# THÈSE

**En vue de l'obtention du**

**DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE**

**Délivré par :** *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

**Présentée et soutenue le** *15/01/2015* **par :**

**Thi Hong Hiep NGUYEN**

**Strong consistencies for Weighted Constraint Satisfaction Problems**

**JURY**

| | | |
|---|---|---|
| MARTIN COOPER | Professeur d'Université | Président du Jury |
| SAMIR LOUDNI | Maître de conférences | Membre du Jury |
| CHRISTOPHE LECOUTRE | Professeur d'Université | Membre du Jury |
| PHILIPPE JÉGOU | Professeur d'Université | Membre du Jury |
| CHRISTIAN BESSIÈRE | Directeur de Recherche | Membre du Jury |
| THOMAS SCHIEX | Directeur de Recherche | Membre du Jury |

**École doctorale et spécialité :**
*MITT : Domaine STIC : Intelligence Artificielle*
**Unité de Recherche :**
*Institut National de la Recherche Agronomique (UR 0875)*
**Directeur(s) de Thèse :**
*Thomas SCHIEX* et *Christian BESSIÈRE*
**Rapporteurs :**
*Christophe LECOUTRE* et *Philippe JÉGOU*

# Remerciements

En premier lieu, je tiens à exprimer ma plus profonde gratitude à mon directeur de thèse Thomas Schiex pour avoir assuré la direction scientifique de ce travail de recherche, pour la confiance qu'il m'a accordé et pour son soutien déterminant tout au long de ces presque quatre années. Ses conseils judicieux et ses encouragements m'ont aidée à surmonter les difficultés au travail ainsi que dans la vie pour aller jusqu'au bout de cette thèse.

Toute ma reconnaissance va également à mon Co-directeur de thèse, Christian Bessière, pour sa direction scientifique, ses commentaires pertinents et ses précieuses corrections. Je suis consciente d'avoir eu la chance de travailler avec lui.

Ma reconnaissance s'adresse aussi à Simon de Givry pour son soutien technique qui m'a permis d'avancer de manière efficace dans mon travail d'implémentation et d'expérimentation. Je tiens à remercier Mickaël qui m'a aidée à régler des problèmes informatiques sur mon ordinateur.

Mes plus sincères remerciements vont également à l'ensemble des personnes qui m'ont fait l'honneur de juger ce travail : à Martin Cooper pour avoir accepté de présider le jury, à Christophe Lecoutre et Phillipe Jégou pour leurs commentaires avisés en tant que rapporteurs, et enfin, à Samir Loudi pour ses remarques précises lors de ma soutenance.

Je remercie les collègues qui ont été à mes côtés plus ou moins longtemps durant cette période de ma vie et avec lesquels je partage de nombreux souvenirs. Magali, mon dieu, merci pour ton aide administrative et tes indications sur le travail de thèse. Un grand merci à ma fillette Charlotte pour ta bonne humeur et ton encouragement qui m'ont aidée à trouver l'équilibre dans la vie et la motivation au travail lorsque j'ai rencontré des difficultés personnelles. Je tiens à remercier Julia et Anaïs pour les bons moments qu'on a partagés et pour les soirées au bord de la Garonne que je n'oublierai jamais. Je voudrais remercier Mahuna et Aurélie pour leur gentillesse et leur sympathie ; grâce à eux, j'ai pu me rapprocher de la bande des non-permanents.

Je tiens à remercier tous les membres, stagiaires, chercheurs, techniciens et secrétaires de l'unité MIAT du laboratoire INRA pour leur gentillesse et sympathie. Merci a Fabienne Ayrignac pour m'avoir accueillie au sein du labo et pour m'avoir aidée pendant mes premiers jours en France. Un grand merci à David Allouche pour les temps d'échange qu'on a partagés, en particulier nos discussions sur la vie, et pour les grands moments à Noël que j'ai passé avec ta famille.

Je remercie mes amis vietnamiens à Toulouse et au Vietnam pour leur soutien et leur aide: Chien, Bai, Hanh, Luyen, Ly, Diep, Thao, Thu, Trung, Lam, Hoai, Bao Anh, Phuong, Binh .... Grâce à eux, ces quatres années sont passées plus vite. Un grand merci à Chien et Bai pour leurs encouragements, leurs soutiens, leurs conseils et le partage des moments les plus difficiles de ma vie. Vous êtes devenus mes grande soeur et grand frère.

Enfin, j'exprime ma gratitude à ma famille pour leurs encouragements et leur amour infini. Cảm ơn bố mẹ đã sinh ra con, luôn động viên và dõi bước trên đường đời của con. Khi vui cũng như buồn, thành công hay thất bại, bố mẹ luôn bên con và nâng đỡ cho con. Tình yêu ấy là động lực để con vượt qua mọi sóng gió cuộc đời và gặt hái nhiều thành công.

# Contents

# Chapter 1

# Introduction

## 1.1 Thesis's motivation

A wide variety of real-world optimization problems can be modeled as weighted constraint satisfaction problems (WCSPs). A WCSP or a cost function network (CFN) consists of a set of variables and a set of cost functions where the costs are associated to assignments to the variables, expressing preferences between solutions. The goal is to find an assignment to the variables which minimizes the combined costs. This kind of problem has applications in *resource allocation* [Cabon et al., 1999], *combinatorial auctions*, *bioinformatics* [Traoré et al., 2013]...

WCSPs, as many optimization problems, can be solved by Depth First Branch-and-Bound Search. This method allows to keep a reasonable space complexity but requires good (strong and cheap) lower-bounds on the minimum cost of a node to be efficient. The quality of lower-bounds should be put into balance with their computational time in order to accelerate the search.

In the last years, increasingly better lower-bounds have been designed by enforcing soft consistencies in WCSPs. Soft consistencies aim at simplifying WCSPs by defining properties on the cost of values or assignments of values that must be satisfied. Soft consistencies are enforced by iteratively applying so-called *Equivalence Preserving Transformations* (EPTs, [Cooper and Schiex, 2004]). EPTs extend the traditional local consistency operations used in classical CSPs by moving costs between cost functions of different arities while keeping the problem equivalent. By ultimately moving cost to a constant function with empty scope, they are able to provide a lower-bound on the optimum cost which can be incrementally maintained during Branch-and-Bound search.

Among the proposed soft consistencies for WCSPs, soft arc consistencies such as AC*, DAC*, FDAC* or EDAC* [Larrosa et al., 2005], extended from the classical AC used in classical CSPs, require a small enforcing time but do not always provide tight lower-bounds that lead to massive pruning. They are enforced by applying, in an arbitrary order, specific EPTs, called Soft Arc Consistency (SAC) operations, which shift costs between values and cost functions of arity either greater than 1 or equal to 0 (defining the lower-bound). Optimal Soft Arc Consistency (OSAC) can provide optimal lower-bounds (in the sense that applying any sequence of SAC operations cannot result in a better lower-bound) but is too expensive for general use. Instead, Virtual Arc Consistency (VAC [Cooper et al., 2008, 2010]) is cheaper than OSAC while providing lower-bounds stronger than other soft arc consistencies.This is obtained by using a planned sequence of SAC operations defined from the result of enforcing classical AC on a classical constraint network which forbids combinations of values with non zero costs.

Beyond arc consistencies, up to now, only few high order consistencies have been proposed for WCSPs such as complete $k$-consistency [Cooper, 2005] extended from hard $k$-consistency, Tuple Consistency [Dehani et al., 2013] considered soft arc consistencies applied to tuples (combinations of values) instead of values, ...

Indeed, this thesis is motivated by two questions. First, is it possible to improve the efficiency of enforcing for existing soft consistencies? Second, can we propose new soft consistencies for WCSPs that provide strong lower-bounds but have a reasonable time complexity?

We realized that VAC can be accelerated by exploiting its iterative behavior. Indeed, each iteration of VAC requires to enforce classical AC on the hardened version of the current network. But this network is just the result of the incremental modifications done by EPTs applied in the previous iterations. Similarly, maintaining VAC during search requires to enforce classical AC on the hardened version of the current network which are slightly modified due to branch operations. This situation, where AC is repeatedly enforced on incrementally modified versions of a constraint network, has been previously considered in Dynamic Arc Consistency algorithms [Barták and Surynek, 2005; Bessière, 1991] for Dynamic CSPs [Dechter and Dechter, 1988]. Thus, integrating Dynamic Arc Consistency into VAC is a potential approach for improving the enforcing time of VAC by inheriting the work done in previous iterations of VAC or in parent nodes.

Soft consistencies can be extended from hard consistencies by replacing the notion of compatibility in classical CSPs by the notion of zero-cost in WCSPs. For example, hard arc consistency in binary CSPs requires that any value of any variable has a compatible value, called an (hard) arc support, in every adjacent domain. Soft arc consistencies redefine this feature by replacing the compatibility of the supporting values by the requirement of zero-cost between values and their supporting values.

Based on this principle, we can create strong consistencies for WCSPs by extending (hard) high order consistencies (HOCs). Among hard HOCs, we are interested in the group of triangle-based consistencies consisting of RPC, PIC, maxRPC because of two reasons: (1) these are domain-based consistencies; (2) they have a strong pruning power while having a reasonable time complexity. Thus, extending these potential hard consistencies to WCSPs would lead to strong soft consistencies, as desired.

In summary, the objective of this thesis is to focus on soft consistencies for efficiently solving WCSPs. We would like to improve the enforcing time of VAC by using Dynamic Arc Consistency for maintaining classical Arc Consistency in the hardened version of WCSPs. In addition, we would like to investigate soft high order consistencies by extending hard triangle-based consistencies to WCSPs in order to provide strong lower-bounds for Branch-and-Bound search.

## 1.2 Thesis's organization and contributions

### Organization

This thesis consists of three chapters. A background survey in the domain of (Weighted) Constraint Satisfaction Problems is introduced in the first chapter while the thesis contributions are presented in the two last chapters.

Chapter 2 outlines in the first section background knowledge on classical CSPs and hard consistencies. Then, it reviews Weighted CSPs, operations of shifting costs, basic soft consistencies and algorithms enforcing them.

In Chapter 3, we propose Dynamic Virtual Arc Consistency, an improved version of Virtual Arc Consistency for WCSPs. An enforcing algorithm and the properties of the algorithm are also discussed in this chapter.

In the last chapter, we propose soft high order consistencies extended from hard RPC, PIC and maxRPC. A general comparison on the performance of soft domain-based consistencies is also proposed in this chapter. The enforcing algrorithms and experimental results of soft high order consistencies will be presented.

### Contributions

This thesis has two main contributions for efficiently solving WCSPs: improving the efficiency of enforcing VAC and proposing high order consistencies. An outline of these contributions is presented below.

#### Dynamic virtual arc consistency

By integrating Dynamic Arc Consistency inside VAC algorithm to maintain hard Arc Consistency in the hardened version of WCSPs $P$, called Bool($P$), we can dynamically maintain VAC during iterations of VAC and during search. This new algorithm is named Dynamic Virtual Arc Consistency (DynVAC). We have proposed two variants of DynVAC, called normal and full DynVAC when maintained during search.

- Normal DynVAC maintains arc consistency in Bool($P$) inside each search node, i.e. during iterations of VAC, and rebuilds Bool($P$) from scratch when the search branches out.
- Full DynVAC maintains arc consistency in Bool($P$) both inside nodes and during search.

We have given an implementation for two variants of DynVAC in the solver `toulbar2`. Domain-based revision order heuristics are also implemented inside each variant of DynVAC.

### Soft high order consistencies

We have proposed 18 soft high order consistencies extended from hard RPC, PIC, and maxRPC, with six variants for each one. The six soft consistencies based on hard RPC are simple RPC, directional RPC, full directional RPC, existential RPC, existential directional RPC and virtual RPC. This is the same for the cases of hard PIC and hard maxRPC.

Two "stronger" relations for comparing soft domain-based consistencies have also been proposed in this thesis. Based on these relations, we have given a general comparison of our soft high order consistencies and soft arc consistencies.

We have designed algorithms for enforcing soft high order consistencies extended from hard PIC and maxRPC. These algorithms have been characterized in terms of termination, time and space complexities.

Finally, an implementation of soft RPCs and soft maxRPCs in `toulbar2` has been done and experimental results are presented. The impact of our consistencies on the search has also been analyzed.

# Chapter 2

# Background

## 2.1 Constraint satisfaction and hard consistencies

### 2.1.1 Constraint satisfaction problems

**Definition 2.1 (Constraint satisfaction problem)** *A constraint satisfaction problem (CSP) or constraint network (CN) is a tuple $P = (X, D, C)$. $X$ is a finite set of variables. Each variable $x_i \in X$ has a finite domain $D(x_i) \in D$. $C$ is a finite set of constraints. Each constraint $c_S \in C$ is a relation defined on a subset of variables $S \subseteq X$ that specifies the allowed combinations of values for variables on $S$. $S$ and $|S|$ are called the scope and the arity of the constraint.*

Many academic and real problems can be formulated as CSPs. For example, the $N$-queens problem can be modeled as a CSP with $N$ variables $X = (x_1, \dots, x_N)$ where $x_i$ is defined for the queen placed in the column $i$. The value of $x_i$ represents its line number and the domain of $x_i$ is a set of integers $D(x_i) = \{1..N\}$. The constraints on non-sharing of lines and diagonals by 2 queens $x_i$ and $x_j$ are respectively represented as $x_i \neq x_j$ and $|x_i - x_j| \neq |i - j|$.

