A Stronger Consistency for Soft Global Constraints in Weighted Constraint Satisfaction

J.H.M. Lee and **K.L. Leung**
Department of Computer Science and Engineering
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
{jlee,klleung}@cse.cuhk.edu.hk

Abstract

Weighted Constraint Satisfaction is made practical by powerful consistency techniques, such as AC*, FDAC* and EDAC*, which reduce search space effectively and efficiently during search, but they are designed for only binary and ternary constraints. To allow soft global constraints, usually of high arity, to enjoy the same benefits, Lee and Leung give polynomial time algorithms to enforce generalized AC* (GAC*) and FDAC* (FDGAC*) for projection-safe soft non-binary constraints. Generalizing the stronger EDAC* is less straightforward. In this paper, we first reveal the oscillation problem when enforcing EDAC* on constraints sharing more than one variable. To avoid oscillation, we propose a weak version of EDAC* and generalize it to weak EDGAC* for non-binary constraints. Weak EDGAC* is stronger than FDGAC* and GAC*, but weaker than VAC and soft k-consistency for $k > 2$. We also show that weak EDGAC* can be enforced in polynomial time for projection-safe constraints. Extensive experimentation confirms the efficiency of our proposal.

Introduction

Soft constraints help model preferences and over-constrained problems. Weighted Constraint Satisfaction Problems (WCSPs), a soft CSP framework, is made practical by powerful consistency techniques applied during search. NC*, AC* and FDAC* (Larrosa and Schiex 2003; 2004) and EDAC* (de Givry et al. 2005) are instrumental in solving radio link frequency problems which are binary in nature. Generalizations of these consistencies for ternary constraints (Sanchez, de Givry, and Schiex 2008) help solve Mendelian error detection problems. Zytnicki *et al.* (2009) introduced BAC$^{\varnothing}$ for solving RNA gene localization problems.

On the other hand, many real-life problems are complex to model, requiring the use of specialized global constraints which usually have high arities. Lee and Leung (2009) generalize AC* and FDAC* to their non-binary counterparts, GAC* and FDGAC*, and show that the new consistencies can be enforced in polynomial time for projection-safe soft global constraints. A natural next step is to generalize also the stronger consistency EDAC* (de Givry et al. 2005) to EDGAC*, but this turns out to be non-trivial. We identify and analyze an inherent limitation of EDAC*: similar to the case of Full AC* (de Givry et al. 2005), ED(G)AC* enforcement will go into oscillation if two constraints share more than one variable, which is common when a problem involves high arity (soft) constraints. Sanchez *et al.* (2008) did not mention the oscillation problem but their method for enforcing EDAC* for the special case of ternary constraints would avoid the oscillation problem. In this paper, we give a weak form of EDAC*, which can be generalized to weak EDGAC* for constraints of *any* arity. Most importantly, weak EDAC* is reduced to EDAC* when no two constraints share more than one variable. Weak EDGAC* is stronger than FDGAC* and GAC* (Lee and Leung 2009), but weaker than VAC (Cooper et al. 2008) and soft k-consistency (Cooper 2005) for $k > 2$. We also give an enforcement algorithm for weak EDGAC*, which can be run in polynomial time for projection-safe (Lee and Leung 2009) soft global constraints. Extensive experimentation confirms the efficiency of our proposal both in terms of pruning and running time.

Background

A weighted CSP (WCSP) (Schiex, Fargier, and Verfaillie 1995) is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$. $\mathcal{X}$ is a set of variables $\{x_1, x_2, \ldots, x_n\}$ ordered by their indices. $\mathcal{D}$ is a set of domains $D(x_i)$ for $x_i \in \mathcal{X}$. Each x_i can only be assigned one value in its corresponding domain. An assignment $\{x_{s_1} \mapsto v_{s_1}, \ldots, x_{s_n} \mapsto v_{s_n}\}$ onto $S = \{x_{s_1}, \ldots, x_{s_n}\} \subseteq \mathcal{X}$ can be represented by a *tuple* ℓ. The notation $\ell[x_{s_i}]$ denotes the value v_{s_i} assigned to $x_{s_i} \in S$, and $\mathcal{L}(S)$ denotes a set of tuples corresponding to all possible assignments on variables S. $\mathcal{C}$ is a set of soft constraints, each C_S of which represents a function mapping a tuple $\ell \in \mathcal{L}(S)$ to a cost in the valuation structure $V(k) = ([0, \ldots, k], \oplus, \leq)$. The structure $V(k)$ contains a set of integers $[0, \ldots, k]$ with standard integer ordering $\leq$. Addition $\oplus$ is defined by $a \oplus b = min(k, a + b)$, and subtraction $\ominus$ is defined only for $a \geq b$, $a \ominus b = a - b$ if $a \neq k$ and $k \ominus a = k$ for any a. To simplify notation, we write $C_{\{x_{s_1}, x_{s_2}, \ldots, x_{s_n}\}}$ as $C_{s_1 s_2 \cdots s_n}$ if the context is clear.

Without loss of generality, we assume the existence of C_i for each $x_i \in D(x_i)$ and $C_\varnothing$ denoting the minimum cost of the problem. If they are not defined, we assume $C_i(v) = 0$

for all $v \in D(x_i)$ and $C_\varnothing = 0$. The *cost* of a tuple $\ell \in \mathcal{L}(\mathcal{X})$ is defined as $cost(\ell) = C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}} C_S(\ell[S])$, where $\ell[S]$ is the tuple formed by projecting ℓ to $S \subseteq \mathcal{X}$. A tuple $\ell \in \mathcal{L}(\mathcal{X})$ is *feasible* if $cost(\ell) < k$, and is a *solution* of a WCSP if $cost(\ell)$ is minimum among all tuples in $\mathcal{L}(\mathcal{X})$.

