Robin Weiss
12.8.2008
COMP221

Ant Colony Optimization for The Traveling Salesman Problem

Introduction

Swarm Intelligence describes the tendency for highly organized behaviors to emerge from the individual actions of a group of independent agents. In nature, groups of social creatures often exhibit complex behaviors that surpass the capabilities of any one of the constituent individuals. This emergent property of swarms is the inspiration for swarm-based algorithms–computer algorithms that are modeled after the real-world behaviors of groups of social animals.

A swarm can be conceived of as a self-organizing distributed system where from of the actions of individuals emerge complex and interesting behaviors of the whole. The main idea of swarm-based algorithms is that the self-organizing properties of a natural swarm can be modeled virtually, and the emergent patterns of this virtual swarm can be exploited for the purposes of solving computational problems.

This paper will focus on so-called "ant algorithms," those that have been inspired by the emergent patterns of ant behavior. While other application will be discussed, the focus of this paper will be how an algorithm (hereafter called the Ant Colony Optimization (ACO) algorithm) inspired by the foraging behaviors of ant colonies might be used for solving the Traveling Salesman Problem (TSP).

Real Ants

Given the limited perceptive capabilities of ants, the ability for ant colonies to establish remarkably efficient paths between nest and food source has spurred much research into the inner workings of the colonies. Entomological studies have reveled that many colony-level behaviors are coordinated through modifications each ant is able to make in its environment. This form of indirect communication between agents is called *stigmergy* (Grassé, 1959) and has been shown to be the primary coordination mechanism of ant colonies.

In the case of foraging behavior, pheromones serve quite simply as trail markers. As each ant moves about, it deposits pheromone on the ground. This pheromone trail-laying behavior produces what amounts to an invisible trail of breadcrumbs being left behind each ant. Subsequent ants exploring the same area randomly choose directions to move in, all the while giving pheromone laden paths greater preference (proportional to how much pheromone is on that trail) in the random selection process.

For example, the foraging ant seen in Figure 1 is confronted with choosing one of the two paths before it. Path-A has been marked by two pheromone trails, while only 1 is marking Path-B. At every such decision point, an ant will make a proportional random choice as to which path to follow. In this case, the ant will randomly select A or B with twice as much probability of choosing A. This can be thought of as randomly selecting an element from the set {A, A, B} and following the path given by the selected element.

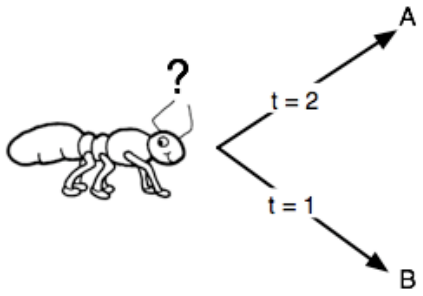

Figure 1: A foraging ant confronted with a two paths. Path-A has 2 units of pheromone and Path-B has 1

It is this collective trail-laying and -following behavior that produces the highly unified global behavior of the colony and is the inspiration for ACO. A particularly brilliant experiment that exposed the process by which foraging ants generate short paths between nest and food source was carried out by biologist J. L. Deneubourg and his colleagues (See Deneubourg et. al., 1990). The double bridge apparatus seen in Figure 2 connects a colony of ants (seen on the left) to a food source (right) via two bridges of unequal length. At the start of the experiment, the ants were allowed to freely move between nest and food source. Over time, it was observed that the ants were using the short path almost exclusively for commuting between nest and food source.

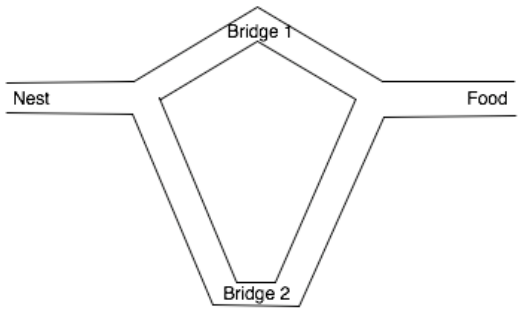

Figure 2: The apparatus for the double bridge experiment. Bridge-2 is exactly twice as long as Bridge-1

