Solving Permutation Constraint Satisfaction Problems with Artificial Ants

Christine Solnon[1]

Abstract. We describe in this paper Ant-P-solver, a generic constraint solver based on the Ant Colony Optimization (ACO) metaheuristic. The ACO metaheuristic takes inspiration on the observation of real ants collective foraging behaviour. The idea is to model the problem as the search of a best path in a graph. Artificial ants walk trough this graph, in a stochastic and incomplete way, searching for good paths. Artificial ants communicate in a local and indirect way, by laying a pheromone trail on the edges of the graph.

Ant-P-solver has been designed to solve a general class of combinatorial problems, i.e., permutation constraint satisfaction problems, the goal of which is to find a permutation of n known values, to be assigned to n variables, under some constraints. Many constraint satisfaction problems involve such global permutation constraints. Ant-P-solver capabilities are illustrated, and compared with other approaches, on three of these problems, i.e., the n-queens, the all-interval series and the car sequencing problems.

1 Introduction

Real ants are able to collectively find solutions to complex problems, such as searching the shortest path between two points. Moreover, they can dynamically take into account changes in their environment, like the sudden appearance of an unexpected obstacle. This collective behaviour is possible thanks to a local and indirect communication mean, i.e., pheromone trails. Pheromone is a volatile hormone. While walking, ants deposit a certain amount of pheromone, thus forming a pheromone trail. When looking for their way, they probabilistically prefer to follow a trail rich in pheromone rather than a poorer one. The collective capability of ants to find short paths mainly comes from the fact that the shorter the path, the quicker ants come back to the nest along this path and deposit pheromone upon it, and the more ants further choose this path, in an autocatalytic process.

The ant colony optimization metaheuristic

This behaviour, that allows ants to collectively solve hard problems, gave rise to artificial ant algorithms. These algorithms were first proposed in [6, 10] as a multi-agent approach to solve hard combinatorial optimization problems. The idea is to model the problem as the search of a best path in a graph. Artificial ants walk through the graph, looking for good paths.

Some features of the behaviour of artificial ants are inspired from real ants. In particular, artificial ants lay pheromone on the edges they follow. They choose their way to go in a probabilistic way, depending on the amount of pheromone previously left on edges. In order to avoid premature convergence, the amount of pheromone is progressively decreased, simulating some kind of evaporation.

Artificial ants also have some extra-capabilities which do not find their counterpart in real ants. In particular, artificial ants can be associated with data structures which contain the memory of their previous actions. In most cases, pheromone trails are only updated after having generated a complete path, and not during the walk. The amount of pheromone left is usually a function of the quality of the path, whereas real ants nearly always deposit the same amount of pheromone. Finally, the probability for an artificial ant to choose an edge usually not only depends on pheromone trails but also on some problem-specific local visibility.

These main features of artificial ants behaviour are introduced as the "Ant Colony Optimization (ACO) metaheuristic" in [7, 8], and have inspired different ant algorithms, that allowed solving hard optimization problems such as the traveling salesman problem [9], the quadratic assignment problem [11] or the vehicle routing problem [2].

Motivations

To solve a new optimization problem within the ACO metaheuristic, one mainly has to model the problem as the search of a best path through a graph. A key point is to define the graph and the stochastic transition rule that is locally used by ants to choose their path. In particular, one has to take into account the constraints of the problem in order to ensure that ants only perform consistent paths with respect to these constraints. Actually, for hard-constrained problems, this modeling phase becomes a challenge.

In this paper, we investigate the ACO metaheuristic capabilities for solving constraint satisfaction problems: the goal is no longer to optimize an objective function, under some constraints, but to find an assignment which actually satisfies all the constraints (or maximizes the number of satisfied constraints). A main motivation underlying this work is to provide a generic tool which can be used to solve a whole set of problems. We more particularly focus in this paper on the class of permutation constraint satisfaction problems, called PermutCSPs.

Definition of PermutCSPs

A PermutCSP consists of finding a permutation of n known values, to be assigned to n variables, under some constraints. Many constraint satisfaction problems involve such permutation constraints, also called cycling or sequencing constraints.

