Existential arc consistency: Getting closer to full arc consistency in weighted CSPs*

Simon de Givry
degivry@toulouse.inra.fr
Matthias Zytnicki
zytnicki@toulouse.inra.fr
INRA
Toulouse, France

Federico Heras
fheras@lsi.upc.edu
Javier Larrosa
larrosa@lsi.upc.edu
LSI, UPC
Barcelona, Spain

Abstract

The weighted CSP framework is a soft constraint framework with a wide range of applications. Most current state-of-the-art complete solvers can be described as a basic depth-first branch and bound search that maintain some form of arc consistency during the search. In this paper we introduce a new stronger form of arc consistency, that we call *existential directional arc consistency* and we provide an algorithm to enforce it. The efficiency of the algorithm is empirically demonstrated in a variety of domains.

1 Introduction

Weighted constraint satisfaction problems (WCSP) is a well-known optimization version of the CSP framework with many practical applications. Recently, the celebrated arc consistency property has been generalized from the classical CSP framework to WCSP [Cooper and Schiex, 2004; Larrosa and Schiex, 2004]. There are three known generalizations: AC*, FDAC* and FAC*, all of them collapsing to classical arc consistency in the CSP case. In [Larrosa and Schiex, 2003] it was shown that maintaining FDAC* during search was usually the best option.

In this paper we introduce a new form of local consistency called *existential directional arc consistency* (EDAC*). We show that EDAC* is stronger than FDAC* and we introduce an algorithm to enforce EDAC* that runs in time $O(ed^2 \times \max\{nd, \top\})$, where e, n and d are the number of constraints, variables and domain values, respectively; $\top$ is the upper bound of the WCSP instance (to be defined later). The main use of EDAC* and its associated filtering algorithm is to embed it into a complete depth-first branch and bound solver that *maintains* EDAC* at every visited node. We have experimentally evaluated this idea on random Max-SAT, Max-CSP, and *uncapacitated warehouse location problems*. We observe that maintaining EDAC* is never worse than maintaining FDAC* and, in many instances, it is orders of magnitude better.

*This research is partially supported by the French-Spanish collaboration PICASSO 05158SM - Integrated Action HF02-69 and the REPLI project TIC-2002-04470-C03.

2 Preliminaries

Valuation structures are algebraic objects to specify costs in valued constraint satisfaction problems [Schiex *et al.*, 1995]. They are defined by 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.

Following [Larrosa and Schiex, 2004], the valuation structure of Weighted CSP (WCSP) is, $S(k) = ([0..k], \oplus, \geq)$ where $k > 0$ is a natural number; $\oplus$ 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 $k = \top$. It is useful to define the *subtraction* $\ominus$ of costs. Let $a, b \in [0..k]$ be two costs such that $a \geq b$,

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

A binary *weighted constraint satisfaction problem* (WCSP) is a tuple $P = (S(k), \mathcal{X}, \mathcal{D}, \mathcal{C})$. $S(k)$ is the valuation structure. $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of variables that we will often call by just their index. Each variable $x_i \in \mathcal{X}$ 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 x_i. $\mathcal{C}$ is a set of unary and binary weighted constraints (namely, cost functions) over the valuation structure $S(k)$. A unary weighted constraint C_i is a cost function $C_i(x_i) \rightarrow [0..k]$. A binary constraint C_{ij} is a cost function $C_{ij}(x_i, x_j) \rightarrow [0..k]$ We assume the existence of a unary constraint C_i for every variable, and a *zero*-arity constraint (i.e. a constant), noted $C_\varnothing$ (if no such constraint is defined, we can always define *dummy* ones: $C_i(x_i) = \bot$, $C_\varnothing = \bot$).

When a constraint C assigns cost $\top$, it means that C forbids the corresponding assignment, otherwise it is permitted by C with the corresponding cost. The *cost* of an assignment $X = (x_1, \ldots, x_n)$, noted $\mathcal{V}(X)$, is the sum over all the problem cost functions,

$$\mathcal{V}(X) = \sum_{C_{ij} \in \mathcal{C}} C_{ij}(x_i, x_j) \oplus \sum_{C_i \in \mathcal{C}} C_i(x_i) \oplus C_\varnothing$$

An assignment X is *consistent* if $\mathcal{V}(X) < \top$. The usual task of interest is to *find a consistent assignment with minimum cost*, which is NP-hard. Observe that WCSP with $k = 1$ reduces to classical CSP.

