# In the quest of the best form of local consistency for Weighted CSP

**Javier Larrosa**
Department of Software
Universitat Politecnica de Catalunya
Barcelona, Spain
larrosa@lsi.upc.es

**Thomas Schiex**
Dépt. de Biométrie et Intelligence Artificielle
Institut National de Recherche Agronomique
Toulouse,France
Thomas.Schiex@toulouse.inra.fr

## Abstract

The weighted CSP (WCSP) framework is a soft constraint framework with a wide range of applications. In this paper, we consider the problem of *maintaining* local consistency during search for solving WCSP. We first refine the notions of *directional arc consistency* (DAC) and *full directional arc consistency* (FDAC) introduced in [Cooper, 2003] for binary WCSP, define algorithms that enforce these properties and study their complexities. We then consider algorithms that maintain either arc consistency (AC), DAC or FDAC during search. The efficiency of these algorithms is empirically studied. It appears that despite its high theoretical cost, the strongest FDAC property is the best choice.

## 1 Introduction

It is well known that *arc consistency* (AC) plays a preeminent role in efficient constraint solving. In the last few years, the CSP framework has been augmented with so-called *soft constraints* with which it is possible to express preferences among solutions [Schiex *et al.*, 1995; Bistarelli *et al.*, 1997]. Soft constraint frameworks associate costs to tuples and the goal is to find a complete assignment with minimum combined cost. Costs from different constraints are combined with a domain dependent operator $\oplus$. Extending the notion of AC to soft constraint frameworks has been a challenge in the last few years. From previous works we can conclude that the extension is direct as long as the operator $\oplus$ is idempotent. Then, [Schiex, 2000] proposed an extension of AC which can deal with non-idempotent $\oplus$. This definition has three nice properties: $(i)$ it can be enforced in polynomial time, $(ii)$ the process of enforcing AC reveals infeasible values that can be pruned and $(iii)$ it reduces to existing definitions in the idempotent operator case. [Cooper, 2003] further introduced *directional arc consistency* (DAC) and *full directional arc consistency* for strictly monotonic $\oplus$.

*Weighted constraint satisfaction problems* (WCSP) is a well known soft-constraint framework with a non-idempotent operator $\oplus$. It provides a very general model with several applications in domains such as *resource allocation* [Cabon *et al.*, 1999], *combinatorial auctions* [Sandholm, 1999], *bioinformatics* and *probabilistic reasoning* [Pearl, 1988]. [Larrosa, 2002] introduced AC*, a refinement of the AC definition for WCSP. This definition provides a stronger yet simple and elegant property to be maintained during search.

In this paper, we take the definitions of DAC and FDAC, strengthen and extend them to binary WCSP, defining the DAC* and FDAC* properties. We then define corresponding enforcing algorithms. As in the classical CSP case, we then consider the problem of maintaining AC*, DAC* and FDAC* during search and empirically compare these algorithms. These algorithms have a wide range of applications and allow a nice integration of hard and soft constraints in a common algorithmic framework.

## 2 Preliminaries

### 2.1 CSP

A binary *constraint satisfaction problem* (CSP) is a triple $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. $\mathcal{X} = \{1, \ldots, n\}$ is a set of variables. Each variable $i \in \mathcal{X}$ has a finite domain $D_i \in \mathcal{D}$ of values that can be assigned to it. $(i, a)$ denotes the assignment of value $a \in D_i$ to variable $i$. A tuple $t$ is an assignment to a set of variables. Actually, $t$ is an ordered set of values assigned to the ordered set of variables $\mathcal{X}_t \subseteq \mathcal{X}$ (namely, the $k$-th element of $t$ is the value assigned to the $k$-th element of $\mathcal{X}_t$). For a subset $B$ of $\mathcal{X}_t$, the projection of $t$ over $B$ is noted $t \downarrow_B$. $\mathcal{C}$ is a set of unary and binary constraints. A unary constraint $C_i$ is a subset of $D_i$ containing the permitted assignments to variable $i$. A binary constraint $C_{ij}$ is a set of pairs from $D_i \times D_j$ containing the permitted simultaneous assignments to $i$ and $j$. The set of variables affected by a constraint is called its *scope*. A tuple $t$ is *consistent* if it satisfies all constraints whose scope is included in $\mathcal{X}_t$. A *solution* is a consistent complete assignment. Finding a solution in a CSP is an NP-complete problem. The task of searching for a solution can be simplified by enforcing arc consistency, which may prune values that cannot participate to a solution.

### 2.2 Weighted CSPs

*Valued* CSP (as well as *semi-ring* CSP) extend the CSP framework by associating *costs* to tuples [Schiex *et al.*, 1995; Bistarelli *et al.*, 1997]. In general, costs are specified by means of a so-called *valuation structure* defined as a triple

$S = (E, \oplus, \succeq)$, where $E$ is the set of costs totally ordered by $\succeq$. The maximum and a minimum costs are noted $\top$ and $\bot$, respectively. $\oplus$ is an operation on $E$ used to combine costs.

