

Make it  
**Probabilistic  
methods**

# Ant colony optimisation

**Specimen # 1**  
Recording ant behaviour

Using ant behaviour to estimate the best solution to problems

## *In this issue...*

### ▶ WHAT'S COVERED

There is a certain class of problems in computer science that have a huge solution space but no simple algorithm to calculate an exact solution. The only way to solve these problems exactly is to enumerate the entire set of solutions and choose the right one. Since this would take far too long, solutions are instead estimated using probabilistic methods. One such method is to copy the behaviour of ants as they wander from the nest to forage for food.

**T**he most famous of these intractable problems is the travelling salesman problem (TSP). We imagine a set of cities that are all interconnected (so each city is connected to all others) and a salesman that has to visit them all. Each link between two cities has a (positive) cost associated with it, the simplest cost being the distance by road between them (if there is no direct road between two cities we just set the cost/distance to infinity). The TSP is then described as: position the salesman in a given city and then find the shortest distance the salesman has to travel in order to visit every other city once and return to his starting point.

For two cities, the TSP is trivial: there's only one link to consider and its cost is the solution. For three cities, it's still pretty easy: there are only three links and there are six possible routes. For five cities, see Figure 1 for a map of England with some cities highlighted and Figure 2 for

the distances between them – see if you can work out the tour with the smallest mileage. Jumping to 20 cities, we're in deep trouble. There will be 190 links and a solution space of 20! routes – or  $2.4 \times 10^{18}$  solutions. If we could calculate solutions at the rate of one billion per second, we'd need 2.4 billion seconds to enumerate them all. A mere 145 years or so.

The solution space explodes in size very quickly when the number of items increases. Enumerating the entire solution space in order to find the smallest is just not a feasible algorithm, even when there are only a small number of items in the problem.

### **Good enough**

Given that finding the exact minimal solution is extremely time-consuming, the only thing we can do is to find a 'good enough' solution. A good enough solution is one that we can arrive at in a reasonable amount of time and that is (probably) sufficiently close to the actual minimal solution that we don't need to take the extra time to refine it.

The accepted method of finding such a solution is via some kind of perturbation or optimisation algorithm. In very hand-wavy terms, we work out a set of solutions and then perturb them in minor ways to try and find a slightly better set of solutions. Repeat this process a few times, making minor perturbations and selecting the best, and we'll eventually find a good enough solution.

The trick is then to work out a good perturbation algorithm – one that doesn't get you stuck in local minima, making you miss the global minimum. The way to think about this is to view the

## *Annealing*

Kirkpatrick, Gelatt and Vecchi devised a famous perturbation algorithm called simulated annealing in the early 1980s. Annealing is the process whereby you heat a metal object and then control its cooling to increase the crystal size and hardness. Heating causes the atoms in the original matrix to become unstuck and move around, and the slow cooling encourages them to find positions in the crystal matrix of lower energy than before. The simulation cycles through possible solutions, wildly varying for high 'temperatures' but becoming less wild as the temperature is decreased. Hopefully, the solution will settle down close to the optimal solution. ■

solution space as a very hilly countryside. There are hills and valleys, and your job is to find the deepest valley. The problem is that when you are in a valley, you can't look around to find the next deeper valley. You have to climb a hill and take a look from there. Eventually you get fed up and the deepest valley you've found by that point will be your good enough solution. The best such algorithms devised so far – such as genetic algorithms, simulated annealing – have mimicked some process found in nature.

In 1992, Marco Dorigo invented a new algorithm inspired by another part of nature: how ants forage for food.

### **Foraging for food**

Worker ants wander out from the nest to try and find food. As they wander around they lay down a pheromone trail (a chemical 'scent', if you like) that ants have evolved to follow. If an ant finds some food, it will return with

## **Spotlight on... ant algorithm flow**

In essence, the flow of the ant colony optimisation algorithm goes like this for a TSP: Load the data for the TSP – the cities and the distances between them. It's best to use a matrix for this. Remember, if there is no link between two of the cities, just make the distance a very large number. Set the pheromone levels in the matrix to some random value.

Start a loop for the number of ants you want to use – 100 or 200, say. Within each loop create a random walk for the

ant. Start at a particular city, and then randomly choose the next city to visit based on the pheromone levels to each city divided by the distance (both raised to some power; try experimenting).

If the tour is smaller than the previous smallest, make a note of it. Lay down a constant amount of pheromone along the ant's tour. Evaporate the pheromone.

