Constraint Retraction in FD

Philippe Codognet[1] Daniel Diaz[1] Francesca Rossi[2]

[1] INRIA - Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France.
E-mail: {Philippe.Codognet,Daniel.Diaz}@inria.fr
[2] Università di Pisa, Dipartimento di Informatica, Corso Italia 40, 56100 Pisa, Italy.
E-mail: rossi@di.unipi.it

Abstract. The possibility of deleting a piece of information is very convenient in many programming frameworks. However, this feature is not available in constraint languages such as Constraint Logic Programming or Concurrent Constraint Programming, which allow only for a monotonic accumulation of constraints. This is mainly due to its high complexity and also to its non-monotonic nature, which would make such a system much more complex to reason with. In this paper we consider the CLP framework over FD (finite domain) constraints, and we propose an incremental algorithm which deletes a constraint from a set of FD constraints, while maintaining partial arc-consistency. The algorithm follows the chain of dependencies among variables which are set by the nature of the FD constraints, and by doing so it updates only the part of the constraint set which is affected by the deletion. This makes constraint deletion in FD a feasible task that can be efficiently implemented.

1 Introduction

Constraint Logic Programming (CLP) [9, 15] and Concurrent Constraint Programming (CC) [13] are both based on the notion of monotonic refinement of domains of variables. In fact, the computation consists of a monotonic accumulation of constraints. However, it is clear that the possibility of removing constraints could be very useful in many applications, especially those requiring some level of interaction between the system and the user. Nevertheless, extending constraint solvers to non-monotonic behaviors selectively raises a number of difficulties, much similar in nature to those appearing in non-monotonic reasoning. Observe however that we have in CLP a distinction between the logical predicates (or the agents in the CC framework) and the constraints and that we will only consider non-monotonicity in the constraint part, making things somewhat easier.

The main point of this paper is the problem of the incrementality of the operations to be done on constraint deletion, because one cannot be satisfied with an algorithm that recomputes the whole status of the computation from scratch or from the point where the constraint that one wants to delete was introduced (as a traditional CLP system would do, relying on chronological backtracking). Instead, the hope is to get an incremental algorithm which is able to keep as much as possible of the computation already done, redoing only the subcomputations directly depending on the deleted constraint.

In domains where the solver has a global view over the store, i.e. all constraints are always present and accessible, it is possible to delete a constraint and all its effects on variables domains in a single step close to a normal constraint solving step. Instances of global solvers traditionally used in CLP are simplex methods for linear arithmetic constraints over reals or rationals, and boolean solvers based on Binary Decision Diagrams. Incremental deletion of linear arithmetic constraint has recently been studied in [8]. However in domains where the constraint solver has only a local view over the store, i.e. constraints are only accessible through variables on which they depend, constraint deletion is more complex. Propagation techniques like AC [10], AC-4 [11] and AC-5 [17] for finite domains, or unification over first-order terms are instances of such local solvers. Up to now, the deletion of unification constraints in Prolog-like systems has been tackled by so-called selective reset in extended intelligent backtracking methods [3, 7]. However, even if such methods can be extended to finite domain constraints [5, 14], they rely on complex dependency-recording mechanisms that amount to runtime overhead, even if no deletion is later used. Instead, we would like to take advantage of the dependency information already present in the constraint network (and use only this information) for avoiding such forward overhead and rather (re)compute constraint dependencies explicitly in an efficient way whenever a constraint deletion occurs.

This paper presents an algorithm for the incremental deletion of constraints over finite domains in the FD constraint system [16]. More precisely, we extend and adapt traditional propagation techniques to selectively delete, from the current constraint set, a constraint and all its consequences on affected variables, while maintaining the rest of the problem untouched.

The algorithm presented here is close in spirit to the one described in [12]. However, due to the different constraint system considered (in [12] they delete constraints from a set of constraints explicitly represented as sets of allowed tuples, as in classical constraint solving [10]) and to the fact the FD constraints are always kept in a partially arc-consistent state, we have to address different problems in order to obtain an algorithm which achieves the desired deletion.