A valuation structure is *idempotent* if $\forall a \in E, (a \oplus a) = a$. It is *strictly monotonic* if $\forall a, b, c \in E, s.t. (a \succ c) \wedge (b \neq \top)$, we have $(a \oplus b) \succ (c \oplus b)$.

Following [Larrosa, 2002], we define Weighted CSP (WCSP) as a specific subclass of valued CSP that relies on a specific valuation structure $S(k)$.

**Definition 1** *$S(k)$ is a triple $([0, 1, \ldots, k], \oplus, \geq)$ where,*

- *$k \in [1, \ldots, \infty]$.*
- *$\oplus$ is defined as $a \oplus b = \min\{k, a + b\}$*
- *$\geq$ is the standard order among naturals.*

Observe that in $S(k)$, we have $0 = \bot$ and $k = \top$.

A binary WCSP is a tuple $P = (k, \mathcal{X}, \mathcal{D}, \mathcal{C})$. The valuation structure is $S(k)$. $\mathcal{X}$ and $\mathcal{D}$ are variables and domains, as in standard CSP. $\mathcal{C}$ is a set of cost functions. A binary constraint $C_{ij}$ assigns costs to assignments to variables $i$ and $j$ (namely, $C_{ij} : D_i \times D_j \rightarrow [0, \ldots, k]$). A unary constraint $C_i$ assigns costs to assignments to variable $i$ (namely, $C_i : D_i \rightarrow [0, \ldots, k]$). We assume the existence of a unary constraint $C_i$ for every variable, and a *zero*-arity constraint, noted $C_\varnothing$ (if no such constraint is defined, we can always define *dummy* ones $C_i(a) = \bot, \forall a \in D_i$ or $C_\varnothing = \bot$).

When a constraint $C$ assigns cost $\top$ to a tuple $t$, it means that $C$ forbids $t$, otherwise $t$ is permitted by $C$ with the corresponding cost. The *cost* of a tuple $t$, noted $\mathcal{V}(t)$, is the sum over all applicable costs,

$$\mathcal{V}(t) = \sum_{C_{ij} \in \mathcal{C},\ \{i,j\} \subseteq \mathcal{X}_t} C_{ij}(t \downarrow_{\{i,j\}}) \oplus \sum_{C_i \in \mathcal{C},\ i \in \mathcal{X}_t} C_i(t \downarrow_{\{i\}}) \oplus C_\varnothing$$

Tuple $t$ is *consistent* if $\mathcal{V}(t) < \top$. The usual task of interest is to *find a complete consistent assignment with minimum cost*, which is NP-hard. Observe that WCSP with $k = 1$ reduces to classical CSP. In addition, $S(k)$ is idempotent iff $k = 1$ and strictly monotonic iff $k = \infty$. Two WCSP defined over the same variables are said to be *equivalent* if they define the same cost distribution on complete assignments.

For simplicity in our exposition, we assume that every constraint has a different scope. For the moment, we also assume that constraints are implemented as tables and that it is possible to consult and modify entries. This is done without loss of generality (see the proof of Theorem 3).

**Example 1** *Figure 1.a shows a WCSP with valuation structure $S(4)$ (the set of costs is $[0, \ldots, 4]$, with $\bot = 0$ and $\top = 4$). It has three variables $\mathcal{X} = \{x, y, z\}$ with values $a, b$. There are 2 binary constraints $C_{xz}, C_{yz}$ and two non trivial unary constraints $C_x$ and $C_z$. Unary costs are depicted inside their domain value. Binary costs are depicted as labelled edges connecting the corresponding pair of values (default cost of 1). Zero costs are not shown. One optimal solution is eg. $x = y = z = b$, with cost 2.*

Our definition of WCSP is the same as in [Larrosa, 2002]. It differs from usual definitions [Schiex *et al.*, 1995; Bistarelli *et al.*, 1997] which restrict WCSP to the $k = \infty$ case, a

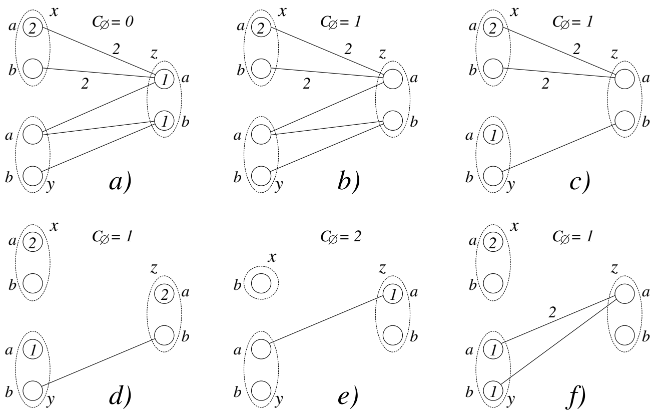


Figure 1: Six equivalent WCSPs (for $k = 4$).

strictly monotonic valuation structure where finite costs cannot lead to deletion. In practice, most *branch and bound*-based solvers maintain an upper bound $ub$, the maximum acceptable cost so-far, and a lower bound $lb$ on the optimal extension of the current assignment. Value pruning occurs as soon as $lb \geq ub$. The WCSP framework makes these two elements explicit: a solver uses the valuation structure $S(ub)$ at every subproblem and $C_\varnothing$ provides the lower bound.