**Definition 2.2 (Normalized CSP)** *A CSP is normalized iff there does not exist any two constraints defined on the same scope.*

A constraint defined over a scope of $k$ variable is called k-ary. The notation $c_i$ and $c_{ij}$ denote the unary and binary constraint on variable $x_i$ and on variables $x_i, x_j$ respectively. A binary CSP has only unary and binary constraints and a non-binary CSP has also non-binary constraints.

**Definition 2.3 (Instantiation and Solution)** *Given a CSP $P = (X, D, C)$.*

- *An instantiation $\tau_S$ on a set of variables $S \subseteq X$ is an assignment of values for variables in $S$: $\tau_S \in \ell(S)$. $S$ is called the scope of $\tau$.*
- *$\tau_S$ is a partial instantiation if $S \subset X$ or a complete instantiation if $S = X$.*
- *An instantiation $\tau_S$ is locally consistent if it satisfies all constraints $c_T$ such that $T \subseteq S$.*
- *A solution of the CSP is a complete instantiation which is locally consistent. The set of solutions of $P$ is denoted by $sol(P)$.*

- *An instantiation $\tau_S$ is globally consistent if it is locally consistent and can be extended to a solution.*

An instantiation is also called a tuple. For a tuple $\tau_S$, a variable $x \in S$ and a subset $W \subset S$, $\tau[x]$ and $\tau[W]$ denote respectively the value of $x$ in $\tau_S$ and the projection of $\tau_S$ on $W$ which is the set of values assigned by $\tau_S$ for the variables of $W$. For a constraint $c_S$ and a tuple $\tau_S$, we denote by $\tau_S \in c_S$ the fact that $\tau_S$ is an allowed instantiation of $c_S$ and $\tau_S \notin c_S$ the fact that $\tau_S$ is forbidden. When the variable $x$ is assigned to a value $a \in D(x)$, this is denoted by $(x, a)$ or $x_a$. For a given CSP, the notations $n, d, e$ respectively denote the number of variables, the maximum domain size and the number of constraints of the CSP.

The task of a CSP is to find a complete instantiation of values which satisfies all the constraints of the problem. If there does not exists any such solution, the problem is unsatisfiable, inconsistent or unfeasible. Otherwise, the problem is satisfiable, consistent or feasible. Deciding consistency is a NP-complete problem.

There are three main approaches for solving constraint satisfaction problems: backtracking search [van Beek, 2006], local search [Hoos and Tsang, 2006], and dynamic programming [Dechter, 2006]. Backtracking search as well as the technique "constraint propagation" for improving its efficiency will be recalled in this thesis.

### 2.1.2 Backtracking search

Backtracking search [Golomb and Baumert, 1965] is the simplest method for solving CSP problems that was first proposed by. It is a modified depth-first search of a tree. The idea is to search in a tree of variable assignments in which each intermediate node is a partial variable assignment and each leaf is a complete variable assignment. As we move down in the tree, we may assign (or restrict the domain of) a variable and a new node is created. Then, the partial variable assignment of that node will be checked for local consistency to determine whether the node can be extended to a solution or not. If the node cannot lead to a solution, it is a dead-end and the search backtracks to the parent node. Otherwise, an unassigned variable will be chosen to be assigned or restricted and the search will continue at the deeper level. A leaf satisfying all the constraints is a solution of the problem.

In the literature, there exists many backtracking search algorithms that are distinguished by the way constraints at a node are checked and by the way the search backtracks. For example, the naive backtracking search only checks constraints with no unassigned variable at a node while the forward checking algorithm [McGregor, 1979; Haralick and Elliot, 1980] only checks constraints with one assigned variable and one unassigned variable. In the naive backtracking search, the root of the search tree is an empty variable assignment. The partial assignment on the path to each node is checked to determine whether it is locally consistent or not. If a constraint check fails, the search stops its depth moves and the next value in the domain will be assigned to the current variable. If no value is left, the search will backtrack up to the most recently assigned variable.

### 2.1.3 Constraint propagation

Constraint propagation is a technique for improving the efficiency of backtracking search. It focuses on search space reduction by early elimination of locally inconsistent instantiations.

The idea is to maintain a necessary property, called a local consistency, that values or instantiations of values need to satisfy to belong to solutions. It requires for each scope $S$ of a given size that each locally consistent instantiation $\tau_S$ can be extended to an instantiation $\tau_{S'}$ on a larger scope: $S \subset S'$ and $\tau_{S'}[S] = \tau_S$. The simplest consistency for CSPs is node consistency which requires that each domain value must satisfy all unary constraints defined for the variable.

**Definition 2.4 (Node consistency)** *A variable $x$ is node consistent iff for every value $v \in D(x)$, for every unary constraint $c_x$, $v \in c_x$. A CSP is node consistent if every of its variables is node consistent.*

For a local consistency, a value (or instantiation) is said to be locally consistent if it satisfies the given local consistency and is called locally inconsistent otherwise. The locally inconsistent values (instantiations) cannot belong to any solution of the problem. They can therefore be removed from the problem without removing solutions. A problem is called locally consistent w.r.t a local consistency property if all its values (or instantiations) are locally consistent. Local consistencies are grouped in two classes: "domain-based" for the ones which define conditions on values and "constraint-based" for those which define conditions on instantiations of arity higher or equal to 2.

Every local consistency can be enforced by a transformation, called constraint propagation or filtering, which iteratively removes locally inconsistent values (instantiations) until no such value (instantiation) exists. This transformation changes the problem while preserving its equivalence in terms of the set of solutions. The strength of constraint propagation comes from the fact that the removal of locally inconsistent values (instantiations) can make other values (instantiations) no longer able to locally satisfy the related constraints, and thus also locally inconsistent.

The final result of constraint propagation for a consistency $\Phi$ on a CSP $P$ is called the $\Phi-$closure of $P$, denoted as $\Phi(P)$. The $\Phi-$closure of $P$ is a problem which satisfies the property $\Phi$ and which is equivalent to $P$ (having the same set of solutions). The $\Phi-$closure is unique for a given $P$ if $\Phi$ is a domain-based property stable under the union operation in the sense that the union of two problems $\Phi-$consistent defined on the same set of variables and constraints is also $\Phi-$consistent: $(X, D_1, C) \cup (X, D_2, C) = (X, D_1 \cup D_2, C)$ [Bessiere, 2006].

### 2.1.4 Arc consistency

Every CSP can be viewed as a network of constraints, where each variable is represented by a node and each binary (non binary) constraint is represented by an arc/edge (hyper arc/edge). Arc consistency defines the consistency of arcs in such a way that for a constraint and an involved variable, every value of the variable can be extensible to a consistent instantiation of the variables of the (hyper)-arc.

**Definition 2.5 (Arc consistency)** *A constraint $c_S$ is arc consistent iff for every variable $x \in S$ and every value $v \in D(x)$, there exists a tuple $\tau_S$ such that $\tau_S[x] = v$ and $\tau_S \in c_S$. Such a tuple is called a support for the value $(x, v)$ on $c$. A CSP is arc consistent iff all its constraints are arc consistent.*

**Example 2.1** *Consider the binary CSP in Figure 2.1(a). It has 3 variables $x_1, x_2, x_3$ with 3 values $1, 2, 3$ in each domain, and 2 constraints $c_{12}, c_{23}$ respectively defined as $(x_1 =$*

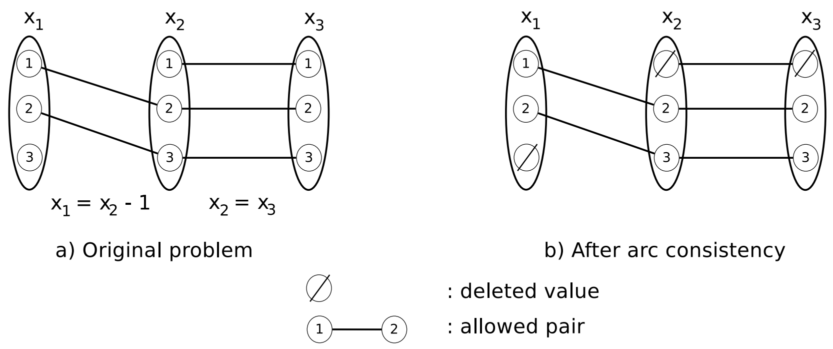


*Figure 2.1: Enforcing arc consistency*

*$x_2 - 1), (x_2 = x_3)$. Each value is represented by a vertex. An arc between two values means that the corresponding pair of values are authorized. Checking constraint $c_{12}$ detects the arc inconsistent value 3 in $D(x_1)$ because there is no value in $D(x_2)$ compatible with it. In other words, there is no support in $D(x_2)$ for $(x_1, 3)$ and this value will be removed. Value $(x_2, 1)$ also can be removed because of $c_{12}$. The removal of value $(x_2, 1)$ makes value $(x_3, 1)$ arc inconsistent and $(x_3, 1)$ will be removed. Now, every remaining value has a support on every constraint, and the resulting problem is arc consistent (Figure 2.1(b)).*

Arc consistency is also called generalized arc consistency (GAC) in many documents when the authors want to emphasise the fact that AC is used on non-binary CSPs. In this case, AC is specialized for binary CSPs.

An arc consistent CSP is not necessarily satisfiable. The coloring problem in Figure 2.2 is an example. It has 3 variables with 2 colors for each one and 3 constraints $c_{12}(x_1 \neq x_2), c_{23}(x_2 \neq x_3), c_{13}(x_1 \neq x_3)$. This problem is arc consistent but it is unsatisfiable since it is impossible to color a 3-clique with 2 colors only.

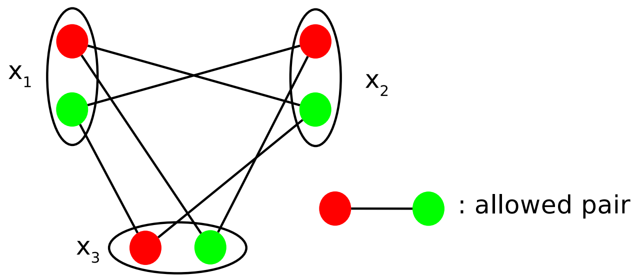


*Figure 2.2: Example of an unsatisfiable CSP which is AC*

### 2.1.5 Arc consistency algorithms

Many algorithms have been proposed for enforcing AC in order to improve the efficiency of enforcing. Bessiere gives two important reasons for studying AC algorithms in [Bessiere, 2006]: AC is the basic mechanism used in all solvers and each technique to improve AC can be applied for other consistency algorithms. AC enforcing algorithms are grouped in 2 classes: coarse-grained and fine-grained, where the former ones propagate the changes in the variable domains while the second ones propagate the value removals.

### AC3

AC3 is one of the first efficient algorithms for enforcing arc consistency, proposed by Mackworth [1977]. As a coarse-grained algorithm, it propagates changes in the domain of variables to neighbor variables by using a propagation list that contains arcs, variables, or constraints related to the reduced domains.

There are three variants for AC3 implementation: arc-oriented [Mackworth, 1977], variable-oriented [McGregor, 1979] and constraint-oriented [Boussemart et al., 2004a] In the first variant, the propagation list contains arcs where each arc is composed of a constraint and a variable involved in the constraint that needs to be revised. The second variant stores in the propagation list the variables with reduced domains and the last one stores the constraints involving at least a variable to be revised.

AC3 does not store any additional information, thus it can make many unnecessary constraint checks. Precisely, when revising a variable in a constraint, it has to check all values of the variable regardless of whether these values have lost their support or not. In binary CSPs, AC3 has a complexity in $O(ed^3)$ in time and $O(e)$ in space where $e$ is the number of constraints and $d$ is the maximum domain size.

### AC4

In order to improve AC3, Mohr and Henderson [1986] proposed the AC4 algorithm that memorizes a maximum amount of information. It maintains the number of supports for each value on each constraint in a counter. Whenever the counter of a value decreases to 0, the value will be removed and the counters of all its neighbor values will be decreased by 1. Thanks to the counter system, no constraint check is performed during the run of AC4 algorithm. Constraints are only checked once at start-up. Furthermore, AC4 stores the list of values supported by each value. When a value is removed, only values that have been supported by the removed value need to be verified. This list allows AC4 to avoid considering unnecessary values which have not been supported by the removed values. Compared to AC3, AC4 has a better time complexity in $O(ed^2)$ but needs more space in $O(ed^2)$.