2 The FD Constraint System

The FD constraint system has been extensively used in many CLP and CC programming systems (like cc(FD) [16], `clp(FD)` [6, 4], or AKL(FD) [2]), the main reason being that it provides the user with the full range of arithmetic and symbolic constraints while being very simple and efficient to handle at the implementation level. For instance, the `clp(FD)` system, a constraint logic programming language based on FD constraints, shows that flexibility and efficiency can indeed be combined, as it is on average several times faster than the CHIP system [1].

A *domain* in FD is a (non empty) finite set of natural numbers (i.e. a *range*). More precisely a range is a subset of $\{0, 1, ..., infinity\}$ where $infinity$ is a particular integer denoting the greatest value that a variable can take. We use the

interval notation $k_1..k_2$ as a shorthand for the set $\{k_1, k_1+1, ..., k_2\}$. From a range r we define $min(r)$ (resp., $max(r)$) as the lower (resp., upper) bound of r. In addition to standard operations on sets (e.g. union, intersection, complementation) we define *pointwise operations* $(+, -, *, /)$ between a range r and an integer i as the set obtained by applying the corresponding operation on each element of d. Finally, $\mathcal{V}_d$ is the set of FD variables, i.e. variables constrained to take a value in a given domain.

Constraints in FD are all of the form X `in` r. Intuitively, a constraint X `in` r enforces X to belong to the range denoted by r that can be not only a *constant range* (e.g. 1..10) but also an *indexical range* using `dom(`Y`)` that represents the whole current domain of Y, or `min(`Y`)` that represents the minimal value of the current domain of Y, or `max(`Y`)` that represents the maximal value of the current domain of Y. When an X `in` r constraint uses an indexical on another variable Y it becomes *store-sensitive* and must be checked each time the domain of Y is updated. A *store* is a finite set of constraints. See [4] for the complete syntax of FD constraints.

Complex constraints such as linear equations and inequations can be defined in terms of the FD constraints. For instance the constraint $X = Y + C$, where C is a constant, is translated (via a clausal notation) into two FD constraints as follows :

```
X=Y+C :- X in min(Y)+C..max(Y)+C,
         Y in min(X)-C..max(X)-C.
```

Observe that this translation has also an operational flavour. In fact, it specifies the type of propagation chosen to solve the constraint: in the above example partial lookahead is used, but full lookahead could have been specified as well by using the `dom(Y)` indexical term (see algorithm *Add* in the next section). The `X in r` constraint can thus be seen as embedding the *core* propagation mechanism for constraint solving over FD.

Let c be a constraint of the form `X in r`. X is called the *constrained variable* of c, denoted by $cv(c)$. We write $V(c)$ for the set of variables on which c depends (i.e. appearing in the indexical range r). Consider a store (set of constraints) S containing c, we note $C(X)$ the set of all constraints depending on X (i.e. $C(X) = \{c' \in S / X \in V(c')\}$). A constraint c is said *static* if $V(c) = \emptyset$. A store S on a set of variables Var is *well-formed* if $\forall X, Y \in Var, ((\exists c \in C(X) \text{ s. t. } Y = cv(c)) \Rightarrow (\exists c' \in C(Y) \text{ s. t. } X = cv(c')))$. We will consider a graph-like description of the store, where variables are nodes and FD constraints are hyperarcs, say c, whose sources are the variable involved in the range (that is, $V(c)$) and whose target is the constrained variable $cv(c)$.

Given a set of FD constraint C, a constraint c *entailed* by C is just a piece of information which is already contained in C. More formally, consider a set of FD constraints C, involving the set of variables V. Let us call $Sol(C)$ the set of all instantiations of variables in V to elements of their domains such that all constraints in C are satisfied. Then, two sets of FD constraints C and C' are said to be *equivalent* if $Sol(C) = Sol(C')$. Consider now a set of FD constraints C over V and a constraint c over $V' \subseteq V$. Then we say that C entails c, written $C \vdash c$, if $Sol(C)$ projected over V' is a subset of $Sol(c)$.

