UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

Intermediate (Learned) Consistencies

Arnaud Lallouet, Andreï Legtchenko,
Thi-Bich-Hanh Dao, AbdelAli Ed-Dbali
Université d'Orléans, LIFO

Rapport N^o **2003-04**

Intermediate (Learned) Consistencies

Arnaud Lallouet, Andreï Legtchenko, Thi-Bich-Hanh Dao, 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 using machine learning techniques (search in an hypothesis space and clustering), we propose to define new automatically generated solvers which form a sequence of consistencies intermediate between bound- and arc-consistency. By associating to a consistency its *quality*, we define a method which allows, from a set of consistencies, to determine the best one to enforce for a given constraint. We apply this method on our generated intermediate consistencies. Surprisingly, traditional consistencies are not always the best choice and experiments show that learned intermediate consistencies sometimes outperform standard ones on various benchmarks according to the prediction.

1 Introduction

Since their introduction [19], 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 dig holes in the variables domains: this is left to the search mechanism ensuring the completeness of constraint solving.

What shall we conclude ? Weaker is quicker, or pruningful is beautiful ? Actually, it just depends on the constraints. Up to our best knowledge, the investigation of the best consistency to use for a constraint has been in the literature left to an experimental evaluation. For example, it is known that bound-consistency performs well for linear constraints, RPC [5] or singleton consistencies [10] for large random constraint networks, etc ... Many of these works are summarized in [11]. 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, it has been shown that consistencies can be built automatically using machine learning techniques. In [9], a consistency weaker than bound-consistency but as close to it as possible was constructed. The technical improvements described in this paper allow to build a full range of comparable consistencies for a given constraint. These *intermediate consistencies* are located between bound- and arc-consistency. We propose a unified parameter to compare them and the main conjecture of the paper is that the performance of a consistency is related to this parameter.

The contributions of this paper are the following:

- *A framework to express approximations.* 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 [3]. We present a framework which allows, starting from arbitrary operators, to progressively add properties in order to build a consistency.
- *A consistency evaluation parameter.* Rather than being ordered only by their strength, we propose to relate the pruning power of a consistency to its computational complexity by a formal parameter we call *quality*. We conjecture that when this parameter is maximal, the obtained consistency has the best performance since the ratio between pruning and computational cost is optimal. The main advantage is that the quality is evaluated *statically*, before the introduction of the constraint in any CSP.
- *A consistency construction method.* In order to implement this notion, we need a set of comparable consistencies. Fortunately, this set is provided by a new solver learning method based on a clustering of the constraint's tuples, a sampling of the search space and a reparation technique able to fix a too weak operator in order to obtain a consistency. We call this consistency *multibound-consistency*. The pruning power and the computational cost is directly related to the number of allowed clusters. The best consistency is chosen by optimizing the quality parameter.
- *An experimental validation of the conjecture.* According to the prediction given by the quality, the optimal consistency is for example bound-consistency for very simple constraints like linear constraints, and arc-consistency for very sparse and irregular ones. But the tests show that on semi-regular constraints, the best performance is obtained for an intermediate consistency which is able to perform the most important part of the work of arc-consistency while saving enough tests to be faster.

In section 2, we present our approximation framework which is the basis of our solver learning method, in section 3, we present the intermediate consistencies and their evaluation, in section 4, we present our techniques for automatic solver generation and in section 5, we give some examples showing the interest of our approach.

2 Consistency and Approximation

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* solution *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 (see [3, 16] for a more general framework). 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 is a subset of 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 $f : S_W \rightarrow S_W$ 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. $\forall s \in S_W, f(s) \subseteq s$.*
- *$\forall s \in S_W, \mathit{\Pi} s \cap sol(c) \subseteq \mathit{\Pi} f(s) \cap sol(c)$.*

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 $f : S_W \rightarrow S_W$ 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 [3]. 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 [3]. 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 [3]. The same property holds for consistencies instead of preconsistencies.

Example 6. Consider the constraint c given by $Y = X \pm 2$ and depicted by the dark blue dots in figure 1. The following operators:

