Triangle-based consistencies for cost function networks

Christian Bessiere, Simon de Givry, Thomas Schiex, Thi Hông Hiêp Nguyên

Noname manuscript No.
(will be inserted by the editor)

Triangle-based Consistencies for Cost Function Networks

Hiep Nguyen · Christian Bessiere · Simon de Givry · Thomas Schiex

Received: date / Accepted: date

Abstract Cost Function Networks (aka Weighted CSP) allow to model a variety of problems, such as optimization of deterministic and stochastic graphical models including Markov random Fields and Bayesian Networks. Solving cost function networks is thus an important problem for deterministic and probabilistic reasoning. This paper focuses on local consistencies which define essential tools to simplify Cost Function Networks, and provide lower bounds on their optimal solution cost. To strengthen arc consistency bounds, we follow the idea of triangle-based domain consistencies for hard constraint networks (path inverse consistency, restricted or max-restricted path consistencies), describe their systematic extension to cost function networks, study their relative strengths, define enforcing algorithms, and experiment with them on a large set of benchmark problems. On some of these problems, our improved lower bounds seem necessary to solve them.

Keywords Cost function networks · Weighted CSP · Constraint optimization problems · High order consistencies · Restricted path consistency · Path inverse consistency · Max-restricted path consistency

Hiep Nguyen
MIAT, UR 875, Université de Toulouse, INRA, Castanet-Tolosan, France
E-mail: ifi.nthhiep@gmail.com

Christian Bessiere
University of Montpellier, Montpellier, France
E-mail: bessiere@lirmm.fr

Simon de Givry
MIAT, UR 875, Université de Toulouse, INRA, Castanet-Tolosan, France
E-mail: degivry@toulouse.inra.fr

Thomas Schiex
MIAT, UR 875, Université de Toulouse, INRA, Castanet-Tolosan, France
E-mail: tschiex@toulouse.inra.fr

1 Introduction

Graphical model processing is a central problem in AI. The Cost Function Network framework (CFN [25] as an instance of the valued CSP framework.), where the goal is to optimize the combined cost of local cost functions, captures problems such as weighted MaxSAT, Weighted CSP or Maximum Probability Explanation in probabilistic networks.[1] CFNs have applications in resource allocation [4], combinatorial auctions, bioinformatics [27,26]...

Dynamic programming approaches such as bucket or cluster tree elimination can be used to tackle such problems but are inherently limited by their exponential time and space behavior on graphical models with high tree-width. Instead, Depth First Branch and Bound allows to keep a polynomial space complexity but requires good (strong and cheap) lower bounds on the minimum cost to be efficient. In the last years, increasingly better lower bounds have been designed by enforcing soft local consistencies on CFNs. Arc consistencies such as AC*, DAC*, FDAC*, EDAC* [17] or VAC [6] are inspired from arc consistency in hard constraint networks. They have a small order polynomial enforcing time but do not always provide tight enough lower-bounds. The linear programming based OSAC consistency [9] provenly gives the strongest lower bound that can be obtained by arc consistency but is usually too expensive to compute. It now becomes useful to look beyond arc consistencies. Up to now, few higher order consistencies have been proposed for CFNs [8,12].

In this paper, we show that strong soft consistencies can be defined for CFNs by extending hard high order consistencies defined for CSPs. Among hard high order consistencies, the family of triangle-based consistencies (Restricted Path Consistency or RPC, Path Inverse Consistency or PIC, and maxRestricted Path Consistency or maxRPC) are specifically interesting because they have a stronger pruning power than arc consistency, and a cheaper computational cost than other high order consistencies. Their extension to CFNs is however non trivial, and enforcing algorithms create ternary cost functions.

The rest of the paper is organized as follows. Section 2 is a background section on constraint and cost function networks, associated local consistencies and enforcing operations. The next sections focus on our contributions. Section 3 gives a definition of new local consistencies. Section 4 introduces different ways to compare the strength of soft consistencies in general. This is then used to compare the proposed consistencies to each other and to existing consistencies. Section 5 focuses on the algorithms for enforcing the proposed consistencies. The last section gives experimental results when these consistencies are used as pre-processing or maintained during search on a large set of benchmarks. We observe that the strengthened bound provided by triangle consistencies (TRICs) are necessary to solve some problems that could not be solved otherwise.

[1] We use the terminology of Cost Function Networks by similarity to Constraint Networks. The Weighted Constraint Satisfaction Problem (WCSP) is the problem of solving a CFN. For outsiders, guessing what a Cost Function Network could be, is also much easier.

2 Background

A constraint satisfaction problem (CSP) is a triple (X, D, C) where X is a set of n variables, D is a set of n domains (variable $i \in X$ takes values from $D_i \in D$), and C is a set of constraints. Each constraint $c_S \in C$ defined over a set S of variables specifies the allowed assignments τ_S of values for variables in S, denoted by $\tau_S \in c_S$. S and $|S|$ are the scope and the arity of the constraint c_S. For simplicity, $c_{\{i\}}$, $c_{\{i,j\}}$ are denoted as c_i, c_{ij}. The constraints c_i, c_{ij}, c_S with $|S| > 2$ are respectively called unary, binary and non-binary. A value a for variable i is denoted by (i, a) or by i_a. Given a set of variables S, $\ell(S)$ denotes the set of assignments (tuples) of values for variables in S, that is, $\ell(S) = D^S = \prod_{i \in S} D_i$. Given a tuple τ_S, a variable $i \in S$ and a subset $S' \in S$, $\tau_S[i]$ and $\tau_S[S']$ denote the projection of tuple τ_S on i and S' respectively. A tuple τ_S is consistent if it satisfies all the constraints whose scope is included in S. A solution is a consistent complete assignment. The problem is consistent if it has at least one solution.

Definition 1 (Local consistencies) Given a CSP $P = (X, D, C)$,

- P is arc consistent (AC) if $\forall i \in X$, $\forall a \in D_i$, $\forall c_S \in C$ such that $i \in S$, there exists a tuple $\tau \in \ell(S)$ such that $\tau[i] = a$ and $\tau \in c_S$. Such a tuple τ is called the support of value (i, a) in the constraint c_S.
- P is restricted path consistent (RPC, [3]) iff it is AC and $\forall i \in X$, $\forall a \in D_i$, $\forall c_{ij} \in C$ on which a has only one support $b \in D_j$, $\forall k$ linked to i and j by c_{ik}, c_{jk}, there exists a value $c \in D_k$ such that $(a, c) \in c_{ik}$ and $(b, c) \in c_{jk}$.
- P is path inverse consistent (PIC, [14]) iff it is AC and $\forall i \in X$, $\forall a \in D_i$, $\forall j, k$ such that i, j, k are linked one-by-one by binary constraints, there exists a value $b \in D_j, c \in D_k$ such that $(a, b) \in c_{ij}, (a, c) \in c_{ik}$ and $(b, c) \in c_{jk}$.
- P is max-restricted path consistent (maxRPC, [11]) iff it is AC and $\forall i \in X$, $\forall a \in D_i$, $\forall c_{ij} \in C$, a has a support $b \in D_j$ such that $\forall k$ linked to both i, j by c_{ik}, c_{jk}, there exists a value $c \in D_k$ such that $(a, c) \in c_{ik}$ and $(b, c) \in c_{jk}$.
- Let Φ be a hard consistency and P a CSP. The Φ–closure of P is the CSP $\Phi(P) = (X, D_\Phi, C)$ such that $D^X_\Phi \subseteq D^X$, $\Phi(P)$ is Φ-consistent, and there does not exist D' such that $D^X_\Phi \subset D'^X$ and (X, D', C) is Φ-consistent.

CFNs extend CSPs by associating costs to tuples [25,24]. A CFN is a tuple (X, D, C, m) where X and D are respectively sets of variables and domains, as in classical CSPs. C is a set of cost functions. Each cost function $c_S \in C$ assigns non negative integer costs to tuples $\tau_S \in \ell(S)$ i.e. $c_S : \ell(S) \to [0..m]$ where $m \in \{1, \ldots, +\infty\}$. The addition and subtraction of costs are bounded operations, defined as $a +_m b = \min(a + b, m)$, $a -_m b = a - b$ if $a < m$ and m otherwise. The combined cost of a tuple τ_S in a CFN P is the sum of costs $Val_P(\tau_S) = \sum_{(S' \subseteq S) \wedge (c_{S'} \in C)} c_{S'}(\tau_S[S'])$, where summation is done using $+_m$. τ_S is inconsistent if $Val_P(\tau_S) = m$, and consistent otherwise. A solution of P is a complete consistent tuple τ_X. An optimal solution has minimum $Val_P(\tau_X)$.

It is important to note that CSPs are just CFNs using $m = 1$. In such problems, a tuple which receives a cost of 1 is forbidden. A cost of 0 is used for allowed tuples.

We assume the existence of a unary cost function c_i for every variable i, and a nullary cost function, noted $c_\varnothing$. This constant non-negative cost defines a lower bound on the cost of every solution. A CFN P can be transformed into an equivalent CFN P' (i.e., $Val_P(\tau) = Val_{P'}(\tau)\ \forall\tau$) by applying so-called equivalence-preserving transformations (EPTs) that shift costs between cost functions. The EPT `Shift`$(\tau_S, c_{S'}, \alpha)$ (Algorithm 1) moves an amount of cost α between a cost function $c_{S'}$ and a tuple τ_S such that $S \subset S'$. The conditions (2) and (3) guarantee that the operation will not create any negative cost in the problem. `Shift` allows to define the three usual EPTs [10] `Project` (from $c_{S'}$ to τ_S, $\alpha > 0$), `Extend` (from τ_S to $c_{S'}$, $\alpha < 0$) and `UnaryProject` (from i to $c_\varnothing$, $\alpha > 0$, $S' = \{i\}$, $S = \varnothing$).

Algorithm 1: Operation for shifting costs in CFNs

Procedure `Shift`$(\tau_S, c_{S'}, \alpha)$
  // condition:(1) $S \subset S', (2) c_S(\tau_S) +_m \alpha \geq 0, (3) c_{S'}(\tau'_{S'}) \geq \alpha : \forall \tau'_{S'} \in \ell(S'), \tau'_{S'}[S] = \tau_S$
  $c_S(\tau_S) \longleftarrow c_S(\tau_S) +_m \alpha$;
  foreach $\tau'_{S'} \in \ell(S')$ $s.t$ $\tau'_{S'}[S] = \tau_S$ **do**
    $c_{S'}(\tau'_{S'}) \longleftarrow c_{S'}(\tau'_{S'}) -_m \alpha$;

By applying EPTs to an original CFN, it is possible to transform it in an equivalent CFN that satisfies a given *local consistency property*. This may increase the lower bound $c_\varnothing$. The simplest local consistency, node consistency (NC [16]), requires that $\forall i \in X$, $\forall a \in D_i$ $c_i(a) + c_\varnothing < m$ and there exists a value $a \in D_i$ such that $c_i(a) = 0$.

Definition 2 (Soft arc consistencies) Given a binary CFN $P = (X, D, C, m)$ and an order $<$ on variables,

- P is arc consistent (AC [24]) iff $\forall i \in X$, $\forall a \in D_i$ and $\forall c_{ij} \in C$, there exists $b \in D_j$ such that $c_{ij}(a, b) = 0$. b is called a (simple) support for (i, a) in c_{ij}.[2]
- P is directional arc consistent (DAC [7]) w.r.t $<$ iff $\forall i$, $\forall a \in D_i$, $\forall c_{ij}$ such that $i < j$, there exists a value $b \in D_j$ such that $c_{ij}(a, b) + c_j(b) = 0$. b is called a full support for (i, a) in c_{ij}.
- P is full directional arc consistent (FDAC [19]) w.r.t $<$ iff it is AC and DAC.
- P is existential arc consistent (EAC [17]) iff $\forall i \in X$, there exists a value $a \in D_i$ such that $c_i(a) = 0$ and $\forall c_{ij} \in C$, there exists $b \in D_j$ such that $c_{ij}(a, b) + c_j(b) = 0$. Value a is called the existential arc consistent support of i.
- P is existential directional arc consistent (EDAC [17]) iff it is EAC and FDAC.

[2] There exists tiny variations on the definition of AC for CFNs. This paper uses the definition in [20] which simplifies the definition in [10] by not considering the propagation of inconsistent tuples.

– Bool(P) is a CSP defined as a CFN $(X, D, \overline{C}, 1)$ such that $\exists \overline{c}_S \in \overline{C}$ iff $\exists c_S \in C$, $S \neq \emptyset$ and $\tau \in \overline{c}_S \Leftrightarrow c_S(\tau) > 0$. P is virtual arc consistent (VAC [6]) iff the AC–closure of Bool(P) is non-empty.

For simplicity, we restrict ourselves to binary CFNs. A binary CFN is AC*, DAC*, FDAC*, EAC*, EDAC* if it is NC and respectively AC, DAC, FDAC, EAC, EDAC [16]. Definitions of soft arc consistencies for non-binary CFNs have been given in [10,5,21,22].

3 Soft triangle-based consistencies (TRICs)

In this section, we extend the hard local consistencies RPC, PIC and maxRPC, defined on triangles of variables to CFNs. For each hard consistency, we define six soft variants, also called *softening* levels: *simple*, *directional*, *full directional*, *existential*, *existential directional*, and *virtual*. This gives rise to eighteen new soft local consistencies. In addition to soft ACs, all these soft versions guarantee the extensibility of arc supports on extra third variables on a so-called *witness*.

Definition 3 (Witness) Given a value (i, a), a pair of values (i_a, j_b) and a variable k linked both to i and j,
– A simple witness of (i_a, j_b) on k is a value $c \in D_k$ such that $c_{ik}(a, c) + c_{jk}(b, c) + c_{ijk}(a, b, c) = 0$.
– A full witness of (i_a, j_b) on k is a value $c \in D_k$ such that $c_k(c) + c_{ik}(a, c) + c_{jk}(b, c) + c_{ijk}(a, b, c) = 0$.

Definition 4 (Extensibility of a pair of values on a variable) Given a pair of values (i_a, j_b) and a variable k linked both to i and j,
– (i_a, j_b) is simply extensible on k if there exists a simple witness on k for it.
– (i_a, j_b) is fully extensible on k if there exists a full witness on k for it.

Definition 5 (Extensibility of a value on a triangle) A triangle is a triple of variables (i, j, k) that are linked one-by-one by binary cost functions. It is noted as Δ_{ijk}. Given a value (i, a) and a triangle Δ_{ijk}.
– (i, a) is simply extensible on triangle Δ_{ijk} if there exists a simple arc support for (i, a) in c_{ij} that is simply extensible on k.
– (i, a) is fully extensible on triangle Δ_{ijk} if there exists a full arc support for (i, a) in c_{ij} that is fully extensible on k.

Definition 6 (Extensibility of a pair of values) For a pair of values (i_a, j_b) and an order $<$ on the variables, (i_a, j_b) is:
– simply extensible if it is simply extensible on every k linked to both i and j.
– fully extensible if it is fully extensible on every k linked to both i and j.
– directionally-fully extensible if it is fully extensible on every $k > i$ linked to both i and j.
– semi-fully extensible if it is simply extensible on every $k < i$ linked both to i and j and is fully extensible on every $k > i$ linked both to i and j.

