# Solving Weighted CSP by Maintaining Arc Consistency

Javier Larrosa [a,*,1] Thomas Schiex [b,2]

[a] *UPC, Jordi Girona 1-3, E-08034 Barcelona, Spain*

[b] *INRA, Chemin de Borde Rouge, BP 27, 31326 Castanet-Tolosan Cedex, France*

**Abstract**

Recently, a general definition of arc consistency (AC) for soft constraint frameworks has been proposed [1]. In this paper we specialize this definition to weighted CSP and introduce two $O(ed^3)$ enforcing algorithms. Then, we refine the definition and introduce a stronger form of arc consistency (AC*) along with two $O(n^2d^2 + ed^3)$ algorithms. As in the CSP case, an important application of AC is to combine it with search. We empirically demonstrate that a branch and bound algorithm that maintains either AC or AC* is a state-of-the-art general solver for weighted CSP. Our experiments cover binary Max-CSP and Max-SAT problems.

*Key words:* Soft constraints, Arc consistency, Branch and Bound, Combinatorial optimization

## 1 Introduction

*Constraint satisfaction problems* (CSPs) involve the assignment of *values* to *variables* subject to a set of *constraints* [2]. Since many interesting problems can be naturally modeled as CSPs, the design of efficient CSP solvers has been an active line of research in the last thirty years. Most state-of-the-art solvers can be described as generic search procedures which maintain some form of local consistency during search. This is a highly desirable feature, because it makes easier to describe, implement and compare different algorithms. There

* Corresponding author.
*Email addresses:* `larrosa@lsi.upc.es` (Javier Larrosa), `tschiex@toulouse.inra.fr` (Thomas Schiex).

[1] Research funded by the Spanish CICYT under project TIC2002-04470-C03-01.
[2] Research funded by the PICASSO French/Spanish integrated action 05158SM.

are several local consistency properties among which *arc consistency* (AC) plays a preeminent role.

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 [3,4]. Soft constraint frameworks associate costs to tuples and the goal is to find a complete assignment with minimum aggregated cost. Costs from different constraints are aggregated with a domain dependent operator. Extending the notion of local consistency to soft constraint frameworks has been a challenge in the last few years. The extension is direct as long as the aggregation operator is idempotent, but difficulties arise in the non-idempotent case. In this paper, we focus on the *weighted constraint satisfaction problem* (WCSP), a well known non-idempotent soft-constraint framework with several applications in domains such as *resource allocation* [5], *scheduling* [6], *combinatorial auctions* [7], *bioinformatics* [8], *CP networks* [9] and *probabilistic reasoning* [10]. Some current solvers for WCSP can be found in [11–15,6].

An extension of AC which can deal with non-idempotent aggregation was introduced in [1,16]. This definition has three desirable properties: (*i*) it can be enforced in polynomial time, (*ii*) the process of enforcing AC reveals unfeasible values that can be pruned and (*iii*) it reduces to the standard definition in the idempotent operator case. The only significant property lost compared to the classical case is the confluence of the enforcing process: one problem may have several arc consistent closures. In this paper, we take this work and make a new step into the practical application of its ideas. Under the usual assumption of binary problems, we introduce two AC algorithms with time complexity $O(ed^3)$ ($e$ is the number of constraints and $d$ is the largest domain size), which is an obvious improvement over the $O(e^2d^4)$ algorithm given in [1]. The algorithms are based on AC3 and AC2001, respectively. Next, we introduce an alternative stronger definition of arc consistency (AC*) along with its enforcing algorithms. Our AC* algorithms have complexity $O(n^2d^2 + ed^3)$ ($n$ is the number of variables). Although asymptotically equivalent in the general case, we show that AC2001-based algorithms improve over AC3-based algorithms under certain conditions. Some of these results were first proposed in [17].

Most current WCSP solvers require a procedural description of the work they perform at every visited node. In some cases, it is even necessary to give details of data structures and low level implementation details. Our work aims at the design of efficient yet semantically well defined solvers, where the lower bound computation done at each node is not only defined operationally, but by the enforcing of a specific property. Each AC definition yields a different solver (namely, a branch and bound which enforces at each visited node the corresponding AC definition). Since the details of how AC is enforced can be omitted, the user can understand the algorithm without getting into implementation details.

It is important to mention that our experiments indicate that, despite their conceptual simplicity, these solvers are competitive with current approaches, especially in highly over-constrained problems. Our experiments include binary Max-CSP and Max-2SAT problems. In the later, $AC$-based branch and bound can outperform specialized algorithms by orders of magnitude.

An additional contribution of this paper is a slightly modified definition of the WCSP framework which allows the specification of a maximum acceptable global cost. As we discuss, this new definition fills an existing gap between theoretical and algorithmic papers on WCSP.

The structure of the paper is as follows: Section 2 gives preliminary definitions. Section 3 motivates the use of arc-consistency with soft constraints. Sections 4 and 5 define AC and introduce two enforcing algorithms. Section 6 introduces AC* and the corresponding enforcing algorithms. Section 7 presents experimental results of using arc consistency algorithms within a branch and bound solver. Finally, Section 8 gives conclusions and directions of future work.

## 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 as $t\downarrow_B$. $\mathcal{C}$ 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 $j$. Binary constraints are bidirectional (*i.e.*, $C_{ij} \in \mathcal{C}$ iff $C_{ji} \in \mathcal{C}$ and $ab \in C_{ij}$ iff $ba \in C_{ji}$). The set of (one or two) variables affected by a constraint is called its *scope*. A tuple $t$ is *consistent* if it satisfies all the constraints whose scope is included in $\mathcal{X}_t$. It is *globally consistent* if it can be extended to a consistent complete assignment. A *solution* is a consistent complete assignment. A problem is consistent if it has at least one solution. Checking the existence of a solution in a CSP is an NP-complete problem.

**Definition 1** *[18]*

- Node consistency. $(i, a)$ *is node consistent if* $a$ *is permitted by* $C_i$ *(namely,* $a \in C_i$*). Variable* $i$ *is node consistent if all its domain values are node consistent. A CSP is* node consistent *(NC) if every variable is node consistent.*
- Arc consistency. $(i, a)$ *is arc consistent with respect to constraint* $C_{ij}$ *if it is node consistent and there is a value* $b \in D_j$ *such that* $(a, b) \in C_{ij}$*. Such a value* $b$ *is called a* support *of* $a$*. Variable* $i$ *is arc consistent if all its values are arc consistent with respect to every binary constraint involving* $i$*. A CSP is* arc consistent *(AC) if every variable is arc consistent.*

Arc inconsistent values can be removed, because they cannot participate in any solution. AC is achieved by removing arc inconsistent values until a fixed point is reached. If enforcing AC yields an empty domain, the problem has been proven inconsistent. Else, its consistency remains uncertain, although the problem size may be reduced. There is a long list of AC algorithms in the literature. Two well-known examples are: AC3 [18] and AC2001 [19], with time complexities $O(ed^3)$ and $O(ed^2)$, respectively ($e$ is the number of constraints and $d$ is the largest domain size).

State-of-the-art solvers (Ilog Solver,[3] Choco,[4] ...) perform backtracking search and enforce arc consistency at every visited node.[5] Backtracking occurs each time AC yields an empty domain. It is well recognized that such feature is fundamental to their efficiency.

## *2.2 Weighted CSPs*

*Valued* CSP [3] (as well as *semi-ring* CSP [4]) extends the classical CSP framework by associating *weights* (*costs*) to tuples. In general, costs are specified by means of a so-called *valuation structure*. A valuation structure is a triple $S = (E, \oplus, \succeq)$, where $E$ is the set of costs totally ordered by $\succeq$. The maximum and minimum costs are noted $\top$ and $\perp$, respectively. When a tuple has cost $\top$, it means that it is maximally forbidden. When a tuple has cost $\perp$, it is maximally accepted. $\oplus$ is a commutative, associative and monotonic operation on $E$ used to combine costs. $\perp$ is the identity element and $\top$ is absorbing. 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 $(a \oplus b) \succ (c \oplus b)$.

**Definition 2** *A valued CSP is a tuple* $P = (S, \mathcal{X}, \mathcal{D}, \mathcal{C})$*. The valuation structure is* $S = (E, \oplus, \succeq)$*.* $\mathcal{X}$ *and* $\mathcal{D}$ *are variables and domains, as in standard CSP.* $\mathcal{C}$ *is a set of unary and binary cost functions (namely,* $C_i : D_i \rightarrow E$*,* $C_{ij} : D_i \times D_j \rightarrow E$*).*

[3] `http://www.ilog.com.`

[4] `http://www.choco-constraints.net.`

[5] Sometimes, they enforce a weaker for of arc consistency called *bound consistency.*

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 aggregation of all applicable costs,

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

A tuple $t$ is *consistent* (also called *feasible*) if $\mathcal{V}(t) < \top$. It is *globally consistent* if it can be extended to a consistent complete assignment. A *solution* is a consistent complete assignment. The usual task of interest is to *find a solution with minimum cost*, which is NP-hard.

Weighted CSP (WCSP) [3,4] is a specific subclass of valued CSP where costs are natural numbers or infinity (*i.e.*, $E = \mathbb{N}\cup\{\infty\}$) and the operator $\oplus$ is the standard sum over the natural numbers. Clearly, in the WCSP model $\bot = 0$ and $\top = \infty$. Observe that the addition of finite costs cannot yield infinity. Therefore, it is impossible in this model to infer global inconsistency out of them. Consequently, finite costs are useless for filtering purpose where the goal is to detect and remove global inconsistent values. This is an obstacle in the characterization of useful local consistency properties leading to filtering algorithms (such as arc consistency in the classical CSP model). People working in WCSP solvers have overcome this limitation in an *ad-hoc* manner. Forbidden tuples do not receive cost $\infty$ but a finite cost $M$, where $M$ is a sufficiently high natural number. Solvers initially aim at finding assignments with cost less than $M$. When they find one assignment with a lower cost cost $M'$, they tight their target to assignments with cost less than $M'$. Therefore, all these solvers are implicitly using a modified version of the WCSP model where it is possible to express a *maximum acceptable cost*. In this modified model finite costs can be used to infer global inconsistency and, as a consequence, filter out values. The first contribution of this paper is the formalization of this variation of the WCSP model, which will be used in the sequel. First, we define the corresponding valuation structure $S(k)$, where $k$ defines the *maximum acceptable cost*.