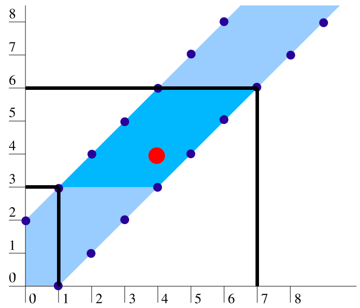

Fig. 1. Preconsistency for $Y = X \pm 2$

```
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 blue 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 red dot. Actually, they accept the light blue area in figure 1 and are a consistency for the constraint $X - 2 \leq Y \leq X + 2$.

Let us now define some 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 $\forall s \in S_W, ac_c(s) = s'$ with $\forall X \in W$, $s'_X = s_X \cap sol(c)|_X$.

We now suppose that each variable domain D_X is equipped with a total ordering $\leq$. The notation $[a..b]$ is used for the classical notion of interval $\{e \in D_X \mid a \leq e \leq b\}$. We call Int_X the interval lattice built on D_X and for $W \subseteq V$, $Int_W = \Pi_{X \in W} Int_X$. For $A \subseteq D_X$, we denote by $[A]$ the set $[\min(A)..\max(A)]$. We extend this notation for any search state: $\forall s \in S_W, [s] = \Pi_{X \in W}[s_X]$. Bound-consistency consists in contracting only the bounds of a variable's domain, represented as an interval:

- bc_c : is the bound-consistency operator defined by $\forall s \in S_W, bc_c(s) = s'$ with $\forall X \in W$, $s'_X = s_X \cap [sol(c)|_X]$. Since $\cap$ on S_W is pointwise, we can also write $bc_c(s) = s \cap [sol(c)]$.

3 Intermediate Consistencies

With respect to a constraint, correctness and singleton completeness are the only two requirements for an operator to be a consistency. In this section, we present a new evaluation method able to compare different consistencies.

Usually, consistencies are compared by considering their pruning power, which is described by the following ordering:

Definition 7 (Precision ordering). *Let f and f' be two operators on S_W and let $E \subseteq S_W$. We say that f is* more precise *than f' on E, denoted by $f \subseteq_E f'$ if $\forall s \in E, f(s) \subseteq f'(s)$. If $E = S_W$, we simply write $f \subseteq f'$.*

For example, it is well-known that $ac_c \subseteq bc_c$. Note that this ordering is not complete and some consistencies are not comparable with others (see for example [11] for a rather extensive survey of this topic).

Proposition 8. *The consistency ac_c is the most precise consistency for c.*

Proof. Suppose that there exists f a consistency for c such that $f \subset ac_c$. Since f is contracting, $\exists s \in S, \exists X \in W, \exists v_X \in D_X$ such that $v_X \in ac_c(s)_X$ and $v_X \notin f(s)_X$. But $ac_c(s)_X = s_X \cap sol(c)_X$. Hence $\exists t \in sol(c)$ such that $t_X = v_X$ and $\lceil t \rceil \subseteq s$. But, since f is a preconsistency, we have $f(\lceil t \rceil) = \lceil t \rceil$ and in particular $f(\lceil t \rceil)_X = \{v_X\}$. But since $v_X \notin f(s)_X$, this contradicts the fact that f is monotonic.

The precision ordering is not always sufficient since for example a consistency for $X = Y$ can be compared to a consistency for, say $X \leq Y$ or $X \geq Y$. The following notion of approximation compares consistencies for the same constraint:

Definition 9 (Approximation). *Let f and f' be two consistencies on S_W for a constraint c. The consistency f'* approximates *f if $f \subseteq f'$ and $f' \subseteq_{Sing_W} f$.*

For example, bc_c is an approximation of ac_c. Consistencies which are approximations of ac_c or even bc_c are what we call *intermediate consistencies*. They define the same constraint but perform different prunings along the search space.

An other tool to compare the pruning power of consistencies is to compare them on average: a consistency f is better than an other one f' if it rejects more tuples on the whole search space. Since arc-consistency is the best consistency which can be devised for a constraint taken outside of a CSP (proposition 8), we define formally the *pruning power* of a preconsistency f from the number of extra tuples maintained by f with respect to arc-consistency over the whole search space. Since $ac_c \subseteq f$, it is enough to assume that f is a preconsistency:

Definition 10 (Pruning power of a preconsistency). *Let f be a consistency on S_W for a constraint c. The* pruning power *of f is given by:*

$$P(f) = \frac{1}{1 + \frac{\sum_{s \in S_W}(|\Pi f(s)| - |\Pi ac_c(s)|)}{|S_W|}}$$

In this formula, $\sum_{s \in S_W}(|\Pi f(s)| - |\Pi ac_c(s)|)$ is a kind of distance between f and arc-consistency which measure the number of tuples "missed" by f and which are removed by arc-consistency. Thus, ac_c has a pruning power of 1, and $P(bc_c) \leq 1$. The pruning power defines a total ordering on consistencies.

The pruning power of a consistency is a very important feature, but it must be put into balance with the computational cost needed to enforce it. This remark is not new and has motivated the in-depth study of the complexity of the consistencies and of the algorithms enforcing them. On the algorithmic side, bound consistency is often better than arc-consistency when the size of the problem is small. Conversely, the usefulness of strong consistencies has been shown for large problems, witness the MAC *versus* FC controversy [6], and the more recent consistencies intermediate between arc- and path-consistency [11].

Traditionally, finding a new consistency has been seen as finding a new (and increasingly subtle) mathematical property for a CSP and a good algorithm to enforce it, both aspects being as important because a good property associated to a bad algorithm could not be recognized as useful as it should be. The validation of the interest of a consistency is always done purely experimentally on real or random CSPs. But who can really foresee if a given consistency will be useful for a particular problem or a particular constraint ? Let us define the *complexity* of a preconsistency:

Definition 11 (Complexity of a preconsistency). *Let c be a constraint and f a preconsistency for c. The* complexity *of f, denoted by $C(f)$ is the sum of the number of operations necessary to compute f on every $s \in S_W$.*

To be meaningful, it is of course needed to agree on the basic operations allowed to compute the consistencies. Note that the complexity is evaluated on the whole search space and is not a function of the size of the data.

We propose a new (up to our best knowledge) consistency evaluation mechanism which can help to tailor the best possible consistency for a given constraint. The idea is to relate the pruning power of a consistency to the complexity of its evaluation measured by the number of operations necessary to enforce it:

Definition 12 (Quality of a preconsistency). *Let c be a constraint and f a preconsistency for c. The* quality *of f is $Q(f) = P(f)/C(f)$.*

The conjecture we make is the following:

Conjecture 13. For a given constraint, the best consistency operator is the one of highest quality.

The quality may be evaluated for any constraint and we can expect that the curve should be approximately the one depicted in figure 2. However, depending on the constraint, the optimum may be at (or before) bound-consistency (BC), at arc-consistency (AC), or somewhere in the middle. The quality could also be evaluated for CSPs, possibly showing the optimum after arc-consistency.

4 Consistency learning

In this section, we summarize our previous work on consistency learning [9], we add sampling and reparation (briefly presented in [17]) and we present a new class of approximations based on clustering.

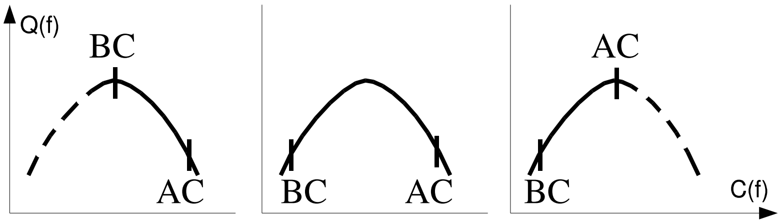

Fig. 2. Expected shape of the curve $Q(f)$ in function of $C(f)$.

The learning problem. In order to build a consistency, we need to define an hypothesis space in which search is performed. Let d be a distance between two operators, let $\mathcal{L}$ be a language in which the operators are expressed and let $[\![.]\!]$ be an interpretation for this language such that every term $l \in \mathcal{L}$ evaluates into an operator $[\![l]\!]$. The problem of learning an operator for a consistency $cons_c$ amounts in finding a term $l_0 \in \mathcal{L}$ such that $l_0 = \min_{l \in \mathcal{L}} d(cons_c, [\![l]\!])$. But only minimizing this distance does not ensure that the learned operator is actually an approximation of $cons_c$. We must also ensure the correctness $cons_c \subseteq [\![l]\!]$. Hence the learning problem is stated as follows:

$$\begin{array}{rl} \texttt{minimize} & d(cons_c, [\![l]\!]) \\ \texttt{subject to} & \forall s \in S_W, cons_c(s) \subseteq [\![l]\!](s) \subseteq s \end{array}$$

The correctness of the algorithm is ensured by construction. Following machine learning vocabulary, $cons_c$ represents the *example* space and $\mathcal{L}$ the *hypothesis* space. In [9], we instantiated this framework with the following parameters:

- the consistency to be learned is the bound-consistency on intervals: $bc_c(s) = s \cap [sol(c)]$.
- we use the indexical language [21] as implemented in Gnu-Prolog [13]. In this language, an operator is written "`X in r`" in which `X` actually represents the domain of the variable X and `r`, called the range, is an expression representing a subset of D_X which possibly involves references to other variables' current domain such as `min(Y)` or `max(Z)`. One indexical is learned for each variable: `X in min`$_r$ `..` `max`$_r$.
- each bound of the indexical is learned separately and its expression has a fixed form. For example, for a constraint c of arity W, the linear form for X is given by $a + \sum_{Y \in W \setminus \{X\}} (b_Y . \min(Y) + c_Y . \max(Y))$.
- for the `min`$_r$ part, the distance d is given by the sum of the differences between bound- and the learned consistency $\sum_{s \in S_W} (\min(bc_c(s)|_X) - \min([\![l]\!](s)|_X))$. We use the opposite difference for `max`$_r$.

Reparation. Building directly a consistency is difficult because the expression of such a consistency may not be possible in the language in which the operators are expressed. It may also be possible but at the price of using very long and tricky expressions. However, in order to have a good efficiency, the generated expression must be quite simple to evaluate. It happens that ensuring a good pruning over the search space and singleton completeness are often two antagonist features.

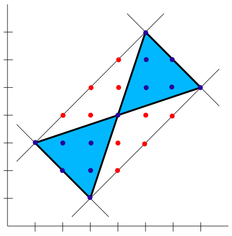

Fig. 3. Linear approximation of the "butterfly" constraint.

Example 14. For example, the "butterfly" constraint drawn in blue in figure 3 cannot be fully represented by using the linear form of indexicals because of its non-convex form. The best linear approximation is represented by the light lines but there is no way to exclude the red dots using this kind of operators.

Using proposition 5, we can build a consistency by combining the mutual strengths of several operators. In particular, this feature is used in the indexical language [21] where each indexical defines a preconsistency for the constraint (because each operator only concerns one variable). The full consistency is obtained for the whole set of operators defining a constraint.

We learn a preconsistency instead of a consistency because enforcing correction on the whole search space and at the same time completeness on singleton states is difficult, and even not always possible within a given language. Indeed, for example, the indexical language of Gnu-Prolog has been tailored for its classical built-in constraints. A full consistency can be reconstituted by combining the consistency id_c which is associated to c but has the weakest reduction strength with the learned preconsistency. We do not care about the rejection of each non-solution tuple but we try to get as close as possible to the desired consistency along the whole search space.

An implementation of id_c in the indexical language of Gnu-Prolog using the primitive delayed evaluation mechanism `val` and a combination of C tests and operators is quite easy. For example, `(val(X) == 3) && (val(Y) == 2)` accepts the tuple $(3, 2)$. The direct C compilation performed by Gnu-Prolog ensures a good performance of this part of the code, even if it could be obviously improved by a dichotomic search or hatching function.