Notice that full extensibility implies semi-full extensibility. Semi-full extensibility implies directional-full and simple extensibility. Conversely, both directional-full and simple extensibility do not imply any other extensibility. Examples in Figure 1 illustrate the different extensibilities of pairs of values.

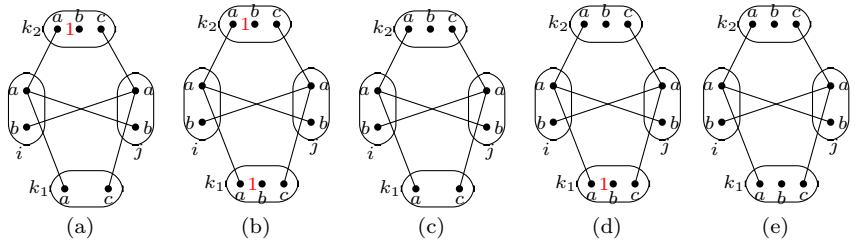


Fig. 1 Example of different extensibilities of the pair of values (i_a, j_a). $k_1 < i < j < k_2$. An edge appears between pairs of values with a non zero cost. In CFN(a), (i_a, j_a) is not simply extensible on k_1. In CFN(b), (i_a, j_a) is simply extensible (on both k_1, k_2) but is not directionally-fully extensible (because it is not fully extensible on k_2). In CFN(c), (i_a, j_a) is directionally-fully extensible w.r.t k_2 but is not semi-fully extensible (because it is not simply extensible on k_1). In CFN(d), (i_a, j_a) is semi-fully extensible (fully extensible on k_2 and simply extensible on k_1) but is not fully extensible (because it is not fully extensible on k_1). In CFN(e), (i_a, j_a) is fully extensible (on both k_1, k_2).

3.1 Soft restricted path consistencies

The idea of soft RPC consistencies is to only check the extensibility of pairs of values (i_a, j_b) that will make a value soft arc inconsistent if their binary cost becomes positive. If a value (i, a) has only one simple support (j, b) on c_{ij} and this support (i_a, j_b) is not extensible on some third variable k, every 3-values tuple over $\{i, j, k\}$, involving (i_a, j_b), has a positive combined cost. Because (j, b) is the unique arc support of (i, a), every complete tuple involving (i, a) has a positive cost evaluation. Thus, the unary cost $c_i(a)$ can be increased by equivalence preserving transformations.

Definition 7 (Soft restricted path consistencies (Soft RPCs)) Given a CFN $P = (X, C, D, m)$ and an order "<" on variables,

- P is RPC if it is AC and $\forall i \in X, \forall a \in D_i, \forall c_{ij} \in C$ on which (i, a) has only one simple arc support $b \in D_j$, (i_a, j_b) is simply extensible.
- P is directional RPC (DRPC) if it is DAC and $\forall i \in X, \forall a \in D_i, \forall c_{ij} \in C$ such that $i < j$ and (i, a) has only one full arc support $b \in D_j$, (i_a, j_b) is directionally-fully extensible.
- P is full directional RPC (FDRPC) if it is FDAC and $\forall i \in X, \forall a \in D_i, \forall c_{ij} \in C$ such that (1) if $i > j$ and (i, a) has only one simple arc support $b \in D_j$ then (i_a, j_b) is simply extensible, or (2) if $i < j$ and (i, a) has only one full arc support $b \in D_j$ then (i_a, j_b) is semi-fully extensible.

- P is existential RPC (ERPC) if $\forall i \in X$, there exists a value $a \in D_i$ such that (1) $c_i(a) = 0$, (2) i_a has a full arc support in every cost function (i.e., P is EAC), and (3) $\forall c_{ij} \in C$ on which (i,a) has only one full arc support $b \in D_j$, (i_a, j_b) is fully extensible. Such a value (i,a) is the ERPC support for i.
- P is existential directional RPC (EDRPC) if it is ERPC and FDRPC.
- P is virtual RPC (VRPC) if the RPC−closure of Bool(P) is non-empty.

VRPC is defined based on the hard CSP Bool(P) and hard RPC. The other softening levels of RPC differ from each other by (1) the strength of supports (simple or full) (2) the strength of witnesses (simple, full, directional-full, semi-full) and (3) the scope of application of these properties (every domain value or one value per domain, every cost function or some specific cost functions).

Example 1 Consider the CFNs in Figure 1.

- CFN(a) is VRPC because the RPC−closure of Bool(P) is not empty, containing values $(i_b), (j,b), (k_1,a), (k_1,c), (k_2,a), (k_2,c)$. However, it is not RPC because the unique support (i_a, j_a) of (i,a) on c_{ij} is not simply extensible on k_1
- CFN(b) is RPC: both (i_a, j_a) and (i_b, j_b) (respectively the unique simple arc support of $(i,a), (j,a)$ on c_{ij} and of $(i,b), (j,b)$ on c_{ij}) are simply extensible on k_1 and k_2 at simple witnesses (k_1,b) and (k_2,b) respectively. However, it is not DRPC because the unique full arc support (i_a, j_a) of (i,a) in c_{ij} is not fully extensible on $k_2 > i$.
- CFN(c) is DRPC because both (i_a, j_a) and (i_b, j_b) (respectively the unique full arc support of (i,a) in c_{ij} and of (i,b) in c_{ij}) are fully extensible on $k_2 > i$ at (k_2,b). Variable $k_1 < i$ is not involved in DRPC for i. However, it is not FDRPC because the unique full support (i_a, j_a) of value (i,a) on c_{ij} is not simply extensible on k_1.
- CFN(d) is FDRPC where the supports (i_a, j_a) and (i_b, j_b) are fully extensible on k_2 at (k_2,b) and simply extensible on k_1 at (k_1,b). At the same time, it is ERPC where (i,b), (j,b), (k_1,a), (k_2,a) are ERPC supports for variables i, j, k_1 and k_2.

3.2 Soft path inverse consistencies

We now consider soft path inverse consistencies. They guarantee the extensibility of domain values on triangles of variables. For all triangles Δ_{ijk} sharing two variables i, j of a cost function c_{ij}, PICs require that one of the arc supports of (i,a) in c_{ij} is extensible on k. The arc supports of (i,a) that are extensible on different k can be different.

Definition 8 (Soft path inverse consistencies (Soft PICs)) Given a CFN $P = (X, C, D, m)$ and an order $<$ on variables,

- P is PIC if it is AC and $\forall i \in X, \forall a \in D_i, \forall \Delta_{ijk}$, (i,a) is simply extensible on Δ_{ijk}.

- P is directional PIC (DPIC) if it is DAC and $\forall i \in X, \forall a \in D_i, \forall \Delta_{ijk}$ such that $i < j, i < k$, (i, a) is fully extensible on Δ_{ijk}.
- P is full directional PIC (FDPIC) if it is FDAC and $\forall i \in X, \forall a \in D_i, \forall \Delta_{ijk}$, (i, a) is fully extensible on Δ_{ijk} if $i < j, i < k$ and simply extensible on Δ_{ijk} otherwise.
- P is existential PIC (EPIC) if $\forall i \in X$, there exists a value $a \in D_i$ such that (1) $c_i(a) = 0$, (2) i_a has a full arc support in every cost function (i.e., P is EAC) and (3) (i, a) is fully extensible on every triangle.
- P is existential directional PIC (EDPIC) if it is EPIC and FDPIC.
- P is virtual PIC (VPIC) if the PIC−closure of Bool(P) is non-empty.

See examples in Figure 2. As in the case of RPC, VPIC is defined based on the hard CSP Bool(P) and hard PIC. The other softening levels differ from each other by the strength of supports, the strength of witnesses, and the scope of application of these properties.

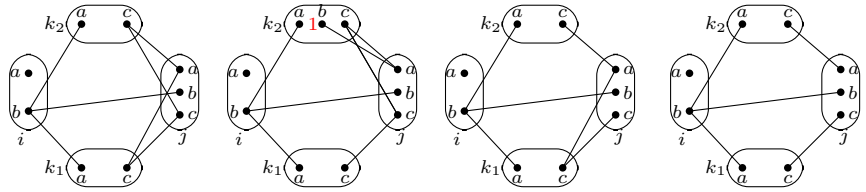


Fig. 2 Example of soft PIC consistencies. $k_1 < i < k_2 < j$ and $\exists \Delta_{ijk_1}, \Delta_{ijk_2}$. The CFN(a) is not PIC because value (i, b) is not simply extensible to triangle Δ_{ijk_1}. The CFN(b) is PIC but is not DPIC because value (i, b) is not fully extensible to triangle Δ_{ijk_2} with $i < j, i < k_2$. The CFN(c) is DPIC (because every value in D_i can be fully extended to Δ_{ijk_2} (the only triangle concerned by DPIC for i) but it is not FDPIC (because value (i, b) is not simply extensible to triangle Δ_{ijk_1}). The CFN(d) is FDPIC where every variable is simply extensible to 2 triangles and i is fully extensible to (i, j, k_2). The CFN(d) is also EPIC where $(i, a), (j, a), (k_1, a), (k_2, a)$ are respectively EPIC supports of i, j, k_1, k_2.

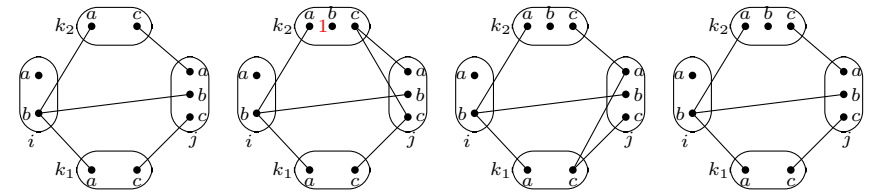
3.3 Soft max-restricted path consistencies

Stronger than PICs, soft max-restricted path consistencies (soft maxRPCs) check the existence of an extensible arc support for each value on each binary cost function whatever the number of arc supports the value has. In contrast to soft PICs, maxRPCs require the extensibility of the same arc support at the same time on all third variables. If value (i, a) has no such extensible arc support in some binary cost function c_{ij}, each support (i_a, j_b) of (i, a) in c_{ij} is not extensible in some extra variable k, i.e. the combined cost of every tuple (i_a, j_b, k_c) is positive. Thus, the binary cost of every arc support of (i, a) in c_{ij} can be increased by equivalence preserving transformations and then (i, a) will no longer be arc consistent.

Definition 9 (Soft max-restricted path consistencies (Soft maxRPCs)) Given a CFN $P = (X, D, C, m)$ and an order $<$ on the variables,

- P is maxRPC if it is AC and $\forall i \in X, \forall a \in D_i, \forall c_{ij} \in C$ there exists a simple arc support $b \in D_j$ such that (i_a, j_b) is simply extensible.
- P is directional maxRPC (DmaxRPC) if it is DAC and $\forall i \in X, \forall a \in D_i$, $\forall c_{ij} \in C$ such that $i < j$, there exists a full arc support $b \in D_j$ such that (i_a, j_b) is directionally-fully extensible.
- P is full directional maxRPC (FDmaxRPC) if it is FDAC and for $\forall i \in D, \forall a \in D_i, \forall c_{ij} \in C$ (1) if $i > j$, there exists a simple arc support $b \in D_j$ such that (i_a, j_b) is simply extensible. (2) otherwise, if $i < j$, there exists a full arc support $b \in D_j$ such that (i_a, j_b) is semi-fully extensible.
- P is existential maxRPC (EmaxRPC) if $\forall i \in X$, there exists a value $a \in D_i$ such that (1) $c_i(a) = 0$, (2) i_a has a full arc support in every cost function (i.e., P is EAC) and (3) $\forall c_{ij} \in C$, there exists a full arc support $b \in D_j$ such that (i_a, j_b) is fully extensible.
- P is existential directional maxRPC (EDmaxRPC) if it is EmaxRPC and FDmaxRPC.
- P is virtual maxRPC (VmaxRPC) if the maxRPC–closure of Bool(P) is non-empty

See examples in Figure 3. Here again, VmaxRPC is defined based on the hard CSP Bool(P) and hard maxRPC. The other softening levels differ from each other by the strength of supports, the strength of witnesses, and the scope of application of these properties.



(a) EmaxRPC not maxRPC (b) maxRPC not DmaxRPC (c) DmaxRPC not FDmaxRPC (d) FDmaxRPC EDmaxRPC

Fig. 3 Example of soft maxRPCs. $k_1 < i < k_2 < j$ and $\exists\ \Delta_{ijk_1}, \Delta_{ijk_2}$. The CFN(a) is not maxRPC because value (i, b) has no arc support in c_{ij} (between (i_b, j_a) and (i_b, j_c)) that is simply extensible on both k_1, k_2. The CFN(b) is maxRPC but is not DmaxRPC because value (i, b) has no full arc support in c_{ij} (between (i_b, j_a) and (i_b, j_c)) that is fully extensible to k_2. The CFN(c) is DmaxRPC (because every value in D_i has full arc support in c_{ij}, c_{ik_2} that is respectively fully extensible on k_2 and j. Triangle Δ_{ijk_1} is not involved in DmaxRPC for i). The CFN(c) is not FDmaxRPC because value (i, b) has no full support in c_{ij} (between (i_b, j_a) and (i_b, j_c)) that is simply extensible on k_1. The CFN(d) is both FDmaxRPC and EmaxRPC where $(i, a), (k_1, a), (j, a), (k_2, a)$ are respectively EmaxRPC supports of variables i, k_1, j, k_2.

4 Comparing soft local consistencies

In this section, we compare the strength of the different soft consistencies proposed in the previous section and soft arc consistencies.

Soft consistencies rise specific difficulties when comparison of strength is considered. Most of the consistencies we considered are domain consistencies in the sense that they define properties that values must satisfy and enforcing them may increase unary costs that NC may ultimately use to increase the lower bound $c_\varnothing$. However, virtual local consistencies are different because they directly try to increase $c_\varnothing$ and do not try to increase unary costs for NC. Thus, the strength of virtual consistencies can be directly measured by the quality of the lower bound provided. For the other soft consistencies, this strength is better measured by the ability to move costs to lower arities. We therefore need to introduce two different order relations between local consistencies to capture this difference between virtual and other consistencies. We denote by $c_\varnothing[P]$ the lower bound $c_\varnothing$ in a problem P.

Furthermore, soft local consistencies are not confluent. A single problem P may have different equivalent problems satisfying a given local consistency property A. For a given CFN P and a soft local consistency A, $A(P)$ is therefore defined as the set of problems that can be obtained after enforcing A in P. When P already satisfies A, we assume that $A(P) = \{P\}$ i.e., that enforcing A on a problem satisfying A does not change P (which is effectively the case for all enforcing algorithms). Similarly, focusing on lower bounds, enforcing a weaker consistency will not change the lower bound.