As said above, FD constraints are always kept in a stable state, which can be called *partially arc-consistent* for its resemblance to arc-consistency [10]. More precisely, given a set of FD constraints C, we say that C is partially arc-consistent if, for each constraint `X in r` in C, the domain of X is smaller than or equal to the value of r. For example, the set of FD constraints {`X in 2..10, X in dom(Y)+1, Y in 9..10`} is not partially arc-consistent, since the domain of X contains all values between 2 and 10, whereas the range of the second constraint evaluates to $\{10, 11\}$. Instead, the set of FD constraints {`X in {10}, X in dom(Y)+1, Y in 9..10`} is partially arc-consistent. Observe that we talk about *partial* arc-consistency firstly because of the use of directional constraints (such as `X in dom(Y)+1` in the previous example, which has no `Y in dom(X)-1` counterpart for ensuring full arc-consistency) and, secondly, because of the possible use of the *min* and *max* indexicals. Also, arc-consistency assumes binary constraints while we may have constraints with arity greater than 2.

Observe that both classical arc-consistency [10] and partial arc-consistency for FD constraints do not ensure global consistency (i.e., satisfiability). This makes it possible to have polynomial algorithms, whereas checking satisfiability is NP-complete.

3 Constraint Addition

Whenever a new constraint is added to the current set, the corresponding FD constraints are added to the current set of FD constraints. This makes some variable domain to be restricted (the domain of the constrained variable), and such a restriction propagates to other variable domains through the FD constraints, until stability (that is, partial arc-consistency). Formally, for any variable X, let us call D_X the current value of the domain of X. Then the algorithm which achieves stability over a set C of FD constraints, called $Add(C)$ in the following, can be described as follows.

```
1. forall c ∈ C do push(c)
2. while stack not empty do
3.       c := pop
4.       let c = X in r
5.       compute r in the current store
6.       D_X-old := D_X
7.       D_X := D_X ∩ r
8.       if D_X-old ≠ D_X
9.           then forall c' ∈ C(X) do push(c')
```

In words, this algorithm considers each constraint, say `X in r`, to see whether it can be used to reduce the domain of X. If so, then such a domain is reduced (line 7) and all constraints depending on X are put into the stack to be considered (line 9). For example, assume we have the following constraints: c_1 = `X in dom(Y)-1`, c_2 = `X in 1..10`, c_3 = `Y in dom(Z)-1`, c_4 = `Y in 1..10` and c_5 = `Z in 10..20`. Then, by considering c_3, the domain of Y is restricted to contain

only 9 and 10, thus: `Y in 9..10`. Then, by considering c_1 (and the newly induced domain of Y), we get `X in 8..9`. Thus the stable situation is the one in which c_2 is replaced by `X in 8..9` and c_4 by `Y in 9..10`. Note that c_5 remains the same, although it is clear that there could be no solution if Z is different from 10 or 11, because there is no indexical constraint which constraints Z.

Theorem 1 (termination). *Given a set of FD constraints C, algorithm $Add(C)$ terminates in a finite number of steps.*

Proof. The statement of the theorem follows from the fact that all domains are finite and can only decrease in size, and that thus the condition in line 8 can be satisfied only a finite number of times.□

Theorem 2 (equivalence). *Given a set of FD constraints C, consider the store $C' = Add(C)$. Take now any constraint c. Then, $C \vdash c$ if and only if $C' \vdash c$.*

Proof. The only changes performed by algorithm *Add* to the given store C are effect of the execution of line 7, which possibly reduces the domain of a variable, say X, to the elements which are in the current range of a constraint c constraining X. Thus, the elements which are removed from D_X are values that, if given to X, would violate some of the constraints (at least c). This means that the set of variable instantiations which satisfy all the constraints in C remains the same after the application of the algorithm. Since a constraint entailed by a store S is just a subset of the instantiations allowed by all the constraints in S, also such subsets will be the same.□

