Ants Can Solve Constraint Satisfaction Problems

Christine Solnon
LISI, University of Lyon 1, Bat. 710,
43 bd du 11 novembre,
69 622 Villeurbanne cedex, France.
E-mail: csolnon@bat710.univ-lyon1.fr

Abstract— **In this paper we describe a new incomplete approach for solving constraint satisfaction problems (CSPs) based on the ant colony optimization (ACO) metaheuristic. The idea is to use artificial ants to keep track of promising areas of the search space by laying trails of pheromone. This pheromone information is used to guide the search, as a heuristic for choosing values to be assigned to variables.**

We first describe the basic ACO algorithm for solving CSPs and we show how it can be improved by combining it with local search techniques. Then, we introduce a preprocessing step, the goal of which is to favor a larger exploration of the search space at a lower cost, and we show that it allows ants to find better solutions faster. Finally, we evaluate our approach on random binary problems.

Keywords— **Ant Colony Optimization (ACO), Constraint Satisfaction Problems (CSPs), Local Search**

I. Introduction

Solving constraint satisfaction problems (CSPs) involves finding an assignment of values to variables that satisfies a set of constraints. This general problem has many real-life applications, such as scheduling, resource allocation, pattern regognition, and machine vision.

To solve CSPs, one may explore the search space in a systematic and complete way, until either a solution is found, or the problem is proven to have no solution [31]. In order to reduce the search space, this kind of complete approach is usually combined with filtering techniques that narrow the variables domains with respect to some partial consistencies. Completeness is a very desirable feature, but it often becomes intractable on hard combinatorial problems. This is particularly true when filtering techniques cannot reduce domains enough to make complete search feasible. Hence, incomplete approaches have been proposed that leave out exhaustivity, trying to quickly find approximately optimal solutions in an opportunistic way. In these approaches, the search space is usually explored stochastically, using heuristics to guide the search towards the most-promising areas.

In this paper, we describe an incomplete approach for solving CSPs based on the ant colony optimization (ACO) metaheuristic [8], [7]. The idea is to keep track of promising areas of the search space, by laying "pheromone" trails. This pheromone information is used to guide the search, as a heuristic for choosing values to be assigned to variables.

A. *The ant colony optimization metaheuristic*

ACO is a stochastic approach that has been proposed to solve different hard combinatorial optimization problems such as traveling salesman problems [6], [10], [9], graph colouring problems [4], quadratic assignment problems [12], [20], and vehicle routing problems [1], [13]. The main idea of ACO is to model the problem as the search for a minimum cost path in a graph. Artificial ants walk through this graph, looking for good paths. Each ant has a rather simple behaviour so that it will typically only find rather poor quality paths on its own. Better paths are found as the emergent result of the global cooperation among ants in the colony.

The behaviour of artificial ants is inspired from real ants: they lay pheromone trails on the graph edges and choose their path with respect to probabilities that depend on pheromone trails; these pheromone trails progressively decrease by evaporation. In addition, artificial ants have some extra-features that do not find their counterpart in real ants. In particular, they live in a discrete world (a graph) and their moves consist of transitions from nodes to nodes. Also, they are usually associated with data structures that contain the memory of their previous actions. In most cases, pheromone trails are updated only after having constructed a complete path, and not during the walk, and the amount of pheromone deposited is usually a function of the quality of the path. Finally, the probability for an artificial ant to choose an edge often depends not only on pheromones, but also on some problem-specific local heuristics.

B. *Motivations and overview of the paper*

When using the ACO metaheuristic to solve a new combinatorial optimization problem, one of the main tasks is to model the problem as the search of a feasible minimum cost path over a weighted graph, where feasibility is defined with respect to a set of constraints. In most cases, these constraints are satisfied in an a priori way, that is, ants only construct "feasible" paths that never violate constraints: at each step of the walk, the next vertex to visit is chosen within a set of feasible candidates (the neighborhood) that only contains vertices that are actually consistent with respect to the constraints of the problem. This set of feasible candidates is supposed never to be empty so that ants are always able to construct feasible paths. For example, in the traveling salesman problem, the constraints impose that each city must appear exactly once in any feasible solution. Therefore, each ant maintains a list of visited cities, and at each step of the walk, it chooses the next city to visit among the set of cities that have not yet been visited.

However, this kind of a priori constraint satisfaction may only be applied to loosely constrained problems, when the difficulty is not to find feasible solutions but the best feasible one with respect to some given objective function. This paper investigates ACO capabilities for solving CSPs, that is, the goal is no longer to optimize an objective function under some constraints, but to find an assignment that actually satisfies all the constraints. A main motivation underlying this work is to provide a generic tool to handle a whole class of problems. Hence, the proposed algorithm may be used to solve any CSP in a generic way.

The paper is organized as follows. In the next section, we introduce some definitions and terminology about CSPs, and we recall some results on phase-transitions. Section III describes the basic ACO algorithm for solving CSPs, called Ant-Solver. Sections IV and V show how Ant-Solver can be improved by integrating local search techniques and by introducing a preprocessing step. Finally, section VI reports experimental results on random binary CSPs and section VII compares our work with some other related approaches and discusses future work.

II. Background

A. Constraint satisfaction problems (CSPs)

A *CSP* [31] is defined by a triple (X, D, C) such that X is a finite set of variables, D is a function that maps every variable $X_i \in X$ to its domain $D(X_i)$, that is, the finite set of values that can be assigned to X_i, and C is a set of constraints, that is, relations between some variables which restrict the set of values that can be assigned simultaneously to these variables.

An *assignment*, noted $\mathcal{A} = \{<X_1, v_1>, \ldots, <X_k, v_k>\}$, is a set of variable-value pairs and corresponds to the simultaneous assignment of values $v_1, \ldots, v_k$ to variables $X_1, \ldots, X_k$, respectively. An assignment $\mathcal{A}$ violates a constraint $c_i \in C$ if each variable involved in c_i is assigned in $\mathcal{A}$ and the relation defined by c_i is not satisfied when replacing each variable of c_i by its associated value in $\mathcal{A}$. The *cost* of an assignment $\mathcal{A}$, denoted by $cost(\mathcal{A})$, is defined by the number of constraints that are violated by $\mathcal{A}$. A *solution of a CSP* (X, D, C) is a complete assignment for all the variables in X, which satisfies all the constraints in C, that is, a complete assignment with zero cost.

Most real-life CSPs are over-constrained, so that no solution exists. Hence, the CSP framework has been generalized to max-CSPs [11]. In this case, the goal is no longer to find a consistent solution, but to find a complete assignment that maximizes the number of satisfied constraints. Hence, a *solution of a max-CSP* is a complete assignment with minimal cost. In this paper, we implicitly solve max-CSPs. Artificial ants naturally solve optimization problems. In our context of constraint satisfaction, they aim at minimizing the number of violated constraints. This approach could also be extended to valued CSPs [26] in a rather straightforward way.

