Node and Arc Consistency in Weighted CSP

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

Abstract

Recently, a general definition of arc consistency (AC) for soft constraint frameworks has been proposed (Schiex 2000). In this paper we specialize this definition to weighted CSP and introduce a $O(ed^3)$ algorithm. Then, we refine the definition and introduce a stronger form of arc consistency (AC*) along with a $O(n^2d^3)$ algorithm. We empirically demonstrate that AC* is likely to be much better than AC in terms of pruned values.

Introduction

It is well known that *arc consistency* (AC) plays a preeminent role in efficient constraint solving. In the last few years, the CSP framework has been augmented with so-called *soft constraints* with which it is possible to express preferences among solutions (Schiex, Fargier, & Verfaillie 1995; Bistarelli, Montanari, & Rossi 1997). Soft constraint frameworks associate costs to tuples and the goal is to find a complete assignment with minimum combined cost. Costs from different constraints are combined with a domain dependent operator $*$. Extending the notion of AC to soft constraint frameworks has been a challenge in the last few years. From previous works we can conclude that the extension is direct as long as the operator $*$ is idempotent. Recently, (Schiex 2000) proposed an extension of AC which can deal with non-idempotent $*$. This definition has three nice 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 existing definitions in the idempotent operator case.

Weighted constraint satisfaction problems (WCSP) is a well known soft-constraint framework with a non-idempotent operator $*$. It provides a very general model with several applications in domains such as *resource allocation* (Cabon *et al.* 1999), *combinatorial auctions* (Sandholm 1999), *bioinformatics* and *probabilistic reasoning* (Pearl 1988). In recent years an important effort has been devoted to the development of efficient WCSP solvers (Freuder & Wallace 1992; Verfaillie, Lemaître, & Schiex 1996; Larrosa, Meseguer, & Schiex 1999).

In this paper, we specialize the work of (Schiex 2000) to WCSP and provide an AC algorithm 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 (Schiex 2000). Next, we introduce an alternative stronger definition of arc consistency (AC*) along with a $O(n^2d^3)$ algorithm (n is the number of variables). Our experiments on a real frequency assignment problem indicate that enforcing AC* is a promising filtering algorithm. 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.

Preliminaries

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 a assignment to a set of variables. Actually, t is an ordered set of values assigned to the ordered set of variables $\mathcal{X}_t \subseteq \mathcal{X}$ (namely, the k-th element of t is the value assigned to the k-th element of $\mathcal{X}_t$). For a subset B of $\mathcal{X}_t$, the projection of t over B is noted $t\downarrow_B$. $\mathcal{C}$ is a set of unary and binary constraints. A unary constraint C_i is a subset of D_i containing the permitted assignments to variable i. A binary constraint C_{ij} is a set of pairs from $D_i \times D_j$ containing the permitted simultaneous assigments to i and j. Binary constraints are symmetric (*i.e.*, $C_{ij} \equiv C_{ji}$). The set of (one or two) variables affected by a constraint is called its *scope*. A tuple t is *consistent* if it satisfies all constraints whose scope is included in $\mathcal{X}_t$. It is *globally consistent* if it can be extended to a complete consistent assignment. A *solution* is a consistent complete assignment. Finding a solution in a CSP is an NP-complete problem. The task of searching for a solution can be simplified by enforcing arc consistency, which identifies globally inconsistent values that can be pruned.

Definition 1 *(Mackworth 1977)*

- Node consistency. (i, a) *is node consistent if* a *is permitted by* C_i *(namely,* $a \in C_i$*). Variable* i *is node consistent*

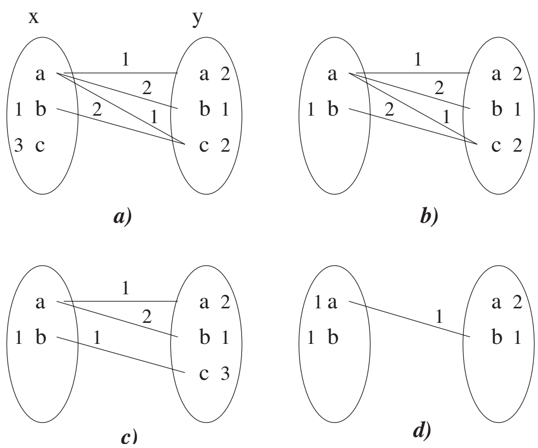

Figure 1: Four equivalent WCSPs.