Definition of a CSP: A constraint satisfaction problem (CSP)[20] is defined by a triple (X, D, C) such that $X = \{X_1, X_2, ..., X_n\}$ is

[1] LISI, University Lyon 1 and INSA de Lyon, bât. 710, 69622 Villeurbanne cedex, France, e-mail: csolnon@bat710.univ-lyon1.fr

a finite set of n variables, D is a function which maps every variable $X_i \in X$ to its domain $D(X_i)$, and C is a set of constraints.
A solution of a CSP (X, D, C) is an assignment for all the variables in X which satisfies all the constraints in C.

Definition of a PermutCSP: A PermutCSP is a particular CSP, such that all the solutions of the CSP are permutations of a given tuple. Such a problem will be defined by a quadruple (X, D, C, P) such that (X, D, C) is a CSP and $P =< v_1, v_2, \ldots, v_n >$ is a tuple of $|X| = n$ values.
A solution of a PermutCSP is a complete assignment

$$A = \{X_1 \leftarrow r_1, X_2 \leftarrow r_2, ..., X_n \leftarrow r_n\}$$

which is a solution of (X, D, C) and such that $< r_1, r_2, \ldots, r_n >$ is a permutation of P.

For example, the 4-queens problem, the goal of which is to place 4 queens on a 4×4 chessboard so that no queen can be attacked, can be defined by the following PermutCSP

$$\begin{array}{lcl} X & = & \{X_1, X_2, X_3, X_4\} \\ D(X_i) & = & \{1, 2, 3, 4\} \;\; \forall X_i \in X \\ C & = & \{\mid X_i - X_j \mid \neq \mid i - j \mid : \; (X_i, X_j) \in X^2, \; i \neq j\} \\ P & = & < 1, 2, 3, 4 > \end{array}$$

A solution to this problem is the permutation $< 2, 4, 1, 3 >$ of P, such that the first queen, on row 1, is on column 2, the queen on row 2 is on column 4, the queen on row 3 is on column 1 and the queen on row 4 is on column 3.

2 Description of Ant-P-solver

In this section, we define Ant-P-solver, an incomplete and stochastic solver based on the ACO metaheuristic of [7] and which can be used to solve any PermutCSP in a generic way. The idea is to associate a graph —called the PermutGraph— with the PermutCSP to solve. A colony of artificial ants walk through this PermutGraph, searching for "good" paths, corresponding to solutions of the PermutCSP.

PermutGraph associated with a PermutCSP: The PermutGraph associates a vertex with each value of the tuple P to be permuted. There is an extra vertex corresponding to the nest, from which ants will start their paths. Hence, the PermutGraph associated with a PermutCSP (X, D, C, P) where $P =< v_1, v_2, \ldots, v_n >$ is the complete oriented graph $G = (V, E)$ such that:

$$\begin{array}{lcl} V & = & \{nest, v_1, v_2, \ldots, v_n\} \\ E & = & \{(v_i, v_j) \in V^2\} \end{array}$$

Ants deposit pheromone on edges of the PermutGraph; the amount of pheromone laying on an edge (v_i, v_j) is noted $\tau(v_i, v_j)$.

Definition of a path: A path in a PermutGraph $G = (V, E)$ is a sequence of vertices of V. We only consider elementary paths, that do not contain cycles. A path starting from a vertex v_i and arriving to a vertex v_j is noted $\pi = v_i \rightsquigarrow v_j$.

Paths are built by ants: ants start from the nest, and successively visit each of the n other vertices of the graph, building thus a hamiltonian path. The path built by an ant corresponds to a complete assignment: the value associated with the i^{th} vertex of the path is assigned to the i^{th} variable of the PermutCSP.

Evaluation of a path: The goal of Ant-P-solver is to find a complete assignment which minimizes the number of violated constraints. Therefore, the fewer constraints are violated in a path, the better it is. Hence, the evaluation of a path π, denoted by $eval(\pi)$, is the number of constraints that are not satisfied in the assignment corresponding to the path π. A complete path which evaluates to 0 is a solution of the PermutCSP.

