Local search with constraint propagation and conflict-based heuristics
Narendra Jussien, Olivier Lhomme

To cite this version:
Narendra Jussien, Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. Artificial Intelligence, Elsevier, 2002, 139 (1), pp.21-45. hal-00312750

HAL Id: hal-00312750
https://hal.archives-ouvertes.fr/hal-00312750
Submitted on 26 Aug 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Local search with constraint propagation and conflict-based heuristics

Narendra Jussien
École des Mines de Nantes – BP 20722
F-44307 NANTES Cedex 3 – FRANCE
Narendra.Jussien@emn.fr

Olivier Lhomme
ILOG – Les Taissouinières HB2
1681 route des Dolines – F-06560 VALBONNE – FRANCE
olhomme@ilog.fr

Abstract
In this paper, we introduce a new solving algorithm for Constraint Satisfaction Problems (CSP). It performs an overall local search helped with a domain filtering technique to prune the search space. Conflicts detected during filtering are used to guide the search. First experiments with a tabu version of the algorithm have shown good results on hard instances of open shop scheduling problems. It competes well with the best highly specialized algorithms.

Introduction
Many industrial and engineering problems can be modeled as constraint satisfaction problems (CSP). A CSP is defined as a set of variables each with an associated domain of possible values and a set of constraints over those variables.

Most of constraint solving algorithms are built upon backtracking mechanisms. Those algorithms usually explore the search space systematically, and thus guarantee to find a solution if one exists. Backtracking-based search algorithms are usually improved by using some relaxation techniques (usually called filtering techniques in the CSP world) which aim at pruning the search space in order to decrease the overall duration of the search.

Another series of constraint solving algorithms are local search based algorithms (eg. min-conflict (Minton, Johnston, & Laird 1992), GSAT (Selman, Levesque, & Mitchell 1992), tabu search (Glover & Laguna 1993)). They perform a probabilistic exploration of the search space and therefore cannot guarantee to find a solution, but may be far more efficient (wrt reponse time) than systematic ones to find a solution.

Several works have studied cooperation between local and systematic search (Yokoo 1994; Pesant & Gendreau 1996; David 1997; Schaerf 1997; Gervet 1998; Richards & Richards 1998). Those hybrid approaches have led to good results on large scale problems. Three categories of hybrid approaches can be found in the literature:

1. performing a local search before or after a systematic search;
2. performing a systematic search improved with a local search at some points of the search (typically for optimisation problems, to try to improve the quality of a solution);
3. performing an overall local search, and using systematic search either to select a candidate neighbor or to prune the search space.

The hybrid approach presented in this paper falls in the third category. It uses filtering techniques to both prune the search space and help in choosing the neighbor in a local search. This leads to a generic search technique over CSP which is called path-repair. An implementation of path-repair which merges a tabu search together with a filtering technique and conflict-based heuristics to guide the search is described in this paper. That implementation has been used to solve open shop scheduling problems. It has given very good results on hard instances well competing with the best highly specialized algorithms. This was quite surprising since, unlike those specialized algorithms, our implementation is general and does not need any tuning of complex parameters.

The paper is organized as follows. The following section presents the notation used. Next we introduce the path-repair algorithm. Then related works are discussed. Finally first results obtained in the field of open shop scheduling problems are presented.

Preliminaries
A CSP is a pair \( \langle V, C \rangle \) where \( V \) is a set of variables and \( C = \{ c_1, \ldots, c_m \} \) a set of constraints. The domains of the variables are handled as unary constraints.

For a given constraints set \( S = \{ c_1, \ldots, c_k \} \), \( S \) will be the logical conjunction of the constraints in \( S \): \( S = \{ c_1 \land \ldots \land c_k \} \). By convention: \( \emptyset = true \).

Classical CSP solving simultaneously involves a filtering algorithm, to \textit{a priori} prune the search tree, and an enumeration mechanism, to overcome the incompleteness of that filtering algorithm. For example, for binary CSP over finite domains, arc-consistency can be used as filtering technique. After a filtering step, three situations may arise:

1. the domain of a variable becomes empty: there is no feasible solution;

2. all the domains are reduced to a singleton: those values assigned to their respective variables provide a feasible solution for the considered problem;
3. there exists at least one domain which contains two values or more: the search has not yet been successful. In a classical approach, it would be time for enumeration through a backtracking-based mechanism.