## 3 Some local consistencies in WCSP

In this Section we define node, arc, directed arc and full directed arc consistencies. For node and arc consistencies, our definitions are equivalent to the NC* and AC* definitions in [Larrosa, 2002]. For DAC, and FDAC, the starred (*) definitions refine the definitions in [Cooper, 2003] to the WCSP case, using node consistency and $C_\varnothing$. In the sequel, we assume that the set of variables $\mathcal{X}$ is totally ordered by $>$.

**Definition 2** *Let $P = (k, \mathcal{X}, \mathcal{D}, \mathcal{C})$ be a binary WCSP.*

- Node consistency*: $(i, a)$ is star node consistent (NC*) if $C_\emptyset \oplus C_i(a) < \top$. Variable $i$ is NC* if: $i)$ all its values are NC* and $ii)$ there exists a value $a \in D_i$ such that $C_i(a) = \bot$. Value $a$ is a support for the variable $i$ . $P$ is NC* if every variable is NC*.*
- Arc consistency. *$(i, a)$ is arc consistent (AC) with respect to constraint $C_{ij}$ if there is a value $b \in D_j$ such that $C_{ij}(a, b) = \bot$. Value $b$ is called a support of the value $(i, a)$. Variable $i$ is AC if all its values are AC wrt. every binary constraint affecting $i$. $P$ is AC* if every variable is AC and NC*.*
- Directional arc consistency. *$(i, a)$ is directional arc consistent (DAC) wrt. constraint $C_{ij}, j > i$, if there is a value $b \in D_j$ such that $C_{ij}(a, b) \oplus C_j(b) = \bot$. Value $b$ is called a* full support *of $a$. Variable $i$ is DAC if all its values are DAC with respect to every $C_{ij}, j > i$. $P$ is DAC* if every variable is DAC and NC*.*
- Full directional arc consistency. *$P$ is fully star directional arc consistent (FDAC*) if it is DAC* and AC*.*

In the CSP case (*i.e,* $k = 1$), being a support for $(i, a)$ is obviously equivalent to being a *full support* for it and both notions reduce to the classical notion of support. Therefore, AC* and DAC* reduce to their classical definitions in CSP while FDAC* reduces to arc consistency. In WCSP, however, a support is not necessarily a full support. For this reason, DAC* (which requires a full support on one side of each constraint) and AC* (which requires a support on both sides) are incomparable. FDAC* (which requires a support on one side and a full support on the other) cannot be weaker than AC* or DAC*[1]. In WCSP, FDAC* can actually be stronger, in the sense that it may provide a better lower bound as the following example shows.

**Example 2** *The problem in Figure 1.a is not NC* since $z$ has no support. Problem 1.b is an equivalent NC* problem. It is not DAC* for order $xyz$ since $(y,a)$ has no full support on $z$. It is not AC* either since eg. $(z,a)$ and $(y,a)$ have no support on $x$ and $z$ respectively. Problem 1.c is an equivalent DAC* problem. It is not AC* since $(z,a)$ has no support on $x$. Problem 1.d is an equivalent AC* problem. It is not DAC* since $(y,b)$ has no full support on $z$. Problem 1.e is an equivalent FDAC* problem. The 2 optimal solutions are made obvious here.*

There is a strong relation between directional arc consistency and mini-buckets [Dechter, 1997]. It can easily be shown that given a WCSP defined over the valuation structure $S(\infty)$ and a variable ordering, the lower bound induced by mini-buckets involving at most 2 variables is the same as the lower bound induced by $C_\varnothing$ after the problem is made directional arc consistent. However, the mini-bucket computation provides only a lower bound while DAC enforcing provides both a lower bound and a directional arc consistent equivalent problem. All the work done to compute the lower bound is captured in this problem which offers the opportunity to perform incremental updates of the lower bound.

## 4 Enforcing Arc Consistencies

The previous node and arc consistency properties can be enforced by applying basic operations until the corresponding property is satisfied: pruning node-inconsistent values, forcing supports to variables (NC*), forcing (full) support to node-consistent values (AC). As pointed out in [Schiex, 2000; Larrosa, 2002], value (resp. variable) supports can be forced by *sending* costs from binary (resp. unary) constraints to unary constraints (resp. $C_\varnothing$). Full support can be forced by first sending costs from a unary constraint $C_j$ to $C_{ij}$ and then sending the cost from $C_{ij}$ to $C_i$ [Cooper, 2003]. Let us review these concepts before introducing basic algorithms.

