

EDUCATOR'S CORNER

Ant Colony System Optimization

Richard Wiener

Successful heuristic algorithms for solving combinatorial optimization problems have mimicked processes observed in nature. Two highly successful families of algorithms that do this are simulated annealing and genetic algorithms. Here, a third family of algorithms, ant colony optimization is explored and implemented in C#. The test bed for evaluating the quality of solutions is based on several Traveling Salesperson Problems (TSP) of varying size using real-world data (Euclidian problems) with known solutions.

Ants form a distributed system and present highly organized social organization. When an ant searches for food it leaves a chemical trail of pheromones. This trail may be used by other ants to follow the trail to food.

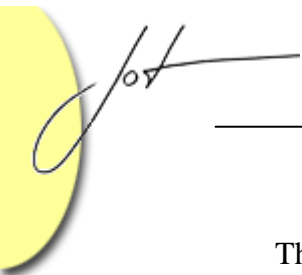
Suppose an ant were deposited on some randomly chosen city in a TSP problem. Tours could be generated by this artificial ant as follows: A probability is computed for each of the possible target cities that the ant can travel to from the source city that it is occupying. This probability is based on two factors: the distance to the target city (shorter distances provide higher probability) and the pheromone that has been previously deposited on the edge from the source to target city. Artificial ants are not allowed to re-visit a city.

The ant system (AS) algorithm works as follows. In AS, m artificial ants are placed on random chosen cities. At each tour construction step for ant k , the probability of choosing city s from city r is:

$$p_k(r, s) = \begin{cases} \frac{[\tau(r, s)] \cdot [\eta(r, s)]^\beta}{\sum_{u \in M_k} [\tau(r, u)] \cdot [\eta(r, u)]^\beta} & \text{if } s \notin M_k \\ 0 & \text{otherwise} \end{cases}$$

The heuristic $\eta(r, s)$ is $1.0 / \text{cost}(r, s)$.

The $\tau(r, s)$ is the pheromone on the edge r, s .



The value β is a number between 2 and 5 and determines how much weight to give the cost heuristic.

So the smaller the edge cost and higher the pheromone value, the higher the probability that the ant will go from city r to s .

On each iteration of the simulation, after all the randomly placed ants have constructed their tours, the pheromone trails are updated. First all the pheromone on each edge is reduced by a constant factor (evaporation). This helps to ensure exploration and stops premature convergence to the edges on the best tour so far which at the beginning is probably not a very good tour. Then each ant deposits pheromone on the edges corresponding to the cities visited given by the reciprocal of the tour cost. So a poor tour (high tour cost) will result in less pheromone being deposited than a good tour. The best tour among the m ants is found (best tour to date). Additional pheromone is deposited based on the best tour to date on the edges of this best tour (reciprocal of best tour cost). After a user-defined number of iterations, the best tour cost is reported.

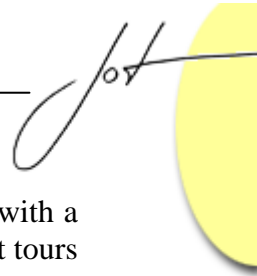
Stagnation seems to be a problem when the Ant System algorithm is applied to problems of size 100 or more. Convergence to a tour that is typically about 5 to 8 percent above the known optimum occurs.

As documented in the book *Ant Colony Optimization* by Marco Dorigo and Thomas Stutzle (MIT Press, 2004), an improved algorithm called Ant Colony System (ACS) is presented. This ACS algorithm is based on changes made to the basic Ant System described above.

Specifically,

- 1) There is no global evaporation applied to all the edges after each iteration as there is in the Ant System.
- 2) When constructing a tour in ACS, with a probability threshold of about 0.9, an ant chooses its next city by finding the maximum of the pheromone and reciprocal of distance raised to the beta power products among all cities not yet visited. With probability 0.1, the Ant System heuristic for choosing the next city is used. The ACS algorithm is much more aggressive since it favors the maximum rather than making a random choice.
- 3) After all ants have completed their tours (one iteration), a global pheromone update is performed only on the edges associated with the best tour to date (i.e. only the best ant to date updates pheromone on its edges). The update is performed as a weighted average of the earlier pheromone for each edge and the reciprocal of the best tour cost to date. The effect of this is to cause some small evaporation on the edges of the optimum tour as well as add some additional pheromone to the edges.
- 4) As each ant moves from city i to j as its tour is being constructed it uses a local trail updating rule that causes the pheromone on that edge to be decreased. This encourages more exploration and works against stagnation.

More details of the ACS algorithm may be found on pages 76 to 78 in the book cited above.



The C# implementation details of this ACS algorithm are presented below along with a GUI application that outputs the progress of the algorithm and depicts the evolving best tours graphically. Sample output is also presented.