**Definition 3** *$S(k) = ([0,1,\dots,k],\oplus,\geq)$ is a valuation structure where,*

- *$k \in \mathbb{N} - \{0\}$ denotes the maximum cost, which is a strictly positive natural number.*
- *$[0,1,\dots,k]$ is the set of costs, which are natural numbers bounded by $k$.*
- *$\oplus$ is the sum of costs. For all $a,b \in [0,1,\dots,k]$,*

  $$a \oplus b = \min\{k, a+b\}$$

- *$\geq$ is the standard order among naturals.*

**Definition 4** *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 unary and binary cost functions (namely, $C_i : D_i \to [0, \ldots, k]$, $C_{ij} : D_i \times D_j \to [0, \ldots, k]$).*

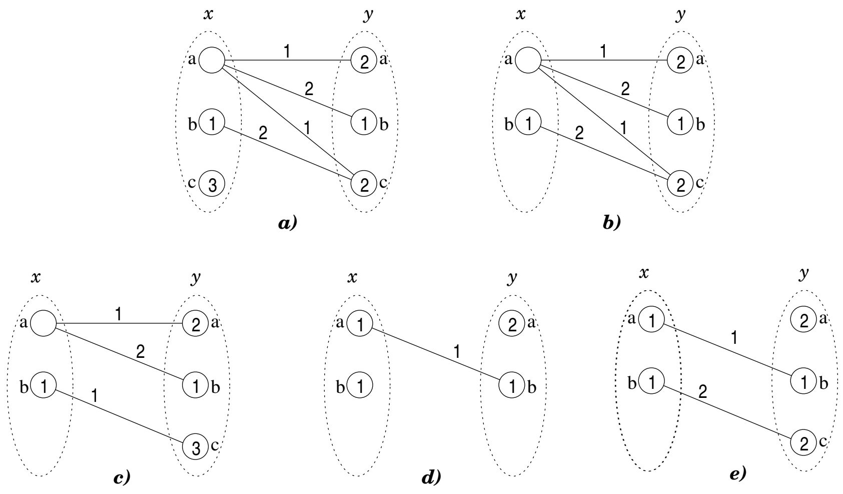


Fig. 1. Five equivalent WCSPs.

**Example 5** *Figure 1.a shows a WCSP with valuation structure $S(3)$ (namely, the set of costs is $[0, \ldots, 3]$, with $\bot = 0$ and $\top = 3$). It has two variables $\mathcal{X} = \{x, y\}$ and three values per domain $D_i = \{a, b, c\}$. There is one binary constraint $C_{xy}$ and two unary constraints $C_x$ and $C_y$. Unary costs are depicted inside a circle, beside their domain value. Binary costs are depicted as labeled edges connecting the corresponding pair of values. Only non-zero costs are shown. The problem optimal solution is the assignment of value $b$ to both variables because it has a minimum cost 2. Observe that all other complete assignments are inconsistent (their valuation is $\top$).*

Clearly, WCSP with $k = 1$ reduces to classical CSP. In addition, $S(k)$ is idempotent iff $k = 1$.

For simplicity in our exposition, we assume that every constraint has a different scope. We also assume that constraints are implemented as tables. Therefore, it is possible to consult as well as to modify entries. This is done without loss of generality: in Subsection 5.3 we will show how the addition of a $O(ed)$ data structure makes all our algorithms feasible for problems where soft constraints are cost functions given in analytical form (*e.g.*, algebraic expressions).

## 3 Solving WCSP with Branch and Bound

WCSP are usually solved with branch and bound search, which explores a tree where each node represents an assignment. Internal nodes stand for partial assignments and leaves stand for total ones. From an arbitrary node, children are obtained by selecting one unassigned variable and assigning one value for each child (an alternative is to split the variable domain into sets and consider one set per child). During search, the algorithm keeps the cost of the best solution found so far, which is an upper bound $ub$ of the problem best solution. At each node, the algorithm computes a lower bound $lb$ of the best solution in the subtree below. If $lb$ is higher than or equal to $ub$, the algorithm prunes the subtree below the current node, because the current best solution cannot be improved by extending the current assignment.

It is possible to associate a WCSP subproblem to each node of the search space. Starting from the original problem and following any branch in the search tree, each time a variable $i$ is assigned with a value $a$, cost $C_i(a)$ must be added to the lower bound and constraint $C_i$ must be removed. In addition, each constraint $C_{ij}$ must be replaced by an increment to constraint $C_j$, $C_j(b) := C_j(b) \oplus C_{ij}(a, b)$ for every $b \in D_j$. Variable $i$ is removed from the set of problem variables in the current subproblem. The valuation structure of the current subproblem is $S(ub)$, where $ub$ is the upper bound of the search.

**Algorithm 1:** Depth-first Branch and Bound

```
Function BranchAndBound($t, v_t, k, \mathcal{X}, \mathcal{D}, \mathcal{C}$)
1.  if ($\mathcal{X} = \varnothing$) then return $lb$ else
2.      $i$ :=ChooseVar($\mathcal{X}$);
3.      foreach $a \in D_i$ do
4.          $\mathcal{DD} := \mathcal{D}$; $\mathcal{CC} := \mathcal{C}$; $Nt := t + (i, a)$; $v_{Nt} := v_t \oplus C_i(a)$;
5.          LookAhead($i, a, \mathcal{CC}$);
6.          if (LocalConsist($k, \mathcal{X} - \{i\}, \mathcal{DD}, \mathcal{CC}$)) then
7.              $k$ :=BranchAndBound($Nt, v_{Nt}, k, \mathcal{X} - \{i\}, \mathcal{DD}, \mathcal{CC}$);
8.  return $k$;

Procedure LookAhead($i, a, \mathcal{CC}$)
9.  $\mathcal{CC} := \mathcal{CC} - \{C_i\}$;
10. foreach $C_{ij} \in \mathcal{CC}$ do
11.     foreach $b \in D_j$ do $C_j(b) := C_j(b) \oplus C_{ij}(a, b)$;
12.     $\mathcal{CC} := \mathcal{CC} - \{C_{ij}\}$;
```

If we have a collection of local consistency properties (such as those proposed in this paper), we can enforce any of them at every subproblem. Different choices of local consistency properties and enforcing algorithms yield different solving algorithms. If the enforcing algorithm detects global inconsistency (*e.g.*, produces an empty domain), the current subproblem does not have any solution (namely, there is no extension of the current assignment with cost below $ub$). Then, branch and bound search can prune the tree below. Otherwise, the current node must be expanded and its children must be generated. All changes made by the local consistency enforcing algorithm in the current subproblem remain in its children. As far as local consistency enforcing preserve the problem semantics (which is always assumed), this schema is valid with any search strategy, from the usual *depth, breadth* or *best first*, to more sophisticated ones. Generally, this schema can be used to solve problems using any valuation structure for which local consistency properties are available. This is the case for so-called *fair valuation structures* which have been characterized in [1,16] and which cover most practical cases including fuzzy and possibilistic CSP [20], weighted CSP [21], probabilistic CSP [22] and lexicographic CSP [3].

Algorithm 1 demonstrates the schema. It traverses the tree in a depth-first manner, which is a usual option due to its low space complexity. $t$ is the current assignment, $v_t$ its associated cost and $(k, \mathcal{X}, \mathcal{D}, \mathcal{C})$ is the current subproblem. If the current subproblem has solutions, the algorithm returns the cost of the best one. Else it returns $k$ (namely, the highest cost of the valuation structure). Procedure `LookAhead` transforms the current subproblem into a new subproblem with a new assignment $(i, a)$. Procedure `LocalConsist` enforces a given local consistency property in the current subproblem. If an empty domain is obtained, it returns *false*, else it returns *true* and possibly updates the current domains by pruning inconsistent values. Different algorithms can be obtained by replacing `LocalConsist` by a specific local consistency enforcing algorithm. Observe that $v_t$ plays the role of the lower bound and is updated in `LookAhead`. Similarly, $k$ plays the role of the upper bound and is updated in line 7 if the recursive call finds a solution with better cost.

## 4 Node and Arc Consistency in WCSP

In this Section we define AC for WCSP. Our definition is a refinement of the general definition given in [1] which takes into account the fact that $S(k)$ valuation structures have only one so-called *absorbing* element (an element $\alpha \in E$ such that $\alpha \oplus \alpha = \alpha$): the valuation $\top = k$. Our formulation also emphasizes the similarity with the CSP case (Definition 1). It will facilitate the extension of AC algorithms from CSP to WCSP. Without loss of generality, we assume the existence of a unary constraint $C_i$ for every variable (we can always define *dummy* constraints $C_i(a) = \bot, \forall a \in D_i$)

**Definition 6** *Let $P = (k, \mathcal{X}, \mathcal{D}, \mathcal{C})$ be a binary WCSP.*

- Node consistency. *$(i, a)$ is node consistent if $C_i(a) < \top$. Variable $i$ is node consistent if all its values are node consistent. $P$ is node consistent (NC) if every variable is node consistent.*
- Arc consistency. *$(i, a)$ is arc consistent with respect to constraint $C_{ij}$ if it is node consistent and there is a value $b \in D_j$ such that $C_{ij}(a, b) = \bot$. Value $b$ is called a support of $a$. Variable $i$ is arc consistent if all its values are*

*arc consistent with respect to every binary constraint affecting i. A WCSP is arc consistent (AC) if every variable is arc consistent.*

In words, node inconsistency characterizes values with an unacceptable unary cost. Its use is that they can be removed from the problem because they cannot participate in any solution. Arc inconsistency characterizes values such that extending their assignment to other variables necessarily produces a cost increment. It is worth to note at this point that, unlike the CSP case, arc inconsistent values cannot be removed in the general case, because the cost increment may not reach $\top$. However, as we will show in the next Section, arc inconsistent values may sometimes become node inconsistent. Observe that NC and AC reduce to the classical definition in standard CSP (i.e, $k = 1$).