Let $a, b \in [0, \ldots, k]$, be two costs such that $a \geq b$. $a \ominus b$ is the *subtraction* of $b$ from $a$, defined as,

$$a \ominus b = \begin{cases} a - b & : \quad a \neq k \\ k & : \quad a = k \end{cases}$$

The *projection* of $\alpha$ cost units from $C_{ij} \in \mathcal{C}$ over value $(i,a)$ is a flow of $\alpha$ cost units from the binary constraint to the unary cost $C_i(a)$. It is embodied in the Procedure Project$(i,a,j,\alpha)$. For the subtraction to be defined, the maximum flow that can be projected is $\min_{b \in D_j}\{C_{ij}(a,b)\}$. The converse *extension* of $\beta$ cost units from value $(i,a)$ to $C_{ij} \in \mathcal{C}$ is a reverse flow of $\beta$ cost units from $(i,a)$ to the binary constraint. It is embodied in the Procedure Extend$(i,a,j,\beta)$. The maximum flow that can be extended is $C_i(a)$. Unless stated otherwise, we always assume that maximum flows are projected or extended in the rest of the paper. Similar operations can be defined between unary constraints and $C_\varnothing$.

```
Procedure Project(i, a, j, α)
    C_i(a) := C_i(a) ⊕ α;
    foreach b ∈ D_j do C_ij(a,b) := C_ij(a,b) ⊖ α;
Procedure Extend(i, a, j, β)
    foreach b ∈ D_j do C_ij(a,b) := C_ij(a,b) ⊕ β;
    C_i(a) := C_i(a) ⊖ β;
```

**Theorem 1** *[Schiex, 2000] Let $P = (k, \mathcal{X}, \mathcal{D}, \mathcal{C})$ be a binary WCSP. Let $\alpha \leq \min_{b \in D_j}\{C_{ij}(a,b)\}$, $\beta \leq C_i(a)$. The* projection *of $\alpha$ cost unit of $C_{ij} \in \mathcal{C}$ over $(i,a)$ or the converse* extension *of $\beta$ cost unit from $(i,a)$ to $C_{ij}$ transform $P$ into an equivalent problem $P'$.*

**Example 3** *Consider the problem in Figure 1.a. To enforce NC* we must force a support for $z$ by projecting $C_z$ onto $C_\varnothing$. The resulting problem 1.b is NC* but not AC*. To enforce AC*, it suffices to force a support for $(y,a)$ and $(z,a)$: we project $C_{yz}$ over $(y,a)$ by adding $1$ to $C_y(a)$ and subtracting $1$ from $C_{yz}(a,a)$ and $C_{yz}(a,b)$ and similarly project $C_{xz}$ over $(z,a)$. We get problem 1.d which is AC* but not FDAC* using the order $xyz$ since $(y,b)$ has no full support on $z$. To force a full support for $(y,b)$, we extend $1$ cost unit from $C_z(a)$ to $C_{yz}$ by adding $1$ to $C_{yz}(a,a)$ and $C_{yz}(b,a)$ and subtracting $1$ from $C_z(a)$. We then project $C_{yz}$ to $(y,b)$ which increases $C_y(b)$ to $1$. $y$ is now node-inconsistent, we project $y$ on $C_\varnothing$ and get $C_\varnothing = 2$. $x$ is now node-inconsistent, we prune $(x,a)$ and get problem 1.e which is FDAC* according to order $xyz$.*

For simplicity, the following descriptions assume that no empty domain is produced and that the initial problem is NC*. It also assumes that the problems have no constraints of arity larger than two. [Larrosa, 2002] defined W-AC*2001, an algorithm based on AC2001 [Bessière and Régin, 2001] to enforce AC*. It is embodied in the AC*() function of Algorithm 1. It requires two data structures $S(i,a,j)$ and $S(i)$ which respectively store the current value support for $(i,a)$ with respect to constraint $C_{ij}$ and the current variable support for $i$. The algorithm uses three auxiliary functions : Function ProjectUnary$(i)$ projects $C_i$ onto $C_\varnothing$, Function PruneVar$(i)$ prunes node inconsistent values in $D_i$ and returns **true** if the domain is changed. Function FindSupportAC*$(i,j)$ forces a support on $C_{ij}$ for each value in $D_i$ by projecting $C_{ij}$ on $C_i$. The main procedure AC*() uses a queue $Q$ containing those variables whose domain has been pruned: adjacent variables may have unsupported values in their domains and new supports must be sought. $Q$ should be initialized with all variables because every variable must find an initial support on every constraint. Ignore for the moment the boolean returned by FindSupportAC* and the use of $R$. [Larrosa, 2002]