Listing 1 Class Global

```
using System;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;

namespace AntColonyGUI {

    public class Global {
        public static List<int> bestTourSoFar;
        public static Random random = new Random();
        public static int initialTourCost;

        public static Point [] rawData;
        public static int [,] cost;
        public static double[,] heuristic; // Reciprocal of cost
        public static double[,] pheromone;
        public static void Initialize(int n, Graphics g, int [,] c) {
            if (c == null) {
                cost = new int[n + 1, n + 1]; // natural indexing
                for (int row = 1; row <= n; row++) {
                    for (int col = row; col <= n; col++) {
                        if (row == col) {
                            cost[row, col] = 0;
                        } else {
                            double xDist =
                                rawData[row].X - rawData[col].X;
                            double yDist =
                                rawData[row].Y - rawData[col].Y;
                            cost[row, col] =
                                (int)(Math.Sqrt(xDist * xDist +
                                    yDist * yDist) + 0.5);
                            cost[col, row] = cost[row, col];
                        }
                    }
                }
            }
            else {
                cost = c;
                for (int index = 1; index <= n; index++) {
                    cost[index, index] = 0;
                }
            }
        }
    }
}
```

```
// Obtain shortest city tour using greedy starting at 1
initialTourCost = 0;
int numbersCitiesVisited = 1;
int city = 1;
List<int> visited = new List<int>();
visited.Add(0);
visited.Add(1);
do {
    // Find shortest distance from city
    double[] distances = new double[n + 1];
    for (int index = 1; index <= n; index++) {
        if (index == city || visited.Contains(index)) {
            distances[index] = Int32.MaxValue;
        } else {
            distances[index] = Global.cost[city, index];
        }
    }
    // Find shortest among distances
    int shortestIndex = 0;
    double shortest = Double.MaxValue;
    for (int index = 1; index <= n; index++) {
        if (distances[index] < shortest) {
            shortestIndex = index;
            shortest = distances[index];
        }
    }
    int previousCity = city;
    city = shortestIndex;
    visited.Add(city);
    numbersCitiesVisited++;
    initialTourCost += cost[previousCity, city];
} while (visited.Count <= n);
initialTourCost += cost[city, 1];
visited.Add(1);

pheromone = new double[n + 1, n + 1]; // natural indexing
for (int row = 1; row <= n; row++) {
    for (int col = 1; col <= n; col++) {
        pheromone[row, col] = 1.0 / (n * initialTourCost);
    }
}

heuristic = new double[n + 1, n + 1];
for (int row = 1; row <= n; row++) {
    for (int col = row; col <= n; col++) {
        if (row == col) {
            heuristic[row, col] = Double.MaxValue;
        } else {
            heuristic[row, col] = 1.0 / cost[row, col];
            heuristic[col, row] = heuristic[row, col];
        }
    }
}
```



```
}  
    }  
}
```

Listing 2 Class Ant

```
using System;  
using System.Collections;  
using System.Drawing;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
namespace AntColonyGUI {  
  
    public class Ant {  
        // Constants  
        private const int ALPHA= 1;  
        private const int BETA = 5;  
        private const double LOCAL_PHEROMONE_UPDATE = 0.1;  
  
        // Fields  
        private List<int> citiesVisited;  
        private int startCity;  
        private int numberCities;  
        private int tourCost;  
        private double probabilityThreshold;  
  
        // Constructor  
        public Ant(int startingCity, int numberCities, double  
            probabilityThreshold) {  
            startCity = startingCity;  
            this.numberCities = numberCities;  
            this.probabilityThreshold = probabilityThreshold;  
            citiesVisited = new List<int>();  
            tourCost = 0;  
        }  
  
        // Commands  
        public void ConstructTour() {  
            citiesVisited.Add(0); // for natural indexing  
            citiesVisited.Add(startCity);  
            int previousCity = startCity;  
            do {  
                int nextCity = AddEdgeFrom(previousCity);  
                if (!citiesVisited.Contains(nextCity)) {  
                    citiesVisited.Add(nextCity);  
                }  
                tourCost += Global.cost[previousCity, nextCity];  
            }  
        }  
    }  
}
```

```

// Ant colony system local trail update
Global.pheromone[previousCity, nextCity] =
    (1.0 - LOCAL_PHEROMONE_UPDATE) *
    Global.pheromone[previousCity, nextCity] +
    LOCAL_PHEROMONE_UPDATE *
    (1.0 / (numberCities * Global.initialTourCost));
previousCity = nextCity;

} while (citiesVisited.Count <= numberCities);
tourCost += Global.cost[previousCity, startCity];
citiesVisited.Add(startCity);
if (citiesVisited.Count != numberCities + 2) {
    Console.WriteLine("ERROR IN CONSTRUCTING TOUR");
}
}

// Queries
public int AddEdgeFrom(int city) {
    // Based on modified Ant Colony System heuristic
    double r = Global.random.NextDouble();
    if (r <= probabilityThreshold) {
        double[] arcWeights = new double[numberCities + 1];
        for (int index = 1; index <= numberCities; index++) {
            if (index == city ||
                citiesVisited.Contains(index)) {
                arcWeights[index] = 0.0;
            } else {
                arcWeights[index] = Global.pheromone[city,
                    index] *
                    Math.Pow(Global.heuristic[city, index], BETA);
            }
        }
        // Get the largest in arcWeights
        double largest = -1.0;
        int largestIndex = 0;
        for (int index = 1; index <= numberCities; index++) {
            if (arcWeights[index] > largest) {
                largest = arcWeights[index];
                largestIndex = index;
            }
        }
        if (arcWeights[largestIndex] == 0.0) {
            // Return the first city not yet visited
            for (int index = 1; index <= numberCities; index++)
            {
                if (!citiesVisited.Contains(index)) {
                    return index;
                }
            }
        } else {
            return largestIndex;
        }
    }
}

