

The Path-repair algorithm

Narendra Jussien, Olivier Lhomme

▶ **To cite this version:**

Narendra Jussien, Olivier Lhomme. The Path-repair algorithm. Electronic Notes in Discrete Mathematics, 2000, 4, pp.2-16. 10.1016/S1571-0653(05)80102-7 . hal-00312820

HAL Id: hal-00312820
https://hal.science/hal-00312820

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.

The Path-repair algorithm

Narendra Jussien and Olivier Lhomme

École des Mines de Nantes – BP 20722 – F-44307 NANTES *Cedex 3*
{Narendra.Jussien,Olivier.Lhomme}@emn.fr

Abstract

In this paper, we introduce a new solving algorithm for Constraint Satisfaction Problems: 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. Different versions are presented and first experiments with both systematic and non systematic versions show promising results.

1 Introduction

Many industrial and engineering problems can be modeled as constraint satisfaction problems (CSPs). A CSP is defined as a set of variables each with an associated domain of possible values and a set of constraints over the 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 some filtering techniques 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 algorithms. They perform a probabilistic exploration of the search space and therefore cannot guarantee to find a solution. The interest of local algorithms (*eg.* Tabu search [12], GSAT [25]) is that, following local gradients in the search space, they 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 [6,8,20,22,23,30]. 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 [1] 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 will use filtering methods to both prune the search space and help in choosing the neighbor in a local search. This leads to a new search technique over CSPs which is called *path-repair*. Differents variations of this search technique are discussed, some of them are shown to be complete. Very promising first results are presented.

The paper is organized as follows. Section 2 gives some notations. Section 3 presents the *path-repair* algorithm. Section 4 discusses related works and finally section 5 summarizes the first results obtained in the field of open shop scheduling problems.

2 Preliminaries

A CSP is a couple $< V, C >$ where V is a set of variables and $C = \{c_1, \ldots, c_m\}$ a set of constraints. Domains of the variables are handled as unary constraints.

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

Classical CSP solving simultaneously involves a filtering algorithm (to *a priori* prune the search tree) and an enumeration mechanism (to overcome filtering algorithm incompleteness). 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 single value: those values assigned to their respective variables provide a feasible solution for the considered problem;
(3) there exists at least one domain which contains at least 2 values: search has not yet been successful. In a classical approach, it would be time for enumeration through a backtrack-based mechanism.

[1] Note that filtering techniques can be considered as a limited form of systematic search.

In a more general way, for any filtering algorithm[2] Φ applied on the set C of constraints of a given CSP (let $C' = \Phi(C)$), there exists a function `obviousInference` which, when applied on C', answers:

- *noSolution* iff it is immediate to infer that no solution can be find for C (as in situation 1 above).
- *allSolution* iff the current constraints system[3] can immediately provide a solution that verifies all the constraints in C' (as in situation 2 above).
- *flounder* in all other situations (as in situation 3 above).

The function `obviousInference` has typically a low computational cost. Its aim is to make explicit the use of some properties that depends on the used filtering algorithm. The example of arc-consistency filtering with an empty domain or with only singleton domains has already been given, but a function `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 a `obviousInference` function would respectively return *noSolution* or *allSolution*.

Enumerating discrete binary CSPs is assigning a value[4] a to a variable v_k *i.e.* adding a new constraint $v_k = a$ in the system. For other kinds of problems, enumerating may be different: for example, for numeric CSP, enumerating is adding a splitting constraint (*eg.* $v_\ell < a$). When dealing with scheduling problems, enumerating is often adding a precedence constraint between two tasks of the problem.

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 Φ which represents the filtering algorithm used;
(2) the function `obviousInference`, tightly related to the used filtering algorithm, that is able to examine a set of constraint in order to continue or not the computation;
(3) the concept of *enumerating* constraint. An hypothesis holds over the way such constraints are generated: there exists an integer[5] N_e such that

[2] A filtering algorithm Φ applied on a set C of constraints returns a new set $C' = \Phi(C)$ such that $C \subseteq C'$ (redundant constraints may have been added).
[3] We consider that domain reductions are added as redundant constraints in the constraint system.
[4] a is an element of the domain of variable v_k.
[5] For discrete CSPs where enumerating constraint are value assignments N_e is clearly the number of involved variables. For numeric CSPs N_e strongly depends on the

whatever the set E of at least N_e different enumerating constraints, the call `obviousInference`$(\Phi(C \cup E))$ will not answer *flounder*. This condition is necessary to ensure termination (in the case of the systematic version of the algorithm), and is fulfilled by any reasonnable search strategy.