Theorem 3 (order independence). *Given a set of FD constraints C, consider the application of algorithm Add to C via a certain strategy S_1 to push constraints into the stack, resulting in a store C_1. Consider also the application of the same algorithm to C via a different strategy S_2, resulting in a store C_2. Then we have that $C_1 = C_2$.*

Proof. It is easy to see that the operation of domain reduction is a closure operator. That is, it is idempotent (that is, by performing the same reduction on the same domain twice nothing changes), extensive (that is, by reducing a domain we always get a smaller domain), and monotone (that is, if we start from a smaller domain, then we get a smaller result). Classical results on closure operators state that they are characterized by their fixed points and not by the way such points are reached. Thus the statements of the theorem.□

Theorem 4 (partial arc-consistency). *Given a set of FD constraints C and the set $C' = Add(C)$, C' is partially arc-consistent.*

Proof. This follows immediately from the fact that line 7 of the algorithm continues to restrict the domain of a variable constrained by a constraint c to the values that are in the range of c until no more restrictions can be done. Thus, when the algorithm terminates, for any constraint c, we have that its range cannot be smaller than the domain of the variable constrained by c.□

The algorithm above is not incremental, that is, it achieves stability over a set of constraints without considering which constraint has been added more recently and whether the set of constraints was in a stable state before its addition. An incremental version which assumes to start from a stable set of FD constraints C and a new constraint to be added, say $c =$ `X in r`, can be obtained by the algorithm above by replacing line 1 with the following: `push(`c`)`. The new algorithm is thus a function with two arguments, and thus will be referred to in the following by $Add(C, c)$. In this way the domain reduction is initiated by constraint c and is propagated through the graph via adjacent constraints, and constraints which have nothing to do with X are never considered. For this incremental algorithm, all properties proved above still hold, except for the one proved by Theorem 4, which now holds only if we start from a partially arc-consistent store. In more general terms, one could consider the addition of more than one constraint at a time. In this case the algorithm $Add(C, S)$, where C and S are sets of constraints, starts by pushing all the constraints in S in the queue, that is, line 1 of $Add(C)$ becomes: **forall** $c \in S$ **do** `push(`c`)`.

Time Complexity. Assume that m is the number of constraints in C, d the maximal cardinality of a variable domain, a the maximal number of variables a constraint might depend on (that is, its arity minus one), and k the maximal cardinality of $C(X)$ for any X. The **while** loop can be executed as many times as constraints can be pushed in the stack. A constraint c will be pushed only if the domain of one of the variables it depends on (i.e. X such that $c \in C(X)$) has been modified. This can happen only d times at most for each variable. As a constraint can only depend on a variables at most, it can be pushed into the stack only $a \times d$ times. Therefore the **while** loop can be executed at most $m \times a \times d$ times. Within each execution of the loop, the only operations that can require more than a constant time are the computation of the range and the pushing of new constraints in the stack. At most k constraints can be pushed. The complexity of the range computation depends on the syntax of the range: if only `min` and `max` indexicals are used, then the operation is constant, if instead one `dom` indexical range is used, then it may require d steps[3]. Considering a and k as constants, the complexity of algorithm $Add(C)$ is therefore either $O(md)$ if only min/max indexicals are used or $O(md^2)$ in the general case. As predictable, the worst-case complexity of our algorithm is the same as that of algorithm AC-5 [17] on comparable constraints. For the more general $Add(C, S)$ algorithm, the complexity is the same, considering m as $|C| + |S|$. Although in the average case the work to be done by this incremental algorithm will be less (because C is already stable), in the worst case the addition of a single constraint can trigger recomputations in the whole graph.

[3] If one considers an extended syntax where two `dom` indexical ranges are allowed within a range, then the range computation becomes quadratic in d. It can be shown that using more than two `dom` indexicals does not bring extra expressive power.

4 Constraint Deletion

Consider a constraint c of the form `X in r` to be removed in a store S. If $V(c) \neq \emptyset$ (that is, some variables are involved in its range), c may have been activated several times during the computation, e.g. each time the domain of any variable in $V(c)$ is updated (via algorithm *Add* of the previous section), leading to several reduction of D_X, the domain of X. Also, each reduction of D_X may have (re)activated all the constraints in $C(X)$, leading to further domain reductions for other variables. All these reductions – consequences of c –, and only these ones, have to be undone when c is deleted.

