Explanation-based repair techniques for constraint programming

Narendra Jussien and **Romuald Debruyne**
École des Mines de Nantes
4, rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3
email: jussien@emn.fr

Résumé

Dans ce document, nous présentons un nouveau paradigme pour la programmation par contraintes : la programmation par contraintes avec explications. Nous montrons plus précisément l'intérêt de l'utilisation des explications pour définir des techniques de réparations qui permettent d'obtenir de nouveaux algorithmes efficaces pour résoudre les problèmes de satisfaction de contraintes.

Abstract

In this paper, we introduce a new paradigm for constraint programming: explanation-based constraint programming. We emphasize the interest of using explanations to design repair techniques in order to provide new efficient algorithms and heuristic for solving constraint satisfaction problems.

1 Introduction

Constraint satisfaction problems (CSP) [37] have proven to be an efficient model for solving many combinatorial and complex problems. Most of work on this topic rely on both enhancing search performances and extending the original model.

Explanations for constraint programming appear to be a good tool to be used for performing both objectives. Indeed, they can be used to store contradictions encountered during search and to identify relevant choice to undo when trapped in a dead-end. They therefore avoid falling repeatedly in the same portion of the search and reduce thrashing. Moreover, they extend the classical static CSP framework by allowing efficient constraint retractions.

In this paper, we introduce some new usage of explanations: designing explanation-based repair techniques to improve search. A new family of algorithms is introduced based on the `decision-repair` algorithm [26]. The interest of the new philosophy introduced by the active use of explanations within constraint programming is illustrated on a combinatorial optimisation problem.

Our paper is organized as follows: first, some theoretical background about CSP is recalled, then explanations are formally introduced. In Section 4, a new paradigm is introduced: explanation-based constraint programming. Finally, an application is described showing the interest of the new approaches.

2 Constraint Satisfaction Problems

We introduce here a formal model for representing both a constraint network and its resolution (domain reductions and constraint propagation).

2.1 The constraint network

Following [37], a *Constraint Satisfaction Problem* is made of two parts: a syntactic part and a semantic part. The syntactic part is a finite set V of variables, a finite set C of constraints and a function $\text{var} : C \to \mathcal{P}(V)$, which associates a set of related variables to each constraint. Indeed, a constraint may involve only a subset of V. For the semantic part, we need to consider various *families* $f = (f_i)_{i \in I}$. Such a family is referred to by the *function* $i \mapsto f_i$ or by the *set* $\{(i, f_i) \mid i \in I\}$.

2.1.1 Domains

$(D_x)_{x \in V}$ is a family where each D_x is a *finite non empty set* of possible values for x. We define the *domain of computation* by $\mathbb{D} = \bigcup_{x \in V}(\{x\} \times D_x)$. This domain allows simple and uniform definitions of (local consistency) operators on a power-set. For reduction, we consider subsets d of $\mathbb{D}$. Such a subset is called an *environment*. Let $d \subseteq \mathbb{D}$, $W \subseteq V$, we denote by $d|_W = \{(x, e) \in d \mid x \in W\}$. d is actually a family $(d_x)_{x \in V}$ with $d_x \subseteq D_x$: for $x \in V$, we define $d_x = \{e \in D_x \mid (x, e) \in d\}$. d_x is the *domain* of variable x.

2.1.2 Constraints as set of allowed tuples

Constraints are defined by their set of allowed tuples. A *tuple* t on $W \subseteq V$ is a particular environment such that each variable of W appears only once: $t \subseteq \mathbb{D}|_W$ and $\forall x \in W, \exists e \in D_x, t|_{\{x\}} = \{(x, e)\}$. For each $c \in C$, T_c is a set of tuples on $\text{var}(c)$, called the solutions of c. Note that a tuple $t \in T_c$ is equivalent to a family $(e_x)_{x \in \text{var}(c)}$ and t is identified with $\{(x, e_x) \mid x \in \text{var}(c)\}$.

We can now formally define a CSP and a solution to it.

Definition 1 *A* Constraint Satisfaction Problem *(*CSP*) is defined by: a finite set V of variables; a finite set C of constraints; a function $var : C \to \mathcal{P}(V)$; a family $(D_x)_{x \in V}$ (the domains); a family $(T_c)_{c \in C}$ (the constraints semantics).*

Definition 2 *A* solution *for a* CSP *$(V, C, var, (D_x)_{x \in V}, (T_c)_{c \in C})$ is a tuple s on V such that $\forall c \in C, s|_{var(c)} \in T_c$.*

2.2 Domain reduction and propagation

Two more key concepts need some details: the domain reduction mechanism and the propagation mechanism itself.

