

# Ant Colony Optimization as a Solution for the Traveling Salesman Problem

<b>Subject</b>	Design and Analysis of Algorithms Lab
<b>Paper Code</b>	CS 591
<b>Team Members</b>	Ronit Ray (19), Abhirup Patra (80)
<b>Stream</b>	Computer Science and Engineering
<b>Section</b>	CSE-3A
<b>Semester</b>	Fifth
<b>Year</b>	2017



University of Engineering and Management, Kolkata

# FOREWORD

A project differs from coursework in the fact that it presents problems with real-life applications that must be solved intuitively and provides avenues to really think, learn new concepts, and then apply them. One is incomplete without the other and we cannot begin to express our gratitude to the professors who were there during our journey to build our foundations and work our way up into a practically applicable project.

We also realize that the initial problem statement required a solution to the Single Source Shortest Path problem and not the Traveling Salesman Problem, but were not able to devise a working algorithm in time to be able to demonstrate the same. The TSP solution is a widely discussed implementation of ACO, and it was hence relatively less challenging to pick up its concepts and implement a solution. Even so, we were unsure of being able to finish it within the submission deadline and cut it very, very close. We immensely regret being unable to solve the SSSP problem and hope not only that the TSP solution submitted herewith is acceptable as a project, but also that we will receive guidance in the future if and when we pick up the SSSP problem as a research topic instead, with more comfortable deadlines and scope for brainstorming.

Our thanks go out to Prof. Sankhadeep Chatterjee and Prof. Pallavi Saha for their support and assistance, and also to Prof. Ratan K. Basak, Prof. Bipasha Mukherjee and Prof. Kartik Sau for their guidance in the classroom.

- Ronit Ray, Abhirup Patra, October 2017.

# STATEMENT AND OBJECTIVE

*Ant Colony Optimization (ACO) is a global optimization algorithm, inspired by the social behavior of ants. ACO can be efficiently used to solve graph based optimization problems. Explore ACO and implement an ACO-based solution to the Traveling Salesman Problem. The problem is as follows: Given a list of cities and the distances between each pair of cities, find the shortest possible tour (a route that visits each city exactly once and returns to the origin city). Implement a GUI for inputting the parameters of ACO and then displaying the result.*

In this project, we are expected to implement a variation of the Ant Colony System algorithm first proposed by Dorigo in 1992 and solve the Traveling Salesman Problem which is a very popular graph optimization problem. The problem is NP-complete which means not only does it have no single solution, but it also has a number of different possible solutions, of which an optimal solution must be reached by various algorithms, and this can only be done in nondeterministic polynomial time. Furthermore, the project will have to be implemented with a GUI such that the user is allowed to input the parameters for the algorithm as per their liking and will also be able to see the output in a visually appealing and easily understandable way.

# THEORETICAL BACKGROUND

When humans reached a standstill in devising mathematical algorithms for computation of various problems, they began looking at the most complex system they know; nature itself. There are a number of different phenomena in nature that are incredibly complex but occur as if automated, and these can be integrated into algorithms to solve problems which were much tougher or even impossible to solve otherwise. This movement spawned the birth of Natural or Bio-Inspired Computing, Genetic Algorithms, Swarm Intelligence and many, many more.

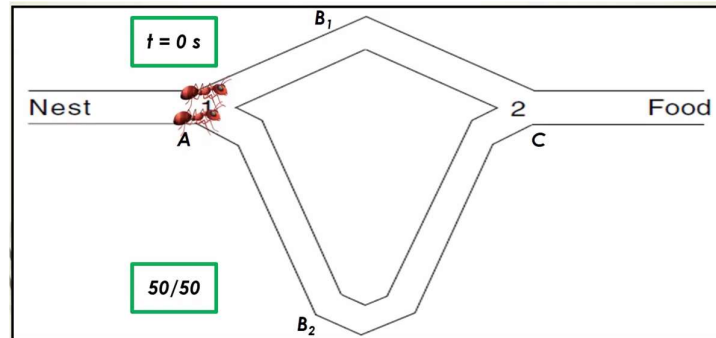
Ants are very small organisms, but are among the most industrious and organized creatures in the whole world. Their communication techniques, planning, and behavior in a number of situations can be studied and modeled to devise new algorithms to solve a variety of problems, and this is the base of Ant Colony Optimization.

**Swarm Intelligence** is a problem solving approach inspired by social behaviors of animals and interests. It is defined as “the collective behavior of decentralized, self-organized systems, natural or artificial,” and finds application in a number of areas, most recently artificial intelligence.

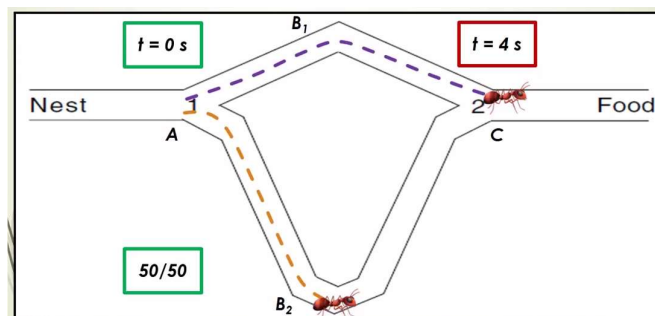
**Stigmergy** as defined by entomologist P. Grasse is the communication system employed by ants or termites that triggers actions as responses to significant chemically-induced stimuli. He went on to say that “workers are stimulated by performance they have achieved.” It is a local and indirect form of communication that depends on the modification of the environment rather than direct exchange of messages. This is done by leaving a trail of chemicals known as pheromones which are bioreceptors that can be detected by other ants.

**Ant Colony Optimization** is a method that has been suggested since the early nineties but was first formally proposed and put forward in a thesis by Belgian researcher Marco Dorigo and Luca Maria Gambardella in 1992, *Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem* and followed up by Dorigo, Birattari, and Stutzle’s thesis in 2006, *Ant Colony Optimization: Artificial Ants as a Computational Intelligence Technique*. To understand how it works, we must go back to a problem known as the Double Bridge Experiment.

