ELSEVIER

Artificial Intelligence 139 (2002) 21–45

Artificial Intelligence

www.elsevier.com/locate/artint

Local search with constraint propagation and conflict-based heuristics ☆

Narendra Jussien [a], Olivier Lhomme [b]

[a] *École des Mines de Nantes, BP 20722, F-44307 Nantes Cedex 3, France*
[b] *ILOG, 1681 route des Dolines, F-06560 Valbonne, France*

Received 7 February 2001

Abstract

Search algorithms for solving CSP (Constraint Satisfaction Problems) usually fall into one of two main families: local search algorithms and systematic algorithms. Both families have their advantages. Designing hybrid approaches seems promising since those advantages may be combined into a single approach. In this paper, we present a new hybrid technique. It performs a local search over partial assignments instead of complete assignments, and uses filtering techniques and conflict-based techniques to efficiently guide the search. This new technique benefits from both classical approaches: *a priori* pruning of the search space from filtering-based search and possible *repair* of early mistakes from local search. We focus on a specific version of this technique: `tabu decision-repair`. Experiments done on open-shop scheduling problems show that our approach competes well with the best highly specialized algorithms. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Constraint satisfaction; Conflict-based search; Local search; Hybrid search algorithm; Repair methods

1. Introduction

Many industrial and engineering problems can be modelled as constraint satisfaction problems (CSP). A CSP is defined by a set of variables along with their associated domain of possible values and a set of constraints over those variables. Algorithms for solving

☆ This is an extended version of a prize-winning paper presented at AAAI-2000, Austin, TX, USA.

E-mail addresses: Narendra.Jussien@emn.fr (N. Jussien), olhomme@ilog.fr (O. Lhomme).
URL address: http://njussien.e-constraints.net (N. Jussien).

0004-3702/02/$ – see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0004-3702(02)00221-7

CSP usually fall into one of two main families: systematic algorithms and local search algorithms.

Systematic algorithms for solving CSP typically explore a search tree which is based on the possible values for each of the variables of the solved problem. Such search algorithms start from an empty variable assignment that is extended until obtaining a complete assignment that satisfies all the constraints in the problem. Backtracking occurs when a dead-end is reached. The biggest problem of such backtracking-based search algorithms is that they are typically cursed with early mistakes in the search, i.e., a wrong variable value can cause a whole subtree to be explored with no success. Two ways of improving chronological backtracking are well identified (e.g., see [4] or [15]):

- *Look-back enhancements*, which exploit information about search which has already been performed. Examples of look-back techniques are *backjumping* and *learning* [40, 45,53], which analyse the conditions of failures during the search in order to determine better backtrack points or new constraints.
- *Look-ahead enhancements*, which exploit information about the remaining search space. Such look-ahead techniques are: *filtering techniques*, which allow the early pruning of subparts of the search space that would necessarily lead to a dead-end; as well as heuristics for *variable or value ordering*.

Look-back and look-ahead enhancements reduce the importance of the early-mistake problem, but unfortunately, they cannot solve it completely: to undo a decision, a proof of inconsistency has to be found, which still requires a whole subtree to be explored.

Local search algorithms (e.g., *min-conflict* [34], GSAT [47], *tabu search* [20]) perform an incomplete exploration of the search space by repairing infeasible complete assignments. They do not suffer from the early-mistake problem: as soon as a decision is suspected to lead to a dead-end, it can be undone, without having anything to prove. Local search algorithms are capable of following a local gradient in the search space. They may be far more efficient (w.r.t. response time) than systematic ones to find a first solution. And for optimization problems they can reach a far better quality in a given time frame.

But, local search algorithms cannot guarantee that they find a solution, and may be unable to find one. And thus, they are not the panacea.

Several works have studied cooperation between local and systematic search [2,13,17, 36,43,44,48,54]. Those hybrid approaches have led to good results on large scale problems. Three categories of hybrid approaches can be found in the literature:

- performing a local search before or after a systematic search;
- performing a systematic search improved with a local search at some point of the search: at each leaf of the tree (i.e., over complete assignments) but also at nodes in the search tree (i.e., on partial assignments);
- performing an overall local search, and using systematic search either to select a candidate neighbor or to prune the search space.

In this paper, we present a hybrid approach that falls into the third category. Indeed, our main goal is to show that the look-back and look-ahead enhancements of backtracking-

based algorithms can be exploited for local search algorithms, and can greatly improve their behavior too. This leads us to propose a generic search technique over CSP which is called `decision-repair`. The basic idea is the following: the algorithm starts with a partial solution which is the result of a set of decisions. It first applies a filtering technique. When no inconsistency is detected, the algorithm adds a decision that extends the current partial solution, and the search continues. When a dead-end is reached, we know that there exists an incompatibility between the decisions made so far. The algorithm tries to *repair* that set of decisions. A *conflict* is identified (the smaller the conflict, the better), and the conflict is used to choose a judicious neighbor of the current set of decisions. For example, a judicious neighbor may be obtained by performing a local change on the current set of decisions: negate one of the decisions that occurs in the conflict.

We focus in this paper on an implementation of `decision-repair` which merges a tabu search [20] together with a filtering technique and conflict-based heuristics to guide the search. This new incomplete technique is studied through solving open-shop scheduling problems. We compare it against highly specialized algorithms on hard instances and study its behavior while varying the values of its parameters.

Another contribution of the paper is to emphasize the efficiency of conflict-based heuristics. A new heuristic, which we called a *weighting-conflict* heuristic, was quite successful: each times a conflict is found, a counter is increased for each decision occuring in the conflict. The counters are then used to choose the decision to undo.

This paper is organized as follows. We first recall basic concepts of *Constraint Satisfaction Problems* (Section 2). Then we introduce the `decision-repair` algorithm (Section 3). After a discussion of related works (Section 4), a case-study of a `tabu` instance of `decision-repair` is presented in Section 5.

2. Constraint satisfaction problems

2.1. Definition

A Constraint Satisfaction Problem (CSP) can be described as a pair $\langle V, C\rangle$ where $V = \{v_1, \ldots, v_n\}$ is a set of variables and $C = \{c_1, \ldots, c_m\}$ a set of constraints. A k-ary constraint on k variables $(v_1, \ldots, v_k)$ is a logical formula that defines the allowed combinations of values for the variables $(v_1, \ldots, v_k)$. A constraint may be given in extension or in intension. The domains of the variables (their set of possible values) are expressed as unary constraints.

In the following, we will use the same notation for a set of constraints ($S = \{c_1, \ldots, c_k\}$) and the logical conjunction of its constitutive constraints ($S = (c_1 \wedge \cdots \wedge c_k)$). By convention: $\emptyset \Leftrightarrow$ `true`.

2.2. Abstracting problem solvers

In the next section, the `decision-repair` algorithm is presented in a framework that abstracts the nature of the problems to be solved: they may be discrete binary CSP,

numeric CSP as well as scheduling problems. This framework has been made possible thanks to:

(1) the parameter Φ, which represents the filtering algorithm used;
(2) the function `obviousInference`, tightly related to the filtering algorithm; it is able to examine a set of constraints in order to decide whether to stop the computation or not;
(3) the concept of decision constraint.

