Article01/25/2016

February 2012

Volume 27 Number 02

Test Run - Ant Colony Optimization

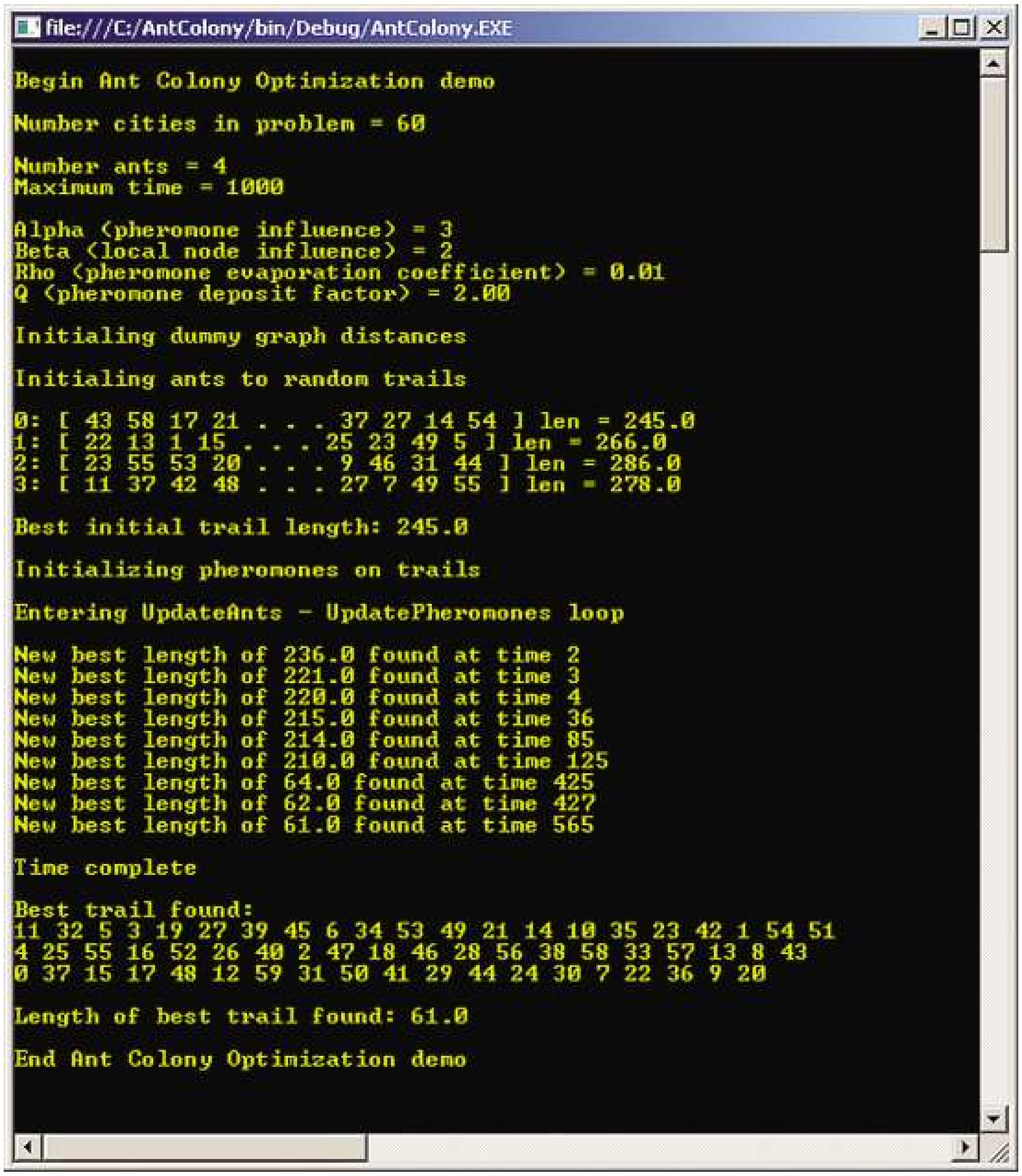
By James McCaffrey | February 2012

In this month's column I present C# code that implements an Ant Colony Optimization (ACO) algorithm to solve the Traveling Salesman Problem (TSP). An ACO algorithm is an artificial intelligence technique based on the pheromone-laying behavior of ants; it can be used to find solutions to exceedingly complex problems that seek the optimal path through a graph. The best way to see where I'm headed is to take a look at the screenshot in **Figure 1**. In this case, the demo is solving an instance of the TSP with the goal of finding the shortest path that visits each of 60 cities exactly once. The demo program uses four ants; each ant represents a potential solution. ACO requires the specification of several parameters such as the pheromone influence factor (alpha) and the pheromone evaporation coefficient (rho), which I'll explain later. The four ants are initialized to random trails through the 60 cities; after initialization, the best ant has a shortest trail length of 245.0 units. The key idea of ACO is the use of simulated pheromones, which attract ants to better trails through the graph. The main processing loop alternates between updating the ant trails based on the current pheromone values and updating the pheromones based on the new ant trails. After the maximum number of times (1,000) through the main processing loop, the program displays the best trail found and its corresponding length (61.0 units).

The 60-city graph is artificially constructed so that every city is connected to every other city, and the distance between any two cities is a random value between 1.0 and 8.0 arbitrary units (miles, km and so forth). There's no easy way to solve the TSP. With 60 cities, assuming you can start at any city and go either forward or backward, and that all cities are connected, there are a total of (60 - 1)! / 2 = 69,341,559,272,844,917,868,969,509,860,194,703,172,951,438,386,343,716,270,410,647,470, 080,000,000,000,000 possible solutions. Even if you could evaluate 1 billion possible solutions per second, it would take about 2.2 * 1063 years to check them all, which is many times longer than the estimated age of the universe.

ACO is a meta-heuristic, meaning that it's a general framework that can be used to create a specific algorithm to solve a specific graph path problem. Although ACO was proposed in a 1991 doctoral thesis by M. Dorigo, the first detailed description of the algorithm is generally attributed to a 1996 follow-up paper by M. Dorigo, V. Maniezzo and A. Colorni. Since then, ACO has been widely studied and modified, but, somewhat curiously, very few complete and correct implementations are available online.