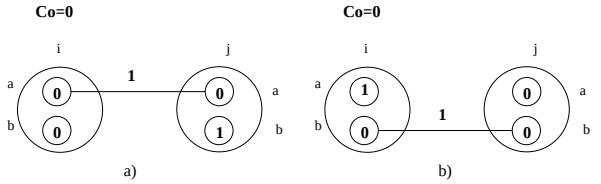

Figure 1: Two equivalent WCSP instances ($\top = 2$).

Algorithm 1: Algorithms to propagate costs.

Function PruneVar(i) : boolean
  $flag := \textbf{false}$;
  foreach $a \in D_i$ **do**
    if $(C_\varnothing \oplus C_i(a) \geq \top)$ **then**
      $D_i := D_i - \{a\}$;
      $flag := \textbf{true}$;
  return $flag$;

Procedure ProjectUnary(i)
  $\alpha := \min_{a \in D_i}\{C_i(a)\}$;
  $C_\varnothing := C_\varnothing \oplus \alpha$;
  foreach $a \in D_i$ **do** $C_i(a) := C_i(a) \ominus \alpha$;

Procedure Project(i, a, j, α)
  $C_i(a) := C_i(a) \oplus \alpha$;
  foreach $b \in D_j$ **do** $C_{ij}(a,b) := C_{ij}(a,b) \ominus \alpha$;

Procedure Extend(i, a, j, α)
  foreach $b \in D_j$ **do** $C_{ij}(a,b) := C_{ij}(a,b) \oplus \alpha$;
  $C_i(a) := C_i(a) \ominus \alpha$;

Example 1 *Consider the problem depicted in Figure1.a. It has two variables i, j with two values (a,b) in their domains. Unary costs are depicted within small circles. Binary costs are represented by edges connecting the corresponding values. The label of each edge is the corresponding cost. If two values are not connected, the binary cost between them is* 0*. In this problem the optimal cost is* 0 *and it is attained with the assignment* (b,a).

3 Some local consistencies in WCSP

Two WCSPs defined over the same variables are said to be *equivalent* if they define the same cost distribution on complete assignments. Local consistency properties are widely used to transform problems into equivalent simpler ones. In general, constraints can be given explicitly as tables of costs, or implicitly as mathematical expressions or algorithmic procedures. For simplicity in our exposition, we will assume the explicit form. This is done without loss of generality as discussed in [Cooper and Schiex, 2004]. The simplest form of local consistency is *Node consistency* (NC*).

Definition 1 *Variable x_i is node consistent if: for all values $a \in D_i$, $C_\varnothing \oplus C_i(a) < \top$, and there exists a value $a \in D_i$ such that $C_i(a) = \bot$. A WCSP is node consistent (NC*) if every variable is node consistent.*

Any WCSP can be easily transformed into an equivalent NC* instance by projecting every unary constraints towards $C_\varnothing$ and subsequently pruning every unfeasible value. Functions ProjectUnary and PruneVar (Algorithm1) perform the projection of a unary constraint towards $C_\varnothing$ and prune unfeasible values, respectively.

Arc consistency properties are based on the notion of *simple* and *full support*. Given a binary constraint C_{ij}, we say that $b \in D_j$ is a *simple support* for $a \in D_i$ if $C_{ij}(a,b) = \bot$. Similarly, we say that $b \in D_j$ is a *full support* for $a \in D_i$ if $C_{ij}(a,b) \oplus C_j(b) = \bot$.