The remarkable outcome of the double bridge experiment can be explained as follows. For simplicity's sake, we will assume that there are only two ants in this particular ant colony, call them "ant1" and "ant2." At the start of the experiment, both ants set out from the nest and begin moving toward the food source. Both ants are then confronted with a choice to take either Bridge-1 or Bridge-2. The ants "smell" the two paths in front of them, and each finds that there is no pheromone on either bridge. Since Bridge-1 and Bridge-2 appear identical to the ants, each randomly chooses a bridge to take with equal preference given to both bridges. This means that roughly half of the ants will select Bridge-1, and the others, Bridge-2. Therefore, we will now assume that ant1 selects Bridge-1, and ant2 selects Bridge-2, and both begin moving along their respective path and depositing pheromone as they move.

Since Bridge-2 is exactly twice as long as Bridge-1, we know that ant1 will arrive at the food source when ant2 is at the midpoint of Bridge-2. As ant1 turns around to return back to the nest, it is now confronted with another bridge-selection decision. Once again, ant1 sees that there is no pheromone on Bridge-2. However, an examination of Bridge-1 reveals that is marked by a single pheromone trail. Ant1 then selects between Bridge-1 and Bridge-2 randomly, but giving Bridge-1 added preference proportional to the amount of pheromone on it. Since Bridge-1 has more pheromone on it than Bridge-2 (which has none), we will assume ant1 selects Bridge-1 to return home on.

The next interesting moment occurs when ant1, having reached the nest, and ant2, having reached the food, turn around and begin heading toward food and nest respectively. At this point, each ant is once again confronted with choosing between Bridge-1 and

Bridge-2. Both ants find that Bridge1 is now marked by 2 pheromone trails (deposited during ant1's journey to *and* from food source). Bridge-2 is now marked by 1 pheromone trail (deposited during ant2's journey to the food source). Because Bridge-1 contains twice as many pheromone trails as Bridge-2, each ant is twice as likely to select Bridge-1 than they are Bridge-2 and so we will assume that both ants select Bridge-1 for the next leg of their respective journeys.

As the two ants continue to move back and forth from nest to food, the ratio of pheromone trails on Bridge-1 to Bridge-2 will ensure that both ants will almost always select Bridge-1. At this point, we can point out an interesting pattern that has emerged from the choices of the individual ants. After an initial exploration phase, both ants end up using the shortest path between nest and food almost exclusively (there is still the possibility that the random choice rule will direct an ant toward the longer path, but these occurrences are few). It is this built-in path optimization characteristic of ant colonies that is the inspiration for ACO.

The TSP and Virtual Ants

With an understanding of ant colony behavior in hand, we can now begin exploring how we might use their optimization characteristics for tackling the TSP, as well as other discrete optimization problems (discussed in the final section of this paper).

The TSP is the problem of finding the shortest possible closed tour through a given set of cities. That is to say, given a set of cities and a starting point, find the shortest possible path that visits every city once and then returns to the starting city. Intuitively, we see that this problem is a kin to that presented to the ants in the double bridge experiment described above. What we must work toward is a generalization of the double bridge experiment that provides for an arbitrary number of nodes (as opposed to the two nodes, nest and food, of the double bridge experiment).

To this end, we begin by showing how the apparatus of the double bridge experiment can be represented by a graph consisting of two vertices and two edges of un-equal cost (see Figure 3). The link between the double bridge experiment and the TSP lies in this representation. "The TSP can be represented by a complete weighted graph $G = (N, A)$ with N being the set of $n = |N|$ nodes (cities), A being the set of arcs fully connecting the

nodes. Each arc (*i, j*) in *A* is assigned a weight d_{ij} which represents the distance between cities *i* and *j*. The TSP is then the problem of finding the minimum cost Hamiltonian circuit of the graph. " (M. Dorigo & T. Stützle, 2004) The key is that in the eyes of the ants, solving the TSP by traversing a connected graph is essentially an *n*-bridge experiment. An example of a problem graph for the TSP can be seen in Figure 4.