This column assumes you have intermediate-to-advanced programming skills. I implement the ACO program using C#, but you shouldn't have too much trouble refactoring my code to a different language, such as JavaScript. I decided to avoid all use of object-oriented programming (OOP) to keep the ideas of the algorithm as clear as possible. Because of space limitations, I can't present all of the code shown running in **Figure 1**. I'll go over the trickiest parts and you can download the complete code from msdn.microsoft.com/magazine/msdnmag0212. Although you might never use ACO code directly, many of its techniques, such as roulette wheel selection, can be interesting and useful additions to your technical skill set.



```
Begin Ant Colony Optimization demo

Number cities in problem = 60

Number ants = 4
Maximum time = 1000

Alpha (pheromone influence) = 3
Beta (local node influence) = 2
Rho (pheromone evaporation coefficient) = 0.01
Q (pheromone deposit factor) = 2.00

Initialing dummy graph distances

Initialing ants to random trails

0: [ 43 58 17 21 . . . 37 27 14 54 ] len = 245.0
1: [ 22 13 1 15 . . . 25 23 49 5 ] len = 266.0
2: [ 23 55 53 20 . . . 9 46 31 44 ] len = 286.0
3: [ 11 37 42 48 . . . 27 7 49 55 ] len = 278.0

Best initial trail length: 245.0

Initializing pheromones on trails

Entering UpdateAnts - UpdatePheromones loop

New best length of 236.0 found at time 2
New best length of 221.0 found at time 3
New best length of 220.0 found at time 4
New best length of 215.0 found at time 36
New best length of 214.0 found at time 85
New best length of 210.0 found at time 125
New best length of 64.0 found at time 425
New best length of 62.0 found at time 427
New best length of 61.0 found at time 565

Time complete

Best trail found:
11 32 5 3 19 27 39 45 6 34 53 49 21 14 10 35 23 42 1 54 51
4 25 55 16 52 26 40 2 47 18 46 28 56 38 58 33 57 13 8 43
0 37 15 17 48 12 59 31 50 41 29 44 24 30 7 22 36 9 20

Length of best trail found: 61.0

End Ant Colony Optimization demo
```

Figure 1 Ant Colony Optimization in Action

Program Structure

I implemented the ACO demo program as a single C# console application. The overall structure of the program, with most WriteLine statements removed, is shown in **Figure 2**. Although some parts are tricky, the program isn't as complicated as **Figure2** suggests because many of the methods are short helper methods.