Definition 2 *Variable x_i is arc consistent if every value $a \in D_i$ has a simple support in every constraint C_{ij}. A WCSP is arc consistent (AC*) if every variable is node and arc consistent.*

Definition 3 *Variable x_i is full arc consistent if every value $a \in D_i$ has a full support in every constraint C_{ij}. A WCSP is full arc consistent (FAC*) if every variable is node and full arc consistent.*

Simple supports for $a \in D_i$ in D_j can be enforced by projecting binary costs $C_{ij}(a,\cdot)$ towards $C_i(a)$, as performed by procedure Project (Algorithm 1). Procedure FindSupports (Algorithm 2) enforces simple supports in D_j for every value in D_i.

Example 2 *The WCSP depicted in Figure2.a is not AC* because value $a \in D_i$ does not have a simple support in variable j. AC* is enforced by executing FindSupports(i, j) (producing the problem in 2.b) and then ProjectUnary(i) (producing the problem in 2.c).*

To enforce full supports for D_i values in D_j (Procedure FindFullSupports in Algorithm 2) we need to extend unary costs from $C_j(\cdot)$ towards $C_{ij}(\cdot,\cdot)$ (procedure Extend in Algorithm 1). Then, binary costs are projected towards $C_i(a)$.

In the CSP case (*i.e,* $k = 1$), being a simple support for (i,a) is equivalent to being a full support for it and both notions reduce to the classical notion of support. Therefore, AC* and FAC* reduce to classical arc consistency. In the WCSP case, however, being a full support is stronger than being a simple support. In [Larrosa and Schiex, 2004] it was shown that every WCSP can be transformed into an equivalent arc consistent one in time $O(ed^3)$. Unfortunately, not every WCSP can be transformed into an equivalent *full* arc consistent one.

Example 3 *The problem in Figure1.a is AC*, but it is not FAC* because value $a \in D_i$ is not fully supported. We can enforce the full support by calling FindFullSupports(i, j). The resulting problem, depicted in Figure1.b is not FAC* because value $b \in D_j$ is not fully supported. If we enforce the full support by calling FindFullSupports(j,i), we return to the problem in Figure1.a. It can be easily proved that this problem does not have an equivalent FAC* one.*

Therefore, FAC* is not a practical property. To circumvent this problem, a weaker property has been proposed. In the sequel, we assume that the set of variables $\mathcal{X}$ is totally ordered by $>$.

Definition 4 *Variable x_i is directional arc consistent if every value $a \in D_i$ has a full support in every constraint C_{ij} such*

Algorithm 2: Algorithms to enforce supports.

Function FindSupports(i, j) : boolean
  $flag :=$ **false** ;
  foreach $a \in D_i$ **do**
    $\alpha := \min_{b \in D_j}\{C_{ij}(a,b)\}$;
    if $(\alpha > \bot \wedge C_i(a) = \bot)$ **then** $flag :=$ **true** ;
    Project(i, a, j, α) ;
  ProjectUnary(i) ;
  return $flag$;

Function FindFullSupports(i, j) : boolean
  $flag :=$ **false** ;
  foreach $a \in D_i$ **do**
    $P[a] := \min_{b \in D_j}\{C_{ij}(a,b) \oplus C_j(b)\}$;
    if $(P[a] > \bot \wedge C_i(a) = \bot)$ **then** $flag :=$ **true** ;
  foreach $b \in D_j$ **do**
    $E[b] := \max_{a \in D_i}\{P[a] \ominus C_{ij}(a,b)\}$;
  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 $flag$;

Function FindExistentialSupport(i) : boolean
  $flag :=$ **false** ;
1 $\alpha := \min_{a \in D_i}\{C_i(a) \bigoplus_{C_{ij} \in C \text{ s.t. } j<i} \min_{b \in D_j}\{C_{ij}(a,b) \oplus C_j(b)\}\}$;
  if $(\alpha > \bot)$ **then**
2   **foreach** $C_{ij} \in C$ s.t. $j < i$ **do**
      $flag := flag \vee$ FindFullSupports(i, j) ;
  return $flag$;

that $j > i$. It is full directional arc consistent (FDAC) if, in addition, every value $a \in D_i$ has a simple support in every constraint C_{ij} such that $j < i$. A WCSP is full directional arc consistent (FDAC) if every variable is node and full directional arc consistent.*

