ACOhg: Dealing with Huge Graphs

Enrique Alba
GISUM Group, Dpto. Lenguajes y Ciencias de la Computación
E.T.S. Ingeniería Informática
University of Málaga, Spain
eat@lcc.uma.es

Francisco Chicano
GISUM Group, Dpto. Lenguajes y Ciencias de la Computación
E.T.S. Ingeniería Informática
University of Málaga, Spain
chicano@lcc.uma.es

ABSTRACT

Ant Colony Optimization (ACO) has been successfully applied to those combinatorial optimization problems which can be translated into a graph exploration. Artificial ants build solutions step by step adding solution components that are represented by graph nodes. The existing ACO algorithms are suitable when the graph is not very large (thousands of nodes) but is not useful when the graph size can be a challenge for the computer memory and cannot be completely generated or stored in it. In this paper we study a new ACO model that overcomes the difficulties found when working with a huge construction graph. In addition to the description of the model, we analyze in the experimental section one technique used for dealing with this huge graph exploration. The results of the analysis can help to understand the meaning of the new parameters introduced and to decide which parameterization is more suitable for a given problem. For the experiments we use one real problem with capital importance in Software Engineering: refutation of safety properties in concurrent systems. This way, we foster an innovative research line related to the application of ACO to formal methods in Software Engineering.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*; D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking*; G.1.6 [**Numerical Analysis**]: Optimization—*Global optimization*

General Terms

Algorithms, Experimentation, Verification

Keywords

Ant colony optimization, metaheuristics, SPIN

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

1. INTRODUCTION

Ant Colony Optimization is a kind of population based metaheuristic algorithm [2] whose foundation is based on the foraging behaviour of real ants [5]. The problems that can be solved with ACOs are those whose tentative solutions can be represented with a sequence of *components*. ACO models found in the literature for combinatorial optimization problems work over graphs with a known number of nodes that is small enough to store in memory the pheromone trails associated to the arcs (or nodes). In addition, these models consider that paths traversed by the ants have a known maximum length. These considerations of the traditional ACO models are also a limitation for the set of problems that can be solved with them. In particular, existing ACO models cannot be applied to problems having an underlying graph with an a priori unknown size and/or whose solutions are paths for which no small enough upper bound is known.

An example of this kind of problems is the refutation of safety properties in concurrent systems. The objective of this problem can be formulated as searching a path between an initial node and a node fulfilling a given condition (objective node). The graph size depends on the concurrent system model and usually it grows in an exponential way with respect to the system size. The number of nodes of the graph is usually unknown and the graph itself normally does not fit in memory. The objective nodes are also unknown and, thus, a useful upper bound for the paths length cannot be estimated[1]. Other two examples of this kind of problems are optimal movements in games and automatic theorem proving. These problems can be translated into an exploration of unknown huge graphs. The amount of possible states in a game (chess for example) is really high and usually it is impossible to store or explore all of them in order to decide the next move. With respect to the automatic theorem proving, some automatic provers break the theorem down into clauses and they use the resolution method in order to get an empty clause. On each step these provers have to select a clause among the available ones to apply the resolution rule. These clauses can be viewed as arcs that end in a new state in the proof. This way, the problem is defined as a search in an upper abstraction graph (the objective is to find a state including the empty clause). For this particular problem the number of states of the graph can be infinity.

[1]Actually, it can be estimated a theoretical upper bound for the number of nodes of the graph, namely, the product of the cardinalities of all variables domain. This is also an upper bound for the paths length, but it will usually be very far from the minimal upper bound and it will not be practical.

In this paper we study a new ACO model, ACOhg (ACO for huge graphs), that overcomes the limitations of the existing ones and can deal with graphs of unknown size and/or too big to fit in memory. For this reason, it can be applied to a larger set of combinatorial optimization problems. In particular, our model can be applied to the problem of refutation of safety properties in concurrent systems, which we detail in the experimental section.

The paper is organized as follows. Section 2 presents a brief overview of ACO algorithms. ACOhg model is detailed in Section 3. In Section 4 we show some results obtained with ACOhg. Finally, the conclusions and future work are depicted in Section 5.

2. BRIEF REVIEW OF ACO

A combinatorial optimization problem can be represented by a triplet (S, f, Ω), where S is the set of candidate solutions, f is the *fitness function* that assigns a real value to each candidate solution related to its quality, and Ω is a set of constraints that the final solution must fulfill. The objective is to find a solution minimizing or maximizing the function f (in the following we assume that we deal with minimization problems). A candidate solution is represented by a sequence of *components* chosen from a set C.

In ACO, there is a set of artificial ants (colony) that build the solutions using a stochastic constructive procedure. In the construction phase, ants walk randomly on a graph $G = (C, L)$ called *construction graph*, where L is the set of *connections* (arcs) among the components (nodes) of C. In general, the construction graph is fully connected (is complete), however, some of the problem constraints (elements of Ω) can be modelled by removing arcs from L. Each connection l_{ij} has an associated pheromone trail τ_{ij} and can also have an associated heuristic value η_{ij}. Both values are used to guide the stochastic construction phase that ants perform. However, pheromone trails are modified by the algorithm along the search whilst heuristic values are established from external sources (the designer). Pheromone trails can also be associated to graph nodes (solution components) instead of arcs (component connections). This variation is especially suitable for problems in which the order of the components is not relevant (e.g. subset problems [10]).