Figure 2 Ant Colony Optimization Program Structure

XML

```
using System;

namespace AntColony
{
  class AntColonyProgram
  {
    static Random random = new Random(0);
    static int alpha = 3;
    static int beta = 2;
    static double rho = 0.01;
    static double Q = 2.0;

    static void Main(string[] args)
    {
      try
      {
        Console.WriteLine("\nBegin Ant Colony Optimization demo\n");
        int numCities = 60;
        int numAnts = 4;
        int maxTime = 1000;

        int[][] dists = MakeGraphDistances(numCities);
        int[][] ants = InitAnts(numAnts, numCities);

        int[] bestTrail = BestTrail(ants, dists);
        double bestLength = Length(bestTrail, dists);

        double[][] pheromones = InitPheromones(numCities);

        int time = 0;
        while (time < maxTime)
        {
          UpdateAnts(ants, pheromones, dists);
          UpdatePheromones(pheromones, ants, dists);

          int[] currBestTrail = BestTrail(ants, dists);
          double currBestLength = Length(currBestTrail, dists);
          if (currBestLength < bestLength)
          {
            bestLength = currBestLength;
            bestTrail = currBestTrail;
          }
          ++time;
        }

        Console.WriteLine("\nBest trail found:");
        Display(bestTrail);
        Console.WriteLine("\nLength of best trail found: " +
          bestLength.ToString("F1"));
        Console.WriteLine("\nEnd Ant Colony Optimization demo\n");
      }
      catch (Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
    } // Main

    static int[][] InitAnts(int numAnts, int numCities) { . . }

    static int[] RandomTrail(int start, int numCities) { . . }

    static int IndexOfTarget(int[] trail, int target) { . . }

    static double Length(int[] trail, int[][] dists) { . . }

    static int[] BestTrail(int[][] ants, int[][] dists) { . . }

    static double[][] InitPheromones(int numCities) { . . }

    static void UpdateAnts(int[][] ants, double[][] pheromones,
      int[][] dists) { . . }

    static int[] BuildTrail(int k, int start, double[][] pheromones,
      int[][] dists) { . . }

    static int NextCity(int k, int cityX, bool[] visited, double[][]
pheromones,
      int[][] dists) { . . }

    static double[] MoveProbs(int k, int cityX, bool[] visited,
      double[][] pheromones, int[][] dists) { . . }

    static void UpdatePheromones(double[][] pheromones, int[][] ants,
      int[][] dists) { . . }

    static bool EdgeInTrail(int nodeX, int nodeY, int[] trail) { . . }

    static int[][] MakeGraphDistances(int numCities) { . . }

    static double Distance(int cityX, int cityY, int[][] dists) { . . }

    static void Display(int[] trail) { . . }

    static void ShowAnts(int[][] ants, int[][] dists) { . . }

    static void Display(double[][] pheromones) { . . }
  }
}
```

I used Visual Studio to create a console application program named AntColony. In the Solution Explorer window I renamed the default Program.cs file to AntColonyProgram.cs, which automatically renamed the single class in the project. I deleted all the template-generated using statements except for the System namespace—ACO typically doesn't need much library support. The two key methods are UpdateAnts and UpdatePheromones. Method UpdateAnts calls helper BuildTrail, which calls NextCity, which calls MoveProbs. Method UpdatePheromones calls helper EdgeInTrail, which calls IndexOfTarget.

I declared a class-scope Random object named random. ACO algorithms are probabilistic as you'll see shortly. The class-scope variables alpha, beta, rho and Q control the behavior of the ACO algorithm. I use these variable names because they were used in the original description of ACO.

Setting up the Problem

I used method MakeGraphDistances to set up an artificial graph:

XML

```
static int[][] MakeGraphDistances(int numCities)
{
  int[][] dists = new int[numCities][];
  for (int i = 0; i < dists.Length; ++i)
    dists[i] = new int[numCities];
  for (int i = 0; i < numCities; ++i)
    for (int j = i + 1; j < numCities; ++j) {
      int d = random.Next(1, 9); // [1,8]
      dists[i][j] = d; dists[j][i] = d;
    }
  return dists;
}
```

For a real-world graph problem, you'd probably read graph-adjacency and distance-between-node data from a text file into some sort of data structure. Here I simulated a graph by creating an array of arrays where the row index i represents the from-city and the column index j represents the to-city. Notice that all cities are connected, distances are symmetric and the distance from a city to itself is 0.