2.2.1. Filtering algorithms

Filtering algorithms are typically used at each node of the search tree developed by systematic algorithms for solving CSP. For example, for binary CSP over finite domains, arc-consistency filtering [33] is often used. After such an arc-consistency 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 singletons: those values assigned to their respective variables provide a feasible solution for the problem under consideration;
(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 Φ applied to a set C of constraints returns a new set $C' = \Phi(C)$ such that $C \subseteq C'$. (Note that we consider domain reduction as additional constraints.) Moreover, for any filtering algorithm Φ applied to the set C of constraints of a given CSP, there exists a function `obviousInference` which interprets the results of the filtering algorithm for the overall search algorithm. When applied to $C' = \Phi(C)$, the function `obviousInference` answers:

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

The function `obviousInference` typically has a low computational cost. Its aim is to make explicit the use of some properties that depend on the filtering algorithm that is used (as done with arc-consistency above when checking empty or singleton domains). A function `obviousInference` can be identified in many other filtering or pruning algorithms. (See Example 1 for linear constraint solving over reals.)

Example 1 (`obviousInference` *for Linear programming*). In integer linear programming, the aim is to find an optimal integer solution. This can be achieved 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 `obviousInference` function would return respectively *noSolution* or *solution*.

2.2.2. Decision constraints

Search for discrete binary CSP consists in trying to assign values to variables, i.e., adding new constraints of the form $v = a$. For other kinds of problems, search may consist in adding other kinds of constraints. We will call those constraints *decision constraints*. A hypothesis holds over the way such constraints are generated: there exists an integer[1] N_e such that whatever the set E of at least N_e different decision constraints, the call `obviousInference(`$\Phi(C \cup E)$`)` will not answer *flounder*. This condition is necessary to ensure termination (for systematic algorithms), and is fulfilled by any reasonable search strategy.

Example 2 (*Decision constraints for numeric* CSP). For numeric CSP, enumeration is performed through adding splitting constraints (e.g., $v \leqslant a$).

Example 3 (*Decision constraints for scheduling problems*). For some scheduling problems, enumeration is performed through adding precedence constraints between tasks of the problem.

3. The `decision-repair` search algorithm

The idea of the `decision-repair` comes from the following observations:

- current local search algorithms mainly work on a total instantiation of the variables;
- backtracking-based search algorithms work, by construction, on a partial instantiation of the variables.

Backtracking-based search algorithms can be combined with filtering techniques because they work on a partial instantiation of the variables. If local search algorithms were working on a partial instantiation of the variables, they could also be combined with filtering techniques. Indeed, the `decision-repair` algorithm is such an algorithm. The partial instantiation under consideration is defined by a set of decision constraints on the variables of the problem. The name `decision-repair` comes from the fact that this algorithm works by repeatedly repairing a set of decisions. In the following, the decision set will be denoted by C_D.

3.1. Principles of `decision-repair`

The `decision-repair` algorithm is presented in Fig. 1.

[1] For discrete CSP where decision constraints are value assignments, N_e is clearly the number of involved variables. For numeric CSP, N_e strongly depends on the desired precision of the result.

```
procedure decision-repair(C)
(1)  C_D ← any initial set of decisions
(2)  repeat
(3)    if conditions of failure satisfied then
(4)      return failure
(5)    else
(6)      C' ← φ(C ∪ C_D)
(7)      switch obviousInference(C')
(8)          case noSolution :
(9)              k ← conflict explaining the failure
(10)             C_D ← neighbor(C_D, k, Γ)
(11)         case solution :
(12)             return C'
(13)         default :
(14)             C_D ← extend(C_D, Γ)
(15)     endswitch
(16)   endif
(17) until false
```

Fig. 1. The `decision-repair` algorithm.

The parameter C of the `decision-repair` algorithm is the set of constraints to be solved. C_D contains the current set of decisions. The algorithm starts by assigning C_D with an initial set of decisions that is determined by some initialization method. (It may range from the empty set to a decision set that defines a complete assignment.) The main loop merely applies the filtering algorithm to $C \cup C_D$ providing a new set of constraints $C' = \Phi(C \cup C_D)$. The `obviousInference` function is then called over C'. Three cases may occur:

- `obviousInference`(C') = *solution*: a solution has been found. The algorithm terminates and returns C'.
- `obviousInference`(C') = *flounder*: the `decision-repair` algorithm tries to extend the current set of decisions C_D by adding a decision constraint. That behavior is similar to that of backtracking-based search algorithms. For that purpose, a function `extend`(C_D, Γ) is assumed to exist that chooses a decision constraint to be added and adds it to C_D. Parameter Γ can be used to store a context that varies according to the chosen version of the algorithm. (Its meaning will be made clear later, but for now let us ignore it.)
- `obviousInference`(C') = *noSolution*: $C \cup C_D$ is inconsistent. We will say that C_D is a *dead-end*, or C_D is *inconsistent*: C_D cannot be extended. The `decision-repair` algorithm will thus try to *repair* the current set of decisions by choosing a new set of decisions through the function `neighbor`(C_D, k, Γ). (Parameter k as Γ will be explained in the following sections.)

This is repeated as long as some *failure conditions*[2] are not satisfied.

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

The `decision-repair` algorithm appears here as a search method that handles partial instantiations and uses filtering techniques to prune the search space. One of the key components of this algorithm is the way the neighborhood is defined.

3.2. *Neighborhood in* `decision-repair`

In a local search algorithm such as GSAT [47] (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 (e.g., selecting the one whose negation will allow the greatest number of clauses to become satisfied). More generally, a local search algorithm uses complete instantiations and replaces an inconsistent one with another complete instantiation chosen among its *neighbors*.

The `decision-repair` algorithm works in the same way except that it uses partial instantiations (sets of decisions): as soon as a decision set becomes inconsistent, one of its neighbors needs to be chosen. A decision set being a partial instantiation, it represents several complete instantiations. Switching the current decision set is like setting aside many irrelevant complete instantiations in one move.

Like any local search algorithm, `decision-repair` may use a heuristic way to select an interesting neighbor. It seems to be a good idea to select a neighboring decision set C'_D which does not have the drawbacks of the current decision set C_D. (Recall that in `decision-repair`, neighbors of decision set C_D are computed only if C_D is inconsistent.) Ideally, we would like to find a consistent neighbor C'_D, i.e., such that `obviousInference`$(\Phi(C \cup C'_D)) = \mathit{solution}$. However, that ideal is equivalent to solving the whole problem.

Instead, we may try to get to a partially consistent neighbor C'_D such that `obviousInference`$(\Phi(C \cup C'_D)) \neq \mathit{noSolution}$. 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, which is not wanted in the `decision-repair` algorithm.

Nevertheless, what can be done is to avoid the neighbors that can already be known as inconsistent. Such information can be extracted from an inconsistent decision set C_D. Indeed, inconsistency means that $C \wedge C_D \Rightarrow \mathit{false}$. It is possible to compute a subset of an inconsistent decision set C_D that is alone inconsistent with C: that is a *conflict*.

Definition 1 (*Conflict*). A *conflict* k for a set of constraints C and a decision set C_D is a subset of C_D, $k \subseteq C_D$ such that $C \wedge k \Rightarrow \mathit{false}$.

Now, we can define a neighbor C'_D of a decision set C_D according to a single conflict k. As long as constraints in the computed conflict k remain altogether in a given decision set C'_D, that decision set will remain inconsistent. Therefore, in order to find a decision set with some hope of being consistent, we need to remove from the current decision set C_D at least one of the decisions in k.