In Figure 1 we reproduce a general ACO pseudo-code found in [5]. It consists of three procedures executed during the search: `ConstructAntsSolutions`, `UpdatePheromones`, and `DaemonActions`. They are executed until a given stopping criterion is fulfilled, such as finding a solution or reaching a given number of steps. In the first procedure each artificial ant follows a path in the construction graph. The ant starts in an initial node and then it selects the next node according to the pheromone and the heuristic value associated with each arc (or the node itself). The ant appends the new node to the traversed path and selects the next node in the same way. This process is iterated until a candidate solution is built. In the `UpdatePheromones` procedure pheromone trails associated to arcs are modified. A particular pheromone trail value can increase if the corresponding arc has been traversed by an ant and it can decrease due to evaporation (a mechanism that avoids the premature convergence of the algorithm). The amount in which a pheromone trail is increased usually depends on the quality of the candidate solution built by the ants traversing the arc. Finally, the last (and optional) procedure `DaemonActions` performs centralized actions that are not performed by individual ants. For example, a local optimization algorithm can be implemented in this procedure in order to improve the tentative solution held in every ant.

```
procedure ACOMetaheuristic
    ScheduleActivities
        ConstructAntsSolutions
        UpdatePheromones
        DaemonActions // optional
    end ScheduleActivities
end procedure
```

Figure 1: Pseudo-code of ACO Metaheuristic.

2.1 ACO Details

The scheme presented above is very abstract and short. It is general enough to match with the different models of ACO algorithms we can find in the literature. These models differ in the way they schedule the three main procedures and in how they update pheromone trails. Some examples of these models are Ant Systems (AS), Elitist Ant Systems (EAS), Ranked-Based Ant Systems (Rank AS), Max-Min Ant Systems ($\mathcal{MM}$AS), and so on. The interested reader can see the book by Dorigo and Stützle [5] for a description of all these ACO variants. The new model ACOhg extends the existing ACO models by introducing new ideas and mechanisms for working with unknown and huge graphs. That is, an ACOhg algorithm can be based on any existing ACO algorithm such as AS, $\mathcal{MM}$AS, and so on, thus yielding the corresponding AShg, $\mathcal{MM}$AShg, etc. algorithms. The ACOhg algorithm used in this paper is based on an ACO algorithm that combines several features belonging to different of these existing models. In the rest of this section we explain the details of this underlying ACO algorithm.

2.1.1 Construction Phase

As we mentioned above, ants stochastically select the following node in the construction graph during the construction phase. In particular, when ant k is in node i it selects node j with probability

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \text{ if } j \in N_i, \quad (1)$$

where N_i is the set of successor nodes for node i, and α and β are two parameters of the algorithm determining the relative influence of the heuristic value and the pheromone trail on the path construction, respectively.

2.1.2 Pheromone Update

During the construction phase the pheromone trails associated to the arcs that ants traverse are updated using the expression

$$\tau_{ij} \leftarrow (1 - x_i)\tau_{ij} \quad (2)$$

where x_i controls the evaporation of the pheromone during the construction phase and it holds $0 \leq x_i \leq 1$. This mechanism increases the exploration of the algorithm, since it reduces the probability that an ant follows the path of a previous ant.

After the construction phase pheromone trails are updated again in order to take into account the quality of

the candidate solutions built by the ants. In this case the pheromone update follows the expression

$$\tau_{ij} \leftarrow \rho\tau_{ij} + \Delta\tau_{ij}^{bs},\ \forall(i,j) \in L, \tag{3}$$

where ρ is the *pheromone evaporation rate* and it holds that $0 \leq \rho \leq 1$. On the other hand, $\Delta\tau_{ij}^{bs}$ is the amount of pheromone that the best ant path ever found deposits on arc (i,j). This quantity is usually in direct relation with the quality of the solution. For example, in a maximization problem, it can be the fitness value of the solution. In a minimization problem (like ours) it can be the inverse of the fitness function.

2.1.3 Trail Limits

In $\mathcal{MM}$AS there is a mechanism to avoid the premature convergence of the algorithm. The idea is to keep the value of pheromone trails in a given interval $[\tau_{min}, \tau_{max}]$ in order to maintain the probability of selecting one node above a given threshold. We adopt here this idea. The values of the trail limits are

$$\tau_{max} = \frac{Q}{1-\rho} \tag{4}$$

$$\tau_{min} = \frac{\tau_{max}}{a} \tag{5}$$

where Q is the highest fitness value found if the problem is a maximization problem or the inverse of the minimum fitness if the problem is a minimization one. The parameter a controls the size of the interval.

When one pheromone trail is greater than τ_{max} it is set to τ_{max} and, in a similar way, when it is lower than τ_{min} it is set to τ_{min}. Each time a new better solution is found the interval limits are updated consequently and all pheromone trails are checked in order to keep them inside the interval.

