Ant Algorithms

1 Introduction

Ant algorithms find solutions and near-solutions to intractable optimisation problems. They have been applied to many optimisation problems including: travelling salesperson problems, quadratic assignment problems, job scheduling, vehicle routing and network routing, showing highly competitive performance.

Ant algorithms are inspired by the behaviour of real ants. In Nature, forager ants communicate indirectly by depositing and sensing *pheromone* trails. This sets up a positive feedback loop that reinforces promising paths: the more ants that choose a path, the more pheromone that gets deposited on that path, increasing the probability that further ants will choose the path.

In ant algorithms, artificial ants complete a series of walks of a graph, each path through the graph corresponding to a potential solution to the optimisation problem in hand. Pheromone is deposited on a path in a quantity proportional to the quality of the solution represented by that path. The ants resolve choices between competing destinations probabilistically, where the probabilities are proportional to the amount of pheromone associated with each option.

2 The Travelling Salesperson Problem

In Computer Science, we often want to find shortest paths in graphs. The vertices and edges of the graph might represent towns and roads, railway stations and tracks, airports and flight paths, components on a circuit board and wires, computers and telecommunications links, etc.

Here we focus on a classic problem of this kind, the *Travelling Salesperson Problem (TSP)*. In the TSP, we are given a graph $G = \langle V, E \rangle$, where V is the set of vertices and E is the set of (undirected, i.e. bidirectional) edges. There is a cost (e.g. distance) associated with each edge. For the edge between vertices v_i and v_j, we will write $\mathrm{dist}(v_i, v_j)$ for this cost. We will assume that the graph is *complete*, which here means that there is an edge between every pair of vertices. In real problems, this may not be the case, e.g. there might not be a direct road between every pair of towns. To model this situation, our complete graph can contain an edge, even where there is no road, but the cost of the edge will be ∞.

The problem is to find a minimum cost path through the graph that starts at some vertex, visits every other vertex exactly once, and then returns to the starting vertex.

The obvious algorithm for solving TSPs is: generate each path; calculate the length of each path; and then choose the cheapest. This, however, is not a practical algorithm. For n vertices, there are $(n-1)/2$ paths, a quantity which grows exponentially.

If you know in advance that the minimum cost path will definitely cost less than some value θ, then a more efficient algorithm is possible. In this algorithm, you calculate the path's cost-so-far while generating the path. If at any point the cost-so-far of the path you are generating exceeds θ, you abandon it. Once this is done, of the paths that were not abandoned, you choose the cheapest. This approach has been able to solve quite large TSPs. But to be successful, you must be able to guess a value for θ which is neither too low (otherwise you'll abandon all paths, including the cheapest) nor too high (otherwise you'll not abandon enough paths and the algorithm will remain impractical). Guessing θ is not always easy!

Instead, *stochastic methods*, which work probabilistically are often used. Ant algorithms fall into this category. Stochastic methods do not guarantee to find the optimal solution to a problem. Optimality is sacrificed for efficiency: stochastic methods often find *good* solutions in reasonable time.

3 Ant Algorithms for TSPs

3.1 Basic ant algorithm for TSPs

Here, in pseudocode, is a basic ant algorithm for solving a TSP:

```
INITIALISEPHEROMONE();
bestOfRun := null;
do
{   bestOfIteration := null;
    for each ant a
    {   path := WALK(a);
        if bestOfIteration = null ∨ cost(path) < cost(bestOfIteration)
        {   bestOfIteration := path;
        }
    }
    UPDATEPHEROMONE(bestOfIteration);
    if bestOfRun = null ∨ cost(bestOfIteration) < cost(bestOfRun)
    {   bestOfRun := bestOfIteration;
    }
}
until termination condition is met;
return bestOfRun;
```

Initially, an amount of pheromone is deposited onto the graph (see Section 3.2). Then, in each iteration, ant a constructs a path by walking the graph. (see Section 3.3). When an iteration is completed (i.e. all ants have walked the graph), the pheromone on the graph is updated (see Section 3.5).

The algorithm iterates until some termination condition is met, e.g. until a predetermined number of iterations, a maximum amount of time or an acceptable path cost is reached. Throughout, the algorithm keeps track of the best path in the current iteration and the best path found overall.

3.2 Initialising the pheromone

Consider an edge between vertices v_i and v_j. Let the amount of pheromone that is associated with that edge be denoted by $\tau_{\langle v_i, v_j \rangle}$.

