Artificial Intelligence 241 (2016) 153–169

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Computing and restoring global inverse consistency in interactive constraint satisfaction ☆

Christian Bessiere [a], Hélène Fargier [b], Christophe Lecoutre [c,*]

[a] LIRMM-CNRS, University of Montpellier, France
[b] IRIT-CNRS, University of Toulouse, France
[c] CRIL-CNRS, University of Artois, France

ARTICLE INFO

Article history:
Received 20 January 2015
Received in revised form 1 June 2016
Accepted 7 September 2016
Available online 14 September 2016

Keywords:
Constraint satisfaction problems
Configuration
Global inverse consistency

ABSTRACT

Some applications require the interactive resolution of a constraint problem by a human user. In such cases, it is highly desirable that the person who interactively solves the problem is not given the choice to select values that do not lead to solutions. We call this property *global inverse consistency*. Existing systems simulate this either by maintaining arc consistency after each assignment performed by the user or by compiling offline the problem as a multi-valued decision diagram. In this article, we define several questions related to global inverse consistency and analyze their complexity. Despite their theoretical intractability, we propose several algorithms for enforcing and restoring global inverse consistency and we show that the best version is efficient enough to be used in an interactive setting on several configuration and design problems.

1. Introduction

Constraint Programming (CP) is widely used to express and solve combinatorial problems. Once a problem is modeled as a constraint network, efficient solving techniques generate a solution satisfying the constraints, if such a solution exists. However, there are situations where the user has strong opinions about the way to build good solutions to the problem but some of the desirable/undesirable combinations will become clear only once some of the variables are assigned. In this case, the constraint solver should be there to assist the user in the solution design and to ensure her choices remain in the feasible space, removing the combinatorial complexity from her shoulders. See the Synthia system for protein design as an early example of using CP to interactively solve a problem [2]. Another well known example of such an interactive solving of constraint-based models is product configuration [3,4]. The person modeling the product as a constraint network for the company knows its technical and marketing requirements. She models the feasibility, availability and/or marketing constraints about the product. This constraint network captures the catalog of possible products, which may contain billions of solutions, but in an intentional and compact way. Nevertheless, the modeler does not know the constraints or preferences of the customer(s). This is the customer who will look for solutions, with her own constraints and preferences on the price, the color, or any other configurable feature.

☆ This paper is an invited revision of a paper which first appeared at the 18th International Conference on Principles and Practice of Constraint Programming (CP 2013) [1]. This article additionally contains a new section on restoring global inverse consistency after the retraction of a decision from the user. It also contains additional experiments.

* Corresponding author.
E-mail addresses: bessiere@lirmm.fr (C. Bessiere), fargier@irit.fr (H. Fargier), lecoutre@cril.fr (C. Lecoutre).

http://dx.doi.org/10.1016/j.artint.2016.09.001
0004-3702/

These applications refer to an interactive solving process where the user selects values for variables according to her own preferences and the system checks the constraints of the network, until all variables are assigned and satisfy all constraints of the network. This solving policy raises an important issue: the person who interactively solves the problem should not be led to a dead-end where satisfying all constraints of the network is impossible. Existing interactive solving systems address this issue either by compiling the constraint network into a multi-valued decision diagram (MDD) at the modeling phase [4–6] or by enforcing arc consistency on the network after each assignment performed by the user [2]. Compiling the constraint network as an MDD can require a significant amount of time and space. That is why compilation is performed offline (before the solving session). As a consequence, configurators based on an MDD compilation are restricted to static constraint networks: non-unary constraints can neither be added nor removed once the network is compiled. It is thus not possible for the user to perform complex requirements, e.g. she is interested in traveling to Venezia only during the carnival period. Arc and dynamic arc consistencies require a lighter computational effort but the user can be trapped in dead-ends, which is very risky from a commercial point of view. It has been shown in [7] that arc consistency (and even higher levels of local consistency) can be very bad approximations of the ideal state where all values remaining in the network can be extended to solutions.

The message of our article is that for many of the problems that require interactive solving of the problem, and especially for real problems, it is computationally feasible to maintain the domains of the variables in a state where they only contain those values which belong to a complete solution extending the current choices of the user. Inspired by the nomenclature used in [8] and [9], we call this level of consistency *global inverse consistency* (GIC).

Our contribution addresses several aspects. First, we formally characterize the questions that underlie the interactive constraint solving loop and we show that they are all NP-hard. Second, we provide several algorithms with increasing sophistication to address those tasks. Third, we experimentally show that the most efficient of our algorithms is efficient enough to be used in an interactive constraint solving loop of several non-trivial configuration and design problems.

2. Background

A (discrete) constraint network (CN) N is composed of a finite set of n variables, denoted by $vars(N)$, and a finite set of e constraints, denoted by $cons(N)$. Each variable x has a domain which is the finite set of values that can be assigned to x. The initial domain of a variable x is denoted by $dom^{init}(x)$ whereas the current domain of x is denoted by $dom(x)$; we always have $dom(x) \subseteq dom^{init}(x)$. Sometimes, we use $dom^N(x)$ to denote the domain of x in the context of the CN N. The maximum domain size of a variable in a given CN is denoted by d. To simplify, a variable–value pair (x, a) such that $x \in vars(N)$ and $a \in dom^N(x)$ is called a value of N; we note $\texttt{values}(N) = \{(x, a) \mid x \in vars(N) \wedge a \in dom^N(x)\}$. Each constraint c involves an ordered set of variables, called the *scope* of c and denoted by $scp(c)$, and is semantically defined by a relation, denoted by $rel(c)$, which contains the set of tuples allowed for the variables involved in c. The *arity* of a constraint c is the size of $scp(c)$, and will usually be denoted by r.

An *instantiation* I of a set $X = \{x_1, \ldots, x_k\}$ of variables is a set $\{(x_1, a_1), \ldots, (x_k, a_k)\}$ such that $\forall i \in 1..k, a_i \in dom^{init}(x_i)$; X is denoted by $vars(I)$ and each a_i is denoted by $I[x_i]$. An instantiation I *on* a CN N is an instantiation of a set $X \subseteq vars(N)$; it is *complete* if $vars(I) = vars(N)$. I is *valid* on N iff $\forall (x, a) \in I, a \in dom(x)$. I *covers* a constraint c iff $scp(c) \subseteq vars(I)$, and I *satisfies* a constraint c with $scp(c) = \{x_1, \ldots, x_r\}$ iff (i) I covers c and (ii) the tuple $(I[x_1], \ldots, I[x_r]) \in rel(c)$. An instantiation I on a CN N is *locally consistent* iff (i) I is valid on N and (ii) every constraint of N covered by I is satisfied by I. A *solution* of N is a complete locally consistent instantiation on N; $sols(N)$ denotes the set of solutions of N. A CN N is *satisfiable* iff $sols(N) \neq \emptyset$.

The ubiquitous example of constraint propagation is enforcement of *generalized arc consistency* (GAC) which removes values from domains without reducing the set of solutions of the constraint network. A value (x, a) of a CN N is GAC on N iff for every constraint c of N involving x, there exists a valid instantiation I of $scp(c)$ such that I satisfies c and $I[x] = a$. N is GAC iff every value of N is GAC. Enforcing GAC means removing GAC-inconsistent values from domains until the constraint network is GAC. In this article, we shall refer to MAC which is an algorithm considered to be among the most efficient generic approaches for the solution of CNs. MAC [10] explores the search space depth-first, enforces (generalized) arc consistency after each decision taken (variable assignment or value refutation) during search, and backtracks when failures happen. A *past* variable is a variable explicitly assigned by the search algorithm whereas a *future* variable is a variable not (explicitly) assigned. The set of future variables of a CN N is denoted by $vars^{fut}(N)$.

3. Problems raised by interactive constraint solving

In this section, we formally characterize the questions that underlie the interactive constraint solving loop and we study their theoretical complexity.

3.1. Formalization

We first define the level of local consistency that is desirable in any interactive solving loop involving a human, that is, the level of consistency that guarantees that all values in all domains belong to a solution. In the nomenclature introduced by Freuder in [11] it corresponds to $(1, n-1)$-consistency if the constraint network contains n variables. To avoid the

reference to n, Freuder has called it *variable completability* in [12], and Dechter has called it *global consistency of values* in [13]. To be consistent with the now widely accepted nomenclature introduced in [8], we decided to call it *global inverse consistency*.

Definition 1 *(Global Inverse Consistency).* A value (x, a) of a CN N is *globally inverse consistent (GIC)* iff $\exists I \in sols(N) \mid I[x] = a$. A CN N is GIC iff every value in $\texttt{values}(N)$ is GIC. The *GIC closure* of a CN N is the CN obtained from N by removing all the values that do not belong to a solution of N.

We can observe that, as usual for levels of consistency, the GIC closure of a constraint network has the same set of solutions as the original network.

There is a close relationship between GIC and minimality of constraint networks as defined by Montanari [14]. A constraint network is minimal according to Montanari if and only if any locally consistent instantiation of length 2 can be extended to a solution. Minimal networks are only defined for *binary* networks, that is, when the arity of all constraints is 2. Despite similarities in the definitions of GIC and minimal networks—they are both based on the notion of extensibility to solutions, a binary constraint network can be GIC and not minimal or minimal and not GIC. Take for instance the constraint network with $dom(x_1) = dom(x_2) = \{1, 2, 3\}$ and the single constraint $x_1 \cdot x_2 = 3$. It is obviously minimal, but it is not GIC because 2 does not belong to any solution. Take now the constraint network with $dom(x_1) = dom(x_2) = dom(x_3) = \{1, 2, 3\}$ and the constraints $x_1 + x_2 = 4$, $x_2 + x_3 = 4$, and $|x_1 - x_3| \leq 1$. It is GIC, but it is not minimal because the tuple $((x_1, 2), (x_3, 1))$ is accepted by the constraint $|x_1 - x_3| \leq 1$ and does not extend to any solution.