3. THE NEW MODEL: ACOhg

The ACO models we found in the literature can be applied (and they have been) to problems with a number of nodes n of several thousands. In these problems the construction graph has a number of arcs of $O(n^2)$, that is, several millions of arcs, and hence the pheromone trails require several megabytes of memory in a computer to be stored. These models are not suitable in problems in which the construction graph has 10^6 nodes (i.e. 10^{12} arcs). They are also not suitable when the amount of nodes is not known beforehand and the nodes and arcs of the construction graph are dynamically generated as the search progresses.

Let us discuss the issues that prevent existing ACO models from solving such kind of problems. First, in the construction phase, ants of a regular ACO walk until a candidate solution is completed. However, if we would allow the ants to walk on the huge unknown graph without repeating a node until they find an objective node they can reach a dead end (a node without non-visited successors). Even although they find an objective node they can wander in the graph for a long time requiring a lot of memory to build a candidate solution since the objective nodes can be very far from the initial node. Thus, in general it is not viable to work with complete candidate solutions as current models do. We must allow the construction of partial candidate solutions. We want to stress that we are not dealing with an implementation detail, but with applications inherently having huge graphs. Second, some ACO models assign to the initial pheromone trails a value that depends on the number of graph nodes. This kind of initialization of the pheromone trails is not suitable when we work with unknown sized graphs. We must be also careful of course with the implementation of pheromone trails. In most ACO models pheromone trails are usually stored in arrays, but this requires to know the number of nodes. In our case, even if we would know that number we could not store pheromone trails in arrays due to the great amount of memory required (often not available).

The main ideas that ACOhg introduces are related to the length of the ant paths, the fitness function and the memory consumption of pheromone trails. We tackle these points in the following paragraphs.

3.1 Length of the Ant Paths

In order to avoid the, in general unviable, construction of complete candidate solutions we limit the length of the paths traversed by ants in the construction phase. That is, when the path of an ant reaches a given maximum length λ_{ant}, the ant is stopped. In this way, the construction phase can be performed in a bounded time and with a bounded amount of memory. However, the limitation of the ant path length implies that most (if not all) of the paths are partial solutions and therefore we need a fitness function that can evaluate partial solutions.

The limitation in the ant path length solves the problem of the "wandering ants" but introduces a new one. There is a new parameter for the algorithm (λ_{ant}) whose optimal value is not easy to establish a priori. If we select a value smaller than the depth[2] of all the objective nodes, the algorithm will not find any solution to the problem. Thus, we must select a value larger than the depth of one objective node (if known). This is not difficult when we know where the objective node is, but the usual situation is the opposite one. In the last case, two alternatives are proposed. In Figure 2 we show graphically the way in which the two alternatives work.

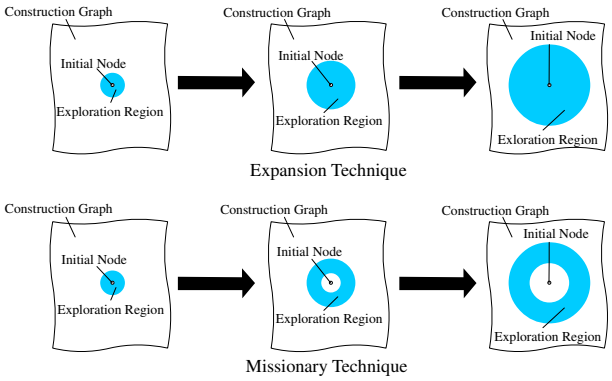

Figure 2: Two alternatives for reaching objective nodes of unknown depth: expansion and missionary techniques. We show snapshots of different moments of the search.

The first consists in dynamically increasing λ_{ant} during the search if no objective node is found. At the beginning,

[2]The *depth* of a node in the construction graph is the length of the shortest path from an initial node to it.

a low value is assigned to λ_{ant} and it is increased in a given quantity δ_l every certain number of steps σ_i. In this way, the length will be hopefully high enough to reach at least one objective node. This is called *expansion technique*. This mechanism can be useful when the depth of the objective nodes is not very high. Otherwise, the length of the ant paths will increase a lot and the same happens with the time and the memory required to build the paths, since it will approach the behaviour of a regular ACO incrementally.

The second alternative consists in starting the path construction of the ants from different nodes during the search. That is, at the beginning the ants are placed on the initial nodes of the graph and the algorithm is executed during a given number of steps σ_s (called *stage*). If no objective node is found, the last nodes of the paths constructed by the ants are used as starting nodes for the next ants. In the next steps (the second stage) of the algorithm the new ants traverse the graph starting in the last nodes of paths computed in the first stage. In this way, the new ants start at the end of previous ant paths trying to go beyond in the graph. This mechanism is called *missionary technique*. The length of the ant paths (λ_{ant}) is kept always constant and the pheromone trails can be discarded from one stage to another in order to keep almost constant the amount of computational resources (memory and CPU time) in all the stages. The assignment of ants to starting nodes at the beginning of one stage is performed in several phases. First, we need to select the paths of the previous stage whose last visited nodes will be used as starting points in the new stage. For this, we store the best paths (according to the fitness value) found in the previous stage. We denote with s the number of paths stored. Once we have the set of starting nodes we need to assign the new ants to those nodes. For each new ant we select its starting node using roulette selection; that is, the probability of selecting one node is proportional to the fitness value of the solution associated with it.