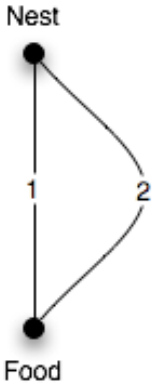

Figure 3: The double bridge apparatus expressed as a graph

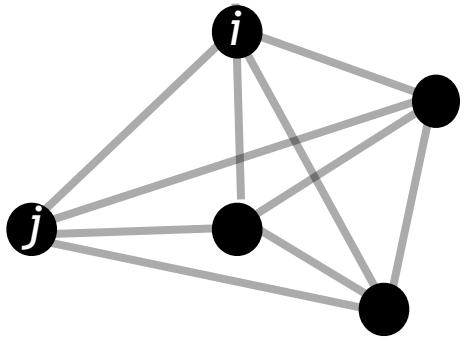

Figure 4: An example of a TSP problem graph, cities i *and* j *are labeled*

Since we are going to be working with virtual ants, we are afforded some advantages not present in real ant colonies. For one thing, we can incorporate information we have about the distances between nodes into the decision making process of each ant. Where with real ants, the inclination for choosing one path over another is given by the amount of pheromone on candidate paths, virtual ants are also able to take into account the lengths of candidate paths. For this reason, we can say that the probability *P* that some ant-*k* will move from city *i* to city *j* is given by:

$$P_{i,j}^{k} = \frac{(\text{pheromone}[i][j])^{\alpha} * \left(\frac{1}{\text{distance}[i][j]}\right)^{\beta}}{\sum_{l \in N_i^k} (\text{pheromone}[i][l])^{\alpha} * \left(\frac{1}{\text{distance}[i][l]}\right)^{\beta}} \quad , \text{ if } j \in N_i^k$$

$$P_{i,j}^{k} = 0 \quad , \text{ else}$$

Where:

- N_i^k is the set of nodes not yet visited by ant-*k*
- α and β are some constants
- pheromone[i][j] is the amount of pheromone on edge-ij
- distance[i][j] is the distance between cities *i* and *j*

Figure 5: The random proportional choice rule

In words, the probability of ant-*k* choosing to move toward city *j* from city *i* is given by computing the choice value of *j*, equaling the product of the amount of pheromone on the edge connecting *i* and *j* and the inverse of the distance between *i* and *j* which is then normalized by dividing this choice value by the sum of the choice values of visiting any other unvisited node. Of course, if ant-*k* has already visited some city *j*, the probability of it returning there is 0 as visiting a node twice contradicts the constraints of the TSP as stated above.

Another advantage of working with virtual ants is that there can be programmatic control over which ants are allowed to generate pheromone trails. Since at the first iteration of the algorithm there are no pheromone trails and ants will explore rather chaotically, a good number of simply horrible paths are explored first. Since real ants always deposit pheromone as they move, subsequent ants may follow, and reinforce, these inefficient paths and in the long run, draw ants away from potentially better routes. This problem can be handled in the virtual world by only allowing ants that generate "good" paths to deposit pheromone. This can be accomplished by, at every iteration, running some number of ants around the graph and then allowing only the ant that created the shortest tour to deposit pheromone.

In order to allow for bad routing decisions to be "forgotten" by the colony, ACO algorithms also implement a pheromone evaporation scheme. That is, after every iteration some percentage of pheromone is removed from every edge. This characteristic of the virtual environment allows for less traveled (and presumably bad) paths to be "forgotten" by the colony as a whole.

Finally, the ACO algorithm described below utilizes upper and lower bounds on the amount of pheromone allowed on any edge. This ensures that no one edge becomes overly attractive and that no edge ends up looking completely unattractive. This is necessary to prevent solution stagnation where the colony prematurely converges to a non-optimal path simply because there is so much pheromone on it.