Sampling method. One of the biggest problem encountered in the search of a good operator is that the correctness constraints have to be verified along the whole search space S_W. Thus the evaluation of a single candidate operator needs a traversal of a potentially huge space, limiting the interest of the method for small domains and arities. Fortunately, monotonic operators have an other interesting property: they preserve the solution tuples of their associated constraint across the search space:

Proposition 15. *Let f be a monotonic and contracting operator such that $\lceil sol(c) \rceil \subseteq Fix(f)$. Then f is a preconsistency for c.*

Proof. Suppose that f is not a preconsistency. Then $\exists s \in S$ and $t \in sol(c)$ such that $t \in \Pi s$ and $t \notin \Pi f(s)$, or equivalently $\lceil t \rceil \subseteq s$ and $\lceil t \rceil \not\subseteq f(s)$. Since $t \in sol(c)$ and $\lceil sol(c) \rceil \subseteq Fix(f)$, we have $\{t\} = \Pi f(\lceil t \rceil)$. It yields that $\lceil t \rceil \subseteq f(\lceil t \rceil)$ and $\lceil t \rceil \not\subseteq f(s)$. Hence we have $\lceil t \rceil \subseteq s$ and $f(\lceil t \rceil) \not\subseteq f(s)$, which contradicts the fact that f is monotonic.

By this proposition, if we can ensure that the constructed operator is monotonic, there is no need to explore the example space S_W to ensure the correction constraints. It is sufficient to be correct with respect to the singletonic solution states $\lceil sol(c) \rceil$. This is the basis of our *sampling method*.

In theory, when using monotonic operators, checking correctness on the set of singletonic states representing the constraint's solutions is sufficient. In practice, the consistency yield by a so small search space is too close to id_c to be useful and we need to take a superset E of $\lceil sol(c) \rceil$ for the learning task in order to have both correctness and a good convergence. However, it allows to reduce considerably the search space which must be checked to verify if an operator is a preconsistency. Good operators are very quickly obtained, especially when learning linear indexicals: besides $\lceil sol(c) \rceil$, only less than a hundred samples are enough. This allows to apply the method on real-world constraints. For example, a constraint of arity of 5 with a domain size of 100 values yields, only for bound-consistency, a search space of size 3.10^{18} which could not be handled without this method. Moreover, monotonic operators are required to be included in a chaotic iteration computation [3].

Clustering and multibound-consistency One drawback of learning linear or quadratic indexicals is that a small syntactic change — adding 1 to a coefficient for example — may yield a considerably different evaluation of the operator. Moreover, the dimension of complexity is not explored because all linear operators have the same computational cost. The new class of operators we propose allows to optimize the quality: they have different pruning power and this pruning power is proportional to their computational cost.

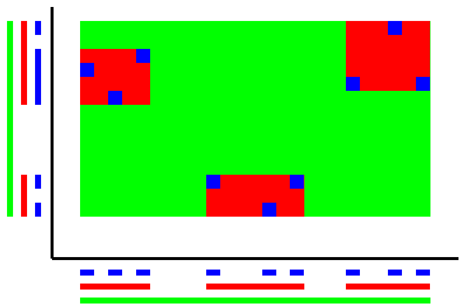

Fig. 4. A constraint as seen by arc, multibound- and bound-consistency.

The idea is to isolate disjunctive chunks of the constraint and to apply bound consistency on each chunk. We call this consistency *multibound-consistency*. The expected property is that the largest holes are dug between the chunks while the small ones are encapsulated and explored by the enclosing search mechanism. This intuition is figured out in figure 4 where the blue dots represent the constraint tuples, the red squares represent the clusters of grouped tuples and the green rectangle shows the constraint as seen by bound-consistency. On each side, the diagram shows in respective color the reduction performed by each consistency. A constraint is cut off into pieces by a clustering algorithm. Technically, a clustering is just a partition of the constraint into subsets:

Definition 16 (Clustering). *Let $c = (W, T)$ be a constraint. A clustering of c is a set of constraints $CL = \{c_1 = (W, T_1), \ldots, c_n = (W, T_n)\}$ such that $\{T_1, \ldots, T_n\}$ form a partition of T.*

There is no universal clustering algorithm because each one is obtained as the optimization of a certain criterion. We use the agglomerative complete-link clustering algorithm [15]. Since our constraints are of numeric nature, the tuples are interpreted as points in a n-dimensional space and the distance used is the Euclidean distance. The algorithm starts by considering each tuple as a cluster, then it evaluates the distance between each pair of clusters. When the clusters have more than one tuple, the distance between two clusters is the distance between their most distant tuples. At each iteration, the pair of closest clusters is agglomerated. The algorithm stops when the desired number of clusters is obtained. This is a simple and greedy algorithm in $O(n^3)$. Note that the clustering algorithm is only a parameter of our framework and should be adapted to fit the constraint as much as possible. The consistency obtained consists in applying the bound-consistency on each separate cluster:

Definition 17 (Multibound-consistency). *Let c be a constraint and CL be a clustering of c. Let bc_{cl} be the bound-consistency for a constraint cl. The* multibound-consistency *is defined by the operator:*

$$\forall s \in S_W, \quad mb_c(s) = s \cap \bigcup_{cl \in CL} bc_{cl}(s)$$

Note that when each cluster only holds one tuple, we get arc-consistency, and when all tuples are in a single cluster, we get bound-consistency. The set of multibound-consistencies form a chain in the precision ordering. The bound-consistency on each cluster can be learned by the preceding method: a search for the best expression in a given language. For the sake of simplicity and the rapidity of evaluation, we have chosen to simplify further the form of the indexical bounds by taking only constant bounds. The approximation of each cluster is thus only its bounding box. In this case, the learning algorithm is maximally simplified since no additional examples besides $\lceil sol(c) \rceil$ are needed. This simplicity allows to proceed very large constraints with several thousands of tuples (see section 5). Since the clustering wraps some holes, it is needed to compose the learned operator with id_c in order to get a consistency. The multibound-consistency operator is obviously monotonic.

Optimization of the quality parameter is simply done by a steepest descent across consistencies with all number of clusters. The pruning power is approximated by evaluation on a random sample of the search space instead of S_W (we substitute S_W by $E \subseteq S_W$ in definition 10). The complexity is linear in function of the number of clusters.

The implementation of this consistency in Gnu-Prolog is not native in the indexical language since it does not include flexible guards. This is why we had to code this consistency in C, using Gnu-Prolog internal functions at the level of the implementation of indexicals. However, the ease of integration of the user-defined indexicals in Gnu-Prolog is preserved since a separate file containing the code is compiled and linked with the Prolog program which uses them (see [13]).

5 Examples

The main purpose of this section is to present some experiments in which different constraints have different quality behaviors and an example of intermediate consistency. For all examples, we have compared multibound-consistency as computed by our clustering algorithm to Gnu-Prolog table constraint `fd_relation` which computes arc-consistency.

Example 18 (Robot vision). In this example, a 100×100 map is filled with 1945 points representing obstacles. This map constraint is intersected randomly 5000 times with a constraint representing the vision angle of a robot (90^o angle oriented from right to left). The solutions are the obstacles visible by the robot. The multibound-consistency is learned for this constraint and the maximal quality is obtained for around 230 clusters, far less than the 620 clusters necessary to get arc-consistency. The maximum speed-up with respect to `fd_relation` is 6.84 for this map (see figure 5). The map, its clustering and an example of intersection with the robot's vision angle are given in figure 6 and 7.