In a more general way, a filtering algorithm $\Phi$ applied on a set $C$ of constraints returns a new set $C' = \Phi(C)$ such that $C \subseteq C'$ (Note that we consider domain reductions as addition of redundant constraints). Moreover, for any filtering algorithm $\Phi$ applied on the set $C$ of constraints of a given CSP, there exists a function $\text{obviousInference}$ which, when applied on $C' = \Phi(C)$, answers:

- $\text{noSolution}$ iff it is immediate to infer that no solution can be find for $C'$ (as in situation 1 above).
- $\text{solution}$ iff the current constraint system can immediately provide a solution that verifies all the constraints in $C'$ (as in situation 2 above).
- $\text{flounder}$ in all other situations (as in situation 3 above).

The function $\text{obviousInference}$ has typically a low computational cost. Its aim is to make explicit the use of some properties that depends on the filtering algorithm that is used. The example of arc-consistency filtering with an empty domain or with only singleton domains has already been given, but a function $\text{obviousInference}$ can be made explicit in many other filtering or pruning algorithms. For example, in integer linear programming, the aim is to find an optimal integer solution. This can be done by using the simplex algorithm over the reals. If there is no real solution or if the real optimum has only integer values, then an $\text{obviousInference}$ function would respectively return $\text{noSolution}$ or $\text{solution}$.

Enumerating discrete binary CSP is assigning a value $a$ to a variable $v$ i.e. adding a new constraint $v = a$. For other kinds of problems, enumerating may be different: for example, when dealing with scheduling problems, enumerating is often adding a precedence constraint between two tasks of the problem. We will call those constraints $\text{decision}$ constraints.

In the next section, the path-repair algorithm is presented through an abstraction of the solved problems: they may be discrete binary CSP, numeric CSP as well as scheduling problems. This will be possible thanks to:

1. the parameter $\Phi$ which represents the filtering algorithm used;
2. the function $\text{obviousInference}$, tightly related to the used filtering algorithm, that is able to examine a set of constraints in order to continue the computation or not;
3. the concept of $\text{decision}$ constraint.

The path-repair algorithm

The idea of the path-repair algorithm is very simple. First observe that:

- current local search algorithms mainly work upon a total instantiation of the variables;
- backtracking-based search algorithms work upon a partial instantiation of the variables.

The ability of backtracking-based search algorithms to be combined with filtering techniques only comes from the fact that they work upon a partial instantiation of the variables. Thus, a local search algorithm working upon a partial instantiation of the variables would have the same ability. Indeed, the path-repair algorithm is such an algorithm. The considered partial instantiation is defined by a set of decision constraints (as described above) on the variables of the problem. Such a constraint set defines a path in the search tree.

Principles of path-repair

The principles of the path-repair algorithm as shown in figure 1 are the following: let $P$ be a path in the search tree. At each node of that path, a decision constraint has been added. Let $C_P$ be the set of added decision constraints while moving along $P$.

The path-repair algorithm starts with an initial path (it may range from the empty path to a path that defines a complete assignment). The main loop first checks the conditions of failure. A filtering algorithm is then applied on $C \cup C_P$ giving a new set of constraints $C' = \Phi(C \cup C_P)$. The function $\text{obviousInference}$ is then called over $C'$. Three cases may occur:

- $\text{obviousInference}(C') = \text{solution}$: a solution has been found. The algorithm terminates and returns $C'$.
- $\text{obviousInference}(C') = \text{noSolution}$: the algorithm terminates.
- $\text{obviousInference}(C') = \text{flounder}$: the path-repair algorithm tries to extend the current path $P$ by adding a decision constraint. That behavior is similar to

```
procedure path-repair(C)
{1} begin
{2} repeat
{3} if conditions of failure verified then
{4} return failure
{5} else
{6} C' ← $\Phi(C \cup C_P)$
{7} switch obviousInference(C')
{8} case noSolution :
{9} k ← nogood explaining the failure
{10} P ← neighbor(P, k, Γ)
{11} case solution :
{12} return C'
{13} default :
{14} P ← extend(P, Γ)
{15} endswitch
{16} endif
{17} until false
{18} end
```

Figure 1: The path-repair algorithm

\footnote{These conditions depend on the instance of the algorithm; examples are given in the following sections.
that of backtracking-based search algorithms. For that
purpose, a function $\text{extend}(P, \Gamma)$ is assumed to exists
that chooses a decision constraint to be added and adds
it to $P$. Parameter $\Gamma$ can be used to store a context that
varies according to the chosen version of the algorithm.
Its meaning will be made clear later.

