Intermediate Consistencies by Delaying Expensive Propagators

Andreï Legtchenko, Arnaud Lallouet, AbdelAli Ed-Dbali
Université d'Orléans – LIFO
BP 6759 – F-45067 Orléans – France

Abstract

What makes a good consistency ? Depending on the constraint, it may be a good pruning power or a low computational cost. By "weakening" arc-consistency, we propose to define new automatically generated solvers which form a sequence of consistencies weaker than arc-consistency. The method presented in this paper exploits a form of regularity in the cloud of constraint solutions: the density of solution orthogonal to a projection. This approach is illustrated on the sparse constraints "to be a n-letters english word" and crossword CSPs, where interesting speed-ups are obtained.

Introduction

Since their introduction (Mackworth 1977), CSP consistencies have been recognized as one of the most powerful tool to strengthen search mechanisms. Since then, their considerable pruning power has motivated a lot of efforts to find new consistencies and to improve the algorithms to compute them.

Consistencies can be partially ordered according to their pruning power. However, this pruning power should be put into balance with the complexity of enforcing them. For example, path-consistency is often not worth it: its pruning power is great, but the price to pay is high. Maintaining path-consistency during search is thus often beaten in practice by weaker consistencies. Similarly, on many useful CSPs, bound-consistency is faster than arc-consistency even if it does not remove values inside the variables' domains: this is left to the search mechanism ensuring the completeness of constraint solving. Moreover, the overall performance of a solver is also determined by the interaction between the consistency and the search strategy. Powerful consistencies are not always the best choice. It is sometimes more interesting to find the optimal ratio between the advantage of pruning and the computational cost needed to enforce it. The absence of prediction of the potential power of a consistency comes from two sources: consistencies exploit subtle properties which are often incomparable and they do not always use the same data-structures.

Recently, there has been a great interest in the automatic building of consistency operators and in finding ad-hoc efficient constraint representations. First (Apt & Monfroy 2001) and (Abdennadher & Rigotti 2003) use CHRs (Frühwirth 1998) as a target language to build a solver. Powerful propagators may be discovered in this framework but at the price of a high complexity which limits the method to constraints of modest arities and rather small domains. In (Dao *et al.* 2002) has been introduced an approximation framework we also use in this paper to approximate bound-consistency. This work has been extended in (Lallouet *et al.* 2003) for arc-consistency: a constraint is approximated by a set of covering blocks obtained by a clustering algorithm and only these blocks are used in the reduction process. But unfortunately, not all constraints are suitable for clustering. It is needed that the constraint has locally dense areas which can be agglomerated meaningfully. Finding new representations to compute a consistency has also been tackled in (Barták & Mecl 2003) and (Cheng, Lee, & Stuckey 2003): they use clustering techniques to find an exact representation of the constraint. Hence the resulting consistency is the full arc-consistency and speed-ups are difficult to obtain while competing with highly optimized implementations. In (Monfroy 1999), the idea of using weaker functions has been proposed as a preprocessing to speed-up the computation of a classical consistency (in this case, box-consistency). From the search mechanism point of view, this is an internal recipe to compute the consistency. In contrast, our operators define on purpose a weaker consistency which is used during search.

In this paper, we contribute to this line of work by providing a range of custom consistencies intermediate between bound- and arc-consistency for sparse constraints which have no dense areas. First, these constraints exist: we take as example the n-ary constraint "to be a n-letter english word". There is absolutely no block regularity and the words are sparsely distributed in the solution space. For example, by using the 4176 five-letters words of `/usr/dict/words`, we get a density of 0,035%. A clustering algorithm would either consider a large number of clusters, limiting the interest of the representation, or filter a too few number of non-solution out of the search space, yielding a very weak consistency.

Our method is as follows. For a given constraint, we express the arc-consistency as a set of particular *elementary reduction functions*. Each elementary function is only con-

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

cerned in the withdrawal of one value in a variable's domain. Depending on the number of supports for the target value, these functions do not have the same computational cost. Our technique consists in applying the computationally cheap operators as a consistency during the reduction phase and delaying the expensive ones until instantiation of all variables. For this we split the set of functions in two: functions smaller than a given threshold are iterated and longer ones are delayed. The closure of all the operators defines a new consistency for a given constraint, weaker and sometimes quicker than arc-consistency.