For example, let us consider again the 4-queens problem. The vertices of the associated PermutGraph are $V = \{nest, 1, 2, 3, 4\}$. Let us now consider an ant that, starting from the nest, successively goes to vertices 2, 3, 1 and 4. The corresponding path is:

$$\pi =< nest, 2, 3, 1, 4 >$$

This path corresponds to the complete assignment:

$$A = \{X_1 \leftarrow 2, X_2 \leftarrow 3, X_3 \leftarrow 1, X_4 \leftarrow 4\}$$

This complete assignment violates one constraint, so that the path π is evaluated to $eval(\pi) = 1$.

Definition of transition probabilities: While searching for a path, an artificial ant chooses the next vertex to visit among the set of vertices it has not yet visited, in a stochastic way with respect to transition probabilities. The probability for an ant that has already performed a path π to go to a vertex v_j is defined proportionally to the attraction capability $\mathcal{A}_\pi(v_j)$ of the vertex v_j:

$$p_\pi(v_j) = \frac{\mathcal{A}_\pi(v_j)}{\sum_{v_k \in V} \mathcal{A}_\pi(v_k)}$$

The attraction capability of a vertex both depends on the pheromone previously layed on the edge leading to this vertex, and on the "local visibility" of the ant. Hence, the attraction capability $\mathcal{A}_\pi(v_j)$ of vertex v_j for an ant that has already performed the path $\pi = nest \rightsquigarrow v_i$, is defined by:

$$\begin{array}{lcll} \mathcal{A}_\pi(v_j) & = & 0 & \text{if } v_j \in \pi \\ \mathcal{A}_\pi(v_j) & = & \tau(v_i, v_j) * (local\_eval(\pi, v_j))^\beta & \text{if } v_j \notin \pi \end{array}$$

where β is a parameter which controls the relative weight of local visibility with respect to the pheromone trail. The local visibility evaluates the goodness of the next vertex v_j and is inversely proportional to the number of new unsatisfied constraints when assigning the value v_j to the next variable, i.e.,

$$local\_eval(\pi, v_j) = \frac{1}{1 + eval(\pi . < v_j >) - eval(\pi)}$$

Path function: The function "path(G)" which computes a complete path starting from the nest in a graph G is:

<u>function</u> path$(G = (V, E))$ <u>returns</u> a path
<u>begin</u>
    $\pi \leftarrow < nest >$
    Candidates $\leftarrow V - \{nest\}$
    <u>While</u> $Candidates \neq \emptyset$ <u>do</u>
        choose $v_j \in Candidates$ with probability $p_\pi(v_j)$
        add v_j at the end of the path π
        remove v_j from Candidates
    <u>return</u>(π)
<u>end</u>

Ant-P-solver function: The function “Ant-P-Solver(G, ρ)”, which returns the best path found by artificial ants within the graph G is described below. The parameter ρ controls the pheromone evaporation rate. At each cycle of this algorithm, each ant a_i computes a path π_i. If this path is better than $best\pi$, the best path found so far since the beginning of the search, then $best\pi$ is updated. At the end of each cycle, a pheromone trail is left on the edges of the best path π found during this cycle. The amount of deposited pheromone is proportional to the goodness of this path, i.e., it is inversely proportional to *eval*(π), the number of violated constraints in π. Finally, pheromone is uniformly decreased on all edges of the graph. The algorithm stops cycling either when an ant has found a solution, or when a maximum number of cycles have been performed.

```
function Ant-P-Solver(G = (V, E), ρ) returns a path
    nbCycles ← 0
    min ← ∞
    for any pair of vertices (v_i, v_j) ∈ V^2 do τ(v_i, v_j) ← 1
    while nbCycles < nbMaxCycles and min > 0 do
        nbCycles ← nbCycles + 1
        /* Each ant a_i computes a path π_i */
        for each ant a_i do
            π_i ← path(G)
            if eval(π_i) <min then min ←eval(π_i)
                                   bestπ ← π_i
        /* The best ant of the current cycle leaves pheromone */
        let π be the best path found during the current cycle
        for each edge (v_i, v_j) of the path π do
            τ(v_i, v_j) ← τ(v_i, v_j) + min/eval(π)
        /* Pheromone evaporates */
        for any pair (v_i, v_j) ∈ V^2 do τ(v_i, v_j) ← τ(v_i, v_j) * ρ
    return(best π)
```

Parameter values: Ant-P-solver is parameterized by the local visibility weight β, the evaporation rate ρ and the number of ants. These parameters have been fixed in an experimental way, by running Ant-P-solver with different parameter values, on different PermutCSPs, and selecting the parameter values which gave the best average results. One should notice that we used the same parameter values for solving all the problems detailed in the next section.