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 constraint* C_{ij} *if it is node consistent and there is a value* $b \in D_j$ *such that* $(a, b) \in C_{ij}$. *Value* b *is called a* support *of* a. *Variable* i *is arc consistent if all its values are arc consistent with respect every binary constraint involving* i. *A CSP is* arc consistent *(AC) if every variable is arc consistent.*

Weighted CSPs

Valued CSP (as well as *semi-ring* CSP) extend the classical CSP framework by allowing to associate *weights* (*costs*) to tuples (Schiex, Fargier, & Verfaillie 1995; Bistarelli, Montanari, & Rossi 1997). In general, costs are specified by means of a so-called *valuation structure*. A valuation structure is a triple $S = (E, *, \succeq)$, where E is the set of costs totally ordered by $\succeq$. The maximum and a minimum costs are noted $\top$ and $\bot$, respectively. $*$ is an operation on E used to combine costs.

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

Weighted CSP (WCSP) is a specific subclass of valued CSP that rely on specific valuation structure $S(k)$.

Definition 2 $S(k)$ *is a triple* $([0, \ldots, k], \oplus, \geq)$ *where,*

- $k \in [1, \ldots, \infty]$ *is either a strictly positive natural or infinity.*
- $[0, 1, \ldots, k]$ *is the set of naturals bounded by* k.
- $\oplus$ *is the sum over the valuation structure 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 $k = \top$.

Definition 3 *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,* $C_i : D_i \to [0, \ldots, k]$, $C_{ij} : D_i \times D_j \to [0, \ldots, k]$*)*

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

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

Tuple t is *consistent* if $\mathcal{V}(t) < \top$. It is *globally consistent* if it can be extended to a complete consistent assignment. The usual task of interest is to *find a complete consistent assignment with minimum cost*, which is NP-hard. Observe that WCSP with $k = 1$ reduces to classical CSP. In addition, $S(k)$ is idempotent iff $k = 1$, and $S(k)$ is strictly monotonic iff $k = \infty$.

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 with the addition of a small data structure (see proof of Theorem 2 for details).

Example 1 *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* a, b, c. *There is one binary constraint* C_{xy} *and two unary constraints* C_x *and* C_y. *Unary costs are depicted besides their domain value. Binary costs are depicted as labelled edges connecting the corresponding pair of values. Only non-zero costs are shown. The problem solution is the assigment of value* b *to both variables because it has a minimum cost* 2.

The previous definition of WCSP differs slightly from previous papers, which may require some justification. A WCSP with $k = \infty$ has an important particularity: Since the addition of finite costs cannot yield infinity, it is impossible to infer global inconsistency out of non-infinity costs. Therefore, non-infinity costs are useless in filtering algorithms where the goal is to detect and filter out globally inconsistent values. While theoretical papers on soft constraints (Schiex, Fargier, & Verfaillie 1995; Bistarelli, Montanari, & Rossi 1997) restrict WCSP to the $k = \infty$ case (*i.e.*, they assume a strictly monotonic valuation structure), most papers on algorithms use implicitly (and exploit intensively) our definition where a finite k is permitted. For instance, most *branch and bound*-based solvers keep the cost of the best solution found so far as an upper bound ub of the maximum acceptable cost in what remains to be searched. When the algorithm detects that assigning a value to a future variable necessarily increases its cost up to ub, the value is detected as unfeasible in the current subproblem and it is pruned. Therefore, all these solvers are implicitly using the valuation structure $S(ub)$ at every subproblem.