3.2 Fitness Function

The objective of ACOhg is to find a low cost path between an initial node and an objective one. For the problem that we solve in the experimental section, the cost of a solution is its length, but, in general, cost and length (number of components) of a solution can be different, and for this reason it is safer to talk about cost. Considering a minimization problem, the fitness function of a complete solution can be the cost of the solution. However, as we said above, the fitness function must be able to evaluate partial solutions. In this case the partial solution cost is not a suitable fitness value since a low cost partial solution can be considered better than one high cost complete solution. This means that low cost partial solutions are awarded and the fitness function will not suitably represent the quality of the solutions. In order to avoid this problem we penalize the partial solutions by adding a fixed quantity p_p to the cost of the solutions.

Another way of solving the problem of awarded partial solutions consists in changing the fitness function in such a way that it be a lower bound of the fitness value of a complete solution that is an extension of the partial solution. If the estimation is very precise the fitness function will be a good measure of the solution quality. However, in some problems it is not easy to get a precise estimation of this lower bound. The alternative of adding a penalty value is general to all the combinatorial problems.

When the cost of a solution (partial or complete) grows with its length, we can add an additional term to the solution cost that can help in the search. During the construction phase, when an ant builds its path, it can stop due to three reasons: the maximum ant length λ_{ant} is reached, the last node of the ant path is an objective node, or the following nodes are in the ant path (visited nodes). This last condition, which is used in order to avoid the construction of paths with cycles, has an undesirable side effect: it awards paths that form a cycle. In effect, this mechanism favors ants with short paths and, consequently, with lower cost and fitness values. In order to avoid this situation we penalize those partial solutions whose path length is shorter than λ_{ant}. The total penalty expression is

$$p = p_p + p_c \frac{\lambda_{ant} - l}{\lambda_{ant} - 1} \tag{6}$$

where p_p is the penalty due to the incompleteness of the solutions, p_c is a penalty constant related to the cycle formation, and l is the ant path length. The second term in (6) makes the penalty higher in shorter cycles. The intuition behind this is that longer cycles are nearer of a path without a cycle. For this reason we add to p_p the maximum cycle penalty (p_c) when the ant length is the minimum ($l = 1$) and no cycle penalty is added when there is no cycle ($l = \lambda_{ant}$).

3.3 Pheromone Trails

In ACOhg, the pheromone trails are stored in a hash table where only the pheromone values of the edges traversed by the ants are stored. However, as the search progresses the memory required by pheromone trails can increase until inadmissible values. We can avoid this by removing from the hash table the pheromone trails with a low influence on the ant construction, that is, the values τ_{ij} with a low associated pheromone trail. We define τ_θ as the threshold value for removing pheromone trails. All the values τ_{ij} below τ_θ are removed from the hash table.

The removing step can be applied when some condition is fulfilled, i.e., when the free memory is below a given threshold or a predefined number of iterations has been reached since the last pheromone removing step. It can also be applied continuously each time a pheromone trail is updated. That is, the pheromone trails below τ_θ are automatically removed in any update step. This way, the search for low pheromone trails in the hash table (that can be a very time consuming task) is avoided.

4. EXPERIMENTAL SECTION

In this section we show some experimental results obtained with ACOhg in order to study its behaviour. In particular, we apply ACOhg to the problem of refutation of safety properties in concurrent systems (see details below). This problem is of a great interest in software engineering and theoretical computer science. We use it because the application of ACOhg to this problem is innovative and the results are very promising from the domain point of view, outperforming the results obtained by the state-of-the-art techniques in model checking.

In the following sections we first introduce the problem of refutation of safety properties in concurrent systems, then we give some details about the parameters used in the experiments. Next, we compare ACOhg against exact algorithms

previously used for this problem and, finally, we empirically analyze the missionary technique mentioned in Section 3.1.

4.1 Systems Verification

From the very beginning of computer research, computer engineers have been interested in techniques allowing them to know if a software module fulfills a set of requirements (its specification). Modern software is very complex and these techniques have become a necessity in most software companies. One of these techniques is *model checking* [3], which consists in analyzing (in a direct or indirect way) all the possible states of a concurrent system in order to prove or refute that the program satisfies a given property. This property is specified using a temporal logic like Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). One of the best known model checkers is SPIN [9], which takes a software model codified in Promela and a property specified in LTL as inputs. SPIN transforms the model and the negation of the LTL formula into Büchi automata in order to perform the synchronous product of them. The resulting product automaton is explored to search for a cycle of states containing an accepting state reachable from the initial state. If such a cycle is found, then there exists at least one execution of the system not fulfilling the LTL property (see [9] for more details). If such kind of cycle does not exist then the system fulfills the property and the verification ends with success. The main drawback of a model checking approach is the so-called state explosion: when the size of the program increases, the amount of required memory also increases but in an exponential way. Even for really small programs the number of states is very large. This phenomenon limits the size of the models to be checked.