To achieve this goal, first we recompute the domain of X in the store which is the current one minus c. This could lead to a larger domain for X, if the restriction imposed by c over X is not imposed by any other constraint. Then we propagate this new domain for X through all constraints in $C(X)$. This leads to recompute new domains for all variables Y such that $Y = cv(c')$ for some $c' \in C(X)$. Thus now we have to deal with this variables as we do with X, that is, we enlarge their domain and we propagate such enlargement through $C(Y)$. This phase ends when a stable state (w.r.t. domain enlargement) is reached. The previous steps could have enlarged the domains too much if we desire a partially arc-consistent state. Thus a subsequent phase where domains are reduced due to the restrictions imposed by the currently present constraints is needed. This computation again ends when a stable state (w.r.t. domain modification) is reached. Note that this second phase can be implemented via algorithm *Add*. Since the domain enlargement phase is performed through the constraint network as long as domains should be enlarged, while it stops when reaching variables whose domain is not affected by the deletion, we thus achieve our objective of minimal recomputation of the store, as parts of the network independent of the deletion are not reconsidered.

Our algorithm, called $Del(C, c)$, deletes a constraint c of the form `X in r` from a set of FD constraints C. The algorithm works on the set of constraints $C - \{c\}$, and C is assumed to be partially arc-consistent. Two stacks S and S' are used: S to push constraints in the enlarging phase, and S' to push constraints pointing to an enlarged variable and will be used in the later restricting phase. Enlarged variables are recognized via a marking.

```
1.  D_new := D_X^s - (r - D_X)
2.  if D_X ≠ D_new
3.      then D_X := D_new
4.           forall c' ∈ C(X) do push(c') in S
5.           while S not empty do
6.               c := pop from S
7.               let c = Y in r
8.               compute r in the current store
9.               D_Y-old := D_Y
10.              D_Y := r ∩ D_Y^s
11.              if D_Y-old ≠ D_Y
```

12. **then** `mark` Y
13. **forall** $c' \in C(Y)$ **do** `push(`c'`) in` S
14. **forall** $c' \in C(Y)$ `s.t.` $cv(c')$ `is marked` **do** `push(`c'`) in` S'
15. `Add'(`$C - \{c\}, S'$`)`

Line 1 of the algorithm computes the domain of X in the store $C - \{c\}$. This is done by taking the initial value of such domain (D^s_X) and eliminating from it those elements that are not in D_X because of constraints other than c. In fact, consider a partially arc-consistent set of FD constraints and a constraint `X in r` in such a set. If $D_X \neq r$, then it has to be properly smaller by definition of partial arc-consistency. Let us then take the non-empty set $S = (r - D_X)$. The elements in S are elements which are not present in D_X but which would be allowed by constraint c. Thus their absence from D_X is caused by some other constraint $c' \neq c$. One could just write $D_X := D^s_X$ and the algorithm would produce the same result, but less efficiently. In fact, in that case, the values in (r - D_X) would for sure be deleted later by the algorithm, since some of the other constraints (and not the removed one) are responsible for their deletion from D_X. If the domain of X is not changed, then we have nothing to do, otherwise (line 2) we set the new domain of X (line 3) and start the enlarging phase. Line 4 pushes into the stack all those constraints which depend on X (that is, which have X in their range expression). Line 5 is the starting point of a while statement which will end only when the stack S is empty, that is, there are no more constraints to reconsider for the enlarging phase. Line 6 and 7 extract one constraint from the stack S and give a shape to the popped constraint. Say it is Y in r. Line 8 recomputes r in the current store. Due to the enlarged domain of X, r may now have a different value than before. Line 9 records the current domain of Y (D_Y) in the variable D_Y-old. Line 10 enlarges the domain of Y to the newly computed value of r, being careful however to not enlarge D_Y more than its initial value. If the domain of Y has been actually enlarged (line 11), we also have to propagate the effect of this enlargement to other variables. More precisely, we have to mark Y (line 12) to record that it has been enlarged and push into the stack S all those constraints which depend on Y (line 13). If instead the domain has not changed (line 14), then we have to add to the stack S' all constraints c' which depend on Y and constrain enlarged variables. Line 15 takes this stack S' and use it to achieve partial arc-consistency and reach a stable state. In fact the enlarged domains might have been enlarged too much and this phase is indeed required if we desire a partially arc-consistent state. This is achieved via the application of algorithm *Add'*, which is identical to *Add*, except that it only pushes into the stack constraints pointing to marked variables. In fact the other variables in the graph have not been enlarged and this part of the graph is therefore still partially arc-consistent.