Node and Arc Consistency in WCSP

In this Section we define AC for WCSP. Our definition is essentially equivalent to the general definition given in (Schiex 2000). However, our formulation emphasizes the similarity with the CSP case. It will facilitate the extension of

AC algorithms from CSP to WCSP. Without loss of generality, we assume the existency of a unary constraint C_i for every variable (we can always define *dummy* constraints $C_i(a) = \perp, \forall a \in D_i$)

Definition 4 *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 $C_{ij}(a, b) = \perp$. 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.*

Clearly, both NC and AC reduce to the classical definition in standard CSP.

Example 2 *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 problem in Figure 1.d is arc consistent.*

Enforcing Arc Consistency

Arc consistency can be enforced by applying two basic operations until the AC condition is satisfied: pruning node-inconsistent values and forcing supports to node-consistent values. As pointed out in (Schiex 2000), supports can be forced by *sending* costs from binary constraints to unary constraints. Let us review this concepts before introducing our algorithm.

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

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

The *projection* of $C_{ij} \in \mathcal{C}$ over $C_i \in \mathcal{C}$ is a flow of costs from the binary to the unary constraint defined as follows: Let α_a be the minimum cost of a with respect to C_{ij} (namely, $\alpha_a = \min_{b \in D_j}\{C_{ij}(a, b)\}$). The projection consists in adding α_a to $C_i(a)$ (namely, $C_i(a) := C_i(a) \oplus \alpha_a, \ \forall a \in D_i$) and subtracting α_a from $C_{ij}(a, b)$ (namely, $C_{ij}(a, b) := C_{ij}(a, b) \ominus \alpha_a, \ \forall b \in D_j, \forall a \in D_i$)

Theorem 1 *(Schiex 2000) 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'.*

Example 3 *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. That means to add cost 1 to $C_y(c)$ and substract 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 consistecy and must be pruned. After that, we can project C_{xy} over C_x, which yields an arc-consistent equivalent problem (Figure 1.d).*

Figure 2 shows W-AC2001, an algorithm that enforces AC in WCSP. It is based on AC2001 (Bessiere & Regin 2001), a simple, yet efficient AC algorithm for CSP. W-AC2001 requires a data structure $S(i, a, j)$ which stores the current support for (i, a) with respect constraint C_{ij}. Initially, $S(i, a, j)$ must be set to **Nil**, meaning that we do not know any support for a. The algorithm uses two procedures. Function `PruneVar`(i) prunes node-inconsistent values in D_i and returns **true** if the domain is changed. Procedure `FindSupports`(i, j) projects C_{ij} over C_i or, what is the same, finds (or forces) a support for every value in D_i that has lost it since the last call. The main procedure has a typical AC structure. Q is a set containing those variables such that their domain has been pruned and therefore adjacent variables may have unsupported values in their domains. Q is initialized with all variables (line 11), because every variable must find a initial supports for every domain value with respect to every constraints. The main loop iterates while Q is not empty. A variable j is fetched (line 13) and for every constrained variable i, new supports for D_i are found, if necessary (line 15). 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 16). 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 was omitted in our description for clarity reasons.

Theorem 2 *The complexity of W-AC2001 is time $O(ed^3)$ and space $O(ed)$. Parameters e and d are the number of constraints and largest domain size, respectively.*

Proof 1 *TIME: Clearly,* `FindSupports`(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 11 plus at most d times in line 16 (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 15 and 16 are executed at most $2e(d + 1)$ times, which yields a global complexity of $O(2e(d + 1)(d^2 + d)) = O(ed^3)$*

