When Ants Attack: Comparing Ant Algorithms for Constraint Problems

Finbarr Tarrant and Derek Bridge

Department of Computer Science,
University College, Cork
`fst1@student.cs.ucc.ie/d.bridge@cs.ucc.ie`

Abstract. We describe an ant algorithm for solving constraint problems [Sol02]. We devise a number of variants and carry out experiments. Our preliminary results suggest that the best way to deposit pheromone and the best heuristics for state transitions may differ from current practice.

1 Introduction

In ant algorithms, artificial ants complete a series of walks of a data structure, known as a *construction graph*, each path corresponding to a potential solution to the optimisation problem in hand [DMC96]. *Pheromone* is deposited on a path in a quantity proportional to the quality of the solution represented by that path. Mimicking the behaviour of real ants, an artificial ant resolves choices between competing destinations probabilistically, where the probabilities are proportional to the amount of pheromone associated with each option.

In this paper, we apply ant algorithms to Constraint Problems. Our goal for the moment is to see which of a number of ant algorithms for constraint problems perform best. The most competitive of these can then be compared with other constraint solvers in future work. We define constraint problems in Section 2. Section 3 describes an ant algorithm for constraint problems and some variants of the basic algorithm. Section 4 reports experimental results. Conclusions are drawn in Section 5.

2 Constraint Problems

We define a *constraint satisfaction problem* (CSP) to be a triple (X, D, C). X is a finite set of *variables.* D is a function that associates each $x \in X$ with its *domain*, this being the finite, non-empty set of values that x can assume. C is a set of *constraints* which restrict the values that the variables can simultaneously assume.

A *label* $\langle x, v\rangle$ associates variable x with a value v drawn from x's domain $D(x)$. An *assignment* A is a set of zero, one or more labels in which no variable appears more than once. A *complete assignment* is an assignment that contains a label for each variable in X. A *solution* is a complete assignment that satisfies all the constraints.

$X = \{x, y, z\}$
$D(x) = \{a, b\}$
$D(y) = \{c, d\}$
$D(z) = \{a, b, c\}$

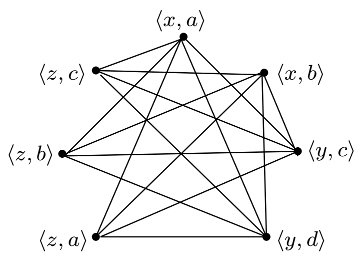

Fig. 1. Construction graph.

Complete solvers guarantee to find a solution, if one exists. However, constraint problems are, in general, NP-hard. For this reason, incomplete solvers are often used. An *incomplete solver* does not guarantee to find a solution; neither can it in general determine when a problem has no solution. A user may be content to use an incomplete solver if, in reasonable time, the solver can find good near-solutions. Ant algorithms are incomplete solvers.

3 Ant Algorithms for Constraint Problems

We investigate a number of different ant algorithms for constraint problems. We have defined them to be systematic variants of a single basic approach [Sol02]. In the algorithms, each ant constructs a complete assignment in a variable-by-variable fashion by walking the construction graph.

3.1 Construction graph

The *construction graph* for a CSP (X, D, C) contains a vertex for every possible label. The graph is fully connected except that $\langle x_i, v\rangle$ need not be connected to any vertex $\langle x_i, w\rangle$ for the same variable x_i. Figure 1 shows an example.

3.2 Algorithm

The basic algorithm is given as Algorithm 1. Initially, an amount of pheromone is deposited onto the graph (see Section 3.3). Then, in each cycle, ant k constructs a complete assignment A_k by walking the graph. When a cycle is completed, ants deposit pheromone onto the graph to denote the desirability of the paths that they followed; at the same time, there is also a degree of pheromone evaporation (see Section 3.3). Cycles continue until either a solution is found or until a predetermined number of cycles has been completed.

The way that ants walk the graph and hence construct a complete assignment is outlined in Algorithm 2. The ant is placed on a starting vertex using the methods described in Section 3.4. It chooses its next vertex from the feasible neighbourhood (Section 3.4), adding a label to its partial assignment for each vertex visited, until it has constructed a complete assignment.