Definition 10 (Stronger relation) Given two soft consistencies A and B,
- A is stronger than B, noted by $A \geq B$, iff for every CFN P that satisfies A, P also satisfies B, i.e. $B(P) = \{P\}$.
- A is stronger than B in terms of lower bound, noted by $A \geq_{c_\varnothing} B$, iff for every CFN P that satisfies A and any $P' \in B(P)$, then $c_\varnothing[P'] = c_\varnothing[P]$.
- A is strictly stronger than B, noted $A > B$, iff $A \geq B$ and $\exists$ a CFN P such that P satisfies B and $A(P) \neq \{P\}$.
- A is strictly stronger than B in terms of lower bound, noted $A >_{c_\varnothing} B$, iff $A \geq_{c_\varnothing} B$ and $\exists$ a CFN P such that P satisfies B and $\forall P' \in A(P), c_\varnothing[P'] > c_\varnothing[P]$.

We first show that $\geq$ entails $\geq_{c_\varnothing}$.

Proposition 1 *Given two soft consistencies A and B, if $A \geq B$ then $A \geq_{c_\varnothing} B$.*

Proof Because $A \geq B$, $B(P) = \{P\}$ for every P that satisfies A. So we have $\forall P' \in B(P), c_\varnothing[P'] \geq c_\varnothing[P]$ and thus $A \geq_{c_\varnothing} B$. □

Similarly to the stronger and strictly stronger relations for hard consistencies, our relations for soft consistencies are transitive.

Proposition 2 (Transitivity) *Given three soft consistencies A, B, and C,*
a. If $A \geq B$ and $B \geq C$ then $A \geq C$.

b. If $A > B$ *and* $B > C$ *then* $A > C$.
c. If $A > B$ *and* $B \geq_{c_\varnothing} C$ *then* $A \geq_{c_\varnothing} C$.

Proof a. Let P be a CFN that satisfies A. Because $A \geq B$ and P satisfies A, $B(P) = \{P\}$, i.e. P also satisfies B. Because $B \geq C$ and P satisfies B, $C(P) = \{P\}$. Thus, if P satisfies A, $C(P) = \{P\}$, i.e. $A \geq C$.
b. (1) Because $>$ implies $\geq$, we have $A \geq B$ and $B \geq C$. So $A \geq C$ from the property (a). (2) Because $A > B$, there exists a CFN P satisfying B and $A(P) \neq \{P\}$. Because P satisfies B and $B \geq C$, P also satisfies C. Thus there exists P that satisfies C and $A(P) \neq \{P\}$. So $A > C$.
c. Because $>$ implies $\geq$, we have $A \geq B$. Let P be a CFN that satisfies A, P also satisfies B. Because $B \geq_{c_\varnothing} C$ and P satisfies B, $\forall P' \in C(P), c_\varnothing[P'] = c_\varnothing[P]$. Thus, for every CFN which satisfies A, $\forall P' \in C(P), c_\varnothing[P'] = c_\varnothing[P]$. i.e. $A \geq_{c_\varnothing} C$. □

To show that a soft consistency A is not stronger (resp. not stronger in terms of lower bound than B), it is enough to show that there exists a CFN P in which A holds and B does better than A: $B(P) \neq \{P\}$ (resp. $\exists P' \in B(P), c_\varnothing[P'] \neq c_\varnothing[P]$).

Two consistencies A and B are incomparable iff A is not stronger than B and B is not stronger than A.

Definition 11 (Incomparable relation) Given two soft consistencies A and B,
- A and B are incomparable, noted $A \not> B$, iff $A \not\geq B$ and $B \not\geq A$
- A and B are incomparable in terms of lower bound, noted $A \not>_{c_\varnothing} B$, if $A \not\geq_{c_\varnothing} B$ and $A \not\geq_{c_\varnothing} B$

Figure 4 is the Hasse diagram that summarizes the relations among soft ACs, RPCs, PICs and maxRPCs. A row of the graph corresponds to six soft consistencies associated with a same hard consistency and a column corresponds to the soft consistencies at a same softening level. A directed path from a consistency A to B, without or with dashed arrow, respectively means that $A > B$ or $A >_{c_\varnothing} B$. If there does not exist any directed path between A and B, they are incomparable. First, we consider the relation between virtual consistencies and domain consistencies. Then, domain consistencies are considered according to the rows and the columns of the graph.

Theorem 1 *Given two hard local consistencies* $\overline{A}$, $\overline{B} \in \{AC, RPC, PIC, maxRPC\}$, *if we denote by* VA, VB *their corresponding virtual consistencies and* A, B *any other softening level of* $\overline{A}$ *and* $\overline{B}$,
a. $VA >_{c_\varnothing} A$.
b. If $\overline{A} > \overline{B}$ *then*
b1 $VA > VB$
b2 $VA >_{c_\varnothing} B$.

Proof a. We first prove that VA $\geq_{c_\varnothing} A$ by contradiction. Suppose that there exists a CFN P satisfying VA and enforcing A can still increase $c_\varnothing$ from

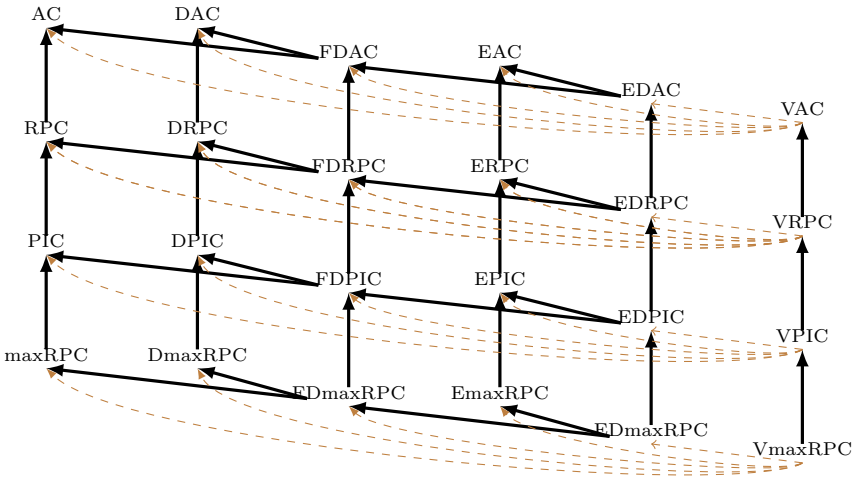


Fig. 4 Hasse diagram of relations between soft consistencies
$A \longrightarrow B : A > B$ $\quad A \longrightarrow B \longrightarrow C$ implies $A \longrightarrow C$
$A \dashrightarrow B : A >_{c_\varnothing} B$ $\quad A \longrightarrow B \dashrightarrow C$ implies $A \dashrightarrow C$

a variable $x_\varnothing$. All values and tuples whose costs have been necessary for increasing $c_\varnothing$ by A are also forbidden when enforcing $\overline{A}$ in the classic CSP Bool(P). So, if we eliminate these values and tuples in the same order that costs are moved by A in P, $x_\varnothing$ will be wiped-out in Bool(P). Thus P is not VA and the assumption is false. This means that VA $\geq_{c_\varnothing} A$. Secondly, Figure 12 shows a problem that satisfies every non-virtual variant of AC, RPC, PIC, maxRPC but not the virtual ones. Enforcing the virtual one will lead to a strictly stronger $c_\varnothing$.

b. Consider first the case of VB. First, we prove that VA $\geq VB$. Let P be a CFN which is VA. The $\overline{A}$–closure of $Bool(P)$ is not empty. Because $\overline{A} \geq \overline{B}$, the $\overline{B}$–closure of $Bool(P)$ will be not empty. Thus, P also satisfies VB, i.e. VB(P) = $\{P\}$. Now we prove that VA $\neq VB$, i.e. VmaxRPC > VPIC > VRPC > VAC. Figures 5, 6, 7 respectively show a CFN which is VAC but not VRPC, VRPC but not VPIC, VPIC but not VmaxRPC and $c_\varnothing$ can be increased by the unsatisfied consistencies.
We now consider the case of any other soft consistency B associated with $\overline{B}$. From VA > VB (just above) and from $VB >_{c_\varnothing} B$ (Theorem 1(a)), by Proposition 2(c), we deduce that VA $\geq_{c_\varnothing} B$. Now, we will prove that VA $>_{c_\varnothing} B$. Because $VB >_{c_\varnothing} B$, there exists a CFN P such that P is B and VB can still increase the lower bound $\underline{c_\varnothing}[P]$. This means that the $\overline{B}$–closure of $Bool(P)$ is empty. Because $\overline{A} > \overline{B}$, the $\overline{A}$–closure of $Bool(P)$ is also empty. Thus, enforcing VA on P will increase $c_\varnothing$ while P satisfies B. □

The following theorem shows that given a softening level, the corresponding soft maxRPC is strictly stronger than the corresponding soft PIC, which

is strictly stronger than the corresponding soft RPC, which itself is strictly stronger than the corresponding soft AC.

Theorem 2 (Vertical comparison)
a. $maxRPC > PIC > RPC > AC$.
b. $DmaxRPC > DPIC > DRPC > DAC$.
c. $FDmaxRPC > FDPIC > FDRPC > FDAC$.
d. $EmaxRPC > EPIC > ERPC > EAC$.
e. $EDmaxRPC > EDPIC > EDRPC > EDAC$.

Proof First, we can note that the "stronger than" relation $\geq$ holds between the considered pairs of consistencies, based on their definition: at each softening level, the soft variant of maxRPC implies the soft variant of PIC. The same applies for PIC and RPC, as well as RPC and AC. Second, we prove the "strictly stronger than" relation between them by showing CFNs in which the weaker consistencies hold while the stronger ones do not.

a. Figure 5 shows a CFN that satisfies AC but does not satisfy RPC. Figure 6 shows a CFN that satisfies RPC but does not satisfy PIC. Figure 7 shows a CFN that satisfies PIC but does not satisfy maxRPC. Thus maxRPC > PIC > RPC > AC.

b-e. The proof is similar to that for (a) by using Figure 5, 6 and 7. □

The following theorem will show that for any hard consistency: (1) the associated existential directional consistency is strictly stronger than both the existential and the full directional ones, (2) the associated full directional consistency is strictly stronger than both the non-directional and the directional ones, (3) other pairs of soft consistencies are incomparable.

Theorem 3 (Horizontal comparison) *Given two different hard consistencies* $\overline{A}$ *and* $\overline{B}$ *in* $\{AC, RPC, PIC, maxRPC\}$*, given* A, DA, FDA, EA, EDA *the simple, directional, full directional, existential, existential directional variant of* $\overline{A}$ *and* B, DB, FDB *the simple, directional, full directional variant of* $\overline{B}$*,*

a. (column 2-1): $A \not> DB$
b. (column 3-1,2): $FDA > A, DA$
c. (column 4-1,2,3): $EA \not> B, DB, FDB$
d. (column 5-3,4): $EDA > FDA, EA$

Proof a. $A \not> DB$: using Figures 8 and 9.

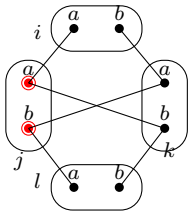
b. $FDA > A, DA$. The "stronger than" relation $\geq$ is implied by the definition of the consistencies. $FDA > A$: Figure 8 shows a problem which is maxRPC, PIC, RPC, AC but is not FDmaxRPC, FDPIC, FDRPC, FDAC. $FDA > DA$: Figure 9 shows a problem which is DmaxRPC, DPIC, DRPC, DAC but is not FDmaxRPC, FDPIC, FDRPC, FDAC.

c. $EA \not> B, DB, FDB$: using Figures 10 and 11.

d. $EDA > FDA, EA$. The proof directly follows from the definitions. □

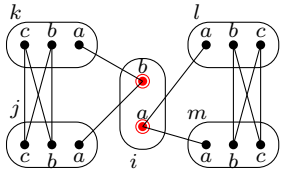
Theorem 4 (Incomparability) *For any pair of consistencies which is not covered by the three previous theorems, the consistencies are incomparable.*

Proof – FDAC $\not\succ$ RPC,PIC,maxRPC, DRPC,DPIC,DmaxRPC: using Figures 5, 8 and 9.
– FDRPC $\not\succ$ PIC,maxRPC, DPIC,DmaxRPC: using Figures 6, 8 and 9.
– FDPIC $\not\succ$ maxRPC, DmaxRPC: using Figures 7, 8 and 9.
– EDAC $\not\succ$ (E/FD/D/-)(RPC/PIC/maxRPC): using Figures 5, 10 and 11.
– EDRPC $\not\succ$ EPIC, EmaxRPC, FDPIC, FDmaxRPC, DPIC, DmaxRPC, PIC, maxRPC: using Figures 6, 10 and 11.
– EDPIC $\not\succ$ EmaxRPC, FDmaxRPC, DmaxRPC, maxRPC: using Figures 7, 10 and 11.
– VAC $\not\succ_{c_\emptyset}$ (ED/E/FD/D/-)(RPC/PIC/maxRPC): using Figures 5 and 12.
– VRPC $\not\succ_{c_\emptyset}$ (ED/E/FD/D/-)(PIC/maxRPC): using Figures 6 and 12.
– VPIC $\not\succ_{c_\emptyset}$ (ED/E/FD/D/-)maxRPC: using Figures 7 and 12. □



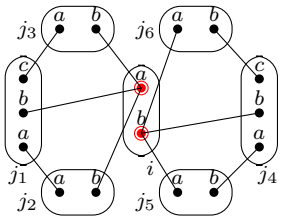
AC	DAC	FDAC	EAC	EDAC	VAC
~~RPC~~	~~DRPC~~	~~FDRPC~~	~~ERPC~~	~~EDRPC~~	~~VRPC~~
~~PIC~~	~~DPIC~~	~~FDPIC~~	~~EPIC~~	~~EDPIC~~	~~VPIC~~
~~max~~	~~Dmax~~	~~FDmax~~	~~Emax~~	~~EDmax~~	~~Vmax~~

Fig. 5 A CFN that satisfies all arc consistencies but does not satisfy any soft RPC (hence does not satisfy any soft PIC, maxRPC). $j < k < i < l$. The table on the right indicates which consistencies are satisfied or not (strikethrough). maxRPC is briefly written as max, and the same for other variants of maxRPCs. The problem does not satisfy any soft RPC because of variable j (the unique support (j_a, k_a) of (j, a) in c_{jk} is not simply extensible on i and the unique support (j_b, k_b) of (j, b) is not simply extensible on l).



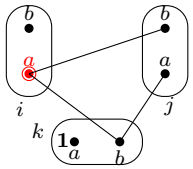
RPC	DRPC	FDRPC	ERPC	EDRPC	VRPC
~~PIC~~	~~DPIC~~	~~FDPIC~~	~~EPIC~~	~~EDPIC~~	~~VPIC~~
~~max~~	~~Dmax~~	~~FDmax~~	~~Emax~~	~~EDmax~~	~~Vmax~~

Fig. 6 A CFN that satisfies all RPC consistencies but does not satisfy any PIC consistency. $i < j < k < l < m$. Every value of i satisfies RPC consistencies because it has more than 2 full (hence simple) arc supports in $c_{ik}, c_{ij}, c_{il}, c_{im}$. The problem does not satisfy any PIC consistency because of variable i (value (i, a) is not simply (hence not fully) extensible to triangle Δ_{ilm} while (i, b) is not simply (hence not fully) extensible to triangle Δ_{ijk}).