WCSPs are usually solved with basic branch-and-bound search augmented with consistency techniques which prune infeasible values from variable domains and push costs into $C_\varnothing$ while preserving the equivalence of the problems. Different consistency notions have been defined such as NC* (Larrosa and Schiex 2004), GAC* (Cooper and Schiex 2004; Lee and Leung 2009), and FDGAC* (Lee and Leung 2009).

A variable x_i is *node consistent* (NC*) if each value $v \in D(x_i)$ satisfies $C_i(v) \oplus C_\varnothing < k$ and there exists a value $v' \in D(x_i)$ such that $C_i(v') = 0$. A WCSP is NC* iff all variables are NC*. Procedure `enforceNC*()` in Algorithm 1 enforces NC*, where `unaryProject()` projects unary constraints towards $C_\varnothing$ and `pruneVal()` removes infeasible values.

Procedure `enforceNC*`()
    foreach $x_i \in \mathcal{X}$ **do** `unaryProject`(x_i);
    foreach $x_i \in \mathcal{X}$ **do** `pruneVal`(x_i);

Procedure `unaryProject`(x_i)
    $\alpha := \min\{C_i(v) | v \in D(x_i)\}$;
    $C_\varnothing := C_\varnothing \oplus \alpha$;
    foreach $v \in D(x_i)$ **do** $C_i(v) := C_i(v) \ominus \alpha$;

Function `pruneVal`(x_i):Boolean
    `flag` := **false** ;
    foreach $v \in D(x_i)$ *s.t.* $C_i(v) \oplus C_\varnothing = k$ **do**
        $D(x_i) := D(x_i) \setminus \{v\}$;
        `flag` := **true** ;
    return *`flag`*;

Algorithm 1: Enforce NC*

A variable $x_i \in S$ *is (G)AC** with respect to a non-unary constraint C_S if it is NC* and all values $v \in D(x_i)$ have a tuple $\ell \in \mathcal{L}(S)$ with $\ell[x_i] = v$ such that $C_S(\ell) = 0$. Such a tuple is a *simple support* of $v \in D(x_i)$ with respect to C_S. *A WCSP is (G)AC** iff all variables are (G)AC* with respect to all non-unary constraints. The procedure `findSupport` in Algorithm 2 enforces simple supports for values in $D(x_i)$ with respect to C_S, which requires time complexity exponential in $|S|$ in general. However, if the constraints are projection-safe, enforcing (G)AC* requires only polynomial time (Lee and Leung 2009).

Suppose variables are ordered by their indices. A *full support* of a value $v \in D(x_i)$ with respect to C_S with $x_i \in S$ and a set of variables $U \subseteq S$ is a tuple $\ell \in \mathcal{L}(S)$ with $\ell[x_i] = v$ such that $C_S(\ell) \oplus \bigoplus_{x_j \in U} C_j(v_j) = 0$. A variable $x_i \in S$ is *directional (generalized) arc consistent star (D(G)AC*)* with respect to a non-unary constraint C_S if it is NC* and all values $v \in D(x_i)$ have full supports with respect to C_S and and $\{x_j | j > i\} \cap S$. A WCSP is *full directional (generalized) arc consistent star (FD(G)AC*)* if all variables are D(G)AC* and (G)AC* with respect to all non-

Function `findSupport`(C_S, x_i):Boolean
    `flag` := **false** ;
    foreach $v \in D(x_i)$ **do**
        $\alpha := min\{C_S(\ell) | \ell \in \mathcal{L}(S) \wedge \ell[x_i] = v\}$;
        if $C_i(v) = 0 \wedge \alpha > 0$ **then** `flag` := **true** ;
        $C_i(v) := C_i(v) \oplus \alpha$;
        foreach *tuple* $\ell \in \mathcal{L}(S)$ *s.t.* $\ell[x_i] = v$ **do**
            $C_S(\ell) := C_S(\ell) \ominus \alpha$;
    `unaryProject`(x_i);
    return *`flag`*;

Function `findFullSupport`(C_S, x_i, U):Boolean
    foreach $x_j \in U$ **do**
        foreach $v_j \in D(x_j)$ **do**
            foreach $\ell \in \mathcal{L}(S)$ *s.t.* $\ell[x_j] = v_j$ **do**
                $C_S(\ell) := C_S(\ell) \oplus C_j(v_j)$;
            $C_j(v_j) := 0$;
    `flag` := `findSupport`(C_S, x_i);
    foreach $x_j \in U$ **do** `findSupport`(C_S, x_j);
    `unaryProject`(x_i);
    return *`flag`*;

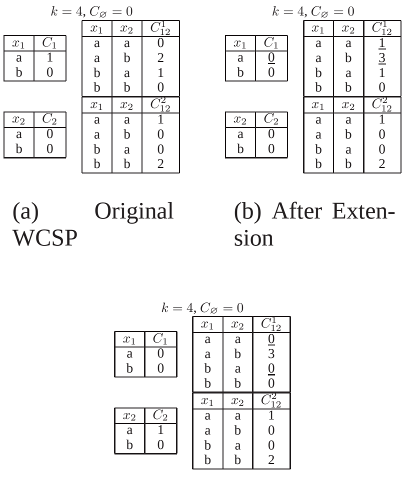
Algorithm 2: Enforcing simple supports and full supports for values in $D(x_i)$

unary constraints. The procedure `findFullSupport` in Algorithm 2 enforces full supports for values in $D(x_i)$ with respect to C_S and $U \subseteq S$, which requires time complexity exponential in $|S|$ in general. Again, if the constraints are projection-safe, enforcing FD(G)AC* requires only polynomial time (Lee and Leung 2009).