At the start of the algorithm, all edges are initialised to the maximum pheromone, τ_{max}. This makes all edges quite attractive in early iterations, keeping exploration high in these iterations.

```
Algorithm: INITIALISEPHEROMONE()
for each edge ⟨v_i, v_j⟩ ∈ E
{   τ_⟨v_i,v_j⟩ := τ_max;
}
```

3.3 Walking the graph

Algorithm: WALK(a)

tabu := { };
path := ⟨ ⟩;
initVtx := a vertex from V, chosen randomly;
insert *initVtx* into *tabu*;
add *initVtx* into *path*;
currVtx := *initVtx*;
while $(V \setminus tabu) \neq \{\}$
{ *currVtx* := CHOOSENXTVTX(*currVtx*, $V \setminus$ *tabu*);
insert *currVtx* into *tabu*;
add *currVtx* to end of *path*;
}
add *initVtx* to end of *path*;
return *path*;

The ant is placed on a randomly chosen initial vertex. As it moves from vertex to vertex, the newly-visited vertices are added onto its path and also into a tabu set. The idea of the tabu set is to prevent the ant from visiting a vertex more than once. Hence, an ant chooses its next vertex from $V \setminus$ *tabu*, i.e. vertices other than ones in *tabu*. By the time $V \setminus$ *tabu* is the empty set, then all vertices have been visited, and the ant can return to the initial vertex.

The observant reader will realise that these ants are not strictly reactive agents. These ants have memory: the tabu set. Maybe this lecture should come after the next two lectures, which bring in the idea of memory. On the other hand, it does seem to follow logically from the lecture on Swarm Intelligence.

3.4 Choosing the next vertex

This part of the algorithm is given the ant's current vertex; call it v_i. And it is given the set of unvisited vertices; call it UV. The next vertex v_j is chosen from UV probabilistically. It is common for the choice to depend on a combination of two factors: a *heuristic factor* and a *pheromone factor*.

The heuristic factor HF should give high scores to vertices which, from the ant's point-of-view, look most promising, ignoring the pheromone. In TSPs, we want low cost paths, so the score assigned by the heuristic factor to a vertex $v_j \in UV$ should be inversely proportional to its distance from the current vertex v_i.

$$HF(v_i, v_j) =_{\text{def}} \frac{1}{\text{dist}(v_i, v_j)}$$

The pheromone factor, on the other hand, gives high scores to vertices which look most promising based on the pheromone.

$$PF(v_i, v_j) =_{\text{def}} \tau_{\langle v_i, v_j \rangle}$$

Since this factor is based on the pheromone deposited on edges of the graph in previous iterations, it represents the way that ants indirectly communicate with each other between iterations.

The pheromone factor and the heuristic factor are combined to compute the probability that a particular vertex will be chosen. Parameters α and β are used to vary the effect of each factor. Given current vertex v_i and set of unvisited vertices UV, the probability that v_j is the next vertex is given by:

$$P(\textit{next} = v_j) =_{\text{def}} \frac{HF(v_i, v_j)^\alpha \times PF(v_i, v_j)^\beta}{\sum_{v_k \in UV} HF(v_i, v_k)^\alpha \times PF(v_i, v_k)^\beta}$$

So, in summary, what does procedure CHOOSENEXTVTX() do? It has two parameters: the current vertex v_i and the unvisited vertices UV. For each $v_j \in UV$, it compute a probability as above. Then, based on these probabilities, it uses roulette wheel selection to choose one of the vertices in UV.

3.5 Updating the pheromone

The algorithm for pheromone update is shown below. This algorithm is called once per iteration of the basic ant algorithm. The parameter is the best path found in that iteration.

Algorithm: UPDATEPHEROMONE(*path*)

$\Delta\tau := 1/\text{cost}(\textit{path})$;
for each edge $\langle v_i, v_j \rangle \in E$
{ $\tau_{\langle v_i, v_j \rangle} := (1 - \rho) \times \tau_{\langle v_i, v_j \rangle}$;
if v_i and v_j appear consecutively in *path*
{ $\tau_{\langle v_i, v_j \rangle} := \tau_{\langle v_i, v_j \rangle} + \Delta\tau$;
}
if $\tau_{\langle v_i, v_j \rangle} < \tau_{min}$
{ $\tau_{\langle v_i, v_j \rangle} := \tau_{min}$;
}
if $\tau_{\langle v_i, v_j \rangle} > \tau_{max}$
{ $\tau_{\langle v_i, v_j \rangle} := \tau_{max}$;
}
}

As can be seen, an amount of an edge's existing pheromone evaporates. ρ is the evaporation rate, $0 \leq \rho \leq 1$, and so $(1 - \rho)$ is the proportion that remains. Additionally, edges that occur in the best path found in the current iteration get rewarded with an amount of extra pheromone. Finally, if ever the amount falls outside the range $\tau_{min} - \tau_{max}$ inclusive, where $0 < \tau_{min} < \tau_{max}$, then the algorithm rounds it towards the nearest end-point.