```



```
    }
} else { // Same as Ant System heuristic
    double denominator = 0.0;
    for (int index = 1; index <= numberCities; index++) {
        if (index != city &&
            !citiesVisited.Contains(index)) {
            denominator += Global.pheromone[city, index] *
                Math.Pow(Global.heuristic[city, index],
                    BETA);
        }
    }
    if (denominator == 0.0) {
        // Return the first city not yet visited
        for (int index = 1;
            index <= numberCities; index++) {
            if (!citiesVisited.Contains(index)) {
                return index;
            }
        }
    }
    // prob of going from city to index
    double[] prob = new double[numberCities + 1];
    for (int index = 1; index <= numberCities; index++) {
        if (index == city ||
            citiesVisited.Contains(index)) {
            prob[index] = 0.0;
        } else {
            prob[index] = Global.pheromone[city, index] *
                Math.Pow(Global.heuristic[city, index],
                    BETA) / denominator;
        }
    }
    double rnd = Global.random.NextDouble();
    double sum = 0.0;
    for (int index = 1; index <= numberCities; index++) {
        sum += prob[index];
        if (rnd <= sum && index != city &&
            !citiesVisited.Contains(index)) {
            return index;
        }
    }
}
// Unreachable
return 0;
}

// Properties
public int TourCost {
    get { // Read-only
        return tourCost;
    }
}
}
```

```

        public List<int> CitiesVisited {
            get {
                return citiesVisited;
            }
        }
    }
}

```

Listing 3 Class AntColonyGUIApp

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.Threading;

namespace AntColonyGUI {

    public partial class AntColonyGUIApp : Form {

        // Fields
        private Graphics g;
        private Thread computation;
        private int numberIterations = 20000;
        private const int INTERVAL_REPORT_OUTPUT = 200;
        private const double GLOBAL_PHEROMONE_UPDATE = 0.1;
        private const int NUMBER_ANTS = 10;
        private List<int> bestTour = null;
        private int bestTourCost = Int32.MaxValue;
        private Ant[] ants = new Ant[NUMBER_ANTS + 1];
        private int numberCities;
        private int largestX, largestY; // city coordinates
        private bool stop = false;

        public AntColonyGUIApp() {
            InitializeComponent();
            g = panel.CreateGraphics();
        }

        public void Start(int n, String filename) {
            // Parameters
            numberIterations =
                Convert.ToInt32(iterationsBox.Text.Trim());

```




```
StreamReader inputStream = new StreamReader(filename);
int[,] cost = null;
try {
    // Read input data
    Global.rawData = new Point[n + 1]; // natural indexing
    String delimiterString = " "; // white space
    char[] delimiter = delimiterString.ToCharArray();
    String line = inputStream.ReadLine();
    int city = 1;
    while (line != null) {
        String[] words = line.Split(delimiter);
        double x = Convert.ToDouble(words[1]);
        double y = Convert.ToDouble(words[2]);
        Global.rawData[city] = new Point((int)x, (int)y);
        line = inputStream.ReadLine();
        city++;
    }
    inputStream.Close();
} catch (Exception) {
    output.AppendText("Error reading input data.\n");
    inputStream.Close();
    return;
}

Global.Initialize(n, g, cost);

// Get largestX and largestY
largestX = 0;
largestY = 0;
for (int index = 1; index <= numberCities; index++) {
    if (Global.rawData[index].X > largestX) {
        largestX = Global.rawData[index].X;
    }
    if (Global.rawData[index].Y > largestY) {
        largestY = Global.rawData[index].Y;
    }
}

// Scale the raw data for display purposes
for (int index = 1; index <= numberCities; index++) {
    Global.rawData[index].X =
        (int)(Global.rawData[index].X * 800.0 / largestX);
    Global.rawData[index].Y =
        (int)(Global.rawData[index].Y * 800.0 / largestY);
}

computation = new Thread(new ThreadStart(Compute));
computation.IsBackground = true;
computation.Start();
}
```