Indeed, a more precise neighborhood can be computed. Let $c \in k$ be a constraint to be removed from C_D. As long as all the constraints in $k \setminus c$ remain in the active decision set,

c will never be satisfiable. Thus, the negation of c can be added to the new decision set. A neighbor of a decision set C_D according to a conflict k is thus defined as follows:

Definition 2 (*Neighbor w.r.t. one conflict*). Let k be a conflict for a decision set C_D, a neighbor of C_D w.r.t. k is a decision set C'_D such that $\exists c \in k, C'_D = C_D \setminus c \cup \{\neg c\}$.

The concept of conflict is crucial for `decision-repair`. It is used to focus the search on judicious (and dynamic) neighborhoods. Conflicts have been widely used in AI in different contexts and under different names, such as nogoods as in [45] or conflict sets as in [41]. See Appendix A for a presentation of a conventional way to compute conflicts.

3.3. `tabu decision-repair`

`Decision-repair` is a generic algorithm for which instances are obtained by specializing several parameters:

- the nature and behavior of Γ the storage structure;
- the neighboring computation function (`neighbor`);
- the extension computation function (`extend`);
- the `failure conditions` that indicate when to halt the search;
- the filtering techniques to be used (the Φ function).

In this section, we present an instance of `decision-repair`, which we called `tabu decision-repair`, that merges a *tabu search* with filtering techniques and conflict-based heuristics.

`Tabu decision-repair` uses a tabu list of a given size s. The s last-computed conflicts are kept in a list Γ. The algorithm maintains as invariant that C_D is compatible with the conflicts in Γ. Formally:

Definition 3 (*Invariant of* `tabu decision-repair`). There does not exist a conflict in Γ that is a subset of C_D.

So far, we have defined a neighbor *w.r.t.* one single conflict. Now, we have to extend the definition when facing multiple stored conflicts.

Definition 4 (*Neighbor w.r.t. several conflicts*). Let Γ be a set of conflicts and let C_D be a decision set. A neighbor C'_D of C_D w.r.t. Γ satisfies $\exists c \in C_D, C'_D = C_D \setminus c \cup \{\neg c\}$ and C'_D is compatible with the conflicts in Γ.

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

Fig. 2 shows an implementation of the *neighbor* function for `tabu decision-repair` that has been used for solving scheduling problems. The *neighbor* function has to record in Γ the new conflict k found by the filtering algorithm and to maintain the invariant.

```
function neighbor(C_D, k, Γ)
        % precondition: k ⊂ C_D, C_D is compatible with Γ
(1)  begin
(2)    add k to the list of conflicts Γ
(3)    if sizeof(Γ) > s then
(4)       remove the oldest element of Γ
(5)    endif
(6)    L ← ordered list (decr. weight) of decisions in k
(7)    repeat
(8)       remove the first decision c from L
(9)       C'_D ← C_D \ {c} ∪ {¬c}
(10)      if C'_D is compatible with all conflicts in Γ then
(11)         return C'_D
(12)      endif
(13)   until L empty
(14)   return stop (or extend the neighborhood)
(15) end
```

Fig. 2. The `neighbor` function for `tabu decision-repair`.

It tries to find one decision in k such that negating this decision makes the decision set compatible with all the conflicts. When several decisions can be negated, we use the following heuristics, which we call weighting-conflict heuristics: a weight is associated with each decision; the weight characterizes the number of times that the decision has appeared in any conflict. A weighting-conflict heuristic that works well takes into account the arity of the conflicts. Each time a conflict is found, the weight of its decision constraints is increased by $1/r$ where r is the arity of the conflict. The *neighbor* function chooses to negate the decision with the greatest weight that, when negated, makes the new decision set compatible with all the conflicts in Γ. If such a decision does not exist, the neighborhood can be extended. For example, we may try to negate two decisions. In our implementation for open-shop problems (see Section 5), 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(`C_D`,`Γ`)` has to use Γ in order to extend the current decision set while maintaining the invariant. For choosing a new decision constraint that extends the current decision set, it is generally worthwhile to use a domain-dependent heuristic. It may be integrated in `decision-repair` as follows: a heuristically ordered list of decision constraints is dynamically generated and the first decision constraint that allows compatibility with all the conflicts in Γ is chosen.

3.4. Example

A run of `tabu decision-repair` where decision constraints are assignments is given below. Assume the maximal size of the tabu list is 2.

Let x_1, x_2, x_3, x_4 be four variables with domain $\{1, 2, 3\}$.

- Suppose that the search has reached a point where one conflict has been encountered and is in the tabu list $\Gamma = \langle\{x_1 = 1, x_3 = 1\}\rangle$.

- Let the current decision set be $\{x_1 = 1, x_2 = 1, x_3 = 2, x_4 = 1\}$, and assume that, for that decision set, the filtering algorithm detects that no solution exists and identify $\{x_1 = 1, x_3 = 2, x_4 = 1\}$ as a conflict.
- The tabu list is increased with the new conflict: $\Gamma = \langle\{x_1 = 1, x_3 = 1\},\ \{x_1 = 1, x_3 = 2, x_4 = 1\}\rangle$. A decision constraint has to be negated to build a relevant neighbor. The respective weights of the decision constraints that appear in Γ are as follows:
  - weight$(x_1 = 1) = 1/2 + 1/3 = 5/6$.
  - weight$(x_3 = 1) = 1/2$.
  - weight$(x_3 = 2) = 1/3$.
  - weight$(x_4 = 1) = 1/3$.
- The decision contraint with the maximal weight is $x_1 = 1$. Its negation in the current decision set allows compatibility with all the conflicts in Γ, so the current decision set becomes $\{x_1 \neq 1, x_2 = 1, x_3 = 2, x_4 = 1\}$.
- Assume the filtering algorithm cannot deduce that there is no solution nor that a solution is found in the current decision set. The current decision set is then extended, and let $x_1 = 2$ be the added decision constraint.
- A new conflict detected by the filtering algorithm is $\{x_1 = 2, x_4 = 1\}$. It is added in Γ, and the older conflict is removed from Γ since the size of the tabu list is bounded by two.
  The respective weights of the decision constraints become:
  - weight$(x_1 = 1) = 5/6$.
  - weight$(x_1 = 2) = 1/2$.
  - weight$(x_3 = 2) = 1/3$.
  - weight$(x_4 = 1) = 1/3 + 1/2 = 5/6$.
- The decision constraint chosen to be negated by the *neighbor* function is $x_4 = 1$. The current decision set becomes $\{x_1 \neq 1, x_1 = 2, x_2 = 1, x_3 = 2, x_4 \neq 1\}$.
- After another extension of the decision set by adding the decision $x_4 = 2$, the filtering algorithm detects a solution.

4. Related works

The `decision-repair` algorithm takes its roots in many other works, among which [18] has probably been the most influential by highlighting the relationships between local search and systematic search, and by the use of conflicts to guide the search and make it systematic. In the same spirit are [7,16,19,45]. `Decision-repair` is a generic algorithm. It generalizes not only some non-systematic algorithms but also some systematic ones. For example, *Dynamic Backtracking* [19] can be obtained by specifying the `decision-repair` parameters as follows: `neighbor` keeps only conflicts that correspond to relevant *eliminating explanations* (see Appendix A), applies resolution over eliminating explanations, and undoes the most recent decision in the computed conflict.