Once I have a distances data structure I can use it for a Distance method:

XML

```
static double Distance(int cityX, int cityY, int[][] dists)
{
  return dists[cityX][cityY];
}
```

To minimize the amount of code presented, I've omitted normal error checking, such as making sure that the cityX and cityY parameters are valid.

Initiating the Ants and the Pheromones

In this non-OOP implementation, an ant is simply an array of int values that represent the trail, or path, from an initial city through all other cities. A collection of ants is an array of arrays in which the first index indicates the ant:

XML

```
static int[][] InitAnts(int numAnts, int numCities)
{
  int[][] ants = new int[numAnts][];
  for (int k = 0; k < numAnts; ++k) {
    int start = random.Next(0, numCities);
    ants[k] = RandomTrail(start, numCities);
  }
  return ants;
}
```

The initialization method allocates a row for the trail for each ant, picks a random start city and then calls a helper method RandomTrail:

XML

```
static int[] RandomTrail(int start, int numCities)
{
  int[] trail = new int[numCities];
  for (int i = 0; i < numCities; ++i) { trail[i] = i; }

  for (int i = 0; i < numCities; ++i) {
    int r = random.Next(i, numCities);
    int tmp = trail[r]; trail[r] = trail[i]; trail[i] = tmp;
  }

  int idx = IndexOfTarget(trail, start);
  int temp = trail[0]; trail[0] = trail[idx]; trail[idx] = temp;
  return trail;
}
```

The RandomTrail helper allocates a trail and initializes it to 0, 1, 2, ... numCities-1. Next, the method uses the Fisher-Yates shuffle algorithm to randomize the order of the cities in the trail. The specified start city is then located and swapped into position trail[0].

Pheromones are chemicals ants place on their trails; they attract other ants. More ants will travel on a shorter trail to a food source—and deposit more pheromones—than on longer trails. The pheromones slowly evaporate over time. Here's method InitPheromones:

XML

```
static double[][] InitPheromones(int numCities)
{
  double[][] pheromones = new double[numCities][];
  for (int i = 0; i < numCities; ++i)
    pheromones[i] = new double[numCities];
  for (int i = 0; i < pheromones.Length; ++i)
    for (int j = 0; j < pheromones[i].Length; ++j)
      pheromones[i][j] = 0.01;
  return pheromones;
}
```

Pheromone information is stored in an array-of-arrays-style symmetric matrix where the row index i is the from-city and the column index j is the to-city. All values are initially set to an arbitrary small value (0.01) to jump start the UpdateAnts-UpdatePheromones cycle.

Updating the Ants

The key to the ACO algorithm is the process that updates each ant and trail by constructing a new—we hope better—trail based on the pheromone and distance information. Take a look at **Figure 3**. Suppose we have a small graph with just five cities. In **Figure 3** the new trail for an ant is under construction. The trail starts at city 1, then goes to city 3, and the update algorithm is determining the next city. Now suppose the pheromone and distance information is as shown in the image. The first step in determining the next city is constructing an array I've called "taueta" (because the original research paper used the Greek letters tau and eta). The taueta value is the value of the pheromone on the edge raised to the alpha power, times one over the distance value raised to the beta power. Recall that alpha and beta are global constants that must be specified. Here I'll assume that alpha is 3 and beta is 2. The taueta values for city 1 and city 3 aren't computed because they're already in the current trail. Notice that larger values of the pheromone increase taueta, but larger distances decrease taueta.