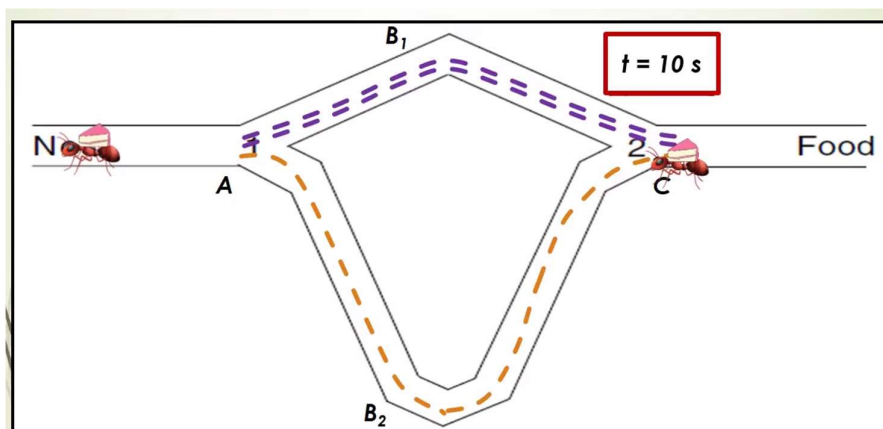
**The Double Bridge Experiment** was the study of the pheromone trail laying and following patterns of certain ant species. The problem statement was thus: a group of ants set out to look for food. They soon reach a spot where the road forks into two bridges, both of which will eventually lead to the food. One bridge is longer than the other. We must now observe their behavior and understand how the ant colony works.



As in the picture, at time  $t=0$ , let us assume two ants are at the crossroads. Since they have no information about either bridge, the probability of choosing between them is 50-50. And so one ant picks  $B_1$  and the other,  $B_2$ . Both ants leave a pheromone trail as they travel along the path they have adopted.



Since the first bridge is shorter, the first ant reaches its destination before the second. On its return, it sees that there is a trail already on  $B_1$  and hence is likely to favour  $B_1$  over  $B_2$ . It may so happen that it returns to the source before the second has even begun its return trip. By now, the first ant has left another pheromone trail on its adopted path. So when it sets out again, it notices a single trail on  $B_2$  and a double trail on  $B_1$ , which tips the probability of path selection from 50-50 to a figure more skewed towards  $B_1$ .



Now when the second ant tries to return, it too will notice a double trail on  $B_1$  and a single on  $B_2$  so it is likely to choose  $B_1$  for its return, thus making a triple-strength trail on  $B_1$ . This further makes  $B_1$  more attractive to ants which reach the fork in the future. This process is

repeated a number of times till it is observed that one of the paths is chosen by 100% or an overwhelming majority of ants, meaning the pheromone trail on it is thick enough to ensure that it is indeed the best choice to make and hence the optimum (shortest) path to and from the goal. Based on these findings, stochastic models of ant colonies were described by two parties, Goss et. al. 1989 and Bonabeau et al. 1997. The stochastic model also came up with a sophisticated equation to determine the probability for choosing a particular path, and based it on an importance factor called  $\alpha$ .

This model made for interesting reading and could then be applied to develop a system of “artificial” ants which are simulations meant to solve more complex problems than a simple two-bridge problem. It is worth noting that they aren’t using actual stigmergy, but simply exploiting certain elements of the behavior to create a heuristic to solve these problems.

**The Traveling Salesman Problem**, as discussed earlier asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science. It has a number of uses in vehicle routing, networking, resource allocation and many more. It has been extensively studied since the 1930s and has become the go-to foundation for optimization problems even in modern applications like logistics, planning, manufacture of electronic circuits and DNA sequencing. Several approaches have been taken towards solving the problem, ranging from Heuristic and Memetic approaches to Genetic algorithms and even modern neural networks, and of course, ant colony optimization.

#### **THE ANT COLONY ALGORITHM:**

The bulk of the ant colony optimization algorithm is made up of only a few steps. First, each ant in the colony constructs a solution based on previously deposited pheromone trails. Next ants will lay pheromone trails on the components of their chosen solution, depending on the solution's quality. In the example of the traveling salesman problem this would be the edges (or the paths between the cities). Finally, after all ants have finished constructing a solution and laying their pheromone trails, pheromone is evaporated from each component depending on the pheromone evaporation rate. This is like

1. procedure ACO
2.   while(running)
3.     generateSolutions()
4.     daemonActions()
5.     pheromoneUpdate()
6.   end while
7. end procedure

These steps are then run as many times as are needed to generate an adequate solution. There are some other factors:

**Randomness:** We must remember that as discussed in the two bridge problem earlier, convergence soon occurs due to one path being preferred more and more. But for the ants

to be able to look beyond the paths taken already and consider newer, better paths that were previously ignored or have recently been added, a degree of randomness must be introduced into the algorithm. This is where the randomness factor discussed later in the project comes in.

**State Selection:** In addition to this, a heuristic value is also computed and considered to guide the search process towards better solutions. This is typically based on the length of the edge between the city being considered - the shorter the edge, the more likely an ant will pick it, obviously.

Mathematically, this works out to

$$p_{ij}^k = \frac{[t_{ij}]^\alpha \cdot [n_{ij}]^\beta}{\sum_{l \in N_i^k} [t_{il}]^\alpha \cdot [n_{il}]^\beta}$$

This equation calculates the probability of selecting a single component of the solution. Here,  $t_{ij}$  denotes the amount of pheromone on a component between states  $i$  and  $j$ , and  $n_{ij}$  denotes its heuristic value.  $\alpha$  and  $\beta$  are both parameters used to control the importance of the pheromone trail and heuristic information during component selection.

This entire selection is done using a roulette-wheel process described as

```

1  rouletteWheel = 0
2  states = ant.getUnvisitedStates()
3  for newState in states do
4      rouletteWheel += pow(getPheromone(state, newState), alpha) * pow(calcHeuristicValue(state, newState), beta)
5
6  randomValue = random()
7  wheelPosition = 0
8
9  for newState in states do
10     wheelPosition += pow(getPheromone(state, newState), alpha) * pow(calcHeuristicValue(state, newState), beta)
11     if wheelPosition >= randomValue do
12         return newState

```

In the TSP, a node or city on the graph is a single state, and the next city is picked based on the distance as well as the amount of pheromone on the available path.

**Pheromone Update:** Similar to ants, the algorithm too will have to deposit pheromones over the traversed trails to ensure that they are preferred in the future as well. There are two parts to this, first being  $Q$ - the amount of pheromone deposited along each trail, a constant. The second is  $C^k$  or the cost of a solution.