SPACE: The algorithm, as described in Figure 2, has space complexity $O(ed^2)$, because it requires binary constraints to be stored explicitely as tables, each one having d^2 entries. However, we can bring this complexity down to $O(ed)$. The idea first suggested by (Cooper & Schiex 2002) 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 (line 5), the whole row, or column is modified. Therefore, for each constraint we only need to record row and column changes. Let $F(i, j, a)$ denote the total cost that has been substracted from $C_{ij}(a, v)$, for all $v \in D_j$ ($F(i, j, a)$ must be initialized to zero). The current value of $C_{ij}(a, b)$ can be obtained as $C^0_{ij}(a, b) \ominus F(i, j, a) \ominus F(j, i, b)$, where C^0_{ij} denotes the

```
procedure FindSupports(i, j)
1.  for each a ∈ D_i if S(i, a, j) ∉ D_j do
2.      v := argmin_{b∈D_j}{C_ij(a, b)}; α := C_ij(a, v);
3.      S(i, a, j) := v;
4.      C_i(a) := C_i(a) ⊕ α;
5.      for each b ∈ D_j do C_ij(a, b) := C_ij(a, b) ⊖ α;
endprocedure
function PruneVar(i): Boolean
6.  change:=false;
7.  for each a ∈ D_i if C_i(a) = ⊤ do
8.      D_i := D_i − {a};
9.      change:=true;
10. return(change)
endfunction
procedure W-AC2001(X, D, C)
11. Q := {1, 2, ..., n};
12. while (Q ≠ ∅)do
13.     j := pop(Q);
14.     for each C_ij ∈ C do
15.         FindSupports(i, j);
16.         if PruneVar(i) then Q := Q ∪ {i};
17. endwhile
endprocedure
```

Figure 2: W-AC2001

original constraint. There is an $F(i, j, a)$ entry for each constraint-value pair, which is space $O(ed)$.

It may look surprising that AC2001 has time complexity $O(ed^2)$ (Bessiere & Regin 2001), while W-AC2001 has complexity $O(ed^3)$. The reason is that AC enforcing in WCSP is a more complex task than in CSP. In classical CSP, domains are assumed to be *ordered* sets. AC2001 records in $S(i, a, j)$ the *first* support $b \in D_j$. When a loses its support, a new support is sought *after* b in D_j, because *new supports cannot have appeared before* b. Therefore, domain D_j is traversed only once during the execution looking for supports for a. In WCSP, binary constraints C_{ij} are projected over unary constraints C_i during the W-AC2001 execution with the purpose of finding new supports for values in D_i (lines 1-5). The projection of C_{ij} over C_i decreases the costs of the binary constraint (line 5), which may create, as a side-effect, new supports for values in D_j. Therefore, each time W-AC2001 searches for a new support for value a in D_j, it needs to traverse the whole domain, because *new supports may have appeared before* a since the last call (line 2). Therefore, domain D_j may be traversed up to d times during the execution looking for supports for a.

Node and Arc Consistency Revisited

Consider constraint C_x in the problem of Figure 1.d. Any assignment to x has cost 1. Therefore, any assignment to y will necessarily increase its cost in, 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, which makes (y, a) 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

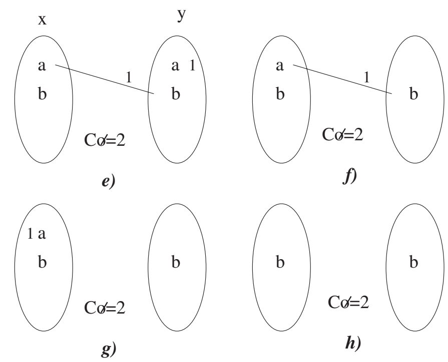

Figure 3: Four more equivalent WCSPs.

minimum cost of all unary constraints can be summed producing a necessary cost of any complete assignment. This idea, first suggested in (Freuder & Wallace 1992), 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, without loss of generality, the existence of a zero-arity constraint, noted $C_\emptyset$. A zero-arity constraint is a constant, which can be initially set to $\bot$. The idea is to project unary constraints over $C_\emptyset$, which will become a global lower bound of the problem solution.