- obviousInference $(C') = \text{noSolution}$: $C \cup C_P$ is inconsistent. We will say that $P$ is a dead-end, or $P$ is inconsistent: $P$ cannot be extended. The path-repair algorithm will thus try to repair the current path by choosing a new path through the function neighbor $(P, k, \Gamma)$. Parameter $k$ as $\Gamma$ will be explained later.

The path-repair algorithm appears here as a search method that handles partial instantiations and uses filtering techniques to prune the search space. The key components of this algorithm are the neighboring computation functions (neighbor) and the extension functions (extend).

**Nogoods in path-repair**

In a local search algorithm such as GSAT (on boolean CSP), an inconsistent instantiation is replaced by a new one built by negating the value of one of its variables. That variable is heuristically chosen (eg. selecting the one whose negation will allow the greatest number of clauses to become satisfied). More generally, a local search algorithm uses complete instantiations (called states) and replaces an inconsistent state with another state chosen among its neighbors.

The path-repair algorithm works in the same way except that it uses partial instantiations (paths): as soon as a path becomes inconsistent, one of its neighbors needs to be chosen. A path (partial instantiation) synthetizes all the included complete instantiations. Switching paths is like setting aside many irrelevant complete instantiations in one movement.

Like any local search algorithm, path-repair may use a heuristic way to select an interesting neighbor. It seems to be a good idea to select a neighboring path $P'$ which does not have the drawbacks of the current path $P$ (recall that in path-repair, neighbors of path $P$ are computed iff $P$ is inconsistent). Ideally, we would like to get to a consistent neighbor $P'$ i.e. such that obviousInference $(\Phi(C \cup C_P')) = \text{solution}$. However, that is equivalent to solve the whole problem.

Instead, we may try to get to a partially consistent neighbor $P'$ i.e. such that obviousInference $(\Phi(C \cup C_P')) \neq \text{noSolution}$. Unfortunately, the only way to get there (without using computing resources) is to get back to an already explored node but, doing so, we would achieve a kind of backtracking mechanism, what is not wanted in the path-repair algorithm.

Nevertheless, what can be done is to avoid the neighbors that can already be known as inconsistent. Such an information can be extracted from an inconsistent path $P$. Indeed, inconsistency means that $\hat{C} \land \hat{C}_P \Rightarrow \text{false}$. It is possible to compute a subset of $C_P$ that is alone inconsistent with $C$. Such a subset is called a conflict set or nogood.

**Definition 1 (Nogood)** A nogood $k$ for a set of constraints $C$ and a path $P$, is a set of constraints such that: $k \subseteq C_P$ and $\hat{C} \land k \Rightarrow \text{false}$.

Now, we can define a neighbor $P'$ of a path $P$ according to a single nogood $k$. As long as constraints in the computed nogood $k$ remain altogether in a given path $P'$, that path will remain inconsistent. Therefore, in order to get a path with some hope to be consistent, we need to remove from the current path $P$ at least one of the constraints in $k$.

Indeed, a more precise neighborhood can be computed. Let $c \in k$ be a constraint to be removed from $C_P$. As long as all the constraints in $k \setminus c$ remain in the active path, $c$ will never be satisfiable. Thus, the negation of $c$ can be added in the new path. A neighbor of a path $P$ according to a nogood $k$ is thus defined as follows:

**Definition 2 (Neighbor wrt one nogood)** Let $k$ be a nogood for a path $P$, a neighbor $P'$ of $P$ wrt $k$ verifies $\exists c \in k, C_{P'} = C_P \setminus c \cup \{-c\}$

**Computing nogoods** Note that if the current path $P$ is inconsistent, $C_P$ is a valid nogood. Obviously, a strict subset will be much more interesting and will give a more precise neighborhood. A minimal (for the inclusion) nogood would be the best, but could be expensive to compute (Verfaillie & Lobjois 1999). The current implementation does not try to find such a minimal nogood. Instead, it tries to find a good nogood in a fast way.

Nogoods are provided by the filtering algorithm as soon as it can prove that no solution exists in the subsequent complete paths derived from the current partial path. In filtering based constraint solving algorithms, a contradiction is raised as soon as the domain of a variable $v$ becomes empty. Suppose that, for each value (or set of values) $a_i$ removed from the domain of $v$, a set of decision constraints $k_i \subseteq C_P$ is given ($k_i$ is called a removal explanation for $a_i$) and is such that: $\hat{C} \land k_i \Rightarrow v \neq a_i$. If so, $k = \bigcup k_i$ is a nogood since no value for $v$ is allowed by the union of the $k_i$. Therefore, in order to compute nogoods, it is sufficient to be able to compute an explanation for each value (or set of values) removal from the domain of the failing variable.