The mathematical representation of this pheromone deposit process is

$$\Delta\tau_{ij}^k = \begin{cases} Q/C^k & \text{if component}(i, j) \text{ was used by ant} \\ 0 & \text{Otherwise} \end{cases}$$

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^m \Delta\tau_{ij}^k$$

**Evaporation:** A vital part of updation is also the evaporation of pheromones. Since they are only chemicals, they are not indelible and tend to fade with time. Pheromone evaporation makes the simulation more realistic and also allows for selection of newer paths, much like

induced randomness. For this, an evaporation factor is taken as a parameter, and the new value is simply the old value multiplied by  $(1 - \text{the evaporation factor})$ .

So far, we have discussed Dorigo's standard Ant Colony System (ACS). There are some popular variations to this, two of which are:

**Elitist ACO:** In elitist ACO systems, either the best current, or global best ant, deposits extra pheromone during its local pheromone update procedure. This encourages the colony to refine its search around solutions which have a track record of being high quality. If all goes well, this should result in better search performance.

**Max-Min ACO:** The MaxMin algorithm is similar to the elitist ACO algorithm in that it gives preference to high ranking solutions. However, in MaxMin instead of simply giving extra weight to elite solutions, only the best current or global solution is allowed to deposit a pheromone trail. Additionally, pheromone trails are kept between a maximum and minimum value, so that by having a limited range between the amount of pheromone on trails, premature convergence around sub-optimal solutions can be avoided. In many MaxMin implementations the best current ant is initially the ant which lays pheromone trails, then later, the algorithm switches so that the global best ant is the only ant which can lay a pheromone trail. This process helps encourage search across the entire search space initially, before eventually focusing in on the all-time best.

Having discussed the algorithm as a concept, we must now talk about implementation.

The best possible method to implement ACO is through an object-oriented language, such that each ant is simply an object of the Ant class. This way, its behavior is defined well and they can all be controlled very easily. The rest of the implementation involves the adjustment of parameters based on user preferences and the generation of a list of  $n$  nodes and a random  $n \times n$  adjacency matrix for the mapping. We then invoke the solution procedure as described in the algorithm above. This project was executed on Java and the program-specific structures and algorithms have been provided in the next few pages.



## THE ANT CLASS

1. trailSize: int
2. trail: int[]
3. visited: Boolean[]

### Constructor Ant(tourSize)

- Step 1. trailSize <- tourSize  
Step 2. Trail <- new int[tourSize]  
Step 3. Visited <- new int[tourSize]

### Void visitedCity(int currentIndex, int city)

- Step 1. Trail[currentIndex+1]=city      //add to trail  
Step 2. Visited[city]=true      //update flag

### Void visited(int i)

- Step 1. Return visited[i]

### Double trailLength(double graph[][])

- Step 1. Length <- graph[trail[trailSize - 1]][trail[0]]      //last node to initial  
Step 2. For all (i: 0 to trailSize -2)  
Step 3.      Length <- length+ graph[trail[i]][trail[i + 1]]  
Step 4. Return length

### Void clear()

- Step 1. For all (i: 0 to trailSize-1)  
Step 2.      Set visited[i]<- false

## THE AntColonyOptimization Class

NAME	TYPE	Description
c	double	number of trails
alpha	double	pheromone importance
beta	double	distance priority
evaporation	double	evaporation factor
q	double	pheromone left on trail per ant
antFactor	double	no. of ants per node
randomFactor	double	induced randomness
maxIterations	int	no. of iterations
numberOfCities	int	size of graph
numberOfAnts	int	ants
graph	int[][]	graph
trails	int[][]	different trails
ants	ArrayList<Ant>	all ants created
random	java.util.Random object	to generate random numbers
probabilities	int[]	stores probability for being next visited node
currentIndex	int	present node
bestTourOrder	int[]	final best tour
bestTourLength	int	length of best tour

### Constructor AntColonyOptimization:

1. sets parameters
2. generates random adjacency (weighted) matrix for graph
3. creates ants

### double [][]generateRandomMatrix(n):

Step 1. randomMatrix <- [][]

Step 2. for(i: 0 to n)

Step 3. for(j: 0 to n)

Step 4.           if(i==j)

Step 5.           then randomMatrix[i][j] <- 0

                  else

Step 6.                       randomMatrix[i][j] <- random value within 1-100

Step 7. display randomMatrix

                  //calculate naive cycle 0-1-2-3-.....-n-0

Step 8. sum <- 0

Step 9. for(i:0 to n-1)

Step 10.           sum <- sum+ randomMatrix[i][i+1]

Step 11. sum <- sum+ randomMatrix[n-1][0]

Step 12. display sum

Step 13. return randomMatrix

**void startAntOptimization()**

Step 1. for(i: 1 to 5)  
Step 2.           solve()

**int[] solve()**

Step 1. setupAnts()  
Step 2. clearTrails()  
Step 3. for(i:0 to maxIterations-1)  
Step 4.           moveAnts()  
Step 5.           updateTrails()  
Step 6.           updateBest()  
Step 7. print best tour and its length

**void setupAnts()**

Step 1. for(i:0 to numberOfAnts-1)  
Step 2.           for all ants  
Step 3.                   ant.clear  
Step 4.                   ant.visitCity(-1, next city selected randomly)  
Step 5. currentIndex <- 0

**void moveAnts()**

Step 1. for(i: currentIndex:numberOfCities-2)  
Step 2.           for all ants  
Step 3.           ant.visitCity(currentIndex, selectNextCity(ant))  
Step 4. currentIndex++

**void selectNextCity()**

Step 1. t <- random city other than current one.  
Step 2. r <- randomly generated double  
Step 3. if(r< randomFactor) then  
Step 4.           find the city corresponding to t in the list of cities and return it if unvisited.  
Step 5. calculateProbabilities(ant)  
Step 6. r <- random double  
Step 7. total <- 0  
Step 8. for(i:0 to numberOfCities-1) |  
Step 9. total <- total+ probabilities[i]  
Step 10.          if(total>r) then  
Step 11.                  return i  
Step 12. prompt "no other cities"

**void calculateProbabilities()**

Step 1.  $i \leftarrow \text{ant.trail}[\text{currentIndex}]$

Step 2.  $\text{pheromone} \leftarrow 0.0$

Step 3. for( $l$ : 0 to  $\text{numberOfCities}-1$ )

Step 4.           if( $\text{ant}$  has not visited  $l$ )