Algorithm 1 Basic algorithm

```
Initialise graph with pheromone
$bestA \leftarrow \{\ \}$ // We take cost($\{\ \}$) $= \infty$
repeat
  for each ant $k$ do
    $A_k \leftarrow$ Walk of graph by ant $k$
    if cost($A_k$) $<$ cost($bestA$) then
      $bestA \leftarrow A_k$
    end if
  end for
  Update pheromone on each component
until cost($bestA$) $= 0 \vee$ max. cycles reached
return $bestA$
```

Algorithm 2 Walk of graph by ant k

```
Select a starting vertex $\langle x_i, v\rangle$ and place ant on this vertex
$A_k \leftarrow \{\langle x_i, v\rangle\}$
while $|A_k| < |X|$ do
  Select vertex $\langle x_j, w\rangle$ from ant's feasible neighbourhood and move ant to this vertex
  $A_k \leftarrow A_k \cup \{\langle x_j, w\rangle\}$
end while
return $A_k$
```

3.3 Pheromone initialisation and update

It is at this point in our presentation that we come upon one of the dimensions along which variants of the algorithm can differ: onto which components of the graph is pheromone deposited?

Vertices: Pheromone can be placed on vertices. In this case, since each vertex represents a label $\langle x_i, v\rangle$, the amount of pheromone $\tau(\langle x_i, v\rangle)$ is proportional to the learned desirability of assigning value v to variable x_i.

Edges: Pheromone can be placed on edges, $\{\langle x_i, v\rangle, \langle x_j, w\rangle\}$. Then, the quantity of pheromone $\tau(\{\langle x_i, v\rangle, \langle x_j, w\rangle\})$ is proportional to the learned desirability of simultaneously assigning v to x_i and w to x_j.

Either way, initialisation and update are similar. Henceforth we will talk of depositing pheromone on *components* of the graph with the understanding that in some variants of the algorithms the components onto which pheromone is deposited are the vertices and in other variants the components are the edges.

As in many ant algorithms, our ants deposit pheromone only *after* they have constructed their solutions. The algorithm for pheromone update is shown as Algorithm 3. As can be seen, an amount of a component's existing pheromone $\tau(i)$ evaporates. ρ is the evaporation rate, $0 \leq \rho \leq 1$. Additionally, certain of the ants deposit pheromone on the components of the graph that they visited. $\Delta\tau(A_k, i)$ is the amount of pheromone that ant k will deposit on component i due to its assignment A_k. The amount $\Delta\tau(A_k, i)$ is inversely proportional to the

Algorithm 3 Update pheromone on each component (where components are vertices or edges depending on which variant of the algorithm is being executed)

```
for each component $i$ in the graph do
    $\tau(i) \leftarrow (1-\rho) \cdot \tau(i) + \sum_{A_k \in BestOfCycle} \Delta\tau(A, i)$
    if $\tau(i) < \tau_{min}$ then $\tau(i) \leftarrow \tau_{min}$
    if $\tau(i) > \tau_{max}$ then $\tau(i) \leftarrow \tau_{max}$
end for
```

cost of the assignment A_k, i.e. the number of constraints that A_k violates. This means that the ants that find the better assignments deposit more pheromone.

The definition of $\Delta\tau(A_k, i)$ differs depending on whether pheromone is being placed on vertices or edges. If pheromone is being deposited on vertices:

$$\Delta\tau(A_k, \langle x_i, v\rangle) = \begin{cases} \frac{1}{\mathrm{cost}(A_k)} & \text{if } \langle x_i, v\rangle \in A_k \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

If pheromone is being deposited on edges:

$$\Delta\tau(A_k, \{\langle x_i, v\rangle, \langle x_j, w\rangle\}) = \begin{cases} \frac{1}{\mathrm{cost}(A_k)} & \text{if } \{\langle x_i, v\rangle, \langle x_j, w\rangle\} \subseteq A_k \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

In some of the very earliest ant algorithms (e.g. [DMC96]), it was realised that not every ant should get to deposit pheromone. Accordingly, we use an *elitist strategy*, where only the best ants in each cycle ($A_k \in \mathit{BestOfCycle}$) deposit pheromone. An elite ant is one whose assignment is no worse than any other ant's assignment.