An ACO Algorithm for the TSP

Presented here is an ACO algorithm very similar to the Min-Max Ant System (MMAS) algorithm originally proposed by Thomas Stützle and Holger H. Hoos. (T. Stützle & H. Hoos, 1997) Roughly, the algorithm works like this:

MIN-MAX AS

1. Generate a graph to find the optimal tour of
2. *allTimeBestTour* = a tour of length infinity
3. Do until *allTimeBestTour* does not change for X consecutive iterations:
    a. Account for evaporation of pheromone by reducing the pheromone levels on every edge by some percentage
    b. Generate some number of ants and have each of them generate a tour of all the vertices of the graph
    c. Select the ant that generated the shortest tour, call this tour *iterationBestTour*
    d. For each edge traversed by *iterationBestTour*, add an amount of pheromone equal to 1/length(*itertionBestTour*)
    e. If *iterationBestTour* is shorter than *allTimeBestTour*, *allTimeBestTour* = *iterationBestTour*
4. Return *allTimeBestTour*

Figure 6: A pseudo code outline of MMAS

The version of MMAS presented here was implemented in Java and consists of two main classes along with some other classes representing useful abstractions. The table below contains descriptions of each class as well as data structures used by the algorithm, followed by a detailed pseudo code outline of the two main classes.

Class Name	**Description**
AntRunner	This is the main entry point for the program. It is a daemon process that is primarily concerned with maintaining the graph (building the graph initially, applying pheromone to edges during runtime, handling evaporation, etc) and spawning ants to explore it.
Ant	This class represents an individual ant. When an ant is instantiated, it automatically begins traversing the graph, keeping a record of the path it is taking. At the end of it's life, it sees if it's tour is the best of it's peers, and if so, sets iterationBestTour to equal it's tour.
Node	This is a helper class and simply represents a node in the graph. A node has x- and y-coordinates and a unique ID number. The problem graph is constructed of Nodes

Tour	This is a helper class that represents a tour through the nodes of the problem graph. It provides methods for getting the length of this tour as well as getting the order in which the nodes are visited.
nodes[]	This is an array of Node objects. It contains all the nodes of the problem graph
distance[][]	This is a matrix of double precision floating point numbers where distance[i][j] = the distance between cities *i* and *j.*
pheromone[][]	This is a matrix of double precision floating point numbers where pheromone[i][j] = the amount of pheromone on the edge connecting nodes *i* and *j.*
tMax	A number dictating the maximum amount of pheromone allowed on any edge. This value is set to the inverse of the length of the best tour found so far.
tMin	A number dictating the minimum amount of pheromone allowed on any edge. This value is set to some fraction of tMax.
evaporationRate	A number dictating how much pheromone is removed from each edge after each iteration. 0 < evaporationRate < 1

Figure 7: A summary of important classes, datastructues, and variables for my algorithm

AntRunner

1. tMax = 1
2. tMin = tMax/64
3. evaporationRate = 0.8
4. Populate nodes[] with *n* Node objects
5. For all nodes i and j, distance[i][j] = distance between nodes i and j
6. For all nodes i and j, pheromone[i][j] = tMax
7. allTimeBestTour = new Tour().setLength(infinity)
8. While (allTimeBestTour has recently changed)
    a. For all nodes i and j, pheromone[i][j] = pheromone[i][j] * evaporationRate
    b. iterationBestTour = new Tour().setLength(infinity)
    c. Do 10 times:
        i. new Ant(nodes[], distance[][], pheromone[][], iterationBestTour)
    d. For all edges e_{ij} traversed in iterationBestTour
        i. pheromone[i][j] = pheromone[i][j] + 1/iterationBestTour.getLength()
    e. If iterationBestTour.getLength() < allTimeBestTour.getLength()
        i. allTimeBestTour = iterationBestTour
        ii. tMax = 1/allTimeBestTour.getLength()
        iii. tMin = tMax/64
9. Return allTimeBestTour

Figure 8: A pseudo code outline of the AntRunner class

Ant

1. thisPath = a list of nodes ordered by the ordering in which the ant visits them
2. iterationBestTour = the shortest tour found so far by the ants of this iteration
3. For all nodes n, set n.visited = false
4. Pick a random node to start at, call this startNode
5. Add startNode to path
6. Set startNode.visited = true
7. currNode = startNode
8. While (all nodes have not been visited)
    a. nextNode = a randomly selected unvisited node, selected by applying the random proportional choice rule
    b. Add nextNode to thisPath
    c. nextNode.visited = true
    d. currNode = nextNode
9. Add startNode to path
10. If thisTour.getLength() < iterationBestTour.getLength()
    a. iterationBestTour = thisTour

Figure 9: A pseudo code outline of the Ant class

A complete copy of the implemented code as well as an executable version can be found on the electronic addendum to this paper.

