

# Best-first search for property maintenance in reactive constraints systems

Narendra Jussien and Patrice Boizumault

École des Mines de Nantes – Département Informatique

4 Rue Alfred Kastler – BP 20722

F-44307 Nantes Cedex 03 – France

Narendra.Jussien@emn.fr, Patrice.Boizumault@emn.fr

## Abstract

Real-life dynamic problems may lead to inconsistent constraints systems for which a solution must be found even if constraints have to be relaxed. In this paper, we propose a best-first search to handle such problems. Classical backtracking search algorithms are extended in two ways: identification of good backtrack points as in Intelligent Backtracking techniques and maximum use of independant work (that would have been discarded with a mere *backtrack*). We first describe an operational semantics for our search method. Then we specialize it to handle constraint relaxation over finite domains. The practical use of this approach is demonstrated by theoretical complexity analysis and experiments.

## 1 Introduction

Constraints are nowadays widely used for solving problems arising in various fields such as Artificial Intelligence, Operations Research, ... Constraint Programming Languages and Systems are mainly designed to handle *static* problems. But, many real life problems are *dynamic*: the problem evolves throughout modifications induced by the outside environment (*e.g.* consider the necessity of on-line rescheduling when a machine is damaged or not usable). Performing a complete reexecution from scratch is not realistic. First, the amount of work can be prohibitive if many modifications are to be taken into account. Second, the resulting solution may be far away from the initial one. It seems more efficient to use the previous computation to ease the search of a solution.

In order to handle dynamic problems, Fages *et al.* [5] have proposed an extension of Constraint Logic Programming to *reactive systems* [8]. The objective of such a system is not to produce a single *input-output* relation, but to maintain a given property of the considered system through any modification. Hence, incrementality is a key aspect of reactive systems.

When considering large amounts of modifications, dynamic problems can lead to inconsistent constraint systems (they are over-constrained). In many cases, a solution is required even if constraints (that are considered as not being important) are not taken into account *i.e.* relaxed. Despite many works handling over-constrained static problems have been developped [6; 20], very few can handle *dynamic* problems [12] without a mere complete reexecution.

In this paper, we present an algorithmic scheme to handle over-constrained problems in a dynamic environment. Our approach consists in considering the constraint solver over a domain  $\mathcal{D}$  as a reactive system which interacts with the outside environment by continuously maintaining a property  $\mathcal{P}$  over the current

constraint system<sup>1</sup>. When this property is not verified, some constraints will be relaxed according to their relative importance. Our approach relies on the notion of configuration : a *configuration* is the splitting of the constraints of the problem in two sets: active and relaxed ones. A user-defined comparator enables discrimination between configurations.

We define a best first search method over the configurations space that is well suited for dynamic problems because it actively uses past computations to ease future computation. This search step is performed when a contradiction occurs during the property maintenance. A configuration that is not contradictory and suits the user’s preferences is determined. The key idea is to benefit from the past computation *i.e.* to provide a search method that combines *intelligent* backtracking with avoiding *thrashing*: recomputation of already explored parts of the search space. This search method is parameterized by the *domain* on which constraints are defined and by the *property* maintained during the computation.

The paper is organized as follows: in section 2, we introduce definitions related to over-constrained problems. In section 3 we briefly recall related works. In section 4, we present our search method for which an operational semantics is given section 5. Section 6 shows how our method can be successfully instantiated for constraints over Finite Domains and the property of Arc-Consistency, and efficiently implemented using a Deduction Maintenance System. Complexity issues and experimental results are addressed in section 7.

## 2 Definitions

In order to *solve* over-constrained problems, **preferences** over the constraints of the problem should be specified by the user. Such a preference<sup>2</sup> represents the will of the user regarding the activation of the constraints. When grouping constraints by preference levels, an **hierarchy** is built upon the constraints.

A **configuration** is a split of a given constraints system in two sets: the set  $A$  of active constraints and the set  $R$  of relaxed ones. It is noted:  $\langle A, R \rangle$ .

We consider a property  $\mathcal{P}$  that is defined for sets of constraints<sup>3</sup>.  $\mathcal{P}$  is necessary (but not necessarily sufficient) for the satisfiability of the constraints system. It thus represents a certain level of consistency. A configuration  $\langle A, R \rangle$  is called  **$\mathcal{P}$ -satisfiable** if  $\mathcal{P}$  holds for  $A$  and  **$\mathcal{P}$ -contradictory** otherwise.

The user provides a **comparator** based upon the hierarchy. Such a comparator enables selection between  $\mathcal{P}$ -satisfiable configurations. More precisely, a comparator is a partial order relation upon configurations. It must respect the hierarchy [20] defined from the user preferences. We can consider the following comparator used in [17]:

**Definition 2.1 (The  $C_{MM}$  comparator)**

Let  $C_1 = \langle A_1, R_1 \rangle$  and  $C_2 = \langle A_2, R_2 \rangle$  two configurations.

$$C_{MM}(C_1, C_2) \equiv \begin{aligned} &\exists k > 0, \text{ tel que} \\ &\forall i < k, R_{1[i]} = R_{2[i]} \\ &R_{1[k]} = \emptyset \text{ and } R_{2[k]} \neq \emptyset \end{aligned} \quad (1)$$

where  $R_{[\ell]}$  is the restriction of  $R$  to the constraints with a preference<sup>4</sup> level  $\ell$  in the hierarchy.