Definition 5 *Let $P = (k, \mathcal{X}, D, \mathcal{C})$ be a binary WCSP. (i, a) is node consistent if $C_\emptyset \oplus C_i(a) < \top$. 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 $C_i(a) = \bot$. Value a is a support for the variable node consistency. P is node consistent (NC*) if every variable is node consistent.*

Example 4 *The problem in Figure 1.d (with $C_\emptyset = 0$) 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 $C_\emptyset$, meaning the addition of cost 2 to $C_\emptyset$, which is compensated by subtracting 1 from all entries of C_x and C_y. The resulting problem is depicted in Figure 3.e. Now, (y, a) is not node consistent, because $C_\emptyset \oplus C_y(a) = \top$ and can be removed. The resulting problem (Figure 3.f) is NC*.*

Property 1

NC reduces to NC in classical CSP.*

NC is stronger than NC in WCSP.*

Algorithm W-NC* (Figure 4) enforces NC*. It works in two steps. First, a support is forced for each variable by projecting unary constraints over $C_\emptyset$ (lines 1-4). After this, every domain D_i contains at least one value a with $C_i(a) = \bot$. Next, node-inconsistent values are pruned (lines 5, 6). The time complexity of W-NC* is $O(nd)$.

An arc consistent problem is, by definition, node consistent. If we take the old definition of arc consistency (Definition 4) with the new definition of node consistency, NC*, we obtain a stronger form of arc consistency, noted AC*. Its higher strength becomes clear in the following example.

procedure W-NC*($\mathcal{X}, \mathcal{D}, \mathcal{C}$)
for each $i \in \mathcal{X}$ **do**
    $v := argmin_{a \in D_i}\{C_i(a)\};\ \alpha := C_i(v);$
    $C_\emptyset := C_\emptyset \oplus \alpha;$
    for each $a \in D_i$ **do** $C_i(a) := C_i(a) \ominus \alpha;$
for each $i \in \mathcal{X}$ **for each** $a \in D_i$ **do**
    if $C_i(a) \oplus C_\emptyset = \top$ **then** $D_i := D_i - \{a\};$
endprocedure

Figure 4: W-NC*.

Example 5 *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 3.f, where value (x, a) has lost its support. Restoring it produces the problem in Figure 3.g, but now (x, a) loses node consistency (with respect to NC*). Pruning the inconsistent value produces the problem in Figure 3.h, which is the problem solution.*

Enforcing AC* is a slightly more difficult task than enforcing AC, because: (i) $C_\emptyset$ has to be updated after the projection of binary constraints over unary constraints, and (ii) each time $C_\emptyset$ is updated all domains must be revised for new node-inconsistent values. Algorithm W-AC*2001 (Figure 5) enforces AC*. 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*. After that, 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, because the problem is NC*). The structure of W-AC*2001 is similar to W-AC2001. We only discuss the main differences. Function `PruneVar` differs in that $C_\emptyset$ is considered for value pruning (line 14). Function `FindSupports`(i, j) projects C_{ij} over C_i (lines 2-7). `flag` becomes **true** if the current support of i is lost, due to an increment in its cost. In that case C_i is projected over $C_\emptyset$ (lines 9-12) to restore the support. The main loop of the algorithm differs in that finding supports and pruning values must be done independently, with separate **for** loops (lines 21 and 22). The reason is that each time $C_\emptyset$ is increased within `FindSupports`(i, j), node-inconsistencies may arise in any domain.

Theorem 3 *The complexity of W-AC*2001 is time $O(n^2d^3)$ and space $O(ed)$. Parameters n, e and d are the number of variables, constraints and largest domain size, respectively.*

Proof 2 *Regarding space, there is no difference with respect to W-AC2001, so the same proof applies. Regarding time, `FindSupports` and `PruneVar` still have complexities $O(d^2)$ and $O(d)$, respectively. Discarding the time spent in line 22, the global complexity is $O(ed^3)$ for the same reason as in W-AC2001. 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 executes `PruneVar` n times. Therefore, the total complexity is $O(ed^3 + n^2d^2)$, which is bounded by $O(n^2d^3)$.*