β has been fixed to 10. With smaller values, ants globally find worse paths, so that it takes longer to reach the optimal solution, whereas with greater values, ants are not enough sensitive to the pheromone trails, and are not able to improve the resolution process.

The number of ants has been fixed to 8. With smaller values, the best path found at each cycle is not good enough, so that the algorithm converges more quickly, but on hard problems it converges towards a sub-optimal path. On the other hand, with a greater number of ants, the best path found at each cycle is not much better than the one found with 8 ants, and therefore, the collective resolution is not greatly enhanced, whereas each cycle is longer to perform.

Finally, the evaporation rate has been fixed to $\rho = 0.99$, so that if the pheromone trail laying on an edge is equal to 1, and if no more ants go trough this edge, then the pheromone trail becomes insignificant in a few hundred cycles.

3 Solving PermutCSPs with Ant-P-solver

Ant-P-solver has been implemented in C++. To solve a new PermutCSP with Ant-P-solver, one only has to implement a C++ class which actually describes the problem to be solved. This class mainly specifies the initial tuple P of values to permute, and the function “local_eval(π, v_j)” which returns the number of new unsatisfied constraints when assigning the value v_j to the next variable, with respect to the partial assignment corresponding to path π.

In this section, we illustrate our approach on three PermutCSPs, and we compare it with Ilog solver, a state of the art tree-search based solver. In the next section, we compare Ant-P-solver with some other related approaches.

All experiments, for both Ant-P-solver and Ilog solver, have been performed on the same machine, i.e., a HP 715/100 station. Ant-P-solver has been run ten times on each problem instance, and the results displayed are an average of these ten runs.

3.1 The n-queens problem

Results obtained for solving the n-queens problem, from 50 to 200 queens, are displayed in figure 1. These results illustrate the feasability of our approach, but they are not very interesting: on small instances, Ilog-solver is much faster, whereas on larger instances, results are comparable. Actually, the n-queens problem is very often used to illustrate CSP solvers, mainly because it is easy to specify and implement. However, as pointed out in [20], benchmarks on this problem must be interpreted with caution, as it has very specific features. In particular, each value assignment for every variable conflicts with at most three values of each other variable, whatever the number of queens is. Therefore, constraints get looser as the number of queens grows larger.

3.2 The all-interval series problem

This problem is described as prob007 in [13]. The goal is to find a permutation of the first n integer numbers, so that the absolute differences of any consecutive pairs of numbers are all different. Tree-search based solvers, like Ilog-solver or CHIP, that use global constraints can find without search a first solution to this problem [18], corresponding to the following regular sequence:
$< 0, (n-1), 1, (n-2), 2, (n-3), \ldots >$
However, even with state of the art tree-search solvers, finding another solution remains a challenge for values of n greater than 16.

The results obtained for solving instances of the all-interval series problem, from 10 to 24 variables, are displayed in figure 1. For the Ilog program, we used the `IlcDistribute` global constraint to express that both the n variables, and the interval values, must be all different, and we searched for a second solution, as the first solution is always found without search. On this problem, Ant-P-solver is clearly much more efficient than Ilog-solver. These results show that a stochastic incomplete approach, provided that it can handle global constraints, is more suitable than a complete approach for solving this problem.

3.3 The car sequencing problem

The car-sequencing problem, described as prob001 in [13], consists of scheduling cars along an assembly line: there are n cars to be produced, that are grouped into k car types. All the cars within a same car type require the same set of options. Each option i is associated with a capacity constraint, represented as a ratio p_i/q_i, which specifies that for any sequence of q_i consecutive cars in the assembly line, at most p_i of them may require that option. The goal is to find a permutation of the n cars which satisfies all the capacity constraints.