2.2.1 Local propagators

A constraint is fully characterized by its behavior regarding modification of the environments of the variables. *Local consistency operators* are associated with the constraints. Such an operator has a *type* (W_{in}, W_{out}) with $W_{in}, W_{out} \subseteq V$. For the sake of clarity, we will consider in our formal presentation that each operator is applied to the whole environment, but, in practice, it only removes from the environments of W_{out} some values which are inconsistent with respect to the environments of W_{in}.

Definition 3 *A* local consistency operator *of type* (W_{in}, W_{out})*, with* $W_{in}, W_{out} \subseteq V$*, is a monotonic function* $r : \mathcal{P}(\mathbb{D}) \rightarrow \mathcal{P}(\mathbb{D})$ *such that:* $\forall d \subseteq \mathbb{D}$, $r(d)|_{V \setminus W_{out}} = \mathbb{D}|_{V \setminus W_{out}}$*, and* $r(d) = r(d|_{W_{in}})$

Example 1 (Constraint $x \geq y + c$):
$x \geq y + c$ is one of the basic constraints in CHOCO. It is represented by the `GreaterOrEqualxyc` class. Reacting to an upper bound update for this constraint can be stated as: if the upper bound of x is modified then the upper bound of y should be lowered to the new value of the upper bound of x (taking into account the constant c). This is encoded as:

```
[awakeOnSup(c:GreaterOrEqualxyc,idx:integer)
 -> if (idx = 1)
       updateSup(c.v2,c.v1.sup - c.cste)]
```

`idx` is the index of the variable of the constraint whose bound (the upper bound here) has been modified. This particular constraint only reacts to modification of the upper bound of variable x (`c.v1` in the CHOCO representation of the constraint). The `updateSup` method only modifies the value of y (`c.v2` in the constraint) when the upper bound is really updated.
The `awakeOnSup` method can be considered as a local consistency operator with $W_{in} =$ {`c.v1`} and $W_{out} =$ {`c.v2`}.

Classically [15, 38, 4, 2], reduction operators are considered as *monotonic*, *contractant* and *idempotent* functions. However, on the one hand, *contractance* is not mandatory because environment reduction after applying a given operator r can be forced by intersecting its result with the current environment, that is $d \cap r(d)$. On the other hand, *idempotence* is useless from a theoretical point of view (it is only useful in practice for managing the propagation queue). This is generally not mandatory to design effective constraint solvers. We can therefore use only *monotonic* functions in definition 3.

The solver semantics is completely described by the set of such operators associated with the handled constraints. More or less accurate local consistency operators may be selected for each constraint. Moreover, this framework is not limited to arc-consistency but may handle any local consistency which boils down to domain reduction as shown in [16].

Of course local consistency operators should be *correct* with respect to the constraints. In practice, to each constraint $c \in C$ is associated a set of local consistency operators $R(c)$. The set $R(c)$ is such that for each $r \in R(c)$: let (W_{in}, W_{out}) be the type of r with $W_{in}, W_{out} \subseteq \text{var}(c)$; for each $d \subseteq \mathbb{D}$, $t \in T_c$: $t \subseteq d \Rightarrow t \subseteq r(d)$.

2.2.2 Constraint propagation

Propagation is handled through a propagation queue (containing events or conversely operators to awake). Informally, starting from the given *initial environment* for the problem, a local consistency operator is selected from the propagation queue (initialized with all the operators) and applied to the environment resulting to a new one. If an environment/domain reduction occurs, new operators (or new events) are added to the propagation queue.

Termination is reached when:

1. a variable environment is emptied: there is no solution to the associated problem;
2. the propagation queue is emptied: a common fix-point (or a desired consistency state) is reached ensuring that further propagation will not modify the result.

The resulting environment is actually obtained by sequentially applying a given sequence of operators. To formalize this result, let consider iterations.

Definition 4 *The* iteration *[2] from the initial environment $d \subseteq \mathbb{D}$ with respect to an* infinite *sequence of operators of R: $r^1, r^2, \ldots$ is the infinite sequence of environments $d^0, d^1, d^2, \ldots$ inductively defined by: $d^0 = d$; for each $i \in \mathbb{N}$, $d^{i+1} = d^i \cap r^{i+1}(d^i)$. Its* limit *is $\cap_{i \in \mathbb{N}} d^i$.*

A chaotic iteration *is an iteration with respect to a sequence of operators of R where each $r \in R$ appears infinitely often.*

The most accurate set which can be computed using a set of local consistency operators in the framework of domain reduction is the *downward closure*. *Chaotic iterations* have been introduced for this aim in [14].

Definition 5 *The* downward closure *of d by a set of operators R is $CL \downarrow (d, R) = \max\{d' \mid d' \subseteq d, \forall r \in R, d' \subseteq r(d')\}$.*

Note that if $R' \subseteq R$, then $CL \downarrow (d, R) \subseteq CL \downarrow (d, R')$.