Intuitively, the cost of an elementary reduction is directly related to the number of supports present in the hyperplane orthogonal to the target value. For example, in figure 1 representing cuts in the solution space of a constraint, the value a for the variable X is supported by a large number of tuples: in order to eliminate a from X's domain, we have to check all these tuples. On the other side, the value b in figure 2 is only supported by a few tuples. But since a is well supported, it is more likely part of a solution than b. Hence we spend a large work trying to eliminate a value which is likely part of a solution. We propose to postpone the evaluation of a's function until instanciation of all variables (i.e. on satisfiability test) while actively trying to eliminate b by consistency. It could be argued that algorithms like AC6 which compute lazily the set of supports could be faster, but since the resulting consistency is weaker than AC, the technique could apply to any AC algorithm. The main difficulty rely in determining the best threshold.

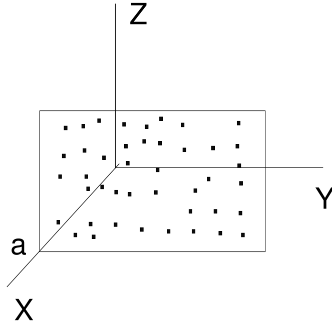

Figure 1: Dense hyperplane in solution space

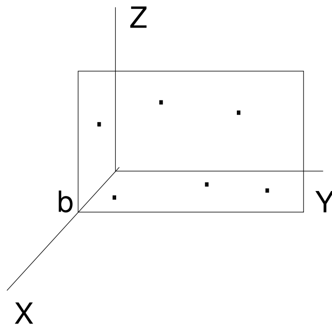

Figure 2: Sparse hyperplane in solution space

The paper is structured as follows:

- *A framework to express consistencies.* Consistencies are usually built as global CSP properties. But it is now rather common to express them modularly by the greatest fixpoint of a set of operators associated to the constraints, which is computed using a chaotic iteration (Apt 1999). We present a framework which allows, starting from arbitrary operators, to progressively add properties in order to build a consistency.
- *A consistency construction method.* For a given constraint, we express the arc-consistency as a set of particular reduction operators. These operators do not have the same computational cost. Expensive ones are delayed until instantiation of all variables. The closure of all the operators defines a new consistency for a given constraint, weaker but quicker than arc-consistency.
- *An example.* The interest of these consistencies is shown by the "word" constraints and the crossword CPSs.

An approximation framework

In this section, we present a general approximation scheme for consistencies. Let V be a set of variables and $D = (D_X)_{X \in V}$ their (finite) domains. For $W \subseteq V$, we denote by D^W the set of tuples on W, namely $\Pi_{X \in W} D_X$. Projection of a tuple or a set of tuples on a variable or a set of variables is denoted by $|$, natural join of two sets of tuples is denoted by $\bowtie$. If A is a set, then $\mathcal{P}(A)$ denotes its powerset and $|A|$ its cardinal.

Definition 1 (Constraint) *A* constraint *c is a pair (W, T) where:*

- *$W \subseteq V$ is the* arity *of the constraint c and is denoted by $var(c)$.*
- *$T \subseteq D^W$ is the set of* solutions *of c and is denoted by $sol(c)$.*

The *join* of two constraints is defined as a natural extension of the join of tuples: $c \bowtie c' = (var(c) \cup var(c'), sol(c) \bowtie sol(c'))$.

A *CSP* is a set of constraints. Join is naturally extended to CSPs and the *solutions* of a CSP C are $sol(\bowtie C)$. A direct computation of this join is too expensive to be tractable, especially when considering that it needs to represent tuples of the CSP's arity. This is why a framework based on approximations is preferred, the most successful of them being the domain reduction scheme where variable domains are the only reduced constraints. So, for $W \subseteq V$, a search state consists in a set of yet possible values for each variable: $s_W = (s_X)_{X \in W}$ such that $s_X \subseteq D_X$. The search space is $S_W = \Pi_{X \in W} \mathcal{P}(D_X)$. The set S_W, ordered by pointwise inclusion $\subseteq$, is a complete lattice. Likewise, union and intersection on search states are defined pointwise. The whole search space S_V is simply denoted by S.