4 Concluding Remarks

It cannot have escaped your notice that these ant algorithms, while inspired by the behaviour of real ants, are far from faithful copies of the behaviour of real ants. For example, the artifical pheromone is updated only *after* the ants have each walked the graph, not *during* the walks. Two other differences are discussed in the next two paragraphs.

Early research into ant algorithms showed that not every ant should get to deposit pheromone. Accordingly, as you have seen, we use an *elitist strategy*, where pheromone is deposited based only on the best path found in an iteration. The reasons for this are clear: on average, it reinforces higher quality paths, while allowing lower quality ones to be forgotten. If every ant were to deposit pheromone, both good and bad paths would be reinforced in every iteration, allowing future ants to be attracted to bad paths. Ultimately, the elitist strategy has a greater chance of converging on a good solution.

Early research also showed the value of imposing minimum and maximum pheromone limits. The idea is to try to avoid stagnation: limiting the differences among paths encourages wider exploration. The upper limit makes it less likely that one or more high-scoring paths dominate the search. The non-zero lower limit ensures that no path is ever wholly excluded from being chosen.

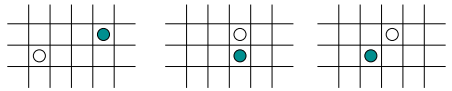
5 Exercise (Past-exam question)

1. Disc-O-Dog-1 is a circular dog-like agent that is being developed for a computer game. Disc-O-Dog-1 will inhabit an infinite grid of equal-sized cells, and is shown in white in the diagrams below. The only other agent in the world is the dog's owner, who is shown in dark grey in the diagrams. Owner and dog must always occupy different cells.

   Disc-O-Dog-1 has two sensors, as follows:

   - s_0 returns the distance along the x-axis from the owner to the dog. Negative values indicate that the dog is to the West of its owner; positive values indicate that the dog is to the East of its owner.
   - s_1 returns the distance along the y-axis from the owner to the dog. Negative values indicate that the dog is to the South of its owner; positive values indicate that the dog is to the North of its owner.

   Here are some examples:

   $s_0 = -3; s_1 = -1$ $\quad$ $s_0 = 0; s_1 = 1$ $\quad$ $s_0 = 1; s_1 = 1$

   Disc-O-Dog-1 has five possible actions: Null, the null action, and MoveN, MoveE, MoveS & MoveW, which move the dog one cell North, East, South and West respectively. The owner also moves only one cell at a time, but may move randomly into any one of its 8 surrounding cells.

   The owner and the dog will take it in turns to select and execute an action. We want the dog to keep as close to its owner as possible.

   (a) **Explain** why Disc-O-Dog-1 cannot be implemented as a *table-driven agent*.

   (b) It is decided to implement Disc-O-Dog-1 using a *production system* of *condition-action rules*. The conflict resolution strategy will be to choose the first rule in the list of rules that matches.

   Define this agent. In other words, say what propositions it will use; explain how the perceptual preprocessor will use the sensor values to set the propositions to true or false; and give the list of condition-action rules.

   For example, you might use a proposition p_0 that determines whether the dog is to the left of its owner. The perceptual preprocessor will set p_0 to true iff $s_0 < 0$. Then, p_0 can be used in a condition-action rule, such as the following

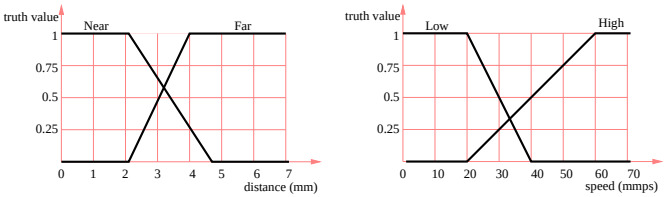
   if p_0 **then** MoveE

   i.e. if the dog is to the left of its owner, then move East. (Of course, something much more sophisticated is needed.)

   Include some paragraphs of explanation (with diagrams if you wish) to **convince** someone that your implementation moves the dog so that it keeps as close to its owner as possible.

2. Disc-O-Dog-2 will operate in a continuous world. It has just one sensor, which measures the *distance in millimetres (mm)* between itself and its owner. It has one action, the Run action, which can take place at different speeds, measured in millimetres per second (mmps).

   The following four fuzzy sets (two per diagram) have been defined for distances and speeds:



   (a) On separate diagrams, draw the membership functions of the following:

   - $High \cup Low$
   - Low'
   - $(Low \cap High)'$

   (b) The following two fuzzy rules are defined:

   if *distance* is *Near* **then** Run at *Low speed*

   if *distance* is *Far* **then** Run at *High speed*

   The sensor tells us that the distance between the dog and its owner is presently 4mm. What, in millimetres per second, should the speed be? **Explain** your answer, including diagrams if you wish. Mathematical exactitude is not necessary: for example, you can guess the centroid, rather than calculating it, provided you explain what you are doing.