The properties that can be specified with LTL formulae can be classified into two groups: *safety* and *liveness* properties [12]. Safety properties can be expressed as assertions that must be fulfilled by all the states of the model, while liveness properties refer to assertions that must be fulfilled by execution paths in the model. Safety properties of a model can be checked by searching for a single accepting state in the product Büchi automaton. That is, when safety properties are checked, it is not required to find an additional cycle containing an accepting state. This means that safety properties verification can be transformed into a search for one objective node (one accepting state) in a graph. Furthermore, the path from one initial node to an objective node represents an execution of the concurrent system in which the given safety property is violated.

This fact has been used in previous works to verify safety properties using classical algorithms in the graph exploration domain, such as depth first search, breadth first search, A* or Weighted A* [7]. The main drawback of an exact algorithm is that it requires a lot of memory and it can need a lot of time to get an error trail in a big concurrent system. In these situations metaheuristic algorithms have proved to be very effective finding good quality solutions in a reasonable time. In this sense, genetic algorithms have been applied to this problem in the past [1, 8]. If the algorithm is able to find a path to an objective node the property is refuted and the path is a counterexample, but verifying that the system has a property requires the exploration of all the possible paths in order to ensure that there is no objective node. Most of the canonical metaheuristic algorithms, due to their approximate nature, cannot ensure that the system fulfills the property, but they can refute it. For this reason we talk about a problem of properties refutation instead of verification. In order to guide the search, one heuristic value is associated to each automaton state. This value is a lower bound of the distance to an objective node. The computation of this value can be based on the LTL formula [6] or on the objective node (if it is known beforehand) [11].

For the experiments we seek to find deadlock states in the Edsger Dijkstra Dining Philosophers problem. We use this model because it is simple and scalable. Thus, we can use a version of the model as large as we want. Its simplicity allows us to study it from a theoretical point of view. The version with n philosophers has 3^n states and only one deadlock state. Furthermore, the optimal error trail has length $n+1$. For all the following experiments we use $n = 20$ philosophers. This means a graph with approximately $3.5 \cdot 10^9$ nodes where there is only one objective node.

4.2 Algorithms and Parameters

It is worth mentioning that a traditional ACO algorithm cannot be even implemented because the construction graph has $3.5 \cdot 10^9$ nodes and it would be necessary memory enough to store up to $1.2 \cdot 10^{19}$ pheromone trails ($9.1 \cdot 10^{10}$ GB of memory), very far from the current computers capacity. For the experiments we use an ACOhg algorithm with the base configuration shown in Table 1. The missionary technique is used and the technique of removing useless pheromone trails is not enabled (that is, $\tau_\theta = -\infty$).

Table 1: Parameters for ACOhg

Parameter	Value
Steps	10
Colony size	5
Ant path length (λ_{ant})	10
Steps per stage (σ_s)	2
Stored paths (s)	10
Constr. phase evaporation (x_i)	0.8
Trail limit control (a)	5
Pheromone evaporation rate(ρ)	0.4
Pheromone value exponent (α)	1.0
Heuristic value exponent (β)	1.0
Partial solution penalty (p_p)	1000
Cyclic solution penalty (p_c)	1000

These parameters are not set in an arbitrary way, they are the result of a previous study aimed at finding the best configuration for tackling the Dining Philosophers problem. That is, we performed a factorial experimental design using a set of values for each parameter and we selected the configuration for which the algorithm obtains the best trade-off between efficacy and quality of solution. The cost of a solution is the length of the path, and due to this we use $p_c > 0$ as recommended in Section 3.2. In this problem, obtaining a complete solution is a difficult task which has interest by itself since it constitutes a counterexample (error trail) of the property checked. For this reason, the stop criterion used in our ACOhg algorithm is to find a complete solution or to reach the maximum number of allowed steps (10). We are not interested here in optimizing the solution length, finding it is harder enough. With respect to the heuristic information, we use $\eta_{ij} = 1/(1 + H_{ap}(j))$, where $H_{ap}(j)$ is the number of active processes in state j.

Since ACOhg is an stochastic algorithm, we need to perform several independent runs in order to get a well-founded conclusion of the behaviour of the algorithm. In the specialized literature it is well established that a minimum of 30 independent runs is required to get statistical confidence of the results [4]. In our experiments we perform 100 independent runs in order to get a high statistical confidence. The machine used for the experiments is a Pentium IV at 2.8GHz with 512 MB of RAM.

4.3 Comparison against Exact Techniques

In this section we briefly show the results obtained with ACOhg using the parameterization shown in Table 1 and we compare them against the results obtained with exact algorithms that are the state-of-the-art in model checking and can be found in most model checkers. In Table 2 we show the hit rate[3], the length of the solutions (quality), the maximum memory used, and the CPU time of ACOhg, Depth First Search (DFS), Breadth First Search (BFS), and Best First Search (BF). For the exact algorithms only one execution is performed since they are deterministic algorithms.

