Málaga, 10 de Diciembre de 2009

Informe Ejecutivo

TÍTULO: ACO-2.0-2009: Estudio de Parámetros de ACOhg

RESUMEN: En este documento presentamos un estudio de parámetros para el algoritmo ACOhg. Entre otros parámetros analizamos la longitud máxima de los caminos de las hormigas, el número de soluciones guardadas en la técnicas misionera, el número de iteraciones antes de avanzar en el grafo, las penalizaciones aplicadas a la función de fitness y los parámetros relacionados con la información heurística. Además, comparamos las técnicas misionera y de expansión.

OBJETIVOS:

1. Presentar un estudio detallado de los parámetros del algoritmo ACOhg.
2. Crear guías para el ajuste de parámetros.

CONCLUSIONES:

1. Cuanto mayor es la longitud máxima del camino que las hormigas pueden construir mayor es la probabilidad de encontrar un error existente en el modelo.
2. En la técnica misionera, cuanto menor sea el número de pasos entre etapas mayor es la probabilidad de encontrar una traza en menos tiempo y usnado menos memoria.
3. La técnica misionera es en la mayoría de los casos mejor opción que la técnica de expansión.
4. La escalabilidad del algortimo para el problema de búsqueda de errores en software concurrente es algo más que lineal pero sin llegar a ser exponencial.
5. Las penalizaciones aplicadas a la función de fitness no tienen efecto sobre los resultados.

RELACIÓN CON ENTREGABLES:

PRE: SOFTW-1.0-2008 (lectura necesaria)

PRE: ACO-1.0-2008 (lectura necesaria)

Málaga, December 10th, 2009

Executive Summary

TITLE: ACO-2.0-2009: Parameter Tuning for ACOhg

ABSTRACT: In this deliverable we present a parameter study for the ACOhg algorithm. We analyze the influence on the results of the maximum ant path length, the number of saved solutions in the missionary technique, the number of iterations per stage in both the missionary and the expansion techniques, the penalties used in the fitness function and the parameters related to the heuristic information used during the search. In addition, we compare the missionary and expansion techniques.

GOALS:

1. Present a detailed parameter analysis for ACOhg.
2. Design guides for parameter tuning.

CONCLUSIONS:

1. The longer the maximum length of the ant paths, the higher the probability of finding an existing error in the concurrent model.
2. In the missionary technique, the lower the number of steps per stage, the higher the probability of finding an error trail in a short time and using a low amount of memory.
3. The missionary technique is in most of the cases the best option compared to the expansion technique
4. The scalability of the algorithm for the problem of finding errors in concurrent software is more more than lineal but less than exponential.
5. The penalties applied to the fitness function have no influence on the results.

RELATION WITH DELIVERABLES:

PRE: SOFTW-1.0-2008 (mandatory reading)

PRE: ACO-1.0-2008 (mandatory reading)

Parameter Tuning for ACOhg

DIRICOM

December 2009

1 Introduction

Ant Colony Optimization is a kind of population based metaheuristic algorithm whose foundation is based on the foraging behaviour of real ants [6]. The main idea consists in using artificial ants that simulate the real ants' behaviour in a artificial scenario: a graph. Artificial ants are placed in initial nodes of the graph and they traverse it by jumping from one node to another one in order to find the shortest path between one initial node to an objective node. Each ant traverses the graph in an independent way with respect to the remaining ants. The following node to visit depends on certain numeric values associated to the arcs or nodes of the graphs. These values model the *pheromone trails* that real ants deposit during the walk. Artificial ants update (as real ants do) the (artificial) pheromone trails of the traversed path, in such a way that the walk of one ant can influence in the path of another one. This way, there is a cooperation mechanism among the ants that constitutes a key factor in the search [12].

In order to solve a problem with ACOs it must be translated into a minimal path graph search problem. In some problems this issue is trivial because the solutions are in fact a sequence of elements. This is the case of the TSP where one solution is a sequence of cities [11]. For other problems, such as Neural Network Training, the representation is not so direct [23]. In general, 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.

One 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 solutions length can not be estimated[1]. In order to solve this problem deterministic classic algorithms for graph exploration have been used such as depth first search, breadth first search, A*, and so on. The main drawback of these exact algorithms is that they require a lot of memory and they 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. ACOs are the metaheuristic algorithms that can be applied to this problem in a natural way. However, the ACO models found in the literature have the limitation above mentioned and this makes impossible their application.

In this deliverable we study a new ACO model without the limitations of the existing models: ACOhg. This new model can be applied with graphs of unknown size and/or too big to fit in memory. ACOhg does not require a maximum length for the solutions. 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, as we can see in the experimental section.

The deliverable is organized as follows. Section 2 presents a brief overview of ACO algorithms. Our model is detailed in Section 3. In Section 4 we show how to apply the proposed model and we analyze the influence on the results of the new parameters. 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

[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 solutions length, but it will usually be very far from the minimal upper bound y it will not be practical.

a set of constraints that the final solution must fulfil. The objective is to find a solution minimizing or maximizing the function f (in the following we assume that we deal with minimization problems). In ACO the candidate solutions are represented by a sequence of *components* chosen from a set of components C. In fact, an important issue when solving an optimization problem with an ACO algorithm is the selection of the set of components C and the way in which the solutions are built using these components. In some problems this issue is trivial because the solutions are in fact a sequence of elements. This is the case of the TSP where one solution is a sequence of cities. For other problems, such as Neural Network Training, the representation is not so direct [23].

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 some arcs of 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 the 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 [20]).

In Figure 1 we reproduce a general ACO pseudo-code found in [12]. It consists of three procedures that are executed during the search: `ConstructAntsSolutions`, `UpdatePheromones`, and `DaemonActions`. They are executed until a given stopping criterion is fulfilled, such as find a solution or reach 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 the pheromone trails associated to the 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 the 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 the ACO Metaheuristic.

2.1 ACO Details

The scheme presented in above is very abstract and small. 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 the 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 [12] for a description of all these ACO variants. In this deliverable we use an ACO variant that combines several features belonging to different models. In the following we explain the details of our variant.

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

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

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).

In order to solve the obstacles that arise when working with large graphs, Alba and Chicano [4, 2] proposed a new ACO model called ACOhg (ACO for huge graphs) that is able to tackle problems with an underlying construction graph of unknown size. The main issues we have to solve are related to the length of the ant paths, the memory consumption of pheromone trails, and the construction graph. 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.

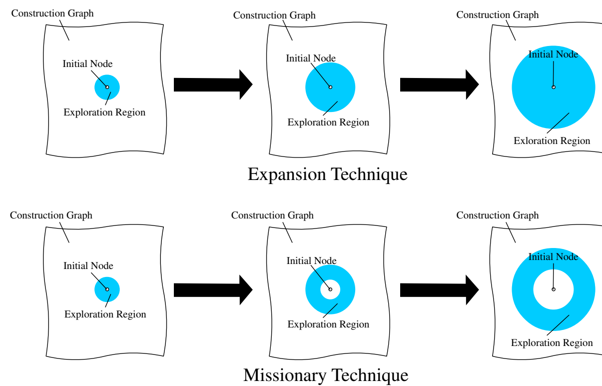

Figure 2: Two alternatives for reaching objective nodes of unknown depth: expansion and missionary techniques. We show snapshots of different moments of the search.

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

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

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 such solutions.

Another way of solving the problem of wrongly awarded partial solutions consists in changing the fitness function in such a way that it becomes an optimistic estimation 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 the best complete solution that can be built from a partial solution. The alternative of adding a penalty value is very common in solving 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 all the successors 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 which form a cycle. In effect, this mechanism favours ants with short paths and, consequently, with lower cost and low 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 we use for taking into account the above discussed issues 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. This is true for both, expansion and missionary techniques. 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 of low pheromone trails in the hash table (that can be a very time consuming task) is avoided.

When the missionary technique is used, we also can discard all the pheromone trails when a new stage begins. In a new stage, the region of the graph explored by the ants is different with respect to previous stages[3]. This means that pheromone trails used in previous stages are not useful in the current one and can be discarded without a negative influence on the results.

3.4 Construction Graph

One last issue to take into account in ACOhg is the generation of the construction graph. This is generated during the search itself, and this means that more memory is required as the search progresses. We need to be careful with the implementation in order to save memory. For example, if the size of one node in memory is larger than one pointer we can reduce the memory required by keeping only one copy of the node in memory and using pointers wherever is required. In this case we have to avoid duplicated nodes. However, if the node size is equal to or less than one pointer the previous mechanism is not useful and makes the implementation inefficient.

On the other hand, the only nodes required by the algorithm are those involved in an edge with an associated pheromone value, the ones appearing in the stored solutions (best solutions), and the ones referenced by ants. When one node is not referenced any more it can be removed from memory. This can save a lot of memory, specially if it is combined with the removing of pheromone trails.

The above mentioned ideas are mechanisms for memory reduction that are used to maximize the portion of the graph that can be stored in memory, since is in this portion where the algorithm can work with high efficacy to maximize the quality of the solutions. These mechanisms can be complemented with other techniques for memory reduction such as state compression or bitstate hashing.

[3] It is possible to find some overlap among the regions explored in different stages, but this overlap can be very small.

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 [3].

Although it is not our purpose here to make a list of possible problems to which the algorithm can be applied we briefly describe two of them in the following: optimal movements in games and automatic theorem proving. These problems can be translated into an exploration of unknown or 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 theorem 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 method. These clauses can be viewed as arcs that end in a new state in the proof. This way, the problem can be defined as a search in a graph (the objective is to find an state including the empty clause). For this particular problem the number of states of the graph can be infinity.

In the following sections we first introduce the problem of refutation of safety properties in concurrent systems and then we give some details about the software used in the experiments. After that we present seven sections in which we analyze the influence on the results of different parameters and techniques used in the ACOhg. We only restrict the analysis to those parameters and techniques that are new in the model and not on the traditional parameters of the ACO algorithms such as pheromone evaporation rate (ρ), colony size or number of total steps. For these parameters lots of work can be found analyzing their influence [12].