Theorem 5 (termination). *Given any set of FD constraints C, and any constraint $c \in C$, algorithm $Del(C, c)$ terminates after a finite number of steps.*

Proof. Since the initial domains of all the variables are finite, each variable can be enlarged only a finite number of times. Each enlarging of a variable domain

pushes a finite number of constraints into the stack S. Thus, after a finite number of steps S will be empty, and therefore the **while** loop will be over. Also from Theorem 1 we know that line 15 takes a finite amount of time. Thus, $Del(C, c)$ terminates in finite time. □

Theorem 6 (equivalence). *Given a set of FD constraints C and a constraint $c \in C$, consider $C' = Del(C, c)$. Then $Sol(C - \{c\}) = Sol(C')$, where by $C - \{c\}$ we mean the set of FD constraints C where c has been deleted and all variables have been restored to their initial domain.*

Proof. We already know, by Theorem 2, that algorithm *Add* does not change the set of solutions. Therefore neither will line 15 of algorithm *Del*. The only other changes that are made during algorithm *Del* are in lines 1 and 10, when domains are enlarged. The domains are not restored to their initial value but set to the intersection of their initial (static) value and a range (see line 10). Values that are hence excluded are not allowed by that range and therefore cannot participate to any solution. Let us now consider the variables whose domain is not changed by *Del*. The first case is a variable that is considered in the **while** loop, but its domain is not enlarged by line 10. Then the same reasoning as above ensures that the excluded values cannot participate to any solution. The last case concerns a variable that is not even considered for enlarging. This implies that all the variables adjacent to it have not been enlarged. Therefore all current ranges of constraints pointing to such a variable are the same as those that have lead to the domain of that variable (since we assume that we start from a partially arc-consistent state). Hence all the values that are not present in this domain are not allowed by the current ranges and thus cannot participate to any solution. □

Theorem 7 (partial arc-consistency). *Given a set of FD constraints C and a constraint $c \in C$, consider $C' = Del(C, c)$. If C is partially arc-consistent, then C' is so.*

Proof. This follows from the fact that line 15 of algorithm *Del* is an application of *Add*, restricted to the part of the graph which has been changed (enlarged variables). As the rest of the graph has not been changed by lines 1 to 14, it is still partially arc-consistent, as assumed. □

Theorem 8 (order independence). *Given a set of FD constraints C and a constraint $c \in C$, consider the application of algorithm $Del(C, c)$ via a certain strategy S_1 to push constraints into S, resulting in a store C_1. Consider also the application of the same algorithm to C via a different strategy S_2, resulting in a store C_2. Then we have that $C_1 = C_2$.*

Proof. Stack S' is only used in line 14, which behaves like *Add*. Thus by Theorem 3 we know that the order in which constraints are pushed into or popped from such stack is not important. Consider now stack S. It is easy to see that the operation of domain enlargement is a closure operator. That is, it is idempotent (that is, by performing the same enlargement on the same domain twice

nothing changes), extensive (that is, by enlarging a domain we always get a bigger domain), and monotone (that is, if we start from a bigger domain, we get a bigger result). Classical results on closure operators state that they are characterized by their fixed points and not by the way such points are reached. Thus the statements of the theorem holds.□