### AC6

AC4 spends a lot of time to compute and update the “counters” as well as the lists of supported values whereas the search for all supports is not necessary for consistent values. Therefore, Bessière [1994] proposed AC6 in order to avoid such unnecessary computation of AC4. Instead of keeping the complete support sets and counters, AC6 remembers only one support for each value. AC6 cancels the “counter” information and stores for each value the list of values that are currently supported by it. When a value is removed, AC6 will search for a new support for the values that were supported by the removed value. Thanks to the order of values in domains, the search for a new support always starts from the current support to the end of the domain, until the next support is found. All the values before the current support of the considered value were verified as incompatible with it. Thus, AC6 can avoid redoing constraint checks despite of the unknown number of supports. AC6 has the same time complexity in $O(ed^2)$ as AC4 but it reduces the space complexity to $O(ed)$.

#### AC2001 & AC3.1

Both AC4 and AC6 are fine-grained algorithms. The disadvantage of these algorithms is that the value-oriented propagation queue is expensive to maintain. Therefore, Bessière and Régin [2001] proposed a coarse-grain algorithm AC-2001 which keeps the optimal time complexity of AC6, by memorizing only the current support for each value in order to avoid redundant constraint checks. This idea is also used in [Zhang, 2001] in an algorithm named AC3.1. AC2001 uses a pointer to store the first support for every value on each constraint. On the one hand, this data structure is easier to implement and maintain than the lists of supported values used in AC6. On the other hand, similarly to the lists of supported values used in AC6, it allows AC2001 to stop the search for supports as soon as possible. The search for a new support for a value on a constraint does not check again values before the current support which were previously proved as incompatible with the considered value. In spite of the fact that AC2011 has the same asymptotic time and space complexity as AC6, it can provide speed-ups in practical experiments because of its simplicity.

### 2.1.6 Restricted arc consistencies

In order to reduce the computational cost of arc consistency when being maintained during the backtracking search, many domain-based consistencies which are weaker than AC in terms of pruning power but have a cheaper computational cost have been proposed. The usual idea of these consistencies is to weaken AC by reducing the work AC does, i.e., reducing either the number of calls to constraint checks or the amount of work inside each constraint check. The former checks AC for only some of constraints while the last checks AC for some of domain values (e.g.; the minimum and the maximum values of domains in the case of consistencies on bound). In summary, these restricted consistencies of AC enforce arc consistency in an incomplete way by skipping some values or some constraints.

#### Directional arc consistency

Directional arc consistency (DAC [Dechter and Pearl, 1988]) tries to reduce the number of constraint checks by defining an order "<" among variables and enforcing AC only on arcs directed along this order. A variable $i$ is directed arc consistent iff it is arc consistent for all constraints $c_{ij}$ such that $i < j$. A variable must be consistent with all variables greater than it, regardless of smaller ones. Thus, each variable needs to be checked for arc consistency with respect to greater variables and the removal of a value from the domain of a variable cannot make greater variables directed arc inconsistent. Thanks to this property, DAC does not need to use a propagation queue, just a loop processing variables w.r.t the DAC order from the greatest to the smallest one.

### 2.1.7 Strong consistencies

#### Path consistency

Path consistency, proposed by Montanari [1974], is the most studied constraint-based local consistency. In binary CSPs, path consistency is simply an extension of arc consistency which consists in extending the pairs of variables, instead of singleton variables, on every other variable. A pair of variables is path-consistent with respect to a third variable iff for every consistent pair of values, there exists a value of the third variable compatible with this pair in such a way that the 3-values tuple satisfies all the binary constraints.

**Definition 2.6 (Path consistency (PC))** *A CSP is path consistent iff for every pair of variables $(x_i, x_j)$, for every consistent pair of values $(v_i, v_j) \in D(x_i) \times D(x_j)$, for every third variable $x_k$ connected with $x_i, x_j$ by $c_{ik}, c_{jk}$, there exists a value $v_k \in D(x_k)$ such that $(v_i, v_k) \in c_{ik}$ and $(v_j, v_k) \in c_{jk}$.*

Many algorithms have been proposed to enforce path consistency, that differ each other by the data structures and by the enforcing efficiency. The first one, PC−1, is a naive algorithm that has time complexity in $O(n^5d^5)$ [Montanari, 1974]. PC−2 [Mackworth, 1977] and PC−3 [Mohr and Henderson, 1986] respectively improve this complexity to $O(n^3d^5)$ and $O(n^3d^3)$ by using the idea of AC3 and AC4 to reduce the number a triple of variables is checked. However, Han and Lee [1988] proved that PC−3 is not correct. When a pair of values has no compatible value on a third variable, PC−3 remove the two values of the pair instead of forbidding the pair. Thus, they proposed a correct version, PC−4, that has a same complexity $O(n^3d^3)$ in time and space. Then, PC−5 [Singh, 1995] and PC−6 [Chmeiss, 1996] were independently proposed to improve the average time complexity of PC−4 to $O(n^3d^2)$ by basing on AC6 instead of AC4. PC−7 [Assef Chmeiss, 1996] has a smaller time complexity in $O(n^2d^2)$.

### $k$−consistency

Based on the idea of node and arc consistencies which respectively consist in consistently extending zero and one variable on every another one, [Freuder, 1978] introduced $k$−consistency for consistently extending $k-1$ variables to every extra one. It guarantees that for each consistent instantiation of $k-1$ variables, there exists a value for every $k$−th variable such that the $k$−values instantiation satisfies all constraints among them.

**Definition 2.7 (k-consistency)** *A CSP is $k$-consistent iff for every set of $k-1$ variables $Y$, for every consistent instantiation $\tau_Y$, for every $k$-th variable $x_k$, there exists a value $v_k \in D(x_k)$ such that $\tau_Y \cup (x_k, v_k)$ is consistent. The CSP is strongly $k$-consistent iff it is $j$-consistent for every $j \leq k$.*

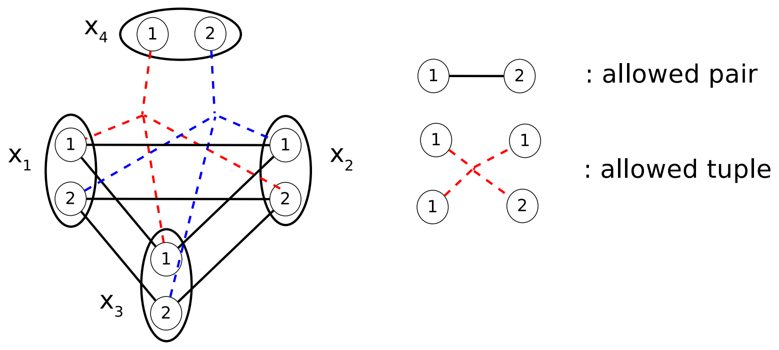


*Figure 2.3: A 4-inconsistent CSP*

**Example 2.2** *Consider the CSP in Figure 2.3. It has 4 variables with 2 values for each one, 3 binary constraints $(x_1 = x_2), (x_2 = x_3), (x_1 = x_3)$ and a 4-ary constraint*

$c_{1234}(x_1, x_2, x_3, x_4) = \{(1, 2, 1, 1), (2, 1, 2, 2\}$. *Allowed pairs of values for binary constraints are represented by continuous black lines, while allowed tuples for the four-ary constraint are represented by dashed lines. Each allowed tuple uses a different color. This problem is 3-consistency but is not 4-consistency because the locally consistent instantiations* $(1, 1, 1)$ *and* $(2, 2, 2, )$ *on* $(x_1, x_2, x_3)$ *cannot be consistently extended to* $x_4$.

For normalized binary CSPs (where there is are two constraints with the same scope), NC, AC and PC respectively correspond to 1,2,3-consistency. If we enforce all NC, AC, PC, we will obtain the level of strong 3-consistency. In non-binary CSPs, PC is not equivalent to 3-consistency because 3-consistency considers ternary constraints whereas PC does not. In non-normalized CSPs, AC is not equivalent to 2-consistency because there exists cases in which 2 variables are connected by more than 2 binary constraints, each one is AC but no pair of values satisfies all constraints.

### Restricted path consistency

Restricted path consistency (RPC), proposed by Berlandier [1995], is half-way between AC and PC. It removes more inconsistent values than AC while avoiding some disadvantages of PC. PC checks the consistency of all pairs of values, even those of two independent variables, on any third variable. This is very expensive and can create new constraints. Thus, RPC checks only pairs of values which, if they are removed, will make a value arc inconsistent. So in addition to AC, RPC checks path consistency for pairs of values which define the *unique support* for an involved value. If such a unique support is inconsistent, it could be removed and this potential removal leads to the removal of the previously supported value.

**Definition 2.8 (Restricted path consistency - RPC)** *A binary CSP is restricted path consistent iff it is AC and for all* $x_i$*, for all value* $v_i \in D(x_i)$*, for all* $c_{ij}$ *on which* $v_i$ *has a unique support* $v_j \in D(x_j)$*, for all* $x_k$ *linked to both* $x_i, x_j$ *by binary constraints* $c_{ik}, c_{jk}$*, there exists a value* $v_k \in D(x_k)$ *such that* $(v_i, v_k) \in c_{ik}$ *and* $(v_j, v_k) \in c_{jk}$*.* $v_k$ *is called a witness for the support* $(i_a, j_b)$ *of* $(i, a)$ *on* $k$*.*

**Example 2.3** *Let's consider the problem in Figure 2.4(a). Please notice that here, edges indicate* forbidden *pairs. It is AC but not RPC. Value* $(x_i, 1)$ *has only one support* $(1, 2)$ *in* $c_{ij}$ *but this support cannot be extended on* $x_k$*. It will be removed by RPC.*

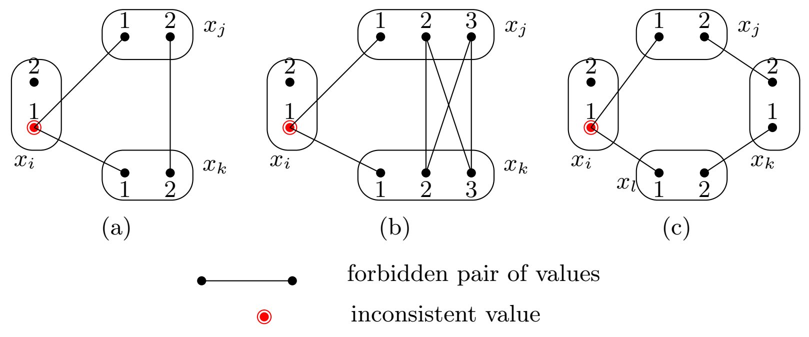


*Figure 2.4: Some examples for comparing AC, RPC, PIC, maxRPC [Bessiere, 2006]. a)A CSP which is AC but is not RPC. b) A CSP which is RPC but is not PIC. c) A CSP which is PIC but is not maxRPC.*

RPC can prune more values than AC. In addition to the arc inconsistent values, RPC also removes the values for which their unique support is path inconsistent in some constraint.

Several algorithms have been proposed for enforcing RPC. Algorithm RPC1, proposed by Berlandier [1995], is based on AC4. It also counts the number of supports for each value in each constraint, and stores the list of values supported by each value. A value needs to be checked for PC whenever its number of supports decreases to 1, and is removed whenever this number decreases to 0. RPC1 uses 2 propagation queues. The first one contains removed values, used for the AC propagation, as in AC4. The second one, used for the PC propagation, contains 3-tuples (value, constraint, triangle) for which the unique support of the value needs to be checked for PC on the triangle. RPC1 has a space complexity in $O(end)$ and a time complexity in $O(ed^2)$ on binary CSPs.

Then, Debruyne and Bessière [1997a] proposed RPC2 which is based on the idea of AC6. RPC2 only stores the two first supports for each value in each constraint, instead of counting all supports and storing the lists of supported values as in RPC1. Whenever a value has lost supports, RPC2 will try to search next supports such that the total number of supports, including the available stored ones, does not exceed 2. RPC2 will remove a value, revise for PC or do nothing if this value has no, one or at least 2 supports respectively. Moreover, RPC2 stores current witnesses for each value on each constraint in each triangle while RPC1 does not. Thus, when checking PC, RPC2 does nothing if the current witnesses for unique supports are still available. The search for next support as well as witness is efficient because it always starts from the previous last found one. RPC2 has a time complexity in $O(ed^2 + cd^2)$ and space complexity in $O(ed + cd)$ where $c$ is the number of triangles (clique of three binary constraints) in the problem.

### Path inverse consistency

Freuder [1985] introduced $k-$inverse consistency in order to provide a domain-based consistency stronger than AC. The idea of $k$-inverse consistency is to extend each variable to every $k-1$ extra variables. In fact, 2-inverse consistency (arc inverse consistency) is the same as 2-consistency (arc consistency). 3-inverse consistency (also called path inverse consistency) is the simplest inverse consistency that has a pruning power stronger than AC.