4.1 Systems Verification

From the very beginning of computer research, computer engineers have been interested in techniques allowing them to know whether a software module fulfils a set of requirements (the specification). Modern software is very complex and these techniques have become a necessity in most software companies. One of these techniques is *model checking* [10], 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) [9] or Computation Tree logic (CTL) [8]. One of the best known model checkers is SPIN [18], which takes a software model codified in Promela (a programming language usually not used in real programs) and a property specified in LTL as input. 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 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 [17] for more details). If such kind of cycle does not exist the system fulfil the property and the verification ends with success. The algorithm used by SPIN for the graph exploration is Nested-DFS [16]. 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. This phenomenon limits the size of the models to be checked. Some techniques used to alleviate this problem are partial order reduction [21], symbolic model checking [7], and symmetry reduction [19].

The properties that can be specified with LTL formulae can be classified into two groups: *safety* and *liveness* properties [22]. 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 of one objective node (one accepting node) in a graph. Furthermore, the path from one initial node to the objective node represents an execution of the concurrent system in which the given safety property is violated.

This fact has been used in previous work to verify safety properties using classical algorithms in the graph exploration domain (such as A* or Weighted A*) [13, 14]) or even Genetic Algorithms [1, 15]. 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 feature, cannot ensure that the system fulfils 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 [14] or on the objective node (if it is known beforehand) [19].

For the experiments we try to find deadlock states in the Edsger Dijkstra Dining Philosophers problem. A brief description of the problem is the following. A given number n of philosophers sit down in a round table to eat

rice. On the table there are n chopsticks distributed in a circular layout. Each philosopher needs two chopsticks to eat, the one on his/her left and the one on his/her right, which s/he takes in that order. This scenario can be modelled in a computer using n processes that take the "chopsticks" to "eat" and then they release them after the lunch, allowing other "philosophers" to eat. In this model a deadlock can occur when all the philosophers take the chopstick on his/her left at the same time and they wait until the chopstick on the right is free. 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 $n+1$.

4.2 Integration of ACOhg and SPIN

There exists a model checker developed by Stefan Edelkamp and Alberto Lluch-Lafuente called HSF-SPIN that is composed by a library of graph exploration algorithms (HSF) and SPIN, allowing the application of heuristic search methods to the verification of PROMELA models [13]. We have implemented ACOhg in the MALLBA library [5] and we have included this library into HSF-SPIN. This way, we can apply ACOhg without dealing with the details related to model representation (PROMELA interpretation, LTL formulae, and so on) and we can use a great amount of heuristic functions ready to use (implemented in HSF-SPIN).

4.3 Algorithms and Parameters

For the experiments we use an ACOhg algorithm with the base configuration shown in Table 1. The missionary technique is used for reaching objective nodes of unknown depth and the technique of removing useless pheromone trails is not enabled (that is, $\tau_\theta = -\infty$). 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. In the following sections some parameters are changed. In those sections we mention the corresponding changes. If no change is mentioned the value of the parameters is that of Table 1.

Parameter	**Value**
Steps	10
Colony size	5
λ_{ant}	10
σ_s	2
s	10
x_i	0.8
a	5
ρ	0.4
α	1.0
β	1.0
p_p	1000
p_c	1000

Table 1: Parameters for the ACOhg.

Since ACOhg is an stochastic algorithm, we need to perform several independent runs in order to get an idea of the behaviour of the algorithm. In the specialized literature is well established that a minimum of 30 independent runs is enough to get statistical confidence of the results. In our experiments we outperform this number and perform 100 independent runs in order to get a high statistical confidence. In order to support the empirical study we applied statistical tests to the results. Due to room issues we show these tests en Appendix A. In this section we just comment if the statistical tests support the observations or not.

With respect to the heuristic information, we use $\eta_{ij} = 1/(1+H_{ap}(j))$, where $H_{ap}(j)$ is the the number of active processes in state j.

4.4 Ant Paths Length

In the first experiment we change the length of the paths traversed by ants during the construction phase. We try different values for the length ranging from the depth of the solution d_{opt} to $3d_{opt}$ by steps of 3 units. In this experiment the λ_{ant} is kept constant during all the search and all the ants start their construction phase in the initial node (that is, the expansion and missionary techniques are not used). We use here $n=20$ philosophers, which forms a model large enough to clearly see the influence of λ_{ant} in the results. In Table 2 we show the results.

λ_{ant}	**hit rate**	**len**	**mem (KB)**	**cpu (ms)**
21	24	21.00	32554.75	603.33
24	23	21.00	30939.00	543.91
27	53	22.58	30762.89	534.72
30	72	24.94	29163.39	519.58
33	86	27.00	31830.63	567.33
36	78	26.74	30263.77	531.28
39	93	28.48	29127.08	498.92
42	99	30.98	25090.01	407.17
45	99	33.00	21951.78	330.10
48	99	33.36	24521.23	394.34
51	100	34.48	21850.14	322.40
54	100	38.20	19057.19	262.80
57	100	38.96	17681.94	228.90
60	100	38.04	17203.26	216.80
63	100	40.36	17581.37	228.50

Table 2: Results obtained with $n = 20$ philosophers when λ_{ant} changes.

We can observe in the second column of Table 2 that hit rate increases when ants can build longer paths. The reason is that the part of the graph explored growths with λ_{ant} and the ants find different paths to get the objective node. However, these paths are, in general, longer than the optimal path. This can be seen in the third column of Table 2, where we can observe that the average length of the error trails increases with λ_{ant}. In fact, all the differences that are statistically significant confirm this observation (see Appendix A).

The fact that the algorithm finds more easily an error trail when λ_{ant} increases influences also the memory and the CPU time required to find the error trail. In the forth and fifth columns of Table 2 we can observe that both memory and CPU time decrease when λ_{ant} increases (with statistical confidence). The probability of finding an error trail is higher and the number of steps required to find it is smaller. In this way, in addition to the reduction in CPU time, the amount of pheromone trails stored in memory is also reduced, and hence, the memory required.

4.5 Missionary technique

In the following experiment we use $n = 20$ philosophers again to study the efficacy of the missionary technique, i.e., the technique that allows the ants to start in nodes different from the initial. This technique is used for reaching a high depth in the graph maintaining fixed the length of the ant paths. 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. First, we fix s to 10 as in the base configuration and we change σ_s and λ_{ant}. Hit rate, average length of the solutions, the average memory used, and the average CPU time are shown in Tables 3, 4, 5, and 6 respectively.

	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

Table 3: Analysis of the missionary technique. Hit rate.

From the results 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 the ants can reach deeper nodes in the graph and they can find more paths reaching the objective node. We observe again that the hit rate increases with λ_{ant}. For this reason we can find the highest hit rate (100%) for small values of σ_s and large values of λ_{ant}.

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. This observation is supported by statistical tests. A large number of stages implies the exploration of deeper nodes in the construction graph and, thus, longer paths to the objective state are found with higher probability. The same happens when λ_{ant} is increased, as we saw in the previous section.

σ_s	λ_{ant} 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

Table 4: Analysis of the missionary technique. Average length.

In general, small values of σ_s and large values of λ_{ant} imply higher hit rate but longer error trails. We have to find a trade-off between efficacy and quality of solution.

σ_s	λ_{ant} 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

Table 5: Analysis of the missionary technique. Average required memory.

If we take a look to the amount of resources required by the algorithm we notice that both memory and CPU time increase with σ_s with statistical confidence. 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. The explanation is related to the pheromone reset after each 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 forgotten and the memory fall down to a small value. We can observe this on the figures. If the pheromone reset is very frequent (small σ_s) the average amount of required memory is low, but if the pheromone reset is not so frequent (high value of σ_s) the average required memory is higher. The influence of λ_{ant} is not so clear. 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 $\lambda_{ant} = 20$ (Figure 4) than for $\lambda_{ant} = 10$ (Figure 3). The reset of pheromone trails introduces an uncertainty factor that prevents from predicting the influence of λ_{ant} on the memory consumption. However, we can see in Figures 3 and 4 that the average memory required in the stages two, three, and followings is approximately the same for $\lambda_{ant} = 20$ and $\lambda_{ant} = 10$. Only in the first stage there is a noticeable difference in the memory required. This difference in the first stage seems to be the reason of the increase in the memory consumption when λ_{ant} increases. The statistical tests support this observation, in fact, in the cases in which this general rule does not hold (forth and fifth columns of Table 5 for $\sigma_s \geq 6$) the differences in the results are not statistically significant.

Concerning the CPU time, the influence of σ_s follows a clear trend (statistically supported). When σ_s is small the probability of finding a solution is higher and it is found faster. The influence of λ_{ant} is again not clear. On one hand, when λ_{ant} increases more time is required to construct the ant paths (columns 1 to 4 in Table 6). The statistical tests support this observation. On the other hand, the probability of finding a solution is higher and the average time required to find it is reduced. Although it is observed this phenomenon in the fifth column of Table 6, the differences in the results are not statistically significant, so we can only state 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 or even the missionary technique might not be used. With respect to λ_{ant}, a large value can increase the probability of finding a solution but it also increase the amount of resources required when the missionary technique is used.

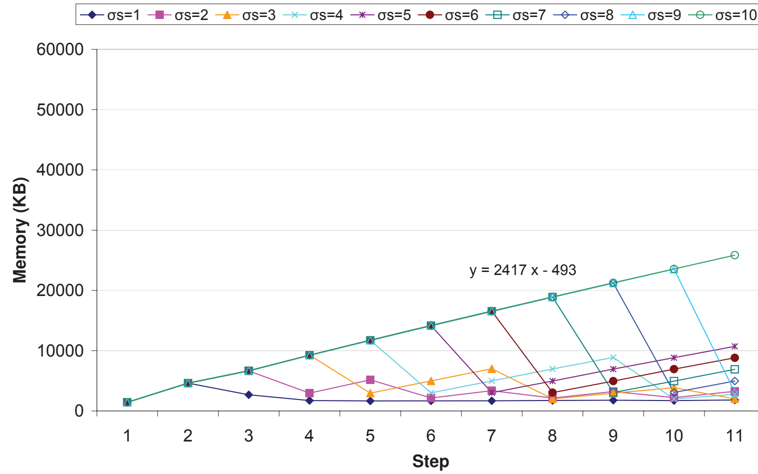

Figure 3: Trace of memory consumption when $\lambda_{ant} = 10$.