The obvious problems that follow from the definition of GIC are to check whether a constraint network is GIC or not, and to enforce GIC.

Problem 1 *(Deciding GIC).* Given a CN N, is N GIC?

Problem 2 *(Computing GIC).* Given a CN N, compute the GIC closure of N.

As we are interested in interactive solving, we define the problem of maintaining GIC after the user has performed a variable assignment or, more generally, added any arbitrary constraint.

Problem 3 *(Maintaining GIC).* Given a CN N that is GIC, and a constraint c_{new} with $scp(c_{new}) \subseteq vars(N)$, recompute GIC after the addition of c_{new} to $cons(N)$.

We also define the problem of restoring GIC after the user has decided to discard an existing constraint.

Problem 4 *(Restoring GIC).* Given a CN N and its GIC closure N_{GIC}, given a constraint $c \in cons(N)$, recompute the GIC closure of N after the retraction of c from $cons(N)$.

In a configuration setting, as soon as some mandatory variables have been set, the user can ask for an automatic completion of the remaining variables. Hence the definition of the following problem:

Problem 5 *(Solving a GIC network).* Given a CN N that is GIC, find a solution to N.

3.2. Complexity results

Not surprisingly, the basic questions related to GIC (Problems 1 and 2) are intractable.

Theorem 1 *(Problem 1). Deciding whether a constraint network N is GIC is NP-complete, even if N is satisfiable.*

Proof. We first prove membership to NP. For each value (x, a) of N, it is sufficient to provide a solution I of N such that the projection $I[x]$ of I on variable x is equal to a. This certificate has size $n \cdot n \cdot d$ and can be checked in polynomial time.

Completeness for NP is proved by reducing 3Col[1] to the problem of deciding whether a satisfiable CN is GIC. Take any instance of the 3Col problem, that is, a graph $G = (V, E)$, and denote the three colors by $1, 2, 3$. Consider the CN N where $vars(N) = \{x_i \mid i \in V\}$, $dom(x_i) = \{0, 1, 2, 3\}, \forall i \in V$, and $cons(N) = \{(x_i \neq x_j) \vee (x_i = 0 \wedge x_j = 0) \mid (i, j) \in E\}$. Clearly the assignment $\{(x_i, 0) \mid i \in V\}$ is a solution of N, so N is satisfiable. If N is GIC then G is 3-colorable because, by construction, N has solutions other than $\{(x_i, 0) \mid i \in V\}$ iff G is 3-colorable. If G is 3-colorable then for any variable x_i, there exists a

[1] The graph 3-colorability problem (3Col) aims at deciding whether the vertices of a graph can be colored by using three colors such that no edge links two vertices having the same color.

solution of N in which x_i is assigned a value in $1, 2, 3$. By swapping 1 with 2, 2 with 3 and 3 with 1 on all variables of this solution, we still have a solution. Similarly by swapping 1 with 3, 2 with 1 and 3 with 2 on all variables. Thus, if G is 3-colorable then N is GIC. Therefore, N is GIC iff G is 3-colorable. □

Our proof shows that hardness for deciding GIC holds for binary CNs (i.e., CNs only involving binary constraints). We have another proof, inspired from that used in Theorem 3 in [15], that shows that deciding GIC is still hard for Boolean domains and quaternary constraints.

Theorem 2 *(Problem 2). Computing the GIC closure of a constraint network N is NP-hard and NP-easy, even if N is satisfiable.*

Proof. We prove NP-easiness by showing that a polynomial number of calls to a NP oracle are sufficient to build the GIC closure of N. For each value (x, a) of N, we ask the NP oracle whether N with the extra constraint $x = a$ is satisfiable (we call this an *inverse check*). Once all values have been tested, we build the GIC closure of N by removing from each $dom(x)$ all values a for which the oracle test returned 'no'. Hardness is a direct corollary of Theorem 1. □

Notice that the two previous intractability results are still valid when the CN is satisfiable, as is the case at the beginning of an interactive resolution session.

We finally show that Problems 3, 4 and 5 are unfortunately not easier than checking GIC or enforcing GIC from scratch. But they are not harder.

Theorem 3 *(Problem 3). Given a CN N that is GIC, and a constraint c_{new} with $scp(c_{new}) \subseteq vars(N)$, computing the GIC closure of the CN N', where $vars(N') = vars(N)$ and $cons(N') = cons(N) \cup \{c_{new}\}$ is NP-hard and NP-easy even if c_{new} is simply a variable assignment $y = b$.*

Proof. NP-easiness is proved as in the proof of Theorem 2 by showing that a polynomial number of calls to a NP oracle are sufficient to build the GIC closure of N'. For each value (x, a) of N we ask the NP oracle whether N' with the extra constraint $x = a$ is satisfiable. Once all values have been tested, we build the closure of N' by removing from $dom(y)$ all values $a \neq b$ and removing from each $dom(x)$ all values a for which the oracle test returned 'no'.

We prove hardness by reducing 3Col to the problem of computing the GIC closure of a GIC network after a variable assignment. Take any instance of the 3Col problem, that is, a graph $G = (V, E)$, and denote the three colors by $1, 2, 3$. Consider the constraint network N where $vars(N) = \{x_i \mid i \in V\} \cup \{y_1, y_2\}$, $dom(x_i) = \{0, 1, 2, 3\}, \forall i$, $dom(y_1) = dom(y_2) = \{0, 1\}$. $cons(N) = \{c(x_i, x_j) \mid (i, j) \in E\} \cup \{c'(x_i, y_1, y_2) \mid i \in 1..n\}$, with $c(x_i, x_j)$ satisfied iff $(x_i \neq x_j \vee x_i = x_j = 0)$ and $c'(x_i, y_1, y_2)$ satisfied iff at most two variables among x_i, y_1 and y_2 are assigned 0. N is GIC because for any $i \in 1..n$, the instantiation where x_i is assigned any of its values, $x_j = 0, \forall j \neq i$, and y_1 and y_2 take 0 and 1 or vice versa, is a solution.

We reduce the 3Col problem to the problem of enforcing GIC on the network N' with $vars(N') = vars(N)$ and $cons(N') = cons(N) \cup \{y_1 = 0\}$. Let dom_{GIC} be the domain of the GIC closure of N'. If G is 3-colorable then assigning all x_i's to a solution of the 3-coloring and $y_2 = 0$ is a solution of N'. If $0 \in dom_{GIC}(y_2)$, then $y_2 = 0$ must belong to a solution, say s. x_i's cannot take value 0 in s otherwise constraints $c'(x_i, y_1, y_2)$ would be violated. Thus, the restriction of s to x_i's variables is a 3-coloring. Therefore, G is 3-colorable iff $0 \in dom_{GIC}(y_2)$. □

Theorem 4 *(Problem 4). Given a CN N, its GIC closure N_{GIC}, and a constraint $c_{old} \in cons(N)$, computing the GIC closure of the CN N', where $vars(N') = vars(N)$ and $cons(N') = cons(N) \setminus \{c_{old}\}$ is NP-hard and NP-easy even if c_{old} is simply a variable assignment $y = b$.*

Proof. NP-easiness is proved as in the proof of Theorem 2. We prove hardness by reducing 3Col to the problem of taking a network N for which we know the GIC closure N_{GIC}, and computing the GIC closure of the network N' obtained from N by retracting a variable assignment. Take any instance of the 3Col problem, that is, a graph $G = (V, E)$. Consider the constraint network N where $vars(N) = \{x_i \mid i \in V\} \cup \{y\}$, $dom(x_i) = \{0, 1, 2, 3\}, \forall i$, $dom(y) = \{0, 1\}$. $cons(N) = \{c_1(x_i, x_j) \mid (i, j) \in E\} \cup \{c_2(x_i, y) \mid i \in 1..n\} \cup \{y = 0\}$, with $c_1(x_i, x_j)$ satisfied iff $(x_i \neq x_j) \vee (x_i = x_j = 0)$ and $c_2(x_i, y)$ satisfied iff $(x_i = y = 0) \vee (x_i \neq 0 \wedge y \neq 0)$. The only solution is the assignment where every x_i is assigned 0 and y is assigned 0. Thus, N_{GIC} has the domain dom_{GIC} defined by $dom_{GIC}(x_i) = \{0\}, \forall i$, and $dom_{GIC}(y) = \{0\}$.

We show that the 3Col problem can be decided polynomially if we have an oracle enforcing GIC on the network N' with $vars(N') = vars(N)$ and $cons(N') = cons(N) \setminus \{y = 0\}$. Let dom'_{GIC} be the domain of the GIC closure of N'. If G is not 3-colorable then, by construction of c_1 and c_2, the only solution is the same as in N, that is, the tuple containing only 0's. If G is 3-colorable then assigning a solution of the 3-coloring to the x_i's and 1 to y is a solution of N'. Therefore, knowing N_{GIC} does not help and G is 3-colorable iff $1 \in dom_{GIC}(y)$. □

Theorem 5 *(Problem 5). Generating a solution to a GIC constraint network cannot be done in polynomial time, unless $P = NP$.*

Proof. The following proof is derived from [16]. It is also a corollary of the recent and more complex Theorem 3.1 in [17].