[1]The stronger local property that would require a full support on both sides suffers from the fact that most WCSP don't have an equivalent WCSP that satisfies this property.

showed that AC*() is time $O(n^2d^3)$ and space $O(ed)$ on general WCSP.

```
Procedure ProjectUnary(i)
    S(i) := argmin_{a∈D_i}{C_i(a)};
    α := C_i(S(i));
    C_∅ := C_∅ ⊕ α;
    foreach a ∈ D_i do C_i(a) := C_i(a) ⊖ α;

Function FindSupportAC*(i, j) : boolean
    flag := false;
    foreach a ∈ D_i s.t. S(i, a, j) ∉ D_j do
        S(i, a, j) := argmin_{b∈D_j}{C_ij(a, b)};
        α := C_ij(a, S(i, a, j));
        if (C_i(a) = ⊥) ∧ (α > ⊥) then flag := true;
        Project(i, a, j, α);
    ProjectUnary(i);
    return flag;

Function PruneVar(i) : boolean
    change := false;
    foreach a ∈ D_i s.t. (C_i(a) ⊕ C_∅ = ⊤) do
        D_i := D_i − {a};
        change := true;
    return change;

Procedure AC*()
    while (Q ≠ ∅) do
        j := pop(Q);
        for C_ij ∈ C do
            if FindSupportAC*(i, j) then R := R ∪ {i};
        foreach i ∈ X do
            if PruneVar(i) then Q := Q ∪ {i};
```

Algorithm 1: Enforcing AC*, initially $Q = \mathcal{X}$

[Cooper, 2003] introduced non incremental algorithms for enforcing DAC and FDAC on *strictly monotonic* valuation structures. These algorithms are inadequate for maintaining DAC* or FDAC* in a WCSP branch and bound algorithm that relies on a non strictly monotonic valuation structure $S(k), k \neq +\infty$ as soon as a feasible solution is found.

The new basic operation needed to enforce {F}DAC* consists in forcing *full supports* for the values of a variable $i$ on one side of a constraint $C_{ij}$. As shown in the example, this can be done by extending unary costs from $C_j$ to $C_{ij}$ and then projecting $C_{ij}$ onto variable $C_i$. However, extending all unary costs may destroy supports for $j$ on $C_{ij}$. Consider the AC* Problem 1.d. If we extend 2 cost units from $(z, a)$ to $C_{yz}$ instead of 1 as in the example and then project on $C_y$, we get Problem 1.f where $(z, a)$ has lost all supports on $y$. In order to smoothly integrate DAC* and AC* enforcing to obtain FDAC* enforcing, we must obtain full supports for variable $i$ on $C_{ij}$ while preserving *supports* for all values of $j$ on $C_{ij}$. This is obtained by extending the minimum cost of $C_j$ required for the subsequent projection onto $C_i$. The correctness of our algorithms is based on the following theorem,

**Theorem 2** *Let $j$, a variable whose values are AC wrt. $C_{ij}$. $\forall a \in D_i, b \in D_j$, let $P[a] = \min_{b\in D_j}\{C_{ij}(a, b) \oplus C_j(b)\}$ and $E[b] = \max_{a\in D_i}\{P[a] - C_{ij}(a, b)\}$. Extending $E[b]$ cost units from $(j, b)$ to $C_{ij}$ for all $b \in D_j$ and projecting $P[a]$ cost units from $C_{ij}$ to $(i, a)$ for all $a \in D_i$ yields an equivalent WCSP s.t. every node consistent value of $i$ is DAC wrt. $C_{ij}$ and s.t. every value $(j, b)$ is supported by $argmax_{a\in D_i}\{P[a] - C_{ij}(a, b)\}$ if it is node consistent.*

**Proof:** We denote $C^0_{ij}(a, b)$ the original value of $C_{ij}(a, b)$. We first show that $E[b]$ and $P[a]$ are possible flows. We first prove that $0 \leq E[b] \leq C_j(b)$: we have $P[a] - C_{ij}(a, b) = \min_{b'\in D_j}(C_{ij}(a, b') \oplus C_j(b')) - C_{ij}(a, b) \leq (C_{ij}(a, b) \oplus C_j(b)) - C_{ij}(a, b) \leq (C_{ij}(a, b) + C_j(b)) - C_{ij}(a, b) = C_j(b)$ and therefore $E[b] \leq C_j(b)$. Since $j$ is AC, it has a support $(i, a)$ s.t. $C^0_{ij}(a, b) = \perp \leq P[a]$. Therefore $P[a] - C_{ij}(a, b) \geq 0$ and $E[b] \geq \perp$.

After the extension of $E[b]$ cost units, $C_{ij}(a, b)$ will be equal to $C^0_{ij}(a, b) \oplus \max_{a'\in D_i}\{P[a'] - C^0_{ij}(a', b)\} \geq C^0_{ij}(a, b) \oplus (P[a] - C^0_{ij}(a, b))$. Either this is equal to $\top$ and obviously $C_{ij}(a, b) \geq P[a]$ or else $C^0_{ij}(a, b) \oplus (P[a] - C^0_{ij}(a, b)) = P[a]$ and again $C_{ij}(a, b) \geq P[a]$.