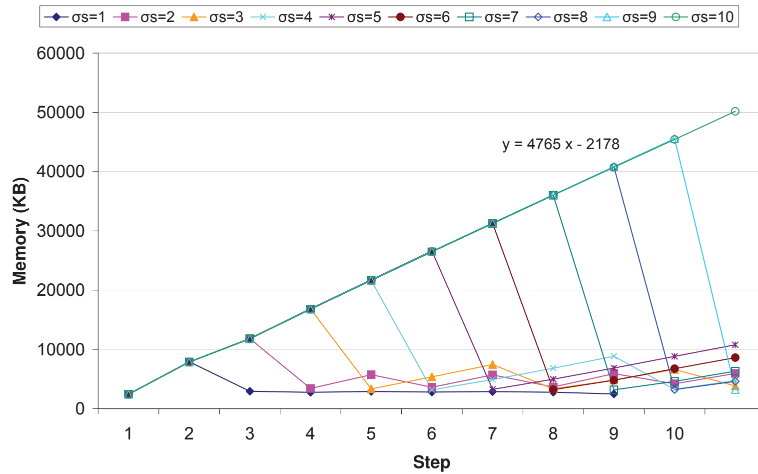

Figure 4: Trace of memory consumption when $\lambda_{ant} = 20$.

	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

Table 6: Analysis of the missionary technique. Average required CPU time.

4.5.1 Saved solutions

Now we are going to study the influence on the results of s, the number of saved solutions between stages. For this experiment we set $\lambda_{ant} = 25$ and we show in Tables 7, 8 and 9 the hit rate, average ant paths length, and average required memory, respectively, when s and σ_s changes. In this experiment the maximum value for σ_s is nine because ten or more means that the missionary technique is not used (because the maximum number of steps is 10 according to Table 1) and, in this case, saving solutions between stages has no sense. In other words: s has no influence on the results.

We observe in Table 7 that s has no influence on the hit rate. That is, using more solutions as starting points for the ants does not increase the probability of finding a solution. However, this behaviour can be due to the particular features of the Dining Philosophers model, in which the deadlock can be reached from any state. In models with

σ_s	s				
	2	4	6	8	10
1	100	100	100	100	100
2	100	100	100	100	100
3	100	100	100	100	100
4	100	100	99	100	100
5	98	98	99	99	99
6	100	100	100	98	97
7	99	98	98	98	96
8	98	99	99	98	95
9	96	92	90	87	82

Table 7: Analysis of saved nodes. Hit rate.

states that can not reach an objective node, perhaps we can find a particular configuration in which s can have an influence on the hit rate.

σ_s	s				
	2	4	6	8	10
1	42.12	52.44	55.24	54.40	55.28
2	32.92	37.92	40.12	39.68	42.20
3	31.80	34.64	33.04	34.36	34.36
4	30.04	31.52	32.64	33.48	33.96
5	28.88	29.82	30.70	31.30	30.98
6	27.64	29.20	29.28	30.96	30.44
7	27.95	29.00	28.18	29.57	27.79
8	27.16	27.79	27.46	28.88	27.99
9	26.96	26.17	26.29	27.76	27.63

Table 8: Analysis of saved nodes. Average length.

With respect to the average length we observe that when s increases the average length is also longer for small values of σ_s with statistical confidence. The algorithm always saves the best s solutions. Thus, if s is small, the average length of the saved solutions is lower than if s is large. That is, if s is large ants can start their construction phase from nodes with a large depth and the solutions they build are longer. This behaviour is clear when the number of stages is high (σ_s small) but it is not observed when σ_s is large. In fact, according to the statistical tests, the differences among the average length of the solutions are not statistically significant when $\sigma_s \geq 5$, that is, the influence of s on the length of the solutions is reduced as σ_s increases.

σ_s	s				
	2	4	6	8	10
1	10240.74	10156.94	10214.54	10058.16	10280.96
2	14091.03	14248.48	14235.47	14406.50	14399.85
3	18754.63	19606.93	18364.97	19181.42	19191.94
4	23519.18	24113.39	24101.95	24499.62	24465.29
5	28279.24	27902.32	27783.02	27126.36	28289.27
6	30541.07	33189.07	30892.58	33144.63	31614.64
7	33801.41	35325.66	35863.67	33544.72	33507.68
8	36728.16	38334.01	40759.55	40125.62	37339.66
9	40382.51	37787.13	43417.22	43634.08	40800.35

Table 9: Analysis of saved nodes. Average required memory.

If we take a look to Table 9 we can observe that there is not a clear influence of s on the memory required. Perhaps we could expect a slight increase in the memory required when s increases (the only two statistically significant differences support this hypothesis). However, the results show that this slight increase in the required memory is masked by the randomness of the algorithm. Thus, we conclude that s has no influence on the required memory, at least for $s \leq 10$. We do not show the results of CPU time because the behaviour is similar to that obtained in the memory.

In conclusion, we can state that a small value of s is preferable because the average ant paths length is smaller in this case and both hit rate and required memory do not depend on the value of s.

4.5.2 Pheromone reset

Finally, in this section 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 it is not performed. We set $\lambda_{ant} = 25$ and $s = 2$. Hit rate, average paths length and average memory required are shown in Table 10. We also show the results of a statistical test (Kruskal-Wallis) comparing the two variations of the algorithms with respect to the average length and the average memory required.

	No reset			Reset			Stat. sig.	
σ_s	hit	len	mem	hit	len	mem	len	mem
1	100	43.68	11519.17	100	42.12	10240.74	-	+
2	100	33.88	16555.47	100	32.92	14091.03	-	+
3	100	31.76	20146.70	100	31.80	18754.63	+	+
4	100	28.60	24442.83	100	30.04	23519.18	-	+
5	100	27.00	28595.77	98	28.88	28279.24	+	+
6	100	27.68	33315.26	100	27.64	30541.07	-	+
7	99	27.10	36525.46	99	27.95	33801.41	-	+
8	99	26.37	38855.26	98	27.16	36728.16	-	-
9	93	25.52	39883.61	96	26.96	40382.51	+	-
10	50	23.24	30753.10	47	23.21	29979.36	-	-

Table 10: Analysis of the pheromone reset.

We can observe in the table that the fact of forgetting the pheromone trails after one stage does not have a great influence on the hit rate or the average path length (only three differences are statistically significant), but, as expected, it has a slight influence on the required memory (seven differences are statistically significant). However, the extra required memory when no reset is performed is not very large. This indicates that most of the memory consumption is performed in the first stage. In order to illustrate this statement 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.

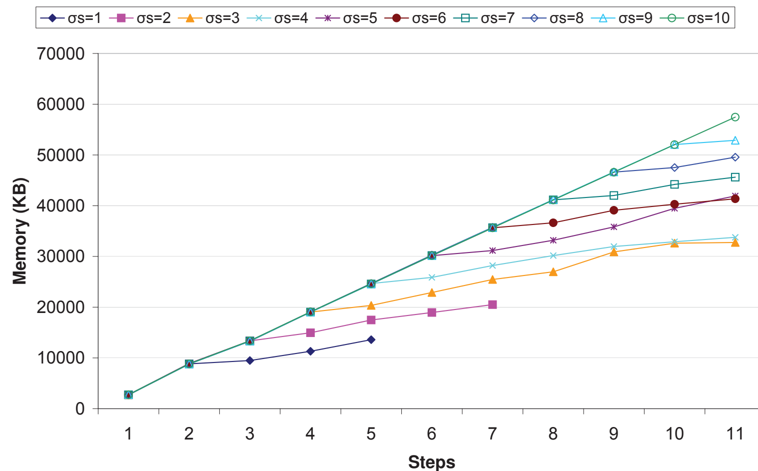

Figure 5: Trace of memory consumption when pheromone trails are not reset.

In Figure 5 we observe that the slope of the curve decreases when the algorithm begins the second stage in all the cases. This means that the number of paths that can be traversed from the initial node is much larger than the one found in other nodes.

4.6 Expansion Technique

In this section we use again $n = 20$ philosophers to study the efficacy of the expansion technique. This technique is used for reaching a high depth in the graph increasing λ_{ant} as the search progresses. There are two parameters that govern this technique: σ_i, and δ_l. We try different values for these parameters in order to investigate their influence on the results. We set the initial λ_{ant} to the same value as δ_l. This way the maximum depth the algorithm can reach in each step is the same as in the missionary technique analyzed in the previous section (allowing a direct comparison). The hit rate, the average length of the solutions, the average memory used, and the average CPU time are shown in Tables 11, 12, 13, and 14, respectively.

From Table 11 we obtain the same conclusions as in the missionary technique. The hit rate increases when σ_i is small, that is, when λ_{ant} is increased frequently. This frequent increase of λ_{ant} allows the algorithm to find different

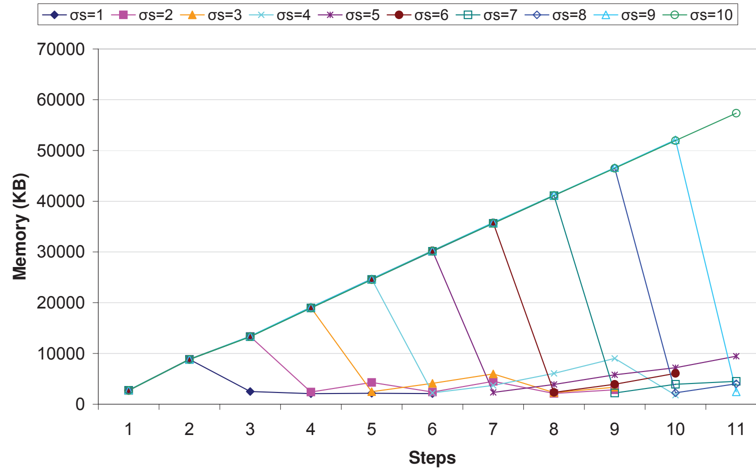

Figure 6: Trace of memory consumption when pheromone trails are reset.