Analysis and Improvements

Proving much of anything about ACO algorithms is inherently difficult for a couple of reasons. First of all, we are confronted with the rather loose definition of the approach. While this generality allows ACO to be applied to a wide variety of problems, each application changes properties of the approach, making overall theoretical analysis difficult, if not impossible. (M. Dorigo & T. Stützle, 2004) Moreover, the stochastic nature of the algorithm introduces a good deal of non-deterministic behavior that further complicates any rigorous proof. Suffice it to say, it can be proven that ACO algorithms such as the one implemented here do converge in solution. That is, it can be shown that given enough running time, the algorithm will produce an optimal solution. The depths of these proofs are, however, beyond the scope of this paper and will not be explored herein (See M. Dorigo & T. Stützle, 2004 for complete proofs).

Combinatorial optimization problems, such as the TSP, are intriguing as they are often quite easy to state, yet exceedingly difficult to solve. Many such problems are

considered *NP*-hard and finding an optimal solution often requires infeasible amounts of computer time and/or power. For this reason, we turn to approximate methods, those that return near-optimal solutions, for solving these types of problems. Algorithms of this type are loosely called heuristic and this is the class to which ACO belongs.

Due to the approximate nature of ACO algorithms, there is often room for improvement in solutions by post-processing the solutions generated by the ants. Combining a constructive algorithm such as MMAS, with a local search algorithm such as a 2-opt or k-exchange scheme will lead to better solutions being generated overall. For this reason, the implementation of MMAS found in the electronic addendum to this paper also contains a step where the iterationBestTour is run through a simple 2-opt algorithm to eliminate any crosses in the path. Performing this step is rather expensive in terms of computation time, but yields dramatically better results overall as can be seen in the data below.

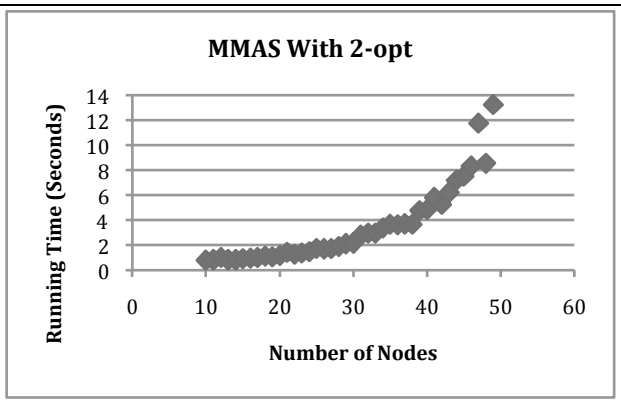

Figure 10: Running time of algorithm with 2-opt enabled

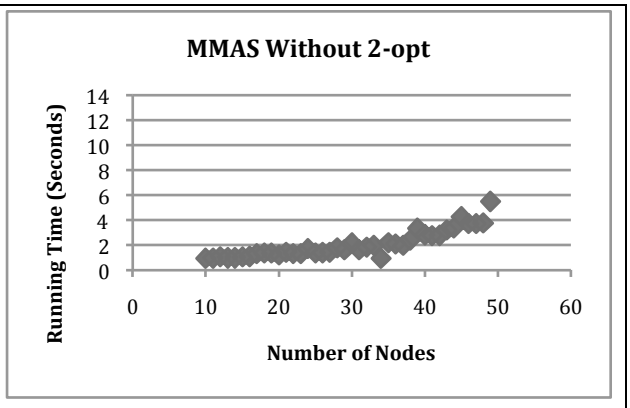

Figure 11: Running time of algorithm with 2-opt disabled

Figures 10 and 11 show the running time required for the algorithm to stabilize as a function of input size, or number of nodes, with 2-opt processing of iterationBestTour enabled and disabled respectively. The 2-opt algorithm being used here exhaustively searches the tour for instances where the path crosses itself so as to reduce the overall length of the tour. Since this generates roughly $\frac{n!}{2}$ comparisons, this implementation of 2-opt is not suitable to be used with large datasets.