Example 4 *The problem in 2.c is AC* but not FDAC*, because value $a \in i$ does not have a full support in variable j. FDAC* is enforced by executing FindFullSupports(i, j) (producing the problem in Figure 2.d).*

Every WCSP can be transformed into an equivalent FDAC* WCSP with time complexity $O(end^3)$ [Larrosa and Schiex, 2003]. In the CSP case, FDAC* also reduces to classical arc-consistency. In the general case FAC* implies FDAC* and FDAC* implies AC* (namely, the satisfaction of one property implies the satisfaction of the other). An important question is whether it is possible to strengthen the FDAC* property to make it closer to FAC*, but still having the guarantee of existence for an arbitrary WCSP instance. In the next Section we address this issue.

4 Existential arc consistency: a stronger new property

Consider the FDAC* problem in Figure2.*d*. Observe that value $a \in D_k$ has a simple support in x_j (required by the FDAC* property) but does not have any full support. We have the same situation with value $b \in D_k$ and variable x_i.

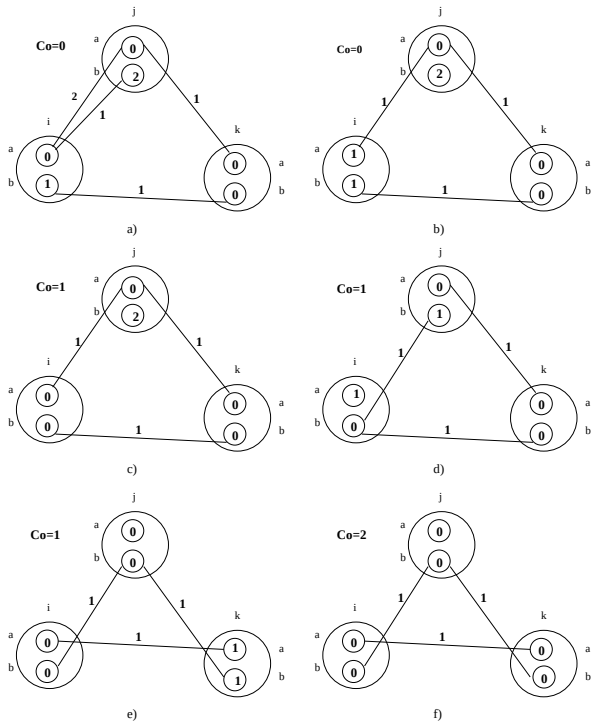

Figure 2: Six equivalent WCSP problems ($\top = 4$): (a) original problem (c) made AC*, (d) made FDAC* with $i < j < k$ and (f) made EDAC*.

As a consequence, the unary cost of both values in D_k can be increased by enforcing full supports (Figure 2.*e*). Since all unary costs of D_i values are larger than zero, node consistency is lost and the lower bound can be increased (Figure 2.f). In general, full supports can be safely enforced in both directions if their enforcement produces an increment in the lower bound. Next, we show that there is a natural local consistency property behind this observation, that we call existential arc consistency (EAC*).

Definition 5 *Variable x_i is existential arc consistent if there is at least one value $a \in D_i$ such that $C_i(a) = \bot$ and it has a full support in every constraint C_{ij}. A WCSP is existential arc consistent (EAC*) if every variable is node and existential arc consistent.*

All the previous forms of arc consistency required the same condition for every domain value. However, EAC* requires the *existence* for every variable of a special value. The important fact to note is: if a variable x_i is not EAC*, then for all domain value a with $C_i(a) = \bot$ there is a variable x_j such that $\forall b \in D_j, C_{ij}(a,b) \oplus C_j(b) > \bot$. Thus enforcing full supports in x_i will break the NC* property and it will be possible to increase the lower bound. It is possible to integrate EAC* with FDAC* in order to exploit the benefits of each. It is achieved by the existential directional arc consistency (EDAC*) property.

Definition 6 *A WCSP is EDAC* if it is FDAC* and EAC*.*

In words, EDAC* requires that every value is fully supported in one direction and simply supported in the other direction (to satisfy FDAC*). Additionally, at least one value per variable must be fully supported in both directions (to satisfy EAC*). The special value is called the *fully supported* value. Observe that in the CSP case EDAC* instantiates to classical arc consistency. Besides, EDAC* is weaker than FAC*, but stronger than FDAC*.