B. Random binary CSPs

Binary CSPs only have binary constraints, that is, each constraint involves two variables exactly. Binary CSPs may be generated at random. A class of randomly generated CSPs is characterized by 4 components $< n, m, p_1, p_2 >$ where n is the number of variables, m is the number of values in each variable domain, $p_1 \in [0, 1]$ is a measure of the connectivity (p_1 determines the number of constraints), and $p_2 \in [0, 1]$ is a measure of the tightness of the constraints (p_2 determines the number of incompatible pairs of values for each constraint).

Experiments reported in this paper were obtained with random binary CSPs generated according to model A as described in [19], that is, for each pair of distinct variables, we add a constraint between them with probability p_1, and then for each constraint we rule out a pair of values for the two constrained variables with probability p_2. As incomplete approaches cannot detect inconsistency, we report experiments performed on feasible instances only, that is, CSPs that do have at least one solution. To generate feasible instances, we first generate a solution randomly and then, for each constraint, we forbid ruling out pairs of values that belong to the generated solution.

C. Phase-transitions

When considering a class of combinatorial problems, rapid transitions in solvability may be observed as an order parameter is changed [2]. These “phase-transitions” occur when evolving from instances that are under-constrained, and therefore solved rather easily, to instances that are over-constrained, whose inconsistency is thus proven rather easily. Harder instances usually occur between these two kinds of “easy” instances, when approximately 50% of the instances are satisfiable.

In order to predict the phase-transition region, [14] introduces the notion of “constrainedness” of a class of problems, noted κ, and shows that when κ is close to 1, instances are critically constrained, and belong to the phase-transition region. For a class of random binary CSPs $< n, m, p_1, p_2 >$, [3] defines this constrainedness by $\kappa = \frac{n-1}{2} p_1 \log_m(\frac{1}{1-p_2})$.

One might think that phase-transitions only concern complete approaches, as they are usually associated with transitions from solvable to unsolvable instances, and incomplete approaches cannot detect unsolvability. However, different studies (e.g., [5], [3]) have shown that very similar phase-transition phenomena may also be observed with incomplete approaches such as local search.

III. Description of Ant-Solver

The proposed algorithm for solving CSPs, called Ant-Solver, follows the classical ACO algorithmic scheme for static combinatorial optimization problems [8], [7], and is sketched in Figure 1. At each cycle, each ant constructs a complete assignment, and then pheromone trails are updated. The algorithm stops iterating either when an ant has found a solution, or when a maximum number of cy-

procedure Ant-Solver
  Set parameters and Initialize pheromone trails
  repeat
    for k **in** $1..nbAnts$ **do**: Construct an assignment $\mathcal{A}_k$
    Update pheromone trails using $\{\mathcal{A}_1, \ldots, \mathcal{A}_{nbAnts}\}$
  until $cost(\mathcal{A}_i) = 0$ for some $i \in \{1..nbAnts\}$
    or max cycles reached

Fig. 1. The Ant-Solver algorithmic scheme.

procedure Construct assignment $\mathcal{A}$ for CSP (X, D, C)
  $\mathcal{A} \leftarrow \emptyset$
  while $\mid \mathcal{A} \mid < \mid X \mid$ **do**
    Select a variable $X_j \in X$ that is not assigned in $\mathcal{A}$
    Choose a value $v \in D(X_j)$ with probability $p_{\mathcal{A}}(<X_j, v>)$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{<X_j, v>\}$
  end

Fig. 2. The assignment construction algorithmic scheme.

cles has been performed. In this section, we first define the construction graph on which artificial ants lay pheromone trails, and we describe pheromone initialization. Then, we describe the assignment construction and the pheromone updating steps. Finally, we discuss setting parameters.

A. Construction graph and pheromone trails initialization

The graph on which artificial ants lay pheromone, called the construction graph, associates a vertex with each variable-value pair $<X_i, v>$ of the CSP. There is an edge between any pair of vertices corresponding to two different variables. More formally, the construction graph associated with a CSP (X, D, C) is the non directed graph $G = (V, E)$ such that

$$\begin{array}{rcl} V & = & \{<X_i, v> \mid X_i \in X \text{ and } v \in D(X_i)\} \\ E & = & \{(<X_i, v>, <X_j, w>) \in V^2 \mid X_i \neq X_j\} \end{array}$$

Ants communicate by laying pheromone on graph edges. The amount of pheromone on edge $(<X_i, v>, <X_j, w>)$ is noted $\tau(<X_i, v>, <X_j, w>)$. Intuitively, this amount of pheromone represents the learnt desirability of assigning value v to variable X_i and value w to variable X_j simultaneously. Notice that since the graph is not directed, $\tau(<X_i, v>, <X_j, w>) = \tau(<X_j, w>, <X_i, v>)$.

As proposed in the $\mathcal{MAX} - \mathcal{MIN}$ Ant System [30], we explicitly impose lower and upper bounds τ_{min} and τ_{max} on pheromone trails (with $0 < \tau_{min} < \tau_{max}$). The goal is to favor a larger exploration of the search space by preventing the relative differences between pheromone trails from becoming too extreme during processing. Also, pheromone trails are set to τ_{max} at the beginning of the algorithm, thus achieving a higher exploration of the search space during the first cycles. In section V, we shall propose to introduce a preprocessing step in order to initialize pheromone trails more suitably, and speed up convergence.

B. Construction of assignments by ants

The construction of complete assignments by ants is described in Figure 2. Ants start with an empty assignment, and then iteratively select a variable to be assigned and choose a vertex in the graph corresponding to a value assignment for this variable, until all variables have been assigned. The selection of a variable (described in subsection B.1) is performed with respect to some given procedure, whereas the choice of a value for the variable (described in subsection B.2) is made probabilistically, depending on pheromone trails.

Notice that by selecting a variable before choosing a value for it, the set of candidate vertices at each step—the neighborhood—is restricted to the set of all possible values for a single variable, instead of the set of all possible values for all unassigned variables. Such a restriction of the neighborhood is motivated mainly by the fact that the computation of transition probabilities is significantly more expensive than for other applications, such as the traveling salesman problem or the quadratic assignment problem.

B.1 Selection of a variable

When constructing an assignment, the order in which the variables are assigned is rather important, as the satisfaction of a constraint can only be checked once all its variables have been assigned. Variable-ordering heuristics have been studied widely [31], [27]. Some commonly used variable-orderings are: *most-constraining-first* ordering, which selects an unassigned variable that is connected (by a constraint) to the largest number of unassigned variables; *most-constrained-first* ordering, which selects an unassigned variable that is connected (by a constraint) to the largest number of assigned variables; *smallest-domain* ordering, which selects an unassigned variable that has the smallest number of consistent values with respect to the partial assignment already built; or *random ordering*, which randomly selects an unassigned variable.

To solve a CSP with Ant-Solver, one has to implement the variable selection procedure which, given a partial assignment, returns the next variable to be assigned. In all experiments reported in this paper, we have used the smallest-domain ordering.