An Inherent Limitation of EDAC*

A variable is *existential arc consistent* (EAC*) if it is NC* and there exists a value $v \in D(x_i)$ with zero unary cost such that it has full supports with respect to all constraints C_{ij} and $\{x_j\}$. A WCSP is *existential directional arc consistent* (EDAC*) if it is FDAC* and all variables are EAC* (de Givry et al. 2005). Enforcing EAC* on a variable x_i requires two main operations: (1) compute $\alpha = \min_{a \in D(x_i)}\{C_i(a) \oplus \bigoplus_{C_{ij} \in \mathcal{C}} \min_{b \in D(x_j)}\{C_{ij}(a, b) \oplus C_j(b)\}\}$, which determines whether enforcing full supports breaks the NC* requirement, and (2) if $\alpha > 0$, enforce full supports by invoking `findFullSupport`(x_i, C_{ij}, $\{x_j\}$) for each $C_{ij} \in \mathcal{C}$, which NC* is no longer satisfied and hence $C_\varnothing$ can be increased by enforcing NC*.

EDAC* enforcement will oscillate with constraints sharing more than one variable. The situation is similar to Example 3 by de Givry *et al.* (2005). We demonstrate by the example in Figure 1(a), which shows a WCSP with two soft constraints C^1_{12} and C^2_{12}. It is FDAC* but not EDAC*. If x_2 takes the value a, $C^1_{12}(v, a) \oplus C_1(v) \geq 1$ for all values $v \in D(x_1)$; if x_2 takes the value b, $C^2_{12}(v, b) \oplus C_1(v) \geq 1$ for all values $v \in D(x_1)$. Thus, by enforcing full supports of each value in $D(x_2)$ with respect to all constraints and



(a) Original WCSP

(b) After Extension

(c) After Projection

Figure 1: Oscillation in EDAC* enforcement

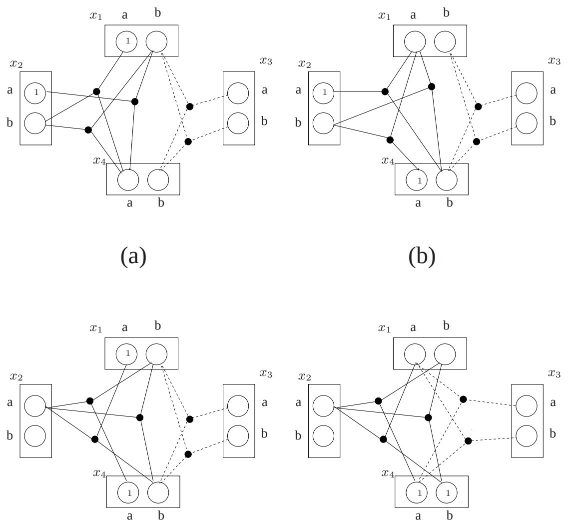


Figure 2: Four Equivalent WCSPs

$\{x_1\}$, NC* is broken and $C_\varnothing$ can be increased. To increase $C_\varnothing$, we enforce full supports: the cost of 1 in $C_1(a)$ is extended (lines 12 to 14 in Algorithm 2) to C^1_{12}, resulting in Figure 1(b). No cost in C_1 can be extended to C^2_{12}. Performing projection (lines 5 to 7 in Algorithm 2) from C^1_{12} to C_2 results in Figure 1(c). The WCSP is now EAC* but not FDAC*. Enforcing FDAC* converts the problem state back to Figure 1(a).

The problem is caused by the first step, which does not tell how the unary costs are distributed to increase $C_\varnothing$. Although an increment is predicted, the unary cost $C_1(a)$ has a choice of moving the cost to C^1_{12} or C^2_{12}. During computation, we obtain no information on how the unary costs are moved. As shown, a wrong movement breaks DAC* without incrementing $C_\varnothing$, resulting in oscillation.

This problem does not occur in existing solvers which handle only up to ternary constraints. The solvers allow only one binary constraint for every pair of variables. If there are indeed two constraints for the same two variables, the constraints can be merged into one, where the cost of a tuple in the merged constraint is the sum of the costs of the same tuple in the two original constraints. However, if we allow high arity global constraints, sharing of more than one variable would be common and necessary in many scenarios. A straightforward generalization of EDAC* for non-binary constraints would inherit the same oscillation problem. For example, Figure 2(a) shows a WCSP with two ternary constraints C_{124} and C_{134}. Each unit-cost ternary tuple is represented by three lines joined by a black dot. The WCSP is FDGAC*. With a similar argument, a lower bound of 1 should be deduced by finding full supports of x_4 with respect to C_{124} and $\{x_1, x_2\}$, and C_{134} and $\{x_1, x_3\}$. Applying full support enforcement would result in the state in Figure 2(b), but enforcing FDGAC* again will convert the problem back to the state in Figure 2(a).