5 Algorithm

We present an algorithm for the enforcement of EDAC* in the case of binary WCSPs. EDAC* (Algorithm 3) transforms an arbitrary problem into an *equivalent* one verifying the EDAC* local property. It uses three propagation queues, Q, R and P, that are implemented as priority queues so that the lowest or the highest variable in the queue can be popped in constant time. If $j \in Q$, it means that some value in D_j has been pruned (neighbors of j higher than j may have lost their simple support and must be revised). If $j \in R$, it means that some value in D_j has increased its unary cost from $\bot$ (neighbors of j lower than j may have lost their full support and must be revised). If $i \in P$, it means that some value in D_j ($j < i$ neighbor of i) has increased its unary cost from $\bot$ (i may have lost the full support of its fully supported value and must be revised). Besides, there is an auxiliary queue S that is used to efficiently build P.

The algorithm is formed by a main loop, with four inner loops. The **while** loops at lines 5, 8 and 10 respectively enforce EAC*, DAC* and AC*; the line 11 enforces NC*. Each time some costs are projected by the enforcement of a local property, another property may be broken. The variables for which the local property may be broken are stored in a queue for revision. The enforcement of NC*, AC* and DAC* is achieved as proposed in [Larrosa and Schiex, 2003]. Thus, we focus on the enforcement of EAC*.

EAC* enforcement is mainly encoded by Function FindExistentialSupport(i), which enforces the existential support in x_i by finding full supports wrt. every lower neighbor x_j (it does not have to worry for higher neighbors because DAC* enforcement takes care of them). While enforcing the existential support in x_i, the cost function C_j decreases or remains the same. As a consequence, we do not need to revise the existential consistency of x_j nor the directional arc consistencies of the lowest neighbors of x_j. Furthermore, as the existential consistency is stronger than the arc one, enforcing AC* between x_i and x_j is useless. However, the directional arc consistency of variable x_i should be checked, because some unary cost of x_i could have been increased (line 6). Similarly, the existential consistency of higher neighbors of x_i should also be verified (line 7).

PruneVar() enforces node consistency after unary cost and lower bound increments (line 11). This function can be executed more often in order to avoid unnecessary value examinations during the **while** loops. It does not change the worst-case time complexity. For simplicity reasons, the case of inconsistent problems where $C_\varnothing$ reaches $\top$ is not described.

Theorem 1 *The complexity of EDAC* is time $O(ed^2 \max\{nd, \top\})$ and space $O(ed)$.*

Algorithm 3: Enforcing EDAC*, initially, $Q = R = S = \mathcal{X}$.

```
Procedure EDAC*
3   while (Q ≠ ∅ ∨ R ≠ ∅ ∨ S ≠ ∅) do
4     P := {l | i ∈ S, l > i, C_il ∈ C} ∪ S ;
      S := ∅ ;
5     while (P ≠ ∅) do
        i := popMin(P) ;
        if FindExistentialSupport(i) then
6         R := R ∪ {i} ;
7         foreach C_ij ∈ C s.t. j > i do P := P ∪ {j} ;
8     while (R ≠ ∅) do
        j := popMax(R) ;
9       foreach C_ij ∈ C s.t. i < j do
          if FindFullSupports(i, j) then
            R := R ∪ {i} ;
            S := S ∪ {i} ;
10    while (Q ≠ ∅) do
        j := popMin(Q) ;
        foreach C_ij ∈ C s.t. i > j do
          if FindSupports(i, j) then
            R := R ∪ {i} ;
            S := S ∪ {i} ;
11    foreach i ∈ X do
        if PruneVar(i) then Q := Q ∪ {i} ;
```

proof 1 *Regarding space, we use the structure suggested by [Cooper and Schiex, 2004] to bring the space complexity to $O(ed)$. [Larrosa and Schiex, 2003] proved that ProjectUnary, Project, Extend and PruneVar are time $O(d)$; FindSupports and FindFullSupports are time $O(d^2)$. Let us now focus on the **while** loop complexities in EDAC*. The loop at line 8 enforces DAC*. As described in previous papers, it is time $O(ed^2)$ because each variable is pushed in R at most once and thus each constraint C_{ij} is checked once (line 9). Similarly, the loop at line 10 which enforces AC*, directed from the lowest variables to the highest ones, is also $O(ed^2)$.*