Table 2: Comparison of ACOhg against exact algorithms

	Hit (%)	Length	Mem. (KB)	Time (ms)
ACOhg	64	35.88	8467.06	271.56
DFS	0	-	-	-
BFS	0	-	-	-
BF	100	101.00	15360.00	60.00

The first observation is that ACOhg finds an error trail in 64 out of the 100 independent runs. This contrast with the classical algorithm used in SPIN for this problem (DFS), which is not able to find any error trail in the model because it requires more than 512 MB. If we compare the results of ACOhg against the other exact algorithms (recently applied to this problem by Edelkamp et al. [7]), we observe that ACOhg outperforms the results of BFS (that is not able to find errors) and BF. Although BF always finds an error trail, ACOhg finds almost 3 times shorter (better) error trails using only half of the memory required by BF. Although a serious study on this topic is required, we can say that ACOhg is a promising algorithm for the problem of refuting safety properties in concurrent systems.

4.4 Analyzing the Missionary Technique

As we mentioned in Section 3.1, there are three parameters that govern this technique: s, σ_s, and λ_{ant}. We try different values for these parameters in order to investigate their influence on the results. Due to room problems we cannot show all the results, so we select the more interesting ones, that is, those in which s is kept fixed to 10 (as in the base configuration) and σ_s, λ_{ant} are changed. The hit rate, the length of the solutions, the maximum memory used, and the CPU time are shown in Tables 3, 4, 5, and 6, respectively (average values over the 100 independent runs).

From the results in Table 3 we conclude that the hit rate is higher when the number of steps per stage σ_s is small, that is, when more stages are performed. This was expected, because in this way ants can reach deeper nodes in the graph and they can find more paths reaching the objective node.

[3]Hit rate is defined as the percentage of runs in which a complete solution is found

Table 3: Missionary technique: hit rate

	λ_{ant}				
σ_s	**5**	**10**	**15**	**20**	**25**
1	38	91	99	100	100
2	10	64	95	99	100
3	0	41	89	99	100
4	0	39	84	98	100
5	0	0	63	84	99
6	0	0	61	85	97
7	0	0	51	84	96
8	0	0	40	76	95
9	0	0	17	53	82
10	0	0	0	0	60

We also observe that the hit rate increases with λ_{ant} due to the same reason.

Table 4: Missionary technique: length

	λ_{ant}				
σ_s	**5**	**10**	**15**	**20**	**25**
1	36.58	51.73	56.64	58.20	55.28
2	22.60	35.88	41.84	41.57	42.20
3	-	26.95	32.33	35.10	34.36
4	-	25.31	28.90	31.08	33.96
5	-	-	24.68	28.19	30.98
6	-	-	23.75	29.05	30.44
7	-	-	25.31	28.57	27.79
8	-	-	24.80	27.95	27.99
9	-	-	24.76	26.58	27.63
10	-	-	-	-	22.87

Concerning the length of error trails (Table 4) we observe that it increases when a large number of stages are performed (σ_s small) and when λ_{ant} is large. A statistical test (not shown) supports this conclusion. A large number of stages implies the exploration of deeper nodes in the construction graph and, thus, longer paths to the objective node are found with higher probability. The same happens when λ_{ant} is increased. In general, small values of σ_s and large values of λ_{ant} imply higher hit rate but longer error trails. A user of these techniques must select a value for these parameters in order to find the desired trade-off between efficacy and quality of solution.

Table 5: Missionary technique: max. memory (KB)

	λ_{ant}				
σ_s	**5**	**10**	**15**	**20**	**25**
1	4016.89	6436.65	8310.70	9719.76	10280.96
2	5507.40	8467.06	11210.11	13229.25	14399.85
3	-	10364.88	15118.38	18245.82	19191.94
4	-	13180.72	18834.29	23071.35	24465.29
5	-	-	22641.78	27928.38	28289.27
6	-	-	26523.28	32635.48	31614.64
7	-	-	30378.67	37388.19	33507.68
8	-	-	34124.80	42213.05	37339.66
9	-	-	37827.76	46930.11	40800.35
10	-	-	-	-	32902.08

If we take a look to the amount of memory required by the algorithm we notice that it increases with σ_s. That is, as in the hit rate, small values of σ_s are preferred in order to get small values in the amount of required resources. A statistical test supports this statement. The explanation is

related to the memory required in the first stage. We show a trace of the required memory along the different steps of the algorithm in Figures 3 and 4. During one stage the amount of memory required increases linearly. When the next stage begins, the pheromone trails are discarded and the memory fall down to a small value. We can observe this on the figures. The slope of the memory consumption is higher in the first stage, because the number of paths starting from the initial node is higher than the one found in other nodes. This means that the maximum memory required by the algorithm is the memory used in the first stage and this value grows with σ_s. On the other hand, when λ_{ant} increases the same happens with the increment of memory per step. For this reason the slope of the curve is larger for $\lambda_{ant} = 20$ (4765) than for $\lambda_{ant} = 10$ (2417). Thus, the memory required in the fist stage, which is also the maximum required memory, increases with λ_{ant}.