In the case of ternary constraints, Sanchez *et al.* (2008) cleverly avoid the oscillation problem by re-defining full supports to include not just unary but also binary constraints. During EDAC enforcement, unary costs are distributed through extension to binary constraints. However, the method is only designed for ternary constraints. In the following, we define a weak version of EDAC*, which is based on the notion of cost-providing partitions.

Cost-Providing Partitions and Weak EDGAC*

A *cost-providing partition* $\mathcal{B}_{x_i}$ for variable $x_i \in \mathcal{X}$ is a set of sets $\{B_{x_i,C_S} | x_i \in S\}$ such that:

- $|\mathcal{B}_{x_i}|$ is the number of constraints related to x_i;
- $B_{x_i,C_S} \subseteq S$;
- $B_{x_i,C_{S_j}} \cap B_{x_i,C_{S_k}} = \varnothing$ for any two different constraints $C_{S_k}, C_{S_j} \in \mathcal{C}$, and;
- $\bigcup_{B_{x_i,C_S} \in \mathcal{B}_{x_i}} B_{x_i,C_S} = (\bigcup_{C_S \in \mathcal{C} \wedge x_i \in S} S) \setminus \{x_i\}$.

Essentially, $\mathcal{B}_{x_i}$ forms a partition of the set containing all variables related to x_i. If $x_j \in B_{x_i,C_S}$, the unary costs in C_j can only be extended to C_S when enforcing EAC* for x_i. This avoids the problem of determining how the unary costs of x_j are distributed when there exists more than one constraint on $\{x_i, x_j\}$.

Based on the cost-providing partitions, we define *weak EDAC**. Given a WCSP $P(\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$ and the cost-providing partitions $\mathcal{B}_{x_i}$ for each variable $x_i \in \mathcal{X}$. A *weak fully supported value* $v \in D(x_i)$ of a variable $x_i \in \mathcal{X}$ is a value with zero unary cost and for each variable x_j and a constraint C^m_{ij}, there exists a value $b \in D(x_j)$ such that $C^m_{ij}(v, b) = 0$ if $B_{x_i,C^m_{ij}} = \{\}$, and $C^m_{ij}(v, b) \oplus C_j(b) = 0$ if $B_{x_i,C^m_{ij}} = \{x_j\}$. A variable x_i is *weak existential arc consistent* (weak EAC*) if it is NC* and there exists at least one weak fully supported value in its domain. P is *weak existential directional arc consistent* (weak EDAC*) if it is FDAC* and each variable is weak EAC*. Weak EDAC* collapses to AC when WCSPs collapse to CSPs for any cost-providing partition. Moreover, weak EDAC* is reduced to EDAC* (de Givry et al. 2005) when the binary soft constraints share at most one variable.

We further generalize weak EDAC* to *weak EDGAC** for n-ary soft constraints. Given a WCSP $P(\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$ and any cost-providing partition $\mathcal{B}_{x_i}$ for each variable $x_i \in \mathcal{X}$. A *weak fully supported value* $v \in D(x_i)$ of a variable x_i is a value with zero unary cost and full supports with respect to all constraints $C_S \in \mathcal{C}$ with $x_i \in S$ and B_{x_i,C_S}. A variable x_i is *weak existential generalized arc consistent* (weak EGAC*) if it is NC* and there exists at least one weak fully supported value in its domain. P is *weak existential directional generalized arc consistent* (weak EDGAC*) if it is FDGAC* and each variable is weak EGAC*. For example, in Figure 2(a), the WCSP is not weak EDGAC* with the cost-providing partition $\mathcal{B}_{x_4} = \{B_{x_4,C_{124}}, B_{x_4,C_{134}}\} = \{\{x_2\}, \{x_1, x_3\}\}$. If we enforce full supports on x_4 with respect to C_{124} and $\{x_2\}$ (Figure 2(c)), and C_{123} and $\{x_1, x_3\}$ (Figure 2(d)), enforcing NC* on x_4 increases $C_\varnothing$ by 1. Given any cost-providing partition, weak EDGAC* is reduced to GAC when WCSPs collapse to CSPs.

To compute the cost-providing partition $\mathcal{B}_{x_i}$ of a variable x_i, we apply Algorithm 3, which is a greedy approach to partition the set Y containing all variables related to x_i defined in line 1, hoping to maximize $\max\{|B_{x_i,C_S}|\}$. Whether such choice deduces the highest lower bound in weak EDGAC* requires further studies.

Procedure `findCostProvidingPartition`(x_i)

1 $Y = (\bigcup_{C_S \in \mathcal{C} \wedge x_i \in S} S) \setminus \{x_i\}$;
2 Sort $\mathcal{C}$ in decreasing order of $|S|$;
3 **foreach** $C_S \in \mathcal{C}$ *s.t.* $x_i \in S$ **do**
4 $B_{x_i,C_S} = Y \cap S$;
5 $Y = Y \setminus S$;