It can also be seen in Algorithm 3 that the amount of pheromone on a component i is constrained to lie between τ_{min} and τ_{max}, i.e. $\tau_{min} \leq \tau(i) \leq \tau_{max}$ where $0 < \tau_{min} \leq \tau_{max}$ [SH98]. The limits restrict differences among paths, which encourages wider exploration.

It remains to mention pheromone initialisation at the start of Algorithm 1. All components are initialised to the maximum allowed, τ_{max}. This makes all choices quite attractive in early cycles, keeping exploration high in these cycles.

3.4 Vertex selection

An ant chooses the next vertex to which it will move from its *feasible neighbourhood*, Nbrs_k. Vertices are in an ant's feasible neighbourhood if they bind values to currently unassigned variables. The next vertex is chosen from the feasible neighbourhood by a probabilistic *state transition rule*. The choice depends on two factors:

Heuristic factor η_{A_k}: This evaluates the promise of each vertex based on information local to the ant (in our case, the assignment so far, A_k).

Pheromone factor τ_{A_k}: This evaluates the learned desirability of each vertex. This is based on the pheromone $\tau(i)$, deposited on components of the graph i in previous cycles and hence represents the way that ants indirectly communicate with each other between cycles.

We will look at these in more detail in turn.

Heuristic factor There are any number of ways of defining the heuristic factor, $\eta_{A_k}(\langle x_j, w\rangle)$. The most obvious for constraint problems, and the one we use in all variants of the algorithm, is to use an amount that is inversely proportional to the number of additional constraints that would be violated.

Pheromone factor The way we compute the pheromone factor will depend on whether ants deposit pheromone on vertices or on edges (Section 3.3). And, in fact, there are two ways of computing the pheromone factor in the case where pheromone has been placed on edges. Suppose ant k is at vertex $\langle x_i, v\rangle$, then the pheromone factor $\tau_{A_k}(\langle x_j, w\rangle)$ for adding vertex $\langle x_j, w\rangle$ to the partial assignment A_k, is given by one of the following three equations for use in different variants.

Vertices: This defines the pheromone factor simply as the pheromone on vertex $\langle x_j, w\rangle$:

$$\tau_{A_k}(\langle x_j, w\rangle) = \tau(\langle x_j, w\rangle) \tag{3}$$

Edges: This defines the pheromone factor simply as the pheromone on the edge that connects current vertex $\langle x_i, v\rangle$ to proposed vertex $\langle x_j, w\rangle$:

$$\tau_{A_k}(\langle x_j, w\rangle) = \tau(\{\langle x_i, v\rangle, \langle x_j, w\rangle\}) \tag{4}$$

Edges with sums: Noting that the order of the labels in A_k is not significant, this defines the pheromone factor in a way that takes into account all the labels in A_k, rather than just the last one:

$$\tau_{A_k}(\langle x_j, w\rangle) = \sum_{\langle z,u\rangle \in A_k} \tau(\{\langle z, u\rangle, \langle x_j, w\rangle\}) \tag{5}$$

Combining the heuristic and pheromone factors 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. However, there are different formulae for combining them depending on another dimension along which variants of the algorithm can differ.

Compound decision: In this alternative, the algorithm chooses a vertex from the feasible neighbourhood, thus simultaneously choosing both an unassigned variable and a value for that variable. Formally, if ant k has built partial assignment A_k, then the probability of choosing $\langle x_j, w\rangle \in Nbrs_k$ is:

$$\mathrm{Prob}(\langle x_j, w\rangle) = \frac{[\tau_{A_k}(\langle x_j, w\rangle)]^\alpha \cdot [\eta_{A_k}(\langle x_j, w\rangle)]^\beta}{\sum_{\langle z,u\rangle \in Nbrs_k} [\tau_{A_k}(\langle z, u\rangle)]^\alpha \cdot [\eta_{A_k}(\langle z, u\rangle)]^\beta} \tag{6}$$