Obviously, each solution to the CSP is in the downward closure. It is easy to check that $CL \downarrow (d, R)$ exists and can be obtained by iteration of the operator $d' \mapsto d' \cap \bigcap_{r \in R} r(d')$. Using *chaotic iteration* provides another way to compute $CL \downarrow (d, R)$ [12]. Iterations proceed by elementary steps. Chaotic iterations is a convenient theoretical definition but in practice each iteration is finite and fair in some sense.

Lemma 1 *The limit of every chaotic iteration of the set of local consistency operators R from $d \subseteq \mathbb{D}$ is the* downward closure *of d by R.*

This well-known result of confluence [12, 14] ensures that any chaotic iteration reaches the closure. Notice that, since $\subseteq$ is a well-founded ordering (i.e. $\mathbb{D}$ is a finite set), every iteration from $d \subseteq \mathbb{D}$ (obviously decreasing) is stationary, that is, $\exists i \in \mathbb{N}, \forall j \geq i, d^j = d^i$: in practice computation ends when a common fix-point is reached (*eg.* using a propagation queue).

2.2.3 Search

The computation of a solution to a constraint problem often needs a so-called *enumeration* phase. Indeed, in many occasions, constraint propagation is not sufficient to reduce the environment to a set of singletons.

The enumeration phase can be modelled as a sequence of constraint additions and retractions (backtracks). As long as no solution is found, a variable with a domain with at least two values is selected and a *decision* is made: reducing the domain[1]. This reduction can be considered as the addition of a *decision constraint* to the current constraint system.

Definition 6 *A* decision constraint *is a constraint that helps reducing at least one domain in an environment when solving a constraint satisfaction problem.*

Then, a propagation step is performed. As usual, if a domain is emptied, a backtrack occurs (the last decision is retracted); if the current environment is not reduced to a set of singletons, the decision process is repeated until a solution is found or no decision constraint is left to be retracted (the problem is over-constrained). This is a tree-based search process.

As the constraints system evolves throughout resolution, a notion of *context* is needed to describe a given state of the resolution.

Definition 7 *A* context *for a constraint satisfaction problem is composed of two sets: the set of the original constraints of the problem and a set of decision constraints. Typically, the latter represents the current path in the search tree that is being explored.*

3 Explanations for constraint propagation

Informally, an *explanation-set* is a set of constraints that *justifies* a domain reduction. As we will see through the concept of *explanation-tree*, explanations can be derived from the resolution of the CSP.

3.1 The basics: explanation-sets

Definition 8 *Let R be the set of all local consistency operators. Let $h \in \mathbb{D}$ and $d \subseteq \mathbb{D}$. We call* explanation-set *for h* w.r.t. *d a set of local consistency operators $E \subseteq R$ such that $h \notin CL \downarrow (d, E)$.*

Since $E \subseteq R$, $CL \downarrow (d, R) \subseteq CL \downarrow (d, E)$. Hence, if E is an explanation-set for h then each super-set of E is an explanation-set for h. An explanation-set E is independent of any chaotic iteration with respect to R.

For each $h \notin CL \downarrow (d, R)$, expl($h$) represents any explanation-set for h. Notice that for any $h \in CL \downarrow (d, R)$, expl($h$) does not exist.

[1] Classically, such a decision amounts to reduce the current domain to a single value (variable assignment) but this can be more general as in numeric CSP where a splitting is made, or for scheduling problem where a precedence constraint is posted

3.2 More information: explanation-trees

Explanation-sets are a compact representation of a sufficient set of constraints to achieve a given domain reduction. A more complete description of the interaction of the constraints responsible for this domain reduction can be introduced through *explanation-trees*. We need to introduce the notion of deduction rule related to local consistency operators.

3.2.1 Deduction rules and local consistency operators

Definition 9 *A* deduction rule *of type* (W_{in}, W_{out}) *is a rule* $h \leftarrow B$ *such that* $h \in \mathbb{D}|_{W_{out}}$ *and* $B \subseteq \mathbb{D}|_{W_{in}}$.

The intended semantics of a deduction rule $h \leftarrow B$ can be presented as follows: if all the elements of B are removed from the environment, then h does not appear in any solution of the CSP and may be removed harmlessly.

A set of deduction rules $\mathcal{R}_r$ may be associated with each local consistency operator r. It is intuitively obvious that this is true for arc-consistency enforcement but it has been proved in [16] that for any local consistency which boils down to domain reduction it is possible to associate such a set of rules (moreover it shows that there exists a natural set of rules for classical local consistencies). It is important to note that, in the general case, there may exist several rules with the same head but different bodies.

We consider the set $\mathcal{R}$ of all the deduction rules for all the local consistency operators of R defined by $\mathcal{R} = \cup_{r \in R} \mathcal{R}_r$.