Algorithm 3: Finding $\mathcal{B}_{x_i}$

The procedure `enforceWeakEDGAC*`() in Algorithm 4 enforces weak EDGAC* of a WCSP. The cost-providing partitions are first computed in lines 1 and 2. The procedure makes use of four propagation queues $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$ and $\mathcal{S}$. If $x_i \in \mathcal{P}$, the variable x_i is potentially not weak EGAC* due to a change in unary costs or a removal of values in some variables. If $x_j \in \mathcal{R}$, the variables x_i involving in the same constraints as x_j are potentially not DGAC*. If $x_j \in \mathcal{Q}$, all variables in the same constraints as x_j are potentially not GAC*. The propagation queue $\mathcal{S}$ helps build $\mathcal{P}$ efficiently. The procedure consists of three inner-while loops and one for-loop. The first inner-while loop from lines 6 to 10 enforces weak EGAC* on each variable by the procedure `findExistentialSupport`() in line 8. If the procedure returns true, a projection from some constraints to C_i has been performed. The weak fully supported values of other variables may be destroyed. Thus, the related variables are pushed back to $\mathcal{P}$ for revision in line 10. The second inner-while loop from lines 12 to 19 enforces DGAC*, while the third inner-while loop from lines 20 to 26 enforces GAC*. A change in unary cost requires re-examining DGAC* and weak EGAC*, which is done by pushing the variables into the corresponding queues in lines 9 and 10, and lines 18 and 19. In the last step, NC* is enforced by the for-loop from lines 28 to 31. Again, if a value in $D(x_i)$ is removed, GAC*, DGAC* or weak EGAC* may be destroyed, and x_i is pushed into the corresponding queues for re-examination.

Procedure `enforceWeakEDGAC*`()

1 **foreach** $x_i \in \mathcal{X}$ **do**
2 `findCostProvidingPartition`(x_i);
3 $\mathcal{R} := \mathcal{Q} := \mathcal{S} := \mathcal{X}$;
4 **while** $\mathcal{S} \neq \varnothing \vee \mathcal{R} \neq \varnothing \vee \mathcal{Q} \neq \varnothing$ **do**
5 $\mathcal{P} := \mathcal{S} \cup \bigcup_{x_i \in \mathcal{S}, C_S \in \mathcal{C}} (S \setminus \{x_i\})$;
6 **while** $\mathcal{P} \neq \varnothing$ **do**
7   $x_i :=$ `pop`($\mathcal{P}$);
8   **if** *`findExistentialSupport`*(x_i) **then**
9    $\mathcal{R} := \mathcal{R} \cup \{x_i\}$;
10    $\mathcal{P} := \mathcal{P} \cup \{x_j | x_i, x_j \in C_S, C_S \in \mathcal{C}\}$;
11 $\mathcal{S} := \varnothing$;
12 **while** $\mathcal{R} \neq \varnothing$ **do**
13   $x_u :=$ `popMax`($\mathcal{R}$);
14   **foreach** C_S *s.t.* $x_u \in S$ *and* $|S| > 1$ **do**
15   **for** $i = n$ **DownTo** 1 *s.t.* $x_i \in S \setminus \{x_u\}$ **do**
16     $U = \{x_j | j > i\} \cup S$;
17   **if** *`findFullSupport`*(C_S, x_i, U) **then**
18      $\mathcal{S} := \mathcal{S} \cup \{x_i\}$;
19      $\mathcal{R} := \mathcal{R} \cup \{x_i\}$;
20 **while** $\mathcal{Q} \neq \varnothing$ **do**
21   $x_u :=$ `pop`($\mathcal{Q}$);
22   `flag` := **false** ;
23   **foreach** C_S *s.t.* $x_u \in S$ *and* $|S| > 1$ **do**
24   **foreach** $x_i \in S \setminus \{x_u\}$ **do**
25   **if** *`findSupport`*(C_S, x_i) **then**
26      $\mathcal{S} := \mathcal{S} \cup \{x_i\}$;
27      $\mathcal{R} := \mathcal{R} \cup \{x_i\}$;
28 **foreach** $x_i \in \mathcal{X}$ *s.t.* *`pruneVal`*(x_i) **do**
29   $\mathcal{S} := \mathcal{S} \cup \{x_i\}$;
30   $\mathcal{Q} := \mathcal{Q} \cup \{x_i\}$;
31   $\mathcal{R} := \mathcal{R} \cup \{x_i\}$;

Function `findExistentialSupport`(x_i):Boolean

32 `flag` := **false** ;
33 $\alpha := \min_{a \in D(x_i)} \{C_i(a) \oplus \bigoplus_{x_i \in S, C_S \in \mathcal{C}} \min_{\ell[x_i]=a} \{C_S(\ell) \oplus \bigoplus_{x_j \in B_{x_i,C_S}} C_j(\ell[x_j])\}\}$;
34 **if** $\alpha > 0$ **then**
35 `flag` := **true** ;
36 **foreach** $C_S \in \mathcal{C}$ *s.t.* $x_i \in S$ **do**
37   `findFullSupport`(C_S, x_i, B_{x_i,C_S});
38 **return** *`flag`*;

Algorithm 4: Enforcing weak EDGAC*

The algorithm must terminate. We analyze the time complexity by abstracting the worst-case time complexity of the procedures `findSupport`(), `findFullSupport`() and `findExistentialSupport`() as f_{GAC}, f_{DGAC}, and f_{EGAC} respectively. The overall time complexity is stated as follows.