Cascaded decision: In this alternative, the algorithm first selects a variable from those available in the feasible neighbourhood. This is done using a separate variable ordering heuristic (discussed below). Once the variable has been chosen, a value is chosen probabilistically from the variable's domain using the pheromone and heuristic factors. Formally, if ant k has built assignment A_k and has separately chosen variable x_j from those available in $Nbrs_k$, then the probability of choosing to bind $w \in D(x_j)$ to x_j is:

$$\mathrm{Prob}(\langle x_j, w\rangle) = \frac{[\tau_{A_k}(\langle x_j, w\rangle)]^\alpha \cdot [\eta_{A_k}(\langle x_j, w\rangle)]^\beta}{\sum_{u \in D(x_j)} [\tau_{A_k}(\langle x_j, u\rangle)]^\alpha \cdot [\eta_{A_k}(\langle x_j, u\rangle)]^\beta} \tag{7}$$

Constraint solvers deploy a variety of heuristics for variable ordering. These can be adopted by cascaded variants of the ant algorithm. Common variable ordering heuristics with which we experiment in this paper are:

Static-random: A random variable ordering is decided during initialisation. Each ant uses this same ordering in each cycle.

Dynamic-random: Each time a vertex must be chosen, the next variable is chosen randomly from those available in the feasible neighbourhood.

Max-static-degree: The next variable to be assigned will be the unassigned variable with the highest static degree (i.e. the highest number of variables to which it is connected by some constraint). Ties are broken arbitrarily, but always in the same way.

Min-static-degree: This is like max-static-degree but it chooses the lowest static degree.

Max-forward-degree: The next variable to be assigned will be the unassigned variable with the highest forward degree (i.e. the highest number of unassigned variables to which it is connected by some constraint). Ties are broken randomly.

Min-forward-degree: This is like max-forward-degree but it chooses the lowest forward degree.

Smallest-domain-first: The next variable to be assigned will be the unassigned variable with the least number of values in its domain which are consistent with the partial assignment built so far. Ties are broken randomly.

Table 1 summarises the algorithm variants that we have described.

Our basic ant algorithm is closely modelled on the one reported by Solnon in [Sol02]. In fact, the algorithm described in [Sol02] corresponds exactly with our variant EdgeSumsCascaded (with the smallest-domain-first heuristic).

Roli et al. briefly describe three ant algorithms for solving constraint problems [RBD01]. Their three algorithms correspond loosely with VertexCompound, EdgeCompound and EdgeSumsCompound. However, the correspondence is not exact for several reasons. First, they do not give any definition for the heuristic factor η_{A_k}. Second, they have a more complicated state transition rule which allows components with zero pheromone to be chosen randomly on occasion. Like [Sol02], our use of τ_{min}, where $0 < \tau_{min}$, guarantees that we have no components with zero pheromone.

	Pheromone deposited on	
	Vertices	Edges
Compound	VertexCompound – pheromone deposit: Equation 1 – pheromone factor: Equation 3 – vertex selection: Equation 6	EdgeCompound – pheromone deposit: Equation 2 – pheromone factor: Equation 4 – vertex selection: Equation 6 EdgeSumsCompound – pheromone deposit: Equation 2 – pheromone factor: Equation 5 – vertex selection: Equation 6
Cascaded	VertexCascaded – pheromone deposit: Equation 1 – pheromone factor: Equation 3 – vertex selection: Equation 7	EdgeCascaded – pheromone deposit: Equation 2 – pheromone factor: Equation 4 – vertex selection: Equation 7 EdgeSumsCascaded – pheromone deposit: Equation 2 – pheromone factor: Equation 5 – vertex selection: Equation 7

Table 1. Variants of the ant algorithm. (All variants define the heuristic factor η_{A_k} the same way.)

4 Experiments

4.1 Datasets

We compare the performance of the algorithm variants on randomly-generated problem instances. We generate only *binary CSPs*, i.e. ones in which all constraints involve exactly two variables. Classes of random binary CSPs can be described by giving four parameters, (n, m, p_d, p_t): n is the number of variables in each problem; m is the uniform domain size; p_d is a measure of problem *density*; and, p_t is a measure of problem *tightness*. We generate instances using what has come to be called Model B, in which both p_d and p_t are proportions rather than probabilities [SD96]. Because incomplete solvers cannot, in general, determine when a problem has no solution, they are often evaluated only on *solvable* CSP instances. This is the practice we follow here. We generate a solvable CSP by generating a random solution and then building random constraints around it, where no randomly generated constraint is admitted if the solution would violate it.