The initial environment must be taken into account in the set of deduction rules: the iteration starts from an environment $d \subseteq \mathbb{D}$; it is therefore necessary to add facts (deduction rules with an empty body) in order to directly deduce the elements of $\overline{d}$: let $\mathcal{R}^d = \{h \leftarrow \emptyset \mid h \in \overline{d}\}$ be this set.

3.2.2 Proof-trees

Definition 10 *A* proof tree *with respect to a set of rules* $\mathcal{R} \cup \mathcal{R}^d$ *is a finite tree such that for each node labelled by* h*, let* B *be the set of labels of its children,* $h \leftarrow B \in \mathcal{R} \cup \mathcal{R}^d$.

Proof trees are closely related to the computation of environment/domain reduction. Let $d = d^0, \ldots, d^i, \ldots$ be an iteration. For each i, if $h \notin d^i$ then h is the root of a proof tree with respect to $\mathcal{R} \cup \mathcal{R}^d$. More generally, $\overline{CL \downarrow (d, R)}$ is the set of the roots of proof trees with respect to $\mathcal{R} \cup \mathcal{R}^d$.

Each deduction rule used in a proof tree comes from a packet of deduction rules, either from a packet $\mathcal{R}_r$ defining a local consistency operator r, or from $\mathcal{R}^d$.

A set of local consistency operators can be associated with a proof tree:

Definition 11 *Let* t *be a proof tree. A* set X of local consistency operators associated with t *is such that, for each node of* t*: let* h *be the label of the node and* B *the set of labels of its children: either* $h \notin d$ *(and* $B = \emptyset$*); or there exists* $r \in X, h \leftarrow B \in \mathcal{R}_r$.

Note that there may exist several sets associated with a proof tree. Moreover, each super-set of a set associated with a proof tree is also convenient (R is associated with all proof trees). It is important to recall that the root of a proof tree does not belong to the closure of the initial environment d by the set of local consistency operators R. So there exists an explanation-set (definition 8) for this value.

Lemma 2 *If t is a proof tree, then each set of local consistency operators associated with t is an explanation-set for the root of t.*

From now on, a proof tree with respect to $\mathcal{R} \cup \mathcal{R}^d$ is therefore called an *explanation-tree*. As we just saw, *explanation-sets* can be computed from *explanation-trees*.

3.3 Basic properties for explanations

Characterizing explanations is helpful when comparing or using explanations.

3.3.1 Preciseness

Definition 12 *An explanation-set e_1 is said to be* more precise *than explanation-set e_2 iff $e_1 \subset e_2$.*

This is a simple way of defining precise explanation-sets. However, there exists other possibilities: preciseness could be defined regarding the reduction power of operators, the scope (number of involved variables) of the constraints, etc.

Notice that determining for any value removal the most precise explanation-set (at least on of them as several not comparable ones may exist) often amounts to solve an NP-hard problem.

3.3.2 Validity and k-relevance

Explanation-sets are a self-contained concept. However, a given explanation-set may not be relevant in the current context (the set of constraints and active decisions).

Definition 13 *An explanation-set e is said to be* valid *or* relevant w.r.t. *the current context C iff*

$$\{c | \exists r \in e, r \in R(c)\} \subset C$$

Following [3], we introduce the concept of k-relevance for explanation-sets to measure their distance from full validity.

Definition 14 *An explanation-set e is said to be k-*relevant w.r.t. *the current context C iff*

$$\#\{c | \exists r \in e, r \in R(c), c \notin C\} < k$$

4 Explanation-based search for constraint programming

The usage of explanations within a constraint programming language leads to new programming gimmicks and new approaches. Those new approches are designed under the name explanation-based constraint programming (e-constraints for short).

4.1 From backtrack-based to explanation-based solving

Most of CSP solving algorithms are derived from a backtrack-based complete search. The drawbacks of this approach have been known for a long time: thrashing and backtracking to irrelevant choice points. The previously developed fancy backtrackers were not really convincing to address that issue (time or space overhead, no real advantages) [5]. Nevertheless, more recently, the `mac-dbt` algorithm [25] showed that explanation-based algorithms could compete well with backtrack-based algorithms in real-world situations. `mac-dbt` algorithm is to `dbt` what `mac` is to `sb`. That algorithm can be described using a very generic algorithm that appears as the archetype of explanation-based constraint solving.

4.1.1 From standard backtracking to dynamic backtracking

Most of complete search algorithms over Constraint Satisfaction Problems (CSP) are based on *Standard Backtracking* (`sb`): a depth-first search is performed using chronological backtracking. Various *intelligent* backtrackers have been proposed: *Conflict-directed BackJumping* (`cbj`) [32], *Dynamic Backtracking* (`dbt`) [17], *Partial order Dynamic Backtracking* (`pdb`) [18], *Generalized Dynamic Backtracking* (`gpb`) [6], etc. In those algorithms, information (namely a special case of explanations: nogoods) is kept when encountering inconsistencies so that the forthcoming search will not get back to already known traps in the search space.