σ_i	δ_l				
	5	10	15	20	25
1	83	100	100	100	100
2	13	86	100	100	100
3	0	42	91	100	100
4	0	15	75	95	100
5	0	0	53	73	95
6	0	0	39	64	92
7	0	0	21	62	92
8	0	0	17	43	73
9	0	0	12	25	65
10	0	0	0	0	55

Table 11: Analysis of the expansion technique. Hit rate.

paths for reaching the objective node, increasing the probability of finding a solution. Hit rate is also increased when δ_l grows, since the portion of the graph explored grows with it.

σ_i	δ_l				
	5	10	15	20	25
1	27.80	32.72	32.36	33.20	34.56
2	22.54	29.09	31.20	32.28	33.12
3	-	25.86	29.40	30.88	31.00
4	-	25.27	28.36	29.97	30.80
5	-	-	24.09	28.34	30.31
6	-	-	24.38	28.62	27.35
7	-	-	25.95	27.45	27.13
8	-	-	24.53	27.23	27.08
9	-	-	25.00	27.40	26.11
10	-	-	-	-	23.47

Table 12: Analysis of the expansion technique. Average length.

The influence of σ_i and δ_l on the average solution length is similar to the influence of σ_s and λ_{ant} in the missionary technique. When the hit rate is high (small σ_s and large δ_l) the length of the solutions is far from the optimum (observation supported by the statistical tests in Appendix A). This configuration favours the construction of longer paths to the objective node. On one hand, this fact increases the probability of finding a solution, but, on the other hand, produces longer error trails.

The required resources (memory and CPU time) take the smallest values usually when σ_i is small. A small value of σ_i increases the amount of memory required on each step because λ_{ant} increases very frequently (see Figure 7). An increase in λ_{ant} implies also an increase in the CPU time. However, when σ_i is low the probability of finding a solution earlier is higher and there is a trade-off between the two trends. We can observe in Table 13 that in this case the increase in the probability of finding a solution is more important and usually the memory required is reduced when σ_i is small. The influence of δ_l on the memory consumption has a similar explanation. When δ_l is increased

σ_i	δ_l				
	5	10	15	20	25
1	39528.87	35973.12	27228.16	26275.84	22835.03
2	33083.08	43781.95	37109.76	33597.44	27924.48
3	-	41179.43	40431.12	41502.72	33232.14
4	-	40413.87	43499.52	46500.38	38780.55
5	-	-	40361.06	45743.34	41158.63
6	-	-	38098.05	47568.00	39905.42
7	-	-	43203.05	50935.74	42128.70
8	-	-	41984.00	52390.70	41161.03
9	-	-	43264.00	53616.64	39422.82
10	-	-	-	-	31028.69

Table 13: Analysis of the expansion technique. Average required memory.

σ_i	δ_l				
	5	10	15	20	25
1	752.05	623.90	419.40	392.30	323.30
2	620.77	848.14	658.80	549.40	428.30
3	-	792.38	759.34	759.20	565.00
4	-	787.33	842.40	907.05	716.80
5	-	-	761.32	873.97	764.74
6	-	-	716.41	923.91	753.70
7	-	-	842.38	1040.65	801.52
8	-	-	820.59	1088.14	796.85
9	-	-	875.83	1146.00	763.23
10	-	-	-	-	542.18

Table 14: Analysis of the expansion technique. Average required CPU time.

we find two opposite trends: the increase in the memory required per step and the increase of the probability of finding a solution. However, in this case it is not clear which one is the strongest trend and we can not observe a clear trend in Tables 13 and 14.

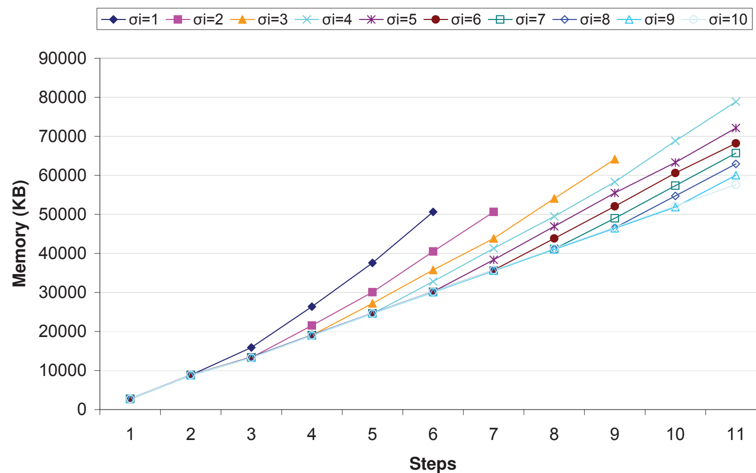

Figure 7: Evolution of the memory required in the search for $\delta_l = 10$.

4.7 Missionary vs Expansion Technique

In this section we compare missionary and expansion techniques, which were analyzed separately in the previous sections. We compare the results of the algorithm implementing the missionary technique with the ones of the algorithm implementing the expansion technique when $\sigma_s = \sigma_i$ and λ_{ant}(missionary)$= \delta_l$ (the rest of the parameters are those of Table 1). In this way, we can make a fair comparison between missionary and expansion techniques, because the maximum depth the algorithms reach during the search in each step is the same. We present in Figures 8, 9, 10, and 11 the hit rate, the length of error trails, the memory used, and the CPU time required by the two techniques. These figures contain all the information shown in the tables of the previous two sections.

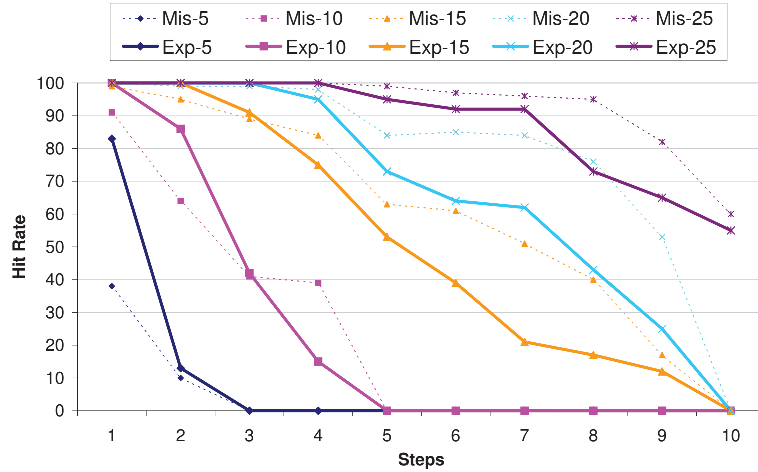

Figure 8: Comparison between missionary and expansion techniques: Hit rate.

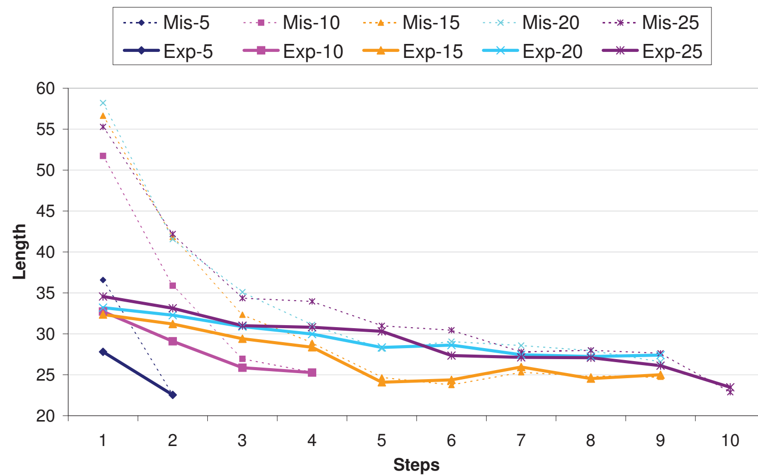

Figure 9: Comparison between missionary and expansion techniques: Length of error trails.

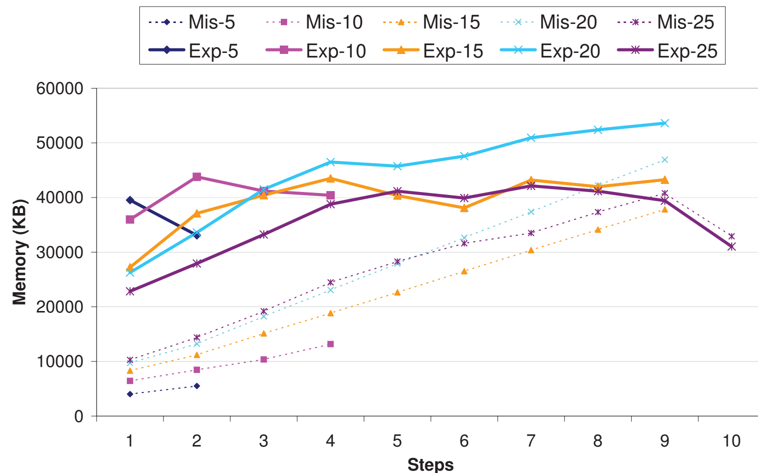

Figure 10: Comparison between missionary and expansion techniques: Memory used.