**Definition 2.9 (Path inverse consistency - PIC)** *A binary CSP is path inverse consistent iff it is AC and for all $x_i$, for all value $v_i \in D(x_i)$, for all $x_j, x_k$ linked to each other by $c_{jk}$ and linked both to $x_i$ by $c_{ij}, c_{ik}$, there exists a value $v_j \in D(x_j)$, a value $v_k \in D(x_k)$ such that $(v_i, v_j) \in c_{ij}$, $(v_i, v_k) \in c_{ik}$ and $(v_j, v_k) \in c_{jk}$.*

PIC can prune more locally inconsistent values than RPC. The problem in Figure 2.4(b) is an example: it is RPC but not PIC. Consider value $(x_i, 1)$. It is of course RPC because it has 2 supports in every constraint $c_{ij}, c_{ik}$. However, none of its support in $c_{ij}$, $c_{ik}$ can be extended on $x_k$, $x_j$ respectively. As a result, $(x_i, 1)$ cannot be extended on triple $(x_i, x_j, x_k)$. Thus, this value is not PIC and will be removed by PIC.

**PIC algorithms** The first algorithm for enforcing PIC, called PIC1, was proposed by Freuder and Elfe [1996]. PIC1 does not use any special technique or data structure. It only uses a propagation queue containing variables which need to be checked for PIC.

When revising a variable popped up from the propagation queue, it considers every triangle involving that variable and deletes values which are not extensible on that triangle. If the domain of the revised variable has been reduced, all neighbor variables are pushed into the propagation queue. The algorithm stops when this queue is empty. The time and space complexity of PIC1 are $O(en^2d^4)$ and $O(n)$ respectively.

R.Debruyne [2000] proposed an enhanced algorithm for enforcing PIC, named PIC2, which stores additional information to avoid useless constraint checks. Firstly, PIC2 stores the list of values and the list of pairs of values currently supported by each value. When a value is removed, PIC2 only considers values and pairs of values in the lists of the removed value to check for AC and PC respectively. If no witness is found for a pair of values, PIC2 tries to search for a new PC support for the involved value by checking next pairs of values. The search for a new support and for a new witness always starts from the old one. Secondly, PIC2 uses a lexicographic order to arrange values in domains and to define an order among pairs of values. Based on AC7 [Bessiere et al., 1999], PIC2 stores the last value checked to find an AC support for each value in each constraint. Thanks to this data, each pair of values and each 3-values tuple are checked at most once for the search for supports and for witnesses respectively. The propagation queue used in PIC2 contains removed values while PIC1 store variables which needs to be checked for PC in this queue. The time and space complexity of PIC2 are $O(en + ed^2 + cd^3)$ and $O(ed + cd)$ respectively.

### max-restricted path consistency

Proposed by Debruyne and Bessière [1997a], maxRPC is an extension of RPC. It checks the existence of a path consistent support for each value in each constraint, whatever the number of supports the value may have. maxRPC is an intermediate between RPC and PC. RPC and PC respectively guarantee path consistency for the unique support and for every support while maxRPC guarantees path consistency for one support of each value in each constraint. The idea of maxRPC is to remove all values which have no AC or PC support in some constraint, where a PC support is an AC support which is path consistent. For binary CSPs, a value is max-restricted path consistent if it has a PC support in every constraint.

**Definition 2.10 (max-restricted path consistency)** *A binary constraint network is max-restricted path consistent iff it is AC and for all $x_i$, for all value $v_i \in D(x_i)$, for all $c_{ij}, v_i$ has a support $v_j \in D(x_j)$ such that for all $x_k$ linked to both $x_i, x_j$ by a constraint, there exists a value $v_k \in D(x_k)$ such that $(v_i, v_k) \in c_{ik}$ and $(v_j, v_k) \in c_{jk}$.*

In the case of non-binary CSPs, maxRPC requires AC for non-binary constraints and maxRPC as defined above for binary constraints.

maxRPC prunes more values than PIC. The CSP in Figure 2.4(c) is PIC but is not maxRPC due to the value $(x_i, 1)$. It has 2 AC supports on $c_{ik}$. The support $(1,1)$ on $(x_i, x_k)$ can be consistently extended on $x_j$ but cannot on $x_l$. Conversely, the support $(1,2)$ on $(x_i, x_k)$ can be consistently extended on $x_l$ but cannot on $x_j$. In summary, $(x_i, 1)$ has no AC support that can be consistently extended on $x_j$ and $x_l$ at the same time.

**maxRPC algorithms** The first algorithm for enforcing maxRPC is a fine-grained one, named maxRPC1, which was proposed by Debruyne and Bessière [1997a] for binary CSPs.

It is built based on RPC2 by using the same data structures and memorizing the same information as in RPC2. maxRPC1 also stores the list of values supported by each value and the list of pairs of values witnessed by each value. The difference between RPC2 and maxRPC1 is that maxRPC1 memorizes the first PC support instead of the 2 first AC supports for each value in each constraint. When a value is removed, maxRPC1 will search for a new PC support for all values currently supported by it and search for a new witness for all pairs of values currently witnessed by it. If no witness is found for a pair of values, maxRPC1 will search for a new PC support by considering next pairs of values. maxRPC1 has a space and time complexity in $O(ed + cd)$ and $O(en + ed^2 + cd^3)$ respectively. This time complexity is known as being optimal for enforcing maxRPC.

maxRpc2 [F. and G., 2003] is a coarse-grained maxRPC algorithm which has the same complexity as maxRPC1 but takes a smaller space complexity in $O(ed)$. Based on AC2001, maxRPC2 uses a pointer to store the last PC support for each value in each constraint. However, maxRPC2 does not store witnesses of PC supports. Thus, maxRPC2 combines the procedure for checking PC witness loss into the procedure for checking PC support loss. Each time a variable with reduced domain is popped up from the propagation queue, maxRPC2 will verify whether neighboring values have lost their current PC supports or not, stored in the "pointer" data structure. If a value has lost its PC supports, maxRPC2 will consider values after the last PC support to find a new one. Conversely, if the PC support of a value is still available, maxRPC2 will check whether this support has lost witnesses on some third variable or not. If yes, maxRPC2 will search for a new witness for this support by trying all the values in the domain of that third variable. As usual, if no witness is found, a new PC support will be searched for the value.

Vion and Debruyne [2000] proposed a coarse-grained maxRPC algorithm called maxRPC$^{rm}$. It uses a propagation queue which contains variables having reduced domains rather than containing removed values as in maxRPC1. Based on AC3$^{rm}$ [Lecoutre et al., 2007] and residues [Likitvivatanavong et al., 2004] maxRPC$^{rm}$ stores the current first PC support and PC witness for each value on in constraint and each triangle. When a reduced variable $j$ is popped up from the propagation queue, maxRPC$^{rm}$ will search for a new PC support for neighboring values $(i, a)$ whose current support has been removed. At the same time, maxRPC$^{rm}$ will search for a new witness for every current support $((i, a), (k, c))$ of $(i, a)$ whose current witness on $j$ has been removed. If no witness is found for such supports, a new PC support will be searched.

Moreover, Vion and Debruyne also proposed 2 relaxed versions of maxRPC$^{rm}$ which enforce 2 approximate levels of maxRPC [Vion and Debruyne, 2000]. The first one is "One pass maxRPC", denoted as O-maxRPC$^{rm}$, which only searches for PC supports and witnesses once for each value in each constraint and each triangle at the initialization step. If variable $x$ is processed before its neighboring variable $y$ in the initialization and some inconsistent values of $y$ are removed when revising maxRPC, O-maxRPC$^{rm}$ does not check for PC support loss or witness loss for values of $x$. In summary, O-maxRPC$^{rm}$ skips the propagation step: the removal of values or the change in domains are not propagated.

The second relaxed algorithm of maxRPC$^{rm}$, "Light maxRPC" or L-maxRPC$^{rm}$, enforces an approximation of maxRPC stronger than O-maxRPC$^{rm}$. It does the same initialization work as O-maxRPC$^{rm}$ while keeping a propagation step. However, the propagation is done in an incomplete way. L-maxRPC$^{rm}$ only searches for new supports in the case of PC-support loss but it does not maintain witnesses for PC supports. In the case of PC witness loss, L-maxRPC$^{rm}$ does nothing. The witness data is only defined once in the initialization step. Indeed, because of the high time complexity of maxRPC, enforcing the approximations of maxRPC by O-maxRPC$^{rm}$ and L-maxRPC$^{rm}$, can be an alternative approach for maintaining maxRPC in search with time complexity in $O(eg + ed^2 + cd^3)$.

Balafoutis et al. [2011] proposed another coarse-grained maxRPC algorithm, named maxRPC3, which is also based on AC3$^{rm}$ and maxRPC$^{rm}$. Differently from maxRPC$^{rm}$, maxRPC3 separates AC supports from PC supports. For each value and each constraint, it stores 2 supports: the smallest AC and the smallest PC supports. Thus, when there is a domain reduction, maxRPC3 has to check for AC support, PC support and PC witness loss for neighboring values. The specificity of maxRPC3 is the use of a heuristic for choosing values when searching PC support and PC witness. To search for a PC support for a value $(i, a)$ in a constraint $c_{ij}$, maxRPC3 always checks first the AC support of $(i, a)$ in $c_{ij}$. If this AC support is not path consistent, it will check other values which are ordered after both the AC and PC support of $(i, a)$ in $c_{ij}$. Similarly, to search for a PC witness for a support $((i, a), (j, b)$ of $(i, a)$ on $k$, maxRPC3 always checks first the AC support of $(i, a)$ in $c_{ik}$ and then that of $(j, b)$ in $c_{jk}$. If none is compatible with the pair of values, maxRPC3 will check other values of $k$. During the search for supports and witnesses, the AC supports are also updated if necessary in such a way that they are always the first/smallest available AC supports for the checked value. This heuristic helps maxRPC3 to avoid traversing values in domains when searching for supports and witnesses, and stop the constraint checks as soon as possible. The number of cases where an AC support is also a PC support is trivially determined in practice. Thus, this heuristic helps maxRPC3 to have a good experimental time complexity for solving many CSPs.

In [Balafoutis et al., 2011], Thanasis et al. also proposed a variant of maxRPC3, named maxRPC3$^{rm}$. In maxRPC3$^{rm}$, the stored AC supports, PC supports and PC witnesses are not guaranteed to be the first available ones. Thus, the search for a new AC support, for a new PC support as well as for a new PC witnesses always have to restart from scratch by considering values from the beginning of domains. During the search for supports and for witnesses, if an AC support is found for a value in a constraint, maxRPC3$^{rm}$ will immediately update the AC support for this value regardless of whether it is the smallest AC support or not.

[Balafoutis et al., 2011] also proposed two light versions of maxRPC3 and maxRPC3$^{rm}$, denoted by lmaxRPC3 and lmaxRPC3$^{rm}$ respectively, which enforce an approximation of maxRPC. Similarly to L-maxRPC$^{rm}$, lmaxRPC3 and lmaxRPC3$^{rm}$ do not maintain witnesses for PC supports. PC witnesses are only searched once in the initialization. The domain reduction only activates the search for new supports, but does not activate the search for new witnesses.

### 2.1.8 Comparison between consistencies

In order to compare the pruning efficiency of the domain-based consistencies, Debruyne and Bessière [1997b] proposed a transitive relation, the "stronger than" relation. A local consistency $A$ is said to be stronger than $B$ if, in any CSP in which $A$ holds, $B$ holds too. This means that $A$ deletes at least all the inconsistent values removed by $B$. $A$ is strictly stronger than $B$ if $A$ is stronger than $B$ and there exists a CSP in which $B$ holds but $A$ does not. In other words, there is at least a CSP for which $A$ deletes more inconsistent values than $B$.

[Bessiere, 2006] gives an overview on the comparison among some domain-based consistencies. It shows that "maxRPC $\longrightarrow$ PIC $\longrightarrow$ RPC $\longrightarrow$ AC", where the arrows go from the stronger consistencies to the weaker ones.

The relation among the maxRPC, PIC, RPC and AC consistencies is also in the following paper [Debruyne and Bessiere, 2001]: maxRPC is stronger than PIC and both are stronger than RPC. If a value $(x_i, a)$ has no support in $c_{ij}$, the three consistencies will remove it. If it has only a support in $c_{ij}$, the consistencies are identical: they will remove $(x_i, a)$ if the unique support is path inconsistent. If $(x_i, a)$ has more than two supports in $c_{ij}$, RPC holds of course for this value while PIC will remove it if it has no support in $c_{ij}$ extensible on some third variable. In other words, PIC does not hold for $(x_i, a)$ if all its supports in $c_{ij}$ are path inconsistent because of the same third variable. In the case that for each third variable, there exists a support in $c_{ij}$ extensible to that variable but this support is path inconsistent because of a different third variable, $(x, a)$ is PIC but it will be removed by maxRPC.