Dependency Directed Backtracking (`ddb`) [35] was the first algorithm to use this enhancement, however it has an important drawback: its space complexity is exponential since the number of nogoods it stores increases monotically. To address this problem, algorithms such as `cbj` or `dbt` eliminate nogoods that are no longer relevant to the current variable assignment. By doing so, the space complexity remains polynomial.

When a failure occurs, those algorithms have to identify the assignment to be reconsidered (suspected to be a **culprit** for the failure).

- `sb` always considers the most recent assignment to be a culprit. This selection may be completely irrelevant for the current failure leading to useless exploration of parts of the search tree already known to be dead-ends (thrashing).
- In `cbj` a conflict-set is associated to each variable: CS_{v_i} (for the variable v_i) contains the set of the assigned variables whose value is in conflict with the value of v_i. When identifying a dead-end while assigning v_i, `cbj` considers the most recent variable in CS_{v_i} to be a culprit. A backtrack then occurs: the conflict-sets and domains of the *future* variables are reset to their original value. By doing so, `cbj` forgets a lot of information that could have been useful. This also leads to thrashing.
- `dbt` selects the most recent variable in the *computed* nogood (the conflict-set of `cbj`) in order to undo the assignment. However, thanks to the explanations, `dbt` only removes related information that depends on it and so avoids thrashing: useful information is kept. Indeed, there is no real backtracking in `dbt` and like in a repair method, only the assignments that caused the contradiction are undone.

Notice that `sb` can also be considered as selecting the most recent assignment of a nogood, namely the nogood that contains all the current variable assignments (which

fails to give really relevant information).

4.1.2 Integrating constraint propagation within dynamic backtracking

Constraint propagation has been included in `sb` leading to forward checking `fc` and to the *Maintaining Arc-Consistency* algorithm (`mac`) [33]. `mac` is nowadays considered as one of the best algorithms for solving CSP[5].

Several attempts to integrate constraint propagation within intelligent backtrackers have been done: for example, Prosser has proposed `mac-cbj` which maintains arc consistency in `cbj` [32]. But, Bessière and Régin [5] have stopped further research in that field by showing that `mac-cbj` was very rarely better than `mac`. They concluded that there was no need to spend time nor space for intelligent backtracking because the brute force of `mac` simply does it more quickly. From our point of view, the inadequacy of `mac-cbj` is more related to the fact that `cbj` does not avoid thrashing[2] than to the cost of the management of nogoods. When backtracking occurs, `cbj` comes back to a relevant assignment, and then forgets all the search space developed since this assignment has been performed: as `sb`, `cbj` has a **multiplicative** behavior on independent sub-problems. `dbt` does not only use nogoods to perform *intelligent* backtracking but also to avoid thrashing and so becomes **additive** on independent sub-problems [17]. [5] had another point preventing the use of nogoods: it is always possible to find an intelligent labelling heuristic so that a *standard backtracking*-based algorithm will perform a search as efficiently as an intelligent backtracker. In our experience, using a good heuristic reduces the number of problems on which the algorithm thrashes but does not make it additive on independent subproblems: there are still problems on which the heuristic cannot prevent thrashing.

In [25], we introduced the `mac-dbt` algorithm which shows the efficient integration of constraint propagation within `dbt` thanks to the use of explanations. The general behavior of `mac-dbt` is described in the algorithm in Figure 1: as long as no solution has been found, choose a variable and a value for it, assign the value to the variable (line 11), propagate this new information (line 12). If a contradiction occurs during that process, use a special handling contradiction procedure (line 15 and the algorithm in Figure 2).

This special handling contradiction procedure merely amounts to a backtrack if a standard backtrack-based search is desired. However, if one wants to take benefit from an explanation-based solver, an explanation for this failure can be computed (see line 2 in the algorithm in Figure 2) and, if possible, a choice (an assignment constraint) to undo[3] is selected (line 7). At this point, an intelligent backtracker can be defined by performing a classical backtrack to that selected choice. In order to use explanations as much as possible, a dynamic constraint removal should be performed (line 12), followed by the posting within a given context[4] of the negation of the undone decision and the propagation of the new information (line 15). It is important to recursively handle any contradiction that may appear (line 18).

[2] A thrashing behavior consists in repeatedly performing the same search work due to the backtrack mechanism.

[3] Notice that this choice is tightly guided by the completeness requirements of the algorithms (see [6]).