3 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 enumerating constraints (as defined above) upon the variables of the problem. Such a constraint set defines a *path* in the search tree.

3.1 Principles of path-repair

The principle of the *path-repair* algorithm as shown in figure 1 is the following: let P be a path in the search tree. At each node of that path, an enumerating constraint has been added. Let C_P be the set of added enumerating 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*[6]. 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 `obviousInference` is then called over C'. Three cases may occur:

desired precision on the result.

[6] These conditions depend on the instance of the algorithm; examples are given in the following sections.

```
procedure Path-repair(C)
P := initial path
loop
     if conditions of failure verified then
        return failure
     else
        C' := Φ(C ∪ C_P)
     if obviousInference(C') = noSolution then
        let k be a nogood explaining the failure
        P := neighbor(P,k,Γ)
     else if obviousInference(C') = allSolution then
        return C'
     else
        P := extend(P,Γ)
end loop
```

Fig. 1. The *path-repair* algorithm

- `obviousInference`$(C') = allSolution$: a solution has been found. The algorithm terminates and returns C'.
- `obviousInference`$(C') = flounder$: the *path-repair* algorithm tries to extend the current path P by adding an enumerating constraint. That behavior is similar to that of backtracking-based search algorithms. For that purpose, a function `extend`(P,Γ) is assumed to exists that chooses an enumerating constraint to be added and adds it to P. The meaning of parameter Γ will be made clear later.
- `obviousInference`$(C') = noSolution$: $C \cup C_P$ 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,Γ). Parameters k and Γ 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`).

3.2 Properties of the neighborhood of path-repair

In a local search algorithm such as GSAT (on boolean CSPs), an inconsistent instantiation is replaced by a new one built from the first one by negating the value of one of its variables. That variable is choosen by a heuristic (for example: 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. The algorithm can even choose a neighbor in order to implement a systematic search algorithm. Completeness comes from the fact that a path summarizes numerous complete instantiations.

The following sections discuss neighboring path, heuristic choices and specific techniques leading to a systematic algorithm. We previously introduced a parameter Γ in the neighboring computation functions (`neighbor`) and extension functions (`extend`). Γ can be used to store a context that varies according to the chosen version of the algorithm. In the primitive version that is being presented in this paper, that context is not used.

3.3 Neighboring path

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). For example, it would be interesting to get to a consistent neighbor P' *i.e.* such that `obviousInference`$(\Phi(C \cup C'_{P'})) = allSolution$. Obviously, that is not affordable to compute in the general case.

Therefore, we may prefer to get at least to a partially consistent neighbor P' *i.e.* such that `obviousInference`$(\Phi(C \cup C'_{P'})) \neq noSolution$. 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 $\hat{C} \wedge \hat{C}_P \Longrightarrow false$. It is possible to compute a subset of C_P that is alone inconsistent with C. Such a subset will be called a *nogood* [7].

Definition 1 (Nogood) *A nogood k for a set of constraints C and a path*

P, is a set of constraints such that: $k \subset C_P$ and $\hat{C} \wedge \hat{k} \Longrightarrow false$.

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

Note that if 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 it is very expensive to compute one [28]. Therefore, non minimal nogoods will be computed in practice.

As for now, our neighborhood remains very general. In the following, more interesting neighborhoods are described. Our point here is to show that the concept of nogood is crucial for *path-repair*:

- nogoods allow relevant neighborhoods to be considered,
- nogoods can be used to derive efficient neighbor selecting heuristics for a non-systematic *path-repair* algorithm,
- nogoods can be used to derive a complete *path-repair* algorithm.

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 enumerating constraints $k_i \subset C_P$ is given. k_i is called a removal explanation for a_i and is such that: $\hat{C} \wedge \hat{k_i} \Longrightarrow v \neq a_i$. If so, $k = \bigcup_i k_i$ is a nogood since no value for v is allowed by the union of 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 for 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 within the filtering algorithm and memorizing the reason why a removal is done [16].