Since $P[a] = \min_{b\in D_j}\{C^0_{ij}(a, b) \oplus C_j(b)\}$ the value $b$ for which this minimum is reached will either be a full support for $(i, a)$ if $P[a] \neq \top$ or $(i, a)$ will be deleted.

On the other side, consider value $(j, b)$ and $a = argmax(P[a] - C^0_{ij}(a, b))$. After extension and projection, either $C_{ij}(a, b) = C^0_{ij}(a, b) \oplus (P[a] - C^0_{ij}(a, b)) \ominus P[a] \leq P[a] \ominus P[a]$ and either $P[a] < \top$ and $a$ is a support of $(j, b)$ or $P[a] = \top$ and $a$ is node inconsistent. □

Based on this theorem, Function FindFullSupportAC*$(i, j)$ forces full supports for all the values of $i$ on $C_{ij}$ while taking care of supports for values in $D_j$. It returns **true** whenever the cost of a value $(i, a)$ has been increased from $\perp$. The Procedure DAC*() has been designed to be used alone to enforce DAC* or in conjunction with AC*() to enforce FDAC*. Therefore, whenever a value is pruned, DAC*() inserts its variable in $Q$ to inform AC*() of the deletion. DAC* further uses a *priority queue* $R$ that contains those variables such that a unary cost has been increased from $\perp$: in this case, some values in lower variables may have lost full support and new supports need to be found. The main loop iterates while $R$ is not empty. At each iteration, the highest variable $j$ is fetched from $R$. Node inconsistent values (due to unary cost and lower bound increments) are removed using PruneVar() and pruned variables are inserted in $Q$. Then new full supports are sought for every lower variable connected to $j$. Finally, all variables are processed to enforce NC* which can be lost during the process, due to lower bound increments. Pruned variables are inserted in $Q$. FDAC*() simply enforces AC* and DAC* simultaneously: the enforcement of AC* empties $Q$ but may add variables to $R$, and the enforcement of DAC* empties $R$ but may add variables to $Q$. FDAC* is achieved when both $R$ and $Q$ are simultaneously empty. Correction of both algorithms follows from theorem 2.

**Theorem 3** *The complexity of* DAC*() *is time $O(ed^2)$ and space $O(ed)$. $n$, $e$ and $d$ are the number of variables, constraints and largest domain size respectively.*

**Proof:** FindFullSupportAC*$(i, j)$ and PruneVar$(i)$ have complexities $O(d^2)$ and $O(d)$ respectively. The only way a variable $j$ may enter the queue $R$ is because some null unary cost $C_j(b)$ has been increased in FindFullSupportAC*. $R$ being a

**Function** FindFullSupportAC*$(i, j)$ : boolean

*flag* := **false**;
**foreach** $a \in D_i$ s.t $C_{ij}(a, S(i,a,j)) \oplus C_j(S(i,a,j)) > \bot$ **do**
  $S(i,a,j) := argmin_{b \in D_j}\{C_{ij}(a,b) \oplus C_j(b)\}$;
  $P[a] := C_{ij}(a, S(i,a,j)) \oplus C_j(S(i,a,j))$;
  **if** $(P[a] > \bot) \wedge (C_i(a) = \bot)$ **then** *flag* := **true**;
**foreach** $b \in D_j$ **do**
  $S(j,b,i) := argmax_{a \in D_i}\{P[a] - C_{ij}(a,b)\}$;
  $E[b] := P[S(j,b,i)] - C_{ij}(a,b)$;
**foreach** $b \in D_j$ **do** Extend$(j, b, i, E[b])$;
**foreach** $a \in D_i$ **do** Project$(i, a, j, P[a])$;
ProjectUnary$(i)$;
**return** *flag*;

**Procedure** DAC*()

**while** $(R \neq \varnothing)$ **do**
  $j := \mathrm{pop}(R)$;
5 **if** PruneVar$(j)$ **then** $Q := Q \cup \{j\}$;
  **foreach** $C_{ij} \in \mathcal{C}$ s.t. $i < j$ **do**
    **if** FindFullSupportAC*$(i, j)$ **then** $R := R \cup \{i\}$;
**foreach** $i \in \mathcal{X}$ **do**
6 **if** PruneVar$(i)$ **then** $Q := Q \cup \{i\}$;

**Procedure** FDAC*()

**while** $(Q \neq \varnothing) \vee (R \neq \varnothing)$ **do**
7 AC*();
8 DAC*();

Algorithm 2: DAC* and FDAC*. Initially, $Q = R = \mathcal{X}$.

priority queue, when a variable $j$ is extracted from $R$, all the variables before $j$ in $R$ have already been processed. Since FindFullSupportAC*() can only increase non zero unary costs of variables strictly lower than $j$, $j$ will never be reintroduced in $R$ and therefore each variable $j$ is added to the queue $R$ at most once. The queue is implemented as an array of booleans and a pointer to the highest *true* element. Adding new elements to $R$ means updating the pointer, the *pop* operation consists on returning the value of the pointer and searching for the new highest *true* element. Clearly, DAC*() only traverses the array once. Thus, the complexity of the *while* loop is in $O(ed^2)$. Since the complexity of the second *foreach* loop is in $O(nd)$, the global complexity is in $O(ed^2)$.