[4] Line 14 in the algorithm in Figure 2 introduces the constraint opposite(ct) which will remain active as long as the context e remains valid. See [25] for more information on that point.

```
function solve(pb: Problem): boolean
(1)  begin
(2)    unassignedVars ← pb.vars
(3)    try (
(4)       while not(empty(unassignedVars))
(5)           let idx ← nextVarToAssign(pb)    // variable choice
(6)               v ← unassignedVars[idx]
(7)               a ← selectValToAssign(pb, v)    // value choice
(8)           in (
(9)               try (
(10)                   unassignedVars :delete v
(11)                   post(pb, v == a)     // instantiation
(12)                   propagate(pb)
(13)               )
(14)               catch LabelingContradiction     // An empty domain found
(15)                   handleContradiction(pb)     // classically: BACKTRACK
(16)               )
(17)          )
(18)      endwhile
(19)      true
(20)   )
(21)   catch ProblemContradiction
(22)      false    // No solution
(23)   )
(24) end
```

Figure 1: Solving a CSP

```
procedure handleContradiction(pb: Problem)
(1)  begin
(2)    let e ← becauseOf(theDom(getFailingVariable(pb)))    // conflict explanation
(3)    in (
(4)       if e empty then
(5)          raiseProblemContradiction()
(6)       else
(7)          let ct ← selectConstraint(e)    // select a to be removed choice
(8)          in (
(9)              if ct exists then
(10)                 unassignedVars :add ct.v1
(11)                 try (
(12)                     remove(ct)    // perform constraint removal
(13)                     e :delete ct
(14)                     post(pb, opposite(ct), e)    // context-guarded negation
(15)                     propagate(pb))    // achieving consistency
(16)                 )
(17)                 catch LabelingContradiction
(18)                     handleContradiction(pb)    // recursive handling
(19)                 )
(20)             else
(21)                 raiseProblemContradiction()
(22)             endif
(23)         )
(24)      endif
(25)   )
(26) end
```

Figure 2: Contradiction handling

4.1.3 A generic algorithm

One can identify three different components[5] in algorithms 1 and 2 that help understand and generalizing their behavior:

- a **propagation** component that is used to propagate information throughout the constraint network when a decision is made during search. Two operators are needed: `filter`ing and `check`ing if a solution can exist.
- a **learning** component that is used to make sure that the search mechanism will avoid (as much as possible) to get back to states that have been explored and proved to be solution-less. Using a rough analogy with the brain, we will use two operators: a `record`ing operator that learns new pieces of information and a `forget`ting operator that will make room for new information to be learnt.
- a **moving** component whose aim is, unlike the other two components, to explore the search space instead of pruning it. There are two moving operators: `repair` to be used when the current constraints system is contradictory and need some modification and `extend` to potentially add new information when no contradiction has not yet been detected but when no solution has be found.

Those three components can be used to design a generic CSP solving algorithm (the PLM algorithm – see figure 3) that encompasses complete and incomplete searches, prospective and retrospective algorithms [27]:

- the search starts from an initial set of decision constraints that may range from the empty set (typically for backtrack-based search) or a total assignment (typically for local search algorithm)
- decisions are made (`extend`) and propagated (`filter`) until a contradiction occurs
- when a contradiction does occur (line 6), the information related to the dead-end (*e.g.* a conflict explanation) is learnt (`record`), the current state is repaired (`repair`) and some information is *forgotten* (`forget`).
- the search terminates as soon as a solution is found (line 8) or the conditions of termination (line 13) are fulfilled. Conditions of termination can be for example a maximum number of iterations, the exhibition of a proof that no solution exists, etc.

Using explanations greatly helps concretely implementing such a generic framework. Indeed, explanations can be used to determine precise conflict explanations, perform dynamic reparations of any constraints system, etc. The PaLM systems effortlessly implements this generic framework.

4.2 A new family of algorithms: decision-repair

The **PLM** generic algorithm has been used to described several well known algorithms [27]:

[5] A formal description about these components can be found in [27].

```
procedure PLM(V,C,C_D)
(1)  begin
(2)     P ← {V, C, C_D}
(3)     repeat
(4)        P ← filter(P)
(5)        switch check(P)
(6)           case no solution :
(7)              P ← forget(repair(record(P)))
(8)           case solution found :
(9)              return P
(10)          case not enough information :
(11)             P ← extend(P)
(12)       endswitch
(13)    until conditions of termination
(14) end
```

Figure 3: A generic algorithm for the PLM components

- systematic algorithms such as `sb`, `cbj`, `dbt`, `mac`, `mac-dbt`, etc.
- non-systematic algorithms such as `tabu search` [19], `gsat` [34], etc.

The **PLM** generic algorithm has been also used to design new algorithms: the `decision-repair` family [26]. The idea of `decision-repair` is to combine the propagation-based nature of `mac-dbt` and the true freedom (in the search space exploration) given by a local search algorithm such as `tabu search`. Therefore, in terms of the **PLM** generic algorithm, we have:

- the starting set of decision constraints is empty;
- `filter` uses standard filtering algorithm for reducing the domain of the variables of the problem;
- `record` computes an explanation-set for the current contradiction and stores it in a *tabu* list of fixed size K;
- `forget` erases the oldest stored explanation-set if the *tabu* list is full;
- `extend` classically adds new decisions (variable assignment, domain splitting, etc.) as long as no solution has been found yet;
- `repair` heuristically selects a decision to undo from the last computed explanation-sets (and whose negation is compatible with the stored explanation-sets).