---

<sup>1</sup>This property could be Arc-consistency for finite domains, B-Consistency for Intervals, “full” consistency for linear constraints over rationals, ...

<sup>2</sup>A preference can be considered as a *weight* on the constraint.

<sup>3</sup>For example, one can consider global consistency for rationals or local consistencies for finite domains or intervals.

<sup>4</sup>The higher the preference level, the less important is the constraint.

A  **$C$ -solution** for an over-constrained problem is then defined as a  $\mathcal{P}$ -satisfiable configuration maximum for a given comparator  $C$ .

### 3 Related Works

Hierarchical Constraint Logic Programming (HCLP) [20] is an extension of CLP handling constraints hierarchies. The operational behavior of HCLP is like building a *tower of constraints*. Constraints are introduced level by level in the constraint store (from the most important to the less important ones) until an inconsistency occurs. This approach needs to completely know the constraints system before resolution (this is not suitable for dynamic environments).

In order to handle dynamic problems within the HCLP framework, Menezes and Barahona [12] proposed the IHCS (Incremental Hierarchical Constraint Solver) system. This approach uses works on Intelligent Backtracking. As all backtracking techniques do, this approach suffers from the *thrashing* behavior: recomputation of already explored parts of the search space. Furthermore, despite its incremental handling of constraint addition, constraint relaxations are handled in the *hard* way: using a *backtrack* and not incrementally. IHCS is then not a completely satisfactory answer to handle over-constrained problems arising in dynamic environments.

As mentioned in the introduction, Fages *et al.* proposed an extension of CLP to reactive systems [5]. A reactive system allows interactions between the user and the solver. The aim is not only to give a solution to an instance of a problem but also to maintain such a solution throughout interactions with the user. The framework proposed in [5] handles constraint addition or suppression but unfortunately does not consider over-constrained systems<sup>5</sup>. Indeed, constraints to be suppressed must be explicitly specified by the user. Our proposal provides the necessary machinery to handle those problems and from this point of view extends Fages' proposal.

Local propagation is well suited for solution maintenance on dynamic functional constraints systems. A constraint is *functional* if, for each of its constrained variables  $v$ , there is a unique value of  $v$  that will satisfy the constraint, given values for the other variables. Such a constraint is defined by the set of functions that compute this value for each set of fixed variables. Local propagation algorithms for functional constraints handling constraint hierarchies with no cycle in the constraint graph provide efficient constraint solvers. Recent works include the DeltaBlue and SkyBlue algorithms [16], and QuickPlan [19]. But such techniques are not so well adapted for dynamic non functional constraints systems. Recently, Borning *et al.* proposed the Indigo system [3] to handling inequality constraints (a particular case of non functional constraints) using local propagation techniques. Unfortunately, such a method seems not suited for a dynamic environment; as said by the authors: *producing an incremental version of Indigo does not seem to be straightforward*.

From another view point, the CSP (Constraint Satisfaction problems) community is used to handle dynamic problems [1; 14]. They mainly implement Maintenance Systems inherited from the TMS community [4]. But, unfortunately no satisfactory handling of over-constrained dynamic CSPs has ever been proposed. Indeed, works such as [2; 6; 17] are meant for static problems and do not provide fully incremental systems for dynamic ones. Nevertheless, the idea of maintaining information seems the wisest thing to do when handling dynamic CSP.

---

<sup>5</sup>When such a system arises, it just says **no solution**.

## 4 A best-first search

We present here a search method over the configurations space that ensures that as soon as a  $\mathcal{P}$ -satisfiable configuration is identified, this configuration is the best one regarding the user-defined comparator (thus providing a  $C$ -solution). The idea of the search inherits from the *Dynamic Backtracking* algorithm<sup>6</sup> [7]. Standard backtracking is enhanced in two ways: first, upon failure, a relevant choice point is reconsidered instead of merely considering the last choice point (this is *Intelligent backtracking*), and, second, *thrashing* is avoided by keeping information gathered between the backtrack point and the current point that is independent from the failure. Of course, conditions of *backtrack* must be recorded in order to ensure the completeness of the search.

### 4.1 The configurations space

The configurations space can be considered as a binary tree. Each node is a constraint whose children represent the two possible states: active or relaxed. This tree is built considering the entrance order of the constraints (the problem is dynamic).

For a node associated with a constraint  $c_a$ , an outgoing edge is labeled  $a$  (*resp.*  $a'$ ) for the active state (*resp.* relaxed state). Figure 1 shows such a tree for three constraints  $c_a, c_b$  and  $c_c$ .

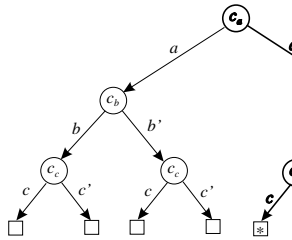


Figure 1: Configurations space for  $\{c_a, c_b, c_c\}$ .

Each leaf (or path from the root to a leaf) of this tree represents a *configuration*. For example, the bold branch ( $a'bc$ ) in figure 1 represents the configuration:  $\langle \{c_b, c_c\}, \{c_a\} \rangle$ .

