the running time, so you’ll want to make sure the method is as efficient as possible.

Helper method AcceptanceProb is:

1. **static** **double** AcceptanceProb(**double** energy, **double** adjEnergy,
2. **double** currTemp)
3. {
4. **if** (adjEnergy < energy)
5. **return** 1.0;
6. **else**
7. **return** Math.Exp((energy - adjEnergy) / currTemp);
8. }

If the energy of the adjacent state is less than (or more than, in the case of a maximization problem) the energy of the current state, the method returns 1.0, so the current state will always transition to the new, better adjacent state. But if the energy of the adjacent state is the same as or worse than the current state, the method returns a value less than 1.0, which depends on the current temperature. For high values of temperature early in the algorithm, the return value is close to 1.0, so the current state will often transition to the new, worse adjacent state. But as the temperature cools, the return value from AcceptanceProb becomes smaller and smaller, so there’s less chance of transitioning to a worse state.

The idea here is that you sometimes—especially early in the algorithm—want to go to a worse state so you don’t converge on a non-optimal local solution. By sometimes going to a worse state, you can escape non-optimal dead-end states. Notice that because the function performs arithmetic division by the value of the current temperature, the temperature can’t be allowed to reach 0. The acceptance function used here is the most common function and is based on some underlying math assumptions, but there’s no reason you can’t use other acceptance functions.

The Display and Interpret Helper methods are extremely simple, as shown in **Figure 5**.

**Figure 5 The Display and Interpret Helper Methods**

1. **static** **void** Display(**double**[][] matrix)
2. {
3. **for** (**int** i = 0; i < matrix.Length; ++i) {
4. **for** (**int** j = 0; j < matrix[i].Length; ++j)
5. Console.Write(matrix[i][j].ToString("F2") + " ");
6. Console.WriteLine("");
7. }
8. }
9. **static** **void** Display(**int**[] vector)
10. {
11. **for** (**int** i = 0; i < vector.Length; ++i)
12. Console.Write(vector[i] + " ");
13. Console.WriteLine("");
14. }
15. **static** **void** Interpret(**int**[] state, **double**[][] problemData)
16. {
17. **for** (**int** t = 0; t < state.Length; ++t) { // task
18. **int** w = state[t]; // worker
19. Console.Write("Task [" + t + "] assigned to worker ");
20. Console.WriteLine(w + ", " + problemData[w][t].ToString("F2"));
21. }
22. }

## Some Weaknesses

SA algorithms are simple to implement and can be powerful tools, but they do have weaknesses. Because these algorithms are most often used in situations where there’s no good deterministic solving algorithm, in general you won’t know when, or even if, you hit an optimal solution. For example, in **Figure 1**, the best solution found had an energy of 20.5 hours, but by running the algorithm a bit longer you can find a state that has energy of 19.5 hours. So, when using SAs, you must be willing to accept a good but not necessarily optimal solution. A related weakness with SA algorithms and other algorithms based on the behavior of natural systems is that they require the specification of free parameters such as the initial temperature and the cooling rate. The effectiveness and performance of these algorithms, including SAs, are often quite sensitive to the values you select for the free parameters.

SA algorithms are closely related to Simulated Bee Colony (SBC) algorithms, which I described in the April 2011 issue ([msdn.microsoft.com/magazine/gg983491](http://msdn.microsoft.com/magazine/gg983491)). Both techniques are well suited for solving combinatorial, non-numeric optimization problems. In general, SAs are faster than SBCs, but SBCs tend to produce better solutions at the expense of performance.

The use of artificial intelligence techniques in software testing is an area that’s almost entirely unexplored. One example where SAs might be used in software testing is as algorithm validation. Suppose you have some combinatorial problem that can in fact be solved using a deterministic algorithm. One example is the graph shortest-path problem, which can be solved by several efficient but relatively complicated algorithms such as Dijkstra’s algorithm. If you’ve coded a shortest-path algorithm, you could test it by quickly coding up a simple SA algorithm and comparing results. If the SA result is better than the deterministic algorithm, you know you have a bug in your deterministic algorithm.