Value removals are direct consequences of the filtering algorithms. Therefore, value removal explanations can be easily computed by using a trace mechanism embedded within the filtering algorithm and memorizing the reason why a removal is done (eg. see (Jussien & Lhomme 1998)).

For example, let us consider two variables $v_1$ and $v_2$ whose domains are both $\{1, 2, 3\}$. Let $c_1$ be the constraint: $v_1 \geq 3$ and let $c_2$ be the constraint: $v_2 \geq v_1$. Let us assume that the used filtering algorithm is arc-consistency filtering. The constraint $c_1$ explains the fact that $\{1, 2\}$ should be removed from $v_1$. Afterwards, $c_2$ leads to remove $\{1, 2\}$ from $v_2$. An explanation of the removal of $\{1, 2\}$ from $v_2$ will be: $c_1 \land c_2$ because $c_2$ makes that removal only because previous removals occurred in $v_1$ due to $c_1$.

**Tabu path-repair**

In a local search algorithm, the neighbor selection is very important. Many heuristics may be used. That is also the
function neighbor(P, k, Γ)  
% precondition: k ⊂ C_P, P covers Γ
(1) begin
(2)  add k to the list of nogoods Γ
(3)  if sizeof(Γ) > s then
(4)    remove the oldest element of Γ
(5)  endif
(6)  L ← ordered list (decr. weight) of constraints in k
(7)  repeat
(8)    remove the first constraint c from L
(9)    P' ← P \ {c} ∪ {¬c}
(10)  if C_P' covers all nogoods in Γ then
(11)    end
(12)  return P'
(13)  until L empty
(14) return stop (or extend the neighborhood)
(15) end

Figure 2: The neighbor function for tabu path-repair

case in path-repair.

The tabu version of path-repair uses a tabu list of a given size s. The s last computed nogoods are kept in a list Γ. The following invariant is maintained by the algorithm: the current path P covers all the nogoods in Γ, ie does not completely contain any of the nogoods in Γ.

We have defined so far a neighbor wrt one single nogood, so we have to extend the definition when facing multiple nogoods.

Definition 3 (Neighbor wrt several nogoods) A valid neighbor is defined as a path that covers all the nogoods in Γ.

In other words, at least one constraint in each nogood of Γ is not (or is negated) in the new neighbor. To compute such a neighbor in a reasonable time, a greedy algorithm can be used.

Figure 2 shows an implementation of the neighbor function for tabu path-repair that has been used for solving scheduling problems.

The neighbor function has to record in Γ the new nogood k found by the filtering algorithm and to maintain the invariant. It tries to find one constraint in k such that negating this constraint makes the path cover all the nogoods. An integer (weight) is associated with each constraint counting the number of times that the constraint has appeared in any nogood. The neighbor function chooses to negate the constraint with the greatest weight that, when negated, makes the new path cover all the nogoods in Γ. If such a constraint does not exist, the neighborhood can be extended. For example, we may try to negate two constraints. In our implementation for open shop problems (see last section), this case is handled as a stopping criterion, so there is no need for any neighborhood extension.

Note that, in the same way, the function extend(P, Γ) has to use Γ in order to extend the partially consistent current path while maintaining the invariant. A heuristically ordered list of constraints which performs an extension of the current path is dynamically generated and the first constraint that covers all the nogoods in Γ is chosen.

As for now, our algorithm seems to need to call the filtering algorithm many times with little changes to handle in the constraint set. Of course, it would not be very efficient to each time recompute for example the arc-consistency closure from scratch. That problem has been addressed for dynamic CSP. The algorithms used in path-repair are similar to those of (Bessières 1991; Debruyne 1996) or other works (Jussien & Lhomme 1998).

Related works

The path-repair algorithm takes its roots in many other works, among which (Ginsberg & McAllester 1994) has probably been the most influential by highlighting the relationships between local search and systematic search, and by the use of nogoods to guide the search and make it systematic. In the same spirit are (Ginsberg 1993), (Frost & Dechter 1994), (Schiex & Verfaillie 1994) and (Blik 1998).

Two algorithms have been designed that have similarities with the path-repair algorithm:

- The algorithm proposed in (Schaerf 1997) can be seen as an instance of the path-repair algorithm where: the decision contraints are instantiations; there is no propagation and no pruning (the filtering algorithm Φ only consists in checking if the constraints containing only instantiated variables are not violated) and it does not make use of nogoods neither in the the neighbor function nor in the extend function.