Step 5.                            $\text{pheromone} \leftarrow \text{trails}[i][l]^\alpha * \left( \frac{1}{\text{graph}[i][l]} \right)^\beta$

Step 6. for( $j$ :0 to  $\text{numberOfCities}-1$ )

Step 7. if( $\text{ant}$  has visited  $j$  already) then

Step 8.            $\text{probabilities}[j] \leftarrow 0$

          else

Step 9.                            $\text{probabilities}[j] \leftarrow \frac{\text{trails}[i][j]^\alpha * \left( \frac{1}{\text{graph}[i][j]} \right)^\beta}{\text{pheromone}}$

**void updateTrails()**

Step 1. for ( $i$ : 0 to  $\text{numberOfCities}-1$ )

Step 2.           for( $j$ : 0 to  $\text{numberOfCities}-1$ )

Step 3.                            $\text{trails}[i][j] \leftarrow \text{evaporation}$

Step 4. for all ants

Step 5.                            $\text{contribution} \leftarrow \frac{Q}{a.\text{trailLength}(\text{graph})}$

Step 6.           for ( $i$ : 0 to  $\text{numberOfCities}-2$ )

Step 7.                            $\text{trails}[a.\text{trail}[i]][a.\text{trail}[i+1]] += \text{contribution}$

Step 8.  $\text{trails}[a.\text{trail}[\text{numberOfCities}-1]][a.\text{trail}[0]] += \text{contribution}$

**void updateBest()**

Step 1. if best tour order does not exist then

Step 2.  $\text{bestTourOrder} \leftarrow \text{ants.get}(0).\text{trail}$

Step 3.            $\text{bestTourLength} \leftarrow \text{ants.get}(0).\text{trailLength}(\text{graph})$

Step 4. for all ants

Step 5.           if ( $a.\text{trailLength}(\text{graph}) < \text{bestTourLength}$ )

Step 6.                            $\text{bestTourLength} = a.\text{trailLength}(\text{graph})$

Step 7.                            $\text{bestTourOrder} = a.\text{trail.clone}()$

**void clearTrails()**

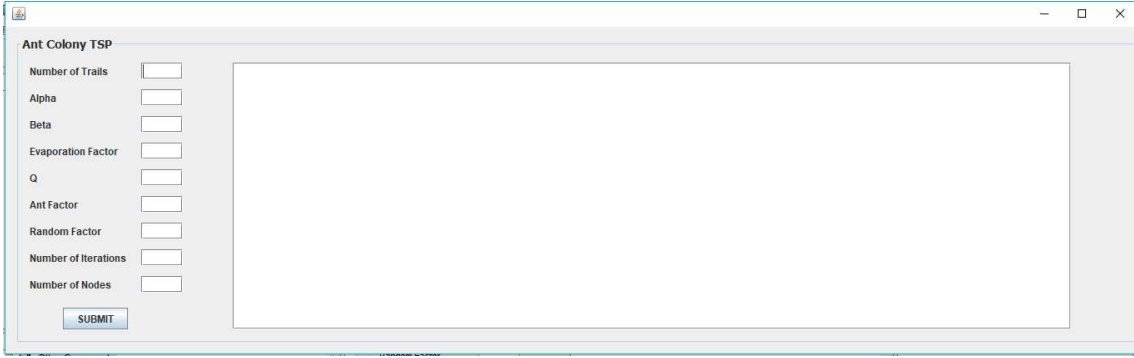
Step 1. for( $i$ : 0 to  $\text{numberOfCities}-1$ )

Step 2.           for( $j$ : 0 to  $\text{numberOfCities}-1$ )

Step 3.                            $\text{trails}[i][j] \leftarrow c$

## The Launcher Class

This class was created as a Java Swing GUI application with a form to input the values of all the parameters and then initialize them by creating an object of the AntColonyOptimization class with a parameterized constructor. There is also a text panel to hold the results of execution. The design is below:

The image shows a Java Swing window titled "Ant Colony TSP". On the left side, there is a vertical list of input parameters, each followed by a text input field: "Number of Trails", "Alpha", "Beta", "Evaporation Factor", "Q", "Ant Factor", "Random Factor", "Number of Iterations", and "Number of Nodes". Below these fields is a "SUBMIT" button. To the right of the input fields is a large, empty rectangular text area for displaying results. The window has standard OS window controls (minimize, maximize, close) in the top right corner.

The next few pages will hold the source code for the entire project. For convenience, the code has also been uploaded to [github.com/RonitRay/](https://github.com/RonitRay/)

```
1 public class Ant
2 {
3     protected int trailSize;
4     protected int trail[];
5     protected boolean visited[];
6
7     public Ant(int tourSize)
8     {
9         this.trailSize = tourSize;
10        this.trail = new int[tourSize];
11        this.visited = new boolean[tourSize];
12    }
13
14    protected void visitCity(int currentIndex, int city)
15    {
16        trail[currentIndex + 1] = city; //add to trail
17        visited[city] = true;           //update flag
18    }
19
20    protected boolean visited(int i)
21    {
22        return visited[i];
23    }
24
25    protected double trailLength(double graph[][])
26    {
27        double length = graph[trail[trailSize - 1]][trail[0]];
28        for (int i = 0; i < trailSize - 1; i++)
29            length += graph[trail[i]][trail[i + 1]];
30        return length;
31    }
32
33    protected void clear()
34    {
35        for (int i = 0; i < trailSize; i++)
36            visited[i] = false;
37    }
38 }
```

```

1 import java.util.ArrayList;
2 import java.util.Arrays;
3 import java.util.List;
4 import java.util.OptionalInt;
5 import java.util.Random;
6 import java.util.stream.IntStream;
7
8 /*
9  * default
10 * private double c = 1.0;           //number of trails
11 * private double alpha = 1;        //pheromone importance
12 * private double beta = 5;         //distance priority
13 * private double evaporation = 0.5;
14 * private double Q = 500;          //pheromone left on trail per ant
15 * private double antFactor = 0.8;  //no of ants per node
16 * private double randomFactor = 0.01; //introducing randomness
17 * private int maxIterations = 1000;
18 */
19
20 public class AntColonyOptimization
21 {
22     public String s="";
23     private double c;           //number of trails
24     private double alpha;       //pheromone importance
25     private double beta;        //distance priority
26     private double evaporation;
27     private double Q;           //pheromone left on trail per ant
28     private double antFactor;    //no of ants per node
29     private double randomFactor; //introducing randomness
30
31     private int maxIterations;
32
33     private int numberOfCities;
34     private int numberOfAnts;
35     private double graph[][];
36     private double trails[][];
37     private List<Ant> ants = new ArrayList<>();
38     private Random random = new Random();
39     private double probabilities[];
40
41     private int currentIndex;
42
43     private int[] bestTourOrder;
44     private double bestTourLength;
45
46     public AntColonyOptimization(double tr, double al, double be, double ev,
47         double q, double af, double rf, int iter, int noOfCities)
48     {
49         c=tr; alpha=al; beta=be; evaporation=ev; Q=q; antFactor=af; randomFactor=rf; maxIterations=iter;
50
51         graph = generateRandomMatrix(noOfCities);
52         numberOfCities = noOfCities;
53         numberOfAnts = (int) (numberOfCities * antFactor);
54
55         trails = new double[numberOfCities][numberOfCities];
56         probabilities = new double[numberOfCities];
57
58         for(int i=0;i<numberOfAnts;i++)
59             ants.add(new Ant(numberOfCities));
60     }
61
62     /**
63      * Generate initial solution