Since algorithm Del implements the deletion of a constraint c from a set of FD constraints C, and produces the new set of FD constraints C' (we recall that a domain is again an FD constraint), it would be nice if one could prove that the information given by c is not present in C' any longer. That is, that $C' \not\vdash c$. However, this is unfortunately not true. In fact, consider the store containing the constraints $X = Y + 1$, $Y = Z + 1$ and $X = Z + 2$ (actually, the store contains their FD counterparts), and assume we delete $X = Z + 2$ via algorithm Del. Then it is possible to see that the resulting store still entails the information that X is equal to Z plus 2. This behaviour however is not due to the fact that Del does not work well, but that the deleted constraint was indeed redundant, that is, it was entailed by the other constraints. Thus it is reasonable that its deletion does not change the set of constraints entailed by the store. Note that this behavior occurs also in other constraint deletion algorithms, like the one for classical constraint solving described in [12]. More precisely, the formal general statement of the behaviour of Del w.r.t. entailment is as follows: given a set of FD constraints C and two constraints $c, c' \in C$ (possibly $c = c'$), consider the set $C' = Del(C, c)$. Then $C' \vdash c'$ if and only if $(C - \{c\}) \vdash c'$. The proof of this statement follows from the definition of entailment and from Theorem 6.

Time Complexity. It is easy to see that lines 1 to 13 have the same structure as algorithm Add and therefore a similar complexity formula. However, let us call m' the number of constraints considered in this part of Del. It is easy to see that this number is bounded by the number of constraints in the subgraph EG involving the enlarged variables and their immediate neighbors. Thus the complexity of lines 1 to 14 is either $O(m'd)$ if only min/max indexicals are used or $O(m'd^2)$ in the general case. Note that line 14 only adds $O(k)$ to each execution of the **while** loop and thus does not change the complexity. The last line of algorithm Del is $Add'(C - \{c\}, S')$, that is, an application of algorithm Add to graph EG (observe that constraints in S' are in EG and that the definition in Add' forbid to go out from EG). Thus its complexity is again either $O(m'd)$ if only min/max indexicals are used or $O(m'd^2)$ in the general case. Therefore the complexity of the overall Del algorithm is $O(m'd)$ in the restricted case and $O(m'd^2)$ in the general case. Note however that, even if in the worst-case m' is equal to m, most of the time it will be much smaller. The complexity of a batch algorithm, which will rebuilt $C - \{c\}$ from scratch and apply partial arc-consistency (with the Add algorithm) is always $O(md)$ in the restricted case and $O(md^2)$ in the general case.

Example: Consider the following set of constraints :

{ X in 1..10, Y in 1..20, Z in 1..10, U in 1..10, V in 1..10,

$$X \geq Y \ , \ X = Z+1 \ , \ X \neq 5 \ , \ Y = Z+U \ , \ Y \geq V \ \}$$

This translates in terms of FD constraints as follows :

X in 1..10, Y in 1..20, Z in 1..10, U in 1..10, V in 1..10,

X in min(Y)..infinity,	(1)
Y in 0..max(X).	(1')
X in dom(Z)+1,	(2)
Z in dom(X)-1.	(2')
X in -{5}	(3)
Z in min(Y)-max(U)..max(Y)-min(U),	(4)
U in min(Y)-max(Z)..max(Y)-min(Z),	(4')
Y in min(Z)+min(U)..max(Z)+max(U).	(4")
Y in min(V)..infinity,	(5)
V in 0..max(Y).	(5')

Let us first apply the *Add* algorithm for achieving partial arc-consistency. This reduces the domain of X, Y, Z, U and V as follows :

$$\{ \text{ X in 2..4:6..10, Y in 2..10, Z in 1..3:5..9, U in 1..9, V in 1..10 } \}$$