Theorem 1 *The procedure* `enforceWeakEDGAC*`*() in Algorithm 4 requires* $O(\max\{nd, k\}(f_{EGAC} + r^2 ef_{DGAC} + nd) + r^2 edf_{GAC})$*, where* $n = |\mathcal{X}|$*,* $d = \max\{|D(x_i)|\}$*,* $e = |\mathcal{C}|$*, and* $r = \max_{C_S \in \mathcal{C}}\{|S|\}$ *Thus,* `enforceWeakEDGAC*`*() must terminate.*

Proof: As lines 1 and 2 require only $O(nr)$, we only analyze the time complexity of each inner while-loop and compute the overall time complexity.

A variable is pushed into $\mathcal{S}$ if a value is removed or weak EGAC* is violated. The former happens $O(nd)$ times, while the latter occurs $O(k)$ times (each time weak EGAC* is violated, $C_\varnothing$ increases). Since $\mathcal{P}$ is built on $\mathcal{S}$, the number of iterations caused by $\mathcal{P}$ is $O(\max\{nd, k\})$. Thus, the first inner while-loop in line 6 requires $O(\max\{nd, k\} f_{EGAC})$.

A variable is pushed into $\mathcal{R}$ if either a value is removed, or unary costs are moved by GAC* or weak EGAC* enforcement. The number of iterations due to $\mathcal{R}$ is $O(\max\{nd, k\})$. Consider the second inner while-loop in line 11. Once a variable is popped out in line 13, it is not pushed back into $\mathcal{R}$ again by line 19. Thus, the loop only iterates $O(n)$ times. It follows that the second inner while-loop in line 12 requires $O(\max\{nd, k\} r^2 ef_{DGAC})$ (Lee and Leung 2009).

A variable is pushed into $\mathcal{Q}$ only if a value is removed. Thus, the number of iterations caused by $\mathcal{Q}$ is $O(nd)$. Thus the third while-loop in line 20 requires $O(r^2 edf_{GAC})$ (Lee and Leung 2009).

The outer while-loop in line 4 terminates when all propagation queues are empty. Thus, the main while-loop iterates $O(\max\{nd, k\})$ times. The last for-loop in line 28 requires $O(\max\{nd, k\}nd)$ times in total.

By summing up all time complexity results, the global time complexity is $O(\max\{nd, k\}(f_{EGAC} + r^2 ef_{DGAC} + nd) + r^2 edf_{GAC})$. ■

The procedure `enforceWeakEDGAC*`() is again exponential due to `findSupport`(), `findFullSupport`() and `findExistentialSupport`(). In the following, we focus on the last procedure. It first checks whether the weak fully supported value exists by computing α, which determines whether NC* still holds if we perform `findFullSupport`() from lines 36 to 37. If α equals 0, the weak fully supported value exists and nothing should be done; otherwise, the weak fully supported value can be formed by the for-loop at lines 36 to 37. The time complexity depends on two operations:

- Computing the value of α in line 33;
- Finding full supports by the line 37.

These two operations are exponential in $|S|$ in general. However, if all constraints are projection-safe, the time complexity of the above operations can be reduced to polynomial time (Lee and Leung 2009).

In the following, we compare the strength of weak EDGAC* against related consistencies. We say that α-consistency is *stronger than* β-consistency if a WCSP P is β-consistent whenever P is α-consistent. If α-consistency is stronger than β-consistency but β-consistency is not stronger than α-consistency, then α-consistency is *strictly stronger* than β-consistency.

By definition, weak EDGAC* implies FDGAC*. We have also shown an example in which the problem is FDGAC* but not weak EDGAC*. Thus, weak EDGAC* is strictly stronger than FDGAC*.

Theorem 2 *Weak EDGAC* with any cost-providing partition is strictly stronger than FDGAC*, which is in turn strictly stronger than GAC* (Lee and Leung 2009)*

In other words, enforcing FDGAC* on a problem which is already weak EDGAC* cannot further improve $C_\varnothing$ or remove more values.

In addition, VAC (Cooper et al. 2008) is strictly stronger than EDAC*. So is soft k-consistency (Cooper 2005) for $k > 2$. Since EDAC* is stronger than weak EDGAC*, we have VAC and soft k-consistency ($k > 2$) strictly stronger than weak EDGAC*.

Theorem 3 *VAC and soft k-consistency ($k > 2$) are strictly stronger than weak EDGAC* with any cost-providing partition.*

Experimental Results

To test the efficiency of weak EDGAC*, we perform the following experiments and compare it with FDGAC* (Lee and Leung 2009). Weaker than FDGAC*, GAC* and strong ∅IC (Lee and Leung 2009) are omitted due to the space limitation. VAC and soft k consistency are omitted as they have not been implemented efficiently for general n-ary constraints. Weak EDGAC* enforcement is implemented in ToulBar2[1]. The following six benchmarks are used in our experiments:

- *The Latin square problem* (CSPLIB003) of order n is to fill an $n \times n$ table using numbers from $\{0, \ldots, n-1\}$ such that each number occurs only once in every row and every column.
- *The generalized round robin tournament problem* (modified from CSPLIB026), parameterized by (N, P, W), is to schedule a tournament of N teams over W weeks, with each week divided into P periods, such that: (1) every team plays at least once a week, (2) every team plays at most twice in the same period over the tournament, and (3) every team plays every other team.
- *The fair scheduling problem*, suggested by the Global Constraint Catalog[2], is to schedule n persons into four shifts over five days such that each person should be assigned the same number of the i^{th} shift.
- *The people-mission scheduling problem*, extending the doctor-nurse rostering problem described by Beldiceanu

[1]http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/ToolBarIntro
[2]http://www.emn.fr/x-info/sdemasse/gccat/

et al. (2004) is to schedule three groups of n people into six missions such that each mission is done by a team containing exactly one person in each group. In this problem, we also place a table constraint on each team, restricting some combinations.