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.

procedure `FindSupports`(i, j)
1. `supported:=`**`true`**`;`
2. **for each** $a \in D_i$ **if** $S(i, a, j)) \notin D_j$ **do**
3. $v := argmin_{b \in D_j}\{C_{ij}(a, b)\};\ \alpha := C_{ij}(a, v);$
4. $S(i, a, j) := v;$
5. $C_i(a) := C_i(a) \oplus \alpha;$
6.   **for each** $b \in D_j$ **do** $C_{ij}(a, b) := C_{ij}(a, b) \ominus \alpha;$
7.   **if** $(a = S(i)$ **and** $\alpha \neq \bot)$ **then** `supported:=`**`false`**`;`
8. **if no**`supported` **then**
9. $v := argmin_{a \in D_i}\{C_i(a)\};\ \alpha := C_i(v)$
10. $S(i) := v;$
11. $C_\emptyset := C_\emptyset \oplus \alpha;$
12.   **for each** $a \in D_i$ **do** $C_i(a) := C_i(a) \ominus \alpha;$
endprocedure
function `PruneVar`(i): Boolean
13. `change:=`**`false`**`;`
14. **for each** $a \in D_i$ **if** $C_i(a) \oplus C_\emptyset = \top$ **do**
15. $D_i := D_i - \{a\};$
16.   `change:=`**`true`**`;`
17. **return**`(change)`
endfunction
procedure W-AC*2001($\mathcal{X}, \mathcal{D}, \mathcal{C}$)
18. $Q := \{1, 2, \ldots, n\};$
19. **while** $(Q \neq \emptyset)$**do**
20. $j := pop(Q);$
21.   **for each** $C_{ij} \in \mathcal{C}$ **do** `FindSupports`$(i, j);$
22.   **for each** $i \in \mathcal{X}$ **if** `PruneVar`(i) **do** $Q := Q \cup \{i\};$
23. **endwhile**
endprocedure

Figure 5: W-AC*2001

Experimental Results

We have tested W-AC2001 and W-AC*2001 in the frequency assignment problem domain. In particular, we have used instance 6 of the CELAR benchmark (Cabon *et al.* 1999). It is a binary WCSP instance with 100 variables, domains of up to 44 values and 400 constraints. Its solution has cost 3389. This value has been obtained using ad-hoc techniques, because the problem is too hard to be solved with a generic solver.

We generated random subproblems with the following procedure: A random tuple t is generated by randomly selecting a subset of variables $\mathcal{Q} \subset \mathcal{X}$ and assigning them with randomly selected values. $\mathcal{V}(t)$ is the cost of t in the original problem. The resulting problem P_t has $\mathcal{X} - \mathcal{Q}$ as variables, each one with its initial domain. Constraints totally instantiated by t are eliminated, binary constraints partially instantiated by t are transformed into unary constraints by *fixing* one of its arguments to the value given by t, constraints not instanciated by t are kept unmodified. The valuation structure of P_t is $S(k)$, where $k = 3389 - \mathcal{V}(t)$. In summary, we are considering the task of proving that a random partial assignment t cannot improve over the best solution.

We experimented with partial assigments of length k ranging from 0 to 30. For each k, we generated 10000 subproblems. For each subproblem, AC and AC* was enforced using W-AC2001 and W-AC*2001, respectively. Figure 6 reports average results for each value of k. The plot on the top shows the pruning power of AC *vs.* AC* as the ratio of pruned values. It can be observed that AC* prunes many more values than AC. For instance, with $k = 5$ AC* prunes 10% of values and AC prunes 1%; with $k = 10$ AC* prunes 70% of values and AC prunes 15%. AC* can prove unsolvability (i.e. there is no consistent solution) with 20 assigned variables, while AC requires 30. We observed that W-AC*2001 is more time consuming than W-AC2001 (it needs around 1.5-3 times more resources). However, this information ignores that W-AC*2001 is doing more work than W-AC2001 at each execution. A more comprehensive information is given in the second plot, which shows the tradeoff between cost and benefit. It depicts the CPU time (in miliseconds) required by each algorithm divided by the number of values that it prunes. It shows that W-AC*2001 prunes values with a lower per-value CPU cost than W-AC2001.