Assume now that constraint X≠5 is deleted. As the domain of X is enlarged to 2..10 (cf. beginning of algorithm Del), X is marked and all constraints depending on X (that is, 1' and 2') have to be reconsider and pushed to S. Let us consider that 1' is popped first and recomputed. It does not change the value of Y, therefore only constraint 1 (because it constrains X) is reconsidered among those depending on Y (that is, 1, 4' and 5'), and therefore pushed into S'. Let us now turn to 2'. This leads to the enlarging of Z, which is restored to 1..9; this variable is marked. Therefore all constraints depending on Z (that is, 2, 4' and 4") have to be reconsidered and pushed in S. Also 2 is pushed into S' because X is marked. Recomputing constraint 2 does not change the domain of X (but 2' is pushed in S' because Z is marked), neither does recomputing 4' for U (but 4 is pushed in S' because Z is marked) nor does recomputing 4" for Y (pushing nothing more in S', because 4 is already present). However reconsidering 4 does not change Z and no more constraints are pushed in S, that is then empty. Hence the first phase of $Del(X \neq 5)$ terminates. The second phase will recompute constraints in S' (i.e. 1, 2, 2' and 4), and restore consistency for marked variables, i.e. the domain of X will be reduced to 2..10 (by 2) and that of Z to 1..9 (by 2') Those changes will not be propagated to variables outside the marked subgraph because the rest of the graph is already stable. The second phase of $Del(X \neq 5)$ therefore terminates and the final constraint store is :

$$\{ \text{ X in 2..10, Y in 2..10, Z in 1..9, U in 1..9, V in 1..10 } \}$$

It is worth noticing that it has not recomputed 5 nor 5', because Y and U have not been affected by the deletion of X≠5. If there have been a larger part of the constraint network linked to Y and U, it would have been kept untouched as well. This means that the recomputations involved by the constraint deletion are kept small, localized to the affected part of the network.

References

1. A. Aggoun and N. Beldiceanu. Overview of the CHIP Compiler System. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press, 1993.
2. B. Carlson, M. Carlsson and S. Janson. Finite Domain Constraints in AKL(FD). In *Proceedings of ILPS 94*, MIT Press, 1994.
3. C. Codognet and P. Codognet. Non-deterministic Stream AND-Parallelism based on Intelligent Backtracking. In *Proceedings of 6th ICLP*, Lisbon, 1989. The MIT Press.
4. P. Codognet and D. Diaz. Compiling Constraints in `clp(FD)`. *Journal of Logic Programming*, vol. 27(3), 1996.
5. P. Codognet, F. Fages and T. Sola. A meta-level compiler for CLP(FD) and its combination with intelligent backtracking. In *Constraint Logic Programming : Selected Research*, A. Colmerauer, F. Benhamou (Eds.), MIT Press, 1993.
6. D. Diaz and P. Codognet. A minimal extension of the WAM for CLP(FD). In *proceedings of the 10th International Conference on Logic Programming*, D. S. Warren (Ed.), Budapest, Hungary, MIT Press 1993.
7. W. S. Havens. Intelligent Backtracking in the Echidna Constraint Logic Programming System. Research Rep. CSS-IS TR 92-12, Simon Fraser University, Vancouver, Canada, 1992.
8. T. Huynh and K. Marriott. Incremental Constraint Deletion in Systems of Linear Constraints. Draft Report, IBM T. J. Watson Research Center, 1992.
9. J. Jaffar and J.L. Lassez. Constraint Logic Programming. In *Proceedings of POPL'87*, ACM Press, 1987.
10. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, vol.8, n.1, 1977.
11. B. A. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence 5 (1989).*
12. B. Neveu and P. Berlandier. Maintaining Arc Consistency through Constraint Retraction. *Proc. TAI94*, IEEE Press.
13. V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.
14. T. Sola. Deduction Maintenance in Constraint Logic Programs. Ph.D. thesis, University of Paris XI, December 1995.
15. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.
16. P. Van Hentenryck, V. Saraswat and Y. Deville. Constraint processing in cc(FD). In *Constraint Programming : Basics and Trends*, A. Podelski (Ed.), LNCS 910, Springer Verlag 1995. First version: Research Report, Brown University, Jan. 1992.
17. P. Van Hentenryck, Y. Deville and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence 57 (1992)*, pp 291-321.

This article was processed using the LaTeX macro package with LLNCS style