Example 19 (Square map). On this example, the map is a simple square of size 10×10 with 10 holes (the map is given in figure 8). On this constraint, one cluster is enough and bound-consistency is optimal. Adding more clusters only increases the computational cost and thus decreases the quality (see figure 9).

Example 20 (Crosswords). Crossword compilation can be done by considering that "being a 3-letter word" is a constraint described by the allowed 3-letter words taken in a dictionary (here we use 48 random words). The test consists in building a 3×3 grid with a central hole. In this constraint, tuples are located sparsely at random places, making the clustering ineffective. The maximal quality is thus obtained around arc-consistency (see figure 10).

On all the examples we tested, the conjecture has been verified and the best performance is obtained for the best consistency quality.

A last remark concerns the universality of this approach. We strongly believe that no single method can account for every constraint. Features like density,

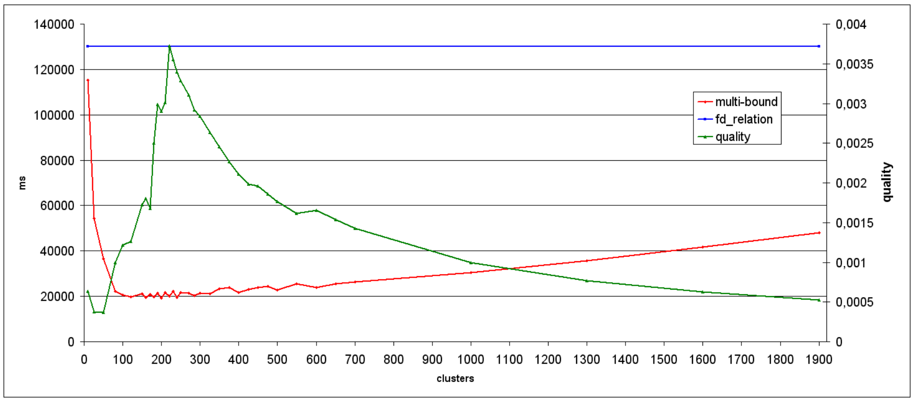

Fig. 5. Quality=f(number of clusters) and running time for the robot vision constraint.

proximity or shape can affect very seriously the suitability of a particular representation. For example, the fact that computing arc-consistency with clusters is faster than with Gnu-Prolog built-in `fd_relation` on example 18 shows that the former representation is better suited for this constraint, independently of quality optimization. In a solver design environment, it could be useful to propose several methods to build intermediate consistencies. The environment could allow to partially automatize the test of many consistencies in order to choose the best one.

6 Conclusion

Related work. Solver learning has been first introduced by [4] in which they automatically generate a set of rules from the tuples defining a constraint. These rules, called equality and membership rules, have a fixed form. The complexity of their generation limits them to small finite domains such as booleans.

The system PropMiner [1, 2] is devoted to the learning of Constraint Handling Rules [14]. All possible expressions in a given language are tested for the left and right hand-side of a rule and only minimal valid rules are kept. The solver produced by PropMiner is often very readable, especially when a small number of rules are produced, since the implication directly express the propagation. It can also express more propagation cases than indexicals due to its symbolic nature. On the other side, with indexicals, we can proceed domain size and constraint arity far beyond PropMiner.

In contrast with exact algorithms to compute arc-consistency such as GAC4 [20], or the GAC-scheme [7], our method only computes an approximation of the desired consistency. This is still useful since the trade-off between the pruning made by the consistency and the efforts needed to enforce it are largely dependent of the nature and the size of the problem. Other formalisms compute upper bounds at run-time, such as Generalized Constraint Propagation [18], or Con-

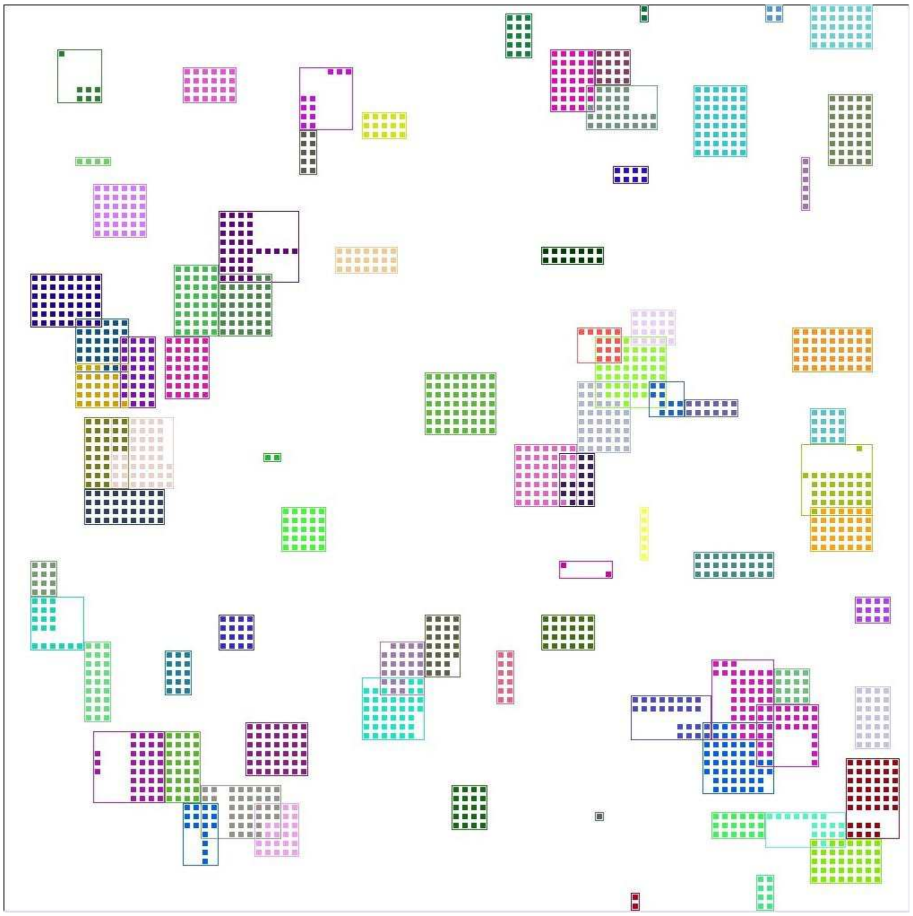

Fig. 6. 80-clustering of a 100 × 100 map with 1945 tuples.

structive Disjunction [21, 8], but are not designed to statically build a solver. Directed arc-consistency [12] can also be viewed as an intermediate consistency but DAC according two variable orderings may be incomparable and thus difficult to use within this framework.