Solving an over-constrained problem requires to find the best (considering the comparator  $C$ )  $\mathcal{P}$ -satisfiable leaf (configuration).

### 4.2 Classical Explorations

Explorations based upon backtracking (standard backtracking or *intelligent backtracking* [5; 12]) have two main drawbacks: on one hand, an optimality proof step is required as a  $\mathcal{P}$ -satisfiable configuration is found; and on the other hand, these approaches suffer from *thrashing*.

We claim here that by recording the same information as in *intelligent backtracking* techniques, the search can be improved by not only computing good backtrack points but also avoiding *thrashing* by using previously gathered information.

<sup>6</sup>*Dynamic Backtracking* is an enumeration algorithm designed for CSPs. Enumeration is done here on the state of the constraints (active or relaxed). Moreover, unlike in *Dynamic Backtracking*, constraint propagation is considered. Finally, our proposal is meant for over-constrained problems which *Dynamic Backtracking* cannot deal with.

### 4.3 A best first approach

Our search method does not perform backtracks but *jumps*. These *jumps* lead to a non classical exploration of the search space that is complete and optimum. The proposed approach is a general method that can be applied to different constraints domain and for different consistency techniques. We suppose from now on that the following three functions are given: **explain-contradiction**, **best-configuration** and **change-configuration**. Section 6 provides complete definitions of these functions for finite domains.

#### 4.3.1 Contradiction explanations

Our method relies upon the key concept of **contradiction explanation**<sup>7</sup>. A *contradiction explanation* is a set of constraints whose conjunction leads to a contradiction. It is a *justification* for the contradiction. This notion is strongly related to the property  $\mathcal{P}$  maintained throughout the computation for the current set of active constraints.

**Definition 4.1 (contradiction explanation)**

Let  $\langle A, R \rangle$  be the current configuration. Let us suppose that this configuration has been identified as over-constrained i.e.  $\mathcal{P}(A)$  does not hold. Let  $E \subset A$  a set of constraints.

$E$  is a **contradiction explanation** iff  $\mathcal{P}(E)$  does not hold.

Contradiction explanations are learnt from the identification of a contradiction. The function **explain-contradiction** computes such a contradiction explanation from a  $\mathcal{P}$ -contradictory configuration<sup>8</sup>.

#### 4.3.2 Promising configurations

Let  $\mathcal{E}_c$  be the set of all the contradiction explanations determined so far during the computation. We can search the  $C$ -solution of the considered problem amongst the *promising configurations*.

**Definition 4.2 (Promising configuration)**

Let  $C_f = \langle A, R \rangle$  a configuration.  $C_f$  is a **promising configuration** iff

$$\forall E \in \mathcal{E}_c, R \cap E \neq \emptyset \quad (2)$$

In other words, a **promising configuration** covers the set of contradiction explanations.

A configuration that is not promising is necessarily  $\mathcal{P}$ -contradictory because its set of active constraints contains an identified contradiction explanation.

The **best-configuration** function computes the best (regarding the given comparator  $C$ ) promising configuration from the current configuration taking into account the contradiction explanations in  $\mathcal{E}_c$ . This leads to solve a set covering problem. Note that set covering is an NP-hard problem in general. We will address this issue on section 6.

#### 4.3.3 A best first exploration

Let  $C_f$  be the current configuration. As the problem has been identified as over-constrained,  $C_f$  is  $\mathcal{P}$ -contradictory. Let  $\mathcal{E}_c$  be the set of computed contradiction explanations so far (from the beginning of the dynamic process). The main idea is to use the information gathered throughout the computation (the set  $\mathcal{E}_c$ ) to directly

<sup>7</sup>We do not use the term *nogood* (partial affectation not found in any solution) because a **contradiction explanation** is more general.

<sup>8</sup>The simplest contradiction explanation is the set  $A$  of active constraints but such an explanation is inefficient. Section 6 will precise the function for finite domains.

explore the best promising configurations. Implicit information embedded within the contradiction explanations (parts of the search tree that lead to contradiction) are thus actively used.

Let us consider the following algorithm:

**Algorithm 4.1 (Best-first search)**

```

    % We start from a promising configuration  $C_f$  that
    % became  $\mathcal{P}$ -contradictory after a modification
(1) begin
(2)   while  $C_f$  is  $\mathcal{P}$ -contradictory do
(3)     add explain-contradiction( $C_f$ ) to  $\mathcal{E}_c$ 
(4)     change-configuration( $C_f$ , best-configuration( $C_f$ ,  $\mathcal{E}_c$ ))
(5)   endwhile //  $C_f$  is a  $C$ -solution //
(6) end

```

The `change-configuration` function performs the configuration *jump* at minimal cost: the system is in the target configuration as it would have been if it had been explored first. This *jump* avoids as much as possible *thrashing*.

This algorithm presents a search method that does not require any optimality proof step. As soon as the explored configuration verifies property  $\mathcal{P}$ , it is a  $C$ -solution. Such an exploration can be characterized as a best first search.

The presented search method does not use any backtrack. Only configuration *jumps* are performed. All the exploration relies in the same level in the tree. This approach implicitly cuts branches leading to a  $\mathcal{P}$ -contradiction at each identification of a contradiction explanation. This supplementary information is used to compute promising configurations. Thus, redundant work is as much as possible avoided.