The loop at line 5 enforces EAC directed to the highest variables. In a single run of this loop, a variable will never be pushed twice in P because the popped variable x_i is always the lowest one, and the pushed variables are higher than x_i. It implies that the **while** loop iterates at most n times, leading to a $O(e)$ time complexity for line 7. The amortized complexity time of FindExistentialSupport is $O(ed^2)$ because this function may be called with each variable, so every binary constraint at line 1 and 2 is observed once. Thus, the time complexity of EAC* is also $O(ed^2)$.*

*The line 4 is time $O(n)$. The **for** at line 11 is time $O(nd)$ (which is less than $O(ed)$, as the graph is supposed to be connected). Compiling the different results, the complexity inside the **while** at line 3 is $O(ed^2)$. It loops when:*

- *either Q is not empty: the **for** at line 11 has pruned a value and this is done at most nd times;*
- *either R is not empty: AC* has enqueued a variable in R and by the preceding remark this cannot be done more than $O(nd)$ times;*

- *or S is not empty: an element has been enqueued while enforcing AC* or DAC*. The first case cannot happen more than $O(nd)$. The second case happens $O(\max\{nd, \top\})$ times because the condition at line 8 is true when AC* has added an element in R ($O(nd)$ times) or EAC* has added an element ($O(\top)$ times because each time EAC* is violated, $C_\varnothing$ increases).*

Consequently, the overall complexity is $O(ed^2 \max\{nd, \top\})$.

6 Experimental results

In this Section, we perform an empirical comparison of EDAC* with FDAC* for the task of proving optimality. The main use of the EDAC* and FDAC* properties is to use them inside a depth-first branch and bound solver. The idea is to *maintain* EDAC* and FDAC* during the search (algorithms called MEDAC* and MFDAC*, respectively). We have implemented (C code) this idea in an efficient incremental version that uses tables of supports *à la* AC2001 [Bessiere and Regin, 2001]. In non-binary problems, we delay the propagation of non-binary constraints until they become binary. For variable selection we use the *dom/deg* heuristics which selects the variable with the smallest ratio of domain size divided by future degree. For value selection we consider values in increasing order of unary cost C_i. The variable ordering for directional arc consistency is given by their index. The optimum of each problem instance (except for MAX-SAT where only a good upper bound is provided) is given to the branch and bound as a first upper bound. We experimented with three domains easily modeled as WCSP. Given a CNF formula, **Max-SAT** is the problem of finding a complete assignment with a maximum number of satisfied clauses. We have generated and solved 80-variable 2-SAT and 40-variable 3-SAT instances with varying the number of clauses using *Cnfgen*[1], a random *k*-SAT generator. Note that this generator prevents duplicate or opposite literals in clauses but not duplicate clauses. Given an overconstrained CSP, **Max-CSP** is the problem of finding a complete assignment with a maximum number of satisfied constraints. We have generated 6 classes of binary random problems with domain size set to 10 (Sparse-Loose (constraint graph connectivity of 12% for 40 variables), Sparse-Tight, Dense-Loose (constraint graph connectivity of 25%), Dense-Tight, Complete-Loose, and Complete-Tight) as proposed in [Larrosa and Schiex, 2003]. In Max-SAT (resp. Max-CSP), samples have 10 (resp. 50) instances and we report mean values. In the **uncapacitated warehouse location problem** (UWLP) a company considers opening warehouses at some candidate locations in order to supply its existing stores. The objective is to determine which warehouses to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply costs is minimized. Each store must be supplied by exactly one open warehouse. We simply model the problem by l boolean variables for the candidate locations, s integer variables for the stores with domain size set to l, $l + s$ soft unary constraints for the costs, and $l \times s$ hard binary constraints connecting stores and warehouses. We took and solved the problem instances cap71-134 from the standard OR Library benchmarks for UWLP, as well as the MO*-MP* instances by courtesy of J. Kratica [Kratica *et al.*, 2001]. The M* instances are very challenging for mathematical programming approaches because they have a large number of suboptimal solutions.