As several parameters remain unprecised (the way of handling the *tabu* list, the heuristic to be used to select decisions to undo, etc.), `decision-repair` is a family of algorithms. However, 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.

A comprehensive study of the behavior of `decision-repair` has shown that the key components of this algorithm are: its explanations-directed heuristics and its ability both to perform a local search and to prune the search space [26]. Experiments with `decision-repair` have shown good results over open-shop scheduling problems (see section 5.2).

5 A case study: solving open-shop scheduling problems

We experienced using explanation-based constraint programming algorithms for solving open-shop scheduling problems for several years now. First we developed intelligent backtrackers that provided first good results and greatly improved our techniques using more explanation-based features of e-constraints. This section provides some details about these results.

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[6]. One of those problems is called the open-shop problem [20]. 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[7].

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

Only two branch-and-bound methods for that problem have been published so far. The first one [7] is based on the resolution of a one-machine problem with positive and negative time-lags. The second one, [9], consists, in each node, in fixing disjunctions on the critical path of a heuristic solution. It combines two concepts:

- a generalization of a branching scheme first introduced by Grabowski *et al.* [21] for one-machine problems with release dates and due dates;
- *immediate selections* [10], a method initially designed to fix disjunctions in Job-Shop problems.

We used that method as a basis for our experiments and enhanced it in two steps:

1. designing an intelligent backtracker by using an explanation-based constraint solver to propagate new decisions made during search;
2. use the `decision-repair` family of algorithms to design a new efficient heuristic for solving open-shop scheduling problems.

5.2 An efficient heuristic technique

The open-shop problem being very hard to solve exactly, various heuristics have been proposed[8]: greedy heuristics such as specific list heuristics [23] or local searches such as highly specialized tabu searches [1, 29] or genetic algorithms [30], etc.

We tried `decision-repair` on the open-shop problems using one of its implementation described below:

[6] 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.

[7] Ending time of the last task.

[8] We mention here heuristics that are considered to be the best thechniques to solve open-shop scheduling problems.

Filtering technique Precedence constraints are handled with 2B-consistency filtering [13, 28] and resource usage constraints are handled through *task-intervals* [11].

Search strategy For shop scheduling 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[9]. We use the branching scheme introduced in [9] to select such decisions.

Tabu list The implementation uses a tabu list of size 7.

Repair The `repair` function we used records in the tabu list the newly computed contradiction explanation k (is is a conflict). It tries to find one decision in k such that negating this decision makes the decision set compatible with all the stored conflicts. When several decisions can be negated, we use the following 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. The `repair` function chooses to negate the decision with the greatest weight that, when negated, makes the new decision set compatible with all the conflicts in tabu list. If such a decision does not exist, it is a considered as a stopping criterion for the overall algorithm.

Stopping criterion The failure conditions specifying the exit of `decision-repair` are either a *stop* returned by the `repair` 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 `decision-repair` until improvement is no longer possible. (See Figure 4.) Improvements are forced 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).

We studied three series of reference problems:

1. **Taillard**'s problems [36]: 10 square instances of size 4, 5, 7 and 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[10]) value: 1000.
3. **Guéret and Prins**' problems[11]: Those 80 problems (8 series of 10 problems of size 3×3 to 10×10) have been generated using results presented in [24] for generating really hard open-shop instances. They all share a common LB (classical lower bound) value and the fact that another lower bound [24] gives a much greater value.

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

[9] When every possible precedence has been posted, setting the starting date of the variable to their smallest value provides a feasible solution.

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

[11] Available at `http://www.emn.fr/gueret/OpenShop/OpenShop.html`.

```
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
```

Figure 4: Algorithm used to solve open-shop problems

- 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 [29] (referred to as **TS-L98** in the results).
- For all the instances, our results are also compared with a genetic algorithm introduced in [30] (referred to as **GA-P99** in the results) which gives very good results on all those problems.

Figure 1 presents results obtained on Taillard's problems, Figure 2 on Brucker's instances, and finally Figure 3 on Guéret and Prins's problems. Cpu time is not available in [29], [1], nor in [30][12]. *Average* cpu time for `decision-repair` is not really significant 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's 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.

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)

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

[12]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 [31] 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!

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)

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 time for which the deviation is computed from the LB value (1000 for each problem).

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

Table 3: **Results on Guéret and Prins's 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 an intelligent backtracker described in [22] 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"

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 our algorithm is a competitive algorithm compared with the other techniques.