Algorithm 1: GIC1(N: CN).

```
1 foreach variable x ∈ vars^fut(N) do
2     foreach value a ∈ dom(x) do
3         I ← searchSolutionFor(N|x=a)
4         if I = nil then
5             remove a from dom(x)
```

Algorithm 2: handleSolution2/3(x: variable, I: instantiation).

```
1 foreach variable y ∈ vars^fut(N) such that y is revised after x do
2     if stamp[y][I[y]] ≠ time then
3         stamp[y][I[y]] ← time
4         nbGic[y]++
```

Algorithm 3: isValid(X : set of variables, I : instantiation): Boolean.

```
1 foreach variable x ∈ X do
2     if I[x] ∉ dom(x) then
3         return false
4 return true
```

Suppose we have an algorithm A that generates a solution to a GIC constraint network N in time bounded by a polynomial $p(|N|)$. Take any instance of the 3Col problem, that is, a graph $G = (V, E)$. Consider the CN N where $vars(N) = \{x_i \mid i \in V\}$, $dom(x_i) = \{1, 2, 3\}, \forall i \in V$, and $cons(N) = \{x_i \neq x_j \mid (i, j) \in E\}$. N has a solution iff G is 3-colorable. Now, if G is 3-colorable then N is GIC because any permutation of the three colors applied to all variables remains a solution (as in the proof of Theorem 1). Thus, it is sufficient to run A during $p(|N|)$ steps. If it returns a solution to N, then the 3Col instance is satisfiable. Otherwise, the 3Col instance is unsatisfiable. Therefore, as 3Col is NP-complete, there cannot exist a polynomial algorithm for generating a solution to a GIC constraint network, unless $P = NP$. □

4. Algorithms for enforcing/maintaining GIC

In this section, we introduce four algorithms to enforce global inverse consistency. These GIC algorithms use increasingly sophisticated data structures and techniques that have recently proved their worth in propagation algorithms proposed in the literature. To simplify our presentation, we assume that the CNs are satisfiable, which is the case in interactive resolution, allowing us to avoid handling domain wipe-outs in the GIC procedures. Note that these algorithms can be used to enforce GIC, but also to maintain it during a user-driven search. This is why we refer to the set $vars^{fut}(N)$ of future variables in some instructions.

The first algorithm, GIC1, is similar to an algorithm proposed in [18]. GIC1 is described in Algorithm 1. It is really basic: it will be used as our baseline during our experiments. For each value a in the domain of a future variable x, a solution for the CN N where x is assigned the value a, denoted by $N|_{x=a}$, is sought using a complete search algorithm. This search algorithm, called here searchSolutionFor, either returns the first solution that can be found, or the special value *nil*. Our implementation choice will be the algorithm MAC that maintains (G)AC during a backtrack search [10]. Hence, in Algorithm 1, when it is proved with searchSolutionFor that no solution exists, i.e. $I = nil$, the value a can be deleted. Note that, in contrary to weaker forms of consistency, when a value is pruned there is no need for GIC to repeat the process of iterating over the values remaining in the CN.

The second algorithm, GIC2 described in Algorithm 4 (ignoring light grey lines), uses timestamping. This is useful when GIC is maintained during a user-driven search. We use an integer variable `time` for counting time, and we introduce a two-dimensional array `stamp` that associates with each value (x, a) of the CN the last time (value of `stamp`$[x][a]$) a solution was found for that value (0, initially). We also assume that variables are implicitly totally ordered (for example, in lexicographic order). Then, the idea is to increment the value of the variable `time` whenever a new call to GIC2 is performed (see line 1) and to test `time` against each value (x, a) of the CN (see line 5) to determine whether it is necessary or not to search for a solution for (x, a). When a solution I is found, function handleSolution2/3 is called at line 10 in order to update stamps. Actually, we only update the stamps of values in I corresponding to variables that are processed after x in the loop of revisions (line 4) in Algorithm 4. These are the variables that have not been processed yet by the loop at line 4 of Algorithm 4. Finally, by further introducing a one-dimensional array `nbGic` that associates with each variable x of the CN the number of values in $dom(x)$ that have been proved to be GIC, it is possible to avoid some iterations of loop 5; see initialization at lines 2–3, testing at line 4 and update at line 4 of Algorithm 2.

The third algorithm, GIC3, described in Algorithm 4 when considering light grey lines, can be seen as a refinement of GIC2 obtained by exploiting residues, which correspond to solutions that have been previously found. Here, we introduce

Algorithm 4: GIC2/3(N: CN).

```
   // GIC3 is obtained by considering light grey colored instructions between lines 5 and 6, and after line 10
1  time++
2  foreach variable x ∈ vars^fut(N) do
3      nbGic[x] ← 0
4  foreach variable x ∈ vars^fut(N) such that nbGic[x] < |dom(x)| do
5      foreach value a ∈ dom(x) such that stamp[x][a] < time do
           if isValid(vars(N),residue[x][a]) then
               handleSolution2/3(x,residue[x][a])
               continue
6          I ← searchSolutionFor(N|x=a)
7          if I = nil then
8              remove a from dom(x)
9          else
10             handleSolution2/3(x,I)
               residue[x][a] ← I
```

Algorithm 5: handleSolution4(I : instantiation).

```
1  foreach variable x ∈ S^sup do
2      if I[x] ∉ gicValues[x] then
3          gicValues[x] ← gicValues[x] ∪ {I[x]}
4          if |gicValues[x]| = |dom(x)| then
5              S^sup ← S^sup \ {x}
```

a two-dimensional array `residue` that associates with each value (x, a) of the CN the last solution found for this value (potentially, during another call to GIC3). Because residual solutions may not be valid anymore, for each value (x, a) we need to test the validity of `residue`$[x][a]$ by calling the function isValid; see instructions between lines 5 and 6. If the residue is valid, we call handleSolution2/3 to update the other data structures, and we continue with the next value in the domain of x. A validity test, Algorithm 3, only checks that all values in a given complete instantiation are still present in the current domains. Of course, when a new solution is found, we record it as a residue; see instruction after line 10.

Our last algorithm, GIC4 described in Algorithm 6, is based on an original use of simple tabular reduction [19]. The principle is to record all solutions found during the enforcement of GIC in a table, so that an (adaptation of an) algorithm such as STR2 [20] can be applied. The current table is given by all elements of an array `solutions` at indices ranging from 1 to `nbSolutions`. As for STR2, we introduce two sets of variables called S^{val} and S^{sup}. The former allows us to limit validity control of solutions to the variables whose domains have changed recently (i.e., since the last execution of GIC4). This is made possible by reasoning from domain cardinalities, as performed at lines 3 and 26–27 with the array `lastSize`. The latter (S^{sup}) contains any future variable x for which at least one value is not in the array `gicValues`[x], meaning that it has still to be proved GIC. Related details can be found in [20]. After the initialization of S^{val} and S^{sup} (lines 1–8), each instantiation `solutions`[i] of the current table is processed (lines 11–16). If it remains valid (hence, a solution), we update structures `gicValues` and S^{sup} by calling the function handleSolution4. Otherwise, this instantiation is deleted by swapping it with the last one. The rest of the algorithm (lines 17–25) just tries to find a solution support for each value not present in `gicValues`. When a new solution is found, it is recorded in the current table (lines 23–24) and handleSolution4 is called (line 25).

Theorem 6. *Algorithms GIC1, GIC2, GIC3, and GIC4 enforce GIC.*

Proof. *Soundness.* Soundness is clear for all four algorithms. GIC1 (Algorithm 1) only removes a value (x, a) in line 5, which means that line 3 has not found any solution containing (x, a). Thus (x, a) is not GIC. GIC2 and GIC3 (Algorithm 4) only remove a value (x, a) in line 8, which means that line 6 has not found any solution containing (x, a). Thus (x, a) is not GIC. GIC4 (Algorithm 6) only removes a value (x, a) in line 21, which means that line 19 has not found any solution containing (x, a). Thus (x, a) is not GIC.

Completeness. Completeness of GIC1 is obvious: searchSolutionFor is called for all values (line 3), so if a value (x, a) is not GIC, it will necessarily be removed in line 5.

In GIC2 (Algorithm 4), a value (x, a) is let in the domain without checking if it belongs to a solution when `nbGic`$[x] \geq |dom(x)|$ (line 4) or `stamp`$[x][a] \geq$ `time` (line 5). As `time` is incremented at each new call to GIC2 (line 1), `stamp`$[x][a]$ is (greater than or) equal to `time` only if line 3 of Algorithm 2 has been executed, which means that a solution containing (x, a) has already been found, and thus (x, a) is GIC. Lines 2 and 4 of Algorithm 2 ensure that `nbGic`[x] is equal to the

Algorithm 6: GIC4(N: CN).

```
   // Initialization of structures
1  S^val ← ∅
2  foreach variable x ∈ vars(N) do
3      if |dom(x)| ≠ lastSize[x] then
4          S^val ← S^val ∪ {x}
5  S^sup ← ∅
6  foreach variable x ∈ vars^fut(N) do
7      gicValues[x] ← ∅
8      S^sup ← S^sup ∪ {x}
   // The table of current solutions is traversed
9  i ← 1
10 while i ≤ nbSolutions do
11     if isValid(S^val,solutions[i]) then
12         handleSolution4(solutions[i])
13         i++
14     else
15         solutions[i] ← solutions[nbSolutions]
16         nbSolutions--
   // Search for values not currently supported is performed
17 foreach variable x ∈ S^sup do
18     foreach value a ∈ dom(x) \ gicValues[x] do
19         I ← searchSolutionFor(N|x=a)
20         if I = nil then
21             remove a from dom(x)
22         else
23             nbSolutions++
24             solutions[nbSolutions] ← I
25             handleSolution4(I)
26 foreach variable x ∈ vars^fut(N) do
27     lastSize[x] ← |dom(x)|
```

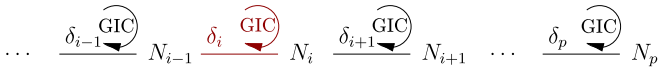
number of values in $dom(x)$ that GIC2 has already found in a solution at the current call to GIC2 ($\texttt{time}^{th}$ call). Hence, if $\texttt{nbGic}[x] \geq |dom(x)|$, all values of x have been proved GIC.