The algorithm proposed in [44] can also be seen as an instance of the `decision-repair` algorithm where the decision constraints are instantiations; there is no propagation and no pruning; (the filtering algorithm Φ only consists in checking whether the constraints containing only instantiated variables are not violated); and it does not make use of con-

flicts, neither in the *neighbor* function nor in the *extend* function. The common idea, which already exists in previous works [24], 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 of using a filtering algorithm during the running of a local search has been also used in [49] and in [48]. In [49], 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 `decision-repair`, the filtering is applied over the current domains at every step. In [48], a technique called *Large Neighborhood Search* (LNS) is proposed for solving optimization problems. Unlike `decision-repair`, LNS performs a local search on *total* assignments of variables. And it uses a filtering algorithm for making a move: it unassigns several variables and uses a tree-based search with constraint propagation to find the next total assignment.

Introducing a stochastic seed in a backtracking algorithm [30,37] can also be seen as some kind of hybrid between local search and systematic search. The neighborhood is no longer defined by conflicts but is stochastically generated. An algorithm like the tabu version of `decision-repair` is a trade-off between keeping all the conflicts encountered during search (which potentially needs exponential space) and replacing the useful information of the conflicts by random actions.

The heuristic we used to select the decision constraint to negate—choose the one that has appeared the greatest number of times in small conflicts—is in some sense a generalization of a similar approach for GSAT counting the number of times that a constraint has not been satisfied [46].

The way conflicts 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 + backjumping [40], forward checking + dynamic backtracking [52], arc-consistency + backjumping [11,42], arc-consistency + dynamic backtracking [28], $2B$-consistency + dynamic backtracking [27]). Nevertheless, as far as we know, the `tabu` version of `decision-repair` is the first time such a technique has been used in combination with a local search algorithm.

5. A case study: Solving scheduling problems

5.1. Open-shop scheduling problems

Classical scheduling shop problems for which a set J of n jobs consisting each of m tasks (operations) must be scheduled on a set M of m machines can be considered as CSP.[3] One of those problems is called the open-shop problem [21]. For that problem, operations

[3] The variables of the CSP 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.

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.[4]

The open-shop scheduling problem is NP-hard as soon as $\min(n, m) \geqslant 3$. This problem although quite simple to enunciate is really hard to solve optimally: instances of size 6×6 (i.e., 36) variables remain unsolved!

5.2. `tabu decision-repair` *for open-shop problems*

The `tabu` version of `decision-repair` has been tried on the open-shop problem. The implementation is described below.

- *Filtering techniques*. Precedence constraints are handled with $2B$-consistency filtering [12,32] and resource usage constraints are handled through *task intervals* [10]. The `decision-repair` algorithm needs to call the filtering algorithm many times with little change of the constraint set. Instead of recomputing the arc-consistency closure from scratch, it is possible to maintain it incrementally. That problem has been addressed for dynamic CSP. The algorithms used in `decision-repair` are similar to those of [5,14] or other works [27].
- *Computation of conflicts*. Conflicts are computed as stated in the annex. More details about the computation of the conflicts can be found in [26].
- Search strategy. For shop problems, enumeration is usually performed on the relative order in which tasks are scheduled on the resources. The decision constraints are thus precedence constraints between tasks. (When every possible precedence has been posted, setting the starting date of the variable to their smallest value provides a feasible solution.)
- *Tabu list*. The implementation uses a tabu list of size 7.
- *Neighborhood*. The `neighbor` function is the one given in Fig. 2.
- *Stopping criterion*. The failure conditions specifying the exit of `decision-repair` (Fig. 1) are either a *stop* returned by the `neighbor` function or 3000 iterations without improvement since the last solution reached.
- *Minimisation of the makespan*. The open-shop problems we consider are optimisation problems. This requires a main loop that calls the function `decision-repair` until improvement is no longer possible. (See Fig. 3.) Improvements are generated by adding a constraint that specifies that the makespan is less than the current best solution found. The initial decision set for each call of the function `decision-repair` is the latest set of decisions (which defines the last solution found).

5.3. *First results*

We present here results obtained by `tabu decision-repair` on three series of reference problems:

[4] Ending time of the last task.

```
procedure minimise-makespan(C)
(1)  begin
(2)     C_D ← initial decision set
(3)     bound ← +∞
(4)     lastSolution ← failure
(5)     repeat
(6)          C ← C ∪ { makespan < bound}
(7)          solution ← decision-repair(C)
(8)          if solution = failure then
(9)               return lastSolution
(10)         else
(11)              bound ← value of makespan in solution
(12)              lastSolution ← solution
(13)         endif
(14)    until false
(15) end
```

Fig. 3. Algorithm used to solve open-shop problems.

(1) *Taillard*'s problems [50]: 40 problems are solved consisting of 4 series of 10 problems of size 4×4, 5×5, 7×7, and 10×10.
(2) *Brucker et al.* problems [8]: 52 problems of size 3×3 to 8×8. Those problems are characterized by a common *LB* (the classical lower bound[5]) value: 1000.
(3) *Guéret and Prins*' problems: Those 80 problems (8 series of 10 problems of size 3×3 to 10×10) have been generated using results presented in [22] for generating really hard open-shop instances. They all share a common *LB* (classical lower bound) value and the fact that another lower bound [22] gives a much greater value. Those problems are downloadable at http://www.emn.fr/gueret/OpenShop/OpenShop.html.

The `tabu decision-repair` algorithm (referred to as TDR in the results) is compared with the best published solving techniques for the open-shop problem:

- For Taillard's instances, our results are compared with two highly specialized tabu searches tailored for solving open-shop problems, one presented in [1] (referred to as TS-A97 in the results) and one presented in [31] (referred to as TS-L98 in the results).
- For all the instances, our results are also compared with a genetic algorithm introduced in [38] (referred to as GA-P99 in the results) which gives very good results on all those problems.

Fig. 1 presents results obtained on Taillard's problems, Fig. 2 on Brucker's instances, and finally Fig. 3 on Guéret and Prins' problems. CPU time is not available in [1,31], nor in [38].[6] *Average* CPU time for `tabu decision-repair` is not really significant

[5] Maximum load of the involved machines and jobs.

[6] This is quite usual for open-shop scheduling results. Indeed, the problem itself being really hard, what is important is the quality of the solution and not the time required to obtain it. Moreover, in real-life applications such as satellite scheduling problems [39] improving a solution by one can save so much money that satellite operators are ready to wait as long as a full day for that improvement!

Table 1
Results on Taillard's instances. Results are presented in the following format: "average deviation from the optimal value"/"maximum deviation from the optimal value" ("number of optimally solved instances")

Series	TS-L98	TS-A97	GA-P99	TDR
4×4	0/0 (10)	(*)	0.31/1.84 (8)	0/0 (10)
5×5	0.09/0.93 (9)	(*)	1.26/3.72 (1)	0/0 (10)
7×7	0.56/1.77 (6)	0.75/1.71 (2)	0.41/0.95 (4)	0.44/1.92 (6)
10×10	0.29/1.41 (6)	0.73/1.67 (1)	0/0 (10)	2.02/3.19 (0)

(*) Only results for 7 problems of size 7×7 and 3 of size 10×10 are given in the paper.

Table 2
Results on Brucker's instances. Results are presented according to the following format: "average deviation from the optimal value"/"maximum deviation from the optimal value" ("number of optimally solved instances") except for 7×7 and 8×8 problems for which the deviation is computed from the *LB* value (1000 for each problem)

Series	GA-P99	TDR
3×3 (8 pbs)	0/0 (8)	0/0 (8)
4×4 (9 pbs)	0/0 (9)	0/0 (9)
5×5 (9 pbs)	0.36/2.07 (6)	0/0 (9)
6×6 (9 pbs)	0.92/2.27 (3)	0.71/3.50 (6)
7×7 (9 pbs)	3.82/8.20 (6)	4.40/11.5 (5)
8×8 (8 pbs)	3.10/7.50 (8)	4.95/11.8 (2)

since CPU time strongly depends on the instance of the problem. Just to give an idea, for Taillard's instances, 10×10 average CPU time is 15 hours and for 7×7 average CPU time is 2 hours. For Brucker's instances and Guéret and Prins' problems, 10×10 average CPU time is 3 to 4 hours and, for size less than 8×8, average CPU time less than 4 minutes.