Some search states we call *singletonic* play a special role in our framework. A singletonic search state comprises a single value for each variable, and hence represents a single

tuple. A tuple is promoted to a singletonic search state by the operator $\lceil\ \rceil$: for $t \in D^W$, let $\lceil t \rceil = (\{t_X\})_{X \in W} \in S_W$. This notation is extended to a set of tuples: for $E \subseteq D^W$, let $\lceil E \rceil = \{\lceil t \rceil \mid t \in E\} \subseteq S_W$. Conversely, a search state is converted into the set of tuples it represents by taking its cartesian product Π : for $s \in S_W$, $\Pi s = \Pi_{X \in W} s_X \subseteq D^W$. We denote by $Sing_W$ the set $\lceil D^W \rceil$ of singletonic search states. By definition, $\lceil D^W \rceil \subseteq S_W$.

A consistency is generally described by a property $Cons \subseteq S$ which holds for certain search states and is classically modeled by the common greatest fixpoint of a set of operators associated to the constraints. By extension, in this paper, we call *consistency* for a constraint c an operator on S_W having some properties which are introduced in the rest of this section. Let f be an operator on S_W. We denote by $Fix(f)$ the set of fixpoints of f which define the set of *consistent states* according to f.

For $W \subseteq W' \subseteq V$, an operator f on S_W can be *extended* to f' on $S_{W'}$ by taking: $\forall s \in S_{W'}, f'(s) = s'$ with $\forall X \in W' \backslash W, s'_X = s_X$ and $\forall X \in W, s'_X = f(s|_W)_X$. Then $s \in Fix(f') \Leftrightarrow s|_W \in Fix(f)$. This extension is useful for the operator to be combined with others at the level of a CSP.

In order for an operator to be related to a constraint, we need to ensure that it is contracting and that no solution tuple could be rejected anywhere in the search space. An operator having such property is called a *preconsistency*:

Definition 2 (Preconsistency) *An operator $f : S_W \to S_W$ is a* preconsistency *for $c = (W, T)$ if:*

- *f is monotonic, i.e. $\forall s, s' \in S_W, s \subseteq s' \Rightarrow f(s) \subseteq f(s')$.*
- *f is contracting, i.e. $\forall s \in S_W, f(s) \subseteq s$.*
- *f is correct, i.e. $\forall s \in S_W, \Pi s \cap sol(c) \subseteq \Pi f(s) \cap sol(c)$.*

In the last property, the second inclusion is also called *correctness* of the operator with respect to the constraint; it means that if a state contains a solution tuple, this one will not be eliminated by consistency. Since a preconsistency is also contracting, this inclusion is actually also an equality.

An operator on S_W is *associated* to a constraint $c = (W, T)$ if its singletonic fixpoints represent the constraint's solution tuples T:

Definition 3 (Associated Operator) *An operator $f : S_W \to S_W$ is* associated *to a constraint c if $Fix(f) \cap Sing_W = \lceil sol(c) \rceil$*

However, nothing is said about its behavior on non-singletonic states. This property is also called *singleton completeness*. Note that a preconsistency is not automatically associated to its constraint since the set of its singletonic fixpoints may be larger. When it coincides, we call such an operator a *consistency*:

Definition 4 (Consistency) *An operator f is a* consistency *for c if it is associated to c and it is a preconsistency for c.*

Note that a consistency can be viewed as an extension to S_W of the satisfiability test made on singletonic states. Consistency operators can be easily scheduled by a chaotic iteration algorithm (Apt 1999). By the singleton completeness property, the consistency check for a candidate tuple can be done by the propagation mechanism itself. Let $C = \{c_1, \ldots, c_n\}$ be a CSP and $F = \{f_1, \ldots, f_n\}$ be a set of consistencies on S associated respectively to $\{c_1, \ldots, c_n\}$. If all constraints are not defined on the same set of variables, it is always possible to use the extension of the operators on the union of all variables which appear in the constraints. The common closure of the operators of F can be computed by a chaotic iteration (Apt 1999). It follows from the main confluence theorem of chaotic iterations that a consistency can be constituted by combining the mutual strengths of several operators. Since we have $Fix(F) = \bigcap_{f \in F} Fix(f)$ and since each consistency does preserve the tuples of its associated constraint, the computed closure of all operators associated to the CSP C does not reject a tuple of $c_1 \bowtie \ldots \bowtie c_n$ for any search state $s \in S$ because of an operator of F.