**Example 7** *The problem in Figure 1.a is not node consistent because $C_x(c) = 3 = \top$. The problem in Figure 1.b is node consistent. However it is not arc consistent, because $(x, a)$ and $(y, c)$ do not have a support. The problems in Figure 1.d and Figure 1.e are arc consistent.*

## 5 Enforcing Arc Consistency

Arc consistency can be enforced by applying two basic operations: pruning node-inconsistent values and forcing supports to node-consistent values. As pointed out in [1], supports can be forced by *sending* costs from binary constraints to unary constraints. Let us review this concept before introducing our algorithm.

**Definition 8** *Let $a, b \in [0, \ldots, k]$, be two costs such that $a \geq b$. $a \ominus b$ is their* difference *as,*

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

The *projection* of $C_{ij} \in \mathcal{C}$ over $C_i \in \mathcal{C}$ is a flow of costs from the binary to the unary constraint where costs added to the unary constraint are *compensated* by subtracting costs from the binary constraint in order to preserve the problem semantics.

**Definition 9** *Let $\alpha_a$ be the minimum cost of $a$ with respect to $C_{ij}$,*

$$\alpha_a = \min_{b \in D_j} \{C_{ij}(a, b)\}$$

*The projection consists of adding $\alpha_a$ to $C_i(a)$,*

$$C_i(a) := C_i(a) \oplus \alpha_a, \ \forall a \in D_i$$

*and subtracting $\alpha_a$ from $C_{ij}(a,b)$,*

$$C_{ij}(a,b) := C_{ij}(a,b) \ominus \alpha_a, \ \forall b \in D_j, \forall a \in D_i$$

**Theorem 10** *[1] Let $P = (k, \mathcal{X}, \mathcal{D}, \mathcal{C})$ be a binary WCSP. The* projection *of $C_{ij} \in \mathcal{C}$ over $C_i \in \mathcal{C}$ transforms $P$ into an equivalent problem $P'$ (namely, solutions and their costs are preserved).*

Projecting binary constraints over unary constraints is a desirable transformation because it may produce new node inconsistent values which can be pruned. As a result, the problem size may decrease and infeasibility may be detected.

## *5.1 W-AC3*

Arc consistency can be achieved by pruning node-inconsistent values and projecting binary constraints over unary constraints until the property is satisfied.

**Example 11** *Consider the arc-inconsistent problem in Figure 1.a. To restore arc consistency we must prune the node-inconsistent value $c$ from $D_x$. The resulting problem (Figure 1.b) is still not arc consistent, because $(x, a)$ and $(y, c)$ do not have a support. To force a support for $(y, c)$, we project $C_{xy}$ over $C_y$. This means to add cost 1 to $C_y(c)$ and subtracting 1 from $C_{xy}(a, c)$ and $C_{xy}(b, c)$. The result of this process appears in Figure 1.c. With its unary cost increased, $(y, c)$ has lost node consistency and must be pruned. After that, we can project $C_{xy}$ over $C_x$, which yields an arc-consistent equivalent problem (Figure 1.d).*

*It is important to note that there are several arc consistent problems that can be obtained from an arc inconsistent problem. The result will depend on the order in which values are pruned and constraints are projected. For instance, if in the arc inconsistent problem of Figure 1.b $(x, a)$ is processed before $(y, c)$, the result is the arc consistent problem of Figure 1.e.*

Algorithm 2 shows W-AC3, an algorithm that enforces AC in WCSP. It is based on AC3 [18], a simple AC algorithm for CSP. The algorithm uses two procedures. Function `PruneVar(`$i$`)` prunes node-inconsistent values in $D_i$ and returns *true* if the domain is changed. Procedure `FindSupportsAC3(`$i, j$`)` projects $C_{ij}$ over $C_i$. For each value $a \in D_i$, it searches the value $v \in D_j$ with the minimum $C_{ij}(a, v)$ (line 2). If value $v$ supports $a$ (namely, $C_{ij}(a, v) = \bot$)

**Algorithm 2:** Algorithmic description of W-AC3

```
Procedure FindSupportsAC3(i, j)
1.  foreach $a \in D_i$ do
2.      $\alpha := \min_{b \in D_j}\{C_{ij}(a,b)\}$;
3.      $C_i(a) := C_i(a) \oplus \alpha$;
4.      foreach $b \in D_j$ do $C_{ij}(a,b) := C_{ij}(a,b) \ominus \alpha$;

Function PruneVar(i) : Boolean
5.  $change$ :=false;
6.  foreach $a \in D_i$ s.t. $C_i(a) = \top$ do
7.      $D_i := D_i - \{a\}$;
8.      $change$ :=true;
9.  return $change$;

Procedure W-AC3($\mathcal{X}, \mathcal{D}, \mathcal{C}$)
10. $Q := \{1, 2, \ldots, n\}$;
11. while ($Q \neq \varnothing$) do
12.     $j$ :=pop($Q$);
13.     foreach $C_{ij} \in \mathcal{C}$ do
14.         FindSupportsAC3(i, j);
15.         if PruneVar(i) then $Q := Q \cup \{i\}$;
```

lines 3 and 4 do not have any effect. Else, $v$ becomes a support for $a$ by sending costs from the binary to the unary constraint (lines 3 and 4).

The main procedure `W-AC3` has a typical AC structure. $Q$ is a set containing the variables whose domain has been pruned and therefore adjacent variables may have new unsupported values in their domains. $Q$ is initialized with all variables (line 10), because every variable must find an initial support for every value with respect to every constraints. The main loop iterates while $Q$ is not empty. An arbitrary variable $j$ is fetched from $Q$ (line 12) and for every constrained variable $i$, new supports for $D_i$ are found, if necessary (line 14). Since forcing new supports in $D_i$ may increase costs in $C_i$, node consistency in $D_i$ is checked and inconsistent values are pruned (line 15). If $D_i$ is modified, $i$ is added to $Q$, because variables connected with $i$ must have their supports revised. If during the process some domain becomes empty, the algorithm can be aborted with the certainty that the problem cannot be solved with a cost below $\top$. This fact is omitted in our description for clarity reasons.

**Theorem 12** *The time complexity of W-AC3 is time $O(ed^3)$, where e and d are the number of constraints and largest domain size, respectively.*

**PROOF.** `FindSupportsAC3(`$i, j$`)` and `PruneVar(`$i$`)` have complexity $O(d^2)$ and $O(d)$, respectively. In the main procedure, each variable $j$ is added to the set $Q$ at most $d+1$ times: once in line 10 plus at most $d$ times in line 15 (each time $D_j$ is modified). Therefore, each constraint $C_{ij}$ is considered in line 14 at most $d+1$ times. It follows that lines 14 and 15 are executed at most $2e(d+1)$ times, which yields a global complexity of $O(2e(d+1)(d^2+d)) = O(ed^3)$.

## *5.2 W-AC2001*

W-AC3 has the same time complexity as AC3, its CSP counterpart. Therefore, it seems natural to expect that extending optimal AC algorithms to WCSP will render a lower complexity $O(ed^2)$. In this section we consider AC2001, the simplest optimal AC algorithm, and introduce its natural extension to WCSP that we call W-AC2001. We show that W-AC2001 has complexity $O(ed^3)$, which gives no asymptotic advantage over W-AC3 in the general case. However, a refined complexity analysis in terms of the number of softly valuated tuples shows that W-AC2001 may be asymptotically better than W-AC3 under some conditions. In particular, it is $O(ed^2)$ in classical CSP.

Let $L_i$ be the *initial* domains of variable $i$ (*i.e,* before any value pruning) augmented with a dummy value `Nil`, which is supposed to be incompatible with any other value. Without loss of generality, we assume that $L_i$ is ordered being `Nil` the first element. Let $a \in L_i$, then `Next`$(a)$ denotes the successor of $a$ in the ordering. If $a$ is the last value of the ordering, `Next`$(a)$ returns `Nil` (*i.e,* `Next` establishes a circular ordering in $L_i$).

**Algorithm 3:** The W-AC2001 algorithm.

```
Procedure FindSupports2001($i, j$)
1. foreach ($a \in D_i$ s.t. $S(i,a,j) \notin D_j$) do
2.     $\alpha := \top$; $v :=$Nil;
3.     $b :=$Next$(S(i,a,j))$;
4.     while ($\alpha \neq \bot \wedge b \neq S(i,a,j)$) do
5.         if ($b \in D_j \wedge \alpha > C_{ij}(a,b)$) then $v := b$; $\alpha := C_{ij}(a,b)$;
6.         $b :=$Next$(b)$;
7.     $S(i,a,j) := v$;
8.     $C_i(a) := C_i(a) \oplus \alpha$;
9.     foreach $b \in D_j$ do $C_{ij}(a,b) := C_{ij}(a,b) \ominus \alpha$;
```

W-AC2001 can be obtained by replacing procedure `FindSupportsAC3(`$i,j$`)` in Algorithm 2 by `FindSupports2001(`$i,j$`)` in Algorithm 3, which computes the projection of $C_{ij}$ over $C_i$ in a more clever way. When W-AC2001 finds a support for $a \in D_i$ in constraint $C_{ij}$, such support is recorded in a data structure $S(i,a,j)$. When new supports are sought for $D_i$ in constraint $C_{ij}$, only values $a \in D_i$ such that $S(i,a,j)$ is invalid are considered (line 1). The search for a new support is done using $L_j$. It starts right after $S(i,a,j)$ (line 3). Because of the circular behavior of function `Next`, successors of $S(i,a,j)$ are considered before its predecessors. The search finishes when a supporting value is found ($\alpha = \perp$) or the whole list $L_j$ has been considered ($b = S(i,a,j)$). At the end of the search, $v$ is the value in $D_j$ giving the lowest valuation $\alpha$ to $a$, which will become the new support. The data structure $S(i,a,j)$ is updated (line 7) and costs are sent from $C_{ij}$ to $C_i$ (lines 8 and 9). All $S(i,a,j)$ must be initially set to `Nil`.