B.2 Choice of a value

Once a variable has been selected, ants have to choose a value for it. The goal is to choose the "most promising" value for the variable, that is, the one that will participate in the smallest number of conflicts in the final complete assignment. However, if it is straightforward to evaluate the number of conflicts with the already assigned variables, it is more difficult to anticipate the consequences of the choice of a value on the further assignments of the remaining unassigned variables. Hence, there are very few heuristics for guiding this choice and these heuristics usually are problem-dependent, so they cannot be applied to general CSPs (e.g., [27] for the car-sequencing problem).

The main contribution of ACO for solving CSPs is to provide a general heuristic for choosing values: this choice is made randomly with a probability that depends on a pheromone factor (to evaluate the learnt desirability of the value), and a heuristic factor (to locally evaluate its quality). More formally, let us consider an ant that has already visited a set $\mathcal{A}$ of vertices, and whose next selected variable to be assigned is X_j. The probability for this ant to select vertex $<X_j,v>$ is defined proportionally to the attraction capability of this vertex with respect to all candidate vertices $<X_j,w>$ such that $w \in D(X_j)$:

$$p_{\mathcal{A}}(<X_j,v>) = \frac{[\tau_{\mathcal{A}}(<X_j,v>)]^\alpha [\eta_{\mathcal{A}}(<X_j,v>)]^\beta}{\sum_{w\in D(X_j)} [\tau_{\mathcal{A}}(<X_j,w>)]^\alpha [\eta_{\mathcal{A}}(<X_j,w>)]^\beta}$$

where $\tau_{\mathcal{A}}(<X_j,v>)$ is the pheromone factor of $<X_j,v>$, $\eta_{\mathcal{A}}(<X_j,v>)$ is its heuristic factor, and α and β are the parameters that determine their relative weights.

A main difference with many ACO algorithms is that pheromone and heuristic factors do not only depend on some local relation between the candidate vertex $<X_j,v>$ and the last visited vertex in $\mathcal{A}$, but on a global relation between the candidate vertex $<X_j,v>$ and the whole set $\mathcal{A}$ of visited vertices. Indeed, when choosing a value for X_j, all assignments stored previously in $\mathcal{A}$ are equally important. Therefore, the pheromone factor $\tau_{\mathcal{A}}(<X_j,v>)$ depends on all pheromones laid on all edges between $<X_j,v>$ and the visited vertices in $\mathcal{A}$:

$$\tau_{\mathcal{A}}(<X_j,v>) = \sum_{<X_k,m>\in\mathcal{A}} \tau(<X_k,m>,<X_j,v>)$$

The heuristic factor $\eta_{\mathcal{A}}(<X_j,v>)$ also depends on the whole set $\mathcal{A}$ of visited vertices. It is inversely proportional to the number of new violated constraints when assigning v to X_j:

$$\eta_{\mathcal{A}}(<X_j,v>) = \frac{1}{1+cost(\{<X_j,v>\}\cup\mathcal{A})-cost(\mathcal{A})}$$

C. Updating pheromone trails

After every ant has constructed a complete assignment, pheromone trails are updated according to ACO: all pheromone trails are decreased uniformly, in order to simulate evaporation and allow ants to forget bad assignments, and then the best ants of the cycle deposit pheromone. More formally, at the end of each cycle, the quantity of pheromone on each edge (i,j) is updated as follows:

$$\begin{array}{l}\tau(i,j) \leftarrow (1-\rho)\cdot\tau(i,j) + \sum_{\mathcal{A}_k\in BestOfCycle} \Delta\tau(\mathcal{A}_k,i,j)\\ \text{if } \tau(i,j) < \tau_{min} \text{ then } \tau(i,j) \leftarrow \tau_{min}\\ \text{if } \tau(i,j) > \tau_{max} \text{ then } \tau(i,j) \leftarrow \tau_{max}\end{array}$$

where

- ρ is the evaporation parameter, such that $0 \le \rho \le 1$,
- *BestOfCycle* is the set of the best assignments constructed during the cycle, that is, $BestOfCycle =$
$$\{\mathcal{A}_i \in \{\mathcal{A}_1,\ldots,\mathcal{A}_{nbAnts}\} \mid cost(\mathcal{A}_i) \text{ is minimal}\}$$
- $\Delta\tau(\mathcal{A}_k,i,j)$ is the quantity of pheromone deposited on edge (i,j) by the ant that has built assignment $\mathcal{A}_k$:
$$\begin{array}{ll}\Delta\tau(\mathcal{A}_k,i,j) = \frac{1}{cost(\mathcal{A}_k)} & \text{if } \{i,j\}\subseteq\mathcal{A}_k\\ \Delta\tau(\mathcal{A}_k,i,j) = 0 & \text{otherwise}\end{array}$$

Contrary to many ACO algorithms, ants deposit pheromone not only on the edges belonging to the path they followed, but on all edges between any pair of visited vertices, that is, on all edges belonging to the clique defined on the set of visited vertices. The quantity of pheromone laid is inversely proportional to the assignment cost, so that the more constraints are violated, the less pheromone is deposited.

D. Parameters setting

Ant-Solver is parameterized by α, β, ρ, and *nbAnts*. We now briefly study the influence of these parameters with respect to two criteria: the success rate, that is, the percentage of runs that find a solution, and the time spent to find a solution.

The number of ants, *nbAnts*, is set to 8: with smaller values, success rates are decreased (as the best assignments constructed at each cycle usually have higher costs); with greater values, running times increase while success rates are not improved (as the best assignments found at each cycle are not significantly better than with 8 ants).

β determines the weight of heuristic factors when computing transition probabilities. It is set to 10: with smaller values, ants globally find worse assignments, and it takes them longer to reach a solution.

α determines the weight of pheromone factors in the computation of transition probabilities, and ρ determines pheromone evaporation. Both parameters have an influence on the exploratory behavior of ants: to emphasize exploration, one may decrease α (so that ants become less sensitive to pheromone when they choose values), or decrease ρ (so that pheromone evaporates more slowly and relative differences between pheromone trails increase more slowly). When emphasizing exploration of the search space in this way, success rates are improved, but as a counterpart running times are increased. This is illustrated in Figure 3 on $< 100, 8, 0.14, p_2 >$ CSPs. As a conclusion, when setting α and ρ, one can either choose values that favor pheromone guidance—such as $\alpha = 3$ and $\rho = 0.02$—or values that favor exploration —such as $\alpha = 2$ and $\rho = 0.01$. In the former case, Ant-Solver will quickly find rather good assignments, but it may not find a solution. In the latter case, Ant-Solver will need more time to find good assignments, but it will more often succeed in finding a solution. One should note that this holds for the various tightness values p_2, although it is more emphasized for the hardest instances.