### 2.1.9 Revision ordering heuristics

Revision ordering heuristics are techniques for improving the efficiency of constraint propagation by appropriately ordering the arcs, variables, or constraints in the revision list in such a way that (1) the inconsistent parts of the problem are pruned early and (2) the constraint propagation converges to the locally consistent closure of the problem after a small number of constraint checks. To do this, heuristics are based on the features concerning the structure of problems such as: the current variable domain size (dom), the proportion of removed values in variable domains, the satisfiability ratio of constraints (sat) defined by the fraction of acceptable pairs of values in the constraints, the variable degree (deg) defined by the number of constraints involving the variable in the initial problem, the current variable degree (ddeg)... The most satisfied elements in the revision list are first chosen to be checked. The simplest and the most naive heuristics are "lexico" where variables are ordered lexicographically and "fifo" where the revision list is implemented as a queue.

Wallace and E.C.Freuder [1992] proposed the first heuristics used for the arc-oriented variant of AC3. These heuristics select first an arc in the revision list with:

- dom: the smallest current variable domain size of the variable involved in the arc. It is argued that a small domain is more potential to become wiped-out than a large one.
- sat: the smallest constraint satisfiability. If the constraint contains a number of acceptable pairs of values smaller than the domain size of the considered variable, the variable has at least one value unsupported in the constraint.
- rel sat: the smallest constraint rational satisfiability, that is the constraint satisfiability divided by the domain size. This heuristic is based on the mean number of supports per value in the constraint. If this mean number is smaller than 1, the considered variable has at least one value unsupported in the constraint.
- deg: the greatest variable degree. A variable related to more constraints is more likely to have values unsupported in one of its constraints.

Boussemart and al. [Boussemart et al., 2004a] adapted these heuristics for the constraint-oriented and variable-oriented variants of AC3, and proposed new ones.

- For the variable-oriented variant, they have proposed the heuristic named $\text{rem}^v$ which selects first the variable having the greatest proportion of removed values in its domain.
- Concerning the arc-oriented variant, they proposed
  - $\text{dom}^c$: the smallest constraint size defined by the Cartesian product of the current domain of the variables involved in the constraint.
  - $\text{rem}^c$: the greatest proportion of the removed pairs of values. This is the complementary of the constraint size.
  - $\text{dom}^c/\text{dom}^v$: the smallest constraint satisfiability, similarly to the satisfiability in [Wallace and E.C.Freuder, 1992].
  - $\text{dom}^c \circ \text{dom}^v$: the smallest current domain size and then the greatest current degree in the case of equivalence.

Boussemart et al. [2004a] shows that "dom" is the most efficient heuristic for every variant of AC3 (in the sense of the number of constraint checks as well as the number of additions to the revision list) but is not always the fastest. The naive heuristic "fifo" is in general the fastest for arc and constraint-oriented algorithms.

Balafoutis and Stergiou [2008a,b] proposed a new heuristics exploiting the conflict-driven weighted degree heuristics (wdeg, dom/wdeg) used for variable selection in search [Boussemart et al., 2004b]. During search, these conflict-driven heuristics maintain the number of times each constraint causes a domain wipe-out during constraint propagation. This number is associated to the constraint. The weighted degree of a variable is then the sum of the weights of the constraints involving it and at least one other unassigned variable. The heuristics wdeg and dom/wdeg respectively select the variable having the greatest weight and the smallest ratio between the current domain size over the current weight. Used at the same time for constraint propagation and for search branching, the new heuristics allow us to reduce not only the number of constraint checks but also the explored search space.

### 2.1.10 Dynamic arc consistency

The traditional static CSPs are not sufficient for solving many real-life applications designed in dynamically changing environments whose set of constraints may evolve. To give only one example, in course scheduling, the teachers can incrementally propose changes in the number and the time of courses and each change requires to reformulate the problem. In order to solve this kind of problem, static CSPs have been extended to so-called dynamic CSPs [Dechter and Dechter, 1988] which facilitate the problem reformulation after additions (restrictions) or retractions (relaxations) of constraints from the problem. Unlike static CSP problems, a dynamic CSP is dynamically designed by adding and retracting constraints one by one in the problem.

**Definition 2.11** *A dynamic CSP is a sequence $P_0, \dots, P_i, P_{i+1}, \dots$ of CSPs where each $P_{i+1}$ is a CSP resulting from the addition or retraction of a constraint in $P_i$.*

Dynamic arc consistency algorithms (DnAC) aim at maintaining arc consistency in the sequence of problems $P_i$. AC enforcing is naturally incremental for restriction because adding new constraints means pruning newly arc inconsistent values: it is sufficient to use AC algorithms after adding a new constraint to the problem.

Conversely, AC is not naturally incremental for relaxation (constraint retraction). This last case needs therefore to be handled specifically: values that have been deleted directly or indirectly because of the retracted constraint need to be restored, but only if there is no other reason to delete them. The existing algorithms for filtering DCSPs cope with the retraction of constraints in three phases: restorations of the values that were directly deleted by the retracted constraint; propagation of value restorations to other domains; and removal of wrongly restored values. Historically, the first algorithm proposed for DnAC is DnAC-4 [Bessière, 1991] which relies on a fine-grain AC-4 algorithm.

In addition to the data structure proposed by AC-4, DnAC-4 uses an additional data structure as justification for value deletions during the AC domain pruning in order to improve the efficiency of constraint retraction by decreasing the number of wrongly restored values. However, DnAC-4 inherits the main disadvantages of AC-4 with a large space complexity in $O(ed^2)$. This was later improved in DnAC-6 [Debruyne, 1996] using the more space efficient AC6 algorithm rather than AC-4 for filtering. DnAC-6 handles the retraction of constraints in the same way as DnAC-4, using the same justification data. DnAC-6 is known, so far, as the fastest algorithm but it still has the disadvantage of fine-grained algorithms characterized by a large space complexity.

In order to keep low memory consumption, Berlandier and Neveu [1994] proposed AC|DC, a simplified approach relying on the AC3 algorithm and requiring essentially no persistent data-structure.

However, AC|DC does not store any information during pruning and restoring values and thus wrongly restores more inconsistent values than AC-4 and AC-6. It only uses two lists of values called "Propagable" and "Restorable". "Propagable" contains values candidate for restoration and "Restorable" contains the restored values that are waiting for being revised for AC. The procedure for adding constraints in AC|DC is simply the standard AC3. As any existing DnAC algorithm, upon retraction of the constraint $c_{ij}$, AC/DC goes through three stages:

- Initialization: the deleted values in the domains of $i$ and $j$ that have no support in $c_{ij}$ are candidate for restoration and marked as "Propagable".
- Propagation: each of the propagable values is propagated to neighbor variables to check if they offer a new valid support for deleted values (which also will be marked as "Propagable"). When a value has been propagated to all neighbor variables, it is marked as "Restorable" and will be restored.
- Filtering: all restored values need to be checked again for arc consistency. This can be done using plain AC enforcing provided the queue $Q_{AC}$ is initialized to enforce the revision of domains of variables with restored values.

The second stage guarantees that no arc consistent value is missed in the resulting problem while the last stage guarantees that every remaining restored value is arc consistent. This guarantees again an equivalent arc consistent problem in the final result.

AC|DC can be improved by (1) improving the efficiency of enforcing AC by replacing AC3 by AC2001 that results in a time and space complexity comparable to DnAC-6. (2) improving the efficiency of retracting constraints, like AC|DC-2 does [Surynek and Barták, 2004; Barták and Surynek, 2005], by introducing persistent data-structures in order to define more precisely the values to be restored.

AC|DC−2 uses an array justif_var$[i, a]$ for remembering the constraint that has been responsible for the deletion of $(i, a)$ and justif_time $[i, a]$[1] for remembering the moment that $(i, a)$ is deleted during pruning. For synchronizing the time, AC|DC−2 uses a global incremental counter gtime that increases whenever a value is deleted. The procedure for adding a constraint is simply the standard AC3 but does extra work, i.e. memorizes value deletions in justif_var, justif_time. Similarly to any existing DnAC algorithm, AC|DC−2 handles the constraint retraction in three stages as described in Algorithm 2.1.

- Initialization: performed by Procedure "initialize" at line 7. Only values which have been deleted because of the retracted constraint $c_{ij}$ (line 10, 11) are considered to be restored (line 12). This can be tested in the *justification* array. $D^0(i)$ denotes the initial domain of variable $i$. All restored values of variable $i$ are stored in restored_$i$ and time_$i$ memorizes the earliest time that these restored values are deleted.

- Propagation: performed by Procedure propagate-acdc2 at line 17. Domain extensions are propagated to the forward neighborhood. A variable $i$ having restored values can check neighboring values $(j, b)$ for restorability just if (1) they have been removed due to the loss of support on the constraint $c_{ji}$, known through *justification*, (line 24) (2) they were deleted before at least a restored value of $i$ (line 25). $(j, b)$ will be restored if there exists a support for it among restored values of $i$ (line 26). In comparison to AC/DC, AC/DC-2 restores fewer values and hence fewer values need to be checked in the filtering phase thanks to these criteria. When $i$ has been extended, restored values of $i$ need to be rechecked for AC and this will be done in the last stage by adding all constraints involving $i$ into $Q_{AC}$ - the propagation queue of value deletions.

- Filtering: this last stage is unchanged that is performed by the AC3 procedure propagate-ac3 at line 6 with an initial propagation queue $Q_{AC}$ established in the second phase.

AC|DC-2 keeps a low space complexity and at the same time has a good practical time complexity.

## 2.2 Weighted CSP and soft consistencies

In this section, we present an existing extension of the CSP framework which associates costs to tuples of constraints to express a violation degree of tuples and therefore preferences between solutions. The goal of these problems is to find a solution for which the combined cost is minimal. These kinds of problem are called Valued CSPs [Schiex et al., 1995] (and are related to semi-ring CSPs [Bistarelli et al., 1997]) and their constraints are called soft constraints or cost functions. In this document, we consider only problems with a totally ordered scale of costs, thus valued CSPs. Filtering techniques, soft consistencies and enforcing algorithms for solving this kind of problem will also be presented in this section.

[1]The authors of AC/DC−2 have acknowledged that justif_time is actually subsumed by the justif_var data-structure (Private communication).

**Algorithme 2.1 :** AC$\|DC-2$ algorithm

```
1  Procedure retract-constraint-acdc2($c_{ij}$)
2      ($i$, time_$i$, restored_$i$) ← initialize($i, j$);
3      ($j$, time_$j$, restored_$j$) ← initialize($j, i$);
4      $C \leftarrow C \backslash \{c_{ij}\}$;
5      $Q_{AC}$ ← propagate-acdc2 ({($i$, time_$i$, restored_$i$), ($j$, time_$j$, restored_$j$)};
6      propagate-ac3 ($Q_{AC}$);
7  Procedure initialize($i, j$)
8      restored_$i \leftarrow \varnothing$;
9      time_$i \leftarrow \infty$ ;
10     foreach $a \in D^0(i) \backslash D(i)$ do
11         if justif_var[$i, a$] = $j$ then
12             add $a$ into $D(i)$;
13             justif_var[$i, a$] ← nil;
14             restored_$i$ ← restored_$i \cup \{a\}$;
15             time_$i$ ← min{time_$i$, justif_time[$i, a$]} ;
16     return ($i$, time_$i$, restored_$i$);
17 Procedure propagate−ac|dc2(restore)
18     $Q_{AC} \leftarrow \varnothing$;
19     while restore ≠ ∅ do
20         ($i$, time_$i$, restored_$i$) ← restore.pop();
21         foreach $c_{ij} \in C$ do
22             restored_$j \leftarrow \varnothing$;
23             foreach $b \in D^0(j) \backslash D(j)$ do
24                 if justif_var[$j, b$] = $i$ then
25                     if justif_time[$j, b$] > time_$i$ then
26                         if $b$ has a support in restored_$i$ then
27                             add $b$ into $D(j)$;
28                             justif_var[$j, b$] ← nil;
29                             restored_$j$ ← restored_$j \cup \{b\}$;
30                             time_$j$ ← min{time$_j$, justif_time[$j, b$]};
31             restore ← restore ∪{($j$, time_$j$, restored_$j$};
32         $Q_{AC} \leftarrow Q_{AC} \cup \{e | e \in C, i \in e\}$;
33     rerun $Q_{AC}$;
```

### 2.2.1 Weighted constraint satisfaction problems

**Definition 2.12 (Weighted CSP [Schiex, 2000])** *A weighted CSP (WCSP) or cost function network (CFN) is defined by a tuple $(X, D, C, m)$ where $X$ is a set of n variables. Each variable $x \in X$ has a domain $D(x) \in D$. $C$ is a set of cost functions. Each cost function $c_S \in C$ defined over a set $S$ of variables assigns costs to assignments of variables in $S$ i.e. $c_S : \ell(S) \to [0..m]$ where $m \in \{1, \dots, +\infty\}$, where $\ell(S)$ denotes the set of tuples over $S$. $c_S(\tau_S)$ is the cost attributed to tuple $\tau_S$ on $S$ by $c_S$. $S$ and $|S|$ are the scope and the arity of c.*

The addition and subtraction of costs are bounded operations, defined respectively as follow:

- $a +_m b = \min(a + b, m)$
- $a -_m b = a - b$ if $a < m$ and $m$ otherwise

For simplicity, $+_m$ and $-_m$ can be briefly denoted by + and − in this document. The cost of a tuple $\tau_S$ of a WCSP $P$ is simply the sum of costs:

$$Val_P(\tau_S) = \textstyle\sum_{(c_{S'} \in C) \wedge (S' \subseteq S)} c_{S'}(\tau_S[S'])$$

A tuple $\tau_S$ is inconsistent if $Val_P(\tau_S) = m$ and feasible (consistent) otherwise. A complete tuple $\tau_X \in \ell(X)$ is a solution of the problem if it is feasible and has the smallest evaluation among all tuples in $\ell(X)$.