As it is, DAC*() is space $O(ed^2)$ since it modifies binary constraints. As [Larrosa, 2002], we note that when a binary constraint $C_{ij}$ is modified (line 1 and 2 in Project and Extend), it is by addition or subtraction of costs that depend either on $i$ or $j$. It is therefore possible to record only row and column changes, the current value of $C_{ij}(a,b)$ being obtained as $C^0_{ij}(a,b) \ominus F(i,j,a) \ominus F(j,i,b)$ where $C^0_{ij}(a,b)$ denotes the original constraint. There is one $F(i,j,a)$ entry per constraint-value pair which is space $O(ed)$. □

**Theorem 4** *The complexity of* FDAC*() *is time* $O(end^3)$ *and space* $O(ed)$.

**Proof:** Regarding space, there is no difference with DAC*() and the same proof applies. Regarding time, a variable $j$ enters $Q$ only if a value has been deleted. Therefore, each variable $j$ is added to $Q$ at most $d+1$ times (once at initialization and then upon value deletion at lines 5, 6 or 4). Therefore, line 3 of Procedure AC*() is executed at most $2e(d+1)$ times and line 4 at most $nd$ times. Globally, line 7 of Algorithm 2 will therefore use $O(n^2d^2 + ed^3)$ elementary operations. For the same reason, line 8 is executed at most $O(nd)$ times. Since DAC* is in $O(ed^2)$, this can generate $O(end^3)$ elementary operations. Globally, the algorithm is time $O(n^2d^2 + ed^3 + end^3) = O(end^3)$. □

A consequence of these complexity results is that all algorithms terminate (even in the $S(\infty)$ structure).

## 5 Experimental results

In this Section we perform an empirical evaluation of the effect of maintaining various forms of arc consistency during search. We consider a depth-first search maintaining either NC*, AC*, DAC* or FDAC* which yields the algorithms MNC*, MAC*, MDAC* and MFDAC*. For comparison, we include results obtained with PFC-RDAC [Larrosa *et al.*, 1999], which is normally considered as a reference algorithm.

For variable selection we use the $dom/deg$ heuristic which for each variable computes the ratio of the domain-size divided by the future degree (*i.e.*, degree considering future variables only) and selects the variable with the smallest value. For value selection we consider values in increasing order of unary cost $C_i$. The variable ordering used for directional arc consistencies is lexicographic.

We consider the Max-CSP problem, where the goal is to find a complete assignment with a maximum number of satisfied constraints in an overconstrained CSP. It can easily be formulated as a WCSP. We experiment with binary random problems using the well-known four-parameters model [Smith, 1994]. A random CSP class is defined by $\langle n, d, e, t \rangle$ where $n$ is the number of variables, $d$ is the domain size, $e$ is the number of binary constraints (i.e, graph *connectivity*), and $t$ the number of forbidden tuples in each constraint (i.e, *tightness*). Pairs of constrained variables and their forbidden tuples are randomly selected using a uniform distribution. Samples have 50 instances and we report average values. The experiments were performed on a 800 MHz Pentium III computer.

For fixed values of $n$, $d$ and $e$ and increasing tightness $t$, most problems are solved almost instantly until the cross-over point is reached. Then, problems become overconstrained and much harder to solve. We denote $t^o$ the lowest tightness where every instance in our sample is overconstrained. Based on this, we define different categories of problems:

- For graph density, we define two problem types: *sparse* (S) with $e = 2.5n$ and , *dense* (D) with $e = \frac{n(n-1)}{8}$.
- For tightness, we define two problem types: *loose* (L) with $t = t^o$, and *tight* (T) with $t = d^2 - 0.25t^o$.

Combining the different types, we obtain 4 different classes, each being denoted by a pair of characters (SL,ST,DL and DT). In each class, the domain size is set to 10 and the number of variables $n$ is used as a varying parameter. Figure 2 shows the average cpu time used with SL, ST, DL and DT from left to right. In each plot, the five algorithms are listed in increasing order of efficiency, from top to bottom. In all cases, the search effort seems to grow exponentially with $n$.