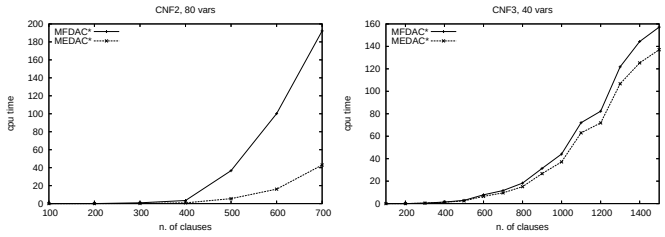

Figure 3: Time in seconds to solve random Max-2SAT and Max-3SAT problems.

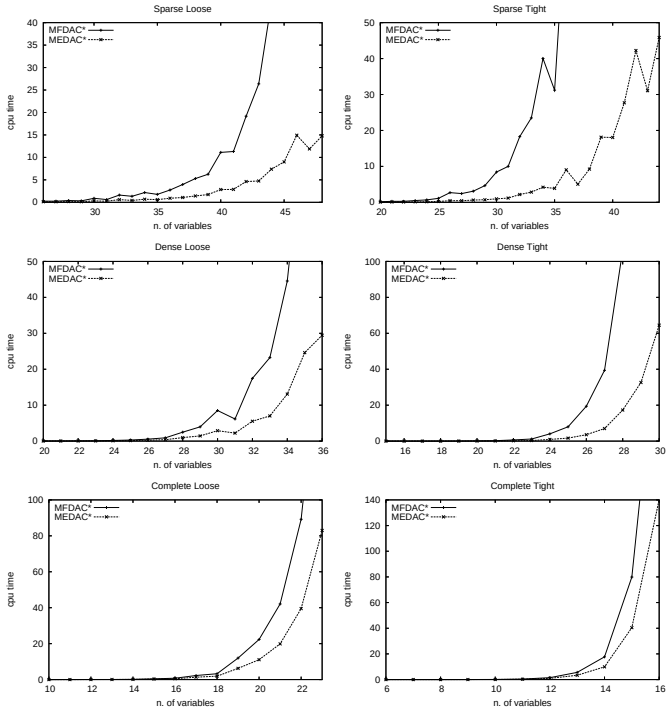

Figure 4: Time in seconds to prove optimality for binary random Max-CSPs.

The experiments were all performed on a 2.0 GHz Pentium 4 computer with 512 MB[2]. Time results in seconds are given in Figures 3 and 4, and Table 5. In all Max-SAT (resp. Max-CSP classes), the search effort seems to grow exponentially with the number of clauses (resp. number of variables). We summed the computation times for all the samples that were completely solved by both algorithms and computed

[1] A. van Gelder ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances

[2] The program named TOOLBAR and the benchmarks are available at `http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP`.

the ratio MFDAC*/MEDAC* . In Max-2SAT, MEDAC* was 5.1 faster than MFDAC*. The improvement is lower in Max-3SAT. We think this result is due to our simple delaying mechanism for tackling n-ary constraints. For Max-CSP, MEDAC* was faster than MFDAC* by a factor ranging from 2.0 (Complete-Tight) to 9.52 (Sparse-Tight). The same results were observed when analyzing the number of nodes expanded by MEDAC* and MFDAC* (not reported here for lack of space). In summary, the speed-up obtained was even more important on problems with small constraint graph connectivity.