A WCSP has no solution if every complete assignment $\tau_X$ of values is inconsistent. Otherwise, the problem has solutions. If the cost of solutions is zero, the problem is totally satisfied.

It is supposed that in every WCSP, there exists a nullary cost function, noted $c_\varnothing$. Since all costs are non negative in CFNs, this constant cost defines a lower bound on the valuation of every solution. This value is very important for the search of solutions that helps to prune sub-problems which have a lower bound greater than the best solution found so far. Thus, improving this value is the central goal of enforcing soft consistencies.

Please notice that if $m = 1$, then WCSPs express crisp constraints where 0 corresponds to authorized and 1 corresponds to forbidden. Thus, WCSPs with $m = 1$ reduce to classical CSPs.

### 2.2.2 Branch-and-Bound search

WCSPs as well as optimization problems can be solved by Branch-and-Bound search [Land and Doig, 1960] [Lawler and Wood, 1966] that is a variant of backtracking search. It memorizes the cost of the best solution found during the search. This value is used as the upper-bound on the cost of solutions (but any other upper-bound can be used). At each node, it computes a lower-bound on the cost of the best solution that lies in the sub-tree below. This lower-bound can be, for example, the nullary cost function $c_\varnothing$ of WCSPs. It will be compared with the current upper-bound of the search. If the lower-bound is higher than or equal to the lower-bound, there does not exists any solution in the sub tree below the node better than the upper bound. In this case, the sub-tree below the node will be pruned, and the search will branch on the next value in the domain of the current variable associated to the node. If no value is left, the search will backtrack up to the most currently assigned variable, as in the backtrack search.

If the lower-bound is not tight, i.e., significantly smaller than the cost of the best solution inside the sub-tree of the node, the search may need to backtrack a lot from a partial solution. The efficiency of Branch-and-Bound search depends on the quality of this lower bound. The higher the lower bound, the more parts of the search are skipped. In the next section, we will introduce a technique for improving the lower-bound provided to Branch-and-Bound search.

### 2.2.3 Equivalent Preserving Transformations

Equivalent Preserving Transformations are operations that transform WCSPs into equivalent problems in terms of the valuation of complete instantiations. Two WCSPs are equivalent iff every *complete* instantiation has the same valuation in both WCSPs.

**Definition 2.13** *Two WCSPs $P = (X, D, C, m)$ and $P' = (X, D, C', m)$ are equivalent iff for every complete assignment of values $\tau_X$, $Val_P(\tau_X) = Val_{P'}(\tau_X)$*

**Example 2.4** *Consider the four binary WCSPs in Figure 2.5. Each WCSP has 2 variables $i, j$ and a cost function $c_{ij}$. All variables have 2 values $a, b$. Contrary to CSPs, an arc in a graph WCSP is used to indicate a tuple of positive cost. Numbers beside values and arcs represent respectively positive unary and binary costs while zero costs edges are not shown. These 4 problems are equivalent because they have the same valuation for every pair of values $(a, b) \in D(i) \times D(j)$ computed as $c_{ij}(a, b) + c_i(a) + c_j(b) + c_\varnothing$*

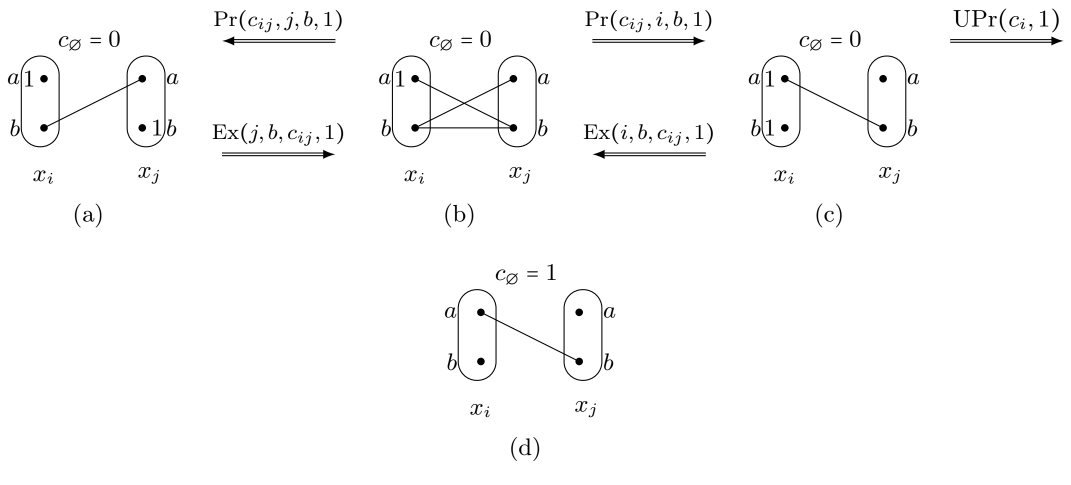


*Figure 2.5: Four equivalent WCSPs. An arrow from a WCSP A to a WCSP B means that B can be transformed from A by applying the EPT indicated above the arrow. Pr, Ex and UPr are respectively the cost projection, cost extension and unary cost projection on $c_\varnothing$.*

Any operation that transforms a WCSP into an equivalent WCSP is called an Equivalence Preserving Transformation (EPT). The operation $\mathsf{Shift}(\tau_S, c_{S'}, \alpha)$ presented in Algorithm 2.2 is such an EPT. It moves an amount of cost $\alpha$ between a cost function $c_{S'}$ and a tuple $\tau_S$ such that $S$ is a subset of $S'$ and $\alpha$ can be negative or positive. Costs must satisfy the last two conditions in order to guarantee that the operation will not create any negative cost in the problem. It is noticed that the second condition is always satisfied if $\alpha > 0$ and the third condition is always satisfied otherwise.

When $\alpha > 0$, Shift sends costs to tuple $\tau_S$ by adding a positive amount of cost $\alpha$ to $c_S(\tau_S)$. This increase of cost will be compensated by subtracting the same amount of cost $\alpha$ from every $c_{S'}(\tau'_{S'})$ of tuple $\tau'_{S'}$ containing $\tau_S$. In this case, Shift corresponds to a *projection* of costs from a cost function of greater arity to a smaller one. When line 2 sets $c_S(\tau_S)$ to $m$ (as $c_S(\tau_S) + \alpha$ reaches $m$), inconsistent tuples over scope $S$ are detected.

Conversely, when $\alpha < 0$, Shift sends costs in the reverse direction, from $\tau_S$ to $c_{S'}$. It adds a positive amount of cost $-\alpha$ to every $c_{S'}(\tau'_{S'})$ and compensates this by subtracting $-\alpha$ from $c_S(\tau_S)$ (or adding $\alpha$ to $c_S(\tau_S)$). In this case, Shift corresponds to an extension of costs from a cost function of smaller arity to a greater one. when line 4 sets $c_{S'}(\tau'_{S'})$ to $m$ (as $c_{S'}(\tau'_{S'}) - \alpha$ reaches $m$), this allows to detect inconsistent tuples over scope $S'$. Notice that here, we understand $c_{S'}(\tau'_{S'}) - \alpha$ as $c_{S'}(\tau'_{S'}) + \alpha'$ where $\alpha'$ is the, non negative, opposite value of $\alpha$.

**Algorithme 2.2 :** Operation for shifting costs in WCSPs

```
1 Procedure Shift($\tau_S, c_{S'}, \alpha$)
      // precondition:;
      1)$S \subset S'$;
      2)$c_S(\tau_S) + \alpha \geq 0$;
      3)$c_{S'}(\tau'_{S'}) \geq \alpha : \forall \tau'_{S'} \in \ell(S'), \tau'_{S'}[S] = \tau_S$;

2     $c_S(\tau_S) \longleftarrow c_S(\tau_S) + \alpha$ ;
3     foreach $\tau'_{S'} \in \ell(S'), \tau'_{S'}[S] = \tau_S$ do
4         $c_{S'}(\tau'_{S'}) \longleftarrow c_{S'}(\tau'_{S'}) - \alpha$;
```

Using the Shift operation, Cooper and Schiex [2004] proposed 3 Soft Arc Consistency (SAC) operations as follows:

- Project($c_S, i, a, \alpha$) is equal to Shift($(i, a), c_S, \alpha$) where $\alpha > 0$. It projects costs from a cost function $c_S$ into a value $(i, a)$.
- Extend($i, a, c_S, \alpha$) is equal to Shift($(i, a), c_S, -\alpha$) where $\alpha > 0$. It extends costs from a value $(i, a)$ onto a cost function on scope $S$.
- Finally, UnaryProject($i, \alpha$) is Shift($\varnothing, c_i, \alpha$) where $\alpha > 0$. This operation is specified for sending costs from a unary cost function $c_i$ to the nullary $c_\varnothing$.

**Example 2.5** *Consider the equivalent WCSPs in Figure 2.5(b). Each one can be transformed to another next to it by applying the Soft Arc Consistency operation indicated on the arrow between them. Let's consider the binary WCSP in Figure 2.5(b). If we project a cost of* 1 *from the binary cost function to value* $(j, b)$*, we will obtain WCSP(a) with the same* $c_\varnothing$*. No unary cost can be projected to* $c_\varnothing$ *in this WCSP. However, if we project a cost of* 1 *from* $c_{ij}$ *to value* $(i, b)$ *in WCSP(b), we will obtain WCSP(c). In WCSP(c),* $i$ *can send a cost of* 1 *to* $c_\varnothing$ *that gives as a result WCSP(d). Observe that WCSP(a),(c) can be converted to WCSP(b) by moving the same amount of cost, but in the reverse direction, as the cost transformation from WCSP(b) to WCSPs(a),(c) respectively.*

The application of the Shift operation, as well as the three SAC EPTs, to a WCSP $P$ produces a valid equivalent WCSP without negative costs [Schiex, 2000]. This important property will be used to enforce all soft consistencies.

### 2.2.4 Soft consistencies

Similarly to hard consistencies which aim at facilitating the backtracking search by making CSPs simpler, soft consistencies are a family of techniques that can improve Branch-and-Bound search by strengthening the lower bound $c_\varnothing$ and pruning inconsistent values (having cost $= m$). The soft consistencies define features for the costs in WCSPs that must be satisfied to simplify the problem. The simplest soft consistency is node consistency which defines a property of unary costs. It requires that every domain value be consistent and that there exists at least one completely consistent value for each domain.

**Definition 2.14 ([Larrosa, 2002])** *A variable $i$ is node consistent (NC) iff $\forall a \in D(i), c_i(a) + c_\varnothing < m$ and there exists a value $a \in D(i)$ such that $c_i(a) = 0$. $a$ is the node support for $i$. A WCSP is node consistent iff each of its variables is node consistent.*

**Algorithme 2.3 :** Algorithm enforcing NC

```
1  Procedure EnforceNC()
2      foreach $i \in X$ do UnaryProject($i$);
3      PruneVars();
4  Procedure PruneVars()
5      foreach $i \in X$ do
6          foreach $v_i \in D(i)$ do
7              if $c_i(v_i) + c_\varnothing = m$ then
8                  remove $v_i$ from $D(i)$;
9                  change ← true;
           // for further consistency enforcement (AC)
10         if change then
11             $Q \leftarrow Q \cup \{i\}$;
12 Procedure UnaryProject($i$)
13     $\alpha \leftarrow \min_{v_i \in D(i)} c_i(v_i)$;
14     UnaryProject($i, \alpha$)
```

**Example 2.6** *The WCSP in Figure 2.5(c) is node inconsistent because no value in $D(i)$ has a zero unary cost. Clearly, a positive cost can be sent from such node inconsistent variable to $c_\varnothing$ and this will result in WCSP (d) which is node consistent.*

If a variable $i$ is not node consistent, this means that some values in $D(i)$ are inconsistent or all values have positive costs. Thus, whenever $NC$ is violated, a value can be deleted or $c_\varnothing$ can be increased. NC can be enforced by Procedure EnforceNC in Algorithm 2.3 where UnaryProject($i$) moves a maximum cost from $c_i$ to $c_\varnothing$ in order to create a node support for $i$, and PruneVars() removes inconsistent values. The time complexity of NC is $O(nd)$.