**Lemma 13** *Let $C_{ij} \in \mathcal{C}$ be an arbitrary constraint, $d$ the largest domain size, and $s_{ij}$ the number of tuples in $C_{ij}$ which receive cost different from $\perp$ and $\top$. The amortized cost of all calls of W-AC2001 to* ***`FindSupports2001`$(i,j)$*** *is $O(d \cdot s_{ij} + d^2)$.*

**PROOF.** Let's consider the total time that `FindSupports2001`$(i,j)$ spends searching supports for a fixed value $a \in D_i$. This is exactly the total number of iteration of the *while-loop* in line 4 with a fixed value $a$. Let $T_a$, $B_a$ and $S_a$ be a partition of $D_j$ such that $T_a$ contains all values such that $C_{ij}(a,b) = \top$, $B_a$ contains all values such that $C_{ij}(a,b) = \perp$ and $S_a$ contains the rest of the values. Clearly, $B_a$ is the set of original supports of $a$. $S_a$ is the set of values that may become a support if their valuation decreases to $\perp$ by means of a projection. Values in $T_a$ cannot possibly support $a$, because their valuation $\top$ cannot decrease. Procedure `FindSupports2001`$(i,j)$ finds all supports in $B_a$ in only one traversal of $D_j$, because each time a new support is needed search is resumed from the old support (as AC2001 does in the CSP case). In order to find supports from $S_a$, one traversal of $D_j$ is required each time, because the value with the minimum valuation is needed. Finally, one more traversal of $D_j$ may be required to go over values in $T_a$, before detecting that there are no more supports for $a$. As a consequence, the while-loop in line 4 traverses $D_j$ at most $2+|S_a|$ times, which is $O(d+|S_a|d)$. Summing the complexity for each $a \in D_i$, results in $O(d^2 + d\sum_{a\in D_i}|S_a|)$. By definition of $s_{ij}$, $s_{ij} = \sum_{a\in D_i}|S_a|$, which yields the final expression $O(ds_{ij} + d^2)$.

**Theorem 14** *The time complexity of W-AC2001 is $O(ed^2+sd)$, where $e$ and $d$ are the number of binary constraints and the largest domain size, respectively. Parameter $s$ is the sum over all binary constraints of the number of tuples receiving a valuation different from $\perp$ and $\top$.*

**PROOF.** Disregarding the time spent in procedure `FindSupports2001`$(i,j)$ the complexity of W-AC2001 is $O(ed^2)$. From lemma 13 we know that time spent by all calls to `FindSupports2001` is $O(\sum_{C_{ij}\in\mathcal{C}}(d\ s_{ij} + d^2))$ which, by definition of $s$, is $O(ds+ed^2)$. Thus, the total complexity is $O(ds+ed^2+ed^2) = O(ed^2 + sd)$.

**Corollary 15** *W-AC2001 has complexity $O(ed^3)$ on arbitrary WCSP instances.*

**Corollary 16** *W-AC2001 has complexity $O(ed^2)$ on classical CSP instances (a WCSP with $k = 1$)*

The previous theorem shows that W-AC2001 is asymptotically better than W-AC3 in problems where the number $s$ of binary tuples valuated with a soft cost is less than $O(ed^2)$. Corollaries 15 and 16 emphasize two important cases: $(i)$ the general WCSP case ($s$ can be as large as $O(ed^2)$) where W-AC2001 has the same complexity as W-AC3 and $(ii)$ the noteworthy case of CSP instances ($s = 0$) where W-AC2001 is $d$ times faster than W-AC3. Observe that in the CSP case W-AC2001 is as efficient as AC2001, which is optimal [19]. Other cases where W-AC2001 outperforms W-AC3 are: problems where the number of constraints with soft costs is asymptotically smaller than the number of hard constraints, and problems where the number of softly valuated tuples per constraint is asymptotically smaller than $d^2$.

## *5.3 The space complexity of enforcing W-AC*

Our description of W-AC3 (Algorithm 2) and W-AC2001 (Algorithm 3) have space complexity $O(ed^2)$, because they require binary constraints to be stored explicitly as tables, each one having $d^2$ entries. However, [16] suggested one way to avoid that. The idea is to leave the original constraints unmodified and record the changes in an additional data structure. Observe that each time a cost in a binary constraint is modified (Algorithm 2 line 4 and Algorithm 3 line 9), the whole row, or column is modified. Therefore, for each constraint we only need to record row and column changes. Let $C^0_{ij}$ denote the original constraint (before any projection) and $F(i, j, a)$ denote the total cost that has been subtracted from $C^0_{ij}(a, v)$, for all $v \in D_j$ ($F(i, j, a)$ must be initialized to $\perp$). The current value of $C_{ij}(a, b)$ can be obtained as $C^0_{ij}(a, b) \ominus F(i, j, a) \ominus F(j, i, b)$. There is an $F(i, j, a)$ entry for each constraint-value pair, therefore the required space is $O(ed)$. Since original constraints are not modified they can be given in *any* form. This is very relevant because cost functions are often given as mathematical expressions (e.g. $f(x, y) = |x-y|$) or in procedural form.

This idea can also be applied to the algorithms introduced in the following Section.

## *5.4 Other classes of problem addressed*

If the previous algorithms have been designed to enforce local consistency in WCSP, their scope of application extends beyond soft constraint networks with an additive criteria.

A first class of problems that can also be processed by the previous algorithms is defined by all problems that can be simply reduced to WCSP. A classical result is that a direct logarithmic reduction can transform probabilistic problems into additive problems (taking $c = -\log(p)$ to turn probability $p$ into cost $c$). This simple reduction allows to process probabilistic networks such as Bayesian networks [10] and probabilistic constraint networks [22] using our algorithms. Beyond this usual result, [3] shows that so-called lexicographic constraint networks, a refinement of fuzzy/possibilistic constraint networks [20] can also be reduced to WCSP.

The general frameworks of semi-ring CSP [4] and valued CSP [3] allow us to give a more general algebraic characterization of the problems that can be tackled by our algorithms. Since the assumption of a total ordering of costs, which is ubiquitous in our algorithms, suffices to reduce semi-ring CSP to valued CSP [23], we can restrict ourselves to the valued CSP case. Beyond the axioms of VCSP, our algorithms assume that a difference operator $\ominus$ is always available. Such cost structures have been defined as *fair* valuation structures in [1]. The specific refinement that our definition of arc consistency brings compared to the definition of [1] lies in the assumption that there is only one so-called *absorbing* or *idempotent* element in the valuation set $E$ (an element $\alpha \in E$ such that $\alpha \oplus \alpha = \alpha$). This element must necessarily be the valuation $\top$. Our algorithms can therefore apply to *any fair valuation structures with a single idempotent element.* In practice, the only significant class excluded by this assumption is the min-max fuzzy/possibilistic constraint network class (for which arc consistency enforcing has been already studied [24] and for which specific enforcing algorithms exist [25,20]).

This also applies to the algorithms introduced in the following Section.

## 6 Node and Arc Consistency Revisited

Consider constraint $C_x$ in the problem of Figure 1.$d$. Every value of $x$ has unary cost 1. Therefore, any assignment to $y$ will necessarily increase its cost by at least 1, if extended to $x$. Consequently, node-consistent values of $y$ are globally inconsistent if their $C_y$ cost plus 1 equals $\top$. For instance, $C_y(a)$ has cost 2 and is node consistent. But it is globally inconsistent because, no matter what value is assigned to $x$, the cost will increase to $\top$. In general, the minimum cost of all unary constraints can be summed up producing a necessary cost of any complete assignment. This idea, first suggested in [26], was ignored in the previous AC definition. Now, we integrate it into the definition of node consistency, producing an alternative definition noted NC*. We assume the

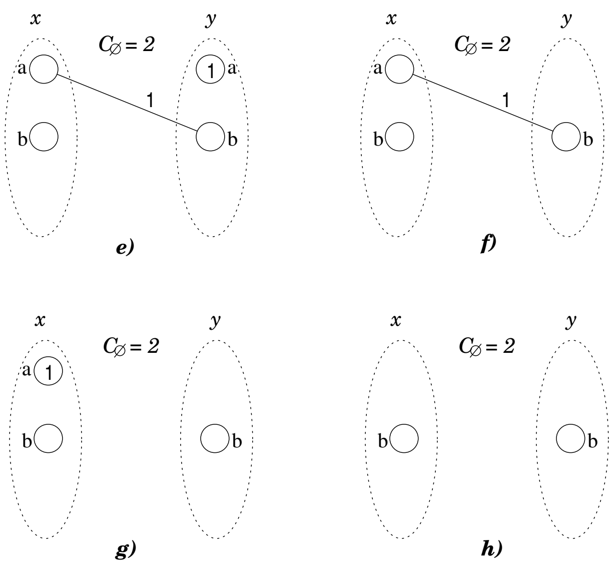


Fig. 2. Four more equivalent WCSPs.

existence of a zero-arity constraint (a constant), noted $C_\varnothing$. This is done without loss of generality, since it can be set to $\perp$ . The idea is to project unary constraints over $C_\varnothing$, which will become a global lower bound of the problem solution.