Proposition 5 (Composition) *The composition of two preconsistencies for a constraint c via chaotic iteration is still a preconsistency for c.*

The proof is straightforward by (Apt 1999). We call $\odot$ the composition of two operators via chaotic iteration. The same property holds for consistencies instead of preconsistencies.

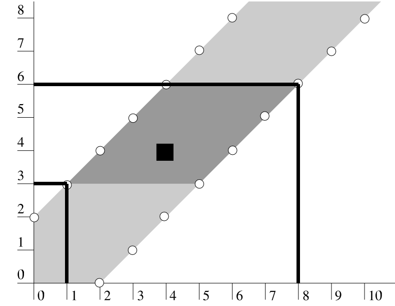

Figure 3: Preconsistency for $Y = X \pm 2$

Exemple 6 *Consider the constraint c given by $Y = X \pm 2$ and depicted by the white dots in figure 3. The following operators:*

```
X in min(Y)-2 .. max(Y)+2
Y in min(X)-2 .. max(X)+2
```

define a preconsistency for c since they are correct with respect to c: they do not reject a tuple which belongs to the constraint. For example, the dark grey area shows the evaluation of the first operator for X when $Y \in [3..6]$. However, the operators are not associated to c since they accept more tuples than necessary, like for example $(4, 4)$ indicated by the black square. Actually, they accept the light grey area in figure 3 and are a consistency for the constraint $X - 2 \leq Y \leq X + 2$.

Let us now define two consistencies associated to a constraint c:

- ID_c is a family of contracting operators such that any $id_c \in ID_c$ verify: $\forall s \in S_W \backslash Sing_W$, $id_c(s) = s$ and $\forall s \in Sing_W, s \in \lceil sol(c) \rceil \Leftrightarrow id_c(s) = s$. In particular, on non-solution singletonic states, id_c reduces at least one variable's domain to $\emptyset$. The non-uniqueness of id_c comes from the fact that all search states s such that $\Pi s = \emptyset$ represent the empty set of solution for a constraint. In the following, we denote by id_c any member of ID_c.
- ac_c is the well-known arc-consistency operator defined by $\forall s \in S_W, ac_c(s) = ((sol(c) \cap \Pi s)|_X)_{X \in W}$.

The goal of an automatic construction procedure is to build for a constraint a set of operators which can ensure a desired level of consistency when included in a chaotic iteration. We can compare two operators f_1 and f_2 by their reductions over the search space:

Definition 7 *An operator f_1 is* stronger *than f_2, denoted by $f_1 \subseteq f_2$ if $\forall s \in S, f_1(s) \subseteq f_2(s)$.*

For example, it is well-known that arc-consistency is stronger than bound-consistency which in turn is stronger than id_c. The notion of preconsistency is interesting in the context of CSP resolution because preconsistency operators have the desired properties to be included in a chaotic iteration. Since they are weaker than consistencies, they are easier to construct automatically. But since they may accept some non-solution singletonic states, it is needed to find a technique to ensure completeness. The framework we propose is to build a fast but incomplete preconsistency and to add *reparation* operators which are delayed until the satisfiability test.

Delaying "long" operators

In order to build operators for an intermediate consistency, we start by giving the expression of arc-consistency with a set of special function we call *elementary reduction functions*. Then we weaken the arc-consistency by delaying a set of functions which are supposed to be too expensive until instantiation of all variables. We use as a mesure of expensiveness the length of the syntactic expression of the function. Hence the reduction power decreases, but the computation becomes quicker. When the choice of the length threshold vary, it defines a sequence of comparable consistencies.