IV. Boosting Ant-Solver with local search

Local search has shown to be effective to solve large CSPs, e.g., the "million queens problem" in [23]. The basic idea is to construct a complete assignment and then gradually and iteratively repair it by changing some variable-value assignments. Local search may be combined with the ACO metaheuristic in a very straightforward way: ants construct assignments, exploiting pheromone trails, and local search improves their quality by iteratively performing local moves. Actually, the best-performing ACO algo-

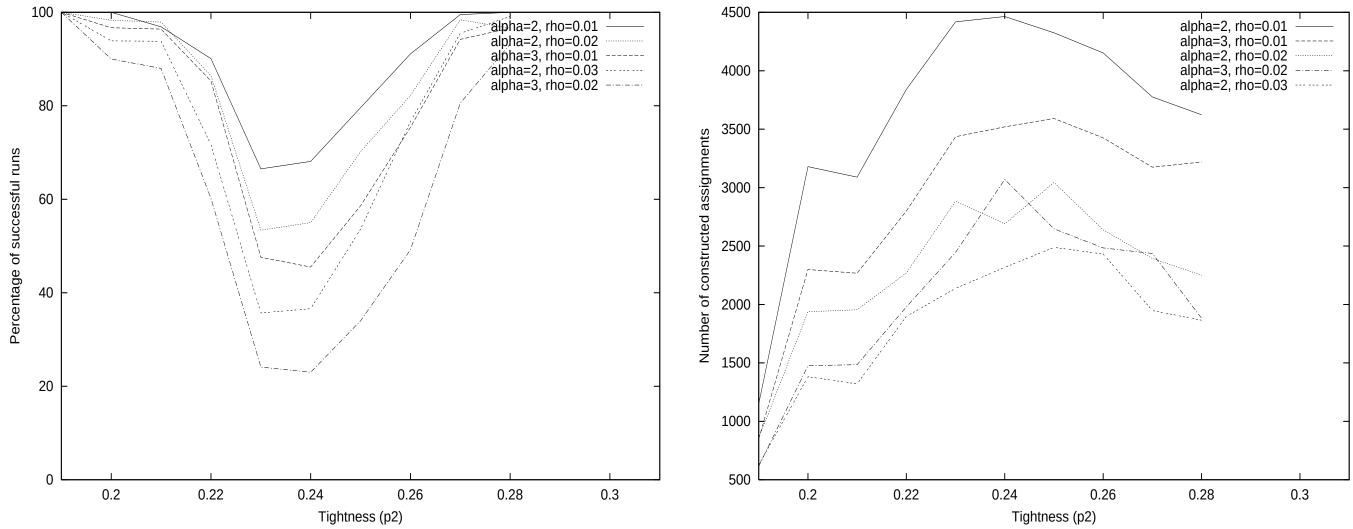

Fig. 3. Influence of α and ρ on the success rate (left curves) and the number of constructed assignments (right curves) when solving $< 100, 8, 0.14, p_2 >$ CSPs (average results on 300 instances for each value of p_2). With respect to the success rate criterion, the best results are obtained when $\alpha = 2, \rho = 0.01$, then when $\alpha = 2, \rho = 0.02$, then when $\alpha = 3, \rho = 0.01$, then when $\alpha = 2, \rho = 0.03$, and the worst results when $\alpha = 3, \rho = 0.02$. With respect to the time criterion (which is proportional to the number of constructed assignments), the best results are obtained when $\alpha = 2, \rho = 0.03$, then when $\alpha = 3, \rho = 0.02$, then when $\alpha = 2, \rho = 0.02$, then when $\alpha = 3, \rho = 0.01$, and the worst when $\alpha = 2, \rho = 0.01$.

rithms for many combinatorial optimization problems are hybrid algorithms that combine probabilistic solution construction by a colony of ants with local search [9], [30]. In this section, we propose to combine local search with Ant-Solver. In this case, one can view Ant-Solver as a way of taking benefit of the different locally optimal assignments found previously by local search in order to guide it towards the most-promising areas of the search space when constructing new assignments to be repaired.

The hybrid algorithm is derived from the algorithm of Figure 1 as follows: once each ant has constructed an assignment, and before updating pheromone trails, we apply a local search procedure to improve the assignment constructed. Various local search procedures may be used to improve assignments (see, e.g., [16] for an experimental comparison of some of them). However, as pointed out in [18], when choosing the local search procedure to use in a metaheuristic, such as evolutionary algorithms, iterated local search or ACO, one has to find a trade-off between computation time and solution quality. In other words, one has to choose between a fast but not-so-good local search procedure or a slower but more powerful one. For all experiments reported in this paper, we have used a local search procedure based on the min-conflicts heuristic, denoted by MCH [23]. More precisely, at each step we choose a conflicting variable (involved in some violated constraints) randomly, and then we choose a value for this variable which minimizes the number of conflicts. This repair procedure stops iterating when the number of violated constraints is not improved after $\mid X \mid$ successive iterations.

Figure 4 shows the evolution of average costs during processing for five different runs of Ant-Solver, and five different runs of Ant-Solver with MCH local search, on a same instance (with $\alpha = 2$, $\beta = 10$, $\rho = 0.01$, and $nbAnts = 8$).

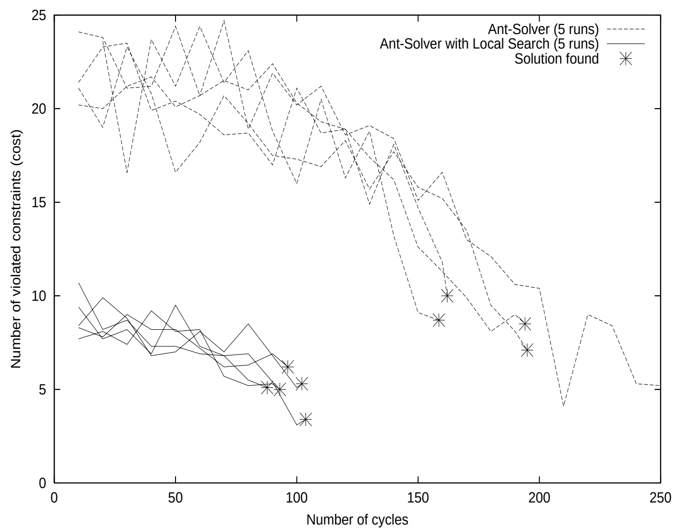

Fig. 4. Evolution of the average number of violated constraints by cycle on a $< 100, 8, 0.14, 22 >$ instance (with $\alpha = 2$, $\beta = 10$, $\rho = 0.01$ and $nbAnts = 8$). Dotted lines correspond to five different runs of Ant-Solver, continued lines correspond to five different runs of Ant-Solver with MCH Local Search.

For the five runs of Ant-Solver, the average number of violated constraints decreases from more than 20 at the beginning of the search to less than 10 after 150 cycles. Finally, in four runs at least one ant has found a solution, between cycle 150 and cycle 200, whereas in the fifth run, none of the ants was able to find a solution. Differently, when combining Ant-Solver with MCH local search, ants find far better assignments from the first cycle on: the average number of violated constraints decreases from 9 at the beginning of the search to less than 5 at cycle 90. Finally, in all the five runs a solution was found before cycle 110. This shows that local search actually improves the solution process, with respect to both success rates and running times. More experimental results are reported in section VI.