**Definition 17** *Let $P = (k, \mathcal{X}, D, \mathcal{C})$ be a binary WCSP. $(i, a)$ is node consistent if $C_\varnothing \oplus C_i(a) < \top$. Variable $i$ is node consistent if: $(i)$ all its values are node consistent and $(ii)$ there exists a value $a \in D_i$ such that $C_i(a) = \perp$. Such a value $a$ is a support for the variable node consistency. $P$ is node consistent (NC*) if every variable is node consistent.*

**Example 18** *The problem in Figure 1.d (with $C_\varnothing = 0$) does not satisfy the new definition of node consistency, because neither $x$, nor $y$ have a supporting value. Enforcing NC* requires the projection of $C_x$ and $C_y$ over $C_\varnothing$, meaning the addition of cost 2 to $C_\varnothing$, which is compensated by subtracting 1 from all entries of $C_x$ and $C_y$. The resulting problem is depicted in Figure 2.e. Now, $(y, a)$ is not node consistent, because $C_\varnothing \oplus C_y(a) = \top$ and can be removed. The resulting problem (Figure 2.$f$) is NC*.*

**Property 19** *NC* reduces to NC in classical CSP. It is stronger than NC in WCSP.*

The procedure W-NC* (Algorithm 4) enforces NC*. It works in two steps. First, a support is forced for each variable by projecting unary constraints over $C_\varnothing$ (lines 2-4). After this, every domain $D_i$ contains at least one value $a$ with $C_i(a) = \perp$. Next, node-inconsistent values are pruned (lines 5-7). The time complexity of W-NC* is $O(nd)$.

**Algorithm 4:** The W-NC* algorithm.

```
Procedure W-NC*($\mathcal{X}, \mathcal{D}, \mathcal{C}$)
1. foreach $i \in \mathcal{X}$ do
2.     $\alpha := \min_{a \in D_i}\{C_i(a)\}$;
3.     $C_\varnothing := C_\varnothing \oplus \alpha$;
4.     foreach $a \in D_i$ do $C_i(a) := C_i(a) \ominus \alpha$;
5. foreach $i \in \mathcal{X}$ do
6.     foreach $a \in D_i$ do
7.         if $C_i(a) \oplus C_\varnothing = \top$ then $D_i := D_i - \{a\}$;
```

An arc consistent problem is, by definition, node consistent. If we combine the previous definition of arc consistency (Definition 6) with the new definition of node consistency (Definition 17) we obtain a stronger form of arc consistency, noted AC*. Observe that AC* reduces to AC in the standard CSP case (Definition 1). The increased strength of AC* over AC in WCSP becomes clear in the following example.

**Example 20** *The problem in Figure 1.d is AC, but it is not AC*, because it is not NC*. As we previously showed, enforcing NC* yields the problem in Figure 2.f, where value $(x, a)$ has lost its support. Restoring it produces the problem in Figure 2.g, but now $(x, a)$ looses node consistency (with respect to NC*). Pruning the inconsistent value produces the problem in Figure 2.h, which is the problem solution.*

## *6.1 W-AC*3*

Enforcing AC* is a slightly more difficult task than enforcing AC, because: ($i$) $C_\varnothing$ has to be updated after projections of binary constraints over unary constraints, and ($ii$) each time $C_\varnothing$ is updated all domains must be checked for new node-inconsistent values. Procedure W-AC*3 (Algorithm 5) enforces AC*. Before executing W-AC*3, the problem must be made NC*. The structure of W-AC*3 is similar to W-AC3. We only discuss the main differences. Function `PruneVar` differs in that $C_\varnothing$ is considered for value pruning (line 10). Function `FindSupportsAC*3(`$i, j$`)` projects $C_{ij}$ over $C_i$ (lines 1-4) and subsequently $C_i$ is projected over $C_\varnothing$ (lines 5-7) to restore the support for the node consistency of $i$. It returns `true` if $C_\varnothing$ is increased. In the main loop, when `FindSupportsAC*3(`$i, j$`)` returns true, every variable needs to be checked for node-inconsistent values (lines 21, 22). The reason is that increments in $C_\varnothing$ may cause node-inconsistencies in any domain.

**Theorem 21** *The time complexity of W-AC*3 is $O(n^2d^2 + ed^3)$, where $n$, $e$ and $d$ are the number of variables, constraints and the largest domain size.*

**Algorithm 5:** The W-AC*3 algorithm.

```
Function FindSupportsAC*3($i, j$)
1.  foreach $a \in D_i$ do
2.      $\alpha := \min_{b \in D_j}\{C_{ij}(a,b)\}$;  $v :=$Nil;
3.      $C_i(a) := C_i(a) \oplus \alpha$;
4.      foreach $b \in D_j$ do $C_{ij}(a,b) := C_{ij}(a,b) \ominus \alpha$;
5.  $\alpha := \min_{a \in D_i}\{C_i(a)\}$;
6.  $C_\varnothing := C_\varnothing \oplus \alpha$;
7.  foreach $a \in D_i$ do $C_i(a) := C_i(a) \ominus \alpha$;
8.  return $\alpha \neq \bot$;
Function PruneVar($i$) : Boolean
9.  $change$ :=false;
10. foreach $a \in D_i$ s.t. $C_i(a) \oplus C_\varnothing = \top$ do
11.     $D_i := D_i - \{a\}$;
12.     $change$ :=true;
13. return $change$;
Procedure W-AC*3($\mathcal{X}, \mathcal{D}, \mathcal{C}$)
14. $Q := \{1, 2, \ldots, n\}$;
15. while ($Q \neq \varnothing$) do
16.     $j$ :=pop($Q$);
17.     $flag$ :=false;
18.     foreach $C_{ij} \in \mathcal{C}$ do
19.         $flag := flag \vee$ FindSupportsAC*3($i, j$);
20.         if PruneVar($i$) then $Q := Q \cup \{i\}$;
21.     if $flag$ then
22.         foreach $i \in \mathcal{X}$ s.t. PruneVar($i$) do $Q := Q \cup \{i\}$;
```

**PROOF.** `FindSupportsAC*3` and `PruneVar` have complexities $O(d^2)$ and $O(d)$, respectively. Discarding the time spent in line 22, the complexity is $O(ed^3)$ for the same reason as in W-AC3. The total time spent in line 22 is $O(n^2d^2)$, because the while loop iterates at most $nd$ times (once per domain value) and, in each iteration, line 22 may execute `PruneVar` $n$ times. Therefore, the total complexity is $O(n^2d^2 + ed^3)$.

The previous theorem indicates that enforcing AC and AC* has nearly the same worst-case complexity in dense problems and that enforcing AC can be up to $n$ times faster in sparse problems. Whether the extra effort pays off or not in terms of pruned values has to be checked empirically.

**Theorem 22** *W-AC*3 has complexity $O(ed^3)$ on classical CSP instances.*

**PROOF.** In classical CSP, $k = 1$. Consequently, $C_\varnothing$ cannot increase to intermediate values between $\perp$ and $\top$. Therefore, function `FindSupportsAC*3` always returns false, which means that line 22 is never executed. Therefore, the algorithm is essentially equivalent to W-AC3.

## *6.2 W-AC*2001*

Algorithm W-AC*2001 enforces AC* using the AC2001 schema. It is implemented by replacing the function `FindSupportsAC*3` in Algorithm 5 by the function `FindSupportsAC*2001` in Algorithm 6. It requires an additional data structure $S(i)$ containing the current support for the node-consistency of variable $i$. Before executing W-AC*2001, the problem must be made NC*. Then data structures must be initialized: $S(i, a, j)$ is set to `Nil` and $S(i)$ is set to an arbitrary supporting value (which must exist, since the problem is NC*).

**Algorithm 6:** The W-AC*2001 algorithm.

```
Function FindSupportsAC*2001($i, j$)
1.  $supported$ :=true;
2.  foreach $a \in D_i$ s.t. $S(i,a,j) \notin D_j$ do
3.      $\alpha := \top$;
4.      $b$ :=Next($S(i,a,j)$);
5.      while $(\alpha \neq \perp \wedge b \neq S(i,a,j))$ do
6.          if $(b \in D_j \wedge \alpha > C_{ij}(a,b))$ then $v := b$; $\alpha := C_{ij}(a,b)$;
7.          $b$ :=Next($b$);
8.      $S(i,a,j) := v$;
9.      $C_i(a) := C_i(a) \oplus \alpha$;
10.     foreach $b \in D_j$ do $C_{ij}(a,b) := C_{ij}(a,b) \ominus \alpha$;
11.     if $(a = S(i) \wedge \alpha \neq \perp)$ then $supported$ :=false;
12. if $\neg supported$ then
13.     $v := argmin_{a \in D_i}\{C_i(a)\}$;
14.     $\alpha := C_i(v)$;
15.     $S(i) := v$;
16.     $C_\varnothing := C_\varnothing \oplus \alpha$;
17.     foreach $a \in D_i$ do $C_i(a) := C_i(a) \ominus \alpha$;
18. return $\alpha \neq \perp$;
```

The main difference between `FindSupports2001` and `FindSupportsAC*2001` is that the latter projects the unary constraint $C_i$ over $C_\varnothing$ if the current support has been lost (Algorithm 6 lines 12-17).

**Theorem 23** *The complexity of W-AC*2001 is time $O(n^2d^2 + sd)$, where $n$ is the number of variables, $d$ is the largest domain size and $s$ is the number of binary tuples receiving cost different from $\perp$ and $\top$.*

**PROOF.** The complexity of pruning values is $O(n^2d^2)$ (see proof of Theorem 21). The complexity of finding supports is $O(sd + ed^2)$ (see proof of Theorem 14). Therefore, the total complexity is $O(n^2d^2 + sd)$.

**Corollary 24** *W-AC*2001 has complexity $O(n^2d^2 + ed^3)$ on arbitrary WCSP instances.*

**Corollary 25** *W-AC*2001 has complexity $O(ed^2)$ on classical CSP instances.*

## 7 Experimental Results

In this Section we perform an empirical evaluation of the effect of maintaining different forms of local consistency during search. All our algorithms perform depth-first search and, at every node, they enforce at least NC*. We consider the following cases:

- $BB_{NC*}$ is a basic branch and bound that only maintains NC*. It is, basically, the PFC algorithm of [26] extended to WCSP.
- $BB_{AC3}$ maintains AC using the W-AC3 algorithm.
- $BB_{AC2001}$ maintains AC using the W-AC2001 algorithm.
- $BB_{AC*3}$ maintains AC* using the W-AC*3 algorithm.
- $BB_{AC*2001}$ maintains AC* using the W-AC*2001 algorithm.