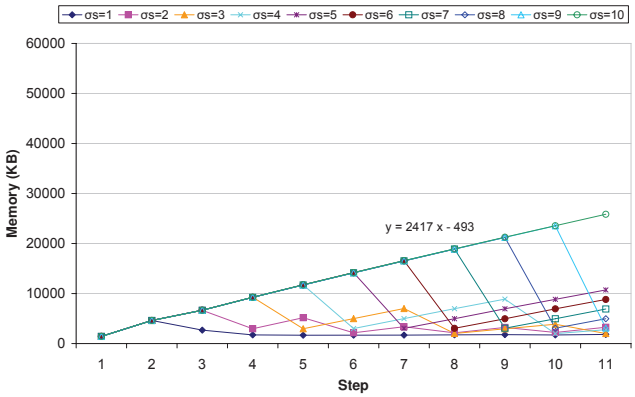

Figure 3: Trace of memory consumption ($\lambda_{ant} = 10$).

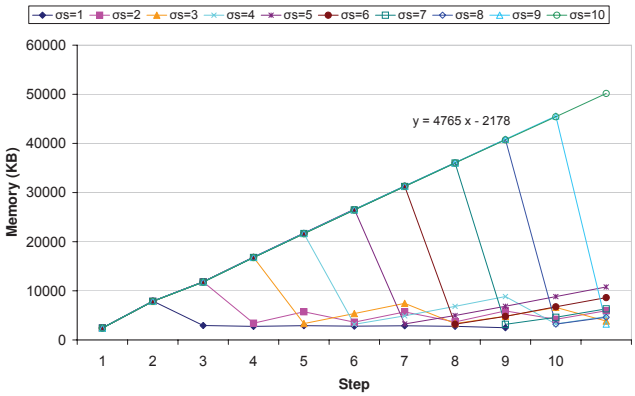

Figure 4: Trace of memory consumption ($\lambda_{ant} = 20$).

Concerning the CPU time, the influence of σ_s follows a clear trend. When σ_s is small the probability of finding a solution is higher and it is found faster. The only exception is that of $\sigma_s = 10$, explained by the low hit rate obtained, which reveals that more time is required to find a complete solution. With respect to the influence of λ_{ant}, there are two opposite trends. On one hand, when λ_{ant} increases more time is required to build the ant paths. This can be observed in columns 1 to 4 of Table 6. But, in this case, the probability of finding a solution is higher and the average

Table 6: Missionary technique: CPU time (ms)

	λ_{ant}				
σ_s	**5**	**10**	**15**	**20**	**25**
1	96.05	147.03	176.46	207.60	221.40
2	168.00	271.56	321.16	382.42	395.30
3	-	381.95	470.79	580.20	565.50
4	-	520.51	653.81	810.20	820.40
5	-	-	837.14	1093.81	1023.54
6	-	-	1070.00	1406.47	1258.66
7	-	-	1317.45	1741.55	1411.46
8	-	-	1578.75	2135.39	1653.47
9	-	-	1872.35	2555.66	1967.20
10	-	-	-	-	953.67

time required to find it is reduced. Although this is shown in the 5th column of Table 6, a statistical test reveals that the differences are not significant, so we conclude that the CPU time required increases with λ_{ant}.

In conclusion, from a practical point of view we can state the following: if an error trail is required fast and/or using a small amount of memory, then a small value of σ_s must be used; but if a short error trail is preferred, a large value must be assigned to σ_s. With respect to λ_{ant}, a large value can increase the probability of finding a solution but it also increases the amount of resources required.

Finally, we want to check whether the pheromone reset among stages has influence on the results. In order to check this we compare one version of the algorithm in which the pheromone reset is performed against one in which no reset is performed. We set $\lambda_{ant} = 25$ to get high hit rate and $s = 2$ to get good solutions (previous results not shown in this paper point out that a low value of s is preferable for obtaining short error trails). The hit rate, the length of the solutions, and the memory required are shown in Table 7 (average values over 100 independent runs).

Table 7: Analysis of the pheromone reset

	No reset			**Reset**		
σ_s	**Hit**	**Len.**	**Mem.(KB)**	**Hit**	**Len**	**Mem.(KB)**
1	100	43.68	11519.17	100	41.52	10112.95
2	100	33.88	16555.47	100	35.72	14171.23
3	100	31.76	20146.70	100	33.56	19660.60
4	100	28.60	24442.83	100	30.00	23982.08
5	100	27.00	28595.77	99	29.85	27962.86
6	100	27.68	33315.26	100	28.88	30155.75
7	99	27.10	36525.46	100	28.20	35313.72
8	99	26.37	38855.26	97	27.23	40044.09
9	93	25.52	39883.61	95	26.64	40853.95
10	50	23.24	30753.10	56	23.07	31547.95

We can observe in the table that the fact of discarding the pheromone trails after one stage has only a negligible influence on the hit rate or the solution length. There are only three cases in which the difference of the average lengths is statistically significant: when σ_s takes values 3, 5, and 9. In these cases, the best results (lower length) are obtained when pheromone trails are not discarded (no reset) between the stages. The reason is that pheromone trails of previous stages can help sometimes the search in the following ones. We can observe also a slight influence on the required memory, which is lower when the pheromone trails are discarded at the end of the stages. This difference is statistically significant for $\sigma_s \leq 7$. However, the extra required memory when

no reset is performed is not very large. This fact supports again the observation that most of the memory consumption is performed in the first stage. In order to illustrate this we show in Figures 5 and 6 the trace of the memory required on each step when the reset is performed and when it is not, respectively. In Figure 5 we observe that the slope of the curve decreases when the algorithm begins the second stage in all the cases.