V. Boosting Ant-Solver with preprocessing

When exploring the search space in a stochastic way, one usually has to find a trade-off between two antagonistic goals. On the one hand, one has to guide the search towards "promising" areas of the search space, usually close to the best solutions found so far. Indeed, a study of the shape of the search space shows that for some combinatorial optimization problems (such as the traveling salesman or the quadratic assignment problems) better local optima tend to be closer to global optima [30]. Actually, this property underlies the motivation of evolutionary approaches, that are of little interest in situations for which the correlation between solution fitness and the distance to optimal solutions is too low [17], [21].

On the other hand, one also has to favor a large exploration of the search space, in order to discover new, and hopefully more successful, areas of the search space. Diversifying the search is particularly important during the first cycles in order to avoid premature convergence—search stagnation—towards local optima. As pointed out previously, this exploratory ability of ants can be emphasized both by decreasing α and ρ. However, in this way the algorithm converges more slowly and it takes longer to find a solution.

In this section, we first study the evolution of the influence of pheromone on ants during processing with respect to the cost of administrating it. We then propose to introduce a preprocessing step that favors a larger exploration of the search space at a lower cost, so that more solutions can be found in a shorter time.

A. *Impact versus cost of pheromone*

In order to favor a larger exploration, and more particularly at the beginning of the search, Ant-Solver derives its features from the $\mathcal{MAX}-\mathcal{MIN}$ Ant System, that is, pheromone trails are limited to an interval $[\tau_{min}, \tau_{max}]$ and are initialized to τ_{max}. Hence, after n cycles of Ant-Solver, the pheromone trail $\tau(i,j)$ on edge (i,j) is such that $(1-\rho)^n * \tau_{max} \leq \tau(i,j) \leq \tau_{max}$, where ρ is the pheromone evaporation parameter and usually has a value very close to 0. As a consequence, during the first cycles, pheromone trails have very similar values, and do not significantly influence the computation of transition probabilities.

Figure 5 illustrates this on four runs of Ant-Solver on a same random binary problem, with different values for α and ρ. This figure displays the evolution of the dispersion of the colony, that is, the average distance between all pairs of constructed assignments for each cycle (where the distance between two assignments is defined as the number of different variable-value assignments). At the beginning of the search, the dispersion of the colony is very high: the assignments constructed take approximately 85% of different values. During this first phase, ants are not influenced

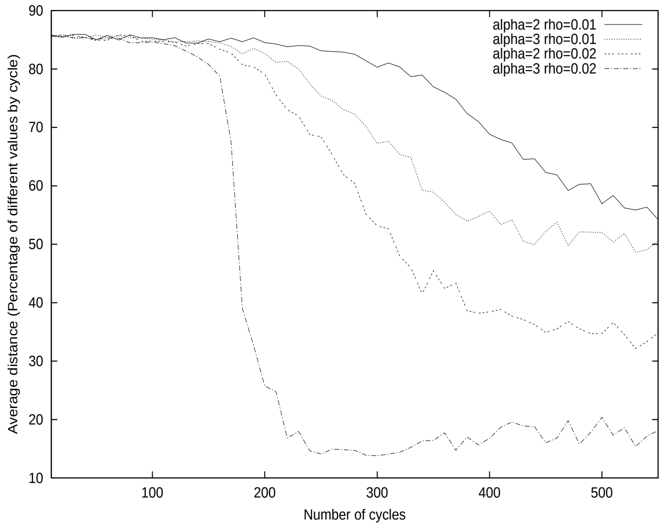

Fig. 5. Evolution of the dispersion of ants. Each curve displays results obtained for one run on a same $< 100, 8, 0.14, 0.23 >$ instance, with different values of α and ρ.

by pheromones, and they explore the search space widely. Then, after a significant number of cycles, the average distance decreases progressively. During this second phase, ants are increasingly influenced by pheromones so that they converge and focus on a smaller area of the search space. Note that, for low values of α and ρ, the exploration phase is longer: when $\alpha = 2$ and $\rho = 0.01$, the dispersion starts to decrease slowly after 200 cycles, whereas when $\alpha = 3$ and $\rho = 0.02$ it decreases quickly and, after 200 cycles, it has already reached a very low value.

However, if the influence of pheromone is reduced deliberately at the beginning of the search in order to favor exploration, administrating pheromone is tedious and time consuming. Indeed, let us consider a CSP (X, D, C), and let $n = \mid X \mid$ be the number of variables and $q = \sum_{X_i \in X} \mid D(X_i) \mid$ be the number of vertices of the associated construction graph. To construct a complete assignment, the computation of pheromone values for all transition probabilities requires $\mathcal{O}(n * q)$ operations. Then, at the end of each cycle, pheromone laying requires $\mathcal{O}(n^2)$ operations for each rewarded assignment and pheromone evaporation requires $\mathcal{O}(q^2)$ operations. The other operations, that do not deal with pheromone, are the selection of variables, the computation of heuristic factors and the local repair phase. The complexity of these operations mainly depends on the kind of constraints considered (e.g., global constraints, binary constraints, numerical constraints). However, when choosing appropriate data structures that allow incremental computations, the complexity of these operations is usually an order lower than $\mathcal{O}(q^2)$. Actually, when q increases, most of the computation time is used to administrate pheromone. For example, on $< 100, 8, 0.14, 0.23 >$ random binary CSPs, assignments are constructed nearly three times faster when pheromone is not used.

procedure preprocessing
compute n_{Best} assignments and store them in *SampleSet*
repeat
$OldCost \leftarrow \sum_{\mathcal{A} \in BestOf(n_{Best}, SampleSet)} cost(\mathcal{A})$
compute n_{Best} assignments and add them to *SampleSet*
$NewCost \leftarrow \sum_{\mathcal{A} \in BestOf(n_{Best}, SampleSet)} cost(\mathcal{A})$
until $NewCost/OldCost > 1 - \epsilon$ **or** a solution has been found

Fig. 6. Preprocessing step.

B. Description of the preprocessing step

The introduction of a preprocessing step is motivated by the fact that pheromone is rather expensive to administrate, whereas it has an effect on ants' search only after some 100 or so cycles. The idea is to collect a significant number of assignments by performing "classical" local search, that is, by iteratively constructing complete assignments, without using pheromone, and repairing them. For easy problems that are far enough from the phase transition region, local search usually finds solutions quickly, so that this preprocessing step stops iterating on a success and the whole algorithm terminates. However, for harder problems within the phase-transition region, local search more often fails in finding a solution. In this case, the goal of the preprocessing step is to collect a representative set of assignments, thus constituting a kind of sampling of the search space. Then, we select from this sample set the best assignments and we use them to initialize pheromone trails. Finally, we continue the search with Ant-Solver.

We shall call *SampleSet* the set of all assignments constructed during the preprocessing step, n_{Best} the number of best assignments that are selected from *SampleSet* to initialize pheromone trails, and $BestOf(n_{Best}, SampleSet)$ the n_{Best} best assignments contained in *SampleSet*.