Definition 8 (Support) *Let $c = (W, T)$ be a constraint, $X \in W$ and $a \in D_X$. We call support of "$X = a$" a tuple $t \in sol(c)$ such that $t_X = a$.*

Exemple 9 *Let* `c(X,Y,Z)` *be a constraint. $D_X = D_Y = \{0, 1\}$ and $D_Z = \{0, 1, 2\}$.*

c :

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1
1	1	2
0	1	2

The sets of supports for Z are:
$T_{Z=0} = \{(0, 0, 0), (0, 1, 0), (1, 0, 0)\}$
$T_{Z=1} = \{(1, 1, 1)\}$
$T_{Z=2} = \{(1, 1, 2), (0, 1, 2)\}$

We call $T_{X=a} \subseteq sol(c)$ the set of all supports of $X = a$. A value a has to be maintained in the current domain of X only if we have at least one $t \in T_{X=a}$ such that all projections on $Y \in W \backslash \{X\}$ are included in s_Y.

Definition 10 (Supp property) *Let $c = (W, T)$ be a constraint, $X \in W$ and $a \in D_X$. We call $\texttt{Supp}_{X=a}(s)$ the following property:*

$$\bigvee_{t \in T_{X=a}} \Big(\bigwedge_{Y \in W \backslash \{X\}} t_Y \in s_Y \Big)$$

The value $X = a$ is supported if $\texttt{Supp}_{X=a}(s) = true$.

Exemple 11 (Example 9 running) *For all $s \in S_{\{X,Y,Z\}}$, we have:*
$\texttt{Supp}_{Z=0}(s) = (0 \in s_X \wedge 0 \in s_Y) \vee (0 \in s_X \wedge 1 \in s_Y) \vee (1 \in s_X \wedge 0 \in s_Y)$
$\texttt{Supp}_{Z=1}(s) = 1 \in s_X \wedge 1 \in s_Y$
$\texttt{Supp}_{Z=2}(s) = (1 \in s_X \wedge 1 \in s_Y) \vee (0 \in s_X \wedge 1 \in s_Y)$.

If $X = a$ is not supported, then a does not participate to any solution of the CSP and therefore can be eliminated. With this notion, we define a function called *elementary reduction function* which concerns only one value in the variable's domain. Each value in the initial domain of each variable has its own elementary reduction function. If a value must be eliminated, its function returns this value as a singleton, and the empty set otherwise.

Definition 12 (Elementary reduction function) *For all $X \in W$ and for all $a \in D_X$, the* elementary reduction function *associated to $X = a$ is $r_{X=a} : S_W \longrightarrow \mathcal{P}(D_X)$ given by*

$$\forall s \in S_W,\ r_{X=a}(s) = \begin{cases} \{a\},\ if\ \neg\texttt{Supp}_{\texttt{X=a}}(\texttt{s}) \\ \emptyset,\ otherwise \end{cases}$$

We call *s-size* (support size) of the elementary reduction function $r_{X=a}$ the cardinality of $T_{X=a}$. Arc-consistency can be defined using elementary reduction functions as follows: $\forall s \in S_W, ac_c(s) = (s_X \backslash \bigcup_{a \in D_X} r_{X=a}(s))_{X \in W}$.

We want a consistency quicker then arc-consistency, even if it is less powerful. For this, we choose a threshold and split arc-consistency into two operators. The first one, called `Short`, is composed of all elementary reduction functions having their s-size less than the chosen threshold:

Definition 13 (Operator Short) *Let c be a constraint, and n an integer. The operator $\texttt{Short}_c(n) : S_W \longrightarrow S_W$ is given by: $\forall s \in S_W$,*
$\texttt{Short}_c(n)(s) = (s_X \backslash \bigcup_{a \in D_X,\ |T_{X=a}| \leq n} r_{X=a}(s))_{X \in W}$.

The second operator, called `Long`, is composed of all other elementary reduction functions, which are fired only on singletonic states for completeness:

Definition 14 (Operator Long) *Let c be a constraint and n an integer. The operator $\texttt{Long}_c(n) : S_W \longrightarrow S_W$ is given by: $\forall s \in S_W$,*

$$\texttt{Long}_c(n)(s) = \begin{cases} (s_X \backslash \bigcup_{a \in D_X,\ |T_{X=a}| > n} r_{X=a}(s))_{X \in W}, \\ \quad if\ s \in Sing_W \\ s,\ otherwise \end{cases}$$

This operator is used only to reject non-solution tuples.