- *The nurse scheduling problem* is to schedule a group of n nurses into four shifts: PM shift, AM shift, Overnight, and Day-Off, over four days such that (1) each nurse must have at most three AM shifts, at least two PM shifts, at least one Overnight, and at least one Day-Off, (2) each AM shift must have two nurses, each PM shift must have one nurse, and each Overnight must have one nurse, and (3) AM-shifts are preferred to be packed together, and the same preference is also posted on Day-Offs.
- *The* `stretch` *modeling problem* consists of a sequence of variables $\{x_1, \ldots, x_n\}$ with domains $D(x_i) = \{a, b\}$. Each subsequence $\{x_i, \ldots, x_{n-5+i}\}$, where $1 \leq i \leq 5$, is required to contain a-stretches of length 2 and b-stretches of length 2 or 3, restricted using `stretch` constraints (Pesant 2001) modeled by `regular` constraints (Pesant 2004).

The above benchmarks are originally hard in nature and modeled using global constraints. We soften these problems by introducing random unary costs ranging from 0 to 9 on each variable. The hard global constraints GC are also replaced by their projection-safe soft variants $soft\_GC^{\mu}$ (Lee and Leung 2009) with different violation measures μ: var, val, $edit$ (van Hoeve, Pesant, and Rousseau 2006).

In the experiments, variables are assigned in lexicographic order. Value assignments start with the value with minimum unary cost. The test is conducted on a Dell Optiplex 280 with an Intel P4 3.2GHz CPU and 2GB RAM. The average runtime and number of search nodes of five randomly generated instances are measured for each value of n with no initial upper bound. Table 1 gives the results. Entries marked with a "*" indicates the execution of one of the five instances exceeds 1 hour. We also mark the best results by the '†' symbol.

Among all instances, weak EDGAC* always prune more than FDGAC*, bettering up to one order of magnitude in the recorded figures. FDGAC* cannot solve a few instances within the time limit, but weak EDGAC* can. This confirms empirically weak EDGAC*'s theoretical strength as stated in Theorem 2. There are three types of timing results. In Tables 1(a), 1(b), 1(c), 1(d), and 1(g), weak EDGAC* beats FDGAC* in all instances. In particular, FDGAC* cannot solve some of the larger and more difficult instances within the time limit. In Tables 1(e), 1(f), and 1(h), FDGAC* always beats weak EDGAC*, but by a small margin only. In Tables 1(i) and 1(j), FDGAC* wins in the smaller instances. In the larger instances, the effort in doing the extra pruning finally pays off and weak EDGAC* prevails. This suggests that weak EDGAC* has a better scaling behavior.

In summary, although weak EDGAC* is a more expensive consistency to enforce in general, the additional pruning can usually compensate for the extra effort.

n	FDGAC*		Weak EDGAC*	
	Time(s)	Nodes	Time(s)	Nodes
4	†0.1	22	†0.1	†17.0
5	†0.1	66.2	†0.1	†48.2
6	4.8	244.6	†1.2	†87.0
7	58.4	1431.2	†16.4	†331.8
8	*	*	†459.6	†4730.8

(a) Latin Square with `soft_GCC`^{var}

n	FDGAC*		Weak EDGAC*	
	Time(s)	Nodes	Time(s)	Nodes
4	†0.1	20.4	†0.1	†17.0
5	†0.1	61.2	†0.1	†45.2
6	3.6	211.0	†1.0	†82.2
7	40.4	1243.6	†13.4	†318.4
8	*	*	†285.2	†3700.4

(b) Latin Square with `soft_GCC`^{val}

(N, P, W)	FDGAC*		Weak EDGAC*	
	Time(s)	Nodes	Time(s)	Nodes
(4,3,2)	0.2	142.2	†0.1	†33.4
(5,4,2)	0.6	171.6	†0.1	†44.6
(6,5,3)	*	*	†583.4	†6508.8
(7,5,3)	*	*	†1283.4	†7476.6

(c) Generalized Round Robin Tournament with `soft_GCC`^{var}

(N, P, W)	FDGAC*		Weak EDGAC*	
	Time(s)	Nodes	Time(s)	Nodes
(4,3,2)	0.2	141.0	†0.1	†33.0
(5,4,2)	0.6	171.0	†0.1	†42.8
(6,5,3)	*	*	†438.2	†6499.6
(7,5,3)	*	*	†765.0	†7413.6

(d) Generalized Round Robin Tournament with `soft_GCC`^{val}

n	FDGAC*		Weak EDGAC*	
	Time(s)	Nodes	Time(s)	Nodes
5	†0.1	27.4	†0.1	†25.4
6	†0.4	40.4	1.0	†34.0
7	†1.0	45.0	1.2	†40.6
8	†2.0	45.4	2.2	†45.0
9	†2.6	49.0	3.2	†49.0
10	†4.0	58.0	4.6	†56.8
11	†5.8	67.2	6.4	†61.6

(e) Fair Scheduling with `soft_same`^{var}

n	FDGAC*		weak EDGAC*	
	Time(s)	Nodes	Time(s)	Nodes
4	†0.2	247.8	0.4	†238.8
5	†3.4	831.2	†3.4	†693.4
6	†55.6	11065.2	69.2	†10957.8
7	†1348.0	333937.6	1714.0	†296019.2