This algorithm terminates because:

- Only new contradiction explanations are produced (only new configurations will then be explored) (see theorem 6.1);
- The number of existing configurations is finite ( $2^e$  where  $e$  is the number of constraints in the problem);
- There exists at least one  $\mathcal{P}$ -satisfiable configuration: all the constraints are relaxed.

The proposed algorithm is correct because as soon as a  $\mathcal{P}$ -satisfiable configuration is obtained, it is the best possible one giving thus a  $C$ -solution.

## 4.4 Example

Let us consider the following example involving 4 constraints:  $(c_a, c_b, c_c, c_d)$ . We suppose that the constraints are introduced in the reverse order of their importance *i.e.*  $c_d$  is the most important one and  $c_a$  is the least important one. Moreover, we will use a comparator based upon a set of preferences that leads to systematically prefer the relaxation of any set of less important constraints against the relaxation of a single more important one. For example, the relaxation of constraints  $\{c_a, c_b, c_c\}$  will be preferred against the relaxation of  $\{c_d\}$ .

Let us suppose that property  $\mathcal{P}$  is verified by  $\{c_a, c_b, c_c\}$  but when adding  $c_d$   $\mathcal{P}$  does not hold. The problem is over-constrained. The exploration is depicted in figure 2.

- The current configuration  $C_f$  (path *abcd* in the tree) is  $\mathcal{P}$ -contradictory. We suppose that `explain-contradiction`( $C_f$ ) =  $(c_a, c_b, c_d)$ . Then `best-configuration`( $\mathcal{E}_c$ ) = *a'bcd*. The constraint  $c_a$ , being the least important con-

straint in the contradiction explanation, is chosen to be relaxed. The following explored configuration (path) is then  $a'bcd$ .

- We suppose that  $C_f = a'bcd$  is  $\mathcal{P}$ -contradictory and that  $\text{explain-contradiction}(C_f) = (c_b, c_d)$ . Then  $\text{best-configuration}(\mathcal{E}_c) = ab'cd$ . Relaxing  $c_a$  is no more needed because the contradiction explanation  $(c_a, c_b, c_d)$  is *covered* by the relaxation of  $c_b$ . Note that when exploring this configuration a *back*<sup>9</sup> jump is performed. Such an unusual behavior would not have been authorized by a classical backtrack-based method but thanks to the set  $\mathcal{E}_c$  it is possible here.
- Let us suppose that  $C_f = ab'cd$  is also  $\mathcal{P}$ -contradictory and that  $\text{explain--contradiction}(C_f) = (c_c, c_d)$ . Then  $\text{best-configuration}(\mathcal{E}_c) = ab'c'd$ .
- At last, let us suppose that  $C_f = ab'c'd$  is  $\mathcal{P}$ -satisfiable. There exists no better configuration regarding the user-defined comparator. This last configuration is thus a  $C$ -solution for the considered problem.

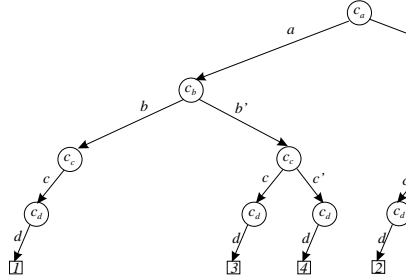


Figure 2: Best first search. Leafs are numbered in their exploration order.

## 5 An operational semantics for the exploration

In this section, we give an operational semantics for the `change-configuration` function. This semantics explicitly shows the main key-points of our approach: efficient identification of constraints to relax and the avoiding of a thrashing behavior.

### 5.1 Changing the current configuration

A configuration change is performed in three steps:

- Identification of parts of the current path (current configuration) that will not be modified in the next configuration and that are independant from the relaxed constraints.
- Effective relaxation of the selected constraint(s) and propagation of this relaxation (*cf.* section 6.5 for an implementation for finite domains),
- Reintroduction of the constraints that are relaxed in the current configuration but that are active in the next one (as shown in section 4.4, *cf.* configuration  $a'bcd$ ).

<sup>9</sup>To the left of the current position.

## 5.2 Modifying the search tree

The example depicted in figure 2 does not emphasize the reuse part of the proposed computations. We give now an operational semantics of the configuration change that enlightens the properties of our approach. This semantics is based upon transformations of the search tree.

The idea of the transformations relies on this statement: if the relaxed constraints were the last ones, a standard backtrack would have been efficient. As the order of introduction of the constraints does not modify the set of solutions (or  $C$ -solutions) for a problem, the search tree can be transformed in such a way that every constraint whose status has changed (active to relaxed, or relaxed to active) is put at the “end” of the tree (the part of computation that will be recomputed).

Thus, the operational behavior of the configuration becomes explicit and the reused parts from the previous computation can be more clearly seen.

### 5.2.1 Transformation Operators

As only a path in the tree is considered at a given time, our operators are defined considering a single path in the tree representing the current configuration.