Our datasets are generated using parameters $(100, 8, 0.14, p_t)$. We generate 10 problems for $p_t = 0.2$, another 10 for $p_t = 0.25$ and a final 10 for $p_t = 0.3$. Since ant algorithms are stochastic, the results of applying a particular ant algorithm to a particular problem instance are averaged over multiple runs.

4.2 Parameters

Ant algorithms have a lot of parameters. Ideally, these should be systematically optimised before comparing different algorithms. This takes a very great deal of time. So, for the preliminary results that we publish here, we have adopted a single set of parameters based on those that Solnon found to be best [Sol02], viz. $\rho = 0.01$, $\alpha = 2$ and $\beta = 10$. We take $\tau_{min} = 0.01$ and $\tau_{max} = 4$. We use 8 ants and algorithms are run until either a solution is found (remember, all our problem instances have solutions) or until 1000 cycles have been completed.

4.3 Comparison of variants

We quickly found that the compound versions of the algorithms (upper row in Table 1) were impractically inefficient: if there are n unassigned variables, each having a domain of size m, then nm probabilities must be computed. By comparison, the cascaded algorithms, which choose variables separately, compute probabilities over much narrower sets of labels: no matter how many variables are unassigned, only one will be chosen by the variable ordering heuristic and then probabilities must be computed for just its m values. Empirically we found the cost of computing many probabilities to be greater than the cost of the separate variable ordering heuristic. For low tightness problem instances, for example, VertexCascaded, EdgeCascaded and EdgeSumsCascaded (using the dynamic random variable ordering heuristic) exhibited cycle times of 0.08, 0.09 and 0.14 seconds respectively; VertexCompound, EdgeCompound and EdgeSumsCompound had cycle times of 3.38, 3.5 and 5.37 seconds respectively.[1] We therefore excluded the compound algorithms from further experiments.

Table 2 gives the results for our three remaining variants. Each is a cascaded variant and, in each, the variable ordering heuristic is dynamic-random. Looking across the three sets of problems, VertexCascaded would appear to be best at finding solutions with low numbers of violated constraints. However, in some cases, EdgeCascaded and EdgeSumsCascaded find actual solutions when VertexCascaded does not. Unsurprisingly, VertexCascaded has the lowest cycle time, followed by EdgeCascaded and then EdgeSumsCascaded. However, for total time what matters is the number of cycles needed. Algorithms need not run for the full 1000 cycles: they stop early if they find solutions. As we have noted, EdgeCascaded and EdgeSumsCascaded sometimes do better at finding solutions than VertexCascaded and so they can be cheaper overall. These preliminary results are by no means clear-cut. One would not confidently follow Solnon [Sol02] and adopt EdgeSumsCascaded.

[1] Experiments are run on a Pentium 4 (1.7Ghz) with 256M RAM. Programs are coded in Java and are written by a single programmer (the first author) to, as far as possible, the same standard (which is made easier by the fact that they are variants of a single approach).

	Avg. cost of best soln. at termination	Std. dev.	Avg. time to termination	Avg. cycle time	Instances solved (out of 300)	Avg. # of cycles to soln. (when solved)
$(100, 8, 0.14, 0.2)$						
VertexCascaded	0.64	0.53	68.37	0.08	116	610.95
EdgeCascaded	0.03	0.16	44.54	0.09	292	492.37
EdgeSumsCascaded	0.02	0.14	66.87	0.14	294	458.95
$(100, 8, 0.14, 0.25)$						
VertexCascaded	2.71	1.70	90.92	0.10	46	473.39
EdgeCascaded	3.48	1.93	102.27	0.11	36	657.14
EdgeSumsCascaded	3.36	1.90	154.85	0.16	30	575.17
$(100, 8, 0.14, 0.3)$						
VertexCascaded	1.18	3.35	62.42	0.11	236	436.07
EdgeCascaded	2.10	4.42	89.52	0.12	206	628.90
EdgeSumsCascaded	2.25	4.27	128.54	0.17	196	598.39

Table 2. Comparing three variants, each using dynamic-random variable ordering. There are three datasets and, for each, results are averaged over 300 (10 problems $\times$ 30 runs on each problem). Times are in seconds.