Exemple 15 (Example 11 running) *For the variable Z, we can make two operators following the definition of $\texttt{Short}_c$ and $\texttt{Long}_c$. Let set the threshold to 2. The operators for Z are:*

$$\begin{array}{rl} \texttt{Short}_c(2)_Z: & s_Z \longrightarrow s_Z \backslash (r_{Z=1}(s) \cup r_{Z=2}(s)) \\ \texttt{Long}_c(2)_Z: & s_Z \longrightarrow s_Z \backslash r_{Z=0}(s) \end{array}$$

The operator $\texttt{Long}_c(2)_Z$ is delayed because its reduction power is small (only one value can be eliminated), and the effort to compute $r_{Z=0}(s)$ is judged too high. This operator is fired only if $s \in Sing_{\{X,Y,Z\}}$.

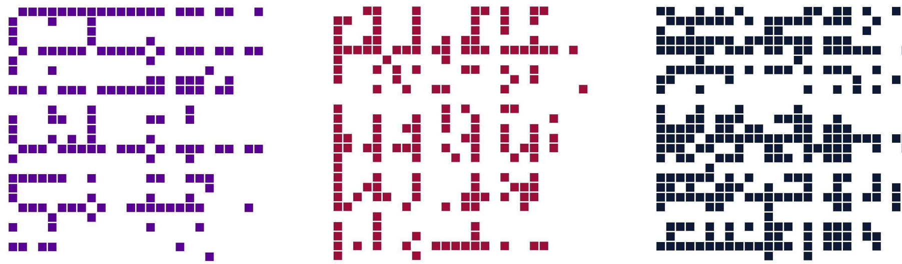

Figure 4: Projections of $word_3(X,Y,Z)$ on the planes XY, YZ, XZ

Proposition 16 *For all integer n, $\texttt{Short}_c(n)$ is a preconsistency for c.*

Proof First, we show that $\texttt{Short}_c(n)$ is monotonic. Let s and s' in S_W such that $s \subseteq s'$. Then, $\forall X \in W, \forall a \in D_X$, $\texttt{Supp}_{X=a}(s) \Rightarrow \texttt{Supp}_{X=a}(s')$. From which it follows that $\forall X \in W, \forall a \in D_X, \neg\texttt{Supp}_{X=a}(s') \Rightarrow \neg\texttt{Supp}_{X=a}(s)$. In the case of s', there are less values to eliminate than in the case of s. So $\texttt{Short}_c(n)(s) \subseteq \texttt{Short}_c(n)(s')$. The operator $\texttt{Short}_c(n)$ is monotonic.

$\texttt{Short}_c(n)$ is contracting by construction. It it also correct by construction: it eliminates less values than arc-consistency. Hence $\texttt{Short}_c(n)$ is a preconsistency. □

The operator $\texttt{Long}_c(n)$ is also a preconsistency, the proof is similar to $\texttt{Short}_c(n)$. When both operators are in a chaotic iteration, we get a consistency:

Proposition 17 *Let $c = (W,T)$ a constraint and n an integer. The composition $\texttt{Short}_c(n) \odot \texttt{Long}_c(n)$ is a consistency for c .*

Proof According to the proposition 5, $\texttt{Short}_c(n) \odot \texttt{Long}_c(n)$ is a preconsistency. But $\forall s \in Sing_W$, $\texttt{Short}_c(n) \odot \texttt{Long}_c(n)(s) = ac_c(s)$, so $\texttt{Short}_c(n) \odot \texttt{Long}_c(n)$ is associated to c. Therefore $\texttt{Short}_c(n) \odot \texttt{Long}_c(n)$ is a consistency for c. □

Elementary reduction functions of $\texttt{Long}_c(n)$ are not fired on non-singletonic states. If the threshold n is not too big, the set of elementary reduction functions of $\texttt{Long}_c(n)$ is not empty. In that case, the consistency $\texttt{Short}_c(n) \odot \texttt{Long}_c(n)$ is weaker than arc-consistency, but it can be computed faster, since that we have less functions to evaluate. If n is large enough, all elementary reduction functions are in $\texttt{Short}_c(n)$ and we get the same reduction power and computational cost as arc-consistency.