(f) People-mission Scheduling with `soft_same`^{var}

n	FDGAC*		weak EDGAC*	
	Time(s)	Nodes	Time(s)	Nodes
3	†0.1	28.6	†0.1	†22.8
4	†0.1	32.6	†0.1	†28.0
5	4.0	379.0	†3.6	†273.6
6	63.4	4017.6	†37.8	†1927.2
7	207.6	12242.0	†42.8	†2167.6
8	821.2	44414.0	†229.2	†10437.0

(g) Nurse Rostering with `soft_regular`^{var}()

n	FDGAC*		weak EDGAC*	
	Time(s)	Nodes	Time(s)	Nodes
3	†5.6	841.4	6.2	†803.2
4	†25.4	2568.8	27.6	†2424.0
5	†535.6	47091.2	546.8	†40244.0

(h) Nurse Rostering with `soft_regular`^{edit}()

n	FDGAC*		weak EDGAC*	
	Time(s)	Nodes	Time(s)	Nodes
30	†30.0	171.4	35.2	†162.6
35	†57.6	239.8	69.0	†233.4
40	†92.2	328.6	108.2	†316.0
45	†240.6	651.8	246.4	†570.6
50	130.2	1660.6	†118.2	†1316.0
55	208.0	2291.8	†193.8	†1856.8

(i) Modeling `stretch` by `soft_regular`^{var}()

n	FDGAC*		weak EDGAC*	
	Time(s)	Nodes	Time(s)	Nodes
30	†34.2	123.8	39.6	†122.4
35	†60.6	164.0	70.8	†162.8
40	†90.8	208.4	101.6	†194.0
45	239.6	371.0	†207.8	†299.6
50	204.8	967.6	†185.0	†823.2
55	264.2	972.8	†234.6	†777.6

(j) Modeling `stretch` by `soft_regular`^{edit}()

Table 1: Experimental results: time (in seconds) and number of nodes

Conclusion

Our contributions are three-fold. First, we discover and give an example of a limitation of EDAC*. When constraints share more than one variable, oscillation similar to the one demonstrated in Full AC* (de Givry et al. 2005) will occur. Second, we introduce cost-providing partitions, which restrict the distribution of the cost when enforcing EDAC*. Based on cost-providing partitions, we define weak EDGAC*, which can be enforced in polynomial time for projection-safe soft global constraints (Lee and Leung

2009). Third, we perform extensive experiments to compare weak EDGAC* and FDGAC*, and confirm the pruning and execution efficiency of our proposal.

One immediate future work is to investigate the effect of cost-providing partitions. It is unclear how different variable arrangement in the cost-providing partitions affect domain pruning as well as lower bound deduction. Another possible direction is to investigate if other even stronger consistencies, such as VAC (Cooper et al. 2008), can also benefit from projection safety to make their enforcement practical. Such work can help enrich the applicability of soft constraints to real-life problems.

Acknowledgement

We are grateful to the anonymous referees for their constructive comments. The work described in this paper was substantially supported by grants CUHK413207 and CUHK413808 from the Research Grants Council of Hong Kong SAR.

References

Beldiceanu, N.; Katriel, I.; and Thiel, S. 2004. Filtering Algorithms for the Same Constraints. In *Proceedings of CPAIOR'2004*, 65–79.

Cooper, M., and Schiex, T. 2004. Arc Consistency for Soft Constraints. *Artificial Intelligence* 154:199–227.

Cooper, M.; de Givry, S.; Sanchez, M.; Schiex, T.; and Zytnicki, M. 2008. Virtual Arc Consistency for Weighted CSP. In *proceedings of AAAI'2008*, 253–258.

Cooper, M. 2005. High-order Consistency in Valued Constraint Satisfaction. *Constraints* 10:283–305.

de Givry, S.; Heras, F.; Zytnicki, M.; and Larrosa, J. 2005. Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *Proceedings of IJCAI'2005*, 84–89.

Larrosa, J., and Schiex, T. 2003. In the Quest of the Best Form of Local Consistency for Weighted CSP. In *Proceedings of IJCAI'2003*, 239–244.

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

Lee, J., and Leung, K. 2009. Towards Efficient Consistency Enforcement for Global Constraints in Weighted Constraint Satisfaction. In *Proceedings of IJCAI'2009*, 559–565.

Pesant, G. 2001. A Filtering Algorithm for the Stretch Constraint. In *Proceedings of CP'2001*, 183–195.

Pesant, G. 2004. A Regular Language Membership Constraint for Finite Sequences of Variables. In *Proceedings of CP'2004*, 482–495.

Sanchez, M.; de Givry, S.; and Schiex, T. 2008. Mendelian Error Detection in Complex Pedigrees using Weighted Constraint Satisfaction Techniques. *Constraints* 13(1):130–154.

Schiex, T.; Fargier, H.; and Verfaillie, G. 1995. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *Proceedings of IJCAI'1995*, 631–637.

van Hoeve, W.; Pesant, G.; and Rousseau, L.-M. 2006. On Global Warming: Flow-based Soft Global Constraints. *J. Heuristics* 12(4-5):347–373.

Zytnicki, M.; Gaspin, C.; and Schiex, T. 2009. Bounds Arc Consistency for Weighted CSPs. *Journal of Artificial Inteliigence Research* 35:593–621.