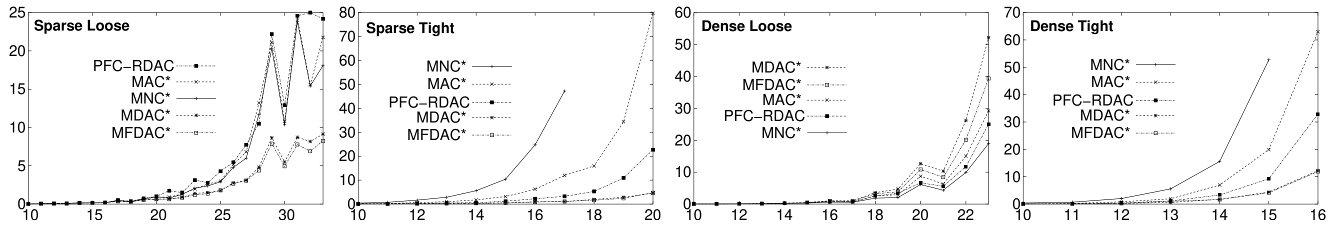


Figure 2: Cpu-time in seconds for an increasing number of variable on our 4 classes of problems. In each case, the 4 algorithms are listed in increasing order of efficiency from top to bottom.

For all classes except the DL class, MFDAC* is the most efficient algorithm, with only minor differences with MDAC* (sometimes they are so closed that the two lines can hardly be distinguished). The best performance of MFDAC* is obtained in the ST problems, where it is up to 5 times faster than PFC-RDAC, 20 times faster than MAC* and 50 times faster than MNC*. For the DL class, however, MNC* is the most efficient algorithm, followed by PFC-RDAC, MAC*, MFDAC* and MDAC*. The differences between the algorithms are however more limited than in previous classes (MNC* is twice faster than MFDAC*).

The ability of directional arc consistency to collect costs along the constraints in order to bring them together in the same variable allows to build stronger lower bounds. This is confirmed by the analysis of the number of nodes expanded by each algorithm (not reported here for lack of space) where MDAC* and MFDAC* always expand less nodes that PFC-RDAC, MNC* or MAC*, with a ratio that can reach 300 between the extreme algorithms on eg. ST problems. On the DL problems however, this ratio is much more limited, typically bounded by 4. With loose constraints, the upper bound reaches low values early in the search which allows pruning at high levels of the search tree and makes sophisticated lower bounds less significant.

It is worth to mention at this point that PFC-RDAC heuristically assigns a direction to every constraint in each subproblem and this has a strong influence on the efficiency on random Max-CSP. Similarly, the behavior of AC, DAC and FDAC based algorithms depends on the order in which variables are fetched from $Q$ and $R$ (i.e., on the variable ordering used to define DAC) and on the order in which values are considered for projection. In our current implementation, $Q$ is implemented as a stack, values are considered in lexicographic ordering and the DAC variable ordering is lexicographic. This leaves room for further improvement.

## 6 Conclusion and Future Work

In this paper we have refined two local consistency properties and adapted them to WCSP. We have developed enforcing algorithms and have studied their complexity.

As in classical CSP, we observe that the choice of the right level of local consistency to maintain during search is important. Despite its theoretical cost, the strongest local consistency we considered (FDAC*) appears to be the best level for solving WCSP. In the future, we want to extend these algorithms to non binary constraints, apply them to other problems and take into account heuristics for the variable and value ordering used in AC, DAC and FDAC enforcing.

## References

[Bessière and Régin, 2001] C. Bessière and J-C. Régin. Refining the basic constraint propagation algorithm. In *Proc. of the 14$^{th}$ IJCAI*, pages 309–315, 2001.

[Bistarelli *et al.*, 1997] S. Bistarelli, U. Montanari, and F. Rossi. Semiring based constraint solving and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[Cabon *et al.*, 1999] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints Journal*, 4:79–89, 1999.

[Cooper, 2003] Martin C. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3), 2003.

[Dechter, 1997] R. Dechter. Mini-buckets: A general scheme for generating approximations in automated reasoning. In *Proc. of IJCAI'97*, Nagoya, Japan, 1997.

[Larrosa *et al.*, 1999] J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1):149–163, 1999.

[Larrosa, 2002] Javier Larrosa. On arc and node consistency in weighted CSP. In *Proc. AAAI'02*, 2002.

[Pearl, 1988] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems, Networks of Plausible Inference*. Morgan Kaufmann, Palo Alto, 1988.

[Sandholm, 1999] T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *Proc. of IJCAI'99*, pages 542–547, 1999.

[Schiex *et al.*, 1995] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: hard and easy problems. In *Proc. of IJCAI'95*, pages 631–637, 1995.

[Schiex, 2000] T. Schiex. Arc consistency for soft constraints. In *CP'2000*, volume 1894 of *LNCS*, pages 411–424, 2000.

[Smith, 1994] B. Smith. Phase transition and the mushy region in constraint satisfaction. In *Proc. of the 11$^{st}$ ECAI*, pages 100–104, 1994.