```

```

64      */
65      public double[][] generateRandomMatrix(int n)
66      {
67          double[][] randomMatrix = new double[n][n];
68
69          for(int i=0;i<n;i++)
70          {
71              for(int j=0;j<n;j++)
72              {
73                  if(i==j)
74                      randomMatrix[i][j]=0;
75                  else
76                      randomMatrix[i][j]=Math.abs(random.nextInt(100)+1);
77              }
78          }
79
80          s+="\t";
81          for(int i=0;i<n;i++)
82              s+=(i+"\t");
83          s+="\n";
84
85          for(int i=0;i<n;i++)
86          {
87              s+=(i+"\t");
88              for(int j=0;j<n;j++)
89
90                  s+=(randomMatrix[i][j]+"");
91              s+="\n";
92          }
93
94          int sum=0;
95
96          for(int i=0;i<n-1;i++)
97              sum+=randomMatrix[i][i+1];
98          sum+=randomMatrix[n-1][0];
99          s+="\nNaive solution 0-1-2-...-n-0 = "+sum+"\n";
100         return randomMatrix;
101     }
102
103     /**
104      * Perform ant optimization
105      */
106     public void startAntOptimization()
107     {
108         for(int i=1;i<=5;i++)
109         {
110             s+="\nAttempt #" + i;
111             solve();
112             s+="\n";
113         }
114     }
115
116     /**
117      * Use this method to run the main logic
118      */
119     public int[] solve()
120     {
121         setupAnts();
122         clearTrails();
123         for(int i=0;i<maxIterations;i++)
124         {
125             moveAnts();
126             updateTrails();
127             updateBest();

```



```
128         s+="\nBest tour length: " + (bestTourLength - numberOfCities));
129         s+="\nBest tour order: " + Arrays.toString(bestTourOrder));
130         return bestTourOrder.clone();
131     }
132
133     /**
134      * Prepare ants for the simulation
135      */
136     private void setupAnts()
137     {
138         for(int i=0;i<numberOfAnts;i++)
139         {
140             for(Ant ant:ants)
141             {
142                 ant.clear();
143                 ant.visitCity(-1, random.nextInt(numberOfCities));
144             }
145         }
146         currentIndex = 0;
147     }
148
149     /**
150      * At each iteration, move ants
151      */
152     private void moveAnts()
153     {
154         for(int i=currentIndex;i<numberOfCities-1;i++)
155         {
156             for(Ant ant:ants)
157             {
158                 ant.visitCity(currentIndex,selectNextCity(ant));
159             }
160             currentIndex++;
161         }
162     }
163
164     /**
165      * Select next city for each ant
166      */
167     private int selectNextCity(Ant ant)
168     {
169         int t = random.nextInt(numberOfCities - currentIndex);
170         if (random.nextDouble() < randomFactor)
171         {
172             int cityIndex=-999;
173             for(int i=0;i<numberOfCities;i++)
174             {
175                 if(i==t && !ant.visited(i))
176                 {
177                     cityIndex=i;
178                     break;
179                 }
180             }
181             if(cityIndex!=-999)
182                 return cityIndex;
183         }
184         calculateProbabilities(ant);
185         double r = random.nextDouble();
186         double total = 0;
187         for (int i = 0; i < numberOfCities; i++)
188         {
189             total += probabilities[i];
```

```

190         if (total >= r)
191             return i;
192     }
193     throw new RuntimeException("There are no other cities");
194 }
195
196 /**
197  * Calculate the next city picks probabilities
198  */
199 public void calculateProbabilities(Ant ant)
200 {
201     int i = ant.trail[currentIndex];
202     double pheromone = 0.0;
203     for (int l = 0; l < numberOfCities; l++)
204     {
205         if (!ant.visited(l))
206             pheromone += Math.pow(trails[i][l], alpha) * Math.pow(1.0 / graph[i][l], beta);
207     }
208     for (int j = 0; j < numberOfCities; j++)
209     {
210         if (ant.visited(j))
211             probabilities[j] = 0.0;
212         else
213         {
214             double numerator = Math.pow(trails[i][j], alpha) * Math.pow(1.0 / graph[i][j], beta);
215             probabilities[j] = numerator / pheromone;
216         }
217     }
218 }
219
220 /**
221  * Update trails that ants used
222  */
223 private void updateTrails()
224 {
225     for (int i = 0; i < numberOfCities; i++)
226     {
227         for (int j = 0; j < numberOfCities; j++)
228             trails[i][j] *= evaporation;
229     }
230     for (Ant a : ants)
231     {
232         double contribution = Q / a.trailLength(graph);
233         for (int i = 0; i < numberOfCities - 1; i++)
234             trails[a.trail[i]][a.trail[i + 1]] += contribution;
235         trails[a.trail[numberOfCities - 1]][a.trail[0]] += contribution;
236     }
237 }
238
239 /**
240  * Update the best solution
241  */
242 private void updateBest()
243 {
244     if (bestTourOrder == null)
245     {
246         bestTourOrder = ants.get(0).trail;
247         bestTourLength = ants.get(0).trailLength(graph);
248     }
249     for (Ant a : ants)
250     {
251         if (a.trailLength(graph) < bestTourLength)
252             //