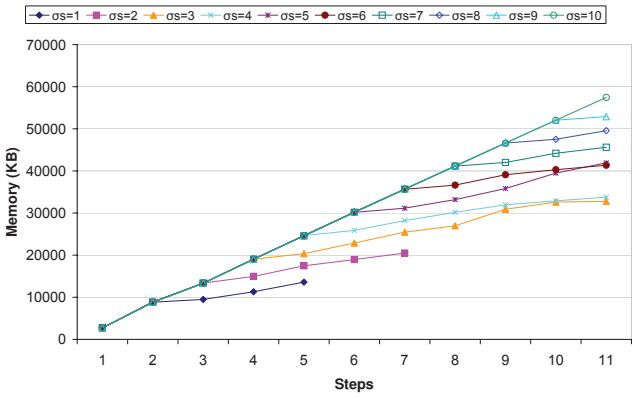

Figure 5: Trace of memory consumption without pheromone reset.

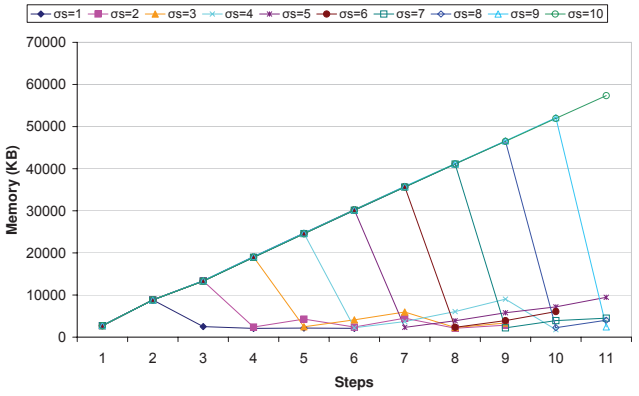

Figure 6: Trace of memory consumption with pheromone reset.

5. CONCLUSIONS AND FUTURE WORK

In this paper we describe a new ACO model, ACOhg, that can solve problems with a huge underlying graph constructed during the search. This model overcomes the limitations that other ACO models have and that prevent them from working with this kind of problems.

In order to illustrate the behaviour of the model we apply it to a problem with a great interest in software engineering: the refutation of safety properties in concurrent systems. We studied in depth one of the techniques used by ACOhg for exploring the unknown graph: the missionary technique. The results show that hit rate increases with the length of ant paths and with the frequency of the stages. The same happens with the average length of the error trails found. In addition, memory and CPU time required decrease when the frequency of the stages increases.

This paper is part of a recently open research line. We need to study the behaviour of the new ACO model. We have to explore the different alternatives mentioned in the paper and study their advantages and drawbacks in order to fast identify which is the best option depending on the problem at hand. In addition, we can apply the ideas presented on Section 3 to other metaheuristic algorithms: although they were thought for ACO, these ideas are extensible to other algorithms. The application of metaheuristic algorithms to the formal methods domain in software engineering and, in particular, to formal verification has not yet been extensively explored. Previous work on this topic is scarce and based only on genetic algorithms.

6. ACKNOWLEDGEMENTS

This work has been partially funded by the Ministry of Education and Science and FEDER (contract TIN2005-08818-C04-01, OPLINK project). Francisco Chicano is supported by a grant (BOJA 68/2003) from the Junta de Andalucía.

7. REFERENCES

[1] E. Alba and J. Troya. Genetic algorithms for protocol validation. In *Proc. of the PPSN IV International Conference (LNCS 1141)*, pages 870–879, Berlin, 1996. Springer.

[2] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, January 2000.

[4] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

[5] M. Dorigo and T. Stützle. *Ant Colony Optimization*. The MIT Press, 2004.

[6] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation Intelligence*, pages 75–83, 2001.

[7] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal of Software Tools for Technology Transfer*, 5:247–267, 2004.

[8] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *LNCS, 2280*, pages 266–280. Springer, 2002.

[9] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.

[10] G. Leguizamón and Z. Michalewicz. A new version of Ant System for subset problems. In P. A. et al., editor, *Proc. of the 1999 Congress on Evolutionary Computation*, pages 1459–1464, Piscataway, New Jersey, USA, 1999. IEEE Computer Society Press.

[11] A. Lluch-Lafuente. Symmetry reduction and heuristic search for error detection in model checking. In *Workshop on Model Checking and Artificial Intelligence*, August 2003.

[12] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.