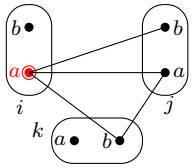
RPC DRPC FDRPC ERPC EDRPC VRPC
PIC DPIC FDPIC EPIC EDPIC VPIC
~~max~~ ~~Dmax~~ ~~FDmax~~ ~~Emax~~ ~~EDmax~~ ~~Vmax~~

Fig. 7 A CFN that satisfies all PIC consistencies but does not satisfy any maxRPC consistency. $i < j_1 < j_2 < j_3 < j_4 < j_5 < j_6$. There are only zero unary costs in this problem, thus simple and full supports (or witnesses) are identical. The problem is EDPIC since both $(i,a),(i,b)$ can be fully extended to all 4 triangles. However, the problem does not satisfy any maxRPC consistency because of variable i (no arc support of value (i,a) in c_{ij_1} can simultaneously be extended on $\Delta_{ij_1j_2}$ and $\Delta_{ij_1j_3}$, the same for value (i,b) in c_{ij_4}).



AC ~~DAC~~ ~~FDAC~~ ~~EDAC~~
RPC ~~DRPC~~ ~~FDRPC~~ ~~EDRPC~~
PIC ~~DPIC~~ ~~FDPIC~~ ~~EDPIC~~
maxRPC ~~DmaxRPC~~ ~~FDmaxRPC~~ ~~EDmaxRPC~~

Fig. 8 A CFN which is non-directional consistent but is directional inconsistent for order $i < j < k$. The problem is not DAC because value (i,a) has no full arc support in c_{ik}. Therefore, it does not satisfy FDAC, EDAC, FDRPC, EDRPC, FDPIC, EDPIC, FDmaxRPC, EDmaxRPC. However, the problem is maxRPC (hence PIC, RPC) because it is AC and every domain value is simply extensible to the triangle.



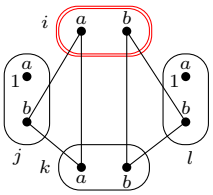
~~AC~~ DAC ~~FDAC~~ ~~EDAC~~
~~RPC~~ DRPC ~~FDRPC~~ ~~EDRPC~~
~~PIC~~ DPIC ~~FDPIC~~ ~~EDPIC~~
~~maxRPC~~ DmaxRPC ~~FDmaxRPC~~ ~~EDmaxRPC~~

Fig. 9 A CFN which is directional consistent $(i > j > k)$ but is non-directional inconsistent. The problem is not AC because (i,a) has no arc support in c_{ij}. However, the problem is DAC because every value of j and k has full arc support in c_{ji}, c_{ki}. Moreover, the problem is DmaxRPC (hence DPIC, DRPC) because every value of j and k can be fully extended on the triangle (in the triangle Δ_{ijk}, only the smallest variable k and c_{ki}, c_{kj} are concerned by triangle-based directional consistencies).

5 Algorithms

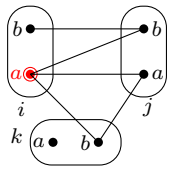
In this section, we present algorithms for enforcing soft PIC, DPIC, FDPIC, EPIC, EDPIC, maxRPC, DmaxRPC, FDmaxRPC, EmaxRPC, and EDmaxRPC. Soft RPCs have not been implemented because they are weaker than their PIC and maxRPC counterparts and because it is costly to maintain the uniqueness of arc supports per value in each cost function –arc supports can be iteratively created and broken when EPTs are applied.

For a value (i,a) that does not satisfy a given TRIC (triangle consistency), the common idea is to create a support for a value (i,a) on c_{ij} that is



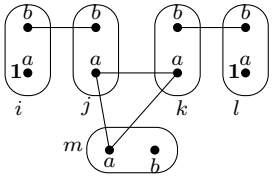
AC	DAC	FDAC	~~EAC~~	~~EDAC~~
RPC	DRPC	FDRPC	~~ERPC~~	~~EDRPC~~
PIC	DPIC	FDPIC	~~EPIC~~	~~EDPIC~~
maxRPC	DmaxRPC	FDmaxRPC	~~EmaxRPC~~	~~EDmaxRPC~~

Fig. 10 A CFN which is full directional consistent but is existential inconsistent for $l < j < k < i$. The problem is not EAC (hence not ERPC, EPIC, EmaxRPC) because of value i (i_a has no full support in c_{ij} while i_b has no full support in c_{il}). However, the problem is FDmaxRPC (hence FDPIC, FDRPC) because it is FDAC and every value of i, k can be simply extended to both triangles and every value of j, l can be fully extended to Δ_{jik} and Δ_{lik} respectively.



~~AC~~	~~DAC~~	~~FDAC~~	EAC	~~EDAC~~
~~RPC~~	~~DRPC~~	~~FDRPC~~	ERPC	~~EDRPC~~
~~PIC~~	~~DPIC~~	~~FDPIC~~	EPIC	~~EDPIC~~
~~maxRPC~~	~~DmaxRPC~~	~~FDmaxRPC~~	EmaxRPC	~~EDmaxRPC~~

Fig. 11 A CFN which is existential consistent but not full directional consistent for $i > j > k$. The problem is not AC (hence is not RPC, PIC, maxRPC) because of value (i, a) (no arc support in c_{ij}) and is not DAC (hence is not DRPC, DPIC, DmaxRPC) because of value (j, b) (no full arc support in c_{ij}). However, the problem is EmaxRPC (hence EPIC, ERPC, EAC) where $(i, b), (j, a), (k, a)$ are respectively EmaxRPC supports of i, j, k.



AC	DAC	FDAC	EAC	EDAC	~~VAC~~
RPC	DRPC	FDRPC	ERPC	EDRPC	~~VRPC~~
PIC	DPIC	FDPIC	EPIC	EDPIC	~~VPIC~~
max	Dmax	FDmax	Emax	EDmax	~~Vmax~~

Fig. 12 A CFN which is existential directional but is not virtual consistent $l < i < j < k < m$. The problem is not VAC (hence not VRPC, VPIC, VmaxRPC) because AC makes Bool(P) wiped-out at j or k. Conversely, the problem is EDmaxRPC where variables j, m, l are FDmaxRPC in Δ_{ijm} and i_b, j_a, k_a, l_b, m_b are EmaxRPC supports of variables.

also extensible on variables k by shifting costs in triangles Δ_{ijk} (consisting of binary, ternary and possibly unary costs) to c_i. We denote by $\Delta_{ijk}(a, b, c) = c_{ij}(a, b) + c_{jk}(b, c) + c_{ik}(a, c) + c_{ijk}(a, b, c)$ the combined cost defined by the sum of binary and ternary costs involved in tuple (i_a, j_b, k_c), where $c_{ijk}(a, b, c) = 0$ if c_{ijk} does not exist. Algorithm 2 presents all the basic operations for shifting costs inside triangles and pruning values.

- `Extend2To3`$(i, a, j, b, c_{ijk}, \alpha)$ extends a cost of α from a pair of values (i_a, j_b) to a ternary cost function c_{ijk}.
- `Project3To2`$(c_{ijk}, i, a, j, b, \alpha)$ projects a cost of α from c_{ijk} on (i_a, j_b).
- `Project3To1`(c_{ijk}, i, a, α) projects a cost of α from c_{ijk} on a value (i, a).
- `Extend1To2`(i, a, c_{ij}, α) extends a cost of α from a value (i, a) to c_{ij}.
- `Project2To1`(c_{ij}, i, a, α) projects a cost of α from c_{ij} on a value (i, a)

– `PruneVars()` removes all inconsistent values having unary cost equal to m.

The queues Q, P, S, T store variables or cost functions which had some change in domain or in cost. They will be used for the propagation of changes in our algorithm.

– Q stores variables i such that some value of D_i has been deleted (Procedure `PruneVars()`, line 24).
– P stores variables i such that some value of D_i has increased its cost from 0 (Procedure `Project3To1` at line 13 and `Project2To1` at line 17
– S is an auxiliary queue with the same contents as P (Procedure `Project3To1` at line 13 and Procedure `Project2To1` at line 17). It is used to efficiently build the propagation queue R which contains variables that need to be checked for the existential consistency. These are all variables of S (those that have values which cost increased from 0) and their neighbors because: (1) for $i \in S$, the value in D_i that has increased its unary cost may be the existential support of i and (2) the existential support of neighbor variables j may be fully supported by this value.
– T contains binary cost functions c_{ij} that have been modified (because of a unary cost extension in Procedure `Extend1To2`, line 4) for which i, j and their common neighbors may have lost simple support/witness and need to be revised.

5.1 Enforcing PICs

5.1.1 Enforcing PIC supports

Simple PIC supports are enforced by Procedure `findPICSupport` in Algorithm 3. To create a simple PIC support for a value i_a on Δ_{ijk}, binary and ternary costs involved in Δ_{ijk} are moved to i_a in such a way that there is a tuple (i_a, j_b, k_c) whose ternary and binary costs decrease to 0. The order for moving costs is presented in Figure 13. First, binary costs c_{ij}, c_{ik}, c_{jk} are extended on ternary cost function c_{ijk} by Procedure `Extend2To3` (lines 10–12). Then, ternary costs c_{ijk} are projected on i_a by Procedure `Project3To1` (line 13). The maximum cost that can be projected on each value $a \in D_i$ is stored in $P_i[a]$. It is computed based on the available binary and ternary costs (line 3). Binary cost extensions E_{ij}, E_{ik}, E_{jk} are then computed based on $P_i[a]$ and the ternary and binary costs (on the two other sides of the triangle, see lines 4–9). The extensions should be sufficiently large so that later projections of $P_i[a]$ will not create negative costs and sufficiently small so that a zero triangle cost remains after projection: there should exist values k_c, j_b and i_a such that the final resulting ternary cost $c_{ijk}(a,b,c)+E_{ij}(a,b)+E_{ik}(a,c)+E_{jk}(b,c)-P_i[a] = 0$. Each computed pairwise extended cost $E_{\cdot\cdot}(\cdot,\cdot)$ is sufficient to satisfy the maximum cost requirements on the third variable. Since these extensions are supposed to be done sequentially, line 7 subtracts $E_{ij}(a,b)$, which will be included in the ternary cost, and does not require $c_{ij}(a,b)$. The same reasoning applies for line 9, for both previous extensions.

Algorithm 2: Elementary operations

```
1  Procedure Extend1To2(i, a, c_ij, α)
2      // precondition: c_i(a) ≥ α > 0
3      Shift((i, a), c_ij, −α);
4      T ← T ∪ {c_ij};
5  Procedure Extend2To3(i, a, j, b, c_ijk, α)
       // precondition: c_ij(a, b) ≥ α > 0
6      Shift((i_a, j_b), c_ijk, −α);
7  Procedure Project3To2(c_ijk, i, a, j, b, α)
8      // precondition: ∀c ∈ D_k, c_ijk(a, b, c) ≥ α
9      Shift((i_a, j_b), c_ijk, α);
10 Procedure Project3To1(c_ijk, i, a, α)
11     // precondition: ∀b ∈ D_j, c ∈ D_k, c_ijk(a, b, c) ≥ α
12     if c_i(a) = 0 ∧ α > 0 then
13         P ← P ∪ {i}; S ← S ∪ {i};
14     Shift(i_a, c_ijk);
15 Procedure Project2To1(c_ij, i, a, α)
       // precondition: ∀b ∈ D_j, c_ij(a, b) ≥ α
16     if c_i(a) = 0 ∧ α > 0 then
17         P ← P ∪ {i}; S ← S ∪ {i};
18     Shift(i_a, c_ij);
19 Procedure PruneVars()
20     foreach i ∈ X do
21         foreach a ∈ D_i do
22             if c_i(a) + c_∅ ≥ m then
23                 D_i ← D_i − {a};
24                 Q ← Q ∪ {i};
25 Procedure isSmallest(i, Δ_ijk)
26     return ((i < j) ∧ (i < k));
```

In the end, binary cost extensions on ternary functions do not lead to the loss of ternary AC supports. Moreover, binary cost extensions do not lead to the loss of PIC supports because PIC supports involve only zero binary costs which cannot be used for extension.

Full PIC supports are similarly enforced by Procedure `findFullPICSupport` in Algorithm 3. The difference is that unary costs on j, k are extended on binary functions c_{ij} and c_{ik} by Procedure `Extend1To2`, in order to create full PIC supports with zero unary costs (lines 23, 24 respectively). Then binary and ternary costs are moved to i_a as for simple PIC supports (line 25). The order in which costs are moved to enforce full PIC supports is also visible in Figure 13. The unary costs of j, k are taken into account for the computation of $P_i[a]$ as well as for the computation of unary cost extensions E_j, E_k (lines 18,20,22). As for binary extensions, unary cost extensions should be sufficiently large to avoid the creation of negative costs by later projections of $P_i[a]$ and

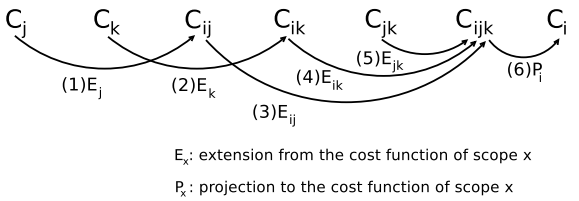


Fig. 13 The order of cost movements for enforcing simple or full PIC supports on variable i, where unary cost extensions are not included in the enforcement of simple PIC supports. The arcs indicate the direction of cost movements and the numbers under the arcs indicate the order in which the corresponding cost movements are performed.

small enough so that the the final binary costs $c_{ij}(a,b) + E_j(b) - E_{ij}(a,b)$ and $c_{ik}(a,c) + E_k(c) - E_{ik}(a,c)$ are equal to 0.

Therefore, unary cost extensions on binary functions cannot lead to the loss of binary AC supports. However, unary cost extensions on binary functions can lead to the loss of simple PIC supports, thus modified binary functions are stored in the list T in order to later enforce PIC supports for related values.