Like GIC2, GIC3 (Algorithm 4) lets a value (x, a) in the domain without checking if it belongs to a solution when $\texttt{nbGic}[x] \geq |dom(x)|$ or $\texttt{stamp}[x][a] \geq \texttt{time}$. But in addition, GIC3 avoids checking if (x, a) belongs to a solution when isValid($vars(N)$,$residue[x][a]$) is true (grey colored line between lines 5 and 6). $residue[x][a]$ stores a solution containing (x, a) found at a previous call to GIC3 (grey colored line after line 10). Function isValid checks if that solution is still valid at the current call to GIC3. If yes, $residue[x][a]$ is a proof of GIC for (x, a) (and also for other values appearing in $residue[x][a]$—call to handleSolution2/3 in grey colored lines between lines 5 and 6).

In GIC4 (Algorithm 6) the conditions to avoid checking if a value (x, a) belongs to a solution are $x \notin S^{sup}$ (line 17) and $a \in \texttt{gicValues}[x]$ (line 18). $\texttt{gicValues}[x]$ is initialized to the empty set in line 7 and values are only added to $\texttt{gicValues}[x]$ in line 3 of Algorithm 5. To prove that these added values are GIC we have to prove that handleSolution4 is always called with valid solutions. handleSolution4 is called in lines 12 and 25 of Algorithm 6. In the call of line 25 I is obviously a valid solution as it is the result of the call to searchSolutionFor in line 19. In line 12 handleSolution4 is called with $\texttt{solutions}[i]$. Thanks to lines 24 and 15, we know that $\texttt{solutions}$ is an array that only contains instantiations that were valid solutions at the previous call to GIC4. As in line 11 isValid has checked that all values of variables with a modified domain are still in the domain, $\texttt{solutions}[i]$ is a valid solution. Thus, values in $\texttt{gicValues}[x]$ are GIC. As for the other condition ($x \notin S^{sup}$), thanks to line 8 we know that GIC4 puts all variables in S^{sup} in the initialization phase. The only place where a variable is removed from S^{sup} is line 5 of Algorithm 5. This line is executed only if $\texttt{gicValues}[x]$ contains all values in $dom(x)$ (test in line 4). Thus, by avoiding checking GIC on values of variables which are not in S^{sup} we do not miss the pruning of any GIC-inconsistent value. □

The worst-case space complexity (for the specific data structures) of GIC1 is in $O(1)$. For GIC2 and GIC3, this is in $O(nd)$ because $\texttt{nbGic}$ is in $O(n)$, $\texttt{stamp}$ and $\texttt{residue}$ are in $O(nd)$. For GIC4, S^{val}, S^{sup} and $\texttt{lastSize}$ are in $O(n)$, $\texttt{gicValues}$ is in $O(nd)$, and the structure $\texttt{solutions}$ is in $O(n^2d)$ because for each of the nd values, we may need to record a solution (of size n). The time complexity of the GIC algorithms can be expressed in term of the number of calls to the (oracle) searchSolutionFor. For GIC1, this is in $O(nd)$. For GIC2, in the *best-case*, only d calls are necessary, each call allowing to prove (through timestamping) that n values are GIC. For GIC3 and GIC4, still in the *best-case* and assuming the case of maintaining GIC (i.e., after the assignment of a variable by the user), no call to the oracle is necessary (residues

Fig. 1. A GIC-staged configuration trace T.



(a) A GIC-staged configuration trace T

(b) The GIC-staged configuration trace T' obtained from T, after discarding arbitrarily the decision δ_i

Fig. 2. Restoring GIC-staged configuration traces.

and the current table are sufficient by themselves to prove that all values are GIC). This rough analysis of time complexity suggests that GIC3 and GIC4 might be the best options.

Observe that when GIC is maintained during search, one can always enforce the weaker (and cheaper) consistency GAC before GIC. This is the approach we systematically follow when maintaining GIC during search (with any of the introduced GIC algorithms).

5. An algorithm for restoring GIC

In this section, we address the issue of restoring GIC after the user decides to discard arbitrarily a decision that has been taken during a configuration process based on GIC. So, the context is a CN N that is given initially, a sequence of p decisions $\Delta = \langle \delta_1, \delta_2, \ldots, \delta_p \rangle$ taken on N (in that order) by the user, and GIC maintained on N.

More formally, a *configuration trace* T on a CN N from a sequence of decisions $\langle \delta_1, \ldots, \delta_p \rangle$ is represented by a sequence of CNs $\langle N_1, \ldots, N_p \rangle$ such that N_i is the CN obtained from N_{i-1} (starting with $N_0 = N$) after taking the decision δ_i and running some propagation algorithm. For each decision δ_i, it is easy to identify the set $\texttt{deleted}(\delta_i)$ of values deleted due to the combined effect of δ_i and constraint propagation: we have $\texttt{deleted}(\delta_i) = \texttt{values}(N_{i-1}) \setminus \texttt{values}(N_i)$. These sets are useful for backtracking, for example with the so-called *trailing* mechanism (see e.g., [21]).

Without loss of generality, we assume that N is (or has been made) initially GIC. A configuration trace $T = \langle N_1, \ldots, N_p \rangle$ on N from $\langle \delta_1, \ldots, \delta_p \rangle$ is *GIC-staged* iff $N_i = GIC(N_{i-1}|_{\delta_i})$, $\forall i \in 1..p$. In other words, a GIC-staged configuration trace is a trace such that GIC is maintained at each step. An illustration is given by Fig. 1, where each edge represents the action of taking a decision δ_i and enforcing GIC: we have $N_1 = GIC(N_0|_{\delta_1})$, $N_2 = GIC(N_1|_{\delta_2})$, $\ldots$, $N_p = GIC(N_{p-1}|_{\delta_p})$.

In our work, we are interested in GIC-staged configuration traces, and our objective is to be able to rebuild GIC-staged configuration traces after discarding arbitrarily any taken decision(s). Note that it has the flavor of dynamic backtracking [22], but in the context of the very strong consistency GIC. An illustration of a GIC-staged configuration trace, $T = \langle N_1, \ldots, N_{i-1}, N_i, N_{i+1}, \ldots, N_p \rangle$, is given by Fig. 2(a): we have, $N_1 = \text{GIC}(N_0|_{\delta_1})$, $\ldots$, $N_{i-1} = \text{GIC}(N_{i-2}|_{\delta_{i-1}})$, $N_i = \text{GIC}(N_{i-1}|_{\delta_i})$, $N_{i+1} = \text{GIC}(N_i|_{\delta_{i+1}})$, $\ldots$, $N_p = \text{GIC}(N_{p-1}|_{\delta_p})$. If ever the decision δ_i is discarded by the user, then we want to compute a new GIC-staged configuration trace, $T' = \langle N_1, \ldots, N_{i-1}, N'_i, \ldots, N'_{p-1} \rangle$, as in Fig. 2(b), where $N_1 = \text{GIC}(N_0|_{\delta_1})$, $\ldots$, $N_{i-1} = \text{GIC}(N_{i-2}|_{\delta_{i-1}})$, $N'_i = \text{GIC}(N_{i-1}|_{\delta_{i+1}})$, $\ldots$, $N'_{p-1} = \text{GIC}(N'_{p-2}|_{\delta_p})$. Hence, we need to compute $p - i$ new CNs: N'_i, $\ldots$, N'_{p-1}, but as observed earlier, it suffices to identify the (new) sets of deleted values at levels i, $\ldots$, $p - 1$. Later, in Algorithm 9, this will be the role of the data structure `Knw`.

Reclassifying a deleted value of trace T means finding the level at which this value must be deleted in the new trace T', or proving that it is no longer deleted. It is worthwhile to analyze which values must be reclassified when a decision is discarded and a new GIC-staged configuration trace is aimed to be computed. First, at each level, i.e. after each decision δ, we can distinguish between the values that are *directly removed* by δ and those that are removed by propagating these initial direct deletions through the CN. For example, if x is a variable such that $dom(x) = \{a, b, c\}$ and δ corresponds to the variable assignment $x = a$, then the values directly removed by δ are $\{b, c\}$. All other values removed while enforcing GIC after taking δ are said to be *indirectly removed* by δ. The different sets of values that are removed either directly or indirectly in a GIC-staged configuration trace are depicted in Fig. 3.

In the following, for the sake of simplicity, we assume that all decisions correspond to variable assignments. Interestingly, once the decision δ_i is discarded, computing the new GIC-staged configuration trace T' only requires to reclassify the deleted values that belong to the grey-colored regions of Fig. 3. The proof is as follows:

1. nothing changes for levels strictly less than i; in other words, for any integer j such that $0 < j < i$, $\texttt{deleted}(\delta_j)$ remains the same.