In Figure 8 we can observe that the expansion technique has higher hit rate than the missionary technique for all values of σ_i and σ_s when the increments in the maximum exploration depth (δ_l and λ_{ant} respectively) are low. However, this fact changes when λ_{ant} and δ_l are increased in the missionary and expansion techniques respectively. During the search, the expansion technique increase the region explored and the pheromone trails of the previous steps guide the construction of the new partial solutions. When δ_l is small the algorithm is able to find good partial solutions because the expansion is slow. On the contrary, the missionary technique is not guided by the pheromone trails of the previous stages and it can go away from the optimum paths When δ_l increases in the expansion technique,

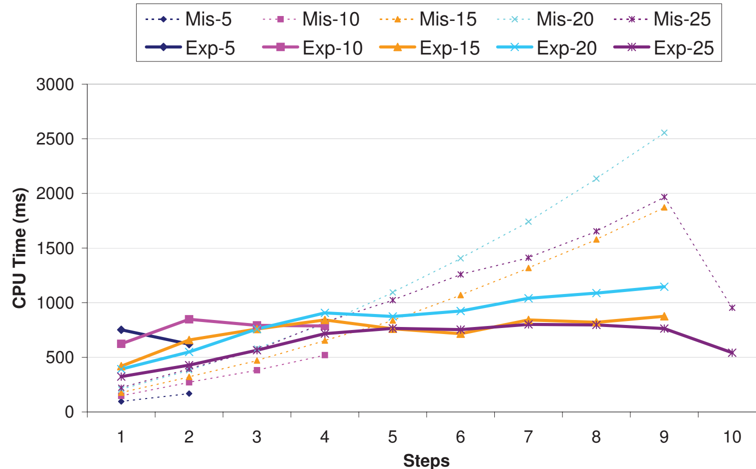

Figure 11: Comparison between missionary and expansion techniques: CPU time required.

it explores large regions of the graph and, in this case, the pheromone trails makes the algorithm to explore regions that have been explored before with the consequent loss of time. The missionary technique behaves better in this situation, since it focuses on more promissory regions of the construction graph.

Analyzing the length of the error trails found (Figure 9) we observe that the two techniques are tied. We can notice only a (statistically significant) advantage of the expansion technique for small values of σ_i. As we said in the previous paragraph, when λ_{ant} is increased in the expansion technique, a larger region of the construction graph is explored. This gives the opportunity to the algorithm of finding short error trails. In the case of the missionary technique, the different stages explore a region of a similar size but in different depths and, thus, it is more likely to find longer error trails. However, this behaviour can only be noticed for small values of σ_i and σ_s.

Let us focus now on the resources (Figures 10 and 11). The expansion technique requires more memory than the missionary technique. This observation is supported by the statistical tests. One reason for this is that longer ant paths have to be managed in the expansion technique. The second main reason is that in the missionary technique the pheromone trails are discarded at the end of each stage. Thus, we conclude that the missionary technique is more efficient than the expansion technique with respect to the memory consumption. In the case of CPU time required, we find a different behaviour in both techniques. For small values of σ_i the expansion technique requires more computational time than the missionary technique. However, from $\sigma_i = \sigma_s$ between 3 and 5 this trend changes: the missionary technique requires more CPU time than the expansion one. This change is also supported by the statistical tests (see Appendix A). We think that the main reason for this behaviour is the fall in the hit rate of the expansion technique. The average values shown in the figures consider only the executions in which an error trail is found. The reduction of the hit rate in the expansion technique with respect to the missionary technique for large σ_i values means that some executions require more steps than the allowed ones (10) to obtain an error trail. These unsuccessful executions are not taken into account to compute the average values and for this reason we get a lower unreal value for the resources required. Only when hit rate is 100% the average values of the resources are reliable. This explains the jump in the average values of memory and CPU time when $\sigma_i = \sigma_s = 10$.

The final conclusion of this experiment is that the missionary technique with intermediate values for σ_s is the best option for the search. This configuration constitutes the best trade-off between quality of solution (efficacy) and required resources (efficiency).

4.8 Scalability

In the next experiment we study the scalability of the algorithm. We change the number of philosophers from 2 to 20 and we observe the average length of the error trail found, memory, and CPU time required by the algorithm. We follow the recommendations depicted in the previous section using the missionary technique with $s = 2$ and $\sigma_s = 2$. For the length of the ant paths we use the next expression $\lambda_{ant} = \lceil 1.25n \rceil$. Using this expression we obtained 100% hit rate in all the cases[4]. The rest of parameters are the ones found in Table 1. In Table 15 and Figures 12, 12, and 12 we show the results.

In the results we observe that the average length of the error trail grows in an almost linear way (Figire 12). We need to try more values of n to get an idea of which kind of curve follows the length of error trails (exponential, potential, etc.). On the other hand, the resources graph (memory and CPU time) follow an exponential-like growth (Figures 13 and 14). This is a direct consequence of the exponential growth of the search space.

[4]This is not a general rule, it just worked with the implementation of the dining Philosophers used in the experiments.

n	len	mem (KB)	cpu (ms)
2	3.00	1909.00	5.80
4	5.00	2003.16	8.30
6	7.16	2401.44	12.80
8	9.84	3084.40	18.20
10	13.60	4325.40	35.00
12	17.36	5906.52	58.00
14	20.56	7835.92	93.30
16	25.56	9697.73	133.60
18	28.48	11509.30	185.10
20	34.84	14291.11	272.80
22	40.48	17142.51	387.00
24	46.28	20664.32	512.00
26	53.12	24248.32	685.90
28	54.08	27777.01	870.60
30	61.44	32194.56	1123.10
32	67.20	36177.92	1382.20
34	69.24	41052.16	1735.30
36	76.64	45895.68	2097.70
38	83.36	51251.20	2561.50
40	88.44	56545.28	3145.20

Table 15: Scalability results. Number of philosophers from 2 to 40.

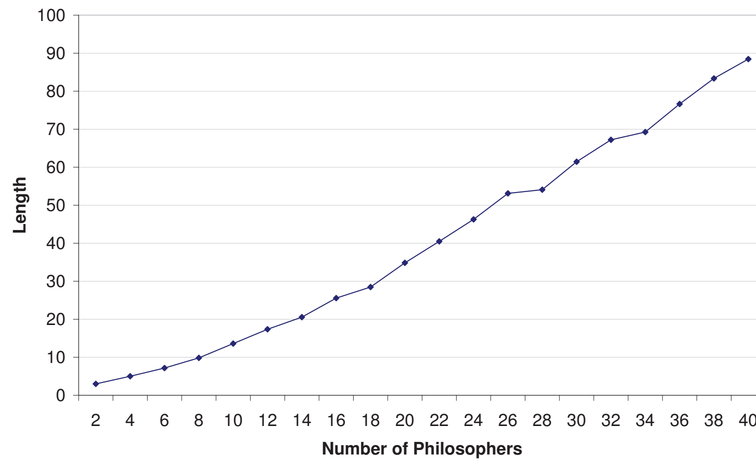

Figure 12: Scalability results. Average length of error trails.

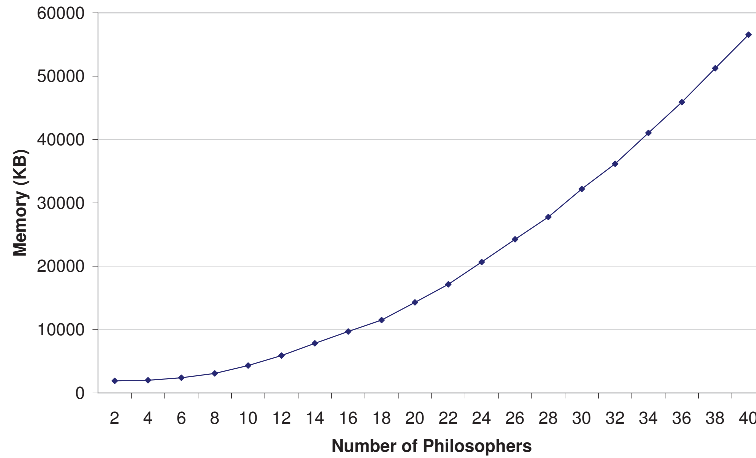

Figure 13: Scalability results. Average memory required.

4.9 Heuristic

In this section we study the influence of the heuristic information on the results. This heuristic information is used during the construction phase to select the next node to visit (see Section 2.1) but it is also used in the fitness

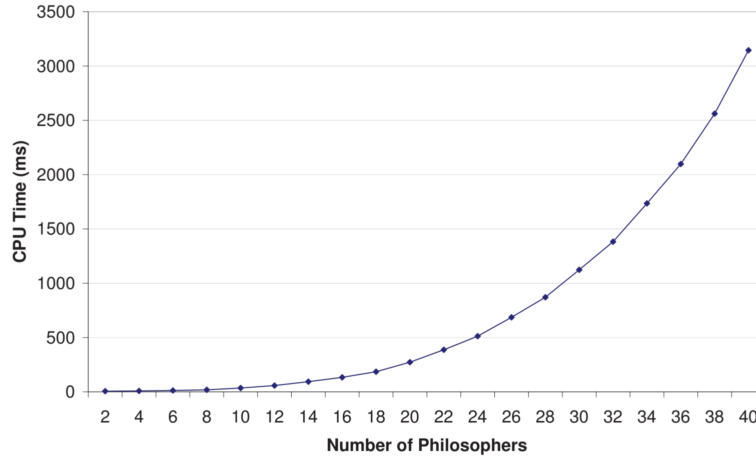

Figure 14: Scalability results. Average CPU time required.

function in order to evaluate partial solutions: it is added to the path length in order to obtain an estimation of the shortest path to the objective node that can be obtained using the given partial solution. In order to determine the importance of the heuristic information in the construction phase, we try here four different values for β. The remaining parameters are those of Table 1. In Table 16 we show the hit rate, the average length of error trails, the average required memory, and the average CPU time of ACOhg when β changes.

β	**Hit rate**	**Avg. length**	**Avg. mem**	**Avg. CPU**
0	9	35.22	8645.00	256.67
1	60	36.20	8515.60	217.33
2	61	37.72	8486.77	218.85
3	58	35.14	8454.03	210.17

Table 16: Influence of β in the results.

As we can observe from the results the use of the heuristic information is beneficial for the search. It increases the probability of finding the objective node (higher hit rate). In addition, the average length of error trails is not longer when $\beta \geq 1$, so the quality of the solution is not decreased (the differences in the average values are not statistically significant). The influence of the heuristic information on the memory consumption and the CPU time is small but real: the statistical test confirms that using heuristic information reduces the resources required. However, there is no statistically significant difference when $\beta \geq 1$. We also observe that the hit rate does not increase when β is greater than 1. In conclusion, we can state that the heuristic information is an important help for the ants building the solution path and it must be used in order to get high hit rate. However, a value of $\beta = 1$ is enough (at least in this model) for obtaining the maximum benefit from the heuristic information.