It should also be noted that when running the algorithm with 2-opt enabled, the optimal tour tends to be found in far fewer iterations than under the non-2-opted version. The problem is that during runtime, the algorithm has no idea if it has reached the optimal solution and must continue to try to improve the tour until the stop condition is met (200 consecutive iterations of no improvement). So while the 2-opt enabled version might terminate after 250 iterations and the other after 450, the overall running time is much longer due to the cost of each iteration. A possible solution to this problem is to create a better exit condition for the 2-opt version as to prevent the algorithm from repeatedly trying to 2-opt an already optimal solution. A yet better scheme would be to apply 2-opt only to allTimeBestTour every time a new one is found and then reinforce this 2-optimal path by applying high levels of pheromone to it. This would cut down on the number of calls to the 2-opt algorithm as well as concentrate the ants' search around a known 2-optimal solution (a presumably fruitful region of solution space).

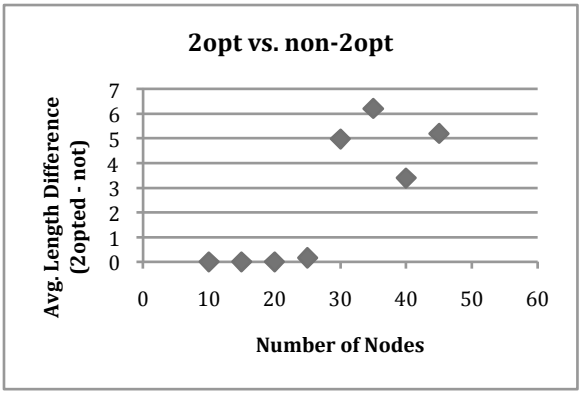

Figure 12: The departure from optimal of the solutions given by the non-2-opt algorithm

Figure 12 shows the average departure from optimal tour length of non-2-opted solutions. Clearly, the approach to cross-eliminating post-processing described above does not scale well but does in fact produce much more optimal solutions than the non-2-opt version.

Other Applications and Future Development of ACO

ACO algorithms similar to that described here can be applied to a range of other optimization problems. ACO has been applied with much success for solving network routing, sequential ordering, and multiple knapsack problems. In the case of network

routing, the problem is quite similar to that of the TSP. In this case, we use link quality (derived from factors such as bandwidth, latency, etc) instead of distance to determine the cost of traversing any given edge. The sequential ordering problem can be conceived of as a type of asymmetric TSP where weights are applied to edges in the graph in such a way as to enforce precedence constraints between nodes. And finally, the multiple knapsack problem can be approached by regarding pheromone trails as representative of the desirability to add some item to the partial solution set. A tour of some subset of nodes then represents the optimal set of items to put in the knapsacks.

Due to the distributed, self-organizing nature of ACO algorithms, the approach lends itself well to parallelization and an efficient parallelized implementation is quite desirable. Other directions for future research in ACO are in the area of dynamic and multi-objective optimization problems. In the case of dynamic optimization problems, we are confronted with that characteristic that over time, aspects of the problem change in relation to some underlying subsystem or function. With multi-objective optimization, the goal is to find the solution that best compromises multiple objectives. In both of these cases, ACO proves to be a fruitful area of research.

Works Cited

Grassé, PP: 1959, La Reconstruction du nid et les coordinations interindividuelles, La théorie de la stigmergie, Insectes Sociaux 6: 41-84.

J.-L. Deneubourg, S. Aron, S. Goss, and J.-M. Pasteels. The self-organizing exploratory pattern of the Argentine ant. Journal of Insect Behavior, 3:159-- 168, 1990.

M. Dorigo & T. Stützle, 2004. Ant Colony Optimization, MIT Press. ISBN 0-262-04219-3

T. Stützle and H. Hoos. The MAX-MIN Ant System and Local Search for the Traveling Salesman Problem. In T. Baeck, Z. Michalewicz, and X. Yao, editors, Proceedings of IEEE-ICEC-EPS'97, IEEE International Conference on Evolutionary Computation and Evolutionary Programming Conference, pages 309-314. IEEE Press, 1997b.