Example 2 Consider the CFN(a) in Figure 14. It has 4 variables $i < j < k < l$ and 5 binary cost functions $c_{ij}, c_{ik}, c_{il}, c_{jk}, c_{jl}$. Binary costs are represented by edges (continuous line) and ternary costs are represented by hyper-edges (dashed lines for c_{ijk} and dotted lines for c_{ijl}). The absence of (hyper)edges indicates a zero cost. The initial problem is FDAC but not FDPIC because value (i,a) is not fully extensible on Δ_{ijk}. Now, consider enforcing full PIC supports for the values of i. Procedure `findFullPICSupport(`i,j,k`)` computes the amounts of cost for projections/extensions: $P_i[a] = E_j[b] = 1$. Other shifted costs are zero. After extending a cost of 1 from (j,b) on c_{ij}, it will call Procedure `findPICSupport(`i,j,k`)`, compute the amounts of shifted cost as follows:

$$P_i[a] = E_{ij}[a,b] = E_{ik}[a,a] = E_{jk}[a,b] = 1.$$

and perform the cost shifts. The resulting problem, presented in Sub-figure 14(d) is still not FDPIC because value (i,b) cannot be fully extended on triangle Δ_{ijl}. Then Procedure `findFullPICSupport(`i,j,l`)` computes and performs the following cost shifting:

$$P_i[b] = E_{ij}[b,b] = E_{il}[b,a] = E_{jl}[a,b] = 1.$$

The final problem, presented in Sub-figure 14(g) is FDPIC. Contrarily to hard PIC, enforcing simple and full PIC supports can create new ternary functions, e.g., c_{ijk}, c_{ijl}. Whenever a binary cost need to be extended to a ternary cost function that does not exist, the ternary cost function needs to be created and initialized with an empty cost for every tuple.

5.1.2 Soft PIC algorithms

Enforcing EDPIC requires enforcing PIC, DPIC, and EPIC simultaneously. We thus only present an algorithm for EDPIC. PIC, DPIC, FDPIC, and EPIC algorithms can be derived by removing blocks of code.

Algorithm 3: Algorithms enforcing PIC supports

```
1  Procedure findPICSupport(i, Δ_ijk)
2      foreach a ∈ D_i do
3          P_i[a] ← min_{b∈D_j, c∈D_k} Δ_ijk(a, b, c);
4      foreach a ∈ D_i, b ∈ D_j do
5          E_ij[a, b] ← max_{c∈D_k} {P_i[a] − c_ijk(a, b, c) − c_ik(a, c) − c_jk(b, c)};
6      foreach a ∈ D_i, c ∈ D_k do
7          E_ik[a, c] ← max_{b∈D_j} {P_i[a] − c_ijk(a, b, c) − c_jk(b, c) − E_ij(a, b)};
8      foreach b ∈ D_j, c ∈ D_k do
9          E_jk[b, c] ← max_{a∈D_i} {P_i[a] − c_ijk(a, b, c) − E_ij(a, b) − E_ik(a, c)};
10     foreach a ∈ D_i, b ∈ D_j do Extend2To3(i, a, j, b, c_ijk, E_ij[a, b]);
11     foreach a ∈ D_i, c ∈ D_k do Extend2To3(i, a, k, c, c_ijk, E_ik[a, c]);
12     foreach b ∈ D_j, c ∈ D_k do Extend2To3(j, b, k, c, c_ijk, E_jk[b, c]);
13     foreach a ∈ D_i do Project3To1(c_ijk, i, a, P_i[a]);
14     α ← min_{a∈D_i} {c_i(a)};
15     UnaryProject(i, α);

16 Procedure findFullPICSupport(i, Δ_ijk)
17     foreach a ∈ D_i do
18         P_i[a] ← min_{b∈D_j, c∈D_k} Δ_ijk(a, b, c) + c_j(b) + c_k(c);
19     foreach b ∈ D_j do
20         E_j[b] ← max_{a∈D_i, c∈D_k} {P_i[a] − Δ_ijk(a, b, c) − c_k(c);
21     foreach c ∈ D_k do
22         E_k[c] ← max_{a∈D_i, b∈D_j} {P_i[a] − c_ijk(a, b, c) − E_j[b];
23     foreach b ∈ D_j do Extend1To2(j, b, c_ji, E_j[b]);
24     foreach c ∈ D_k do Extend1To2(k, c, c_ki, E_k[c]);
25     findPICSupport(i, Δ_ijk);

26 Procedure findEPICSupport(i)
27     α ← min_{a∈D_i} {c_i(a) + Σ_{Δ_ijk, i>j or i>k} min_{b∈D_j, c∈D_k} {Δ_ijk(a, b, c) + c_j(b) + c_k(c)}};
28     if α > 0 then
29         foreach Δ_ijk do
30             if ¬isSmallest(i, Δ_ijk) then findFullPICSupport(i, Δ_ijk);
31             R ← R ∪ ⋃_{Δ_ijk} {j, k};
32         UnaryProject(i, α);
```

EDPIC is enforced by Procedure `enforceEDPIC` in Algorithm 4. This procedure consists of four inner-while loops that respectively enforce EPIC, DPIC and PIC. It also enforces NC by calling `PruneVars` at line 24.

The first while-loop (lines 5-7) enforces EPIC. It first puts in R all variables that need to be checked for EPIC based on the auxiliary queue S (line 4). EPIC supports of variables $i \in R$ are enforced by Procedure `findEPICSupport` (line 7). When enforcing the existential support for i, EPIC is only responsible for triangles on which i is not the smallest variable because DPIC will take care of the remaining ones (Algorithm 3, line 27). If i has no fully supported value (i.e., $\alpha > 0$) such a value can be created by enforcing full PIC supports for every value of i on every triangle in which i is not the smallest variable

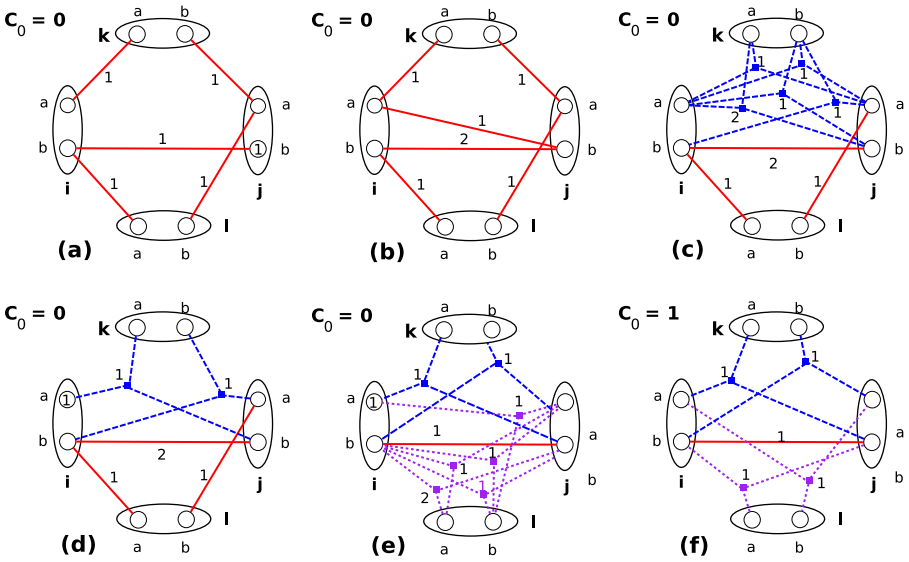


Fig. 14 Cost evolution in a CFN during the enforcement of full PIC supports (a) original problem with 5 binary cost functions $c_{ij}, c_{ik}, c_{il}, c_{jk}, c_{jl}, i < j < k < l$. It is FDAC but not FDPIC because of variable i where (i, a) and (i, b) cannot be fully extended on Δ_{ijk} and Δ_{ijl} respectively. (b) extending a cost of 1 from j_b on c_{ij} with $E_j[b] = 1$. (c) extending a cost of 1 from (i_a, j_b), (i_a, k_a) and (j_a, k_b) on c_{ijk} with $E_{ij}[a, b] = E_{ik}[a, a] = E_{jk}[a, b] = 1$. (d) projecting a cost of 1 from c_{ijk} on i_a with $P_i[a] = 1$. (e) extending a cost of 1 from $(i_b, j_b), (i_b, l_a)$ and (j_a, l_b) on c_{ijk} with $E_{ij}[b, b] = E_{il}[b, a] = E_{jl}[a, b] = 1$. (f) projecting a cost of 1 from c_{ijk} on i_b with $P_i[b] = 1$ and then enforcing NC by projecting a cost of 1 from c_i on $c_\varnothing$. The resulting problem is FDPIC.

(Algorithm 3, line 30). The EPIC supports of neighbor variables of i can also be destroyed (due to new values of non-zero cost made by the enforcement of full PIC supports on i) and thus are pushed back to R to be later checked for EPIC (Algorithm 3, line 31).

DPIC is enforced by the second while-loop at line 8. For a variable $j \in P$, only variables that are linked to j by a triangle (line 10) and are the smallest variable of the triangle (lines 11, 12) are considered for checking for DPIC.

PIC is enforced by two while-loops at lines 13 and 18. For a variable $j \in Q$, every neighbor variable of i is checked for PIC. For each $c_{ij} \in T$, i, j and all variables connected to both i and j are checked for PIC. Simple PIC supports are enforced in the reverse direction of the DAC order, i.e. in triangles in which the considered variables are not the smallest (lines 16-17, 21-23).

From Algorithm 4, algorithms for enforcing other levels of PICs can be obtained by appropriately keeping the right inner while-loops: the first loop (lines 4-7) for EPIC, the second one at line 8 for DPIC, the third one at line 13 for PIC, and three loops at lines 8, 13, 18 for FDPIC.

Algorithm 4: Algorithm enforcing EDPIC

```
1  Procedure enforceEDPIC()
2      S = P = Q = X;  T ← ∅;
3      while Q ≠ ∅ or P ≠ ∅ or S ≠ ∅ or T ≠ ∅ do
4          R ← S ∪ ⋃_{i∈S, Δ_ijk} {j, k};
5          while R ≠ ∅ do
6              i ← R.pop();
7              findEPICSupport(i);
8          while P ≠ ∅ do
9              j ← P.pop();
10             foreach Δ_ijk do
11                 if isSmallest(i, Δ_ijk)) then findFullPICSupport(i, Δ_ijk);
12                 if isSmallest(k, Δ_ijk)) then findFullPICSupport(k, Δ_ijk);
13         while Q ≠ ∅ do
14             j ← Q.pop();
15             foreach Δ_ijk do
16                 if ¬isSmallest(i, Δ_ijk) then findPICSupport(i, Δ_ijk);
17                 if ¬isSmallest(k, Δ_ijk) then findPICSupport(k, Δ_ijk);
18         while T ≠ ∅ do
19             c_ij ← T.pop();
20             foreach Δ_ijk do
21                 if ¬isSmallest(i, Δ_ijk) then findPICSupport(i, Δ_ijk);
22                 if ¬isSmallest(j, Δ_ijk) then findPICSupport(j, Δ_ijk);
23                 if ¬isSmallest(k, Δ_ijk) then findPICSupport(k, Δ_ijk);
24         PruneVars()
```

5.2 Enforcing maxRPCs

In contrast to PICs that are enforced on triangles sharing a variable, maxRPCs are enforced on triangles sharing two variables of a binary cost function. The extensible arc support of a value (i, a) in a binary cost function c_{ij} is stored in $\texttt{maxRPCSupport}[i, a, j]$ and the witness for this support on a variable k is stored in $\texttt{maxRPCWitness}[i, a, j, k]$. In our algorithms for enforcing soft maxRPCs, we will use a parameter named `fullLevel`, where `fullLevel` = `false` indicates that semi-fully extensible arc supports are used (FDmaxRPC) and `fullLevel` = `true` indicates that fully extensible supports are used (EmaxRPC). We will use the following functions:

- $\lambda_{ijk}(a, b, c) = c_{ik}(a, c) + c_{jk}(b, c) + c_{ijk}(a, b, c)$: denotes the incompletely combined cost of tuple (a, b, c) (excluding $c_{ij}(a, b)$ from $\Delta_{ijk}(a, b, c)$).
- $\lambda^k_{ij}(a, b) = \min_{c \in D_k} \lambda_{ijk}(a, b, c)$ denotes the minimum incompletely combined cost of tuples involving two values (i_a, j_b). This is the maximum cost that can be projected on the pair of values (i_a, j_b) from two sides c_{ik}, c_{jk} of the triangle Δ_{ijk} without creating negative costs.

$\operatorname{argmin}\lambda^k_{ij}(a,b)$ is used to denote a value $c \in D_k$ for which this minimum is reached. It is a simple witness for the pair (a,b) on the variable k.

- $\ddot{\lambda}^k_{ij}(a,b,\texttt{fullLevel}) = \begin{bmatrix} \min_{c\in D_k}\{\lambda_{ijk}(a,b,c) + c_k(c)\} & (\texttt{fullLevel} = \texttt{true}) \vee (i < k) \\ \min_{c\in D_k}\{\lambda_{ijk}(a,b,c)\} & (\texttt{fullLevel} = \texttt{false}) \end{bmatrix}$
  is similar to $\lambda^k_{ij}(a,b)$ but it takes into account the unary cost c_k of witnesses in the case of (1) fully extensible (`fullLevel=true`) or (2) semi-fully extensible arc supports on triangles w.r.t DAC order $(i < k)$.
  $\operatorname{argmin}\ddot{\lambda}^k_{ij}(a,b)$ is used to denote the value $c \in D_k$ for which this minimum is reached. It is a (full) witness for the pair (a,b) on the variable k.
- $\lambda\!\!\lambda_{ij}(a,b) = \sum_k\{\lambda^k_{ij}(a,b)\}$ is the maximum sum of costs that can be projected on the pair of values (i_a, j_b) from all triangles Δ_{ijk} sharing i,j.
- $\ddot{\lambda}\!\!\lambda_{ij}(a,b,\texttt{fullLevel}) = \sum_k\{\ddot{\lambda}^k_{ij}(a,b,\texttt{fullLevel})\}$ is similar to $\lambda\!\!\lambda_{ij}(a,b)$ but takes into account the unary costs of witnesses c_k according to `fullLevel` and the order between i and k as in the definition of $\ddot{\lambda}^k_{ij}$.

5.2.1 Enforcing maxRPC supports and witnesses

Simple maxRPC support for a value (i,a) on c_{ij} is enforced by Procedure `findmaxRPCSupport` in Algorithm 5. The main idea is to move costs from two sides c_{ik}, c_{jk} of all triangles Δ_{ijk} to c_{ij} via c_{ijk} (lines 18-20) and finally project costs from c_{ij} to (i,a) (line 21) in such a way that, for each triangle Δ_{ijk}, there exists a value $b \in D_j$ and a value $c \in D_k$ such that the binary and ternary costs involved in the tuple (i_a, j_b, k_c) decrease to 0. The maximum cost P_i that can be projected to (i,a) without creating negative costs is the minimum over all $b \in D_j$ of the binary cost of (i_a, j_b) that can be obtained by combining the original cost $c_{ij}(a,b)$ and the cost that can be shifted to it from all triangles (computed by function $\lambda\!\!\lambda_{ij}$, line 8).From this, it becomes possible to compute the actual cost $P_{ij}[a,b]$ that each triangle Δ_{ijk} provides to (i_a, j_b) for this amount of projection to (i,a). It is the minimum of what is needed for this pair of values $(P_i - c_{ij}(a,b))$ and of what can be provided for it by Δ_{ijk} (line 13). This condition guarantees that c_{ij} has enough costs to make a unary cost projection P_i on (i,a) without creating negative costs. Moreover, if more cost is projected on c_{ij}, this cannot lead to a unary cost projection greater than P_i. In order to project a cost of $P_{ij}[a,b]$ from c_{ijk} to (i_a, j_b) (line 20), each side (i_a, k_c) and (j_b, k_c) has to extend an amount of cost $E_{ik}[a,c]$ and $E_{jk}[b,c]$ to c_{ijk} (lines 19 and 18). These binary cost extensions $E_{ik}[a,c]$, $E_{jk}[b,c]$ are also the minimum of the available cost $c_{ik}(a,c), c_{jk}(b,c)$ that $(i_a, k_c), (j_b, k_c)$ have and the cost that they need to provide to c_{ijk} (lines 15, 17).