N-queens problem

n	Ant-P	Ilog
50	2.7s	0.6s
75	6.8s	3.1s
100	18.1s	0.4s
125	87.6s	0.8s
150	144.5s	555.1s
175	256.0s	>3 600.0s
200	282.6s	256.6s

All-interval series problem

n	Ant-P	Ilog
10	0.0s	0.1s
12	0.1s	1.1s
14	0.5s	29.3s
16	2.0s	882.4s
18	3.7s	>3 600.0s
20	10.4s	>3 600.0s
22	20.6s	>3 600.0s
24	65.0s	>3 600.0s

Car sequencing problem

	Ant-P-solver		Ilog solver	
Instances	Success	Time	Success	Time
car-60	100%	1.7s	40%	17.7s
car-70	100%	2.5s	20%	17.8s
car-80	100%	4.4s	60%	19.0s
car-85	98%	33.1s	50%	23.5s
car-90	57%	119.8s	80%	229.7s

Figure 1. Experimentations report

Utilization percentages and ordering heuristics: The hardness of a car sequencing problem instance depends on the number n of cars to be produced and the number k of car types, but also on the utilization percentage of the different options. The utilization percentage of an option i corresponds to the ratio of the number m_i of cars requiring option i with respect to the maximum number of cars in a sequence which could have option i while satisfying the capacity constraint on i (i.e., $100 * m_i * q_i/(n * p_i)$) [5]. A high utilization percentage indicates that the demand is very close to the capacity. Hence, [19] and [17] introduced value ordering heuristics: the idea is to assign first the cars requiring options with high utilization percentages, corresponding to "critical" options. Such value ordering heuristics have been incorporated in Ant-P-solver in a very straightforward way, by increasing the attraction capabilities of the vertices corresponding to cars which require "critical" options, i.e., proportionally to their associated utilization percentages.

Experimentations report: We consider fifty problem instances (provided by J. Lee [15]), grouped into five sets of ten instances each (respectively called car-60, car-70, car-80, car-85 and car-90), with respect to their mean utilization percentages (respectively of 60 %, 70%, 80%, 85% and 90%). All these instances are feasible, and have 200 cars to permute, 5 options, and from 17 to 30 different car types. Figure 1 displays the results obtained with Ant-P-solver and Ilog solver. Ant-P-solver has been limited to 5000 cycles (corresponding roughly to 1000 seconds of cpu time), whereas Ilog-solver has been limited to 3600 seconds of cpu time. Within these limits, the solvers were not always able to find a solution. Hence, we report for each solver, along with the cpu-time, the percentage of successful runs. For the Ilog program, we used the `IlcSequence` global constraint, which is dedicated to this kind of sequencing problem, and we used ordering heuristics, as described in [17]. For Ant-P-solver, we used similar value ordering heuristics.

On "easy" instances of this problem, with utilization percentages ranging between 60% and 80%, one can remark that Ant-P-solver is more efficient, and much more successfull: Ant-P-solver never failed, whereas Ilog-solver failed on more than half of the problems. Indeed, these instances are not enough constrained to allow the filtering algorithm used by Ilog-solver to reduce the search space. In this case, an incomplete stochastic approach is clearly more appropriate. As a counterpart, for the harder instances of car-90, the filtering algorithm becomes more effective, and Ilog-solver is more successful than Ant-P-solver, whereas the cpu time becomes comparable.

Infeasible instances: Ant-P-solver explores the search space in a stochastic and incomplete way. Therefore, it cannot be used to prove inconsistency of infeasible instances. However, it can be used to search for a complete assignment which minimizes the number of violated constraints. In a more general way, Ant-P-solver can be used to solve "max-CSPs", i.e., over-constrained problems, the goal of which is to find an assignment which maximizes the number of satisfied constraints. Let us consider for example instance `pb19/71` of the car sequencing problem extracted from [13]. The feasibility of this instance remained open in [17], i.e., Ilog solver can neither find a solution, nor prove infeasibility, on this particular instance within a reasonable amount of time. [12] proves that this instance is actually infeasible, and that any complete assignment violates at least two capacity constraints. On this instance, Ant-P-solver has found an assignment that violates 2 capacity constraints, in 81s cpu time. This assignment is actually optimal, even though Ant-P-solver cannot be used to prove this optimality.