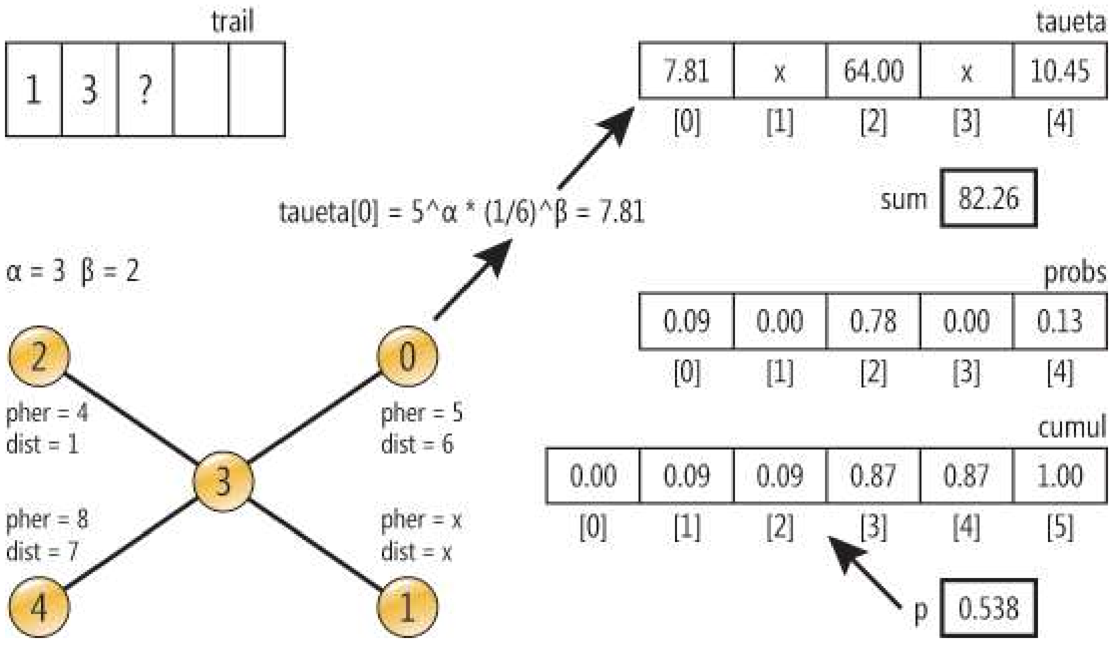


Figure 3 Updating Ant Information

After all the taueta values have been computed, the next step is to convert those values to probabilities and place them in an array I've labeled probs. The algorithm sums the taueta values, getting 82.26 in this example, and then divides each taueta value by the sum. At this point, city 0 has a probability of 0.09 of being selected and so on. Next, the algorithm needs to select the next city based on the computed probabilities. As I mentioned earlier, the ACO algorithm I'm presenting in this article uses a neat technique called roulette wheel selection. I constructed an augmented array called cumul, which holds cumulative probabilities. The size of the augmented array is one greater than the probs array, and cell [0] is seeded with 0.0. Each cell in cumul is the cumulative sum of the probabilities. After the cumul array has been constructed, a random number p between 0.0 and 1.0 is generated. Suppose p = 0.538 as shown. That p value falls between the values at [2] and [3] in the cumul array, which means that [2], or city 2, is selected as the next city.

The top-level method for updating is named UpdateAnts:

XML

```
static void UpdateAnts(int[][] ants, double[][] pheromones, int[][] dists)
{
  int numCities = pheromones.Length;
  for (int k = 0; k < ants.Length; ++k) {
    int start = random.Next(0, numCities);
    int[] newTrail = BuildTrail(k, start, pheromones, dists);
    ants[k] = newTrail;
  }
}
```

Notice that each ant is assigned a new, random starting city rather than preserving the old start city. Most of the actual work is performed by helper method BuildTrail, as shown in **Figure 4**.

Figure 4 The BuildTrail Method

XML

```
static int[] BuildTrail(int k, int start, double[][] pheromones,
  int[][] dists)
{
  int numCities = pheromones.Length;
  int[] trail = new int[numCities];
  bool[] visited = new bool[numCities];
  trail[0] = start;
  visited[start] = true;
  for (int i = 0; i < numCities - 1; ++i) {
    int cityX = trail[i];
    int next = NextCity(k, cityX, visited, pheromones, dists);
    trail[i + 1] = next;
    visited[next] = true;
  }
  return trail;
}
```

BuildTrail maintains an array of Boolean visited, so that the trail created doesn't contain duplicate cities. The value at trail[0] is seeded with a start city, then each city is added in turn by helper method NextCity, shown in **Figure 5**.