```

```
254         bestTourLength = a.trailLength(graph);
255         bestTourOrder = a.trail.clone();
256     }
257 }
258 }
259
260 /**
261  * Clear trails after simulation
262  */
263 private void clearTrails()
264 {
265     for(int i=0;i<numberOfCities;i++)
266     {
267         for(int j=0;j<numberOfCities;j++)
268             trails[i][j]=c;
269     }
270 }
271 }
```

```

1
2 public class Launcher extends javax.swing.JFrame {
3
4     public Launcher() {
5         initComponents();
6     }
7     @SuppressWarnings("unchecked")
8     // <editor-fold defaultstate="collapsed" desc="Generated
      Code">
9     private void initComponents() {
10
11         jPanel1 = new javax.swing.JPanel();
12         jLabel1 = new javax.swing.JLabel();
13         jLabel2 = new javax.swing.JLabel();
14         jLabel3 = new javax.swing.JLabel();
15         jLabel4 = new javax.swing.JLabel();
16         jLabel5 = new javax.swing.JLabel();
17         jLabel6 = new javax.swing.JLabel();
18         jLabel7 = new javax.swing.JLabel();
19         jLabel8 = new javax.swing.JLabel();
20         jLabel9 = new javax.swing.JLabel();
21         tr = new javax.swing.JTextField();
22         al = new javax.swing.JTextField();
23         be = new javax.swing.JTextField();
24         ef = new javax.swing.JTextField();
25         q = new javax.swing.JTextField();
26         af = new javax.swing.JTextField();
27         rf = new javax.swing.JTextField();
28         iter = new javax.swing.JTextField();
29         numberOfCities = new javax.swing.JTextField();
30         submit = new javax.swing.JButton();
31         jScrollPane1 = new javax.swing.JScrollPane();
32         jTextArea1 = new javax.swing.JTextArea();
33
34         setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
35
36         jPanel1.setBorder(javax.swing.BorderFactory.createTitledBorder(null, "Ant
      Colony TSP", javax.swing.border.TitledBorder.DEFAULT_JUSTIFICATION,
      javax.swing.border.TitledBorder.DEFAULT_POSITION, new
      java.awt.Font("Tahoma", 1, 14))); // NOI18N
37
38         jLabel1.setText("Number of Trails");
39
40         jLabel2.setText("Alpha");
41
42         jLabel3.setText("Beta");
43
44         jLabel4.setText("Evaporation Factor");
45
46         jLabel5.setText("Q");
47
48         jLabel6.setText("Ant Factor");
49
50         jLabel7.setText("Random Factor");
51
52         jLabel8.setText("Number of Iterations");
53
54         jLabel9.setText("Number of Nodes");
55
56         tr.setMinimumSize(new java.awt.Dimension(6, 50));
57         al.setMinimumSize(new java.awt.Dimension(6, 50));
58         be.setMinimumSize(new java.awt.Dimension(6, 50));
59         ef.setMinimumSize(new java.awt.Dimension(6, 50));
60         q.setMinimumSize(new java.awt.Dimension(6, 50));
61         af.setMinimumSize(new java.awt.Dimension(6, 50));
62         rf.setMinimumSize(new java.awt.Dimension(6, 50));
63
64
65
66
67
68
69

```

```

70         iter.setMinimumSize(new java.awt.Dimension(6, 50));
71
72         numberOfCities.setMinimumSize(new java.awt.Dimension(6, 50));
73
74         submit.setText("SUBMIT");
75         submit.addActionListener(new java.awt.event.ActionListener() {
76             public void actionPerformed(java.awt.event.ActionEvent evt) {
77                 submitActionPerformed(evt);
78             }
79         });
80
81         jTextArea1.setEditable(false);
82         jTextArea1.setColumns(20);
83         jTextArea1.setRows(5);
84         jScrollPane1.setViewportView(jTextArea1);
85
86         javax.swing.GroupLayout jPanel1Layout = new javax.swing.GroupLayout(jPanel1);
87         jPanel1.setLayout(jPanel1Layout);
88         jPanel1Layout.setHorizontalGroup(
89
90             jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
91                 .addGroup(jPanel1Layout.createSequentialGroup()
92                     .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
93                         .addGroup(jPanel1Layout.createSequentialGroup()
94                             .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
95                                 .addComponent(jLabel1)
96                                 .addComponent(jLabel2)
97                                 .addComponent(jLabel3)
98                                 .addComponent(jLabel4)
99                                 .addComponent(jLabel5)
100                                .addComponent(jLabel6)
101                                .addComponent(jLabel7)
102                                .addComponent(jLabel8)
103                                .addComponent(jLabel9))
104                             .addGap(18, 18, 18)
105
106                             .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
107                                 .addComponent(numberOfCities,
108                                     javax.swing.GroupLayout.DEFAULT_SIZE,
109                                     Short.MAX_VALUE)
110                                 .addComponent(iter,
111                                     javax.swing.GroupLayout.DEFAULT_SIZE,
112                                     Short.MAX_VALUE)
113                                 .addComponent(rf, javax.swing.GroupLayout.DEFAULT_SIZE,
114                                     Short.MAX_VALUE)
115                                 .addComponent(af, javax.swing.GroupLayout.DEFAULT_SIZE,
116                                     Short.MAX_VALUE)
117                                 .addComponent(q, javax.swing.GroupLayout.DEFAULT_SIZE,
118                                     Short.MAX_VALUE)
119                                 .addComponent(ef, javax.swing.GroupLayout.DEFAULT_SIZE,
120                                     Short.MAX_VALUE)
121                                 .addComponent(tr,
122                                     javax.swing.GroupLayout.PREFERRED_SIZE, 50,
123                                     javax.swing.GroupLayout.PREFERRED_SIZE)
124                                 .addComponent(al, javax.swing.GroupLayout.DEFAULT_SIZE,
125                                     Short.MAX_VALUE)
126                                 .addComponent(be, javax.swing.GroupLayout.DEFAULT_SIZE,
127                                     Short.MAX_VALUE))
128                             .addGap(60, 60, 60))
129                         .addGroup(jPanel1Layout.createSequentialGroup()
130                             .addGap(52, 52, 52)
131                             .addComponent(submit)
132
133                             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
134                             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATIVE,
135                                     javax.swing.GroupLayout.DEFAULT_SIZE,
136                                     Short.MAX_VALUE)))
137                     .addContainerGap())
138             )
139         );