```

public void Compute() {
    for (int iteration = 1;
        !stop && iteration <= numberIterations; iteration++) {
        // Reuse ants array by inserting a fresh collection of
        // new ants
        for (int antNumber = 1; antNumber <= NUMBER_ANTS;
            antNumber++) {
            int startingCity =
                loba1.random.Next(numberCities) + 1;
            Ant workerAnt = new Ant(startingCity, numberCities,
                Convert.ToDouble(thresholdBox.Text.Trim()));
            workerAnt.ConstructTour();
            ants[antNumber] = workerAnt;
        }
        for (int antNumber = 1; antNumber <= NUMBER_ANTS;
            antNumber++) {
            if (ants[antNumber].TourCost < bestTourCost) {
                bestTourCost = ants[antNumber].TourCost;
                bestTour = ants[antNumber].CitiesVisited;
            }
        }
        // Only the best ant so far deposits pheromone
        for (int index = 1; index <= numberCities; index++) {
            Global.pheromone[bestTour[index],
                bestTour[index + 1]] =
                (1.0 - GLOBAL_PHEROMONE_UPDATE) *
                Global.pheromone[bestTour[index],
                bestTour[index + 1]] +
                GLOBAL_PHEROMONE_UPDATE * 1.0 /
                estTourCost;
        }
        if (iteration == 1 ||
            iteration % INTERVAL_REPORT_OUTPUT == 0) {
            output.AppendText(
                "\nIteration: " + iteration + ":");
            output.AppendText(
                " Best tour cost: " + bestTourCost + "\n");

                // Apply local optimization
            ThreeOpt();
            output.AppendText(
                "After applying 3-opt local search, Best tour
                cost: " + bestTourCost + "\n");

            if (knownOptimumBox.Text.Trim().Length > 0) {
                int opt =
                    Convert.ToInt32(knownOptimumBox.Text.Trim());
                output.AppendText("\tError: " +
                    String.Format("{0:f}", 100.0 *
                        (bestTourCost - opt) / opt) +
                    " percent.\n");
            }
        }
    }
}

```



```
    }

    // Test to see whether best tour is valid
    for (int index = 1;
        index <= numberCities; index++) {
        if (!bestTour.Contains(index)) {
            output.AppendText(
                " Invalid tour since city " + index +
                " is missing.\n");
            return;
        }
    }
    DrawTour();
}
}
output.AppendText("Best tour: \n");
foreach (int city in bestTour) {
    if (city != 0) {
        output.AppendText(city + " ");
    }
}
// Final check on tour cost
int sum = 0;
for (int index = 1; index <= numberCities; index++) {
    sum += Global.cost[bestTour[index],
        bestTour[index + 1]];
}
output.AppendText(
    "\nCost for this best tour: " + sum + "\n");
output.AppendText("\n");
// Test to see whether best tour is valid
for (int index = 1; index <= numberCities; index++) {
    if (!bestTour.Contains(index)) {
        output.AppendText(
            "Invalid tour since city " + index + " is
            missing.\n");
        return;
    }
}
output.AppendText("All cities present and valid tour.\n");
}

private void DrawTour() {
    g.Clear(Color.White);
    for (int index = 1; index <= numberCities; index++) {
        g.DrawEllipse(new Pen(Color.Red, 1),
            new Rectangle(Global.rawData[bestTour[index]],
                g.DrawLine(new Pen(Color.Black, 1),
                    Global.rawData[bestTour[index]],
                    Global.rawData[bestTour[index + 1]]);
        }
    }
}
```

```
}

private void ThreeOpt() {
    int x1 = 0, x2 = 0, x3 = 0, x4 = 0, x5 = 0, x6 = 0;
    int[] xb = new int[numberCities + 2];
    int[] origb = new int[numberCities + 2];

    int x1orig = 0, x2orig = 0, x3orig = 0, x4orig = 0,
        x5orig = 0, x6orig = 0;
    int firstb = 0, secondb = 0, thirdb = 0;

    for (int count = 1; count <= numberCities; count++) {

        // Permute tour
        for (int index = 1; index <= numberCities; index++) {
            bestTour[index] = bestTour[index + 1];
        }
        bestTour[numberCities + 1] = bestTour[1];

        int bestGain = 0;
        int gain1 = 0;
        int gain2 = 0;
        int type2 = 3;
        // See page 445 in
        // Design and Analysis of Algorithms by
        // Levitin (Second Edition), Addison Wesley
        x1 = 1;
        x2 = 2;

        for (int i = 5; i <= numberCities - 2; i++) {
            x5 = i;
            x6 = i + 1;
            for (int k = 3; k <= i - 2; k++) {
                x3 = k;
                x4 = k + 1;

                x1orig = x1;
                x2orig = x2;
                x3orig = x3;
                x4orig = x4;
                x5orig = x5;
                x6orig = x6;

                int first =
                    Global.cost [bestTour [x1orig],
                        bestTour [x2orig]] +
                    Global.cost [bestTour [x3orig],
                        bestTour [x4orig]] +
                    Global.cost [bestTour [x5orig],
                        bestTour [x6orig]];
```