4 Related works

Global vs local constraints: Generally speaking, to solve a CSP that contains permutation constraints, one had better to use a global constraint which usually handles it more efficiently. Indeed, [5] shows that using such global constraints always reduces the search space, even though it can be slower to actually solve the problem (this case usually happens on easy instances).

The advantage of using global constraints is well illustrated by the all-interval series problem. Indeed, [14] describes the formulation of the all-interval series problem as a SAT problem, and then uses it to compare different local search algorithms. The results show that the instance with 12 variables causes great difficulties even for the very best local search methods for SAT. Actually, the difficulty probably does not come from the kind of approach considered, but from the fact that the global permutation constraints are lost by the SAT encoding, so that the search space to be explored becomes much larger.

Hence, many constraint satisfaction solvers introduced global constraints. The way of tackling these global constraints depends on the kind of approach considered, i.e., complete tree-search, incomplete repair-based or genetic approaches.

In complete tree search approaches, the search space is explored in an exhaustive way, until either a solution is found, or the problem is proved to have no solution. In order to reduce the search space, some propagation techniques are usually applied. The idea is to verify, at each node of the search, that the problem satisfies some partial consistency (e.g., arc-consistency). This pruning is performed by a filtering algorithm which removes inconsistent values from variables domains, with respect to the considered partial consistency [20].

Dedicated filtering algorithms have been defined for handling global permutation constraints, e.g., the `IlcDistribute` and `IlcSequence` constraints of Ilog solver [17] and the `cycle` constraint of CHIP [1]. In the previous section, we showed that such filtering algorithms are actually effective to solve the more difficult instances of the car sequencing problem. However, on less constrained problems, like the all-interval series problem or easier instances of the car-sequencing problem, the search space is not sufficiently reduced by the filtering algorithms and remains too large to be explored in a complete way. In this case, an incomplete approach, like Ant-P-solver, is more suitable.

Repair-based approaches are incomplete and (usually) stochastic approaches which work on complete inconsistent assignments, and repair them gradually towards a consistent solution. On hard combinatorial problems, they usually find an approximately optimal solution in fairly quick time. As a counterpart, they do not guarantee finding the optimal solution, nor can they prove inconsistency.

GENET [4] is a repair-based approach which uses a variation of the min-conflicts heuristic: it escapes from local minima by increasing the weight of the violated constraints. The idea behind this is to "learn" critical constraints that are hard to satisfy, by making them prioritary. Two extensions of GENET have been proposed that allow to handle global permutation constraints:

- In SWAPGENET [5], the idea is to select possible moves from a "swap" neighbourhood, so that the considered complete assignments always satisfy the permutation constraint in an a priori way. SWAPGENET has been illustrated on the car-sequencing problem. The given cpu-times are comparable to the one obtained with Ant-P-solver. Nothing is reported about the percentage of successful runs.
- In E-GENET [15], the idea is to work on tuples of values and to associate penalty values to tuples with respect to constraint satisfaction. Experimentations of E-GENET on the car-sequencing problem show that it is less successful than Ant-P-solver (73% of successful runs for car-80, nothing is reported about car-90). Cpu-times are not reported.

Genetic algorithms are incomplete and stochastic approaches which take inspiration from natural evolution. Exploration of the search space is achieved through selection, cross-over and mutation operators upon a population (which represent candidate solutions).

GAcSP [21] is a solver which combines a genetic algorithm with local repair-based technics to solve the car-sequencing problem. In this approach, global permutation constraints are handled by a greedy repair function, which ensures that offsprings created at each generation (by cross-over operations) actually satisfy permutation constraints. After repair, each offspring is hill-climbed by a swap function (similar to the one used in [5]). Experiments show that this approach actually allows to solve "easy" instances of the car sequencing problem, with low utilization percentages. However, with higher utilization percentages, the number of successful runs is severely decreased: on instances with 200 cars and with an utilization percentage of 80%, only 9% of the runs succeed, in 7289 seconds.