Let  $\bullet$  be the **concatenation** operation over sub-paths:  $ab \bullet cd = abcd$ . In the following,  $x$  is any part of a path. The 3 basic transformations are:

- **Constraint Adding:** Operator  $\oplus$

The adding of the constraint  $c_c$  in the current search tree whose current configuration is path  $A$  is defined from using the operator  $\oplus$  by:

$$A \oplus c_c = A \bullet c$$

- **Constraint Relaxation (or deletion):** Operator  $\ominus$

The relaxation of constraint  $c_c$  in the current configuration is recursively defined as:

$$(x \bullet A) \ominus c_c = \begin{cases} A \bullet c' & \text{if } x = c \\ x \bullet (A \ominus c_c) & \text{otherwise} \end{cases}$$

- **Constraint Reintroduction:** Operator  $\odot$

The reintroduction of constraint  $c_c$  in the current configuration is recursively defined as:

$$(x \bullet A) \odot c_c = \begin{cases} A \bullet c & \text{if } x = c' \\ x \bullet (A \odot c_c) & \text{otherwise} \end{cases}$$

When considering a configuration change, the comparison of the current  $\mathcal{P}$ -contradictory configuration (path  $A$ ) and the target configuration (given by **best--configuration**) defines two sets: a set of constraints to relax  $c_{k_1}, \dots, c_{k_n}$  and a set of constraints to reintroduce  $c_{\ell_1}, \dots, c_{\ell_m}$ . Using the operators, the new current configuration is thus defined as:

$$A \ominus c_{k_1} \cdots \ominus c_{k_n} \odot c_{\ell_1} \cdots \odot c_{\ell_m}$$

### 5.2.2 Graphical Illustration

Let us take back the example presented section 4.4.

- From the first explored configuration  $abcd$  which is  $\mathcal{P}$ -contradictory, the function **best-configuration** gives the configuration  $a'bcd$ . There is only one constraint to relax ( $c_a$ ) in this configuration and none to reintroduce. The performed transformation is thus:

$$abcd \ominus c_a = bcda' \quad (3)$$

This new path defines a new tree for the search (*cf.* figure 3 (left)). Note that part of this new tree has been already explored (bold). Indeed, everything on the left of the current configuration leads to a  $\mathcal{P}$ -contradictory configuration. The figure describes what part of the tree has been reused without modification.

- The next configuration is  $ab'cd$ . The transformation is:

$$bcda' \ominus c_b \odot c_a = cdb'a \quad (4)$$

The resulting tree is depicted figure 3 (right).

- The next and final configuration is  $ab'c'd$ . The transformation is:

$$cdb'a \ominus c_c = db'ac' \quad (5)$$

The resulting tree is depicted figure 4. Note that the method implicitly excludes here any part of the graph that is on the left of the current configuration. This is due to the the used comparator.

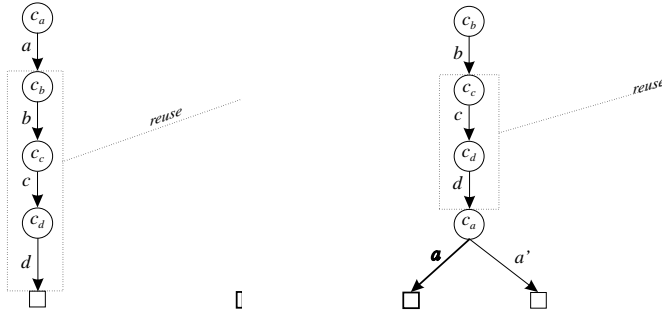


Figure 3: Transformations of the search tree – Transformation (3) (left) – Transformation (4) (right)

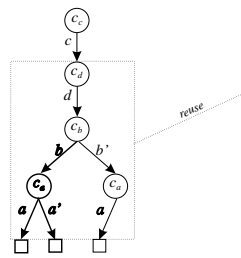


Figure 4: Transformations of the search tree – Transformation (5)

## 6 Specialization for finite domains and Arc-consistency

In this section, we present the specialization of the three functions `explain-contradiction`, `change-configuration` and `best-configuration` in order to handle CSP (Constraint Satisfaction Problems) *i.e.* Finite Domains problems.

Let us recall that a CSP can be defined as a set  $\mathcal{V}$  of variables; the set  $\mathcal{D}$  of their respective domains (discrete finite set of the possible values for each variable) and the set  $\mathcal{C}$  of constraints specifying acceptable combinations of values for the variables of the problem.

## 6.1 The best-configuration function

As precised in definition 4.2, the new best promising configuration must *cover* the set  $\mathcal{E}_c$  of *contradiction explanations* whilst optimizing the comparator-based partial order.

This problem can be modeled as the determination of a set covering in the hypergraph  $\mathcal{H}$ :

### Definition 6.1 (The Hypergraph $\mathcal{H}$ )

The hypergraph  $\mathcal{H}$  is defined from the set  $\mathcal{E}_c$  of contradiction explanations. Each constraint appearing in any element of  $\mathcal{E}_c$  is a vertex in  $\mathcal{H}$ . Each contradiction explanation  $E$  is an hyper-edge in  $\mathcal{H}$ .