First recall that `decision-repair` is a generic algorithm which has been instantiated simply to solve a very specific problem which has its own research community. The results obtained on the three sets of problems are therefore very interesting because they show that `tabu decision-repair` is a competitive algorithm compared with the other techniques.

As far as Taillard's instances are concerned, `tabu decision-repair` gives comparable results but the *tabu search* of [31] is still the best technique except for 10×10 problems where the genetic algorithm shows the best results.

On Brucker's instances, `tabu decision-repair` is far better than the genetic algorithm on small instances but the latter becomes better on larger problems.

For the third set of problems (the really hard instances of Guéret and Prins) `tabu decision-repair` shows all the interest of combining local search and constraint propagation: `tabu decision-repair` closed[7] 6 of these instances. Furthermore, it

[7] An optimal solution was found and proved—a lower bound is known—for the first time.

Table 3
Results on Guéret and Prins' problems. Results are presented according to the following format: "number of problems solved giving the best results"/"number of optimally solved problems". BB-G00 reports the results of a systematic search introduced in [23] and stopped after 350 000 backtracks (which represents around 24 hours of CPU time). What `tabu decision-repair` gave to the solving of those problem (TDR yield) is indicated by "the number of closed instances"/"the number of newly improved instances"

Series	BB-G00	GA-P99	TDR	Open instances	TDR yield
3×3	10/10	10/10	10/10	0	–
4×4	10/10	10/10	10/10	0	–
5×5	10/10	8/8	10/10	0	–
6×6	9/7	2/1	10/8	3	1/1
7×7	3/1	6/3	10/4	9	1/3
8×8	2/1	2/1	10/4	9	3/7
9×9	1/1	0/0	10/2	9	1/9
10×10	0/0	5/0	5/0	10	0/5

provided new best results for 19 other instances; thus it improved known results for 25 instances out of 40 open ones.

Up to size 9×9, `tabu decision-repair` gives far better results than both the genetic algorithm and branch and bound search (that has been truncated by a time criterion). For 10×10 problems, `tabu decision-repair` is still better than the branch and bound but is matched by the genetic algorithm.

Such good behavior of `tabu decision-repair` was quite surprising because, unlike the other specialized algorithms, our implementation remains general and does not need any tuning of complex parameters. This is probably due to the search used for the open-shop problem, which dynamically builds independent sub-problems by adding precedence constraints: classical backtracking algorithms may start by partially solving a sub-problem, then go to another one, solve it, and then continue to solve the first sub-problem. In cases where it has to backtrack to choices in the first part of its work, the search space of the two sub-problems are multiplied. Our `decision-repair`, thanks to its use of conflicts, can identify independent sub-problems and stay in a sub-problem until it has been solved. Also the heuristic we have introduced seems to be good. Once again, this is another benefit from the use of conflicts.

5.4. In-depth analysis

In this section, we proceed to a more in-depth analysis of the behavior of `decision-repair` in order to better understand its excellent performance.

A series of experiments were made in order to study:

- how `decision-repair` performance is affected by a modification of the structural parameters of the algorithm;
- if the use of precise conflicts is important for the search performed;
- if the use of repair techniques is really crucial compared to other conflict-based techniques.

Notice that we consider constraint propagation as mandatory when dealing with combinatorial problems such as open-shop scheduling and will therefore not compare `decision-repair` with enumeration-only techniques.

5.4.1. Varying parameters

The `tabu decision-repair` algorithm has three main parameters:

- the maximum number of allowed movements without improvement;
- the length of the tabu list;
- the selection of the candidate neighbor for repair (handled through the definition of the weight of each constraint in the conflict).

5.4.1.1. Protocol of the experiments. Experiments were done on the *Guéret and Prins*' problems [22]. We tested `decision-repair` with different sets of values for the parameters. All possible combinations were used:

- the maximum number of authorized moves with no improvements: {1500, 3000}.
- the length of the tabu list: $\{2, 7, 12, 17\}$.
- the definition of the weight of a constraint to be used when repairing an infeasible partial solution (Notice that constraints with higher weight are most likely to be selected for repair.):
  - *nb-conflicts* (*nc*) The weight associated with each constraint is its number of occurrences in all the conflicts encountered so far.
  - *weighted-nb-conflicts* (*wnc*) The weight associated with each constraint is the number of occurrences in all the conflicts encountered so far taking into account the size of each conflict. Let C be the set of conflicts in which a constraint c appears. The weight $w(c)$ of constraint c is defined as:

$$w(c) = \sum_{\sigma \in C} \text{size}(\sigma)^{-1}.$$

  - *most-recent* (*mr*) The weight associated with each constraint is the time-stamp of its activation.
  - *most-ancient* (*ma*) The weight associated with each constraint is the inverse time-stamp of its activation.

5.4.1.2. Varying the maximum number of moves. Tables 4 and 5 present the results obtained when varying the maximum number of consecutive moves allowed without improvement. Results with other techniques are also reported on those tables.

As we can see, allowing more possible moves leads to better results. Doubling the number of allowed moves leads to an increase in total moves by only 37%.

What is important to notice here is that `decision-repair` does not seem really to rely on that parameter since considering the two possible values given, results remain far better than the other techniques presented.

Table 4
`decision-repair`: varying the maximum number of moves—Final makespan

Problem	1500 moves	3000 moves	BB-G00	GA-P99
9x9-1	1131	1129	1150	1149
9x9-2	1111	1111	1226	1126
9x9-3	1115	1115	1150	1118
9x9-4	1130	1130	1181	1151
9x9-5	1180	1180	1180	1181
9x9-6	1096	1094	1136	1122
9x9-7	1091	1091	1173	1124
9x9-8	1106	1105	1193	1111
9x9-9	1123	1123	1218	1138
9x9-10	1118	1111	1166	1131

Table 5
`decision-repair`: varying the maximum number of moves—Total number of moves

Problem	1500 moves	3000 moves
9x9-1	5717	10701
9x9-2	7136	8928
9x9-3	5632	7233
9x9-4	5013	5013
9x9-5	1686	1686
9x9-6	10941	15648
9x9-7	8571	10301
9x9-8	15348	19033
9x9-9	8110	9748
9x9-10	8089	16212

5.4.1.3. Length of the tabu list. Tables 6 and 7 present the results obtained when varying the length of the tabu list. Results with other techniques are also reported in those tables.

As we noticed for the maximum number of consecutive moves without improvement allowed, whatever the value given for the size of the tabu list, `tabu decision-repair` remains a better technique than other efficient ones.

However, focusing the comparison on `tabu decision-repair` leads to the observation that a tabu list of size 2 or 12 seems to be the best value and that 17 is more often the worst one (especially considering the total number of moves—Table 7).

5.4.1.4. Constraint weights. Tables 8 and 9 present the results obtained when varying the weight associated with each constraint appearing in the conflicts. (See Section 5.4.1.1.) Results with other techniques are also reported in those tables.