**Definition 2.15 (Consistent and inconsistent with a soft consistency)** *Given a soft* *consistency. A value or a tuple is called consistent if it satisfies the consistency, or inconsistent otherwise.*

### 2.2.5 Soft arc consistencies

Arc consistencies define features for pairs of a variable and a cost function involving the variable. They are based on the notion of simple and full support which are distinguished by the fact that unary costs can be taken into account in the valuation of supports or not.

**Definition 2.16 (Simple and full arc support)** *Given a variable $i$, a value $a \in D(i)$ and a cost function $c_S$ such that $i \in S$.*

- *A simple arc support of $(i, a)$ in $c_S$ is a tuple $\tau_S$ such that $\tau_S[i] = a$ and $c_S(\tau_S) = 0$.*
- *A full arc support of $(i, a)$ in $c_S$ is a tuple $\tau_S$ such that $\tau[i] = a$ and $c_S(\tau_S) + \sum_{j \in S, j \neq i} c_j(b) = 0$.*

The simplest form of arc consistency is based on simple supports and is called arc consistency. Arc consistency is also called generalized arc consistent (GAC) when we want to emphasize the fact that is applied to non-binary WCSPs. There exists different definitions for arc consistency. In this document, I will use the definition in [Larrosa and Schiex, 2004] which simplifies the definition in [Cooper and Schiex, 2004] by not considering the propagation of completely inconsistent tuples. A value is arc consistent if it has a simple AC support in every cost function.

**Definition 2.17 (GAC)** *A WCSP $P$ is generalized arc consistent iff for every of its variables $i$, for every value $a \in D(i)$ and for every cost function $c_S$ with $|S > 1|$, there exists a simple arc support for $(i, a)$ in $c_S$. $P$ is GAC* if it is GAC and NC.*

**Example 2.7** *Consider the binary problem in Figure 2.5(b). It is arc inconsistent because value $(i, b)$ has no arc support in $c_{ij}$. Applying* Project$(c_{ij}, i, a, 1)$ *results in Problem (c) which is arc consistent but node inconsistent. Applying* UnaryProject$(i, 1)$ *in Problem (c) to enforce NC will increase $c_\varnothing$ by 1 and result in an AC* problem (d).*

Enforcing GAC can break NC. When a value $(i, a)$ is arc inconsistent in $c_S$, every tuple $\tau_S$ such that $\tau_S[i] = a$ has a positive cost $c_S(\tau_S) > 0$. Using Project operation to move costs from $c_S$ to $(i, a)$ allows to (1) create a simple support for $(i, a)$ and (2) increase the unary cost $c_i(a)$, that possibly reaches $m$.

We observe that both WCSPs in Figure 2.5(a),(d) are AC−closures of the WCSP in Figure 2.5(b). The soft AC−closure of a WCSP is not unique as in classic CSPs. The quality of a closure, represented by the associated value of $c_\varnothing$, may depend on the order of application of the EPTs. Finding the best order for EPT application (in terms of the increase in $c_\varnothing$) under the condition that costs remain integer to enforce AC is NP-complete [Cooper and Schiex, 2004]. Beyond general cost functions, Lee and Leung [2012] presents an algorithm enforcing GAC* for global constraints (non binary cost functions with specific semantics). For simplicity, we restrict ourselves to binary WCSPs. It is shown in [Larrosa and Schiex, 2004] that every binary WCSP can be transformed into an equivalent AC* one in time $O(ed^3)$.

Procedure EnforceAC() described in Algorithm 2.4 enforces AC*. Notice that this procedure can be adapted in the implementation. For example, in the solver `toulbar2`, unary costs

**Algorithme 2.4 :** Algorithm enforcing AC

```
1  Procedure EnforceAC()
2      $Q \leftarrow X$;
3      AC();

4  Procedure AC()
5      while $Q \neq \varnothing$ do
6          $j \leftarrow Q$.pop();
7          foreach $c_{ij}$ do
8              if FindSupport($i, c_{ij}$ ) then
                   // for further consistency enforcement (DAC, EAC)
9                  $P \leftarrow P \cup \{i\}$;
10                 $S \leftarrow S \cup \{i\}$;
11         PruneVars();

12 Procedure FindSupport($i, c_{ij}$)
13     flag ← false;
14     foreach $v_i \in D(i)$ do
15         $\alpha \leftarrow \min_{v_j \in D(j)}\{c_{ij}(v_i, v_j)\}$;
16         if $\alpha > 0$ then
17             if $c_i(v_i) = 0$ then  flag ← true; ;
18             Project($c_{ij}, i, v_i, \alpha$);
19     UnaryProject($i$);
20     return flag;
```

are organized in buckets to avoid rechecking NC on all values when $c_0$ increases. This procedure is based on AC3. It uses a propagation queue $Q$ to contain variables whose domain has been reduced. At each iteration, a variable $j$ is popped out from $Q$. Values in neighbors of $j$ may have lost AC support in $D(j)$ and thus need to be checked for AC using Procedure FindSupport($i, c_{ij}$)(line 8). This Procedure forces simple AC supports for values of $i$ which have no arc support in $c_{ij}$ (line 16) by moving costs from $c_{ij}$ to $c_i$ (line 18). This projection of costs on $c_i$ can make $i$ node inconsistent and thus a new node support needs to be enforced for $i$ by UnaryProject to move cost from $c_i$ to $c_\varnothing$ (line 19). In addition, Procedure PruneVars is used to remove new inconsistent values (line 11). Whenever a value is removed in PruneVars, the corresponding variable will be pushed into $Q$ for further AC propagation. The Boolean value returned by FindSupport and queue $P$ and $S$ will be used for further consistencies.

In Procedure AC, each constraint $c_{ij}$ is checked for AC by FindSupport at most $2d$ times because each variable is pushed into $Q$ at most $d$ times. Because FindSupport($i, c_{ij}$) takes $O(d^2)$ time, the time complexity of AC as well as EnforceAC is $O(ed^3)$.

### Directional arc consistency

An intuitively appealing stronger arc consistency for WCSPs is full arc consistency (FAC) which would be based on the notion of full AC support [Zlomek and Bartak, 2005]. FAC requires full AC supports, on both sides, for each cost function. Unfortunately, FAC is not a practical property because it cannot be always satisfied [Larrosa et al., 2005] (see Figure 2.6). This motivated the introduction of weaker than FAC properties. The first proposed one is directional arc consistency (DAC) [Cooper, 2003]. The purpose of DAC is to still offer full arc supports as in FAC while guaranteeing the existence of an equivalent locally consistent problem, by restricting the search for full AC supports to one side for each cost function according to a defined order "<" of variables.

For simplicity, in this section, we restrict ourselves to binary WCSPs. The definitions and algorithms for enforcing soft arc consistencies in non-binary WCSPs are introduced in [Cooper and Schiex, 2004; Lee and Leung, 2009, 2012]. For binary WCSPs, a value $(i, a)$ is directional arc consistent if it has a full AC support in every cost function $c_{ij}$ such that $i < j$.

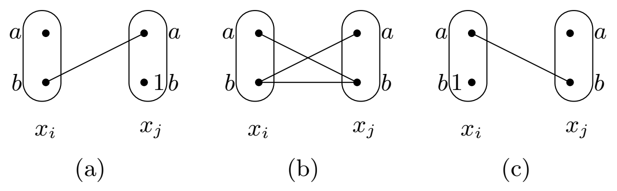


*Figure 2.6: The non existence of a FAC problem among all equivalent WCSPs [Larrosa et al., 2005]*

**Definition 2.18 (DAC [Larrosa and Schiex, 2003])** *A binary WCSP P is directional arc consistent (DAC) with respect to an order < of variables iff for every variable $i$, for every value $a \in D(i)$, for every cost function $c_{ij}$ such that $i < j$, there exists a full arc support for $(i, a)$ in $c_{ij}$. P is DAC* if it is DAC and NC.*

DAC and DAC⋆ can be enforced for binary WCSPs in time $O(ed^2)$ and in space $O(ed)$.

**Example 2.8** *Consider the WCSP in Figure 2.5(a). Suppose that $i < j$. $(i, b)$ has no full AC support in $c_{ij}$ despite the fact that $(j, b)$ is a simple AC support for it. Thus the problem is not directional arc consistent. Conversely, WCSP in Figure 2.5(d) is directional arc consistent whatever the order between $i$ and $j$.*

Procedure EnforceDAC() in Algorithm 2.5 enforces DAC*. The propagation queue $P$ contains variables with values that have increased costs from 0. This queue is only useful when DAC is enforced with other local consistencies. Variables in $P$ are arranged with respect to the "<" order and processed from the greatest to the smallest in the queue in order to minimize the number of constraint checks. At each iteration of Procedure DAC(), a variable $j$ is popped out from $P$. Some values in lower variables may have lost full AC support due to new positive unary costs in $D(j)$. Thus, a new full AC support needs to be sought for such values by Procedure FindFullSupport($i, c_{ij}$) (line 8). This procedure enforces full AC supports for values of $i$ in $c_{ij}$ such that $i < j$ by extending a cost $E[b]$ from every value $(j, b)$ to $c_{ij}$ (line 20) and then projecting a cost $P[a]$ from $c_{ij}$ to every value $(i, a)$ (line 19). These amounts of cost $E[b], P[a]$ are computed in such a way that maximum costs can be projected on $c_i$ while not creating any negative cost and ensuring simple AC supports for values of $j$ (line 15, 18). Node consistency of $i$ can be broken due to cost projections in $i$ and thus Procedure UnaryProject($i$) will be used to enforce a NC support for $i$. If a value increases cost from 0, the procedure returns `true` ( line16). In this case,

**Algorithme 2.5 :** Algorithm enforcing DAC

```
1  Procedure EnforceDAC()
2      $P \leftarrow X$;
3      DAC ();
4  Procedure DAC()
5      while $P \neq \emptyset$ do
6          $j \leftarrow P$.popmax();
7          foreach $c_{ij}$ such that $i < j$ do
8              if FindFullSupport($i, c_{ij}$) then
9                  $P \leftarrow P \cup \{i\}$;
10                 $S \leftarrow S \cup \{i\}$; // for further consistency enforcement (EAC)
11         PruneVars();
12 Procedure FindFullSupport($i, c_{ij}$)
13     flag ← false;
14     foreach $v_i \in D(i)$ do
15         $P[v_i] \leftarrow \min_{v_j \in D(j)}\{c_{ij}(v_i, v_j) + c_j(v_j)\}$;
16         if $c_i(v_i) = 0$ and $P[v_i] > 0$ then flag ← true ;
17     foreach $v_j \in D(j)$ do
18         $E[v_j] \leftarrow \max_{v_i \in D(i)}\{P[v_i] - c_{ij}(v_i, v_j)\}$ ;
19     foreach $v_i \in D(i)$ do Project($c_{ij}, i, v_i, P[v_i]$) ;
20     foreach $v_j \in D(j)$ do Project($j, v_j, c_{ij}, E[v_j]$) ;
21     UnaryProject($i$);
22     return flag;
```

neighboring values of $i$ also may have lost full support and thus, $i$ will be pushed into $P$ for further propagation (line 9).

When a variable $j$ is popped out from $P$, all variables before (greater than) $j$ have been processed. Only variables smaller than $i$ can have lost full support and can increase cost by FindFullSupport. Thus, $j$ as well as greater variables than $j$ will never be pushed again into $P$. Each variable $j$ is pushed into $P$ at most once and the while loop in Procedure DAC() traverses $P$ once. As a result, each constraint $c_{ij}$ is enforced for DAC by FindFullSupport() at most once. Since the complexity of FindFullSupport is $O(d^2)$, the time complexity of DAC and EnforceDAC() is $O(ed^2)$. The space complexity of DAC() and EnforceDAC() is $O(ed)$

### Full directional arc consistency

Full directional arc consistency ensures DAC on one side of each cost function and AC on the other side. For binary WCSPs, a value $(i, a)$ is full directional arc consistent iff it has a full AC support on every cost function $c_{ij}$ such that $i < j$ and a simple AC support on every $c_{ik}$ such that $i > k$.

**Definition 2.19 (FDAC [Larrosa and Schiex, 2003])** *A WCSP $P$ is full directional arc consistent (FDAC) with respect to an order $<$ of variables if it is arc consistent and* *directional arc consistent with respect to $<$. $P$ is FDAC* if it is FDAC and NC.*

**Algorithme 2.6 :** Algorithm enforcing FDAC

```
1 Procedure EnforceFDAC()
2     $P \leftarrow Q \leftarrow X$;
3     while $Q \neq \varnothing$ or $P \neq \varnothing$ do
4         AC ();
5         DAC () ;
```

It is shown in [Larrosa and Schiex, 2003] that every WCSP can be transformed into an equivalent FDAC* one in time $(end^3)$. Indeed, FDAC* can be enforced simply by enforcing AC* and DAC* simultaneously as in Procedure EnforceFDAC. Enforcing AC empties $Q$ but can add variables into $P$ while enforcing DAC empties $P$ but can add variables into $Q$. As previously mentioned, enforcing singleton AC only uses $Q$ and DAC only uses $P$. The procedure only terminates when both queues are empty.