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: $v_1 \geqslant 3$ and let c_2 be the constraint: $v_2 \geqslant v_1$. Let us assume 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 forces to remove $\{1, 2\}$ from v_2. An explanation of the removal of $\{1, 2\}$ from v_2 will be: $c_1 \wedge c_2$ because c_2 makes that removal only because previous removals occured in v_1 due to c_1.

3.4.1 Heuristic choice of neighbors

In a local search algorithm, the neighbor selection is very important. Many heuristics may be used. For *path-repair* it is the same, different heuristics can be used.

As for now, we have defined a neighbor of a path P according to a nogood k as a path that does not contain at least one constraint from k. Indeed, a more precise neighborhood can be computed. Let c 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 possible neighborhood for an inconsistent set of constraints C_P, according to a nogood $k \subset C_P$ is made from the sets of constraints C_{P_i} different from C_P by the negation of one constraint in k. Let us take an example. Let P be the path $(c_1, c_2, c_3, \neg c_4, c_5)$. Let the nogood k be the set $\{c_2, c_3, \neg c_4\}$. The neighborhood so defined is the set of the three paths $(c_1, \neg \mathbf{c_2}, c_3, \neg c_4, c_5)$, $(c_1, c_2, \neg \mathbf{c_3}, \neg c_4, c_5)$, and$(c_1, c_2, c_3, \mathbf{c_4}, c_5)$.

Now, there remains to specify which neighbor to choose among the above defined neighbors. That degree of freedom for the choice of the constraint in k to be negated allows the use heuristic techniques. In an initial version, we wanted to try to adapt the *min-conflict* heuristic [19] that minimizes the number of unverified constraints. But, when using a filtering algorithm such a mechanism may not be very efficient: the first unverified constraint stops the algorithm.

In our current implementation, an integer (weight) is associated with each constraint counting the number of times that the constraint appeared in a nogood. The heuristic consists in choosing to negate the constraint with the greatest weight. A similar approach counting the number of times that a constraint has not been verified has been successfully used for GSAT [24].

3.4.2 Tabu 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. A valid neighbor is defined as a path that does not completely contain any of 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. It chooses

```
function neighbor(P, k, Γ)
/* precondition: k ⊂ C_P */
add k to the list of nogoods Γ
if sizeOf(Γ) > s then
    remove the oldest element of Γ
L := ordered list (by decreasing weight) of constraints in k
repeat
    remove the first constraint c from L
    P' := P except that ¬c replaces c
    if C_P' covers all nogoods in Γ then
      return P'
until L empty
return stop (or extend the neighborhood)
```

Fig. 2. The `neighbor` function for *tabu path-repair*

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 could be extended (for example, we may try to negate 2 constraints). In our implementation for open shop problems (see section 5), this case is handled as a stopping criterion.

Note that, in the same way, the function `extend(`P`,`Γ`)` should use Γ in order to correctly extend the partially consistent current path.

3.4.3 A systematic instance

Backtracking-based approaches are interesting because they provide systematic search algorithms. Filtering techniques are then used for efficiency reasons. Using filtering techniques is even more interesting within local search algorithms: it can make them more efficient but also complete. Let's see how *path-repair* can become a systematic algorithm.

Nogoods bring that completeness. The easy way is merely to keep all computed nogoods. If during the resolution no valid neighbor exists, the considered problem does not have any feasible solution. Of course, this leads to potentially exponential storage space in order to keep all the nogoods. It is possible to avoid this problem and to keep a polynomial storage space. That is what is done in algorithms such as *dynamic backtracking* [10], *partial order dynamic backtracking* [11] and *general partial order backtracking* [2].

Those algorithms work on an instantiation of the variables which is locally repaired using nogoods. The used recording mechanism requires only polynomial space. For example, considering *dynamic backtracking*:

- Only nogoods for which at most one constraint is not in the current path are kept in Γ;
- the neighbor to be processed is completely deterministic (the chosen enumerating constraint to be undone is the most recent one in the lat encountered nogood).

In *path-repair*, such a nogood recording mechanism can be used thus providing a systematic search algorithm. Such algorithms can be found in a slightly different way in [16] for dynamic CSPs and [17] for numeric CSPs.