As far as Taillard's instances are concerned, `decision-repair` gives comparable results but the *tabu search* of [29] is still the best technique except for 10×10 problems where the genetic algorithm shows the best results.

On Brucker's instances, `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) `decision-repair` shows all the interest of combining local search and constraint propagation: `decision-repair` closed[13] 6 of these instances. Furthermore, it 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, `decision-repair` gives far better results than both the genetic algorithm and branch and bound search (that has been truncated by a time criterion).

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

For 10×10 problems, `decision-repair` is still better than the branch and bound but is matched by the genetic algorithm.

Such good behavior of `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. `decision-repair`, thanks to its use of explanations, 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 explanations.

5.3 Analysis

We performed a comprehensive study of the behavior of `decision-repair` in [26]. It 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.

6 Conclusion

In this paper, we introduced a new paradigm for constraint programming: explanation-based constraint programming. We emphasized the interest of using explanations to design repair techniques in order to provide new efficient algorithms and heuristic for solving constraint satisfaction problems. Our results clearly advocate for the use of explanations within constraint programming for providing new techniques and new programmation gimmicks.

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] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2):179–210, 1999.

[3] Roberto J. Bayardo Jr. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI'96*, 1996.

[4] Frédéric Benhamou. Heterogeneous constraint solving. In Michael Hanus and Mario Rofríguez-Artalejo, editors, *International Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pages 62–76. Springer-Verlag, 1996.

[5] Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problem. In *CP'96*, Cambridge, MA, 1996.

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

[7] P. Brucker, T. Hilbig, and J. Hurink. A branch and bound algorithm for scheduling problems with positive and negative time-lags. Technical report, Osnabrueck University, may 1996.

[8] P. Brucker, B. Jurish, B. Sievers, and B. Wöstmann. A branch and bound algorithm for the open-shop problem. *Discrete Applied Mathematics*, 76:43–49, 1997.

[9] P. Brucker, S. Knust, A. Schoo, and O. Thiele. A branch and bound algorithm for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 107:272–288, 1998.

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

[11] Yves Caseau and François Laburthe. Improving clp scheduling with task intervals. In P. Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming, ICLP'94*, pages 369–383. MIT Press, 1994.

[12] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions mathematical foundation. In *Symposium on Artificial Intelligence and Programming Languages*, volume 12(8) of *ACM SIGPLAN Not.*, pages 1–12, 1977.

[13] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(2):281–331, 1987.

[14] François Fages, Julian Fowler, and Thierry Sola. A reactive constraint logic programming scheme. In *International Conference on Logic Programming*. MIT Press, 1995.

[15] François Fages, Julian Fowler, and Thierry Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37(1-3):185–212, 1998.

[16] Gérard Ferrand, Willy Lesaint, and Alexandre Tessier. Theoretical foundations of value withdrawal explanations for domain reduction. *Electronic Notes in Theoretical Computer Science*, 76, 2002.

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

[18] 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.

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

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

[21] J. Grabowski, E. Nowicki, and S. Zdrzalka. A block approach for single-machine scheduling with release dates and due dates. *European Journal of Operations Research*, 26:278–285, 1986.

[22] 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*, 127(2):344–354, 2000.

[23] Christelle Guéret and Christian Prins. Classical and new heuristics for the open-shop problem. *European Journal of Operations Research*, 107(2):306–314, 1998.

[24] Christelle Guéret and Christian Prins. A new lower bound for the open-shop problem. *AOR (Annals of Operations Research*, 92:165–183, 1999.

[25] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.

[26] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, July 2002.

[27] Narendra Jussien and Olivier Lhomme. Unifying search algorithms for CSP. Research Report 02-3-INFO, École des Mines de Nantes, Nantes, France, 2002.

[28] O. Lhomme. Consistency techniques for numeric CSPs. In *IJCAI'93*, pages 232–238, Chambéry, France, August 1993.

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

[30] Christian Prins. Competitive genetic algorithms for the open shop scheduling problem. Research report, École des Mines de Nantes, 99/1/AUTO, 1999.

[31] Christian Prins and Jacques Carlier. Resource optimization in a TDMA/DSI system: the eutelsat approach. In *Proceedings of the International Conference on Digital Satellite Communications (ICDSC 7)*, pages 511–518, Munich, Germany, 1986.

[32] Patrick Prosser. MAC-CBJ: maintaining arc-consistency with conflict-directed backjumping. Research Report 95/177, Department of Computer Science – University of Strathclyde, 1995.

[33] Daniel Sabin and Eugene Freuder. Contradicting conventional wisdom in constraint satisfaction. 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).

[34] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *AAAI-92: Proceedings 10th National Conference on AI*, pages 440–446, San Jose, July 1992.

[35] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

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

[37] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[38] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming. MIT Press, 1989.