Problem	*Size*	*MFDAC*$*$	*MEDAC*$*$	*CPLEX*
cap71	16×50	0.01	0.00	0.02
cap72	16×50	0.01	0.01	0.00
cap73	16×50	0.02	0.01	0.00
cap74	16×50	0.02	0.01	0.02
cap101	25×50	0.03	0.02	0.02
cap102	25×50	0.13	0.03	0.01
cap103	25×50	0.35	0.03	0.01
cap104	25×50	0.24	0.04	0.01
cap131	50×50	65.02	0.11	0.05
cap132	50×50	155.08	0.12	0.04
cap133	50×50	194.36	0.14	0.05
cap134	50×50	44.75	0.12	0.05
MO1	100×100	-	216.57	202.48
MO2	100×100	11775.60	44.45	36.67
MO3	100×100	15123.70	74.60	199.84
MO4	100×100	3525.61	31.74	43.26
MO5	100×100	2354.95	33.98	42.96
MP1	200×200	-	3403.37	2296.65
MP2	200×200	-	1443.93	917.92
MP3	200×200	-	981.28	1209.15
MP4	200×200	-	1768.51	1936.57
MP5	200×200	-	1099.88	697.51

Table 5: Time in seconds to prove optimality for uncapacitated warehouse location problems. Problem size is $l \times s$, with l the number of warehouses and s the number of stores. A "-" means the instance was not solved in less than 5 hours.

For UWLP, MEDAC* is several orders of magnitude faster than MFDAC* for problem size $l \geq 50$. We concluded that the lower bound computed by EDAC* for UWLP is stronger and less variable ordering dependent than the FDAC* one. Although UWLP is well-solved by dedicated mathematical programming approaches [Erlenkotter, 1978; Körkel, 1989] and heuristic search methods [Kratica *et al.*, 2001; Michel and Hentenryck, 2004], it is worth noticing that our generic WCSP algorithm was able to solve to optimality moderately-sized problems in reasonable time (in less than 1 hour). For comparison purposes, we solved UWLP instances using a state-of-the-art mixed-integer programming solver Ilog CPLEX 9.0 with default parameters (and without initial upper bounds) and a direct formulation of the problem. MEDAC* and CPLEX obtained results within the same order of magnitude in terms of cpu time.

Reversing the order of lines 5 (EAC* loop), 8 (DAC* loop), and 10 (AC* loop) in MEDAC* slowed down the algorithm by a factor $\frac{3}{2}$ on our Max-CSP benchmarks. It is important to increase the lower bound as soon as possible as it is done first by EAC*. We are currently investigating a lazy implementation of EAC* where the constraints in line 2 are projected only for the purpose of increasing the lower bound and not for pruning the values. This allows to reduce the number of variables inserted in R and P by EAC* and, consequently, the number of iterations of the different loops.

7 Conclusions and future work

In this paper we have introduced a new local consistency property called EDAC* and adapted it to WCSP. We have studied its complexity, and despite its theoretical cost, it has been shown that maintaining EDAC* is always better than maintaining FDAC* in a variety of domains. In the future, we want to study a lazy version of EDAC* and to apply it to other problems.

Acknowledgments

The authors are grateful to Thomas Schiex for useful comments on an early version of the paper and to Julia Larrosa for being born after the submission deadline.

References

[Bessiere and Regin, 2001] C. Bessiere and J.C. Regin. Refining the basic constraint propagation algorithm. In *IJCAI-01*, pages 309–315, 2001.

[Cooper and Schiex, 2004] M. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154:199–227, 2004.

[Erlenkotter, 1978] D. Erlenkotter. A Dual-Based Procedure for Uncapacitated Facility Location. *Operations Research*, 26(6):992–1009, 1978.

[Körkel, 1989] M. Körkel. On the exact solution of large-scale simple plant location problems. *European Journal of Operational Research*, 39:157–173, 1989.

[Kratica *et al.*, 2001] J. Kratica, D. Tosic, V. Filipovic, and I. Ljubic. Solving the Simple Plant Location Problems by Genetic Algorithm. *RAIRO Operations Research*, 35:127–142, 2001.

[Larrosa and Schiex, 2003] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted CSP. In *IJCAI-03*, pages 239–244, 2003.

[Larrosa and Schiex, 2004] J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc-consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.

[Michel and Hentenryck, 2004] L. Michel and P. Van Hentenryck. A Simple Tabu Search for Warehouse Location. *European Journal on Operations Research*, 157(3):576–591, 2004.

[Schiex *et al.*, 1995] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: hard and easy problems. In *IJCAI-95*, pages 631–637, 1995.