4.10 Penalties in Fitness Function

In this section we want to analyze the influence on the results of the two different penalties we use in the fitness function as discussed in Section 3.2. In this experiment the algorithm stops when the total number of steps (10) is reached and not when one solution is found. This is done in this way in order to appreciate the influence of p_p on the results (if any). If we would allow the algorithm to stop when the first solution is found, then most of the paths traversed by the ants are penalized during the search and only a few paths are not penalized. As a consequence, there would be no influence of p_p on the results. We try different values for p_c and p_p and we show the hit rate, the average length of error trails, the average required memory, and the average CPU time required in Tables 17, 18, 19,and 20, respectively.

	p_c			
p_p	0.0	10.0	100.0	1000.0
0.0	52	55	59	67
10.0	62	61	56	61
100.0	60	59	63	54
1000.0	61	58	60	63

Table 17: Analysis of the penalties. Hit rate.

p_p	p_c			
	0.0	10.0	100.0	1000.0
0.0	35.38	35.84	35.71	32.28
10.0	35.13	35.56	33.71	35.03
100.0	34.67	34.63	34.21	34.33
1000.0	33.20	34.79	35.47	35.60

Table 18: Analysis of the penalties. Average Length.

p_p	p_c			
	0.0	10.0	100.0	1000.0
0.0	8485.38	8497.36	8498.42	8489.00
10.0	8493.19	8484.48	8495.14	8481.52
100.0	8491.00	8475.51	8502.52	8469.30
1000.0	8493.85	8468.38	8509.27	8461.32

Table 19: Analysis of the penalties. Average Memory.

p_p	p_c			
	0.0	10.0	100.0	1000.0
0.0	237.31	235.45	235.93	233.73
10.0	234.03	238.69	234.46	236.72
100.0	238.00	235.76	237.30	235.37
1000.0	236.72	234.66	241.67	233.49

Table 20: Analysis of the penalties. CPU time.

From the tables we can observe that there is no influence of p_p and p_c on the hit rate or the results. A statistical test confirms this observation. That is, the values of p_p and p_c are not relevant when we try to find a deadlock in the Dining Philosophers Problem with $n = 20$ philosophers. This is an unexpected result, since the penalties are used to guide the search into more promissory paths in the construction graph. Due to this we expected, at least, higher hit rate and lower average solutions lengths when higher penalties are used. However, in this particular instance this is not true. The length of ant paths λ_{ant} is low (10) and this can explain why there is no influence of p_c, since it is relatively easy for the algorithm to find solutions without cycles.

5 Conclusions and Future Work

In this deliverable we analyze in depth a new ACO model, ACOhg, that can solve problems with a huge underlying graph that must be 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 searched deadlock states in a concurrent system implementing the Dijsktra Dining Philosophers problem. We analyzed the influence on the results of several parameters of the model such as the ant path length, the different techniques used to reach high depth nodes (missionary and expansion techniques), the heuristic coefficient and so on. With this experimental study we want to establish some guidelines about how to change the new parameters. This is very useful for the researches using the ACOhg model.

This deliverable is part of a recently open research line and lot of work have to be done. First, 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. Second, the application of metaheuristic algorithms to the formal methods domain in software engineering and, in particular, to formal verification is has not been extensively explored. Previous work on this topic is based on genetic algorithms and is limited by representation of solution issues. In this sense, we must go beyond in the application and optimize the proposed model in order to outperform the algorithms currently used in model checking. Third, we want to parallelize the ACOhg model in order to perform the search in a distributed computer system or even in a grid computing environment.

A Statistical Validation

In this appendix we include the statistical tests performed for the experiments of this deliverable. This is a very important practice that researchers in metaheuristics and in non-deterministic algorithms in general should include

in their work. Nowadays, authors not doing statistical tests often report "clear" advantages for their proposals based on rather small negligible numerical improvements. Although it is an intensive and time consuming task, work including this information is supposed to improve the overall research quality in literature.

In each case the procedure for generating the statistical information presented in the tables is the following. First a Kolmogorov-Smirnov test is performed in order to check whether the variables are normal or not. If they are, an ANOVA I test is performed; otherwise we perform a Kruskal-Wallis test. After that, we do a multiple comparison test whose results we present in the following figures. In order to save room we present the results of the multiple comparisons in a compact form. We illustrate this representation in Figure 15, where we see a multiple comparison of five samples. The cell (i, j) in the figure is black if the difference of the average values of samples i and j are statistically significant. That is, the black square in position $(2, 3)$ means that samples 2 and 3 have average values that are different with statistical confidence.

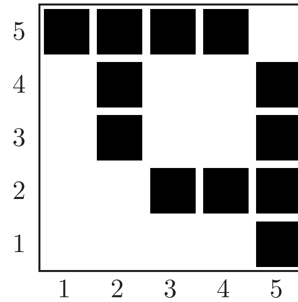

Figure 15: An example of multiple comparison.

In the following we present the statistical tests for all the comparisons. All of them are shown in the compact form above mentioned except the comparison between missionary and expansion techniques. In this last case (Tables 21, 22, and 23) we present a table filled with + and − signs indicating if the differences between the missionary and expansion techniques are statistically significant or not, respectively.

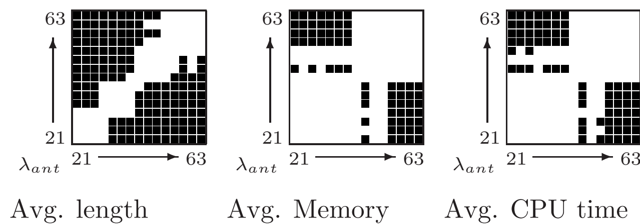

Figure 16: Influence of λ_{ant} on the results (Table 2).

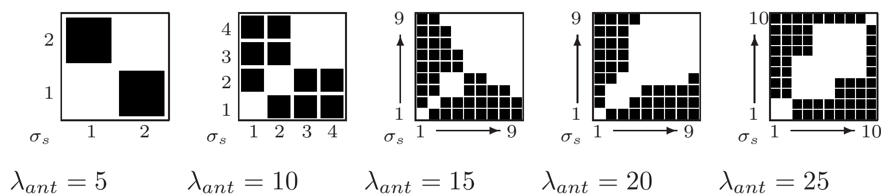

Figure 17: Missionary technique: influence of σ_s on the average length (Table 4).

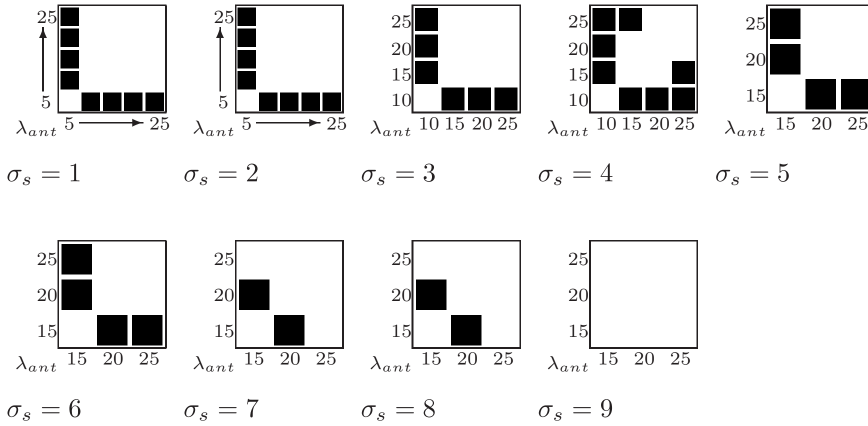

Figure 18: Missionary technique: influence of λ_{ant} on the average length (Table 4).

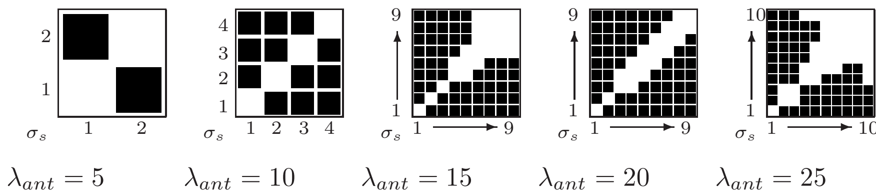

Figure 19: Missionary technique: influence of σ_s on the average length (Table 5).

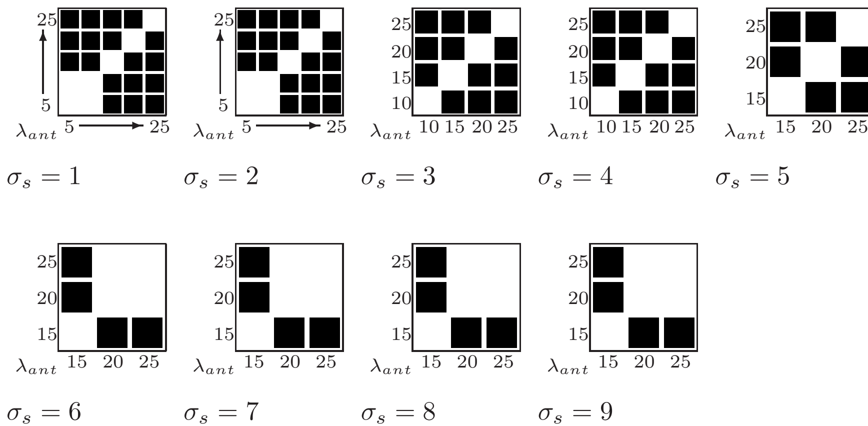

Figure 20: Missionary technique: influence of λ_{ant} on the average memory (Table 5).

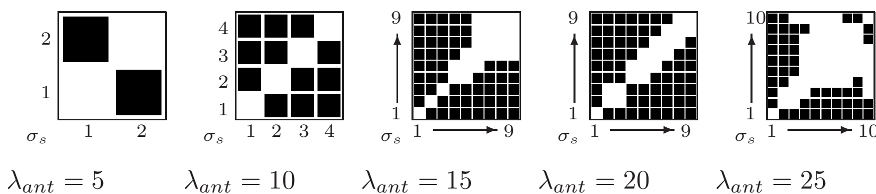

Figure 21: Missionary technique: influence of σ_s on the average CPU time (Table 6).