Full maxRPC supports (covering both fully extensible supports for EmaxRPC and semi-fully extensible for FDmaxRPC) are enforced by `findFullmaxRPCSupport` in Algorithm 5. The idea for enforcing a full maxRPC support for value (i,a) on c_{ij} is to extend unary costs from j to c_{ij} (line 28) and from third variables k to c_{ik} (line 32). Then, costs are moved in the same way as for simple

maxRPC support in Procedure `findmaxRPCSupport` (line 33). The maximum cost P_i that can be projected on (i, a) is recomputed by taking into account the unary cost c_j of supporting values and the unary costs c_k of witnesses via ⫛ (line 25). In order to achieve this unary projection, each value $(j, b), (k, c)$ needs to extend respectively on c_{ij} and c_{ik} an amount of cost E_j, E_k (lines 27 and 31). The order in which costs are moved when enforcing full maxRPC supports is described in Figure 15.

An EmaxRPC support for a variable i is enforced thanks to Procedure `findEmaxRPCSupport(`i`)` in Algorithm 5. It first checks the EmaxRPC property at line 36. If there does not exist any EmaxRPC support (line 37), the procedure will search for a full maxRPC support for any value of i in any cost function c_{ij} by calling `findFullmaxRPCSupport` with the option `fullLevel`=true. It only has to take care of the triangles Δ_{ijk} in which i is the smallest variable, because DmaxRPC takes care of the remaining cases (the condition at line 29 of Procedure `findFullmaxRPCSupport`).

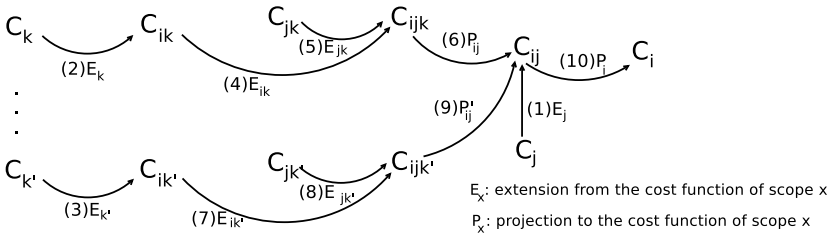


Fig. 15 The order of cost movements for enforcing simple full maxRPC supports where unary cost extensions are not included in the enforcement of simple PIC supports.

Example 3 Consider the CFN(a) in Figure 16. It is FDPIC but not FDmaxRPC because i_a has no full AC support in c_{ij} which can be can be extended on both Δ_{ijk} and Δ_{ijl}: (i_a, j_a) can be extended on Δ_{ijl} but not on Δ_{ijk} while (i_a, j_c) can be extended on Δ_{ijk} but Δ_{ijl}. The positive projection/extension costs computed by Procedure `findFullmaxRPCSupport(`$i, a, j,$`false)` are: P_i = $2, E_j[b] = 1$. The procedure extends a cost of 1 from j_b on c_{ij} and then calls $findmaxRPC(i, a, j)$ which computes the following positive projections/extension costs: $P_i = E_{jk}[a, b] = P_{ij}[a, a] = E_{jl}[c, b] = E_{jl}[c, b] = P_{ij}[a, a] = 2$. The final problem presented in Sub-figure 16(g) is FDmaxRPC.

Let j be a variable that had a change in the domain D_j or in unary cost c_j (increasing from 0). The former case can break the witness for simple or semi-full supports of variable i neighbor to j in some c_{ij}, while the last case can break the witness for semi-full and full supports. The check and search for new witnesses is performed by Algorithm 6.

Procedure `findWitness_remove` (i, k, j) handles the case of domain reduction in D_j. For any value (i, a), it checks the availability of its current (simple or semi-full) support in c_{ik} (line 5, algorithm 6), as well as the availability of

Algorithm 5: Algorithms enforcing maxRPC supports

1 **Procedure** `findmaxRPCSupport`(i, j)
2 | **foreach** $a \in D_i$ **do** `findmaxRPCSupport`(i, a, j) ;
3 | `UnaryProject`$(i,\ min_{a \in D_i}\{c_i(a)\})$;

4 **Procedure** `findFullmaxRPCSupport`$(i, j, \mathit{fullLevel})$
5 | **foreach** $a \in D_i$ **do** `findFullmaxRPCSupport`$(i, a, j, \mathit{fullLevel})$;
6 | `UnaryProject`$(i,\ min_{a \in D_i}\{c_i(a)\})$;

7 **Procedure** `findmaxRPCSupport`(i, a, j)
8 | $P_i \leftarrow \min_{b \in D_j}\{c_{ij}(a,b) + \text{从}_{ij}(a,b)\}$;
9 | $b^* \leftarrow \operatorname{argmin}_{b \in D_j}\{c_{ij}(a,b) + \text{从}_{ij}(a,b)\}$;
10 | **if** $P_i = 0$ **then** return;
11 | **foreach** Δ_{ijk} **do**
12 | | **foreach** $b \in D_j$ **do**
13 | | | $P_{ij}[a,b] \leftarrow \min\{P_i - c_{ij}(a,b),\ \text{人}^k_{ij}(a,b)\}$;
14 | | **foreach** $c \in D_k$ **do**
15 | | | $E_{ik}[a,c] \leftarrow \min\{c_{ik}(a,c),\ \max_{b \in D_j}\{P_i - c_{ijk}(a,b,c) - c_{ij}(a,b) - c_{jk}(b,c)\}\}$;
16 | | **foreach** $b \in D_j, c \in D_k$ **do**
17 | | | $E_{jk}[b,c] \leftarrow \min\{c_{jk}(b,c),\ P_i - c_{ijk}(a,b,c) - c_{ij}(a,b) - E_{ik}[a,c]\}$;
18 | | **foreach** $b \in D_j, c \in D_k$ **do** `Extend2To3`$(j, b, k, c, c_{ijk}, E_{jk}[b,c])$;
19 | | **foreach** $c \in D_k$ **do** `Extend2To3`$(i, a, k, c, c_{ijk}, E_{ik}[a,c])$;
20 | | **foreach** $b \in D_j$ **do** `Project3To2`$(c_{ijk}, i, a, j, b, P_{ij}[a,b])$;
21 | `Project2To1`$(c_{ij}(a,b), i, a, P_i)$;
22 | `maxRPCSupport`$[i, a, j] \leftarrow b^*$;
23 | **foreach** Δ_{ijk} **do** `maxRPCWitness`$[i, a, j, k] \leftarrow \operatorname{argmin} \text{人}^k_{ij}(a, b^*)$;

24 **Procedure** `findFullmaxRPCSupport`$(i, a, j, \mathit{fullLevel})$
| // condition: $i < j$ or fullLevel = true
25 | $P_i \leftarrow \min_{b \in D_j}\{c_j[b] + c_{ij}(a,b) + \text{从}_{ij}(a,b)\}$;
26 | **foreach** $b \in D_j$ **do**
27 | | $E_j \leftarrow P_i - \text{从}_{ij}(a,b) - c_{ij}(a,b)$;
28 | | `Extend1To2`(j, b, c_{ij}, E_j);
29 | **foreach** Δ_{ijk} *s.t (*$\mathit{fullLevel}$ *and* $\neg$`isSmallest`(i, Δ_{ijk})*) or (*$\neg\mathit{fullLevel}$ *and* $i < k$*)* **do**
30 | | **foreach** $c \in D_k$ **do**
31 | | | $E_k \leftarrow \min(c_k[c], \max_{b \in D_j}\{P_i - \Delta_{ijk}(a,b,c)\})$;
32 | | | `Extend1To2`(k, c, c_{ik}, E_k);
33 | `findmaxRPCSupport`(i, a, j);

34 **Procedure** `findEmaxRPCSupport`(i)
35 | `fullLevel` $\leftarrow$ true;
36 | $\alpha \leftarrow \min_{a \in D_i}\{c_i(a) + \sum_{c_{ij}} \min_{b \in D_j}\{c_{ij}(a,b) + \text{从}_{ij}(a,b,\texttt{fullLevel})\}\}$;
37 | **if** $\alpha > 0$ **then**
38 | | **foreach** c_{ij} **do**
39 | | | **foreach** $a \in D_i$ **do** `findFullmaxRPCSupport`$(i, a, j, \mathit{fullLevel})$;
40 | | | $R \leftarrow R \cup \bigcup_{c_{ij}}\{j\}$;
41 | | `UnaryProject`(i, α);

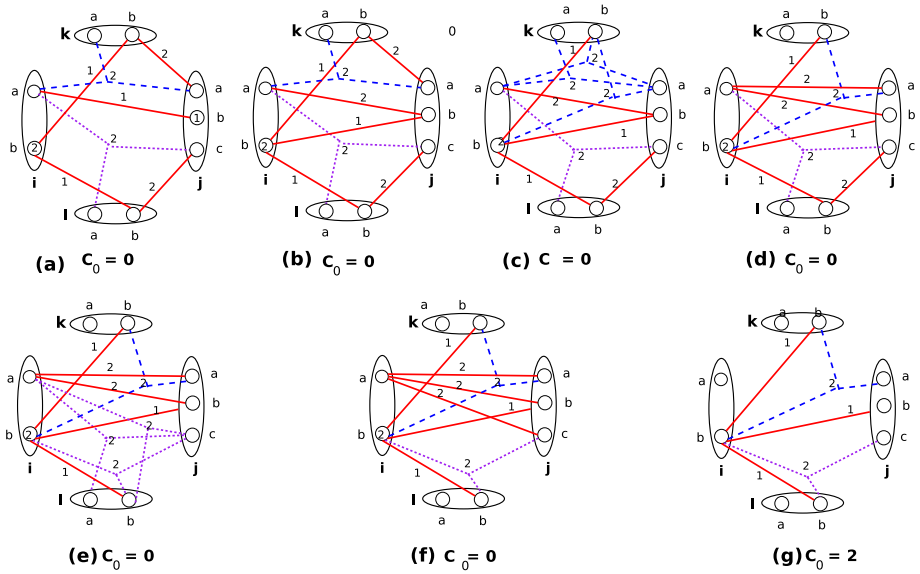


Fig. 16 Cost evolution during enforcing full maxRPC supports in a CFN (a) original problem with 5 binary cost functions $c_{ij}, c_{ik}, c_{il}, c_{jk}, c_{jl}$ and 2 ternary functions c_{ijk}, c_{ijl}, $i < \{j, k, l\}$. It is FDPIC but not FDmaxRPC due to i_a (no full maxRPC support in c_{ij}) (b) extending a cost of 1 from j_b on c_{ij} with $E_j[b] = 1$ (c) extending a cost of 2 from (j_a, k_b) on c_{ijk} with $E_{jk}[a, b] = 2$ (d) projecting a cost of 2 from c_{ijk} on (i_a, j_a) with $P_{ij}[a, a] = 2$ (e) extending a cost of 2 from (j_c, l_b) on c_{ijk} with $E_{jl}[c, b] = 2$ (f) projecting a cost of 2 from c_{ijk} on (i_a, j_c) with $P_{ij}[a, a] = 2$ (g) projecting a cost of 2 from c_{ij} on i_a with $P_i = 2$ and then making NC by projecting a cost of 2 from i to $c_\varnothing$. The resulting problem is FDmaxRPC.

the current witness for this support on D_j (line 7, algorithm 6). When the current support has been lost, another support needs to be created for (i, a) (line 11). Similarly, if the current witness is no longer available (line 7), another witness will be searched (line 8). If no witness exists (line 10), another simple or full support needs to be searched for (i, a) according to $i > k$ (line 13) or $i < k$ (line 14) respectively.

Procedure `findWitness_project` (i, k, j) handles the case where unary costs c_j become positive. This procedure is responsible for semi-full supports as it is only called in the while-loop enforcing DmaxRPC at line 7 of Algorithm 7. It differs from `findWitness_remove()` by the fact that unary costs are taken into account when checking the availability of the current supports and witnesses (line 19, 21) and when looking for another witness (⅄̈ instead of ⅄, line 22).

5.2.2 Soft maxRPC algorithms

Like PIC, EDmaxRPC includes all softening levels. We thus only present an algorithm for EDmaxRPC. maxRPC, DmaxRPC, FDmaxRPC, and EmaxRPC algorithms can be derived by keeping suitable blocks of code. EDmaxRPC is

Algorithm 6: Algorithms enforcing maxRPC witnesses

```
1  Procedure findWitness_remove (i, k, j)
       // called upon reduction of domain D_j, used to search for a witness
          in D_j for the support of values of i in c_ik such that i > j or i > k
2      foreach a ∈ D_i do
3          s ← maxRPCSupport[i, a, k];
4          need ← false; // need to search for a new simple support from
               scratch
5          if s ∈ D_k and c_ik(a, s) = 0 and (i > k or c_k(s) = 0) then
6              w ← maxRPCWitness[i, a, k, j];
7              if w ∉ D_j or Δ_ikj(a, s, w) > 0 then
8                  if ⅄^j_ik(a, s) = 0 then
9                      maxRPCWitness[i, a, k, j] ← argmin ⅄^j_ik(a, s);
10                 else need ← true;
11             else need ← true;
12             if need=true then
13                 if i > k then findmaxRPCSupport (i, a, k);
14                 else findFullmaxRPCSupport (i, a, k, ¬fullLevel);

15 Procedure findWitness_project (i, k, j)
       // called upon cost projection on a value of zero cost in D_j, used to
          search for a full witness in D_j for the full supports of values of
          i in c_ik such that i < j, i < k
16     foreach a ∈ D_i do
17         s ← maxRPCSupport[i, a, k];
18         need ← false; // need to search for a new full support from scratch
19         if s ∈ D_k and c_k(s) + c_ik(a, s) = 0 then
20             w ← maxRPCWitness[i, a, k, j];
21             if w ∉ D_j or c_j(w) > 0 or Δ_ikj(a, s, w) > 0 then
22                 if ⅄̈^j_ik(a, s, ¬fullLevel) = 0 then
23                     maxRPCWitness[i, a, k, j] ← argmin⅄̈^j_ik(a, s);
24                 else need ← true;
25             else need ← true;
26             if need=true then findFullmaxRPCSupport (i, a, k, ¬fullLevel);
```

enforced by Procedure `enforceEDmaxRPC()` in Algorithm 7. It consists of four inner-while loops that handle the same propagation queues S, P, Q, T as in the EDPIC algorithm.