The common idea, which already exists in previous works (Jackson 1990), is essentially to extend a partial instantiation when it is consistent, and to perform a local change when the partial solution appears to be a dead-end.

- The idea to use a filtering algorithm during the running of a local search has been also used in (Stuckey & Tam 1998), where an extension to GENET, a local search method based on an artificial neural network aiming at solving binary CSP, is introduced. This extension achieves what is called “lazy arc-consistency” during the search. The lazy arc-consistency filtering performs a filtering over the initial domains. The result is at most the one obtained by filtering the domains before any search. In path repair, the filtering is applied over the current domains at every step.

The heuristic we used to select the decision constraint to negate – choose the one that has appeared the greatest number of times in a nogood – is an adaptation of a similar approach for GSAT counting the number of times that a constraint has not been verified (Selman & Kautz 1993).

The way nogoods are computed by the filtering algorithm is a well-known technique that has already been used with slight variations for different combinations of filtering algorithms with systematic search algorithms (forward checking + intelligent backtracking (Prosser 1993), forward checking + dynamic backtracking (Verfaillie & Schiex 1994), arc-consistency + intelligent backtracking (Codognet, Fages, & Sola 1993), arc-consistency + dynamic backtracking
(Jussien 1997), 2B-consistency + dynamic backtracking (Jussien & Lhomme 1998). Nevertheless, as far as we know, the tabu version of path-repair is the first time such a technique is used in combination with a local search algorithm.

Solving scheduling problems

Classical scheduling shop problems for which a set \( J \) of \( n \) jobs consisting each in \( m \) tasks (operations) must be scheduled on a set \( M \) of \( m \) machines can be considered as CSP upon intervals\(^2\). One of those problems is called the Open Shop problem (Gonzales & Sahni 1976). For that problem, operations for a given job may be sequenced as wanted but only one at a time. We will consider here the building of non preemptive schedules of minimal makespan\(^3\). That problem is NP-hard as soon as \( \min(n, m) \geq 3 \).

Constraints on resources (machines and jobs) are propagated thanks to *immediate selections* from (Carlier & Pinson 1994). The consistency level achieved by that technique does not ensure the computation of a feasible solution. An enumeration step is therefore needed. For shop problems, enumeration is classically performed on the relative order on which tasks are scheduled on the resources. When every possible precedence has been posted, setting the starting date of the variable to their smallest value provides a feasible solution. Such a precedence constraint is therefore a decision constraint as described above.

One of the best systematic search algorithms developed for the Open Shop problem is the branch and bound algorithm presented in (Brucker *et al.* 1994). It consists in adding precedence constraints along the critical path of a heuristic solution in each node. As far as we know, although this is one of the best methods ever, some problems of size \( 7 \times 7 \) remain unsolved.

We tested a tabu version of path-repair. That version is fully systematic due to the used heuristics. Table 1 presents the results obtained on a series of 30 problems from Taillard (1993). In order to put in perspective our results, we recall results presented in (Alcaide, Sicilia, & Vigo 1997) and (Liaw 1998). Those papers present tabu searches specifically developed for the Open Shop problem. Those methods both use carefully chosen complex parameter values. Results presented in table 1 show that our simple approach which merely applies principles presented in this paper already gives very good results. More precisely, the time required to solve those problems is similar to those reported by (Alcaide, Sicilia, & Vigo 1997) and (Liaw 1998). Moreover, for the problems of size \( 4 \times 4 \), the obtained results are the same as those of (Liaw 1998); for \( 5 \times 5 \) problems, if our algorithm gives bad results, they are not very far from the results of (Liaw 1998) but it often gives the same results and even a better one for the 5x5-3 problem; for \( 7 \times 7 \) problems, our algorithms gives the same results as the best of the two others except for four problems among which three are worse (but not much) and one is better (7x7-5). In a few words, our algorithm seems to compete well with those highly customized algorithms.

Our implementation uses a tabu list of size 15. The *neighbor* function is the one given in figure 2. The conditions of failure specifying the exit of the main loop (figure 1) are either a *stop* returned by the *neighbor* function or 1500 iterations without improvement of the last solution reached.

Taillard’s problems are optimization problems. This requires a main loop that calls the function *path-repair* until improvement is no longer possible (see figure 3). Improvements are generated by adding a constraint that specifies that the makespan is less than the current best solution found. The initial path for each call of the function *path-repair* is the latest path (which describes the last solution found).