5 Conclusion

We have defined in this paper Ant-P-solver, an incomplete and stochastic approach for solving permutation constraint satisfaction problems which is based on the ACO metaheuristic.

This work could be enhanced in many different ways. In particular, we could introduce repair-based technics in order to improve the best solution found at each cycle, and speed up the convergence process. Moreover, Ant-P-solver could be parallelized in a very straightforward way: the idea would be to execute different Ant-P-solver processes, each of them working on a different pheromon matrix, while the best solutions found by each solver could be exchanged. A similar approach, where genetic algorithms cooperate with an ant algorithm is proposed in [3].

Furthermore, we propose in [16] a generalization of this approach to the resolution of any CSP. The graph used by ants associates a vertex with each variable-value pair. The transition rule uses a local evaluation function similar to the one used in Ant-P-solver. This solver, called Ant-solver, can be used to solve any CSP in a generic way. We more particularly illustrate its capabilities on the propositionnal satisfiability (SAT) problem.

REFERENCES

[1] N. Beldiceanu and E. Contejean, 'Introducing global constraints in CHIP', *Journal of Mathematical and Computer Modelling*, **20**(12), 97–123, (1994).

[2] 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).

[3] P.R. Calégari, *Parallelization of population-based evolutionary algorithms for combinatorial optimization problems*, Ph.D. dissertation, Ecole polytechnique fédérale de Lausanne, 1999.

[4] A. Davenport, E. Tsang, Kangmin Zhu, and C. Wang, 'Genet: a connectionist architecture for solving constraint satisfaction problems by iterative improvement', in *Proceedings of AAAI'94*, pp. 325–330, (1994).

[5] A.J. Davenport and E.P.K. Tsang, 'Solving constraint satisfaction sequencing problems by iterative repair', in *Proceedings of the first international conference on the practical applications of constraint technologies and logic programming (PACLP)*, pp. 345–357, (1999).

[6] M. Dorigo, *Learning and Natural Algorithms (in italian)*, Ph.D. dissertation, Politecnico di Milano, 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 metaheuristic', in *New Ideas in Optimization*, eds., D. Corne, M. Dorigo, and F. Glover, 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] L. Gambardella, E. Taillard, and M. Dorigo, 'Ant colonies for the quadratic assignment problem', *Journal of the Operational Research Society*, **50**, 167–176, (1999).

[12] I.P. Gent, 'Two results on car-sequencing problems', Technical report (http://www.apes.cs.strath.ac.uk/apesreports.html), APES, (1998).

[13] I.P. Gent and T. Walsh, 'Csplib: a benchmark library for constraints', Technical report, APES-09-1999, (1999). available from http://csplib.cs.strath.ac.uk/. A shorter version appears in CP99.

[14] H. Hoos, *Stochastic Local Search - Methods, Models, Applications*, Ph.D. dissertation, TU Darmstadt, 1998.

[15] J.H.M. Lee, H.F. Leung, and H.W. Won, 'Performance of a comprehensive and efficient constraint library using local search', in *11th Australian JCAI*, LNAI, Springer-Verlag, (1998).

[16] S. Pimont and C. Solnon, 'A generic ant algorithm for solving constraint satisfaction problems', Technical report, LISI, (2000).

[17] J.-C. Regin and J.-F. Puget, 'A filtering algorithm for global sequencing constraints', in *CP97*, volume 1330 of *LNCS*, 32–46, Springer-Verlag, (1997).

[18] H. Simonis and N. Beldiceanu, 'A note on csplib prob007', Technical report, Cosytech, (1998).

[19] 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*, pp. 321–330, (1996).

[20] E.P.K. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[21] T. Warwick and E. Tsang, 'Tackling car sequencing problems using a genetic algorithm', *Journal of Evolutionary Computation - MIT Press*, **3**(3), 267–298, (1995).