To determine the number of assignments that should be collected in *SampleSet*, one has to find a trade-off between computing a large number of assignments, that is more representative of the search space, and a smaller number of assignments, that is more quickly computed. Experiments show that the harder a problem is, the more assignments should be computed. Therefore, instead of fixing the number of assignments that should be computed in *SampleSet*, we only fix n_{Best}, the number of assignments that will be selected from *SampleSet* to initialize pheromone trails, and we introduce a limit ϵ on the quality improvement of these best assignments.

The preprocessing procedure is more precisely described in Figure 6. This procedure iteratively collects assignments in *SampleSet* until the average cost of the n_{Best} best assignments in *SampleSet* has not been improved by more than a percentage given by ϵ. The assignments are computed by applying local search on complete assignments that are constructed according to the algorithm in Figure 2, but where values are chosen with respect to quality factors only, that is, the probability of choosing a value v for a variable X_j is $p_{\mathcal{A}}(<X_j, v>) = \eta_{\mathcal{A}}(<X_j, v>)^{\beta} / \sum_{w \in D(X_j)} \eta_{\mathcal{A}}(<X_j, w>)^{\beta}$.

p_2	Success rate		Nb assign. built		CPU time spent	
	During prepro	After prepro	During prepro	After prepro	During prepro	After prepro
0.19	99.5	0.5	92	1	0.4	0.1
0.21	48.6	51.1	1401	197	11.0	6.1
0.23	0.5	95.0	1619	704	15.7	21.4
0.25	2.1	96.1	1502	355	15.3	11.0
0.27	65.7	34.3	763	8	8.3	0.4

TABLE I

Results obtained by Ant-Solver with preprocessing on $< 100, 8, 0.14, p_2 >$ CSPs (with n_{Best}=200 and ϵ=0.02). For each value of p_2, the table successively displays the percentage of runs that have succeeded during preprocessing, the percentage of runs that have succeeded after preprocessing, the number of assignments constructed during preprocessing, the number of assignments constructed after preprocessing, the time spent for preprocessing and the time spent after preprocessing.

Finally, at the end of the preprocessing step, if no solution is found, the n_{Best} best assignments of *SampleSet* are used to initialize pheromone trails: the pheromone trail on each edge (i, j) of the construction graph is set to

$$\tau(i,j) \leftarrow \sum_{\mathcal{A}_k \in BestOf(n_{Best}, SampleSet)} \Delta\tau(\mathcal{A}_k, i, j)$$

where $\Delta\tau(\mathcal{A}_k, i, j)$ is the quantity of pheromone deposited on edge (i, j) for assignment $\mathcal{A}_k$ as defined in section III.

C. Experimental study of the preprocessing step

The preprocessing step is parameterized by n_{Best}, the number of assignments that are extracted from the sample set to initialize pheromone trails, and ϵ, the limit on the quality improvement of the n_{Best} best assignments of the sample set. We studied the influence of these two parameters on $< 100, 8, 0.14, p_2 >$ instances (with the other parameters set to α=2, β=10, ρ=0.01 and *nbAnts*=8). We performed experiments with different values of n_{Best} (ranging from 50 to 400) and ϵ (ranging from 0.005 to 0.03). With respect to running times, the best results are obtained with the highest values of ϵ and the smallest values of n_{Best}, on all the classes of problems. With respect to success rates, the best results are usually obtained with the smallest values of ϵ and the largest values of n_{Best}. However, we noticed that success rates often decrease when $n_{Best} \geq 300$. A good trade-off between running time and success rate appears to be n_{Best}=200 and ϵ=0.02.

Table I displays results obtained with Ant-Solver with MCH local search and preprocessing when n_{Best}=200 and ϵ=0.02. One can notice that for easy problems many solutions are found during preprocessing, whereas for harder problems within the phase-transition region, only few solutions are found during preprocessing. Also, the number of assignments constructed during preprocessing increases with the hardness of instances. More experimental results are given in the next section.

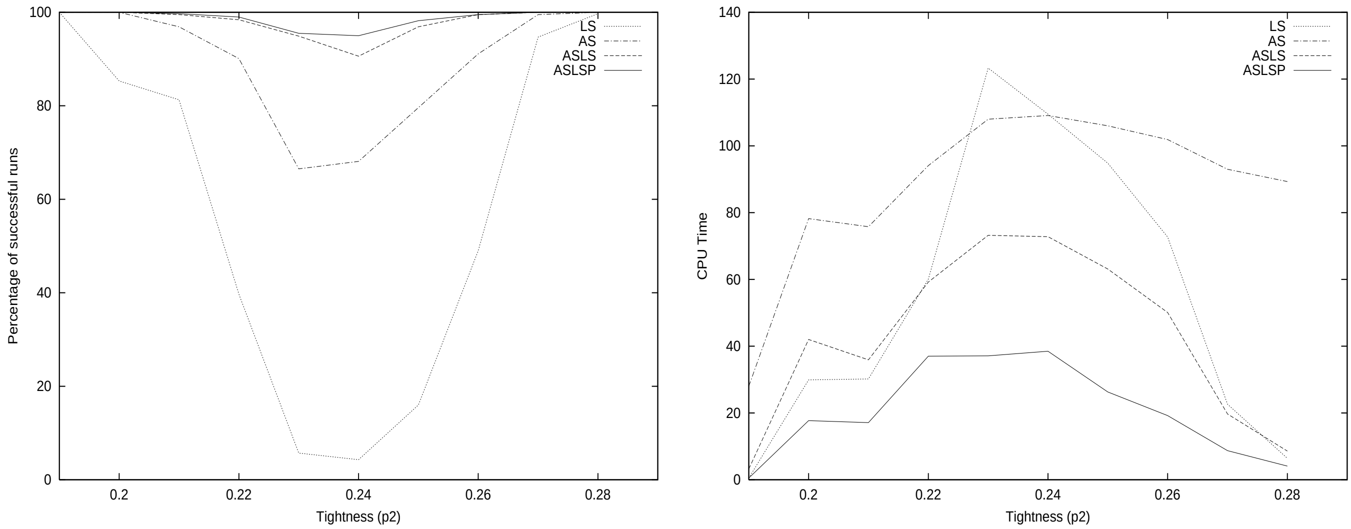

Fig. 7. Comparison of MCH local search (LS), Ant-Solver (AS), Ant-Solver with MCH local search (ASLS) and Ant-Solver with MCH local search and preprocessing (ASLSP) on $< 100, 8, 0.14, p_2 >$ CSPs (average results on 300 instances for each value of p_2). Left curves display success rates w.r.t. p_2. Right curves display CPU time (average on successful runs) w.r.t. p_2.

VI. Experimental results

We now experimentally compare the different algorithms that we have presented in this paper, that is, Ant-Solver (AS), MCH local search (LS), Ant-Solver combined with MCH local search (ASLS), and Ant-Solver combined with MCH local search and preprocessing (ASLSP). All these algorithms are implemented in C++. To solve a new CSP with one of these algorithms, one only has to implement a C++ class that describes the problem to be solved. This class specifies the number of variables, the number of values associated with each variable, the evaluation function (which returns the number of new unsatisfied constraints when assigning a value to a variable with respect to a partial assignment), the variable selection function, and the local search procedure.