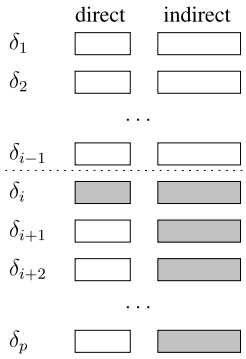


Fig. 3. The deleted values that need to be reclassified, when the decision δ_i is retracted, are those in grey-colored regions.

2. any value (x, a) directly removed by a decision δ_j with $j > i$ will necessarily be again removed directly by δ_j in the new trace ; in other words, (x, a) remains in $\texttt{deleted}(\delta_j)$.[2] Indeed, the relaxation (retracted decision) does not allow us to remove (x, a) before taking δ_j in the new trace.

All values that must be reclassified are put in a data structure called `Unk` and transferred progressively, while running the algorithm we propose, to a data structure called `Knw` that we introduce later.

To restore GIC, Function restoreAfterDeleting(), presented in Algorithm 9, must be called. When this function is called for, it is for a GIC-staged configuration trace $T = \langle N_1, \ldots, N_p \rangle$ on a CN N from a sequence of decisions $\Delta = \langle \delta_1, \ldots, \delta_p \rangle$. For simplicity and because we assume that at each decision level, i.e. for each decision δ, we know $\texttt{deleted}(\delta)$, only N_p is specified as a parameter (as well as the decision δ_i to be discarded). General statements useful for describing our algorithm are:

- take(δ) : the decision δ is added at the end of the current sequence of decisions (push operation) and $\texttt{deleted}(\delta)$ initially contains the values of $dom(var(\delta))$ that are not compatible with δ (direct deletions).
- backtrack() : the last taken decision δ is removed from the current sequence of decisions (pop operation) and values in $\texttt{deleted}(\delta)$ are restored in domains.

The data structures used by our function are the following:

- Δ^{replay} is the sequence $\langle \delta_{i+1}, \delta_{i+2}, \ldots, \delta_p \rangle$ of decisions to be replayed, once δ_i is discarded.
- `Knw` is an array of size $p - i + 1$, indexed from i to p, of sets of values. This is a central data structure in our algorithm, allowing us to rebuild the GIC-staged configuration trace T'. Once the computation of this array is finished, all values are reclassified. If $(x, a) \in \texttt{Knw}[h]$ with $i \leq h < p$, this means that (x, a) must be deleted at level h, but if $(x, a) \in \texttt{Knw}[p]$, this means that (x, a) will no more be deleted in the new trace (because we only kept $p - 1$ decisions).
- `Unk` is a map that associates an integer interval $h_{min}..h_{max}$, called *classification interval*, with each key of the form (x, a). The meaning of an entry $((x, a), h_{min}..h_{max})$ of `Unk` is that the value (x, a) must be reclassified at a level ranging from h_{min} to h_{max}. During the execution of the algorithm, the classification interval of every entry is refined until the precise deletion level is known, i.e. until $h_{min} = h_{max}$, in which case the entry is deleted and transferred to the structure `Knw`, described above. Notice that when the classification interval associated with a value ends up at $p..p$, this actually means that the value is no more deleted by GIC. As any map, `Unk` supports the following operations:
  - `Unk.clear()` empties the map,
  - `Unk.containsKey`$((x, a))$ indicate whether or not there is an entry for key (x, a),
  - `Unk.get`$((x, a))$ returns the integer interval associated with key (x, a),
  - `Unk.put`$((x, a), h_{min}..h_{max})$ stores the pair $((x, a), h_{min}..h_{max})$ possibly replacing any previous entry with key (x, a),
  - `Unk.size()` returns the number of entries in the map,
  - `Unk.delete`$((x, a))$ deletes the entry with key (x, a).

Before presenting function restoreAfterDeleting() in detail, we describe the two primitive functions increaseMin and decreaseMax that will be used to update the classification interval of a key (x, a) in `Unk` and move such a key to `Knw` when needed. Function increaseMin$((x, a), h)$ (Algorithm 7) first retrieves the interval for key (x, a) in `Unk` (line 1). If the new lower bound h reduces the interval to a singleton, the key (x, a) is transferred from `Unk` to `Knw` with the right deletion level (lines 2–4). Otherwise, if the lower bound changed, the interval is updated (lines 5–6). Function decreaseMax$((x, a), h)$

[2] Strictly speaking, the value (x, a) is deleted by δ_j, but not at the same level (since δ_i has been discarded).

Algorithm 7: increaseMin((x, a) : value, h : integer).

```
1  h_min..h_max ← Unk.get((x, a))
2  if h = h_max then
3  |  Unk.delete((x, a))
4  |  Knw[h] ← Knw[h] ∪ {(x, a)}
5  else if h > h_min then
6  |  Unk.put((x, a), h..h_max)
```

Algorithm 8: decreaseMax((x, a) : value, h : integer).

```
1  h_min..h_max ← Unk.get((x, a))
2  if h = h_min then
3  |  Unk.delete((x, a))
4  |  Knw[h] ← Knw[h] ∪ {(x, a)}
5  else if h < h_max then
6  |  Unk.put((x, a), h_min..h)
```

Algorithm 9: restoreAfterDeleting(N_p: CN, δ_i : decision).

Input: N_p is the last CN of the current GIC-staged configuration trace $T = \langle N_1, \ldots, N_i, \ldots, N_p \rangle$ from $\langle \delta_1, \ldots, \delta_i, \ldots, \delta_p \rangle$.
Input: δ_i is the decision to discard.
Result: A new GIC-staged trace T' from $\langle \delta_1, \ldots, \delta_{i-1}, \delta_{i+1}, \ldots, \delta_p \rangle$

```
   // Initialization of structures
1  Δ^replay ← ⟨δ_{i+1}, δ_{i+2}, ..., δ_p⟩
2  Unk.clear()
3  foreach j from i to p do
4  |  Knw[j] ← ∅
   // Deleted values of the current trace T to (re)classify
5  foreach j from p downto i + 1 do
6  |  foreach (x, a) ∈ deleted(δ_j) do
7  |  |  if x = var(δ_j) then
8  |  |  |  Knw[j − 1] ← Knw[j − 1] ∪ {(x, a)} // directly removed
9  |  |  else
10 |  |  |  Unk.put((x, a), j − 1..p) // value to reclassify
11 |  backtrack()
12 foreach (x, a) ∈ deleted(δ_i) do
13 |  Unk.put((x, a), i..p)
14 backtrack()
   // Refining classification intervals
15 refineIntervals(Unk, Knw, Δ^replay)
   // Finalizing classification
16 while Unk.size ≠ 0 do
17 |  foreach ((x, a), h_min..h_max) ∈ Unk do
18 |  |  pick a value s in [h_min..h_max − 1]
19 |  |  I ← searchSolutionFor(N|_{δ_j ∈ Δ^replay | j ≤ s+1} ∪ {x=a})
20 |  |  if I = nil then
21 |  |  |  decreaseMax((x, a), s)
22 |  |  else
23 |  |  |  foreach (y, b) ∈ I do
24 |  |  |  |  if Unk.containsKey((y, b)) then
25 |  |  |  |  |  increaseMin((y, b), s + 1)
   // Building the new trace T'
26 foreach δ_j ∈ Δ^replay (with j from i + 1 to p) do
27 |  take(δ_j)
28 |  foreach (x, a) ∈ Knw[j − 1] do
29 |  |  remove (x, a)
```

(Algorithm 8) behaves the same way: it retrieves the interval for (x, a), checks whether the new upper bound h reduces the interval to a singleton, and depending on the answer transfers (x, a) to Knw or updates the upper bound.

Function restoreAfterDeleting() works as follows. Lines 1–4 initialize the structures: the decisions to be replayed are put in Δ^{replay}, and structures Unk and Knw are emptied because deleted values to be reclassified are not known yet, and so,

Algorithm 10: refineIntervals(Unk, Knw, Δ^{replay}).

Input: Unk, Knw: data structures.
Input: Δ^{replay}: sequence of decisions.
Result: Updated classification intervals in Unk and possibly new keys in Knw.

```
1  foreach δ_j ∈ Δ^replay (with j from i + 1 to p) do
2      take(δ_j)
3      enforceGAC()
4  foreach j from p downto i + 1 do
5      foreach (x, a) ∈ deleted(δ_j) do
6          if Unk.containsKey((x, a)) then
7              decreaseMax((x, a), j − 1)
8      backtrack()
```

no classification has been performed yet. Line 5–14 handle all values that have been deleted from level i in the current trace T. All levels, from p down to i, are iterated over, by systematically backtracking (lines 11 and 14). The new status of any deleted value (x, a) at a level $j > i$ is either known (line 8), because of a direct deletion, or unknown (line 10). For the former case, the test at line 7 is sufficient because we only consider variable assignments. For the latter case, the interval $j-1..p$ bounds the different possibilities (the value cannot be deleted at a level less than $j-1$ and possibly can remain valid at the end of the new trace). For level i (lines 12–13), as shown in Fig. 3, all deleted values must be reclassified. Line 15 attempts to refine the classification intervals of values by applying a polynomial process, such as simulating GIC by an efficient local consistency technique. We show later how to use GAC for refining the intervals. Lines 16–25 finalize classification. For each unclassified value (x, a), we have an interval of the form $h_{min}..h_{max}$. Given (x, a), we select a value (level) s that will be used to decrease the size of the interval of (x, a); for our experimentation, we shall select $h_{max} - 1$. In line 19, we call searchSolutionFor in the network where we force all decisions from δ_{i+1} to δ_{s+1} (that correspond to new levels i to s) plus $x = a$. The purpose of this call to searchSolutionFor is to check if there exists a solution for (x, a) at level s. If this is not the case, we can decrease the upper bound of the interval to s (since, we know that GIC is enough to prune this value at level s). Otherwise, we can increase the lower bound of all unclassified values that are present in the found solution I (see Lines 23–25). This includes of course (x, a). Lines 26–29 build the new GIC-staged configuration trace T' from data in Knw.