We implemented all these algorithms in C using the common BB procedure of Algorithm 1 replacing the generic call to a local consistency enforcer by a call to the corresponding algorithm. For variable selection we use the $dm/dg$ 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 $C_i$. Experiments were executed on a PC with a Pentium III processor running at 800 MHz. All our plots report *average* values over a sample of instances.

Throughout all our experiments we observed that AC2001-based algorithms were typically better (in terms of cpu time) and never worse than AC3-based algorithms. Gain ratios ranged from 1 (i.e, both approaches gave similar times) to 2, typical values being around 1.3. For the sake of clarity in the presentation of results, in the following we will omit results regarding AC3-based algorithms.

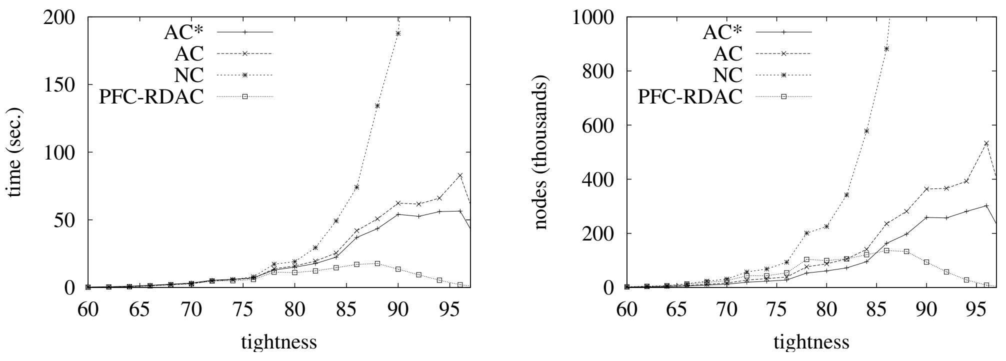


Fig. 3. Experimental results on the $\langle 20, 10, 47, t \rangle$ class with four algorithms. The plot on the left reports average cpu time in seconds. The plot on the right reports average number of visited nodes.

## *7.1 Max-CSP*

In our first set of experiments we consider the Max-CSP problem, where the goal is to find the complete assignment with a maximum number of satisfied constraints in an over-constrained CSP [26]. Max-CSP can be easily formulated as a WCSP by taking the CSP instance and replacing its hard constraints by cost functions where allowed and forbidden tuples in the hard constraint receive cost 0 and 1, respectively. The maximum acceptable cost $k$ can be set to any value larger than the number of constrains.

We experiment with binary random problems using the well-known four-parameters model [27]. A random CSP class is defined by $\langle n, d, e, t \rangle$ where $n$ is the number of variables, $d$ is the number of values per variable, $e$ 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. In all our experiments class samples have 50 instances. In this domain, we compare our algorithms with PFC-RDAC [12], which is normally considered as a reference algorithm. We used the implementation available on the web [6].

We made a preliminary experiment with the $\langle 20, 10, 47, t \rangle$ class (tightness is left as a varying parameter). Low tightness problems are solved almost instantly with optimum cost 0 (i.e, all constraints are satisfied). As $t$ approaches the crossover point, problems become abruptly over-constrained and instances become harder. Let $n$, $d$ and $e$ be fixed, we denote $t^o$ the lowest tightness where every instance in our sample has optimal cost greater than 0 (i.e, all CSP instances are inconsistent). In the $\langle 20, 10, 47, t \rangle$ class we found $t^o = 64$.

Figure 3 shows our first results. The plot on the right reports average number of

[6] `www.lsi.upc.es/~larrosa/PFC-MRDAC`

visited nodes (in thousands) and the plot on the left reports average cpu time (in seconds). As can be observed, all algorithms have a similar performance with low values of $t$, where problems are easier ($t^o$ seems to be a good representative of this region). When tightness increases differences arise, with $BB_{NC*}$ behaving extremely poorly, PFC-RDAC having a very good performance and $AC$-based algorithms lying in between. Compared with PFC-RDAC, $BB_{AC}$ and $BB_{AC*}$ show a good performance up to tightness around 85. Beyond that point, they are not competitive (with tightness around 95, PFC-RDAC is two orders of magnitude faster than the $AC$-based methods). Comparing $BB_{AC}$ and $BB_{AC*}$, the later seems to be slightly better both in terms of nodes and time.

In the following, we perform experiments to see if this behavior can be extended to different problem classes and how it scales up. For the following experiments we define different categories of problems.

- According to their tightness, we define two types of problems: *loose* (L) with $t = t^o$, and *tight* (T) with $t = \frac{3}{4}d^2 + \frac{1}{4}t^o$.
- According to the graph density we define three types of problems: *sparse* (S) with $e = 2.5n$, *dense* (D) with $e = \frac{n(n-1)}{8}$, and *completely constrained* (C) with $e = \frac{n(n-1)}{2}$.
- We consider three different domain sizes: $d = 5$, $d = 10$ and $d = 15$.

Combining the different types, we obtain 18 different classes. Each one can be specified with three symbols. For instance LD5 denotes *loose dense* problems with $d = 5$. In each class, the number of variables $n$ is not fixed and can be used as a varying parameter.

In the first experiment, we explore the performance of our algorithms on loose problems. Figure 4 shows, from top to bottom, the results obtained with LS10, LD10 and LC10 (results on problems with $d = 5$ and $d = 15$ were very similar and are omitted). Plots on the left report average cpu time and plots on the right report average number of visited nodes (note the log scale). The first observation is that, in all cases, search effort grows exponentially with $n$. The performance of all the algorithms seems to be roughly equal up to a constant factor. Regarding time, PFC-RDAC and $BB_{NC*}$ give very similar results (in the LC10 problems, they are hardly distinguishable), and the same happens with $BB_{AC}$ and $BB_{AC*}$ (except in LS10, where $BB_{AC}$ is slightly better). PFC-RDAC and $BB_{NC*}$ are faster than $BB_{AC}$ and $BB_{AC*}$ and the advantage grows with problem density (the speed-up ratio is about 1.6, 2 and 2.6, in LS10, LD10 and LC10, respectively). Regarding the number of visited nodes, performance is reversed. $BB_{AC}$ and $BB_{AC*}$ clearly visit less nodes than PFC-RDAC and $BB_{NC*}$, and the difference decreases with problem density (the gain ratio is about 2.5, 2, 1.5 in LS10, LD10 and LC10, respectively). Therefore, we conclude that in loose problems it is unnecessary to use sophisticated lower bounds, because the upper bound reaches low values early in the search which allows pruning at high levels of the search tree. While $BB_{NC*}$, visits more nodes than $AC$-based algorithms, its low overhead clearly pays-off. PFC-RDAC, which uses a greedy approach, detects very quickly that it cannot improve the lower bound and uses the same lower bound as $BB_{NC*}$. Comparing $BB_{AC}$ and $BB_{AC*}$, the former is always slightly better both in terms of time and visited nodes.

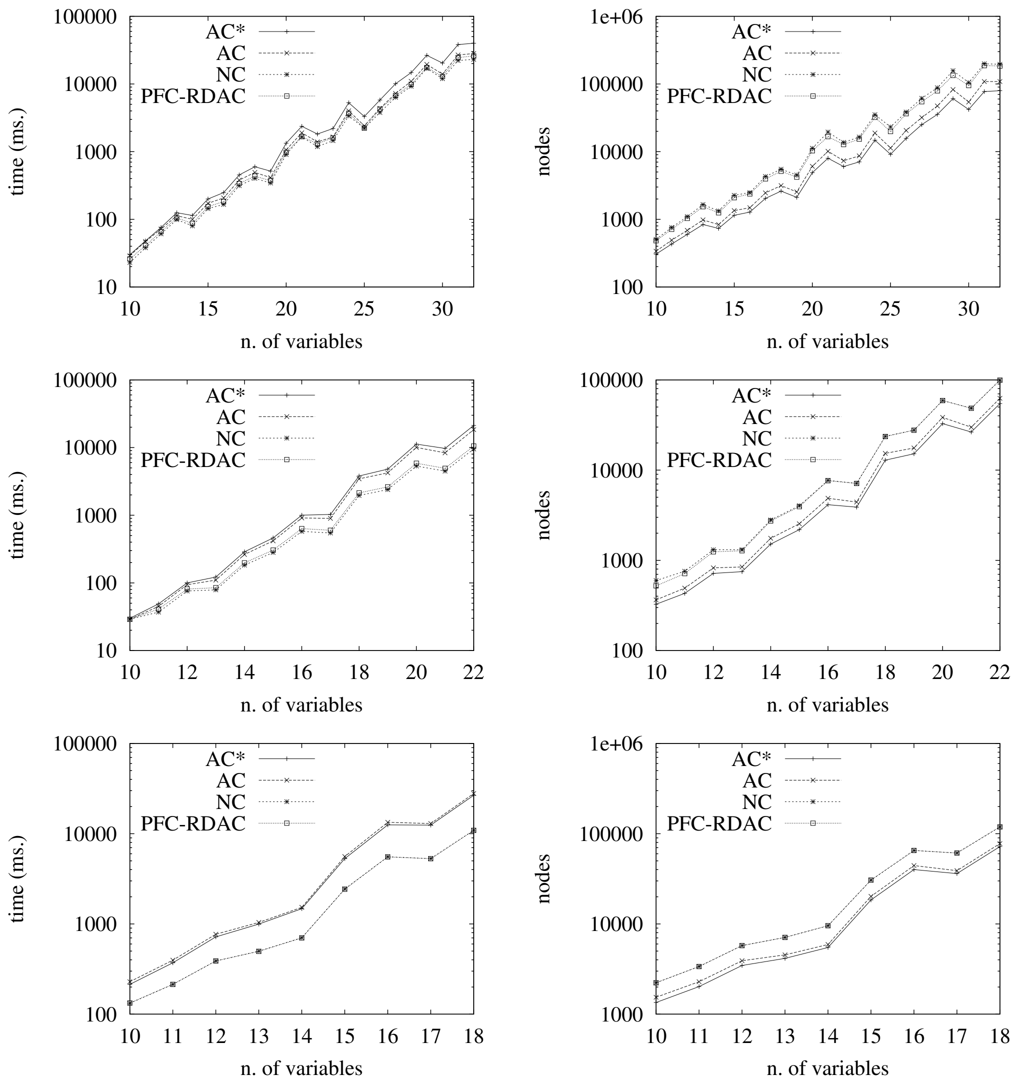


Fig. 4. Experimental results with loose problems with domain size 10 and four different algorithms. From top to bottom, plots correspond to sparse, dense and totally connected problem. Plots on the left report average time in milliseconds. Plots on the right report average number of visited nodes.