The general set covering problem in an hypergraph is NP-complete. When using the  $C_{MM}$  comparator (*cf.* definition 2.1) this problem becomes polynomial. Indeed, the best promising configuration  $\langle A', R' \rangle$  regarding  $C_{MM}$  computed from a  $\mathcal{P}$ -contradictory configuration is:

$$R' = \{c \in A \cup R \mid \exists E \in \mathcal{E}_c, c = \min_{\text{pref}}\{c_i \in E\}\}$$

$$\text{and } A' = (A \cup R) \setminus R'$$

Note that using this comparator, best promising configurations can be computed in an incremental and efficient way ( $O(e)$  – number of constraints) by simply adding a constraint to relax (the least important of the new *contradiction explanation*). We call such a comparator a **contradiction-local** one. A contradiction-local comparator is interesting for the complexity results (*cf.* section 7) that can therefore be obtained. It is also of great use when all the preferences associated with constraints are not known; they can be specified only when required.

## 6.2 A Deduction Maintenance System

**Arc-consistency** is the property maintained for CSP using domain reduction. We suggest here to *record* such domain reductions (more precisely their *explanation*) in order to provide identification of responsibilities for a contradiction (function **explain-contradiction**). Our recording system is called a **deduction maintenance system** (DMS).

A **deduction** is associated with any of the three following actions:

- Removing a value from the domain of a variable. The deduction associated with the removal of the value  $a$  from the domain  $d_x$  of the variable  $x$  is denoted:  $\delta_{(x \neq a)}$ .
- Removing a constraint from the constraint store (relaxing). The deduction associated with the relaxation of the constraint  $c$  is denoted:  $\delta_{(\bar{c})}$ .
- Raising a contradiction (the domain of a variable becomes empty). The associated deduction is denoted  $\delta_{(\perp)}$ .

Let  $\Delta$  be the set of deductions performed during the resolution.

A **deduction explanation**  $E$ , for a deduction  $\delta$ , is a set of constraints whose conjunction leads to perform the action associated with the deduction.

**Definition 6.2 (Deduction explanation)**

Let  $E$  be a set of constraints.

- $E$  is a **deduction explanation** for a deduction  $\delta_{(x \neq a)}$  iff  $a$  is removed from  $d_x$  when achieving  $\mathcal{P}$  for the set of constraints  $E$ .
- $E$  is a **deduction explanation** for a deduction  $\delta_{(\perp)}$  iff the configuration  $\langle E, \emptyset \rangle$  is  $\mathcal{P}$ -contradictory.
- $E$  is a **deduction explanation** for a deduction  $\delta_{(\bar{c})}$  iff the configuration  $\langle \{c\} \cup E, \emptyset \rangle$  is  $\mathcal{P}$ -contradictory.

A deduction explanation  $E$  is **valid** in a given configuration  $\langle A, R \rangle$  if  $E \subset A$ .

A configuration becomes  $\mathcal{P}$ -contradictory as soon as the domain of a variable becomes empty, *i.e.* all the deduction explanations of the removal of each value in the domain are valid.

### 6.3 Providing deduction explanations

The simplest *deduction explanation* for any deduction is the complete set of the active constraints of the current configuration. This kind of *deduction explanation* is obviously of no use and would lead to a complete enumeration process of all the possible configurations (that we try to avoid).

The *best deduction explanation* is the minimal set of constraints that verifies the definition 6.2. When using a particular algorithm during property  $\mathcal{P}$  enforcement, a *good deduction explanation* must reflect the knowledge used by this algorithm to perform any deduction. We detail possible *deduction explanation* when achieving arc-consistency.

**Using the AC4 algorithm:** The AC4 algorithm [13] uses two main steps to remove unsupported values from the domain of the variables. The first step performs value removals directly due to a unique constraint (thus, the *deduction explanation* is built from this constraint). In the second step, indirect removals are performed, they are the consequence of the removals in the first step (the *deduction explanation* is then the applied constraint associated with the *deduction explanation* of the propagated removal).

**Using the AC5 algorithm:** The AC5 algorithm [18] actively uses the semantics of the handled constraints. This leads to a better *deduction explanation* system. For example, when filtering a constraint  $x > y$  considering that the current domain of  $x$  is  $\{\dots, a\}$ , values greater than  $a$  will be removed from  $d_y$  and the *deduction explanation* for these removals is  $E = \bigcup_{b \in d_x, b > a} E_{\delta_{x \neq b}}$  where  $E_\delta$  represents an explanation for deduction  $\delta$ .

### 6.4 Function explain-contradiction

When a contradiction (deduction  $\delta_{(\perp)}$ ) occurs, the domain of at least one variable is empty. Let  $x$  be such a variable. Let note  $E_\delta$  the *deduction explanation* associated with the deduction  $\delta$ . Thus  $E_{\delta_{(\perp)}} = \bigcup_{a \in d_x} E_{\delta_{(x \neq a)}}$ .

**Theorem 6.1 (New contradiction explanation)**

When computing a new contradiction explanation for a (previously promising)  $\mathcal{P}$ -contradictory configuration, only a new contradiction explanation is produced *i.e.*  $E_{\delta_{(\perp)}} \not\subset \mathcal{E}_c$ .

**Proof:** As  $E_{\delta_{(\perp)}}$  is valid, it cannot be an already computed *contradiction explanation* because the current configuration would not have been a promising one (not covering the current set of *contradiction explanations*).  $\square$

## 6.5 Function change-configuration

When performing a configuration change, we compute the set  $C_\alpha$  of new active constraints and the set  $C_\beta$  of the new relaxed constraints.