Implementation and Example

Implementation. A system generating the operators $\texttt{Short}_c(n)$ and $\texttt{Long}_c(n)$ has been implemented. The language used to express the operators is the indexical language (van Hentenryck, Saraswat, & Deville 1991) of GNU-Prolog (Diaz & Codognet 2001). An indexical operator is written "`X in r`" where `X` is the name of a variable, and `r` is a range of possible values for `X` and which may depend on other variables' current domains. If we call x the current domain of X, the indexical `X in r` can be read declaratively as the second-order constraint $x \subseteq r$ or operationally as the operator $x \mapsto x \cap r$. For a given constraint and a threshold, our system returns two sets of $|W|$ indexical operators, i.e. one for each variable. The first set defines the $\texttt{Long}_c(n)$ operator, and the second $\texttt{Short}_c(n)$. In total, we have $2 * |W|$ indexical operators for a constraint $c = (W,T)$. The closure by chaotic iteration of all $2 * |W|$ operators is equivalent to the consistency $\texttt{Short}_c(n) \odot \texttt{Long}_c(n)$. The indexical operators for $\texttt{Long}_c(n)$ are delayed with a special trigger `val` in the indexical language which delays the operator until a given variable is instantiated. This is how we get that these operators are not iterated on non-singletonic search states. This delay is obtained at no cost because Gnu-Prolog uses separate queues. In all cases, the generation time is about one second.

$X_{1,1}$	$X_{1,2}$	■	$X_{1,4}$	$X_{1,5}$	$X_{1,6}$	$X_{1,7}$
$X_{2,1}$	■	$X_{2,3}$	$X_{2,4}$	$X_{2,5}$	■	$X_{2,7}$
$X_{3,1}$	$X_{3,2}$	$X_{3,3}$	■	$X_{3,5}$	$X_{3,6}$	$X_{3,7}$
$X_{4,1}$	$X_{4,2}$	$X_{4,3}$	■	■	$X_{4,6}$	■
■	$X_{5,2}$	$X_{5,3}$	$X_{5,4}$	$X_{5,5}$	■	$X_{5,7}$
$X_{6,1}$	$X_{6,2}$	$X_{6,3}$	■	$X_{6,5}$	$X_{6,6}$	$X_{6,7}$
$X_{7,1}$	$X_{7,2}$	■	$X_{7,4}$	$X_{7,5}$	$X_{7,6}$	$X_{7,7}$

Table 1: A model for 7x7 grid

Example. The sparse distribution described in figure 1 and 2 occurs, for example, in the case of the constraints $word_n(X_1, X_2, \ldots, X_n)$. For $n = 3$, the constraint $word_3(X,Y,Z)$ means that XYZ is a 3-letters english word. We consider that domains are ordered by lexicographic ordering. In figure 4 is presented the projections of $word_3(X,Y,Z)$ on different planes. When using UNIX dictionary `/usr/dict/words`, this constraint has 576 solutions. The constraints $word_4(X,Y,Z,U)$ (with 2236 solutions) and $word_5(X,Y,Z,U,V)$ (with 4176 solutions) have the same regularity.

The CSP we use consists in finding a solution for crossword grids of different sizes. The CSP is composed only by a set of constraints $word_n$. The domain of all variables is $\{a, .., z\}$. An example of a 7x7 grid and its model are presented table 1.