Reclassifying values may be expensive because for proving GIC of a value we have to run a complete search procedure (see line 19), and possibly several times. One idea is to use a cheap process, such as applying GAC, as an approximation of GIC in a preliminary stage. For example, suppose that $((x, a), 10..14)$ is present in Unk and that MAC removes (x, a) at level 10. We can then deduce that (x, a) is GIC-inconsistent at level 10, and consequently directly classify (x, a) in T'. If MAC removes (x, a) only at level 12, then we can replace $((x, a), 10..14)$ by $((x, a), 10..12)$ in Unk. To summarize, running MAC on decisions of Δ^{replay} allows us to refine the classification intervals at a very moderate price. This what is done by the function refineIntervals(Unk, Knw, Δ^{replay}) in Algorithm 10.

In lines 1–3, GAC is maintained after each decision taken in sequence from Δ^{replay}. Then, in lines 4–7, we process each level in sequence from bottom to top. If (x, a) has been deleted (at level $j-1$) by GAC after decision δ_j, we know that GIC will prune (x, a) at level $j-1$ at the very last. Hence, we update the interval of (x, a) accordingly. After having processed all values of a level, we have to call the backtrack function to restore domains in the state they were before applying GAC (line 8).

We have the guarantee that restoreAfterDeleting performs as if GIC had been maintained on N from $\langle \delta_1, \ldots, \delta_{i-1}, \delta_{i+1}, \ldots, \delta_p \rangle$. The proof is based on the invariant property that Unk is *sound*, that is, the level of deletion of any (x, a) present in Unk is contained in the interval stored in Unk.

Lemma 1. *If* Unk *is sound before a call to* refineIntervals*, then it remains sound after the execution of* refineIntervals *as described in Algorithm 10.*

Proof. The interval of (x, a) in Unk is modified in line 7 only if GAC has removed (x, a) after decision δ_j has been applied. By enforcing GIC instead of GAC, it is obvious that (x, a) cannot be removed later. Thus, the last level at which (x, a) can be removed is $j-1$ because δ_j is the $(j-1)$th decision. □

Theorem 7. *Let* $\langle N_1, \ldots, N_p \rangle$ *be a GIC-staged configuration trace on a CN* N *from a sequence of decisions* $\langle \delta_1, \ldots, \delta_p \rangle$*. Calling* restoreAfterDeleting(N_p, δ_i) *builds the GIC-staged configuration trace on* N *from* $\langle \delta_1, \ldots, \delta_{i-1}, \delta_{i+1}, \ldots, \delta_p \rangle$*.*

Proof. Lines 11 and 14 ensure that, after line 14, all values removed in levels i to p have been restored and put either in Knw or in Unk. They will thus all be processed to find their right level of deletion.

We first prove that before line 26, all values are put in Knw at their right level of deletion. After lines 1–14 are executed, all values in Knw have been put in it at line 8. They have been put at their right level because they cannot be higher

Table 1
Features of six Renault configuration instances.

	n	d	e	r	t	D	T
Souffleuse	32	12	35	3	55	145	350
Megane	99	42	113	10	48,721	396	194,838
Master	158	324	195	12	26,911	732	183,701
Small	139	16	147	8	222	340	3,044
Medium	148	20	174	10	2,718	424	9,532
Big	268	324	332	12	26,881	1,273	225,989

(relaxation) and they are necessarily removed (line 7 tells us they belong to the instantiated variable). After lines 1–14, `Unk` is sound because it stores the largest possible interval for every non-reclassified value. After line 14, intervals are refined in lines 15, 21, and 25. By Lemma 1, `Unk` remains sound after line 15. In line 21, the h_{max} of (x, a) is decreased correctly because there were no solutions containing (x, a) at level s. In line 25, it is also obviously correct to increase the h_{min} of (y, b) to $s + 1$ as we found a solution at level s. By construction of functions increaseMin and decreaseMax, we know that after line 25, `Knw` contains only correct levels of deletion.

We then prove that the loop in lines 16–25 terminates. When we enter the loop in line 16, `Unk` only contains values with a non-singleton interval. By construction of increaseMin and decreaseMax, any value with an interval shrunk to singleton is moved to `Knw`. Thus, line 17 can only select values with non-singleton intervals, and the way s is selected in line 18 ensures that any iteration of the loop of line 17 strictly decreases the size of at least one interval in `Unk`. As all intervals have finite size and as `Unk` contains a finite number of values, the loop of line 16 terminates.

The fact that after line 25 all values are put in `Knw` at their right level of deletion and the fact that the algorithm will eventually reach line 26 guarantees that lines 26–29 build a GIC-staged configuration trace. □

The algorithm restoreAfterDeleting is obviously exponential in time as it solves an NP-hard problem (see Theorem 4). Nevertheless, we can analyze the number of times it calls the NP-hard oracle searchSolutionFor.

Theorem 8 *(Complexity). Let $\langle N_1, \ldots, N_p \rangle$ be a GIC-staged configuration trace on a CN N from a sequence of decisions $\langle \delta_1, \ldots, \delta_p \rangle$. The number of times* restoreAfterDeleting(N_p, δ_i) *calls* searchSolutionFor *is in $O(nd \cdot log_2(p - i))$.*

Proof. Each time searchSolutionFor is called in line 19, the interval of (x, a) is shrunk either to $[h_{min}, s]$ or to $[s + 1, h_{max}]$, depending on whether line 21 or line 25 is executed. Hence, if s is selected in a dichotomic way (that is, $s = \lfloor \frac{h_{min}+h_{max}}{2} \rfloor$), each call to searchSolutionFor leads to an interval of size at most $\lceil \frac{h_{max}-h_{min}+1}{2} \rceil$. `Unk` contains at most nd values and intervals cannot be larger than $p - i + 1$. As a result, the number of times restoreAfterDeleting can call searchSolutionFor is in $O(nd \cdot log_2(p - i))$. □

6. Experiments

In order to show the practical interest of our approach, we have performed several experiments mainly using a computer with processors Intel(R) Core(TM) i7-2820QM CPU 2.30 GHz; however, for GIC restoration, we used a cluster of Xeon 3.0 GHz processors with 13 GB of RAM. Our main purpose was to determine whether maintaining/restoring GIC is a viable option for configuration-like problem instances (and for interactive puzzle creation), as well as to compare the relative efficiency of the four GIC algorithms described in Section 4.

In Table 1, we show relevant features of car configuration instances, generated with the help of our industrial partner Renault. For each of the six instances currently available,[3] we indicate

- the number of variables (n),
- the size of the greatest domain (d),
- the number of constraints (e),
- the greatest constraint arity (r),
- the size of the greatest table (t),
- the total number of values ($D = \sum_{x \in vars(N)} |dom(x)|$),
- and the total number of tuples ($T = \sum_{c \in cons(N)} |rel(c)|$).

[3] See www.irit.fr/~Helene.Fargier/BR4CP/benches.html and www.xcsp.org.

Table 2
CPU time (in seconds) to establish GIC on Renault configuration instances, and to maintain it (average over 100 random greedy executions).

	Establishing GIC with				Maintaining GIC with			
	GIC1	GIC2	GIC3	GIC4	GIC1	GIC2	GIC3	GIC4
Souffleuse	0.02	0.01	0.01	0.01	0.13	0.07	**0.02**	**0.02**
Megane	2.94	0.71	0.72	0.71	4.26	1.18	0.05	**0.04**
Master	2.45	1.35	1.33	1.33	9.81	3.57	0.07	**0.06**
Small	0.14	0.02	0.03	0.03	0.32	0.05	**0.01**	**0.01**
Medium	0.26	0.04	0.05	0.04	0.35	0.04	**0.01**	**0.01**
Big	4.19	1.16	1.10	1.10	12.6	2.60	**0.05**	**0.05**

Table 3
Number of conflicts encountered when running nFC2 and MAC (sum over 100 random executions).

	Souffleuse	Megane	Master	Small	Medium	Big
nFC2	252,605	313,910	time-out	3,728	7,824	time-out
MAC	0	7	5	0	3	3

Table 4
CPU time (in seconds) to establish GIC on some Crosswords instances, and to maintain it on average over 100 random greedy executions.