4.4 Comparison of variable ordering heuristics

Solnon adopts the smallest-domain-first variable ordering heuristic [Sol02] but gives no reasons and no experimental support. We wanted to compare the different heuristics in our winning algorithm. Since our preliminary experiments (above) found no clear-cut winner, we decided instead to test the heuristics using EdgeSumsCascaded, as this gives consistency with Solnon. Table 3 reports our results for the 10 problems in which $p_t = 0.3$. We chose this dataset because in our previous experiment it contained both problems that EdgeSumsCascaded with dynamic-random variable ordering was able to solve and others that it was not able to solve.

Only dynamic-random and smallest-domain-first find assignments with low numbers of violated constraints within 1000 cycles, and they are the only two that are able to find actual solutions. Of these two, dynamic-random would seem to find better quality solutions more often.

The cost per cycle of dynamic-random is much the same as that of the static strategies. For this reason, it convincingly outperforms smallest-domain-first. Smallest-domain-first is a costly strategy since it requires that constraints be evaluated on all legal ways of extending the partial assignment with one more label. It is conceivable that, by using constraint propagation techniques, we could reduce the cost of this strategy (at the expense of memory) but our preliminary results give no strong motivation for doing so.

	Avg. cost of best soln. at termination	Std. dev.	Avg. time to termination	Avg. cycle time	Instances solved (out of 40)	Avg. # of cycles to soln. (when solved)
Static-random	29.63	5.70	173.18	0.17	0	N/A
Dynamic-random	1.03	2.35	122.63	0.17	29	593.59
Max-static-degree	23.80	4.93	176.45	0.18	0	N/A
Min-static-degree	33.68	5.96	173.29	0.17	0	N/A
Max-forward-degree	24.33	5.94	223.06	0.22	0	N/A
Min-forward-degree	26.45	5.89	234.45	0.23	0	N/A
Smallest-domain-first	3.73	6.19	1442.56	1.82	23	641.39

Table 3. Comparing different variable ordering heuristics. The algorithm used is EdgeSumsCascaded. There is one dataset, $(100, 8, 0.14, 0.3)$. Results are averaged over 40 (10 problems $\times$ 4 runs on each problem). Times are in seconds.

5 Conclusions

We presented six ant algorithms for constraint problems. We found three of them, the ones that made compound decisions, to be impractically inefficient. These three correspond (loosely) to the three that Roli et al. adopted [RBD01].

The remaining three algorithms all make cascaded decisions but have different pheromone placement policies. EdgeSumsCascaded is exactly the algorithm described by Solnon in [Sol02]. However, our preliminary results show no clear-cut winner. More experiments are needed before we could decide which of these three should be adopted.

Finally, we compared seven different variable ordering heuristics that can be used in cascaded systems. Solnon uses smallest-domain-first [Sol02]. Our results strongly favour the dynamic-random heuristic.

There is much future experimental work to be done; and there are avenues to explore such as the addition of local search daemons [Sol02,RBD01].

References

[DMC96] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics — Part B*, 26(1):1–13, 1996.

[RBD01] A. Roli, C. Blum, and M. Dorigo. ACO for maximal constraint satisfaction problems. In *Procs. of the 4th Metaheuristics International Conference*, pages 187–191, 2001.

[SD96] B. M. Smith and M. E. Dyer. Locating the phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 18(1-2):155–181, 1996.

[SH98] T. Stützle and H. Hoos. Improvements on the ant system: Introducing the $\mathcal{MAX}$-$\mathcal{MIN}$ ant system. In *Procs. of Artificial Neural Nets and Genetic Algorithms 1997*, pages 245–249, 1998.

[Sol02] C. Solnon. Ants can solve constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 6(4):347–357, 2002.