Next, we experiment with tight problems. Figure 5 shows, from top to bottom, the results obtained with TS10, TD10 and TC10 (again, we omit results with $d = 5$ and $d = 15$ because they were similar to the $d = 10$ case). Plots on the left report average cpu time and plots on the right report average number of visited nodes. Once more, we observe that search effort grows exponentially with $n$ and the performance of all the algorithms seems to be equal up to a constant factor. The main observation here is that $BB_{NC*}$ behaves very poorly. In these problems optimal solutions have higher costs, which causes a high upper bound during search. The weak lower bound of $BB_{NC*}$ is unable to prune until deep levels of the search tree. PFC-RDAC dominates $AC$-based algorithms (PFC-RDAC is about 3, 2 and 1.8 times faster than $BB_{AC*}$ in TS10, TD10 and TC10, respectively). Although being close to PFC-RDAC in terms of visited nodes, $BB_{AC}$ and $BB_{AC*}$ have a higher overhead, which makes them significatively slower in time.

It is worth to mention at this point that PFC-RDAC assigns a direction to every constraint in the current subproblem. It has been shown that the efficiency of the algorithm depends greatly on these directions. In the implementation that we are using, they are computed using a heuristic method which is very useful in random Max-CSP. Similarly, the behavior of AC-based algorithms depends on the order in which variables are fetched from the stream $Q$ and the order in which values are considered for projection (e.g. observe that not all ordering transform the problem in $1.a$ to the problem in $2.h$). In our current implementation we implemented $Q$ as a *stack* and always consider values in lexicographic order. We have not explored other alternatives, which can be more effective in this domain. A final remark on the experiments is that $BB_{AC*}$ slightly outperforms $BB_{AC}$ in time and visited nodes.

## *7.2 Max-SAT*

In our second set of experiments we consider the Max-2SAT problem, where the goal is to find the assignment that satisfies a maximum number of clauses in an 2SAT problem without any solution[7] [28]. Max-2SAT can also be formulated as a WCSP. There is a variable for each logical proposition. Each variable can be assigned with two values: the two truth assignments. There is a cost function $C_{ij}$ for each pair of variables $x_i$ and $x_j$ such that there is at least one clause referring to them. Each binary tuple receives as cost the number of clauses violated by the assignment. The maximum acceptable cost $k$ can be set to any value larger than the number of clauses.

We experiment with randomly generated 2-SAT instances. A 2-SAT class of problems is defined by the number of proposition $n$ and the number of clauses $m$. Instances are generated by randomly selecting two propositions that form each of the $m$ clauses. Propositions are negated with .5 probability. Duplication of clauses is not permitted. All samples in our experiments had 50 instances.

[7] A 2SAT problem is a SAT problem such that every clause has exactly two literals. While 2SAT can be solved in polynomial time, Max-2SAT is NP-*hard*.

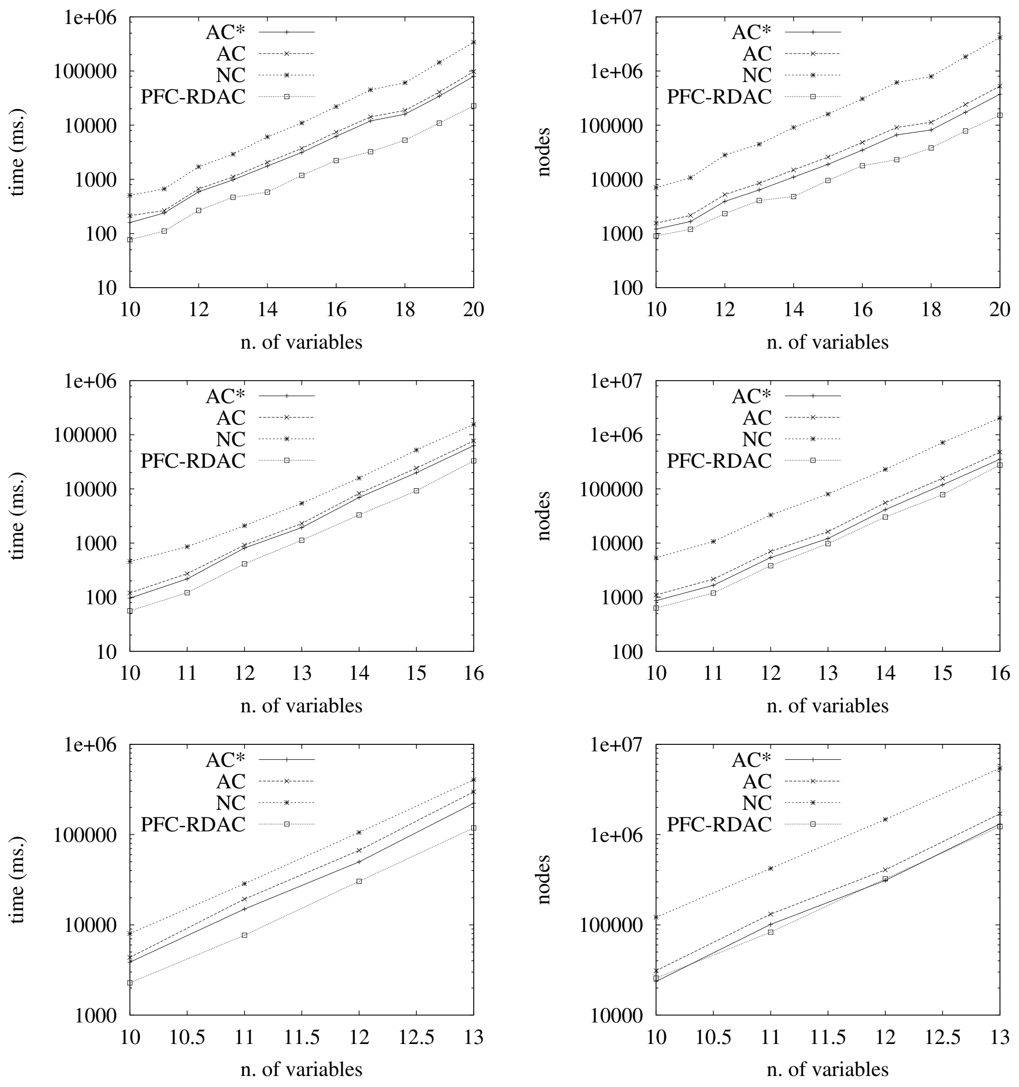


Fig. 5. Experimental results with tight problems with domain size 10 and four different algorithms. From top to bottom, plots correspond to sparse, dense and totally connected problem. Plots on the left report average time in milliseconds. Plots on the right report average number of visited nodes.

In this domain, we compare our algorithms with DPL [28], which extends the *Davis-Putnam-Loveland* procedure to Max-SAT. We used the implementation available on the web [8].

We made a preliminary experiment with problems having $n = 50$ and increasing $m$. Figure 6 plots the results. The first observation is that problems become exponentially more difficult as the number of clauses increases. $BB_{NC}$, $BB_{AC}$ and $BB_{AC*}$ seem to have a very close exponential growth. Among these three algorithms, $BB_{NC}$ shows the worst performance and $BB_{AC*}$ is slightly better than $BB_{AC}$. The differences seem to grow slowly as the number of clauses

[8] `www.nmt.edu/~borchers/maxsat.html`

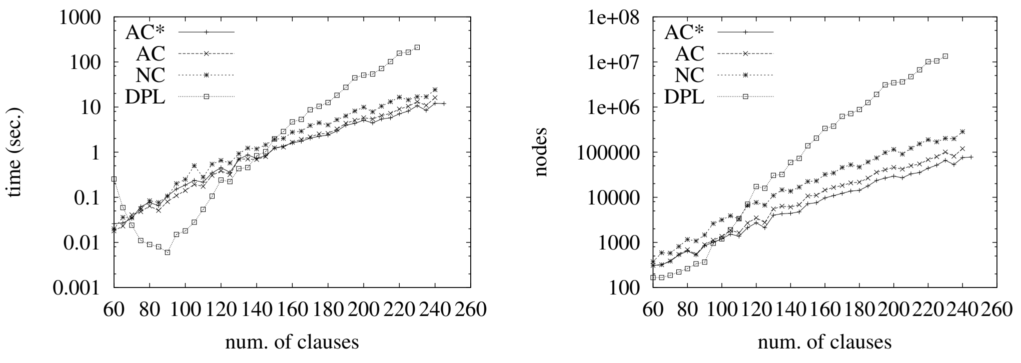


Fig. 6. Experimental results with Max-2SAT problems with 50 variables and four different algorithms. The plot on the left reports average time in seconds. The plot on the right reports average number of visited nodes.