The `most-recent` technique is clearly out performed by all the other techniques. The `most-ancient` technique is worse than `nb-conflicts` and `weighted-nb-conflicts`. Those last two are the best methods (including the genetic algorithm of GA-P99 and the branch and bound of BB-G00). Furthermore, `weighted-nb-conflicts` is a little better than the others.

Table 6
`decision-repair`: varying the length of the tabu list—Final makespan

Problem	2	7	12	17	BB-G00	GA-P99
9x9-1	1129	1129	1129	1129	1150	1149
9x9-2	1111	1111	1110	1110	1226	1126
9x9-3	1115	1115	1115	1115	1150	1118
9x9-4	1130	1130	1130	1130	1181	1151
9x9-5	1180	1180	1180	1180	1180	1181
9x9-6	1094	1094	1093	1094	1136	1122
9x9-7	1094	1091	1097	1092	1173	1124
9x9-8	1108	1105	1112	1105	1193	1111
9x9-9	1123	1123	1123	1123	1218	1138
9x9-10	1110	1111	1119	1113	1166	1131

Table 7
`decision-repair`: varying the maximum number of moves—Total number of moves

Problem	2	7	12	17
9x9-1	13531	10701	12655	11578
9x9-2	10046	8928	10822	8921
9x9-3	5183	7233	6290	8343
9x9-4	6665	5013	8651	9330
9x9-5	1769	1686	894	795
9x9-6	11572	15648	15180	13893
9x9-7	18618	10301	12756	25117
9x9-8	11364	19033	10322	22695
9x9-9	7852	9748	11322	10679
9x9-10	15384	16212	13267	12372

Table 8
`decision-repair`: varying the heuristic selection—Final makespan

Problem	`nc`	`wnc`	`mr`	`ma`	BB-G00	GA-P99
9x9-1	1129	1129	1182	1129	1150	1149
9x9-2	1112	1111	1164	1138	1226	1126
9x9-3	1116	1115	1192	1115	1150	1118
9x9-4	1131	1130	1188	1170	1181	1151
9x9-5	1180	1180	1191	1180	1180	1181
9x9-6	1093	1094	1177	1132	1136	1122
9x9-7	1092	1094	1139	1094	1173	1124
9x9-8	1110	1108	1180	1105	1193	1111
9x9-9	1123	1123	1184	1176	1218	1138
9x9-10	1116	1110	1146	1111	1166	1131

5.4.1.5. Conclusion. This comprehensive study of the parameters of `tabu decision-repair` shows that the length of the tabu list and the maximum number of allowed consecutive moves without improvement are not key parameters for our technique.

Table 9
`decision-repair`: varying the heuristic selection—Total number of moves

Problem	nc	wnc	mr	ma
9x9-1	13945	13531	3555	5479
9x9-2	10614	10046	2995	893
9x9-3	6983	5183	3465	3004
9x9-4	13301	6665	3791	2030
9x9-5	1240	1769	3438	1040
9x9-6	12158	11573	3894	2152
9x9-7	16779	18618	3512	17607
9x9-8	13552	11364	6895	12847
9x9-9	12835	7852	4178	2047
9x9-10	13185	15384	6154	11637

However, the weighting technique seems quite important and our results show that the techniques that make use as much as possible of the information stored in past conflicts give better results.

5.4.2. Impact of using conflicts

As recalled in Appendix A, there exist several ways of computing a single conflict. One question that may arise about `decision-repair` is: is it worthwhile designing precise explanation-based algorithms rather that providing very general explanations when propagating constraints? We compared results both on Taillard's instances (Table 10) and $9x9$ Guéret and Prins' instances (Table 11) with two sets of algorithms:

- a propagation algorithm for scheduling problems that is based upon Carlier and Pinson's *immediate selections* (TDR-IS) [9] that gives only general explanations;
- a propagation algorithm for scheduling problems that is based upon the *task intervals* (TDR-TI) of Caseau and Laburthe [10]. *Task intervals* give very precise explanations for value removals.

The results (Table 8) clearly show that using precise conflicts leads to better results. Moreover, when computing conflicts using *task intervals* the overall search requires less CPU time than the search obtained when computing conflicts with *immediate selections*. Precise conflicts are priceless for algorithms such as `decision-repair` !

5.4.3. Impact of using repair

Another question arises about `tabu decision-repair` results: is the repair component of the algorithm is a key component? In order to answer that question, we are comparing here three algorithms:

- `tabu decision-repair` as described in this paper;
- `mac-dbt` [28] which combines arc-consistency with dynamic backtracking and which is indeed a special case of `decision repair` that does not really performs a local search because of the completeness requirements;

Table 10
`decision-repair`: varying the explanation system—Final makespan—Taillard's instances

Problem	TDR-IS	TDR-TI	TS-L98	TS-A97	GA-P99
5x5-1	301	300	300	–	301
5x5-2	262	262	262	–	263
5x5-3	323	323	326	–	335
5x5-4	311	310	310	–	316
5x5-5	326	326	326	–	330
5x5-6	314	312	312	–	312
5x5-7	304	303	303	–	308
5x5-8	300	300	300	–	304
5x5-9	356	353	353	–	358
5x5-10	326	326	326	–	328
7x7-1	435	435	435	437	436
7x7-2	439	447	447	444	447
7x7-3	473	477	474	476	472
7x7-4	463	463	463	464	463
7x7-5	416	416	417	417	417
7x7-6	460	455	459	–	455
7x7-7	430	425	429	429	426
7x7-8	424	424	424	–	424
7x7-9	458	458	458	458	458
7x7-10	398	398	398	398	398

Table 11
`decision-repair`: varying the explanation system—Final makespan—Guéret and Prins' instances

Problem	TDR-IS	TDR-TI	BB-G00	GA-P99
9x9-1	1129	1129	1150	1149
9x9-2	1111	1111	1226	1126
9x9-3	1115	1115	1150	1118
9x9-4	1188	1130	1181	1151
9x9-5	1180	1180	1180	1181
9x9-6	1093	1094	1136	1122
9x9-7	1102	1091	1173	1124
9x9-8	1109	1105	1193	1111
9x9-9	1212	1123	1218	1138
9x9-10	1119	1111	1166	1131

- the branch and bound algorithm of [23], which is actually an intelligent backtracker that makes a limited use of computed conflicts.

Results presented in Table 12 show that `tabu decision-repair` out-performs `mac-dbt` (which has been cut after 3000 consecutive moves without improvements). This tends to show that the local search part of `decision-repair` is really important. Moreover, `tabu decision-repair` is much quicker than `mac-dbt`.

Table 12
`decision-repair`: varying the repair technique—Final makespan

Problem	TDR	MAC-DBT	BB-G00
9x9-1	1129	1182	1129
9x9-2	1111	1164	1226
9x9-3	1115	1192	1150
9x9-4	1130	1188	1181
9x9-5	1180	1191	1180
9x9-6	1094	1177	1136
9x9-7	1091	1139	1173
9x9-8	1105	1180	1193
9x9-9	1123	1184	1218
9x9-10	1111	1146	1166

Our results also show that `tabu decision-repair` out-performs the intelligent backtracker BB-G00 (which has been cut after 350 000 backtracks). This tends to show the repairing part of `decision-repair` is a key component of the algorithm.

Moreover, whereas BB-G00 needs 12 hours on average to solve one problem, `tabu decision-repair` needs only 4–6 hours and gives far better results.

Notice that `mac-dbt` competes well with `BB-G00`, showing once more that the interest of `decision-repair` relies on the combination of efficient propagation, free moves, and a conflict-directed heuristic.