**Relaxing a constraint** The relaxation of any element of  $C_\beta$  needs to be propagated. The aim of this propagation is to delete the past effects of the relaxed constraints. We present here an extension of DNAC4 (which performs incremental constraint deletions for dynamic CSP) [1].

First of all, a *deduction explanation* can be added for the removal of the constraint  $c_c$ . It consists in the invalidated *contradiction explanation* in  $\mathcal{E}_c$  (except  $c$ ).

Let  $c$  be an element of  $C_\beta$ . Let  $\alpha(c)$  be the subset of  $\Delta$  (the deductions of the problem) whose validity relies upon  $c$ .  $\alpha(c) = \{\delta \in \Delta \mid c \in E_\delta\}$ . Our extension of DNAC4 proceeds in two steps:

- in the first step, values considered in  $\alpha(c)$  are put back in their respective domain,
- in the second step, each of these values are considered through arc-consistency achievement to see if they can be removed with a new explanation. If it is possible, the associated value removal is propagated as usual.

**Completing the jump** In order to terminate the jump, every constraint in  $C_\alpha$  needs to be introduced in the constraint system. Note that when using our comparator ( $C_{MM}$ ) no constraint in  $C_\alpha$  needs to be reintroduced.

## 6.6 Enumeration and Relaxation

Enumeration can be modeled as the dynamic addition/removal of equality constraints (eg.  $x = a$  for  $a \in d_x$ ). This is completely transparent in our approach. After each addition of such a constraint, the property  $\mathcal{P}$  is enforced and if the current configuration is  $\mathcal{P}$ -contradictory then the constraint relaxation process is initiated by handling enumeration constraints in the same way as the other ones. The preference associated to such constraints is the lowest possible; so in case of failure, enumeration constraints will be relaxed in priority.

In fact, our algorithm is extended in order to enforce the introduction of another constraint (testing another value) when relaxing an equality constraint due to the enumeration. This leads to model an exclusive disjunction between equality constraints.

During this process, informations (namely the explanations associated with constraint removals) are kept as in *Dynamic Backtracking* [7] in order to ensure the completeness of the approach.

## 7 Complexity issues

Let  $n$  be the number of variables,  $e$  the number of constraints and  $d$  the maximum size of the domains. There are  $c = e + n \times d$  possible constraints in the problem (we add the equality constraints produced by the enumeration process). Note that there are only  $c' = e + n$  active constraints at a given time of computation since a variable can only be assigned to a single value.

The good complexity results presented here are due to the properties of contradiction-local comparators (such as  $C_{MM}$  – cf. section 6.1).

## 7.1 Space Complexity

The space occupation is related to the *deduction explanations* recording. Thus, the overall space complexity is defined as the number of possible deductions multiplied by the worst-case size of an explanation.

A *deduction explanation* can contain at most  $e+n$  constraints since it is a subset of the active constraints. There are  $O(a) = O(n \times d + e + 1)$  possible deductions in the system:

- one for each value removal *i.e.*  $n \times d$  in the worst case;
- one for each constraint removal *i.e.*  $e$  real constraints plus  $n \times d$  enumeration-related constraints;
- one for the last contradiction (when using comparator  $C_{MM}$ , only the last contradiction need to be kept).

Hence, in the worst case, we encounter  $O(n \times d + e)$  *deduction explanations*.

This leads to the following worst case space complexity:

$$O(c' \times a) = O((n + e) \times (n \times d + e))$$

## 7.2 Time complexity

In the worst-case, the overall complexity of the search is obviously exponential since solving a CSP is NP-complete. As usual, we will give the time complexity of the basic steps of our approach:

- the arc-consistency management is dependent from the used arc-consistency algorithm used augmented with the providing of explanations. For example, when using an AC4-based algorithm, explanations are computed in  $O(d)$  (from the supporting values). Hence, the resulting complexity of the algorithm is in  $O(e \times d^2 \times d)$  because AC4 is in  $O(e \times d^2)$ ;
- computing a contradiction explanation can be computed in  $O(d)$  as explained before;
- determining a constraint to relax is done in  $O(e+n)$ : the size of an explanation;
- performing the constraint removal is achieved in  $O(e \times d^2 \times d)$  since DNAC4 gives the same complexity for constraint addition or removal.

## 7.3 Experimental results

DECORUM (Deduction-based Constraint Relaxation Management) is an implementation (done in C++) which provides a constraint solver for over-constrained CSPs. The efficient complexities enabled by our Deduction Maintenance System allow the handling of large CSPs: hundred of variables for thousand of constraints.

We give here experimental results on large random problems since the main worry when dealing with a Deduction Maintenance System is practability. We will focus on single examples. Random problems are classically categorized using 4 parameters : the number of variables, the uniform size of their domain, the density of the constraint graph and the tightness<sup>10</sup> of the constraints.

Here are some results that give a hint of how our system behaves for large problems<sup>11</sup>:

---

<sup>10</sup>Ratio between the number of allowed combinations of value and the total number of such combinations ( $d \times d$  for binary constraints).

<sup>11</sup>The problems considered here are very over-constrained because of the high value of the tightness of constraints. This explains the large number of relaxed constraints.