```
int second = Global.cost [bestTour [x2] ,
    bestTour [x5]] +
    Global.cost [bestTour [x1] ,
    bestTour [x4]] +
    Global.cost [bestTour [x3] , bestTour [x6]] ;
int third = Global.cost [bestTour [x1] ,
    bestTour [x4]] +
    Global.cost [bestTour [x3] , bestTour [x5]] +
    Global.cost [bestTour [x2] , bestTour [x6]] ;
gain1 = first - second;
gain2 = first - third;
if (gain1 > 0 && gain1 >= gain2 &&
    gain1 > bestGain) {
    bestGain = gain1;
    type2 = 1;

    xb[1] = x1;
    xb[2] = x2;
    xb[3] = x3;
    xb[4] = x4;
    xb[5] = x5;
    xb[6] = x6;
    origb[1] = x1orig;
    origb[2] = x2orig;
    origb[3] = x3orig;
    origb[4] = x4orig;
    origb[5] = x5orig;
    origb[6] = x6orig;
    count = 0; // Must start another set of
                // permutations

    firstb = first;
    secondb = second;
    thirdb = third;
} else if (gain2 > 0 && gain2 > gain1 &&
    gain2 > bestGain) {
    bestGain = gain2;
    type2 = 2;

    xb[1] = x1;
    xb[2] = x2;
    xb[3] = x3;
    xb[4] = x4;
    xb[5] = x5;
    xb[6] = x6;
    origb[1] = x1orig;
    origb[2] = x2orig;
    origb[3] = x3orig;
    origb[4] = x4orig;
    origb[5] = x5orig;
    origb[6] = x6orig;
    count = 0; // Must start another set of
                // permutations
```

```

        firstb = first;
        secondb = second;
        thirdb = third;
    }
}

if (type2 == 1) {
    // Figure b on page 445 of Levitan's book

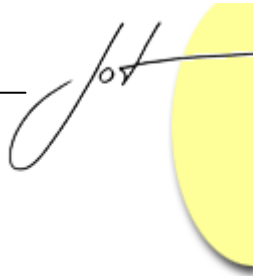
    int[] next = new int[numberCities + 2];
    int[] temp = new int[numberCities + 2];
    int z = 0;
    next[z++] = 0;
    next[z++] = 1;
    next[z++] = xb[4];
    for (int j = 1; j <= origb[5] - origb[4]; j++) {
        next[z++] = xb[4] + j;
    }
    next[z++] = xb[2];
    for (int j = 1; j <= origb[3] - origb[2]; j++) {
        next[z++] = xb[2] + j;
    }
    next[z++] = xb[6];
    for (int j = 1; j <= numberCities - origb[6]; j++)
    {
        next[z++] = xb[6] + j;
    }
    next[z++] = 1;

    for (int index = 1;
        index <= numberCities + 1; index++) {
        temp[index] = bestTour[next[index]];
    }

    for (int index = 1;
        index <= numberCities + 1; index++) {
        bestTour[index] = temp[index];
    }

    // Test to see whether best tour is valid
    for (int index = 1;
        index <= numberCities; index++) {
        if (!bestTour.Contains(index)) {
            MessageBox.Show("Invalid tour since city "
                + index + " is missing.\n");
        }
    }
}

```



```
bestTourCost = 0;
for (int index = 1;
    index <= numberCities; index++) {
    bestTourCost += Global.cost[bestTour[index],
                             bestTour[index + 1]];
}
}