6. Conclusion and future work

In this paper, we introduced a new algorithm for solving CSP, the `decision-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 concept of *conflict* has been crucial to implement that algorithm: conflicts allow relevant neighborhoods to be considered, and conflicts can be used to derive efficient neighbor-selecting heuristics for a `decision-repair` algorithm.

Experiments with a `tabu` version of `decision-repair` have shown good results over open-shop scheduling problems. Several well known hard instances of the open-shop scheduling problems have been solved for the first time thanks to `decision-repair`, and the best bounds of some other instances have been improved. So, at least, `decision-repair` competes well with the best highly specialized algorithms. This result was quite surprising since, unlike those specialized algorithms, our implementation is general and does not need any tuning of complex parameters.

A comprehensive study of the behavior of `decision-repair` has shown that the key components of this algorithm are: its conflict-directed heuristics and its ability both to perform a local search and to prune the search space.

In future work, we will investigate the following issues:

- In our implementation for open-shop scheduling problems, the conflicts we compute are not minimal. A method like the one presented in [25] is able to compute minimal conflict in reasonable time. As more precise conflicts may greatly improve the efficiency of a `decision-repair` implementation, such a conflict-detection method deserves experimentation.
- Also experiments of `decision-repair` over other fields than scheduling problems have to be performed.

Acknowledgements

We would like to thank Christian Bliek, Ulrich Junker and Paul Shaw, for their constructive comments on an early draft of the paper. We also thank the anonymous reviewers for their useful suggestions that greatly improve the quality of the paper. Finally, we thank Kathleen Callaway who helped us a lot to repair English mistakes.

Appendix A. Conflicts for constraint programming

A **conflict** (*a.k.a.* **nogood** [45]) is a subset (here $dc_1, \ldots, dc_k$) of the current decision constraints (the set C_D) that, taken along with the original constraints (the set C), leads to a contradiction (no feasible solution contains a conflict) (Eq. (A.1))

$$\neg(C \wedge dc_1 \wedge \cdots \wedge dc_k). \tag{A.1}$$

Notice that, from that definition, if the current decision set C_D is inconsistent, C_D is a valid conflict. Obviously, a strict subset will be much more interesting. A minimal (w.r.t. the inclusion) conflict would be the best, but could be expensive to compute [25,51]. Our current implementation does not try to find such a minimal conflict. Instead, it tries to compute a *good* conflict quickly.

A conflict must be computed each time the filtering algorithm can prove that no solution exists with the current decision set. This happens when the domain of a variable v becomes empty. Assume that each value a_i of the domain of v has been removed. Assume also that, for each value a_i, a subset of C_D, denoted by $\texttt{expl}(v \neq a_i)$, is given. $\texttt{expl}(v \neq a_i)$ is called an *eliminating explanation* (explanation for short) and justifies the removal of value a_i from the domain $d(v)$ of variable v. Formally, $\texttt{expl}(v \neq a_i)$ is such that: $C \wedge \texttt{expl}(v \neq a_i) \Rightarrow v \neq a_i$. Thus, $k = \bigcup_i \texttt{expl}(v \neq a_i)$ is a conflict since no value for v is allowed by the union of the eliminating explanations.[8]

Thus, computing explanations is sufficient to compute conflicts within constraint programming. Filtering operations in CSP can be considered as a sequence of value removals which can all be explained by an eliminating explanation. The simplest of all explanations is to merely consider the complete set of decisions. Much more useful explanations can be provided by instrumenting the propagation algorithms. Indeed, value removals are direct consequences of the filtering algorithms. For some algorithms, this

[8] Note that this inference is called hyper-resolution in [29].

instrumentation is quite simple. For AC-4 [35], value removal explanations can be computed by a simple trace mechanism embedded within the filtering algorithm and by storage of the reason for a removal (e.g., see [28]). For numeric CSP and $2B$-consistency (arc-consistency restricted to the bounds of the domains [12,32]) a better way to compute conflicts is not to keep why values have been removed but to maintain an explanation for the current values of the two bounds of a domain (e.g., see [27]). When a domain becomes empty, the lower bound becomes greater than the upper bound, and the union of the explanations of the two current bounds is a conflict. (Recall that decisions are typically made by adding *splitting* constraints in numeric CSP. See Example 2.)

For other filtering algorithms, like AC-6 [6], computing the eliminating explanations cannot be done without changing the asymptotic behavior of the algorithm. In such a case, a practical implementation is to consider an explanation for the current domain of a variable (e.g., see [42]), but this implementation is less precise than considering only the supports of the values.

Example A.1 (*Computing eliminating explanations*). For example, let us consider two variables v_1 and v_2 whose domains are both $\{1, 2, 3\}$.

- Let c_1 be a first decision constraint: c_1: $v_1 \geqslant 3$. Let us assume that the filtering algorithm in use is $2B$-consistency filtering. The constraint c_1 leads to the removal of $\{1, 2\}$ from the domain of v_1. An explanation for the new domain $\{3\}$ of v_1 is thus $\{c_1\}$.
- Let c_2 be a second constraint: c_2: $v_2 \geqslant v_1$. Value 1 and value 2 of v_2 have no support in the domain of v_1, and thus c_2 leads to the removal of $\{1, 2\}$ from v_2. An explanation of the removal of $\{1, 2\}$ from v_2 will be: $c_1 \wedge c_2$ because c_2 precipitates that removal only because previous removals occurred in v_1 due to c_1.

Several explanations generally exist for the removal of a given value. Recording all of them leads to an exponential space complexity. Another technique relies on *forgetting* (erasing) explanations that are no longer relevant[9] to the current variable assignment. By doing so, the space complexity remains polynomial. Here, we retain *one* explanation at a time for a value removal.

References

[1] D. Alcaide, J. Sicilia, D. Vigo, A tabu search algorithm for the open shop problem, TOP (Trabajos de Investigación Operativa) 5 (2) (1997) 283–296.
[2] D. Applegate, W. Cook, A computational study of the job-shop scheduling problem, ORSA J. Comput. 3 (2) (1991) 149–156.
[3] R.J. Bayardo Jr., D.P. Miranker, A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem, in: Proc. AAAI-96, Portland, OR, 1996, pp. 298–304.

[9] An explanation is said to be *relevant* if all the decision constraints in it are still valid in the current search state [3].