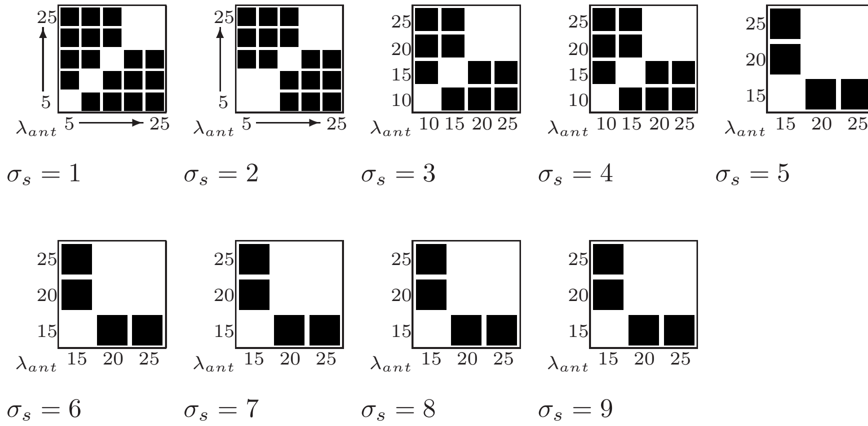

Figure 22: Missionary technique: influence of λ_{ant} on the average memory (Table 6).

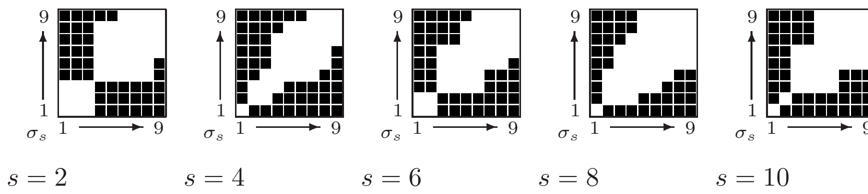

Figure 23: Missionary technique (saved solutions): influence of σ_s on the average average length (Table 8).

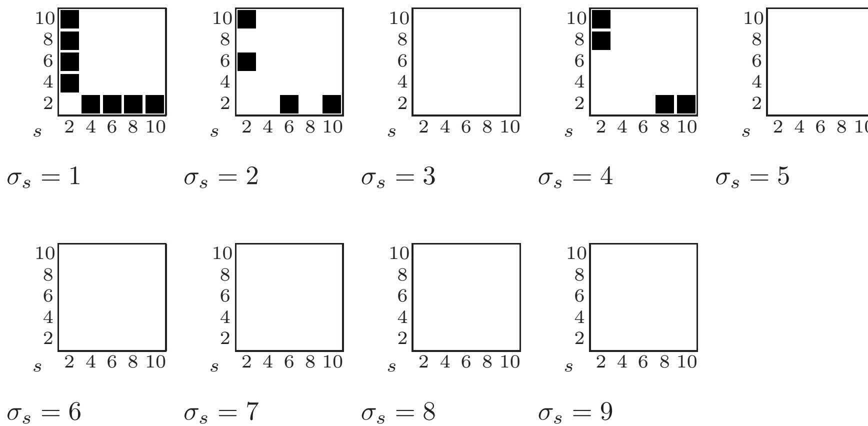

Figure 24: Missionary technique (saved solutions): influence of s on the average length (Table 8).

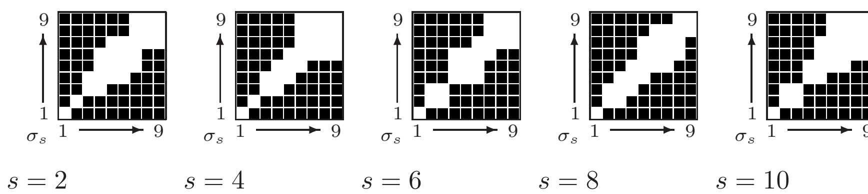

Figure 25: Missionary technique (saved solutions): influence of σ_s on the average average memory (Table 9).

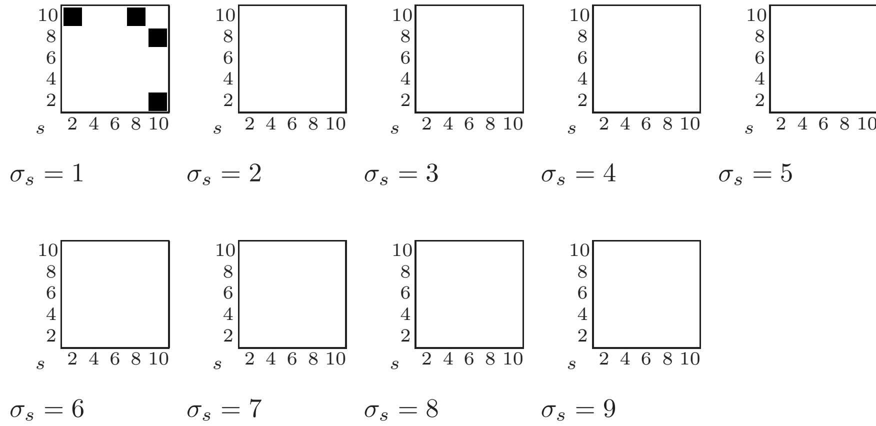

Figure 26: Missionary technique (saved solutions): influence of s on the average memory (Table 9).

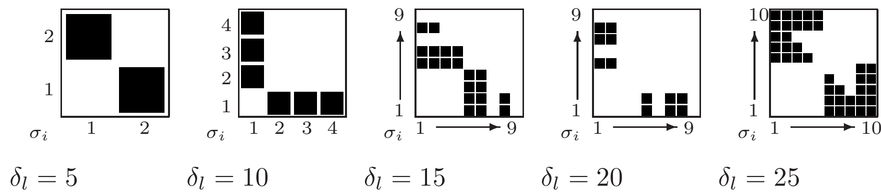

Figure 27: Expansion technique: influence of σ_i on the average length (Table 12).

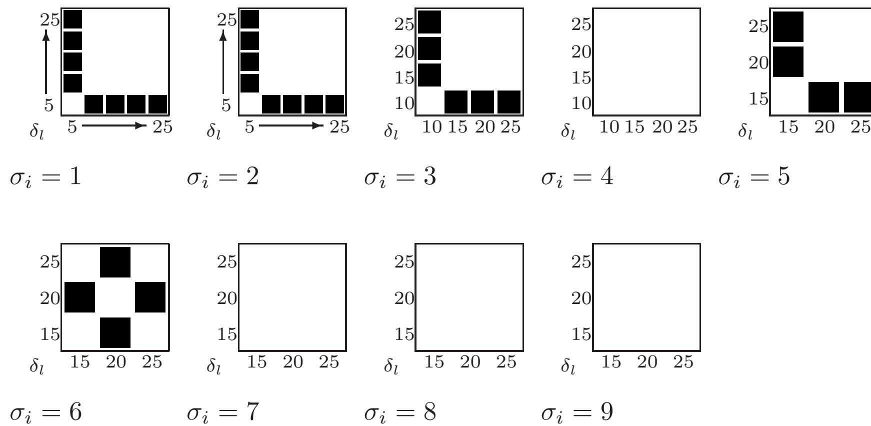

Figure 28: Expansion technique: influence of δ_l on the average length (Table 12).

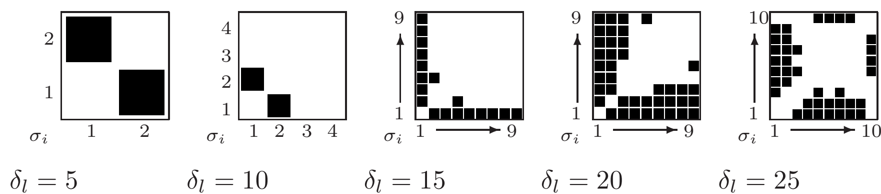

Figure 29: Expansion technique: influence of σ_i on the average memory (Table 13).

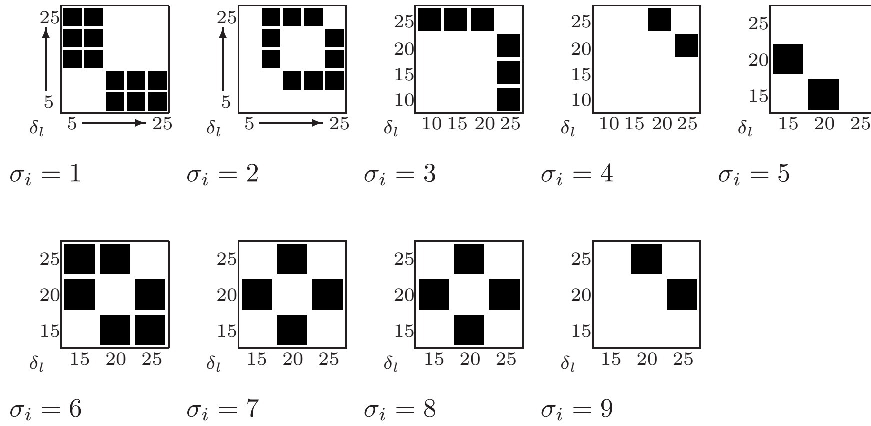

Figure 30: Expansion technique: influence of δ_l on the average memory (Table 13).

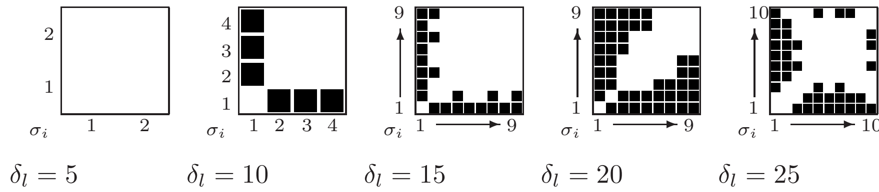

Figure 31: Expansion technique: influence of σ_i on the average CPU time (Table 14).