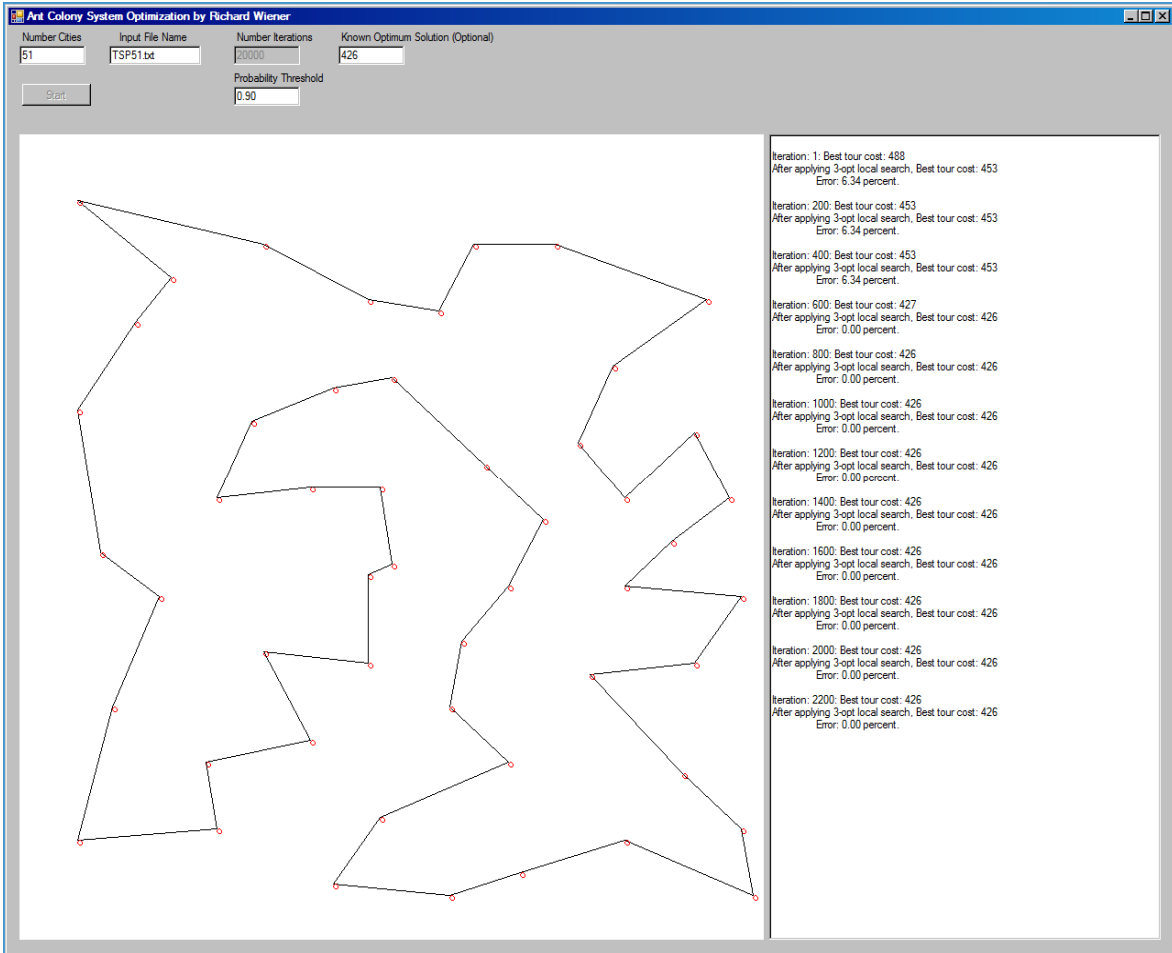
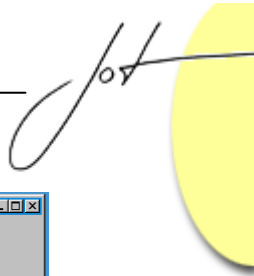
else if (type2 == 2) {
    // Figure c on page 445 of Levitan's book

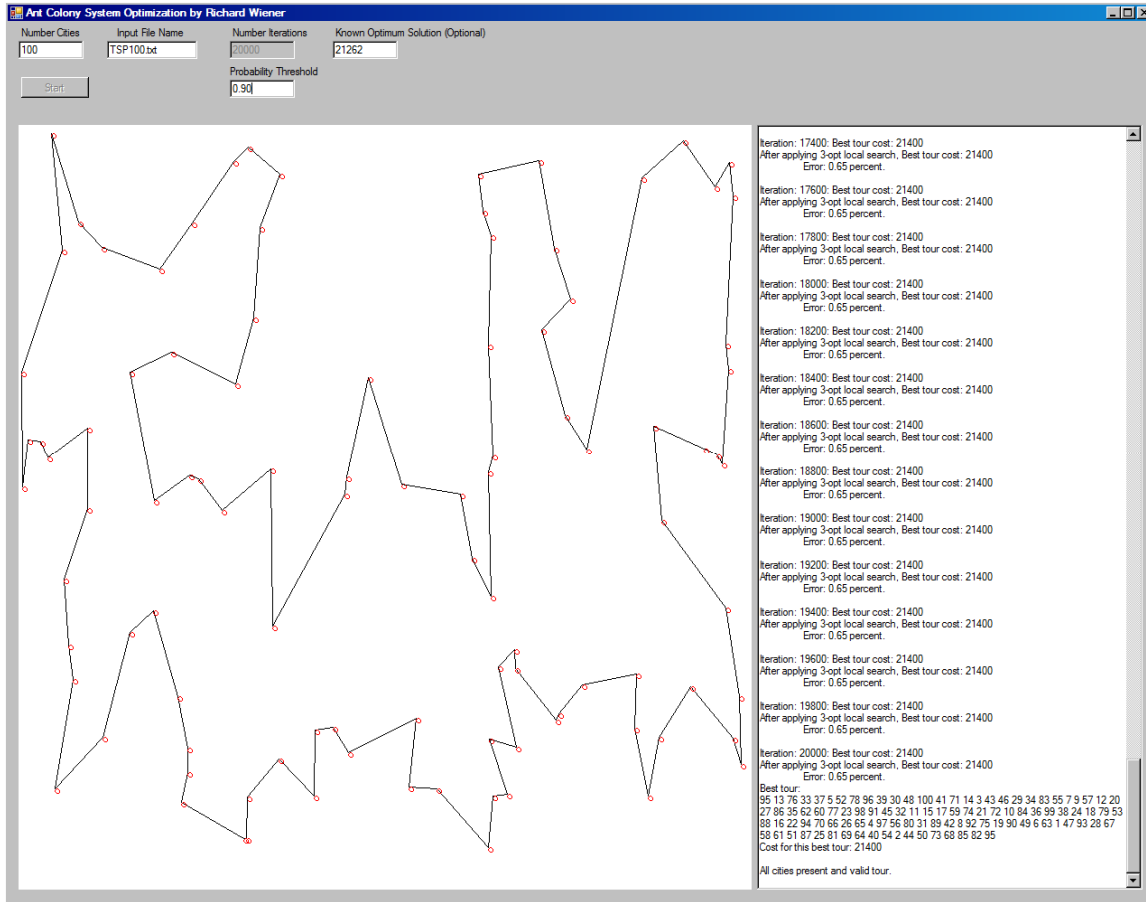
    int[] next = new int[numberCities + 2];
    int[] temp = new int[numberCities + 2];
    int z = 0;
    next[z++] = 0;
    next[z++] = 1;
    nextString += 1 + " ";
    next[z++] = xb[4];
    nextString += xb[4] + " ";
    for (int j = 1; j <= origb[5] - origb[4]; j++) {
        next[z++] = xb[4] + j;
        nextString += (xb[4] + j) + " ";
    }
    next[z++] = xb[3];
    nextString += xb[3] + " ";
    for (int j = 1; j <= origb[3] - origb[2]; j++) {
        next[z++] = xb[3] - j;
        nextString += (xb[3] - j) + " ";
    }
    next[z++] = xb[6];
    nextString += xb[6] + " ";
    for (int j = 1;
        j <= numberCities - origb[6]; j++) {
        next[z++] = xb[6] + j;
        nextString += (xb[6] + j) + " ";
    }
    next[z++] = 1;
    nextString += "1 ";