Summary. In this paper, we propose a new consistency generation method which allows to build a full range of consistencies between bound- and arc-consistency. These consistencies can be compared both by their pruning power and by their computational complexity. This allows to optimize the ratio pruning/complexity to find the optimal consistency for a constraint. This happens to be experimentally confirmed.

Acknowledgements. We would like to thank Daniel Diaz for his kind support on the internals of Gnu-Prolog, Slim Abdennadher for fruitful discussions on automatic

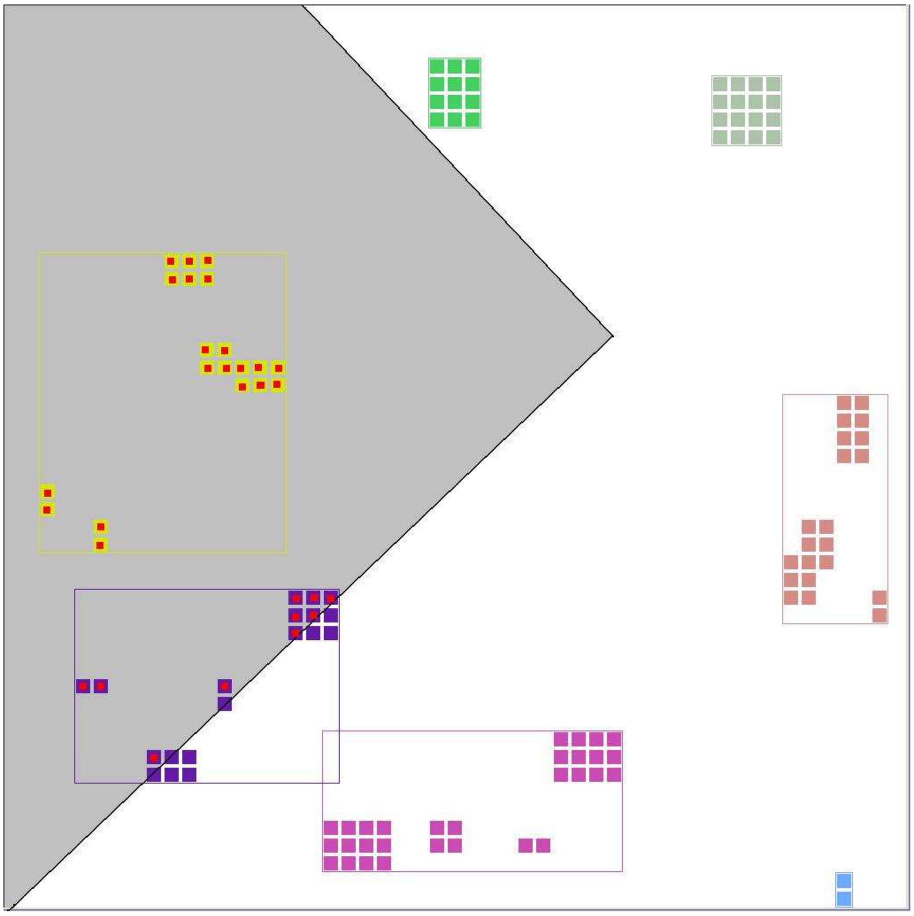

Fig. 7. Intersection of a 50 × 50 map with the robot's eye.

solver construction, Lionel Martin for his advices on clustering algorithms and the rest of the Solar Team. This project is supported by French CNRS grant 2JE095.

References

1. Slim Abdennadher and Christophe Rigotti. Automatic generation of rule-based solvers for intensionally defined constraints. *IJAIT*, 11(2):283–302, 2002.
2. Slim Abdennadher and Christophe Rigotti. Automatic generation of rule-based constraint solvers over finite domains. *ACM TOCL*, 2003. accepted for publication.
3. K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.
4. K. R. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In Joxan Jaffar, editor, *International Conference*

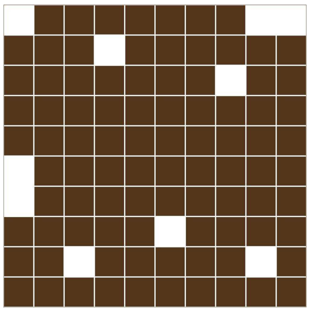

Fig. 8. 10×10 map with 10 holes.

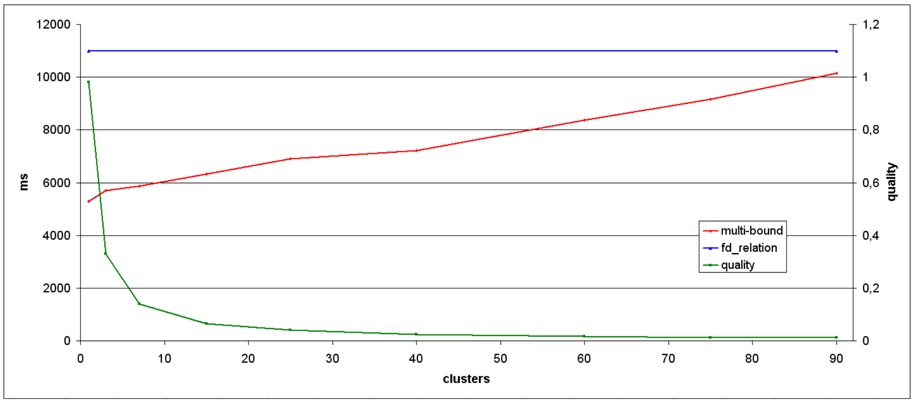

Fig. 9. Quality=f(number of clusters) and running time for the cube constraint.

on Principles and Practice of Constraint Programming, volume 1713 of *LNCS*, pages 58–72, Alexandria, Virginia, USA, 1999. Springer.

5. P. Berlandier. Improving domain filtering using restricted path consistency. In *CAIA*. IEEE, 1995.
6. Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In Eugene C. Freuder, editor, *Principles and Practice of Constraint Programming*, volume 1118 of *LNCS*, pages 61–75, Cambridge, MA, USA, 1996. Springer.
7. Christian Bessière and Jean-Charles Régin. Arc-consistency for general constraint networks: preliminary results. In *IJCAI'97*, pages 398–404, Nagoya, 1997. Morgan Kaufmann.
8. Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 27(3):185–226, 1996.
9. Thi-Bich-Hanh Dao, Arnaud Lallouet, Andrei Legtchenko, and Lionel Martin. Indexical-based solver learning. In Pascal van Hentenryck, editor, *International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *LNCS*, pages 541–555, Ithaca, NY, USA, Sept. 7 - 13 2002. Springer.
10. Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *International Joint Conference on Ar-*

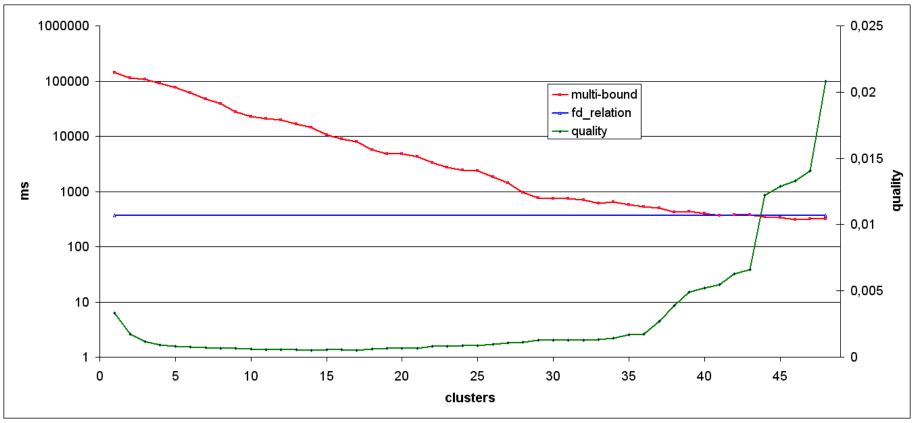

Fig. 10. Quality=f(number of clusters) and running time for the crossword constraint.

tificial Intelligence, pages 412–417, Nagoya, Japan, 1997. AAAI.

11. Romuald Debruyne and Christian Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, May 2001.
12. R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.
13. Daniel Diaz and Philippe Codognet. Design and implementation of the Gnu-Prolog system. *Journal of Functional and Logic Programming*, 2001(6), 2001.
14. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
15. A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.
16. Arnaud Lallouet, Thi-Bich-Hanh Dao, and AbdelAli Ed-Dbali. Language, definition and optimal computation of CSP approximations. In Susan Haller and Ingrid Russell, editors, *Flairs'03, International Florida Artificial Intelligence Conference*, pages 182–186, St Augustine, FL, USA, 2003. AAAI Press.
17. Arnaud Lallouet, Thi-Bich-Hanh Dao, Andreï Legtchenko, and AbdelAli Ed-Dbali. Finite domain constraint solver learning. In Georg Gottlob, editor, *International Joint Conference on Artificial Intelligence*, Acapulco, Mexico, 2003. AAAI Press. Poster.
18. Thierry Le Provost and Mark Wallace. Generalized constraint propagation over the CLP Scheme. *Journal of Logic Programming*, 16:319–359, 1993.
19. Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
20. Roger Mohr and Gérald Masini. Good old discrete relaxation. In Yves Kodratoff, editor, *ECAI'88*, pages 651–656, Munich, Germany, 1988. Pitmann Publishing.
21. P. van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(fd). draft, 1991.