- a  $\langle 700, 2, 0.05, 0.5 \rangle$  which involves 12500 constraints is solved in 27s. This boolean satisfaction problem (binary domains) is solved using 211320 constraint checks and a space occupation of 17368.<sup>12</sup> 5967 constraints need to be relaxed. The enumeration step is done without *backtrack*<sup>13</sup>.
- a  $\langle 120, 5, 0.32, 0.6 \rangle$  which thus involves 2300 constraints is solved in 51s with 172070 constraint checks and using a 49581 space. The search requires 1240 *backtracks* and 1297 constraint relaxations.
- a  $\langle 50, 10, 0.49, 0.75 \rangle$  which involves 600 constraints is solved in 260s using 927587 constraint checks and using a 31229 space. The search requires 1980 *backtracks* and 362 constraint relaxations.
- a  $\langle 50, 25, 0.20, 0.32 \rangle$  which involves 200 constraints is solved in 113s using 615117 constraint checks and using a 23547 space. The search requires 16 *backtracks* and no constraint is relaxed.

These results simply show that large problems can be dealt with using DECORUM. Our various experiments [10] show that the behavior of DECORUM on a large set of random problems conforms to the well known phase transition [15] despite the relaxation process. Other experiments [9] show that the search provided by DECORUM is of much interest on *structured* problems (with set of relatively independent variables, ...).

## 8 Conclusion

We have proposed a best first search on the configurations space that is well suited for dynamic problems. This search method is parameterized by the *domain* on which constraints are defined and by the *property* maintained during the computation. This method has been instantiated for finite domains and the property of arc-consistency. Algorithms, a complexity analysis and an implementation have shown the efficiency of the approach. Our current works lie on two topics:

- the use of DECORUM to solve real-life dynamic resource allocation problems for military purposes and for communication networks;
- the instantiation of our search method maintaining the property of B-consistency [11] for intervals also processed by domain reduction; therefore, the proposed approach (including the DMS) seems easily adaptable to intervals.

## References

- [1] Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.
- [2] Stefano Bistarelli, Hélène Fargier, Ugo Montanari, Francesca Rossi, Thomas Schiex, and Gérard Verfaillie. Semiring-based cps and valued cps: basic properties and comparison. In Michael Jampel, Eugene Freuder, and Michael Maher, editors, *Over-Constrained Systems*, number 1106 in Lecture Notes in Computer Science, pages 111–150. Springer Verlag, 1996.
- [3] Alan Borning, Richard Anderson, and Bjorn Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *ACM Symposium on User Interface Software and Technology*, pages 129–136, 1996.

<sup>12</sup>The space occupation is represented by the number of constraints references in all the explanations of the system.

<sup>13</sup>Since no *real* backtrack occur in DECORUM, we call backtrack each constraint relaxation related to enumeration.

- [4] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.
- [5] François Fages, Julian Fowler, and Thierry Sola. A reactive constraint logic programming scheme. In *International Conference of Logic Programming, ICLP'95*, Tokyo, 1995.
- [6] Eugene C. Freuder and Richard Wallace. Partial constraint satisfaction. In Michael Jampel, Eugene Freuder, and Michael Maher, editors, *Over-Constrained Systems*, number 1106 in Lecture Notes in Computer Science, pages 63–110. Springer Verlag, 1996.
- [7] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
- [8] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume F13 of *NATO ASI Series*. Springer Verlag, January 1985.
- [9] Narendra Jussien. *Relaxation de Contraintes pour les problèmes dynamiques*. PhD thesis, Université de Rennes I, 24 October 1997.
- [10] Narendra Jussien and Patrice Boizumault. A best first approach for solving over-constrained dynamic problems. In *IJCAI'97*, Nagoya, Japan, August 1997. (poster – RR 97-6-INFO – École des Mines de Nantes).
- [11] Olivier Lhomme, Arnaud Gotlieb, Michel Rueher, and Patrick Taillibert. Boosting the interval narrowing algorithm. In *JICSLP'96*, Bonn, Germany, 2–6 September 1996.
- [12] Francisco Menezes and Pedro Barahona. Defeasible constraint solving. In Michael Jampel, Eugene Freuder, and Michael Maher, editors, *Over-Constrained Systems*, number 1106 in Lecture Notes in Computer Science, pages 151–170. Springer Verlag, 1996.
- [13] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
- [14] Bertrand Neveu and Pierre Berlandier. Arc-consistency for dynamic constraint satisfaction problems: an RMS free approach. In *Proc. ECAI-94, Workshop on Constraint satisfaction issues raised by practical applications*, Amsterdam, The Netherlands, 1994.
- [15] Patrick Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. Technical Report AISL-49-94, Department of Computer Science, University of Strathclyde, Glasgow, Scotland, 1994.
- [16] Michael Sannella. The SkyBlue constraint solver and its applications. In Paris Kanelakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Providence RI, 1993.
- [17] Thomas Schiex. Possibilistic constraint satisfaction problems or “How to handle soft constraints?”. In *8th International Conference on Uncertainty in Artificial Intelligence*, Stanford, July 1992.
- [18] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, October 1992.
- [19] Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM transactions on Programming Languages and Systems*, 18(1):30–72, January 1996.
- [20] Molly Wilson and Alan Borning. Hierarchical constraint logic programming. *Journal of Logic Programming*, 16(3):277–318, July 1993.