4 Related works

The *path-repair* algorithm takes its roots in many other works, among which [9] has probably been the most influential by highlighting the relationships between local and systematic search, and by the use of nogoods to guide the search and make it systematic.

Two algorithms have been designed that have similarities with the non-systematic *path-repair* algorithm (see section 3.4.2):

- The algorithm proposed by Schaerf [23] can be seen as an instance of the *path-repair* algorithm where
  - the enumerating 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),
  - 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 [15], 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 [26], where an extension to GENET, a local search method based on an artificial neural network aiming at solving binary CSPs, 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 way nogoods are computed by the filtering algorithm is a well-known technique that has already been used for different combinations of filtering algorithm with systematic search algorithms (forward checking + intelligent back-

tracking [21], forward checking + dynamic backtracking [29], arc-consistency + intelligent backtracking [5], arc-consistency + dynamic backtracking [16], 2B-consistency + dynamic backtracking [17]. 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.

5 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 CSPs upon intervals[7]. One of those problems is called the Open Shop problem[13]. 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[8]. 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 [4]. 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 an enumerating constraint as defined in section 2.

One of the best systematic search algorithms developed for the Open Shop problem is the branch and bound algorithm presented in [3]. 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 7×7 remain unsolved.

Enumerating techniques used for the Open Shop problem are interesting for *path-repair* because they dynamically build independent sub-problems (by adding precedence constraints). We can suppose that *path-repair* will be able to make profit of that situation.

We first tested systematic versions of *path-repair* on the Open Shop problem. We obtained a very high improvement in terms of number of explored nodes comparing with the results of [3]. Moreover, a problem of size 10×10 has been solved for the first time. Those results have been presented in [14].

[7] Variables are starting date of the tasks. Bounds thus represent the least feasible starting time and the least feasible ending time.
[8] Ending time of the last task.

```
procedure minimize-makespan(C)
P := initial path
bound := +∞
lastSolution := failure
loop
      C := C ∪ "makespan < bound"
      Solution := path-repair(C)
      if Solution = failure then
                  return lastSolution
      else
                  bound := value of makespan in Solution
                  lastSolution := Solution
end loop
```

Fig. 3. Algorithm used to solve Taillard's problems

We also tested a *tabu* version of *path-repair*. Table 1 presents the results obtained on a series of 30 problems from Taillard [27]. In order to put in perspective our results, we recall results presented in [1,18]. 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.

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 "stop"returned by the `neihgbor` function or 1500 iterations 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).

6 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. We showed that the most useful tool to implement that algorithm was the use of *nogoods*.

Problem	Solution	PR	Dist.	L	A
4x4-1	193	193	-	193	-
4x4-2	236	236	-	236	-
4x4-3	271	271	-	271	-
4x4-4	250	250	-	250	-
4x4-5	295	295	-	295	-
4x4-6	189	189	-	189	-
4x4-7	201	201	-	201	-
4x4-8	217	217	-	217	-
4x4-9	261	261	-	261	-
4x4-10	217	217	-	217	-
5x5-1	300	301	0.33 %	300	-
5x5-2	262	262	-	262	-
5x5-3	323	323	-	326	-
5x5-4	310	311	0.32 %	310	-
5x5-5	326	326	-	326	-
5x5-6	312	314	0.64 %	303	-
5x5-7	303	304	0.33 %	303	-
5x5-8	300	300	-	300	-
5x5-9	353	356	0.85 %	353	-
5x5-10	326	326	-	326	-
7x7-1	435	435	-	435	437
7x7-2	443	449	1.35 %	447	444
7x7-3	468	**473**	1.07 %	474	476
7x7-4	463	463	-	463	464
7x7-5	416	**416**	-	417	417
7x7-6	451	460	2.00 %	459	-
7x7-7	422	430	1.90 %	429	429
7x7-8	424	424	-	424	-
7x7-9	458	458	-	458	458
7x7-10	398	398	-	398	398

Table 1
Results on Taillard's problems – **PR** : results using *path-repair* 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.

First experiments with both systematic versions (based upon a managing of the nogoods inspired from *dynamic backtracking*) and non systematic versions (using a tabu list) of *path-repair* have shown promising results.

References

[1] David Alcaide, Joaquín Sicilia, and Daniele Vigo. A tabu search algorithm for the open shop problem. *TOP : Trabajos de Investigación Operativa*, 5(2):283–296, 1997.