    for (int index = 1;
        index <= numberCities + 1; index++) {
        temp[index] = bestTour[next[index]];
    }

    for (int index = 1;
        index <= numberCities + 1; index++) {
        bestTour[index] = temp[index];
    }

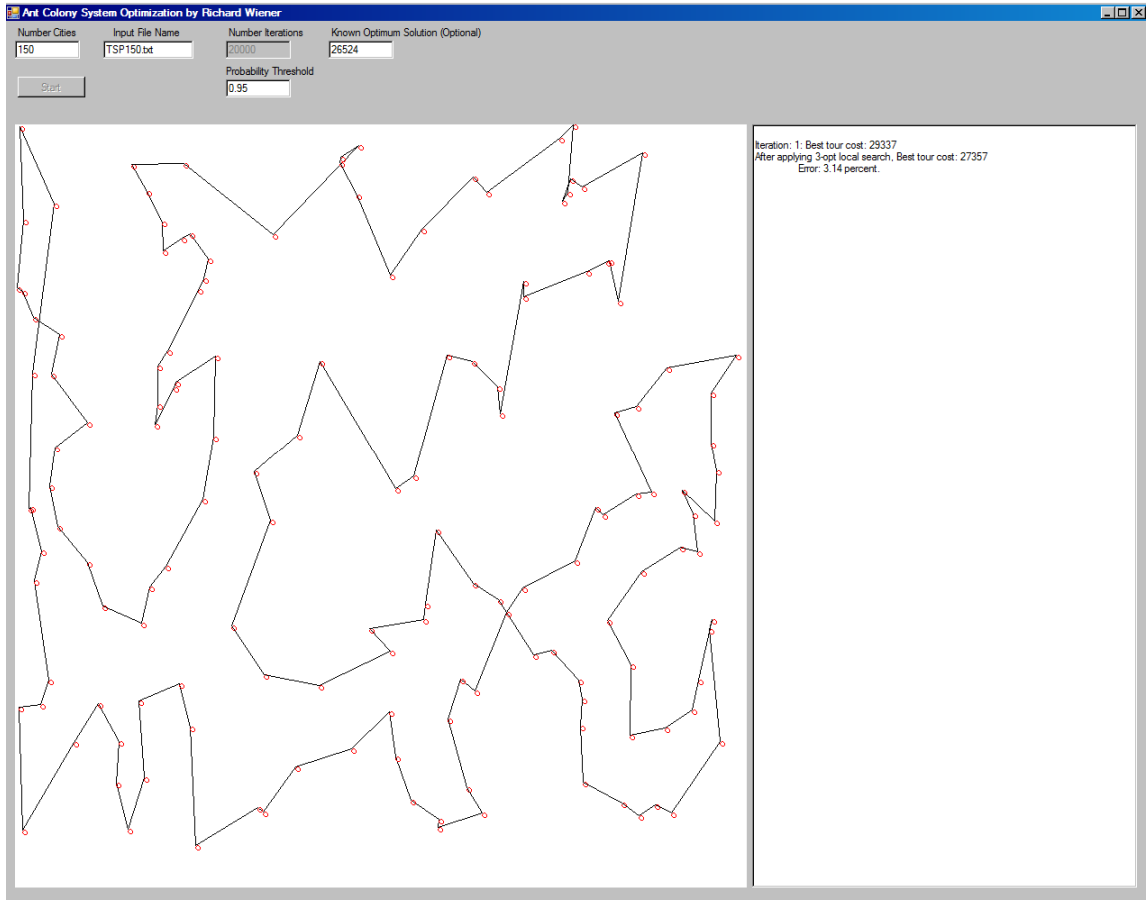
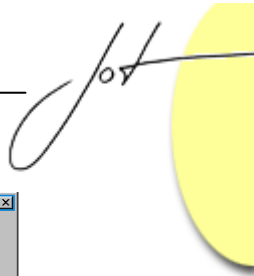
    // Test to see whether best tour is valid
}
```

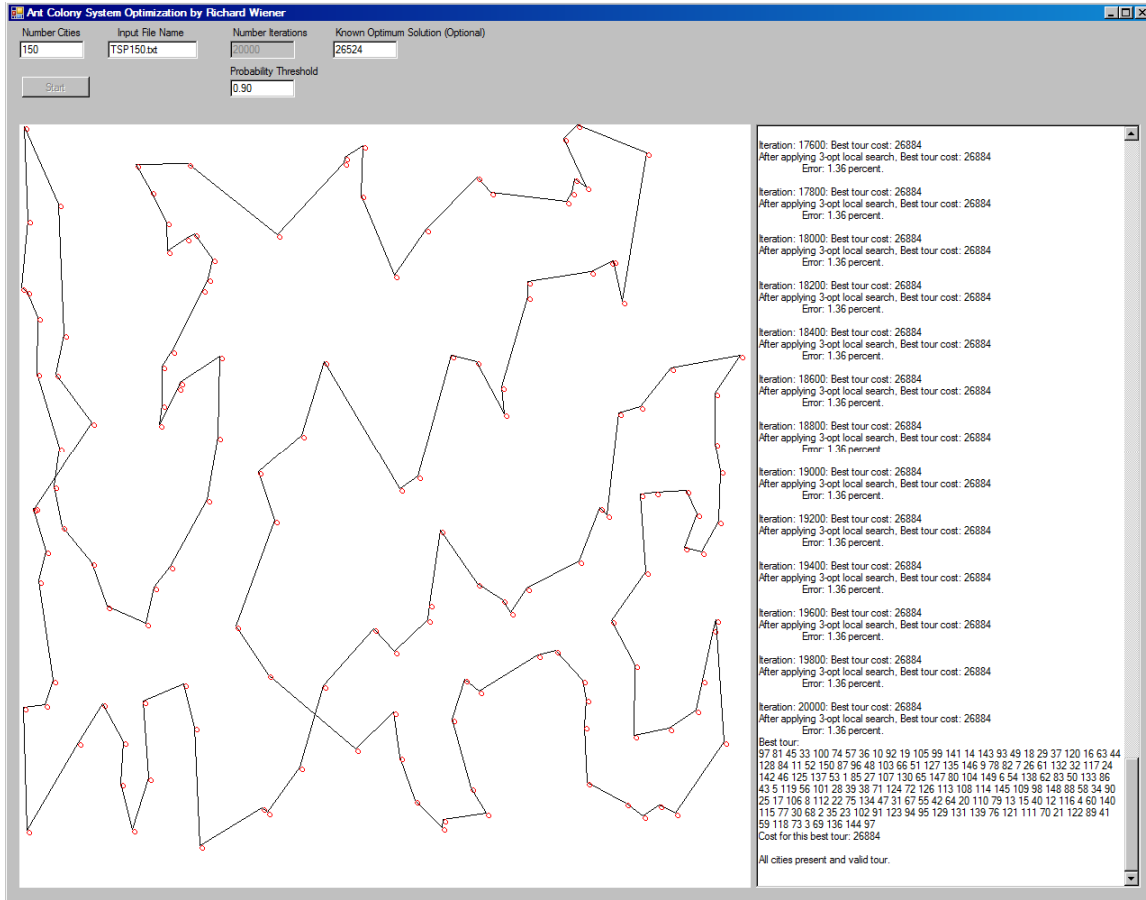
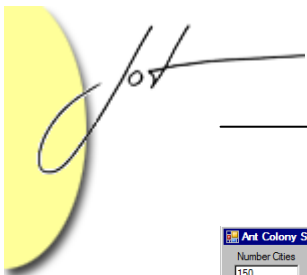


For the 100 city problem the solution, after 20,000 iterations, is within 0.65 percent of optimum. Once again lines do not cross in this near optimum solution.

For the 150 city problem, the initial solution is in error by 3.14 percent.



After 20,000 iterations, the solution is within 1.36 percent of the optimum. The output is:



The Ant Colony System produces results that are superior to those obtained by this author using either simulated annealing or genetic programming. What is particularly attractive is the small number of parameters that need to be tuned, especially compared to simulated annealing.

About the author



Richard Wiener is Chair of Computer Science at the University of Colorado at Colorado Springs. He is also the Editor-in-Chief of JOT and former Editor-in-Chief of the Journal of Object Oriented Programming. In addition to University work, Dr. Wiener has authored or co-authored 22 books and works actively as a consultant and software contractor whenever the possibility arises. His latest book, published by Thomson, Course Technology in April 2006, is entitled *Modern Software Development Using*

C#/.NET.