[4] R. Bayardo Jr., R. Schrag, Using CSP look-back techniques to solve exceptionally hard SAT instances, in: Proc. CP-96, 1996, pp. 46–60.
[5] C. Bessière, Arc consistency in dynamic constraint satisfaction problems, in: Proc. AAAI-91, Anaheim, CA, 1991.
[6] C. Bessière, M. Cordier, Arc-consistency and arc-consistency again, in: Proc. AAAI-93, Washington, DC, 1993, pp. 108–113.
[7] C. Bliek, Generalizing partial order and dynamic backtracking, in: Proc. AAAI-98, Madison, WI, 1998, pp. 319–325.
[8] P. Brucker, B. Jurish, B. Sievers, B. Wöstmann, A branch and bound algorithm for the open-shop problem, Discrete Appl. Math. 76 (1997) 43–49.
[9] J. Carlier, E. Pinson, Adjustment of heads and tails for the job-shop problem, European J. Oper. Res. 78 (1994) 146–161.
[10] Y. Caseau, F. Laburthe, Improving CLP scheduling with task intervals, in: P. Van Hentenryck (Ed.), Proc. 11th International Conference on Logic Programming, ICLP-94, MIT Press, Cambridge, MA, 1994, pp. 369–383.
[11] P. Codognet, F. Fages, T. Sola, A metalevel compiler of CLP(FD) and its combination with intelligent backtracking, in: F. Benhamou, A. Colmerauer (Eds.), Constraint Logic Programming—Selected Research, MIT Press, Cambridge, MA, 1993, Chapter 23, pp. 437–456.
[12] E. Davis, Constraint propagation with interval labels, Artificial Intelligence 32 (2) (1987) 281–331.
[13] P. David, A constraint-based approach for examination timetabling using local repair techniques, in: Proc. Second International Conference on the Practice and Theory of Automated Timetabling (PATAT-97), Toronto, ON, 1997, pp. 132–145.
[14] R. Debruyne, Arc-consistency in dynamic CSPs is no more prohibitive, in: Proc. 8th Conference on Tools with Artificial Intelligence (TAI-96), Toulouse, France, 1996, pp. 299–306.
[15] R. Dechter, Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition, Artificial Intelligence 41 (3) (1990) 273–312.
[16] D. Frost, R. Dechter, Dead-end driven learning, in: Proc. AAAI-94, Seattle WA, 1994, pp. 294–300.
[17] C. Gervet, Large combinatorial optimization problem methodology for hybrid models and solutions, in: JFPLC, 1998, Invited talk.
[18] M. Ginsberg, D.A. McAllester, GSAT and dynamic backtracking, in: Proc. International Conference on the Principles of Knowledge Representation (KR-94), 1994, pp. 226–237.
[19] M. Ginsberg, Dynamic backtracking, J. Artificial Intelligence Res. 1 (1993) 25–46.
[20] F. Glover, M. Laguna, Tabu search, in: C. Reeves (Ed.), Modern Heuristic Techniques for Combinatorial Problems, Blackwell Scientific Publishing, Oxford, 1993, pp. 70–141.
[21] T. Gonzales, S. Sahni, Open-shop scheduling to minimize finish time, J. ACM 23 (4) (1976) 665–679.
[22] C. Guéret, C. Prins, A new lower bound for the open-shop problem, AOR (Annals of Operations Research) 92 (1999) 165–183.
[23] C. Guéret, N. Jussien, C. Prins, Using intelligent backtracking to improve branch and bound methods: An application to open-shop problems, European J. Oper. Res. 127 (2) (2000) 344–354.
[24] P. Jackson, Introduction to Expert Systems, Addison Wesley, Reading, MA, 1990.
[25] U. Junker, QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms, Technical Report, 2001.
[26] N. Jussien, e-constraints: Explanation-based constraint programming, in: Proc. CP01 Workshop on User-Interaction in Constraint Satisfaction, Paphos, Cyprus, 2001.
[27] N. Jussien, O. Lhomme, Dynamic domain splitting for numeric CSP, in: Proc. European Conference on Artificial Intelligence, Brighton, 1998, pp. 224–228.
[28] N. Jussien, R. Debruyne, P. Boizumault, Maintaining arc-consistency within dynamic backtracking, in: Proc. Principles and Practice of Constraint Programming (CP-2000), Lecture Notes in Computer Science, Vol. 1894, Springer, Berlin, 2000, pp. 249–261.
[29] J. de Kleer, A comparison of ATMS and CSP techniques, in: Proc. IJCAI-89, Detroit, MI, 1989, pp. 290–296.
[30] P. Langley, Systematic and non-systematic search strategies, in: Proc. AAAI-92, San Jose, CA, 1992.
[31] C.-F. Liaw, A tabu search algorithm for the open shop scheduling problem, Computers and Operations Research 26.

[32] O. Lhomme, Consistency techniques for numeric CSPs, in: Proc. IJCAI-93, Chambéry, France, 1993, pp. 232–238.
[33] A. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1) (1977) 99–118.
[34] S. Minton, M. Johnston, P. Laird, Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems, Artificial Intelligence 58 (1992) 161–206.
[35] R. Mohr, T.C. Henderson, Arc and path consistency revisited, Artificial Intelligence 28 (1986) 225–233.
[36] G. Pesant, M. Gendreau, A view of local search in constraint programming, in: Proc. Principles and Practice of Constraint Programming, Springer, Berlin, 1996, pp. 353–366.
[37] S. Prestwich, A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences, in: Proc. Principles and Practice of Constraint Programming (CP-2000), Lecture Notes in Computer Science, Vol. 1894, Springer, Berlin, 2000, pp. 337–352.
[38] C. Prins, Competitive genetic algorithms for the open shop scheduling problem, Research Report, École des Mines de Nantes, 99/1/AUTO, 1999.
[39] C. Prins, J. Carlier, Resource optimization in a TDMA/DSI system: The EUTELSAT approach, in: Proc. Internat. Conference on Digital Satellite Communications (ICDSC 7), Munich, Germany, 1986, pp. 511–518.
[40] P. Prosser, Hybrid algorithms for the constraint satisfaction problem, Comput. Intelligence 9 (3) (1993) 268–299. Also available as Technical Report AISL-46-91, Stratchclyde, 1991.
[41] P. Prosser, Domain filtering can degrade intelligent backtracking, in: Proc. IJCAI-93, Chambéry, France, 1993, pp. 262–267.
[42] P. Prosser, MAC-CBJ: Maintaining arc-consistency with conflict-directed backjumping, Research Report 95/177, Department of Computer Science, University of Strathclyde, 1995.
[43] E.T. Richards, B. Richards, Non-systematic search and learning: An empirical study, in: Proc. Conference on Principles and Practice of Constraint Programming, Pisa, 1998, pp. 370–384.
[44] A. Schaerf, Combining local search and look-ahead for scheduling and constraint satisfaction problems, in: Proc. IJCAI-97, Nagoya, Japan, Morgan Kaufmann, San Mateo, CA, 1997, pp. 1254–1259.
[45] T. Schiex, G. Verfaillie, Nogood recording for static and dynamic constraint satisfaction problems, Internat. J. Artificial Intelligence Tools 3 (2) (1994) 187–207.
[46] B. Selman, H. Kautz, Domain-independent extensions to GSAT: Solving large structured satisfiability problems, in: R. Bajcsy (Ed.), Proc. IJCAI-93, Chambéry, France, Morgan Kaufmann, San Mateo, CA, 1993, pp. 290–295.
[47] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, in: Proc. AAAI-92, San Jose, CA, 1992, pp. 440–446.
[48] P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, in: Proc. 4th Conference on Principles and Practice of Constraint Programming, Pisa, 1998, pp. 417–431.
[49] P. Stuckey, V. Tam, Extending GENET with lazy arc consistency, IEEE Trans. Systems Man Cybernet. 28 (5) (1998) 698–703.
[50] É. Taillard, Benchmarks for basic scheduling problems, European J. Oper. Res. 64 (1993) 278–285.
[51] G. Verfaillie, L. Lobjois, Problèmes incohérents: Expliquer l'incohérence, restaurer la cohérence, in: Actes des JNPC, 1999.
[52] G. Verfaillie, T. Schiex, Dynamic backtracking for dynamic CSPs, in: T. Schiex, C. Bessière (Eds.), Proc. ECAI-94 Workshop on Constraint Satisfaction Issues raised by Practical Applications, Amsterdam, 1994.
[53] G. Verfaillie, T. Schiex, Solution reuse in dynamic constraint satisfaction problems, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 307–312.
[54] M. Yokoo, Weak-commitment search for solving constraint satisfaction problems, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 313–318.