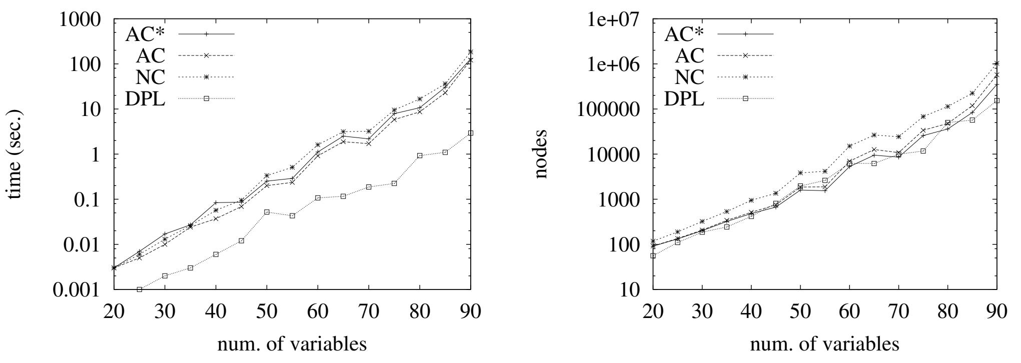
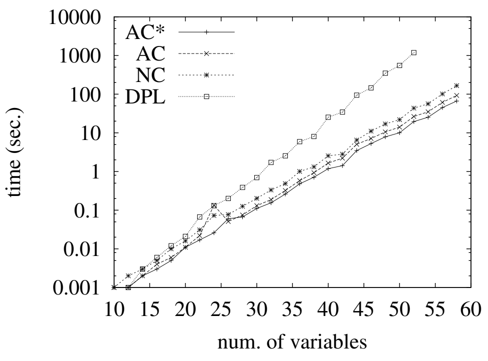


Fig. 7. Experimental results with Max-2SAT problems slightly over-constrained ($\frac{n}{m} = 2$) and four different algorithms. The plot on the left reports average time in seconds. The plot on the right reports average number of visited nodes.

increases. Apparently, $DPL$ has a much faster exponential growth. While it is the best algorithm for problems with a small $\frac{m}{n}$ ratio, it clearly stops being competitive as $\frac{m}{n}$ grows.

Next, we take a close look to problems with a low $\frac{m}{n}$ ratio. In the following experiment we let $n$ grow while keeping $\frac{m}{n} = 2$. Figure 7 shows the obtained results. Regarding time, $BB_{NC}$, $BB_{AC}$ and $BB_{AC*}$ have a very close performance, with $AC$ slightly outperforming the others. $DPL$ is clearly faster than the others (the gain ratio grows with $n$ and goes as high as 40). Regarding nodes, $BB_{AC}$, $BB_{AC*}$ and $DPL$ give very similar results, which means that $DPL$ has the same pruning power, with much less overhead.

Finally, we consider problems with a higher $\frac{m}{n}$ ratio. Now, we let $n$ grow while keeping $\frac{m}{n} = 5$. Figure 8 shows the obtained results. As anticipated by our preliminary experiment, $DPL$ is very inefficient in these problems and $AC$-based algorithms, as well as $BB_{NC}$ clearly outperform it. $BB_{AC*}$, which is the best algorithm, can be more than 300 times faster than $DPL$ (and the gain seems to increase with $n$). It is 2.5 times faster than $BB_{NC}$ and 1.5 times



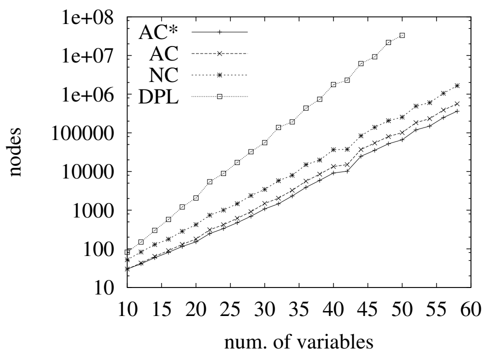


Fig. 8. Experimental results with Max-2SAT problems highly over-constrained ($\frac{n}{m} = 5$) and four different algorithms. The plot on the left reports average time in seconds. The plot on the right reports average number of visited nodes.

faster than $BB_{AC}$.

## 8 Conclusions and Future Work

We have presented two alternative forms of arc consistency for weighted CSP: AC (due to [1]) and AC*, along with their corresponding filtering algorithms: W-AC3, W-AC2001, W-AC*3 and W-AC*2001. It constitutes the first attempt to develop soft constraint solvers which *maintain some form of local consistency*. We believe that it is an appealing approach because: ($i$) it produces conceptually simple algorithms and ($ii$) it has been a great success in classical CSP.

Our definitions have the additional advantage of integrating nicely within the soft-constraints theoretical models two concepts that have been used in previous BB solvers: ($i$) the cost of the best solution found at a given point during search (upper bound in BB terminology) becomes part of the current subproblem as value $k$ in the valuation structure $S(k)$, ($ii$) the minimum necessary cost of extending the current partial assignment (lower bound in BB terminology) becomes part of the current subproblem as the zero-arity constraint $C_{\varnothing}$.

Our experiments in Max-CSP and Max-SAT show that our algorithms are competitive with state-of-the-art solvers, and sometimes even outperform them by several orders of magnitude. PFC-RDAC is still superior to our algorithms, but we are getting very close without heuristically optimizing the ordering in which we remove variables from the stream $Q$ and in which we project costs to unary costs. Since these orderings have an effect in the outcome of the AC algorithms, we expect to get closer to PFC-RDAC when considering more sophisticated methods.

We would like to mention two lines of future work, related to the main weakness of our approach. From the experiments, we observed that enforcing AC causes too much overhead in easy problems (i.e, lightly over-constrained problems). Consequently, we need to find ways to switch on the propagation only when the effort pays off. Our current implementation is restricted to binary problems. We need to extend our work (possibly using ideas from [29,30]) to non-binary problems.

## References

[1] T. Schiex, Arc consistency for soft constraints, in: CP-2000, Singapore, 2000, pp. 411–424.

[2] R. Dechter, Constraint Processing, Morgan Kaufmann, San Francisco, 2003.

[3] T. Schiex, H. Fargier, G. Verfaillie, Valued constraint satisfaction problems: hard and easy problems, in: IJCAI-95, Montréal, Canada, 1995, pp. 631–637.

[4] S. Bistarelli, U. Montanari, F. Rossi, Semiring-based constraint satisfaction and optimization, Journal of the ACM 44 (2) (1997) 201–236.

[5] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, J. Warners, Radio link frequency assignment, Constraints 4 (1999) 79–89.

[6] G. Verfaillie, M. Lemaître, T. Schiex, Russian doll search, in: AAAI-96, Portland, OR, 1996, pp. 181–187.

[7] T. Sandholm, An algorithm for optimal winner determination in combinatorial auctions, in: IJCAI-99, 1999, pp. 542–547.

[8] D. Gilbert, R. Backofen, R. Yap (Eds.), Constraints: an International Journal (Special Issue on Bioinformatics), Vol. 6(2-3), Kluwer, 2001.

[9] F. Rossi, K. B. Venable, T. Walsh, CP neworks: semantics, complexity, approximations and extensions, in: Proceedings of the 4th. International Workshop on soft constraints, Soft 02, 2002, pp. 73–86.

[10] J. Pearl, Probabilistic Inference in Intelligent Systems. Networks of Plausible Inference, Morgan Kaufmann, San Mateo, CA, 1988.

[11] R. Wallace, Enhancements of branch and bound methods for the maximal constraint satisfaction problem, in: Proc. of the $13^{th}$AAAI, Portland, OR, 1996, pp. 188–195.

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

[13] M. S. Affane, H. Bennaceur, A weighted arc consistency technique for Max-CSP, in: Proc. of the $13^{th}$ ECAI, Brighton, United Kingdom, 1998, pp. 209–213.

[14] R. Dechter, K. Kask, J. Larrosa, A general scheme for multiple lower bound computation in constraint optimization, in: CP-2001, 2001, pp. 346–360.

[15] P. Meseguer, M. Sanchez, Specializing russian doll search, in: CP, 2001, pp. 464–478.

[16] M. Cooper, T. Schiex, Arc consistency for soft constraints, Artificial Intelligence 154 (1-2) (2004) 199–227.

[17] J. Larrosa, Node and arc consistency in weighted csp, in: Proceedings of the 18th AAAI, 2002, pp. 48–53.

[18] A. Mackworth, Consistency in networks of constraints, Artificial Intelligence 8.

[19] C. Bessiere, J.-C. Regin, Refining the basic constraint propagation algorithm, in: IJCAI-2001, 2001, pp. 309–315.

[20] T. Schiex, Possibilistic constraint satisfaction problems or "How to handle soft constraints ?", in: UAI, Stanford, CA, 1992, pp. 268–275.

[21] L. Shapiro, R. Haralick, Structural descriptions and inexact matching, IEEE Transactions on Pattern Analysis and Machine Intelligence 3 (1981) 504–519.

[22] H. Fargier, J. Lang, Uncertainty in constraint satisfaction problems: a probabilistic approach, in: ECSQARU, 1993.

[23] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, Semiring-based CSPs and valued CSPs: Frameworks, properties and comparison, Constraints 4 (1999) 199–240.

[24] A. Rosenfeld, R. A. Hummel, S. W. Zucker, Scene labeling by relaxation operations, IEEE Transactions on Systems, Man, and Cybernetics 6 (6) (1976) 173–184.

[25] P. Snow, E. Freuder, Improved relaxation and search methods for approximate constraint satisfaction with a maximin criterion, in: Proc. of the $8^{th}$ biennal conf. of the canadian society for comput. studies of intelligence, 1990, pp. 227–230.

[26] E. Freuder, R. Wallace, Partial constraint satisfaction, Artificial Intelligence 58 (1992) 21–70.

[27] B. Smith, Phase transition and the mushy region in constraint satisfaction, in: Proceedings of the 11th ECAI, 1994, pp. 100–104.

[28] B. Borchers, J. Furman, A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems, Journal of Combinatorial Optimization 2 (1999) 299–306.

[29] R. Mohr, G. Masini, Good old discrete relaxation, in: ECAI, 1988, pp. 651–656.

[30] C. Bessiere, J.-C. Regin, Arc consistency for general constraint networks: Preliminary results, in: IJCAI, 1997, pp. 398–404.