The loop in line 12 enforces maxRPC by propagating domain reductions of j stored in the queue Q. For any neighbor value (i, a) of j, the deleted values in D_j could have been: (1) the simple maxRPC support for (i, a) when $i > j$; (2) the simple witness for the simple maxRPC supports of (i, a) in some c_{ik} when $i > k$; and (3) the simple witness for semi-full maxRPC supports of (i, a) in c_{ik} (of course $i < k$) when $i > j$. The deleted values in D_j could not have been the full supports and witnesses because they must have a cost of m to be removed. The loop must check and search for (1) a simple maxRPC (line 15-16) and (2) a simple witness (line 17).

Algorithm 7: Algorithm enforcing EDmaxRPC

```
1  Procedure enforceEDmaxRPC
2      while S ≠ ∅ or P ≠ ∅ Q ≠ ∅ or T ≠ ∅ do
3          R ← S ∪ ⋃_{i∈S,c_ij} {j};
4          while R ≠ ∅ do
5              j ← R;
6              findEmaxRPCSupport (j);
7          while P ≠ ∅ do
8              j ← P;
9              foreach c_ij, i < j do
10                 foreach a ∈ D_i do findFullmaxRPCSupport (i, a, j, ¬fullLevel) ;
11                 foreach Δ_ikj, i < k do findWitness_project (i, k, j) ;
12         while Q ≠ ∅ do
13             j ← Q;
14             foreach c_ij do
15                 if i > j then
16                     foreach a ∈ D_i do findmaxRPCSupport (i, a, j) ;
17                 foreach Δ_ikj s.t. i > j or i > k do findWitness_remove (i, k, j) ;
18         while T ≠ ∅ do
19             c_ij ← T; i* ←max{i, j}; j* ←min{i, j};
20             foreach a ∈ D_i* do
21                 findmaxRPCSupport(i*, a, j*);
22             foreach Δ_ijk do
23                 findWitness_remove(i, k, j);
24                 findWitness_remove(k, i, j);
25                 findWitness_remove(j, k, i);
26                 findWitness_remove(k, j, i);
27         PruneVars();
```

The loop at line 7 enforces DmaxRPC by propagating the increase from 0 of a unary cost c_j stored in P. The predecessors i of j (such that c_{ij} exists with $i < j$, line 9) may have lost full supports in c_{ij} and thus new full supports need to be searched for values of i (line 10). Moreover, the full supports in $c_{ik}, i < k$ (line 11) can have lost full witnesses on j if $i < j$ (line 9) and thus need to be searched for new witnesses (line 11).

The loop at line 4 enforces EmaxRPC by processing variables in the propagation queue R. The construction of R from the auxiliary queue S (line 3) is the same as in EDPIC. The search for a EmaxRPC support for a variable i is done by Procedure `findEmaxRPCSupport(i)` in Algorithm 5.

The loop at line 18 enforces maxRPC by propagating changes in binary costs c_{ij} (caused by unary cost extensions from the greater variable between i and j on c_{ij}) stored in queue T. Let i^* and j^* be respectively the greater and the smaller variable between i and j. The modified c_{ij}:

- cannot break the full or semi-full maxRPC supports of the smaller variable j^* because its values have been supported by values of zero cost.

- can break the simple maxRPC supports for the values of the greater variable i^* and thus new supports need to be searched for such values (line 21).
- can break the witnesses for maxRPC supports in c_{ik} (line 23, 24) or in c_{jk} (line 25, 26).

From Algorithm 7 enforcing EDmaxRPC, we can obtain algorithms for enforcing other levels of maxRPCs by keeping the first while-loop at line 4 for EmaxRPC, the loop at line 7 for DmaxRPC, the loop at line 12 for maxRPC, and the three loops at lines 7, 12, 18 for FDmaxRPC.

6 Experimentation

In this section we provide an experimental evaluation of our soft consistencies. During experimentation, it quickly appeared that maintaining such strong consistencies during search was too time consuming. We therefore decided to relax them in three different ways, denoted as our three *use cases*.

The three use cases we have considered for our TRICs (triangle-based consistencies) are denoted as:

- TRIC^p: uses some TRIC for pre-processing and EDAC during search.
- TRIC^{rp}: uses a restriction of some TRIC, a resTRIC (that will be explained later) for pre-processing, and EDAC during search.
- TRIC^{rs}: uses some resTRIC for both pre-processing and during search.

Restricted TRICs (resTRICs) are defined by limiting the number of triangles to be checked by the consistencies to some maximum. Defining the triangle density of a problem as the ratio of its number of triangles over the number of triangles in a complete graph, we observed that our soft consistencies are often too expensive when used for pre-processing problems having a triangle density larger than 10^{-4}. We have therefore chosen to bound the number of triangles that are processed in our restricted TRICs to a maximum number $(n(n-1)(n-2)/6)/10^4$ denoted as c^*. If $c^* < 10$, we do not enforce resTRICs and use EDAC only. Otherwise, we bound the number of processed triangles to c^*. When needed, the triangles chosen to be processed are selected as follows: for each binary cost function c_{ij} we compute its mean cost as $(\sum_{a \in D_i, b \in D_j} c_{ij}(a,b))/(|D_i| \times |D_j|)$. Triangles are then ranked by decreasing sum of the mean cost of the three involved binary cost functions and the first c^* are selected.

In order to evaluate the practical interest of establishing TRICs and their variants, we compared them to the default local consistency enforced in `toulbar2`[3]: EDAC. Indeed, EDAC is still the preferred local consistency for Depth First Branch-and-Bound search. We used a large set of benchmarks, as described in Table 1, which has recently been used in [15] for comparing the performance

[3] `http://mulcyber.toulouse.inra.fr/projects/toulbar2/` version 0.9.6 branch maxrpc.

of the `toulbar2` solver with other solvers[4]. This set consists of the following groups of benchmarks[5]:

Table 1 The set of benchmarks where each line corresponds to a category of benchmarks (#inst: number of instances, $\overline{n}$: mean number of variables, $\overline{d}$: mean domain size, $\overline{e}$: mean number of cost functions, $\overline{r}$: mean arity of cost functions, $\overline{c}$: mean number of triangles, $\overline{c'}$: mean number of triangles used by TRICs^{rp}, and $\overline{\Delta\text{dens}}$: mean triangle density).

Categories	#inst	$\overline{n}$	$\overline{d}$	$\overline{e}$	$\overline{r}$	$\overline{c}$	$\overline{c'}$	$\overline{\Delta\text{dens}}$
CVPR	1453							
ChineseChars	100	9147	2	276677	2	86557	86557	1.14E-06
ColorSeg	21	108910	9	474745	2	131805	32998	2.73E-09
GeomSurf-3	300	505	3	2140	3	8	8	4.46E-07
GeomSurf-7	300	505	7	2140	3	1366	1265	0.00018
InPainting	4	14400	4	57121	2	17732	17732	3.56E-08
Matching	4	19	19	166	2	701	0	0.679
MatchingSte	2	138407	18	414477	2	8	8	2.70E-14
ObjectSeg	5	68160	6	203947	2	31	31	5.91E-13
PhotoMont	2	469856	6	1408134	2	521	521	4.03E-14
SceneDecp	715	183	8	672	2	48	42	4.80E-05
MaxCSP	503							
BlackHole	37	114	27	657	2	5375	38	0.01
Coloring	22	120	4	1323	2	1227	277	0.024
Composed	80	58	10	517	2	791	0	0.079
EHI	200	306	7	4549	2	13604	475	0.0029
Geometric	100	50	20	471	2	1694	0	0.086
Langford	4	25	22	352	2	2722	0	0.736
QCP	60	159	7	1384	2	2671	108	0.0057
MaxSAT	427							
Haplotyping	100	150428	2	534105	483	61646	61646	2.39E-10
MaxClique	62	484	2	50093	2	1070886	2019	0.079
MIPLib	12	10523	2	45991	20	104	104	5.92E-07
PackupWei	99	9492	2	23731	61	9236	9236	6.87E-07
PlanWithPre	29	14991	2	111259	64	8026	8026	1.76E-06
TimeTabling	25	128243	2	785222	21	40052	40052	1.58E-09
Upgrad	100	18169	2	105097	77	1884	1884	1.88E-09
UAI	211							
Grid	21	3143	2	9379	2	2	2	3.74E-08
ImageAlign	10	191	70	1819	2	6218	37	0.0058
Linkage	22	917	5	1560	4	13	13	2.23E-07
ObjDetect	37	60	17	1830	2	34220	0	1
ProteinFold	21	486	267	2291	2	4698	273	0.52
Segment	100	229	12	851	2	315	185	0.00016
CFN	226							
Auction	170	140	2	3593	2	47707	57	0.0869
CELAR	16	126	44	641	2	837	46	0.228
Pedigree	10	1758	11	3247	3	70	70	3.96E-06
ProteinDsn	10	13	123	97	2	311	0	0.966
SPOT5	20	385	4	6603	3	35976	2900	0.0055

[4] We only excluded from the set all 35 Minizinc instances as well as two subcategories (UAI/DBN, 108 instances and CSP/warehouse, 55 instances) that contain no triangle of binary cost functions. Over the original 3,018 original instances, 2,820 remain.

[5] All the instances are available at `http://genoweb.toulouse.inra.fr/~degivry/evalgm`.

- WCSP: contains cost function networks extracted from the Cost Function Library[6], including Combinatorial Auctions [18], Radio Link Frequency Assignment problems [4], Mendelian error correction problems on complex pedigree [23], Computational Protein Design problems [1] and SPOT5 satellite scheduling problems [2].
- MaxCSP[7]: contains unsatisfiable binary CSP instances with constraints defined in extension, including BlackHole, Langford, Quasi-group completion problem,graph coloring, random composed, and random Geometric.
- UAI: consists of Markov Random Field problems that are collected from the Probabilistic Inference Challenge 2011[8] and Genetic Linkage Analysis problems[13].
- MaxSAT: contains Max-SAT instances that are collected from the Max-SAT Evaluation[9].
- CVPR: contains MRF instances from the Computer Vision and Pattern Recognition (CVPR) OpenGM2 benchmark[10].

Our algorithms were all implemented with a *time-limit* after which we consider the instance as not solved by this algorithm. For all categories of instances except ChineseChars and GeomSurf7, the time-limit was set to 1200 seconds. For ChineseChars and GeomSurf7 we set the time-limit to 3600 seconds as the instances in these categories were significantly harder.

Number of solved instances

Table 2 reports the number of instances per category of benchmarks that are solved by a Depth-First Branch&Bound algorithm using each consistency (EDAC and TRICs implemented in the three use cases). The first block of three lines reports the total number of instances solved by each algorithm in all categories. The remaining lines focus on selected categories where a difference in behavior was observed. These results show that in general TRIC^{rp} is the best choice, independently of the chosen triangle consistency. It however works best in combination with the strongest EDmaxRPC consistency. Then EDAC, TRIC^{rs}, and finally TRIC^{p} follow. One can observe that for the three use cases, TRICs (especially EDmaxRPC) are very efficient on the ChineseChars and GeomSurf7 problems, which are defined on grid graphs. While EDAC cannot solve any ChineseChars instance (this is also the case for all the other solvers reported in [15], including ILP and `toulbar2` using VAC), TRICs can solve a certain number of instances (8 for PIC and 16 EDmaxRPC). Similarly, TRICs can solve up to 5% instances more than EDAC on GeomSurf-7. The advantage of TRICs on these problems are more clearly shown in Table 3: many instances cannot be solved in 1 hour by maintaining EDAC but can be solved by TRICs in less than 100 seconds.

[6] `https://mulcyber.toulouse.inra.fr/scm/viewvc.php/trunk/?root=costfunctionlib`

[7] `http://www.cril.univ-artois.fr/CPAI08/`, `../~lecoutre/benchmarks.html`

[8] `http://www.cs.huji.ac.il/project/PASCAL/realBoard.php`

[9] `http://maxsat.ia.udl.cat:81/13/benchmarks/`

[10] `http://hci.iwr.uni-heidelberg.de/opengm2`

On the rest of the benchmarks, especially on categories having a very large mean triangle density such as Geometric, MaxClique, ProteinFold, Auction and CELAR (respectively 0.086, 0.079, 0.52, 0.0869 and 0.228), TRIC^p becomes worse than EDAC and solves 5,5%, 64%, 47%, 25% and 75% less instances respectively. The same behavior is observed on PackupWei (decrease by 11,5%). For these problems, the restricted versions TRIC^{rp} can significantly improve the efficiency of TRIC^p and give results comparable to EDAC, thanks to the reduction in the number of triangles processed. In all cases, TRIC^{rs} are less efficient than TRIC^{rp}.

Table 2: The number of instances per category solved in less than 1200 seconds (1 hour for the CVPR group). Each block corresponds to a category of benchmarks whose name and size are given in the two first columns. The number of instances per category solved by EDAC and TRICs are respectively given in the 3rd and the 8 last columns. Three lines of the combined blocks correspond respectively to the three use cases: TRICs^p, TRICs^{rp} and TRICs^{rs}. For the categories absent from the table, TRICs^p, TRICs^{rp} and TRICs^{rs} give the same result as EDAC. Best results are in bold.