**Conclusion and future works**

In this paper, we introduced a new solving algorithm for CSP: the *path-repair* algorithm. The two main points of that algorithm are: it makes use of a repair algorithm (local search) as a basis and it works on a partial instantiation in order to be able to use filtering techniques. The most useful tool to implement that algorithm was the use of nogoods: nogoods allow relevant neighborhoods to be considered and nogoods can be used to derive efficient neighbor selecting heuristics for a *path-repair* algorithm.

First experiments with a tabu version of *path-repair* have shown good results over open shops scheduling problems. It competes well with the best highly specialized algorithms. This was quite surprising since, unlike those specialized algorithms, our implementation is general and does not need any tuning of complex parameters. Experiments over other problems are currently being done.

**Acknowledgements**

We would like to thank Christian Bliek for his useful suggestions.

---

\(^2\)Variables are the starting date of the tasks. Bounds thus represent the least feasible starting time and the least feasible ending time of the associated task.

\(^3\)Ending time of the last task.
Table 1: Results on Taillard’s problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
<th>PR</th>
<th>Dist.</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x4.1</td>
<td>193</td>
<td>193</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4.2</td>
<td>236</td>
<td>236</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4.3</td>
<td>271</td>
<td>271</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4.4</td>
<td>250</td>
<td>250</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4.5</td>
<td>295</td>
<td>295</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4.6</td>
<td>189</td>
<td>189</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4.7</td>
<td>201</td>
<td>201</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4.8</td>
<td>217</td>
<td>217</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4.9</td>
<td>261</td>
<td>261</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4.10</td>
<td>217</td>
<td>217</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5x5-1</td>
<td>300</td>
<td>301</td>
<td>0.33 %</td>
<td>301</td>
</tr>
<tr>
<td>5x5-2</td>
<td>262</td>
<td>262</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5x5-3</td>
<td>321</td>
<td>322</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5x5-4</td>
<td>310</td>
<td>311</td>
<td>0.32 %</td>
<td>310</td>
</tr>
<tr>
<td>5x5-5</td>
<td>326</td>
<td>326</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5x5-6</td>
<td>312</td>
<td>314</td>
<td>0.64 %</td>
<td>312</td>
</tr>
<tr>
<td>5x5-7</td>
<td>303</td>
<td>304</td>
<td>0.33 %</td>
<td>303</td>
</tr>
<tr>
<td>5x5-8</td>
<td>300</td>
<td>300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5x5-9</td>
<td>353</td>
<td>356</td>
<td>0.85 %</td>
<td>353</td>
</tr>
<tr>
<td>5x5-10</td>
<td>326</td>
<td>326</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7x7-1</td>
<td>435</td>
<td>435</td>
<td></td>
<td>453</td>
</tr>
<tr>
<td>7x7-2</td>
<td>443</td>
<td>449</td>
<td>1.57 %</td>
<td>447</td>
</tr>
<tr>
<td>7x7-3</td>
<td>468</td>
<td>473</td>
<td>1.07 %</td>
<td>474</td>
</tr>
<tr>
<td>7x7-4</td>
<td>463</td>
<td>463</td>
<td></td>
<td>464</td>
</tr>
<tr>
<td>7x7-5</td>
<td>416</td>
<td>416</td>
<td></td>
<td>417</td>
</tr>
<tr>
<td>7x7-6</td>
<td>451</td>
<td>460</td>
<td>2.00 %</td>
<td>459</td>
</tr>
<tr>
<td>7x7-7</td>
<td>422</td>
<td>430</td>
<td>1.90 %</td>
<td>429</td>
</tr>
<tr>
<td>7x7-8</td>
<td>424</td>
<td>424</td>
<td></td>
<td>424</td>
</tr>
<tr>
<td>7x7-9</td>
<td>458</td>
<td>458</td>
<td></td>
<td>458</td>
</tr>
<tr>
<td>7x7-10</td>
<td>398</td>
<td>398</td>
<td></td>
<td>398</td>
</tr>
</tbody>
</table>

PR: results using patch-vega; restricted to 1500 moves without improvement. Dist. represents the distance to the optimum value. L: results obtained by Liaw with 50 000 moves without improvement and A: results obtained by Alcaide et al. with 100 000 moves without improvement. *: represents unknown values.

References


Debruyne, R. 1996. Arc-consistency in dynamic CSPs is no more prohibitive. In S1/6 Conference on Tools with Artificial Intelligence (TAl96), 299–306.


Gervet, C. 1998. Large combinatorial optimization problem methodology for hybrid models and solutions (invited talk). In JFPLC.