Once you've cycled through all of the different ants, you should have a good idea of what the smallest tour is. ■



▲ **Figure 1: An example travelling salesman problem in England.**

some of the food and continue to lay down a pheromone trail on the way back. Other ants, on finding a pheromone trail, are likely to follow it and lay down their own pheromone trail as they do so, increasing the pheromone levels along that path. The net effect is that when a large food supply is found, the pheromone trail leading to it is emphasised more than surrounding trails because more ants follow it. This increases the likelihood that more ants will follow it — therefore depositing more pheromone and encouraging even more ants to follow the path to the food. It's a positive feedback system, in other words.

Luckily for the ants, the pheromone they use as markers

slowly evaporates; so little used trails tend to remain little used. Another benefit of this evaporation is a distinction between long and short paths. Because it takes longer to walk the longer trails, the pheromone has more time to evaporate than for shorter trails. This means that shorter trails tend to have higher levels of pheromone. Without this effect, shorter paths might not get explored at all. Again, this is a positive feedback system.

All in all, this sounds like a good system to mimic for solving optimisation problems. The ant foraging behaviour also lends itself well to the TSP since both the ant and the salesperson want to minimise distance.

	London	Manchester	Birmingham	Newcastle	Carlisle
London	0	181	121	289	306
Manchester	181	0	96	145	121
Birmingham	121	96	0	205	193
Newcastle	289	145	205	0	59
Carlisle	306	121	193	59	0

▲ **Figure 2: The distances between cities located in the UK.**

Suppose that we have a traditional TSP with  $n$  cities. We have an  $n \times n$  matrix containing the distances between cities. Let's mimic pheromone with a simple floating-point number: the higher the number, the higher the level of pheromone. We can create another  $n \times n$  matrix containing the pheromone level along the paths between cities (so the value in the cell  $(i, j)$  is the level of pheromone between city  $i$  and city  $j$ ).

We'll suppose that an ant is an abstraction in our program, one of many that we'll create. Each ant will visit every city in our problem. The path it takes to visit them all is probabilistic in nature. That is, the ant doesn't just follow the most attractive path every time; instead, it chooses the next city to visit by tossing a coin. The attractiveness of choosing a particular path between cities is dependent on the level of pheromone (the higher the better) and on the distance between them (the lower the better).

### Speeding it up

Marco Dorigo actually added a power component to the pheromone levels and distances, and investigated various values for the exponents to find out which ones produce a better solution faster. So the 'attractiveness' of a particular path is the ratio of the pheromone level raised to a power to the distance raised to another power.

We release ants one by one to visit every city and then record their path and pheromone trail at the end of their exploration. Once an ant has completed his path, we update the levels of pheromone along each link of his route. The amount we use for a particular ant is inversely proportional to the total length of his route, so that an ant that takes a very long tour deposits a smaller amount of pheromone compared to an ant that takes a very short tour. This makes the short tour more attractive in the future.

### Genetic algorithms

Another variant of optimisation algorithms is genetic algorithms. Here we create a large population of solutions to the problem and then use evolutionary methods to perturb that population. In each cycle, the fitness of each individual in the population to survive is evaluated. Certain individuals are selected based on their fitness and combined in an evolutionary way (essentially the individuals are mapped to some chromosomal representation, and these 'chromosomes' are recombined and mutated) to form the next generation of solutions. The algorithm stops once the fitness of all individuals in the population is to a certain tolerance or after a certain number of generations have passed. ■

We then release each of the other ants in our population (we suppose we have  $N$  ants in our population – yet another 'knob' for our algorithm) to visit every city and update the results after each ant. Once each of the  $N$  ants has made its tour, we 'evaporate' the pheromone levels. For this we just multiply the level of pheromone in each link by 0.5 (halving it, but this is essentially another algorithmic knob we can tweak),

In essence, using this formula, we are assured that the amount of pheromone for a link never reaches zero. If it did, it would mean that that particular link between cities would never be followed. In fact, that observation implies that we must take care to seed all of the links between the cities with a non-zero level of pheromone before we even start on our algorithm, otherwise our ants would not visit any city at all.

The process of pheromone evaporation gives us a benefit: it allows the algorithm to slowly 'forget' past history. This gives the whole algorithm a way to continue to find new routes without being constrained unduly by past decisions.

Using this algorithm and tweaking the various 'knobs' produces the best-known solutions to various travelling salesman problems. ■

*Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express. [feedback@peplus.co.uk](mailto:feedback@peplus.co.uk)*