Figure 5 The NextCity Method

XML

```
static int NextCity(int k, int cityX, bool[] visited,
  double[][] pheromones, int[][] dists)
{
  double[] probs = MoveProbs(k, cityX, visited, pheromones, dists);

  double[] cumul = new double[probs.Length + 1];
  for (int i = 0; i < probs.Length; ++i)
    cumul[i + 1] = cumul[i] + probs[i];

  double p = random.NextDouble();
  for (int i = 0; i < cumul.Length - 1; ++i)
    if (p >= cumul[i] && p < cumul[i + 1])
      return i;
  throw new Exception("Failure to return valid city in NextCity");
}
```

The NextCity method implements the roulette wheel selection algorithm.Note that the algorithm will fail if the last value in the cumul array is larger than 1.00 (possibly due to rounding errors), and so you might want to add logic to always set cumul[cumul.Length-1] to 1.00. NextCity requires an array of probabilities produced by helper method MoveProbs, shown in **Figure 6**.

Figure 6 The MoveProbs Method

XML

```
static double[] MoveProbs(int k, int cityX, bool[] visited,
  double[][] pheromones, int[][] dists)
{
  int numCities = pheromones.Length;
  double[] taueta = new double[numCities];
  double sum = 0.0;
  for (int i = 0; i < taueta.Length; ++i) {
    if (i == cityX)
      taueta[i] = 0.0; // Prob of moving to self is zero
    else if (visited[i] == true)
      taueta[i] = 0.0; // Prob of moving to a visited node is zero
    else {
      taueta[i] = Math.Pow(pheromones[cityX][i], alpha) *
        Math.Pow((1.0 / Distance(cityX, i, dists)), beta);
      if (taueta[i] < 0.0001)
        taueta[i] = 0.0001;
      else if (taueta[i] > (double.MaxValue / (numCities * 100)))
        taueta[i] = double.MaxValue / (numCities * 100);
    }
    sum += taueta[i];
  }

  double[] probs = new double[numCities];
  for (int i = 0; i < probs.Length; ++i)
    probs[i] = taueta[i] / sum;
  return probs;
}
```

The taueta values can be very small (if the distance value is very large) or very large (if the pheromone value is large), which can produce difficulties for the algorithm. To deal with this, I check the taueta values and impose arbitrary min and max values.

Updating the Pheromones

Updating pheromone information is much easier than updating the ant trail information. The key lines of code in method UpdatePhermones are:

XML

```
double length = Length(ants[k], dists);
double decrease = (1.0 - rho) * pheromones[i][j];
double increase = 0.0;
if (EdgeInTrail(i, j, ants[k]) == true)
  increase = (Q / length);
pheromones[i][j] = decrease + increase;
```

Each pheromone value is decreased, simulating evaporation, and increased, simulating the deposit of pheromones by ants on the trail. The decrease effect is produced by multiplying the current pheromone value by a factor less than 1.0 that depends on global parameter rho. The larger rho is, the greater the decrease in pheromone value. The increase effect is produced by adding a proportion of the current ant's total trail length, where the proportion is determined by global parameter Q. Larger values of Q increase the amount of pheromone added. Method UpdatePheromones calls helper EdgeInTrail, which determines if a segment between two cities is on the ant's current trail. You can check out the code download for this article for the details (msdn.com/magazine/msdnmag0212).

Wrapping Up

Let me emphasize that there are many variations of ACO; the version I've presented here is just one of many possible approaches. ACO advocates have applied the algorithm to a wide range of combinatorial optimization problems. Other combinatorial optimization algorithms based on the behavior of natural systems include Simulated Annealing (SA), which I covered last month (msdn.microsoft.com/magazine/hh708758), and Simulated Bee Colony (SBC), which I covered in my April 2011 column (msdn.microsoft.com/magazine/gg983491). Each approach has strengths and weaknesses. In my opinion, ACO is best-suited for problems that closely resemble the

Traveling Salesman Problem, while SA and SBC are better for more general combinatorial optimization problems, such as scheduling.

ACO, in common with other meta-heuristics based on natural systems, is quite sensitive to your choice of free global parameters—alpha, beta and so on. Although there has been quite a bit of research on ACO parameters, the general consensus is that you must experiment a bit with free parameters to get the best combination of performance and solution quality.

Dr. James McCaffrey *works for Volt Information Sciences Inc., where he manages technical training for software engineers working at the Microsoft Redmond, Wash., campus. He's worked on several Microsoft products including Internet Explorer and MSN Search. McCaffrey is the author of ".NET Test Automation Recipes" (Apress, 2006) and can be reached at jmccaffrey@volt.com or jammc@microsoft.com.*

Thanks to the following Microsoft technical experts for reviewing this article: **Dan Liebling** and **Anne Loomis Thompson**