```

```

120         .addComponent(jScrollPane, javax.swing.GroupLayout.PREFERRED_SIZE,
121         1007, javax.swing.GroupLayout.PREFERRED_SIZE)
122         .addGap(78, 78, 78))
123     );
124     jPanel1Layout.setVerticalGroup(
125
126         jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
127         .addGroup(jPanel1Layout.createSequentialGroup())
128         .addContainerGap()
129
130         .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Al
131         ignment.LEADING)
132         .addGroup(jPanel1Layout.createSequentialGroup())
133
134             .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupL
135             ayout.Alignment.BASELINE)
136             .addComponent(jLabel1)
137             .addComponent(tr,
138             javax.swing.GroupLayout.PREFERRED_SIZE,
139             javax.swing.GroupLayout.DEFAULT_SIZE,
140             javax.swing.GroupLayout.PREFERRED_SIZE))
141
142             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UN
143             RELATED)
144
145             .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupL
146             ayout.Alignment.BASELINE)
147             .addComponent(jLabel2)
148             .addComponent(al,
149             javax.swing.GroupLayout.PREFERRED_SIZE,
150             javax.swing.GroupLayout.DEFAULT_SIZE,
151             javax.swing.GroupLayout.PREFERRED_SIZE))
152
153             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UN
154             RELATED)
155
156             .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupL
157             ayout.Alignment.BASELINE)
158             .addComponent(jLabel3)
159             .addComponent(be,
160             javax.swing.GroupLayout.PREFERRED_SIZE,
161             javax.swing.GroupLayout.DEFAULT_SIZE,
162             javax.swing.GroupLayout.PREFERRED_SIZE))
163
164             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UN
165             RELATED)
166
167             .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupL
168             ayout.Alignment.BASELINE)
169             .addComponent(jLabel4)
170             .addComponent(ef,
171             javax.swing.GroupLayout.PREFERRED_SIZE,
172             javax.swing.GroupLayout.DEFAULT_SIZE,
173             javax.swing.GroupLayout.PREFERRED_SIZE))
174
175             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UN
176             RELATED)
177
178             .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupL
179             ayout.Alignment.BASELINE)
180             .addComponent(jLabel5)
181             .addComponent(q, javax.swing.GroupLayout.PREFERRED_SIZE,
182             javax.swing.GroupLayout.DEFAULT_SIZE,
183             javax.swing.GroupLayout.PREFERRED_SIZE))
184
185             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UN
186             RELATED)
187
188             .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupL
189             ayout.Alignment.BASELINE)
190             .addComponent(jLabel6)
191             .addComponent(af,

```

```

        javax.swing.GroupLayout.PREFERRED_SIZE,
        javax.swing.GroupLayout.DEFAULT_SIZE,
        javax.swing.GroupLayout.PREFERRED_SIZE))
152
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UN
RELATED)
153
        .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupL
ayout.Alignment.BASELINE)
154            .addComponent(jLabel7)
155            .addComponent(rf,
                javax.swing.GroupLayout.PREFERRED_SIZE,
                javax.swing.GroupLayout.DEFAULT_SIZE,
                javax.swing.GroupLayout.PREFERRED_SIZE))
156
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UN
RELATED)
157
        .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupL
ayout.Alignment.BASELINE)
158            .addComponent(jLabel8)
159            .addComponent(iter,
                javax.swing.GroupLayout.PREFERRED_SIZE,
                javax.swing.GroupLayout.DEFAULT_SIZE,
                javax.swing.GroupLayout.PREFERRED_SIZE))
160
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UN
RELATED)
161
        .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupL
ayout.Alignment.BASELINE)
162            .addComponent(jLabel9)
163            .addComponent(numberOfCities,
                javax.swing.GroupLayout.PREFERRED_SIZE,
                javax.swing.GroupLayout.DEFAULT_SIZE,
                javax.swing.GroupLayout.PREFERRED_SIZE))
164        .addGap(18, 18, 18)
165        .addComponent(submit)
166        .addGap(0, 0, Short.MAX_VALUE))
167        .addComponent(jScrollPane1))
168        .addContainerGap())
169    );
170
171    javax.swing.GroupLayout layout = new
javax.swing.GroupLayout(getContentPane());
172    getContentPane().setLayout(layout);
173    layout.setHorizontalGroup(
174        layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
175        .addGroup(layout.createSequentialGroup()
176            .addContainerGap()
177            .addComponent(jPanel1, javax.swing.GroupLayout.DEFAULT_SIZE,
                javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
178            .addContainerGap())
179        );
180    layout.setVerticalGroup(
181        layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
182        .addGroup(layout.createSequentialGroup()
183            .addContainerGap()
184            .addComponent(jPanel1, javax.swing.GroupLayout.DEFAULT_SIZE,
                javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
185            .addContainerGap())
186        );
187
188    pack();
189    } // </editor-fold>
190
191    private void submitActionPerformed(java.awt.event.ActionEvent evt)
192    {
193        double t=Double.parseDouble(tr.getText());
194        double a=Double.parseDouble(al.getText());
195        double b=Double.parseDouble(be.getText());
196        double e=Double.parseDouble(ef.getText());
197        double qq=Double.parseDouble(q.getText());

```

```

197         double aaf=Double.parseDouble(af.getText());
198         double rrf=Double.parseDouble(rf.getText());
199         int it=Integer.parseInt(iter.getText());
200         int noc=Integer.parseInt(numberOfCities.getText());
201
202         AntColonyOptimization antTSP = new
203         AntColonyOptimization(t,a,b,e,qq,aaf,rrf,it,noc);
204         antTSP.startAntOptimization();
205         jTextArea1.setText(antTSP.s);
206     }
207
208     public static void main(String args[]) {
209         java.awt.EventQueue.invokeLater(new Runnable()
210         {
211             public void run() {
212                 new Launcher().setVisible(true);
213             }
214         });
215
216         // Variables declaration - do not modify
217         private javax.swing.JTextField af;
218         private javax.swing.JTextField al;
219         private javax.swing.JTextField be;
220         private javax.swing.JTextField ef;
221         private javax.swing.JTextField iter;
222         private javax.swing.JLabel jLabel1;
223         private javax.swing.JLabel jLabel2;
224         private javax.swing.JLabel jLabel3;
225         private javax.swing.JLabel jLabel4;
226         private javax.swing.JLabel jLabel5;
227         private javax.swing.JLabel jLabel6;
228         private javax.swing.JLabel jLabel7;
229         private javax.swing.JLabel jLabel8;
230         private javax.swing.JLabel jLabel9;
231         private javax.swing.JPanel jPanel1;
232         private javax.swing.JScrollPane jScrollPane1;
233         private javax.swing.JTextArea jTextArea1;
234         private javax.swing.JTextField numberOfCities;
235         private javax.swing.JTextField q;
236         private javax.swing.JTextField rf;
237         private javax.swing.JButton submit;
238         private javax.swing.JTextField tr;
239         // End of variables declaration
240     }

```



# RESULTS AND OUTPUT

Having coded the project, we must now observe and verify the output. Here are a few situations.