	Establishing GIC with				Maintaining GIC with			
	GIC1	GIC2	GIC3	GIC4	GIC1	GIC2	GIC3	GIC4
ogd-vg5-5	2.25	0.67	0.67	0.67	2.34	0.79	0.73	**0.70**
ogd-vg5-6	6.40	2.18	2.19	2.19	7.42	2.82	2.58	**2.48**
ogd-vg5-7	25.8	9.91	9.87	9.84	33.4	15.2	14.3	**13.8**

6.1. Establishing/maintaining GIC

The left part of Table 2 presents the CPU time required to establish GIC on the six Renault configuration instances. Clearly GIC1 is outperformed by the three other algorithms, which have here rather similar efficiency. The right part of Table 2 aims at simulating the behavior of a configuration software user who makes the variable choices and value selections. It presents the CPU time required to maintain GIC along a complete branch built by performing random variable assignments down to a leaf. (Random variable assignment simulates the user, who chooses the variables and the values according to her preference.) Specifically, variables and values are randomly selected in turn, and after each assignment, GIC is systematically enforced to maintain this property. Of course, no conflict (dead-end) can occur along the branch due to the strength of GIC, which is why we use the term of greedy executions. CPU times are given on average for 100 executions (different random orderings). When establishing GIC, any call to searchSolutionFor is performed with the help of the algorithm MAC (table constraints being filtered with the technique called Simple Tabular Reduction [19,20,23]). For all instances, GIC3 and GIC4 are maintained very fast, whereas on the biggest instances, GIC2 requires a few seconds and GIC1 around ten seconds.

One great advantage of GIC is that it guarantees that a conflict can never occur during a configuration session. However, one may wonder whether the risk of failure(s) is really important in user-driven searches that use a weaker consistency such as GAC or a partial form of it (Forward Checking). Table 3 shows the number of conflicts (sum over 100 executions using random orderings) encountered when following a MAC or a nFC2 [24] strategy. The number of conflict situations can be very large with nFC2 (for two instances, we even report the impossibility of finding a solution within 10 minutes with some random orderings). For MAC, the number of failures is rather small but the risk is not null (for example, the risk is equal to 7% for megane).

The encouraging results obtained on Renault configuration instances led us to test other problems, in particular to get a better picture of the relative efficiency of the various GIC algorithms. For example, on classical Crossword instances (see Table 4), GIC1 is once again clearly outperformed while the three other algorithms are quite close, where there is still a small benefit of using GIC4.

It is worthwhile to note that GIC is a nice property that can be useful when puzzles, where hints are specified, have to be created. Typically, one looks for puzzles where only one solution exists. One way of building such puzzles is to add hints in sequence, while maintaining GIC, until all domains become singleton. For example, this is a possible approach for constructing Sudoku and Magic Square grids, with the advantage that the user can choose freely the position of the hints.[4] On the left part of Table 5, we report the time to enforce GIC on empty Sudoku grids of size 9×9 and 16×16, and on empty Magic squares of size 4×4 and 5×5. On the right part we report the average time required to maintain GIC until

[4] However, we are not claiming that maintaining GIC is the unique answer to this problem.

Table 5
CPU time (in seconds) to establish GIC on Puzzle instances, and to maintain it on average over 100 random greedy executions until a unique solution is found.

	Establishing GIC with				Maintaining GIC with			
	GIC1	GIC2	GIC3	GIC4	GIC1	GIC2	GIC3	GIC4
sud-9 × 9	1.58	0.32	0.32	0.31	15.3	2.71	2.10	**1.74**
sud-16 × 16	6.04	0.51	0.50	0.50	246	25.5	26.5	**18.9**
magic-4 × 4	0.96	0.26	0.28	0.28	1.63	**0.69**	0.71	0.71
magic-5 × 5	14.7	3.01	3.10	2.99	55.1	15.9	15.6	**13.7**

all variables become fixed (i.e., with only singleton domains), meaning that after several hints have been randomly selected and propagated, we have the guarantee of having a one-solution puzzle. GIC4 is a clear winner, with for example, a 30% speedup over GIC2 and GIC3 on sudoku-16 × 16, and more than one order of magnitude over GIC1. Overall, the results we obtain show that maintaining GIC is a practicable solution (at least for some problems) as the average time between each decision of the user is small with GIC4.

6.2. Restoring GIC

In a second set of experiments, we have focused on the issue of restoring GIC, as developed in Section 5. We have tested Algorithm 9 against all Renault car configuration instances introduced earlier. For each instance, the protocol we have used is the following. First, we have built a complete GIC-staged configuration trace T (branch), by randomly assigning each variable in turn (and subsequently maintaining GIC). Then, we have discarded arbitrarily a decision used for T, and applied our algorithm that restores GIC. Actually, we have successively collected information about GIC restoration for decisions discarded at levels i ranging respectively from 0 (first decision taken) to 80.[5] After decision 80, all these instances only contain singleton domains, and so, are completely solved. In other words, we have independently tested GIC restoration on T, when a decision was discarded at level $i = 0$, at level $i = 1$, and so on until level 80. The results we present are given on average over 100 random complete GIC-staged configuration traces per level. Such traces are computed initially by randomly selecting decisions.

We have been interested in:

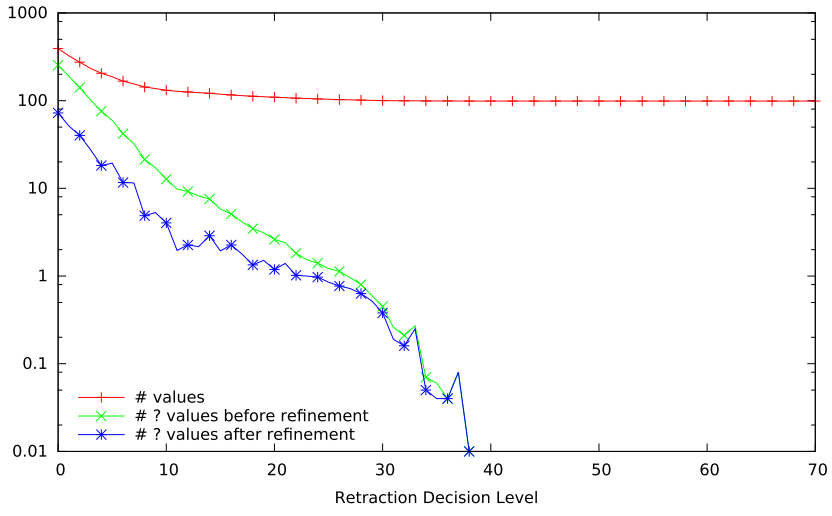
- the total number of values (i.e., the number of values over all variable domains), denoted by # values
- the number of unclassified values when discarding a decision, at the point before refining classification intervals through MAC; this is denoted by "#? values before refinement"
- the number of unclassified values when discarding a decision, at the point just after having refined classification intervals through MAC; this is denoted by "#? values after refinement"

We have observed the same trend when testing the 6 configuration instances: i) it is only when the retracted decision belongs to the first ones taken by the user that a substantial computing effort is required, and ii) classification refinement by MAC is very useful. Figs. 4(a) and 4(b) show results for the car models Megane (a mid-size instance) and Master (a large size instance). The x-axis indicates the level at which the decision (from the complete GIC-staged configuration trace) was discarded before restoring GIC. For example, when the retraction decision level is 8, we can see for Master that the number of unclassified values after MAC refinement is around 10 (on average). The impact of classification refinement by MAC is clearly visible, as it corresponds to the gap between the two bottom curves. (Note the logarithmic scale for the y-axis.) The two curves only merge when the average number of unclassified values is around 1.

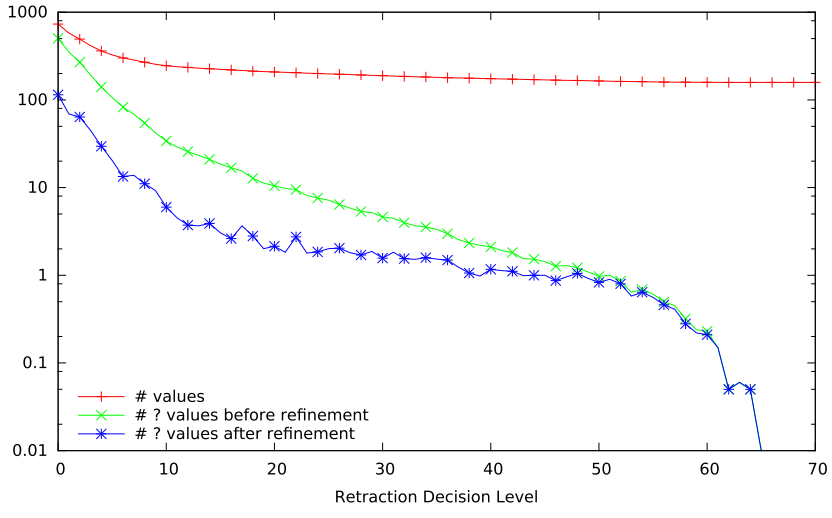
It is also interesting to see how many calls to the procedure (oracle) searchSolutionFor are necessary to classify the remaining unclassified values after refining intervals through MAC. In Section 5, we mentioned the possibility of a dichotomic approach (see Theorem 8). However, for configuration instances, picking $s = h_{max} - 1$ at line 18 of Algorithm 9 is a relevant choice because most of the time it allows us to prove directly that the value is GIC-consistent. This is what we have observed: the proportion of successful calls (i.e., of calls returning a solution) is very high. Besides, when a solution is found, multidirectionality can be used to refine classification intervals of other values (lines 23–25 in Algorithm 9). Fig. 5 shows the average number of calls to searchSolutionFor for each instance and each retraction decision level. For the largest instances, in the worst-case (level 0), the number of calls is around 100. At level 8, less than 10 calls are required. Table 6 shows the average number of unclassified values after GAC refinement (# ? values) and the average number of calls to searchSolutionFor (# calls), computed over all retraction decision levels. The number of calls to the oracle is never more than 75% of the number of unclassified values.