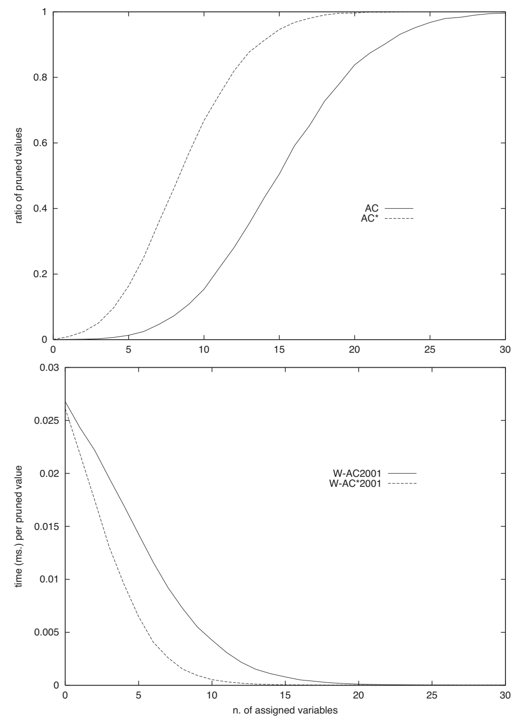

Figure 6: Experimental results on CELAR instance 6.

Conclusions and Future Work

We have presented two alternative forms of arc consistency for weighted CSP (AC (Schiex 2000) and AC*), along with their corresponding filtering algorithms (W-AC2001 and W-AC*2001). Although our algorithms may not be tuned to maximal efficiency, they could be a starting point towards an efficient branch and bound solver (BB) that maintains AC during search. Having seen the big advantage of maintaining AC in CSP (Bessiere & Regin 1996), we can expect even larger benefits in the WCSP case, because solving WCSP is much more time consuming than solving CSP. Comparing AC and AC*, our experiments seem to indicate that AC* presents a better cost-benefit trade-off.

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 definition 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) can be expressed as the initial value of the zero-arity constraint $C_\emptyset$.

Acknowledgements

I would like to thank Pedro Meseguer, Thomas Schiex and the anonymous referees for their helpful comments. Supported work by the IST Programme of the Commission of the E.U. through the ECSPLAIN project (IST-1999-11969), and by the Spanish CICYT project TAP99-1086-C03.

References

Bessiere, C., and Regin, J.-C. 1996. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. *Lecture Notes in Computer Science* 1118:61–75.

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

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

Cabon, B.; de Givry, S.; Lobjois, L.; Schiex, T.; and Warners, J. 1999. Radio link frequency assignment. *Constraints* 4:79–89.

Cooper, M., and Schiex, T. 2002. Arc consistency for soft constraints. submitted.

Freuder, E., and Wallace, R. 1992. Partial constraint satisfaction. *Artificial Intelligence* 58:21–70.

Larrosa, J.; Meseguer, P.; and Schiex, T. 1999. Maintaining reversible DAC for max-CSP. *Artificial Intelligence* 107(1):149–163.

Mackworth, A. 1977. Consistency in networks of constraints. *Artificial Intelligence* 8.

Pearl, J. 1988. *Probabilistic Inference in Intelligent Systems. Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann.

Sandholm, T. 1999. An algorithm for optimal winner determination in combinatorial auctions. In *IJCAI-99*, 542–547.

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

Schiex, T. 2000. Arc consistency for soft constraints. In *CP-2000*, 411–424.

Verfaillie, G.; Lemaître, M.; and Schiex, T. 1996. Russian doll search. In *AAAI-96*, 181–187.