Only the first solution is computed. Some benchmarks are presented in the tables 2, 3, 4. The full reduction power of arc-consistency is obtained for the following thresholds:

Threshold for				Time
$word_2$	$word_3$	$word_4$	$word_5$	
8	30	50	10	280ms
3	70	240	510	45ms
1	5	300	10	11ms
3	6	400	10	16ms
1	5	200	10	120ms
`fd_relation`				53ms

Table 2: Some results for 7x7 grid

Threshold for				Time
$word_2$	$word_3$	$word_4$	$word_5$	
3	70	240	310	380ms
3	30	80	200	>15min
3	70	240	410	58ms
3	70	260	410	76ms
3	65	240	410	78ms
`fd_relation`				102ms

Table 3: Some results for 10x10 grid

8 for $word_2$, 130 for $word_3$, 500 for $word_4$, 1200 for $word_5$. It means that with these thresholds (and higher), the $\texttt{Long}_c(n)$ operator is empty. The `fd_relation` time is the computation time with the built-in GNU-Prolog predicate implementing arc-consistency for a constraint given by the table of its solutions. These new consistencies show interesting speed-ups, from 1.68 to 4.8. It appears that the best threshold to use depends on the CSP in which the constraint is put. We did compute static thresholds giving the best average ration between pruning and computational cost, but this method did not extend to CSPs and results were not foreseeable. Depending on the CSP, a constraint can be used in different part of its solution space, thus requiring a different tunig of the threshold. The main contribution of this approach so far is to show the interest of using constraint regularities to construct efficient operators. We are currently investigating how to help the user to determine the best threshold.

Conclusion

In this paper, we propose a new method which allows to build a full range of consistencies weaker but quicker than arc-consistency. In this approach, we exploit a form of regularity in the constraint solutions to construct a set of operators. The operators which are too expensive to compute are delayed, so the closure of the remaining ones by a chaotic iteration defines a new consistency, weaker but quicker than arc-consistency. The interest of this method is illustrated on crossword CSPs.

Further work can be to consider other representations for other regularities. For every representation, there should exists suitable constraints and other for which it is unadequate. It also includes a way to determine dynamically the parameters at run-time in order to exploit all parts of the constraint optimaly.

Threshold for				Time
$word_2$	$word_3$	$word_4$	$word_5$	
1	5	50	310	1h
3	130	500	600	193ms
3	50	240	510	6490ms
3	5	240	310	5490ms
3	70	260	510	160ms
3	100	240	510	160ms
3	130	300	510	170ms
1	70	240	510	150ms
`fd_relation`				252ms

Table 4: Some results for 15x15 grid

References

Abdennadher, S., and Rigotti, C. 2003. Automatic generation of rule-based constraint solvers over finite domains. *Transaction on Computational Logic*. accepted for publication.

Apt, K., and Monfroy, E. 2001. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming* 1(6):713 – 750.

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

Barták, R., and Mecl, R. 2003. Implementing propagators for tabular constraints. In Krzysztof R. Apt, Francois Fages, F. R. P. S., and Váncza, J., eds., *Joint Annual Workshop of the ERCIM Working Group on Constraints and the CoLogNET area on Constraint and Logic Programming*, 69–83.

Cheng, K. C.; Lee, J. H.; and Stuckey, P. J. 2003. Box constraint collections for adhoc constraints. In Rossi, F., ed., *International Conference on Principles and Practice of Constraint Programming*, volume 2833 of *LNCS*, 214–228. Kinsale, County Cork, IE: Springer.

Dao, T.-B.-H.; Lallouet, A.; Legtchenko, A.; and Martin, L. 2002. Indexical-based solver learning. In van Hentenryck, P., ed., *International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *LNCS*, 541–555. Ithaca, NY, USA: Springer.

Diaz, D., and Codognet, P. 2001. Design and implementation of the Gnu-Prolog system. *Journal of Functional and Logic Programming* 2001(6).

Frühwirth, T. 1998. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming* 37(1-3):95–138.

Lallouet, A.; Legtchenko, A.; Dao, T.-B.-H.; and Ed-Dbali, A. 2003. Intermediate (learned) consistencies. In Rossi, F., ed., *International Conference on Principles and Practice of Constraint Programming*, number 2833 in LNCS, 889–893. Kinsale, County Cork, Ireland: Springer. Poster.

Mackworth, A. K. 1977. Consistency in networks of relations. *Artificial Intelligence* 8(1):99–118.

Monfroy, E. 1999. Using weaker functions for constraint propagation over real numbers. In *Symposium on Applied Computing*.

van Hentenryck, P.; Saraswat, V.; and Deville, Y. 1991. Constraint processing in cc(fd). draft.