Ant Colony TSP

Number of Trails

1

Alpha

1

Beta

5

Evaporation Factor

0.5

Q

500

Ant Factor

0.8

Random Factor

0.01

Number of Iterations

1000

Number of Nodes

5

SUBMIT

	0	1	2	3	4
0	0.0	66.0	55.0	7.0	9.0
1	19.0	0.0	78.0	7.0	74.0
2	3.0	93.0	0.0	51.0	15.0
3	30.0	88.0	95.0	0.0	43.0
4	22.0	7.0	17.0	13.0	0.0

Naive solution 0-1-2-...-n-0 = 260

Attempt #1

Best tour length: 109.0

Best tour order: [4, 1, 3, 0, 2]

Attempt #2

Best tour length: 109.0

Best tour order: [1, 3, 0, 4, 2]

Attempt #3

Best tour length: 109.0

Best tour order: [3, 0, 4, 1, 2]

Attempt #4

Best tour length: 109.0

Best tour order: [0, 3, 4, 1, 2]

Attempt #5

Best tour length: 109.0

Best tour order: [0, 3, 1, 2, 4]

Ant Colony TSP

Number of Trails

1

Alpha

1

Beta

5

Evaporation Factor

0.5

Q

500

Ant Factor

0.8

Random Factor

0.01

Number of Iterations

1000

Number of Nodes

10

SUBMIT

	0	1	2	3	4	5	6	7	8	9
0	0.0	76.0	15.0	2.0	85.0	16.0	44.0	66.0	59.0	24.0
1	46.0	0.0	34.0	72.0	53.0	94.0	33.0	70.0	39.0	84.0
2	75.0	93.0	0.0	95.0	42.0	53.0	77.0	62.0	71.0	76.0
3	32.0	68.0	44.0	0.0	65.0	2.0	82.0	2.0	100.0	78.0
4	88.0	67.0	74.0	98.0	0.0	78.0	93.0	46.0	24.0	58.0
5	45.0	42.0	79.0	92.0	77.0	0.0	60.0	49.0	42.0	49.0
6	99.0	31.0	23.0	97.0	49.0	87.0	0.0	10.0	72.0	56.0
7	42.0	15.0	42.0	48.0	22.0	82.0	83.0	0.0	69.0	67.0
8	97.0	27.0	72.0	39.0	6.0	70.0	64.0	32.0	0.0	57.0
9	1.0	59.0	37.0	8.0	59.0	8.0	61.0	81.0	75.0	0.0

Naive solution 0-1-2-...-n-0 = 545

Attempt #1

Best tour length: 258.0

Best tour order: [2, 7, 1, 8, 4, 9, 0, 3, 5, 6]

Attempt #2

Best tour length: 258.0

Best tour order: [2, 7, 1, 8, 4, 9, 0, 3, 5, 6]

Attempt #3

Best tour length: 238.0

Best tour order: [0, 3, 5, 7, 1, 6, 2, 4, 8, 9]

Attempt #4

Best tour length: 225.0

Best tour order: [8, 4, 9, 0, 3, 7, 1, 6, 2, 5]

Attempt #5

Best tour length: 225.0

Best tour order: [8, 4, 9, 0, 3, 7, 1, 6, 2, 5]

In each situation we see our result is at least twice better than the naïve solution (a sequential walk from 0 to n and back to 0), which is a very desirable result. The results are accurate and the implementation successful.

# CONCLUSION

This project was tremendously challenging and research-intensive, certainly more than any other project we have been assigned in the recent past. We hope that we have implemented it well and delivered satisfactory results. That said, we realize a number of shortcomings that could have been addressed. For one, the interface could have been a lot more intuitive. The outputs are still in text form, and while a weighted adjacency matrix is very accurate, it is not the easiest thing to visualize. A GUI with a clickable interface to add nodes, a method to find the space between them on the canvas automatically and then display not only the best tour but also the working as well as the probability along the way would be the ideal solution. Such a system is already in place at <http://www.theprojectspot.com/tutorial-post/ant-colony-optimization-for-hackers/10>. Due to lack of graphics development expertise and time constraints, the development of such a visualization was not possible.

In the future, we hope to refine these results and also implement such a GUI system. We will also look into the solution for the Single Source Shortest Path problem that this project was originally intended to propose, and hope to be able to successfully develop the same and have it published.

As of today, Ant Colony Optimization is one of the better methods to arrive at solutions for the Traveling Salesman Problem.

# REFERENCES

1. *Ant Colony Optimization- Artificial Ants as a Computational Intelligence Technique*, Marco Dorigo, Mauro Birattari, and Thomas Stützle Université Libre de Bruxelles, BELGIUM
2. *Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem* Marco Dorigo and Luca Maria Gambardella
3. *Ant Colony Optimization: an Introduction*, by Vittorio Maniezzo, Luca Maria Gambardella, Fabio de Luigi
4. [http://www.scholarpedia.org/article/Ant\\_colony\\_optimization](http://www.scholarpedia.org/article/Ant_colony_optimization)
5. <http://www.theprojectspot.com/tutorial-post/ant-colony-optimization-for-hackers/10>
6. <http://www.baeldung.com/java-ant-colony-optimization>
7. [https://en.wikipedia.org/wiki/Ant\\_colony\\_optimization\\_algorithms](https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms)
8. P. P. Grasse, *Les Insectes Dans Leur Univers*. Paris, France: Ed. du Palais de la découverte, 1946.
9. J.-L. Deneubourg, S. Aron, S. Goss, and J.-M. Pasteels, "The selforganizing exploratory pattern of the Argentine ant," *Journal of Insect Behavior*, vol. 3, p. 159, 1990
10. M. Dorigo, V. Maniezzo, and A. Coloni, "Ant System: Optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, vol. 26, no. 1, pp. 29–41, 1996