For AS, ASLS, and ASLSP, we set the parameters to α=2, β=10, ρ=0.01, and *nbAnts*=8. For LS, ASLS and ASLSP, we used a local search procedure based on the min-conflicts heuristic that stops iterating when the number of violated constraints is not improved after $\mid X \mid$ successive iterations. For ASLSP, we set n_{Best} to 200 and ϵ to 2%. For all algorithms, we used the smallest domain ordering to select at each step the next variable to assign. Finally, we limited all runs to 200 seconds of CPU time. All experiments were performed on a PC with an AMD Athlon Processor, 256 Mb RAM, and 900 MHz.

Figure 7 displays some experimental results obtained on $< 100, 8, 0.14, p_2 >$ random binary CSPs [1]. When considering success rates, one can remark that ASLSP is slightly more successful than ASLS. This shows that the preprocessing step allows to solve a few more instances. Then, ASLSP and ASLS are both much more successful than AS. This shows that combining local search with Ant-Solver improves the solution process. Finally, MCH Local Search is far less successful than AS and it has hardly solved four percent of the hardest instances, when $p_2 = 0.24$.

When considering the time criterion, one can remark that ASLS and ASLSP are always much more efficient than AS. The benefit of providing ants with local search abilities is even more remarkable for tightly constrained instances that are beyond the phase-transition region, when $p_2 > 0.24$: for these instances, local search is usually capable of finding a solution after the construction of very few assignments while ants slowly converge to the solution. Also, ASLSP is always much faster than ASLS. This shows that preprocessing speeds up the solution process.

Experimental comparison with random walk

The local search procedure considered in our first experiments is based on the greedy min-conflicts heuristic, incapable of escaping from local optima. This local search procedure gives good results for instances that are far enough from the phase-transition region. However, it gives very poor results for instances within the phase-transition region, as the search space landscape of these instances contains many more local optima [33], [3].

To help it escape from local optima, greedy local search can be combined with different metaheuristics. We now compare our approach to the random walk MCH local search, denoted in the following by RWLS, which is one of the best-performing extensions of MCH [24], [32]. At each step, the RWLS procedure either randomly changes the value of a variable, with probability p_{noise}, or repairs a conflicting variable using the min-conflicts heuristic, with probability $1 - p_{noise}$. We set the probability p_{noise} of performing a random move to 0.07 [2].

First experiments on $< 100, 8, 0.14, 0.23 >$ instances

[1] We have also performed experiments on $< 100, 8, 0.05, p_2 >$ and $< 200, 8, 0.05, p_2 >$ binary CSPs and on graph colouring problems with 3 colours and from 200 to 600 vertices. On all these problems we obtained results very similar to those displayed in Figure 7.

[2] We made experiments with different values of p_{noise} ranging from 0.01 to 0.1. The best results were obtained when $p_{noise} = 0.07$.

Time	50	100	150	200	250	300	350
Results on $< 100, 8, 0.3, 0.13 >$ CSPs							
ASLS	0	13	72	94	98	100	100
ASLSP	4	59	91	96	99	100	100
RWLS	87	99	100	100	100	100	100
Results on $< 100, 8, 0.5, 0.08 >$ CSPs							
ASLS	0	0	15	79	95	97	98
ASLSP	0	23	87	98	99	99	99
RWLS	11	19	26	34	45	51	73
Results on $< 100, 8, 0.7, 0.06 >$ CSPs							
ASLS	0	0	5	21	51	78	93
ASLSP	0	4	24	56	81	94	99
RWLS	10	17	25	33	38	46	51

TABLE II

Comparison of Ant-Solver with MCH Local Search (ASLS), Ant-Solver with MCH Local Search and Preprocessing (ASLSP) and Random Walk MCH Local Search (RWLS). Each line successively displays the percentage of successful runs after 50, 100, 150, 200, 250, 300 and 350 seconds of CPU Time (over 100 runs).

showed that RWLS clearly outperforms Ant-Solver: indeed, RWLS always finds a solution much faster than Ant-Solver. However, the RWLS peformance is drastically degraded when the number of constraints is increased. This is illustrated in Table II, that reports experimental results obtained when increasing the connectivity probability p_1—that determines the number of constraints. We considered three classes of instances (all of them having 100 variables and 8 values in each variable domain): when $p_1 = 0.3$, instances have around 1500 constraints, when $p_1 = 0.5$, instances have around 2500 constraints and when $p_1 = 0.7$, instances have around 3500 constraints. For each value of p_1, we considered instances within the phase-transition region by setting p_2 in such a way that the constrainedness κ is equal to 1 (that is, when $p_1 = 0.3$, $p_2 = 0.13$, when $p_1 = 0.5$, $p_2 = 0.08$, and when $p_1 = 0.7$, $p_2 = 0.06$).

Table II shows the evolution of success rates within successive running time limits by reporting the percentage of successful runs every 50 seconds. From this table, one can remark that, when $p_1 = 0.3$, Random Walk finds solutions much faster than Ant-Solver: RWLS solves 87 % of the instances in less than 50 seconds, and 100 % in less than 150 seconds, whereas ASLSP requires 300 seconds to solve all the instances. However, when the number of constraints increases, RWLS becomes much less successful. For example, when $p_1 = 0.7$, RWLS solves 10 % of the instances, in less than 50 seconds, but then the success rate increases very slowly so that it has hardly solved half of the instances after 350 seconds. For these instances, ASLSP is much more successful and is capable of solving 99 % of them within 350 seconds.

VII. Conclusion

We described in this paper Ant-Solver, a generic algorithm for solving CSPs based on the ACO metaheuristic. Pheromone is used to learn the global desirability of a set of variable-value assignments, and it is used as a heuristic for choosing values when constructing an assignment. Experiments reported in this paper show that ants "alone", without local search abilities, are able to solve hard instances, although they are rather slow at finding solutions.

This ACO algorithm may be combined with any local search procedure, thus boosting performances of both approaches. For all experiments reported in this paper, we used the min-conflicts heuristic for local search. Further work will concern the integration in Ant-Solver of other greedy local search heuristics (such as GSAT [25]), but also of local search procedures capable of escaping from local minima (such as tabu search [15] or random walk). Indeed, experimental comparisons of Ant-Solver with random walk reported in section VI show that these approaches have complementary features: random walk can find solutions to some problems very quickly, while it fails solving others; Ant-Solver requires more time to converge, but it actually solves many more instances. Therefore, the combination of random walk local search with Ant-Solver could allow one to take advantage of both features.

In this paper, we also introduced a preprocessing step that favors, at a lower cost, a larger exploration of the search space at the beginning of the search. The idea is to initialize pheromone trails with respect to a representative sample of the search space. We showed on random binary CSPs that this preprocessing step improves the solution process, both with respect to success rates and running times. This preprocessing step could be introduced as well in other ACO algorithms, and it would be interesting to evaluate its use on other problems, like the traveling salesman problem or the quadratic assignment problem.