Table 7 gives the CPU time required to restore GIC. The worst-case maximum time is 1.5 seconds for megane, but note this is only 0.2 second on average. This confirms that our approach can perfectly be used in an interactive configuration context.

[5] This is only 32 for souffleuse, as this corresponds to the number of variables of this instance.



(a) Instance Megane



(b) Instance Master

Fig. 4. Restoring GIC at different decision levels. Average values over 100 executions.

Table 6
Average number of unclassified values after GAC refinement (# ? values) and average number of calls to searchSolutionFor (# calls) for Renault car configuration instances.

	Souffleuse	Megane	Master	Small	Medium	Big
# ? values	3.23	3.68	5.95	2.05	2.27	9.67
# calls	2.22	2.92	4.92	0.40	0.49	4.44

Finally, Table 8 shows how the algorithm we propose for restoring GIC after discarding a decision δ_i (referred to as optimized GIC restoration) behaves with respect to a simple algorithm (referred to as naive GIC restoration) that just erases all decisions from δ_i to the last before replaying all of them except the discarded decision δ_i, while maintaining GIC with GIC4. The results are given on average for 100 random GIC-staged configuration traces, with GIC restoration triggered after 80 decisions have been taken (32 for souffleuse) and the first of these decisions has been discarded. Note that discarding the first decision is the most adverse case (i.e., requiring the most computing effort), which is the reason for studying this particular case. We observe that during this process, the number of calls to the oracle searchSolutionFor is very limited when our optimized algorithm is used. Our algorithm is between 2 and 5 times faster than the naive one. On these instances, our algorithm never requires more than 1 second.

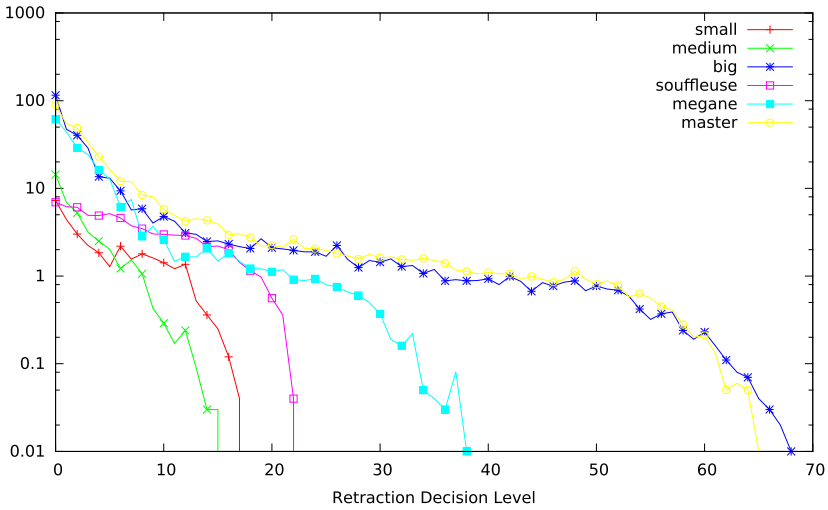


Fig. 5. Restoring GIC at different decision levels. Number of calls to the procedure (oracle) searchSolutionFor. Average over 100 executions.

Table 7
Minimum, maximum and average CPU times in second(s) to restore GIC, after discarding any decision (in range 0..80). Values are computed over 100 random traces.

	min	avg	max
Souffleuse	0	0.002	0.006
Megane	0	0.247	1.440
Master	0.001	0.225	0.722
Small	0	0.006	0.013
Medium	0	0.006	0.011
Big	0.001	0.212	0.855

Table 8
Minimum, maximum and average CPU times in millisecond(s) to restore GIC, after discarding the first decision of a sequence composed of 80 decisions. The average number of calls to the procedure (oracle) searchSolutionFor is also indicated. Values are computed over 100 random traces.

	Naive GIC restoration				Optimized GIC restoration			
	min	avg	max	# calls	min	avg	max	# calls
Souffleuse	1	5.1	20	157.1	0	0.1	5	6.9
Megane	35	507	1,387	223.4	13	145	632	61.2
Master	27	1,033	2,860	582.2	8	202	924	89.2
Small	1	8.9	93	53.7	0	2.9	18	7.2
Medium	2	12.6	39	42.1	1	6.3	24	14.3
Big	39	877	2,251	514.8	6	247	988	115.1

7. Conclusion

We have analyzed the problems that arise in applications that require the interactive resolution of a constraint problem by a human user. The central notion is global inverse consistency of the network because it ensures that the person who interactively solves the problem is not given the choice to select values that do not lead to solutions. We have shown that deciding, computing, or restoring global inverse consistency, and other related problems are all NP-hard. We have proposed several algorithms for enforcing/maintaining/restoring global inverse consistency and we have shown that the best version is efficient enough to be used in an interactive setting on several configuration and design problems. This is a great advantage compared to existing techniques usually used in configurators. As opposed to techniques maintaining arc consistency, our algorithms give an exact picture of the values remaining feasible. As opposed to compiling offline the problem as a multi-valued decision diagram, our algorithms can deal with constraint networks that change over time (e.g., an extra non-unary constraint posted by a customer who does not want to buy a car with more than 100,000 miles except if it is a Volvo). One direct perspective of this work is to try computing diverse solutions when enforcing GIC. This should allow, on average, to reduce the number of search runs. Techniques such as those developed in [25] might be useful.

Acknowledgements

This work has been funded by the ANR ("Agence Nationale de la Recherche"), project BR4CP (ANR-11-BS02-008). The third author also benefits from the financial support of both CNRS and OSEO (BPI France) within the ISI project 'Pajero'.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.artint.2016.09.001.

References

[1] C. Bessiere, H. Fargier, C. Lecoutre, Global inverse consistency for interactive constraint satisfaction, in: Proceedings of CP'13, 2013, pp. 159–174.
[2] P. Janssen, P. Jégou, B. Nouguier, M. Vilarem, B. Castro, SYNTHIA: assisted design of peptide synthesis plans, New J. Chem. 14 (12) (1990) 969–976.
[3] E. Gelle, R. Weigel, Interactive configuration using constraint satisfaction techniques, in: Proceedings of PACT'96, 1996, pp. 37–44.
[4] J. Amilhastre, H. Fargier, P. Marquis, Consistency restoration and explanations in dynamic CSPs – application to configuration, Artif. Intell. 135 (1–2) (2002) 199–234.
[5] T. Hadzic, H. Andersen, Interactive reconfiguration in power supply restoration, in: Proceedings of CP'05, 2005, pp. 767–771.
[6] T. Hadzic, E. Hansen, B. O'Sullivan, Layer compression in decision diagrams, in: Proceedings of ICTAI'08, 2008, pp. 19–26.
[7] R. Debruyne, C. Bessiere, Domain filtering consistencies, J. Artif. Intell. Res. 14 (2001) 205–230.
[8] E. Freuder, C. Elfe, Neighborhood inverse consistency preprocessing, in: Proceedings of AAAI'96, 1996, pp. 202–208.
[9] D. Martinez, Résolution interactive de problemes de satisfaction de contraintes, Ph.D. thesis, Supaero, Toulouse, 1998.
[10] D. Sabin, E. Freuder, Contradicting conventional wisdom in constraint satisfaction, in: Proceedings of CP'94, 1994, pp. 10–20.
[11] E. Freuder, A sufficient condition for backtrack-bounded search, J. ACM 32 (4) (1985) 755–761.
[12] E. Freuder, Completable representations of constraint satisfaction problems, in: Proceedings of KR'91, 1991, pp. 186–195.
[13] R. Dechter, From local to global consistency, Artif. Intell. 55 (1) (1992) 87–108.
[14] U. Montanari, Network of constraints: fundamental properties and applications to picture processing, Inf. Sci. 7 (1974) 95–132.
[15] J. Astesana, L. Cosserat, H. Fargier, Constraint-based vehicle configuration: a case study, in: Proceedings of ICTAI'10, 2010, pp. 68–75.
[16] C. Papadimitriou, Private communication, 1999.
[17] G. Gottlob, On minimal constraint networks, Artif. Intell. 191–192 (2012) 42–60.
[18] K. Bayer, M. Michalowski, B. Choueiry, C. Knoblock, Reformulating CSPs for scalability with application to geospatial reasoning, in: Proceedings of CP'07, 2007, pp. 164–179.
[19] J. Ullmann, Partition search for non-binary constraint satisfaction, Inf. Sci. 177 (2007) 3639–3678.
[20] C. Lecoutre, STR2: optimized simple tabular reduction for table constraints, Constraints 16 (4) (2011) 341–371.
[21] C. Schulte, Comparing trailing and copying for constraint programming, in: Proceedings of ICLP'99, 1999, pp. 275–289.
[22] M. Ginsberg, Dynamic backtracking, J. Artif. Intell. Res. 1 (1993) 25–46.
[23] C. Lecoutre, C. Likitvivatanavong, R. Yap, STR3: a path-optimal filtering algorithm for table constraints, Artif. Intell. 220 (2015) 1–27.
[24] C. Bessiere, P. Meseguer, E. Freuder, J. Larrosa, On forward checking for non-binary constraint satisfaction, Artif. Intell. 141 (2002) 205–224.
[25] E. Hebrard, B. Hnich, B. O'Sullivan, T. Walsh, Finding diverse and similar solutions in constraint programming, in: Proceedings of AAAI'05, 2005, pp. 372–377.