Problems	inst		EDAC	PIC	DPIC	FDPIC	EDPIC	maxRPC	DmaxRPC	FDmaxRPC	EDmaxRPC
Summary	2820	p	2053	1972	1980	1979	1979	1967	1982	1980	1981
		rp		2051	2055	2060	2058	2059	2069	2072	**2074**
		rs		2013	2031	2030	2018	1993	2010	1992	1998
ChineseChars	100	p	0	8	8	10	10	10	9	10	**16**
		rp		9	7	9	10	14	10	13	15
		rs		9	9	10	10	11	10	12	11
GeomSurf-7	300	p	281	280	287	285	288	281	292	292	**295**
		rp		280	284	288	290	280	292	292	294
		rs		278	283	289	287	273	285	281	282
Coloring	22	p	17	**18**	**18**	17	17	**18**	**18**	**18**	**18**
		rp		17	17	17	17	17	17	17	17
		rs		17	17	16	16	17	16	16	16
Geometric	100	p	91	88	87	87	86	87	87	86	86
		rp		92	91	92	92	92	**93**	91	92
		rs		90	90	90	86	87	87	88	86
QCP	60	p	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**
		rp		**14**	**14**	**14**	**14**	**14**	**14**	**14**	**14**
		rs		**14**	**14**	**14**	**14**	13	13	13	**14**
Haplotyping	100	p	1	1	1	1	1	1	**2**	**2**	1
		rp		1	1	1	1	**2**	**2**	**2**	**2**
		rs		1	1	1	1	1	1	1	1
MaxClique	62	p	**33**	15	14	15	14	13	12	14	13
		rp		28	29	30	29	29	29	30	30
		rs		24	27	24	26	23	23	22	22
MIPLib	12	p	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**
		rp		**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**
		rs		2	2	2	2	2	2	2	2
PackupWei	99	p	**52**	48	47	47	47	47	46	47	47
		rp		48	46	48	47	48	47	47	47
		rs		41	39	42	40	41	39	40	40
Upgrad	100	p	**100**	**100**	**100**	**100**	98	**100**	**100**	99	98

Continued on next page

Table 2 – *Continued from previous page*

Problems	inst		EDAC	PIC	DPIC	FDPIC	EDPIC	maxRPC	DmaxRPC	FDmaxRPC	EDmaxRPC
		rp		96	**100**	96	92	97	99	98	97
		rs		92	**100**	94	92	89	96	93	91
ImageAlign	10	p	**10**	7	9	7	7	6	7	5	5
		rp		**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
		rs		**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
Linkage	22	p	13	13	13	13	14	14	13	**15**	**15**
		rp		14	13	14	14	14	14	**15**	**15**
		rs		11	10	11	10	9	10	10	9
ProteinFold	21	p	19	10	10	10	10	10	10	10	10
		rp		**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
		rs		**20**	**20**	**20**	**20**	**20**	**20**	19	**20**
Segment	100	p	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**
		rp		99	**100**	**100**	**100**	99	99	**100**	**100**
		rs		98	**100**	99	98	98	**100**	98	98
Auction	170	p	166	126	128	130	129	125	129	129	125
		rp		**167**	**167**	166	**167**	**167**	**167**	**167**	165
		rs		154	156	156	154	147	146	146	144
CELAR	16	p	**12**	4	4	3	3	3	3	3	3
		rp		**12**	**12**	11	11	**12**	**12**	**12**	**12**
		rs		11	**12**	11	11	11	11	11	11
Matching	4	p	**4**	**4**	**4**	**4**	**4**	2	**4**	0	0
		rp		**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
		rs		**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
ProteinDsn	10	p	**9**	5	5	5	5	5	5	5	4
		rp		**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**
		rs		**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**

In Figure 17 we present a cactus plot over all instances of all categories. This gives us an overall view on the performance of our 18 TRICs and EDAC. This cactus plot makes obvious that TRICs^{rp} are consistently the best while TRICs^{p} are the worst, not only in terms of the number of solved instances, as shown in Table 2, but also in terms of running time. EDAC is ranked somewhere among the worst of the TRICs^{rp} use case. EDmaxRPC^{rp}, FDmaxRPC^{rp}, and DmaxRPC^{rp} are the three algorithms that seem to be the more reliable, being among the best whatever the time allowed. Surprisingly, there is no consistency that clearly outperforms the others on all use cases.

The apparent dominance of TRICS^{rp} over TRICS^{p} is also the direct consequence of the presence of a number of relatively easy instances in our set of benchmark problems. As we have seen, on families of hard instances, TRICS^{p} can be more efficient than TRICS^{rp}.

Number of backtracks

Figure 18 presents the mean number of backtracks, computed over all instances that could be solved by all approaches, for EDAC and the three use cases of TRICs. It shows that the number of backtracks is consistent with the

Table 3 Solving time (in seconds) for maintaining EDAC and using TRICs^p for a subset of benchmarks. This subset contains only ChineseChars and GeomSurf-7 instances that respectively can and cannot be solved by one of the consistencies in 1 hour. "–" means that the problem cannot be solved. Best results are in bold.

Problem	EDAC	PIC	DPIC	FDPIC	EDPIC	maxRPC	DmaxRPC	FDmaxRPC	EDmaxRPC
ChineseChars (TST_)									
0012_88_103	-	195	575	34	41	**21**	119	27	44
0020_96_94	-	-	2647	249	260	363	1709	**158**	**158**
0024_88_126	-	-	-	-	-	-	-	-	**662**
0027_88_109	-	-	-	-	-	-	-	-	**865**
0041_88_96	-	-	-	3149	1569	269	-	760	**114**
0047_112_121	-	215	83	23	49	**18**	26	28	64
0052_96_107	-	1230	3564	1842	154	77	690	183	**46**
0059_104_73	-	130	93	37	32	**11**	28	20	33
0067_96_121	-	201	590	117	143	47	151	**41**	76
0070_88_96	-	460	2194	392	158	**56**	210	99	73
0084_120_115	-	-	-	-	-	-	-	-	**416**
0087_88_124	-	-	-	-	-	-	-	-	**1910**
0089_72_92	-	-	-	-	-	-	-	-	**1148**
0099_72_105	-	502	2012	112	101	**30**	347	65	75
0100_80_102	-	1199	-	591	532	227	1577	402	**78**
GeomSurf-7									
gm113	1487	-	349	254	486	678	166	**95**	138
gm125	-	-	1877	2938	-	-	344	**274**	2516
gm126	-	-	2135	-	-	-	1196	**119**	201
gm144	-	-	-	-	2806	-	2770	**1461**	1481
gm157	-	-	1914	2173	-	-	403	438	**262**
gm169	-	-	-	-	3146	-	2108	2971	**962**
gm179	-	1431	366	806	137	-	74	72	**67**
gm186	-	-	-	-	1674	-	951	842	**223**
gm187	-	-	2206	365	281	-	685	600	**182**
gm189	-	-	-	-	-	-	-	-	**2961**
gm223	-	-	2383	-	922	-	477	**426**	1473
gm246	-	-	-	-	-	-	-	-	**1744**
gm256	-	-	-	-	-	-	-	-	**2291**
gm25	1490	-	1387	-	653	2880	279	**171**	395
gm269	-	-	-	1656	**452**	-	1046	2948	1182
gm275	-	-	-	-	-	-	1180	2664	**568**

strength of the consistencies. In almost all use cases, TRICs use less backtracks than EDAC, TRICs^p being the best. Compared to EDAC, (1) TRICs^p reduce the number of backtracks by 35% (2) TRICs^{rp} only slightly decrease the number of backtracks because of their reduced strength, and on many categories of benchmarks with a significantly reduced number of triangles they become almost equivalent to EDAC (3) TRICs^{rs} produce a smaller number of backtracks than TRICs^{rp} (thanks to the strengthened filtering during search) but still larger than TRICs^p (because of their significant reduced strength).

To summarize these experiments, it appears that some problems, as illustrated by the ChineseChars and GeomSurf7 cases, require the tightened lower

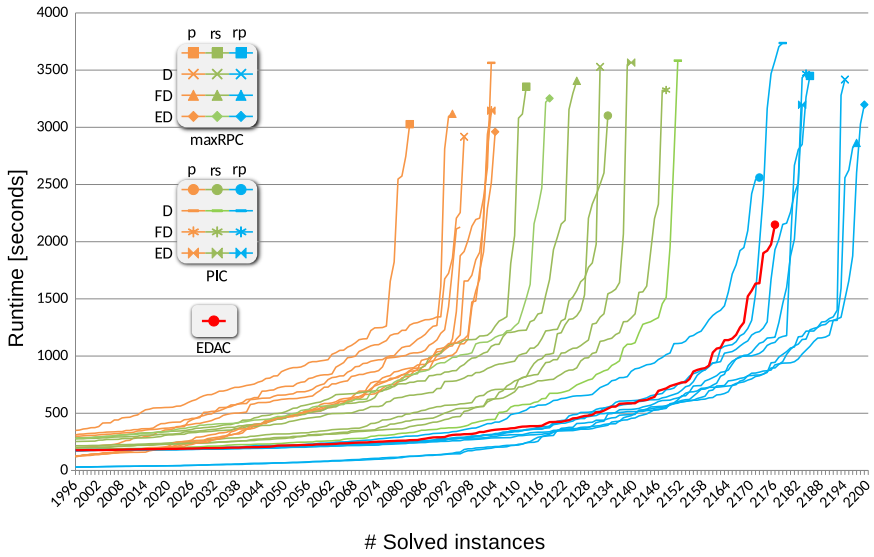


Fig. 17 Cactus plot on the full set of benchmarks. A point (x, y) for a method m on this diagram means that method m is able to solve x problems if a deadline of y seconds is used for each problem independently.

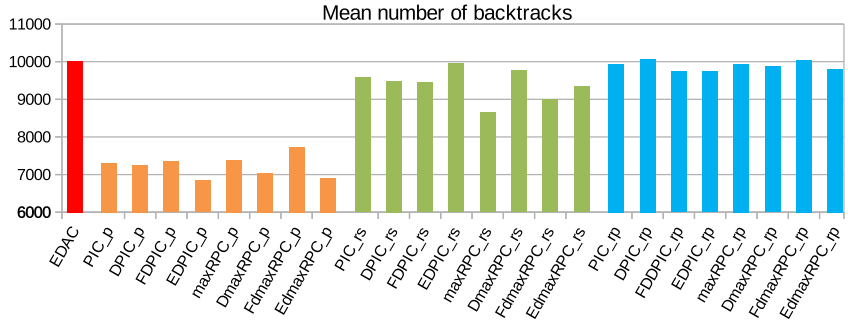


Fig. 18 The mean number of backtracks, computed on the overall set of benchmarks, that are used by EDAC and TRICs in the three use cases

bounds offered by TRICs to be solved. On these problems, when using TRICs for pre-processing, we can actually solve more instances in less time than EDAC. On these problems, restricted versions slightly reduce the advantage of TRICs. However, on problems having large triangle density, TRICs becomes significantly slower and thus solve less instances than EDAC. In these cases, using the restricted versions for pre-processing allows to improve the results. Finally, when the restricted TRICs are applied during both pre-processing and search, they always behave worse than when used for pre-processing only.

7 Conclusion

In this paper, we have proposed six softening levels for strong triangle-based consistencies. This gives rise to eighteen soft extensions of hard RPC, PIC and maxRPC to CFNs. We have done a pairwise comparison of all these consistencies, among themselves and against their AC counterparts. We have shown that the new consistencies are strictly stronger than their AC counterparts in the sense that they provide tighter lower bounds than ACs. This improvement in lower bound is important for reducing the number of backtracks and for accelerating search. We have proposed algorithms for enforcing the soft consistencies of the PIC and maxRPC families. The experimentation shows that our soft consistencies are efficient when applied as a pre-processing on graphs with a relatively low triangle density such as ChineseChars and GeomSurf-7, defined on grid graphs. However, their performance decreases on graphs having a large triangle density. To make these soft consistencies practicable on problems where the number of triangles is large, we designed a restricted version by limiting the number of triangles to be processed. The best choice overall seems to be to use this restricted version for pre-processing and to switch to EDAC during search.

Acknowledgements This work has been partly funded by the "Agence nationale de la Recherche", reference ANR-10-BLA-0214.

References

1. Allouche, D., Bessiere, C., .Boizumault, P., Givry, S., Gutierrez, P., Loudni, S., Metivier, J., Schiex, T.: Decomposing global cost functions. In: Proc. of AAAI (2012)
2. Bensana, E., Lemaître, M., Verfaillie, G.: Earth observation satellite management. Constraints **4**(3), 293–299 (1999)
3. Berlandier, P.: Improving domain filtering using restricted path consistency. In: Proceedings IEEE Conference on Artificial Intelligenece and Applications (CAIA'95) (1995)
4. Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.: Radio link frequency assignment. Constraints **4**, 79–89 (1999)
5. Cooper, M., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M., Werner, T.: Soft arc consistency revisited. Artificial Intelligence **174**, 449–478 (2010)
6. Cooper, M., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M.: Virtual Arc Consistency for Weighted CSP. In: Proc. of AAAI'2008. Chicago, USA (2008)
7. Cooper, M.C.: Reduction operations in fuzzy or valued constraint satisfaction. Fuzzy Sets and Systems **134**(3), 311–342 (2003)
8. Cooper, M.C.: High-order consistency in Valued Constraint Satisfaction. Constraints **10**, 283–305 (2005)
9. Cooper, M.C., de Givry, S., Schiex, T.: Optimal soft arc consistency. In: Proc. of IJCAI'2007, pp. 68–73. Hyderabad, India (2007)
10. Cooper, M.C., Schiex, T.: Arc consistency for soft constraints. Artificial Intelligence **154**(1-2), 199–227 (2004)
11. Debruyne, R., Bessière, C.: From restricted path consistency to max-restricted path consistency. In: Proc. of CP'97, no. 1330 in LNCS, pp. 312–326. Springer-Verlag, Linz, Austria (1997)
12. Dehani, D., Lecoutre, C., Roussel, O.: Extension des cohérences wcsps aux tuples. In: Proc. of JFPC-13 (2013)

13. Favier, A., de Givry, S., Legarra, A., Schiex, T.: Pairwise decomposition for combinatorial optimization in graphical models. In: Proc. of IJCAI'11. Barcelona, Spain (2011)
14. Freuder, E.C., Elfe, C.D.: Neighborhood inverse consistency preprocessing. In: Proc. of AAAI'96. Portland, OR (1996)
15. Hurley, B., O'Sullivan, B., Allouche, D., Katsirelos, G., Schiex, T., Zytnicki, M., de Givry, S.: Multi-Language Evaluation of Exact Solvers in Graphical Model Discrete Optimization. In: Proc. of CP-AI-OR'2016. Banff, Canada (2016)
16. Larrosa, J.: On arc and node consistency in weighted CSP. In: Proc. AAAI'02, pp. 48–53. Edmondton, (CA) (2002)
17. Larrosa, J., de Givry, S., Heras, F., Zytnicki, M.: Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In: Proc. of the 19^{th} IJCAI, pp. 84–89. Edinburgh, Scotland (2005)
18. Larrosa, J., Heras, F., de Givry, S.: A logical approach to efficient max-sat solving. Artif. Intell. **172**(2-3), 204–233 (2008)
19. Larrosa, J., Schiex, T.: In the quest of the best form of local consistency for weighted CSP. In: Proc. of the 18^{th} IJCAI, pp. 239–244. Acapulco, Mexico (2003)
20. Larrosa, J., Schiex, T.: Solving weighted CSP by maintaining arc consistency. Artif. Intell. **159**(1-2), 1–26 (2004)
21. Lee, J., Leung, K.: Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. In: Proc. of the 21^{rd} IJCAI, pp. 559–565. Pasadena (CA), USA (2009)
22. Lee, J., Leung, K.: Consistency techniques for flow-based projection-safe global cost functions in weighted constraint satisfaction. Artificial Intelligence **43**, 257–292 (2012)
23. Sánchez, M., de Givry, S., Schiex, T.: Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. Constraints **13**(1-2), 130–154 (2008)
24. Schiex, T.: Arc consistency for soft constraints. In: Principles and Practice of Constraint Programming - CP 2000, *LNCS*, vol. 1894, pp. 411–424. Singapore (2000)
25. Schiex, T., Fargier, H., Verfaillie, G.: Valued constraint satisfaction problems: hard and easy problems. In: Proc. of the 14^{th} IJCAI, pp. 631–637. Montréal, Canada (1995)
26. Simoncini, D., Allouche, D., de Givry, S., Delmas, C., Barbe, S., Schiex, T.: Guaranteed discrete energy optimization on large protein design problems. Journal of chemical theory and computation **11**(12), 5980–5989 (2015)
27. Traoré, S., Allouche, D., André, I., de Givry, S., Katsirelos, G., Schiex, T., Barbe, S.: A new framework for computational protein design through cost function network optimization. Bioinformatics **29**(17), 2129–2136 (2013)