A main motivation of this work was to provide a generic tool, that can be used to solve any CSP: to solve a new problem with Ant-Solver, one only has to implement a C++ class that describes the variables, their domains, the evaluation function, and the local search procedure. However, an efficient implementation of this class is not always straightforward. In particular, one has to introduce appropriate data structures so that the number of constraints violated can be evaluated incrementally when constructing assignments, or when repairing them by local search. Actually, Ant-Solver could be integrated within a modelling language for local search, such as Localizer [22]. Such a language would allow one to design efficient and incremental procedures for local search more easily.

A drawback of the genericity of Ant-Solver is that, on some particular problems, it may be less efficient than specialized algorithms dedicated to these problems. In particular, many CSPs involve global constraints, e.g., "AllDiff" constraints (that constrain a set of variables to take different values) or "AtMost" constraints (that fix the maximal number of variables that may take a given value). To handle such global constraints, one can translate them into an equivalent set of simpler constraints, and then use Ant-Solver. However, one usually obtains better results when using specialized algorithms for these global constraints. For example, in [29] we have described an ACO algorithm for handling permutation constraints (that constrain a set of n variables to be assigned with a permutation of n known values). Many CSPs involve such global permutation constraints, e.g., the n-queens, the all-interval series problem, and the car-sequencing problem [28]. For these particular constraints, we have introduced a dedicated graph that is much smaller than the construction graph used in Ant-Solver, and that takes permutation constraints into account in an a priori way, so that the search space is reduced and solutions are found quicker. Hence, further work will also concern the integration of dedicated graphs within Ant-Solver in order to deal more efficiently with such global constraints.

Acknowledgments

The author would like to thank Marco Dorigo and the referees for their fruitful comments on preliminary versions of this paper.

References

[1] B. Bullnheimer, R.F. Hartl, and C. Strauss. An improved ant system algorithm for the vehicle routing problem. *Annals of Operations Research*, 89:319–328, 1999.
[2] P. Cheeseman, B. Kanelfy, and W. Taylor. Where the *really* hard problems are. In *Proceedings of IJCAI'91, Morgan Kaufmann, Sydney, Australia*, pages 331–337, 1991.
[3] D.A. Clark, J. Frank, I.P. Gent, E. MacIntyre, N. Tomv, and T. Walsh. Local search and the number of solutions. In *Proceedings of CP'96, LNCS 1118, Springer Verlag, Berlin, Germany*, pages 119–133, 1996.
[4] D. Costa and A. Hertz. Ants can colour graphs. *Journal of the Operational Research Society*, 48:295–305, 1997.
[5] A. Davenport. A comparison of complete and incomplete algorithms in the easy and hard regions. In *Proceedings of CP'95 workshop on Studying and Solving Really Hard Problems*, pages 43–51, 1995.
[6] M. Dorigo. *Optimization, Learning and Natural Algorithms* (in Italian). PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1992.
[7] M. Dorigo, G. Di Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5(2):137–172, 1999.
[8] M. Dorigo and G. Di Caro. The Ant Colony Optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw Hill, UK, 1999.
[9] M. Dorigo and L.M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.
[10] M. Dorigo, V. Maniezzo, and A. Colorni. The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, 26(1):29–41, 1996.
[11] E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.
[12] L. Gambardella, E. Taillard, and M. Dorigo. Ant colonies for the quadratic assignment problem. *Journal of the Operational Research Society*, 50:167–176, 1999.
[13] L.M. Gambardella, E.D. Taillard, and G. Agazzi. MACS-VRPTW: A multiple ant colony system for vehicle routing problems with time windows. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 63–76. McGraw Hill, London, UK, 1999.
[14] I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proceedings of AAAI-96, AAAI Press, Menlo Park, California*, 1996.
[15] F. Glover and M. Laguna. Tabu search. In *Modern Heuristics Technics for Combinatorial Problems*, pages 70–141. Blackwell Scientific Publishing, Oxford, UK, 1993.
[16] J.K. Hao and R. Dorne. Empirical studies of heuristic local search for constraint solving. In *Proceedings of CP'96, LNCS 1118, Springer Verlag, Berlin, Germany*, pages 194–208, 1996.
[17] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of International Conference on Genetic Algorithms, Morgan Kaufmann, Sydney, Australia*, pages 184–192, 1995.
[18] H.R. Lourenço, O. Martin, and T. Stützle. Iterated local search. In F. Glover and G. Kochenberger, editors, *Handbook of metaheuristics*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002, in press.
[19] E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. Random constraints satisfaction: theory meets practice. In *CP98, LNCS 1520*, pages 325–339. Springer Verlag, Berlin, Germany, 1998.
[20] V. Maniezzo and A. Colorni. The Ant System applied to the quadratic assignment problem. *IEEE Transactions on Data and Knowledge Engineering*, 11(5):769–778, 1999.
[21] P. Merz and B. Freisleben. Fitness landscapes and memetic algorithm design. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 245–260. McGraw Hill, UK, 1999.
[22] L. Michel and P. Van Hentenryck. Localizer: a modeling language for local search. In *Proceedings of CP'97, LNCS 1330, Springer Verlag, Berlin, Germany*, 1997.
[23] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satistaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.
[24] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of IJCAI'94, MIT Press, Cambridge, MA*, pages 337–343, 1994.
[25] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of AAAI'92, AAAI Press, Menlo Park, California*, pages 440–446, 1992.
[26] T. Shiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of IJCAI-95, MIT Press, Cambridge, MA*, pages 631–637, 1995.
[27] B. Smith. Succeed-first or fail-first: A case study in variable and value ordering heuristics. In *Third Conference on the Practical Applications of Constraint Technology PACT'97, http://www.scs.leeds.ac.uk/bms/papers.html*, pages 321–330, 1997.
[28] B. Smith. Dual models of permutation problems. In *CP01*, volume 2239 of *LNCS*, pages 615–619. Springer Verlag, Berlin, Germany, 2001.
[29] C. Solnon. Solving permutation constraint satisfaction problems with artificial ants. In *Proceedings of ECAI'2000, IOS Press, Amsterdam, The Netherlands*, pages 118–122, 2000.
[30] T. Stützle and H.H. Hoos. $\mathcal{MAX}-\mathcal{MIN}$ Ant System. *Journal of Future Generation Computer Systems*, 16:889–914, 2000.
[31] E.P.K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London, UK, 1993.
[32] R. Wallace and E. Freuder. Heuristic methods for over-constrained constraint satisfaction problems. In *OCS'95: Workshop on Over-Constrained Systems at CP'95, http://citeseer.nj.nec.com/wallace95heuristic.html*, Cassis, Marseilles, 1995.
[33] M. Yokoo. Why adding more constraints makes a problem easier for hill-climbing algorithms: analyzing landscapes of CSPs. In *Proceedings of CP'97, LNCS 1330, Springer Verlag, Berlin, Germany*, pages 356–370, 1997.