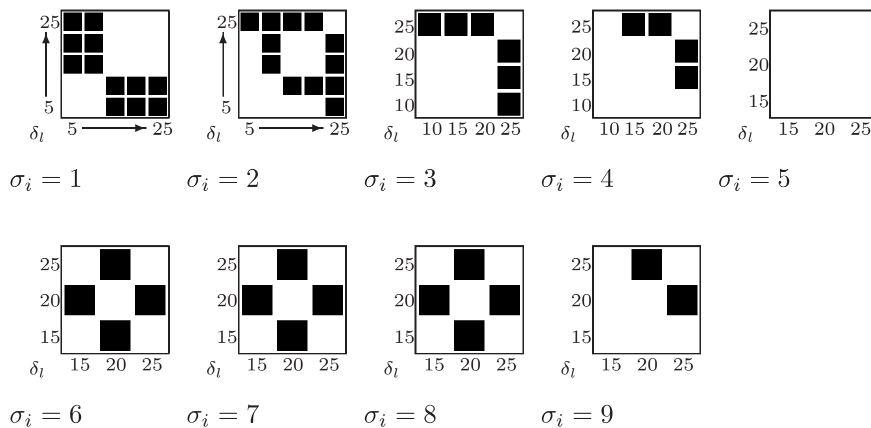

Figure 32: Expansion technique: influence of δ_l on the average CPU time (Table 14).

$\sigma_i = \sigma_s$	$\delta_l = \lambda_{ant}$				
	5	10	15	20	25
1	+	+	+	+	+
2	-	+	+	+	+
3	-	-	+	+	+
4	-	-	-	-	+
5	-	-	-	-	-
6	-	-	-	-	+
7	-	-	-	-	-
8	-	-	-	-	-
9	-	-	-	-	+
10	-	-	-	-	-

Table 21: Comparison between missionary and expansion techniques. Length of error trails.

$\sigma_i = \sigma_s$	$\delta_l = \lambda_{ant}$				
	5	10	15	20	25
1	+	+	+	+	+
2	+	+	+	+	+
3	-	+	+	+	+
4	-	+	+	+	+
5	-	-	+	+	+
6	-	-	+	+	+
7	-	-	+	+	+
8	-	-	+	+	+
9	-	-	+	+	-
10	-	-	-	-	-

Table 22: Comparison between missionary and expansion techniques.Average memory required.

$\sigma_i = \sigma_s$	$\delta_l = \lambda_{ant}$				
	5	10	15	20	25
1	+	+	+	+	+
2	+	+	+	+	-
3	-	+	+	+	-
4	-	+	+	-	+
5	-	-	+	+	+
6	-	-	+	+	+
7	-	-	+	+	+
8	-	-	+	+	+
9	-	-	+	+	+
10	-	-	-	-	+

Table 23: Comparison between missionary and expansion techniques. Average CPU time required.

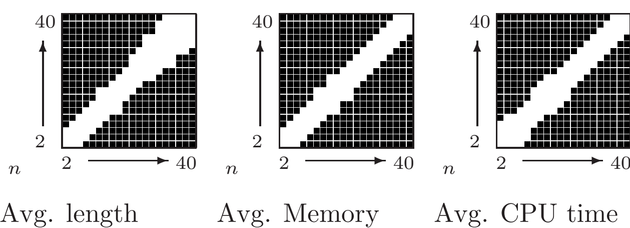

Figure 33: Influence of the number of philosophers, n, on the results (Table 15).

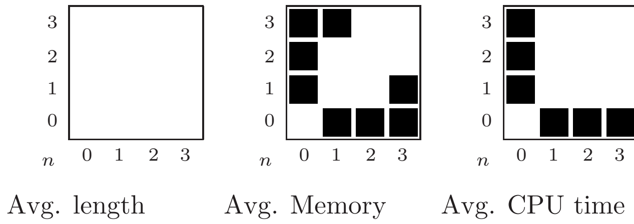

Figure 34: Influence of the heuristic power, β, on the results (Table 16).

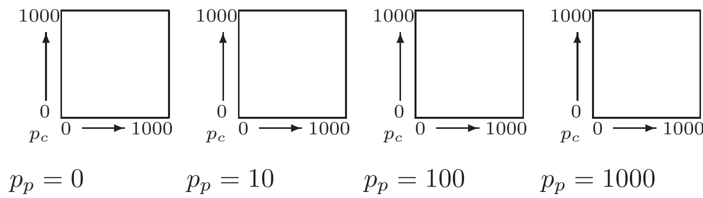

Figure 35: Penalties: influence of p_c on the average length (Table 18).

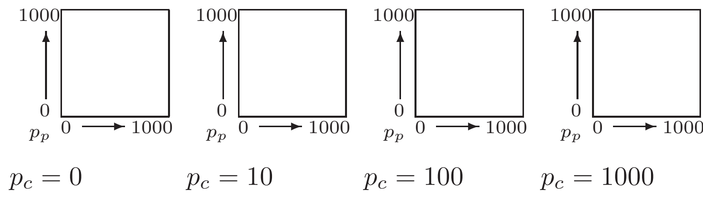

Figure 36: Penalties: influence of p_p on the average length (Table 18).

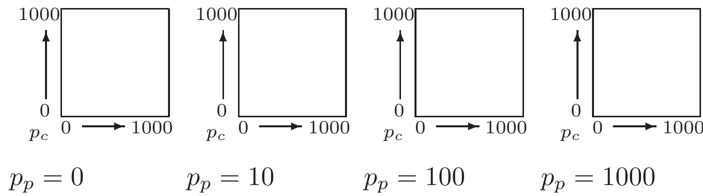

Figure 37: Penalties: influence of p_c on the average memory (Table 19).

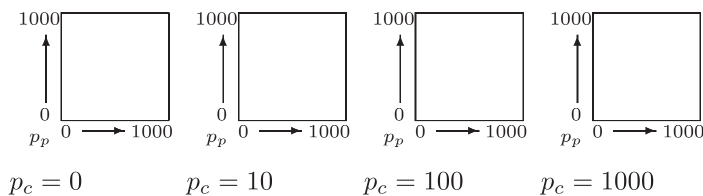

Figure 38: Penalties: influence of p_p on the average memory (Table 19).

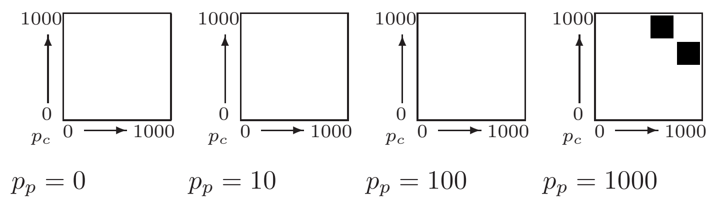

Figure 39: Penalties: influence of p_c on the average CPU time (Table 20).

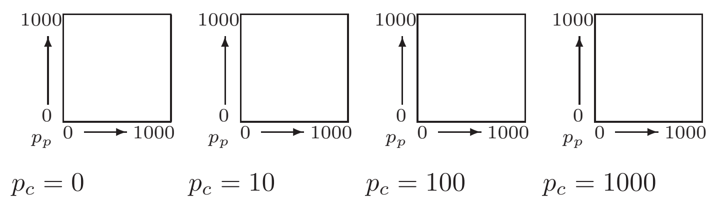

Figure 40: Penalties: influence of p_p on the average CPU time (Table 20).

References

[1] E. Alba and J.M. Troya. Genetic Algorithms for Protocol Validation. In *Proceedings of the PPSN IV International Conference*, pages 870–879, Berlin, 1996. Springer.

[2] Enrique Alba and Francisco Chicano. ACOhg: Dealing with huge graphs. In *Proceedings of the Genetic and Evolutionary Conference*, pages 10–17, London, UK, July 2007. ACM Press.

[3] Enrique Alba and Francisco Chicano. Finding safety errors with ACO. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, pages 1066–1073, London, UK, July 2007. ACM Press.

[4] Enrique Alba and Francisco Chicano. Una versión de ACO para problemas con grafos de muy gran extensión. In *Actas del Quinto Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, MAEB 2007*, pages 741–748, Puerto de la Cruz, Tenerife, Spain, February 2007.

[5] Enrique Alba, Gabriel Luque, José García-Nieto, Guillermo Ordo nez, and Guillermo Leguizamón. MALLBA: A software library to design efficient optimization algorithms. *International Journal of Innovative Computing and Applications (IJICA)*, 1(1), 2007. (to appear).

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

[7] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4), April 1994.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[9] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000.

[11] M. Dorigo, V. Maniezzo, and A. Colorni. Positive feedback as a search strategy. Technical Report 91-016, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1991.

[12] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. The MIT Press, 2004.

[13] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed Explicit Model Checking with HSF-SPIN. In *Lecture Notes in Computer Science, 2057*, pages 57–79. Springer, 2001.

[14] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Protocol Verification with Heuristic Search. In *AAAI-Spring Symposium on Model-based Validation Intelligence*, pages 75–83, 2001.

[15] Patrice Godefroid and Sarfraz Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. In *Lecture Notes in Computer Science, 2280*, pages 266–280. Springer, 2002.

[16] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32. American Mathematical Society, 1996.

[17] Gerald J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.

[18] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1–17, May 1997.

[19] Alberto Lluch Lafuente. Symmetry Reduction and Heuristic Search for Error Detection in Model Checking. In *Workshop on Model Checking and Artificial Intelligence*, August 2003.

[20] G. Leguizamón and Z. Michalewicz. A new version of Ant System for subset problems. In P.J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the 1999 Congress on Evolutionary Computation*, pages 1459–1464, Piscataway, New Jersey, USA, 1999. IEEE Computer Society Press.

[21] Alberto Lluch-Lafuente, Stefan Leue, and Stefan Edelkamp. Partial Order Reduction in Directed Model Checking. In *9th International SPIN Workshop on Model Checking Software*, Grenoble, April 2002. Springer.

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

[23] Krzysztof Socha and Christian Blum. *Metaheuristic Procedures for Training Neural Networks*, chapter 8, Ant Colony Optimization, pages 153–180. Springer, 2006.