[2] C. Bliek. Generalizing partial order and dynamic backtracking. In *Proceedings of AAAI*, 1998.

[3] P. Brucker, J. Hurink, B. Jurisch, and B. Westmann. A branch and bound algorithm for the open-shop problem. Technical report, Osnabrueck University, 12 1994.

[4] Jacques Carlier and Éric Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.

[5] P. Codognet, F. Fages, and T. Sola. A metalevel compiler of CLP(FD) and its combination with intelligent backtracking. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming - Selected Research*, chapter 23, pages 437–456. Massachussetts Institute of Technology, 1993.

[6] Philippe David. A constraint-based approach for examination timetabling using local repair techniques. In *Proceedings of the Second International Conference on the Practice And Theory of Automated Timetabling (Patat'97)*, pages 132–145, Toronto, Canada, August 1997.

[7] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

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

[9] Matthew Ginsberg and David McAllester. GSAT and dynamic backtracking. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer, May 1994. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).

[10] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[11] Matthew L. Ginsberg and David A McAllester. Gsat and dynamic backtracking. In *International Conference on the Principles of Knowledge Representation (KR94)*, pages 226–237, 1994.

[12] F. Glover and M. Laguna. *Modern heuristic Techniques for Combinatorial Problems, chapter Tabu Search, C. Reeves.* Blackwell Scientific Publishing, 1993.

[13] T. Gonzales and S. Sahni. Open-shop scheduling to minimize finish time. *Journal of the Association for Computing Machinery*, 23(4):665–679, 1976.

[14] Christelle Guéret, Narendra Jussien, and Christian Prins. Using intelligent backtracking to improve branch and bound methods: an application to open-shop problems. *European Journal of Operational Research*, page to appear, 1999.

[15] Peter Jackson. *Introduction to Expert Systems.* Readings. Addison Wesley, 1990.

[16] Narendra Jussien. *Relaxation de Contraintes pour les problèmes dynamiques.* 1. thèse, Université de Rennes I, 24 October 1997.

[17] Narendra Jussien and Olivier Lhomme. Dynamic domain splitting for numeric csp. In *European Conference on Artificial Intelligence*, pages 224–228, Brighton, United Kingdom, August 1998.

[18] Ching-Fang Liaw. A tabu search algorithm for the open shop scheduling problem. *Computers and Operations Research*, 26, 1998.

[19] S. Minton, M.D. Johnston, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–206, 1992.

[20] G. Pesant and M. Gendreau. A view of local search in constraint programming. In *Proc. of the Principles and Practice of Constraint Programming*, pages 353–366. Springer-Verlag, 1996.

[21] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993. (Also available as Technical Report AISL-46-91, Stratchclyde, 1991).

[22] E. T. Richards and E. B. Richards. Non-systematic search and learning: An empirical study. In *Proc. of the the Conference on Principles and Practice of Constraint Programming*, Pisa, 1998.

[23] Andrea Schaerf. Combining local search and look-ahead for scheduling and constraint satisfaction problems. In *Proc. of the 15th International Joint Conf. on Artificial Intelligence (IJCAI-96)*, pages 1254–1259, Nagoya, Japan, 1997. Morgan Kaufmann.

[24] Bart Selman and Henry Kautz. Domain-independent extensions to gsat: Solving large structured satisfiability problems. In Ruzena Bajcsy, editor, *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 290–295, Chambery, France, August 1993. Morgan Kaufmann.

[25] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *AAAI 92, Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[26] P.J. Stuckey and V.W.L. Tam. Extending genet with lazy arc consistency. *IEEE Transactions on Systems, Man, and Cybernetics*, 28(5):698–703, 1998.

[27] É. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[28] G. Verfaillie and L. Lobjois. Problemes incohérents: expliquer l'incohérence, restaurer la cohérence. In *Actes des JNPC*, 1999.

[29] Gérard Verfaillie and Thomas Schiex. Dynamic backtracking for dynamic csps. In Thomas Schiex and Christian Bessière, editors, *Proceedings ECAI'94 Workshop on Constraint Satisfaction Issues raised by Practical Applications*, Amsterdam, August 1994.

[30] M. Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *Proceedings of AAAI*, 1994.