The maximum number of value removals, caused by both DAC() and AC(), is $nd$. The number of times executing line 8 (for enforcing AC for a constraint) and line 9 (for adding variables into $P$) in Procedure AC() of Algorithm 2.4 is $O(nd)$. Thus, line 5 of EnforceFDAC takes $(ed^3)$ in times similarly to AC(), and line 4 of EnforceFDAC takes $O(nd) \times f(\mathsf{DAC}()) = O(end^3)$. The total time complexity of EnforceFDAC is $O(end^3)$.

### Existential arc consistency

In previous arc consistencies, every domain value must satisfy a given property which is weaker than FAC. Instead, Existential arc consistency requires the existence of a specific value in each domain which satisfies a property stronger than FAC [Larrosa et al., 2005]. A variable is existential arc consistent if there exists a value of zero unary cost in its domain such that this value has a full AC support in every soft function involving the variable. It is noticed that if a variable $i$ is not existential arc consistent, for every value $(i,a)$ such that $c(i) = 0$, there exists a cost function $c_S$ such that $i_a$ has no full AC support in $c_S$. Enforcing full AC support in such cost functions will move cost to $(i,a)$ and break node consistency of the variable. Enforcing NC for this variable will increase $c_\varnothing$.

**Definition 2.20 (EAC [Larrosa et al., 2005])** *A binary WCSP is existential arc consistent (EAC) iff for every variable $i$, there exists a value $a \in D(i)$ such that $c_i(a) = 0$ and for every cost function $c_{ij}$, 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 for variable $i$. $P$ is EAC* if it is EAC and NC.*

**Example 2.9** *Consider the WCSP in Figure 2.7. Value $(j,a)$ has no full arc support in $c_{ij}$ while value $(j,b)$ has no full arc support in $c_{jk}$. Therefore variable $j$ has no existential arc support.*

EAC is enforced by Procedure EnforceEAC(). It uses a propagation queue $R$ containing variables that need to be checked for EAC. Each iteration of Procedure EAC() considers a variable $i$ popped out from $R$ (line 6). Procedure FindEACSupport($i$) enforces EAC supports for variable $i$. The condition at line 14 is used to check whether the variable $i$ is still existential arc consistent or not. If not (line 15), full arc supports will be enforced for values of $i$ in every cost function. The procedure returns `true` if at least a value in $D(i)$

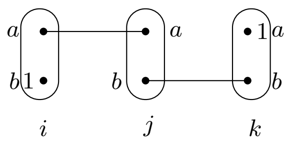


*Figure 2.7: An existential arc inconsistent WCSP*

**Algorithme 2.7 :** Algorithm enforcing EAC

```
1  Procedure EnforceEAC
2      $R \leftarrow X$;
3      EAC();

4  Procedure EAC()
5      while $R \neq \varnothing$ do
6          $i \leftarrow R$.pop();
7          if FindEACSupport($i$) then
8              $P \leftarrow P \cup \{i\}$;
9              foreach $c_{ij}$ s.t $j > i$ // for further consistency enforcement (EDAC)
10             do $R \leftarrow R \cup \{j\}$ ;
11         PruneVars();

12 Procedure FindEACSupport($i$)
13     flag ← false;
14     $\alpha \leftarrow \min_{a \in D(i)}\{c_i(a) + \sum_{c_{ij}, i>j} \min_{b \in D(j)}\{c_{ij}(a,b) + c_j(b)\}\}$;
15     if $\alpha > 0$ then
16         foreach $c_{ij}$ s.t $i > j$ do
17             flag ← flag ∨ FindFullSupport($i, c_{ij}$);
18     UnaryProject($i$);
19     return flag;
```

has increased cost from 0 during FindFullSupport($i, c_{ij}$) (line 17). In this case, neighbor variables $j$ of $i$ may have lost full AC support in $D(i)$ due to new positive unary costs in $D(i)$ and thus $j$ may have lost EAC support. Therefore, all neighboring variables of $i$ will be added into $R$ to be checked for EAC later (line 10). Ignore for the moment the conditions marked in gray. They will be used later.

It is noticed that whenever EAC is violated, $c_\varnothing$ will increase at least by 1. EAC cannot be violated more than $m$ times because the procedure will stop when $c_\varnothing$ reaches $m$. As a result, line 10 cannot be executed more than $m$ times. The total number of times that variables are pushed again into $R$ is less than $m$. In addition to the initialization phase, the maximum size of queue $R$ is $n + m$. Procedure FindEACSupport() takes time $O(gd^2)$, where $g$ is the maximum variable degree. Globally, EAC() and EnforceEAC() have a time complexity in $O(n + m)gd^2 = O(\max(ed^2, mgd^2))$.

### Existential directional arc consistency

Existential directional arc consistency gets closer to FAC by using both FDAC and EAC properties [Larrosa et al., 2005]. It requires every domain value to be full directional arc consistent and at least a value per domain is existential arc consistent.

A variable is existential directional arc consistent if every value in its domain is fully supported in one direction and simply supported in other direction while at least one value is fully supported in every direction.

**Definition 2.21 ( [Larrosa et al., 2005])** *A normalized binary WCSP $P$ is existential directional arc consistent (EDAC) with respect to an order $<$ of variables iff it is existential arc consistent and full directional arc consistent with respect to $<$. $P$ is EDAC* if it is EDAC and NC.*

EDGAC* can oscillate with constraints sharing more than one variable [Lee and Leung, 2010], where enforcing FDGAC* breaks EAC* and vice versa. In the case of binary WCSPs, the problems must therefore be normalized to apply EDAC*. In general, in order to avoid this oscillation, [Lee and Leung, 2010] proposed a weak variant of EGDAC by modifying the definition of EAC support. The idea of the weak EAC support is to distribute unary costs to each value domain $D(i)$ in such a way that each unary cost $c_j$ is taken into account exactly once via a cost function $c_S$: $\{i, j\} \subset S$. Suppose that $j$ is a variable shared by two cost functions $c_S, c_{S'}$ related to $i$. $c_j$ is taken into account in the full arc support of $i$ on either $c_S$ or $c_{S'}$, rather than on both, as in EAC.

Precisely, weak EAC groups variables adjacent to each variable $i$ in $k$ sets, where $k$ is the number of cost functions related to $i$, in such a way that no two sets share the same variables. Each cost function $c_S$ of $i$ is associated to a subset of variables of $S$, denoted by $B_{i,c_S}$. Only unary costs of the variables in this set are taken into account in the definition of the weak full arc support for $i$ on $c_S$. A weak full support for a value $(i, a)$ on $c_S$ is a tuple $\tau_S \in \ell(S)$ such that $\tau_S[i] = a$ and $c_S(t) + \sum_{j \in B_{i,c_S}} c_j(t[j]) = 0$. A value $(i, a)$ is weak EGAC iff it has a weak full arc support on every cost function. A WCSP is weak EDAC if it is FDGAC* and EGAC.

EDAC* can be enforced in binary normalized WCSPs in time $O(\max\{nd, m\}ed^2)$ [Larrosa et al., 2005] by Procedure EnforceEDAC.

**Algorithme 2.8 :** Algorithm enforcing EDAC

```
1 Procedure EnforceEDAC
2     $Q \leftarrow P \leftarrow S \leftarrow X$;
3     while $Q \neq$ or $P \neq$ or $S \neq$ do
4         $R \leftarrow S \cup \{i \mid j \in S, c_{ij} \in C, i > j\}$ ; $S \leftarrow \varnothing$ ;
5         EAC();
6         DAC();
7         AC();
```

This procedure enforces EDAC by simply enforcing EAC, DAC and AC simultaneously. It uses 3 propagation queues $S, P, Q$ where $P, Q$ have been previously introduced, in Algorithm 2.4 and 2.5. Queue $S$ contains variables for which some values have their cost increased from 0 during AC() and DAC() (line 10 of Algorithm 2.4 and line 10 of Algorithm 2.5). $S$ is a supplementary queue to effectively build queue $R$ which is used for EAC() and which contains variables that need to be checked for EAC. If $j \in S$, one of the removed values in $D(j)$ could be the EAC support of $j$ and thus $j$ needs to be checked for EAC. At the same time, values of variables $i$ adjacent to $j$ may have lost full supports in $D(j)$ and thus also need to be checked for EAC. Globally, variables in $S$ and their neighbors need to be checked for EAC and thus are pushed into $R$ (line 4). Each iteration of the procedure enforces EAC, DAC and then AC. EAC() empties $S$ but can add variables into $P$. DAC() empties $P$ but can add variables into $S$. AC() empties $Q$ but can add variables into $S, P$. PruneVars() can add variables into $Q$. The procedure only terminates when all three queues are empty.

### 2.2.6 High order consistencies

#### Complete k-consistency

Complete $k$-consistency is a strong consistency for WCSPs that was proposed by Cooper [2005] based on $(i, j)$-consistency for CSPs Freuder [1985], where $i+j=k$. This consistency guarantees the consistency for any sub-problem of $k$ variables in such a way that each partial tuple of size smaller than $k$ can be extended without extra cost on a tuple of $k$ variables.

The extra cost for extending a tuple $\tau_I \in \ell(I)$ to $\tau'_J \in \ell(J)$ such that $I \subset J, \tau'_J$ compatible to $\tau_I$ (i.e. $\tau'_J[I] = \tau_I$) is computed as the sum $\sum_{S \subset J, |S|>|I|} c_S(\tau'_J[S])$. The extra cost for extending $\tau_I$ to the set $J$ of variables is the minimum among the extra costs for extending $\tau_I$ to compatible tuples in $\ell(J)$. Complete $k$-consistency requires for any set $J$ of $k$ variables, for any subset $I$ such that $\varnothing \subseteq I \subset J$, for any tuple $\tau_I$ over $I$, that the extra cost for extending $\tau_I$ to $J$ is zero.

**Definition 2.22 (Complete $k$-consistency)**

- *Given a set $J$ of $k$ variables, a subset $I$ of variables: $\varnothing \subseteq I \subset J$. A tuple $\tau_I$ of $\ell(I)$ is complete $k$-consistent on $J$ iff there exists a tuple $\tau'_J$ in $\ell(J)$ compatible with $\tau_I$ ($\tau'_J[I] = \tau_I$) such that $\sum_{S \subset J, |S|>|I|} c_S(t'[S]) = 0$. Such a tuple $\tau'_J$ is called the complete $k$-consistency support for $\tau_I$ on $J$.*
- *$\tau_I$ is complete $k$-consistent iff it is complete $k$-consistent on every set of $k$ variables.*
- *A subset $I$ of variables is complete $k$-consistent iff every tuple of $\ell(I)$ is complete $k$-consistent.*
- *A WCSP is complete $k$-consistent iff every subset of less than $k$ variables is complete $k$-consistent.*

The complete $k$-consistency support of a tuple $\tau_I \in \ell(I)$ on $J$ takes into account all costs associated to cost functions of arities greater than $|I|$. When $|I| = 0$, i.e., $I = \varnothing$, complete $k$-consistency means that there exists at least an instantiation for any set of $k$ variables such that the combined cost, including unary, binary, ..., $k$-ary costs, is zero. When $|I| = 1$, complete $k$-consistency means that for any value of any variable, for any set of $k$ variables including the variable, there exists at least an instantiation such that the combined cost including binary, ..., $k$-ary costs, is zero. In general, the property that complete $k$-consistency defines for each size $|I|$ corresponds to $(|I|, k - |I|)$-consistency in

CSPs in the sense of the extensibility of $|I|$ variables on $k - |I|$ extra variables. $|I| = 0$ is a special case because $(0, k)$−consistency has not been defined for CSPs but it is the same idea. In summary, complete $k$-consistency corresponds to the full set of $\{(0, k), (1, k - 1), \ldots, (k - 1, 1)\text{-consistencies}\}$.

In binary WCSPs, complete 2-consistency is related to simple arc consistency: the property it defines for values is exactly simple arc consistency. By considering the fact that consistency in CSPs is equivalent to a zero cost in WCSPs, we notice that in binary CSPs (1) complete 3-consistency corresponds to hard PIC because of the property it defines for values ($|I| = 1$) and (2) complete 3-consistency is related to hard PC because of the property it defines for pairs of values ($|I| = 2$).

In order to enforce complete $k$-consistency supports for a tuple of $\ell(I)$ in a set of $k$ variables, [Cooper, 2005] performs a process of two steps.

- The first step extends costs from all cost function $c_S$ such that $\varnothing \subseteq S \subset J$ and $|S| > |I|$ to $c_J$. If $|J| = |I| + 1$, this step is skipped. This steps allows to empty every cost of arities greater than $|I|$ and smaller than $|J| = k$. However it can create a new cost function on scope $J$ in the case that $c_J$ does not exist.
- The second step projects costs from $J$ to $I$. This step allows to create $k$-ary instantiations of zero cost. The combined cost of such instantiations, including costs of arities $\in (|I|, |J|]$, is zero because the associated costs of arities $\in (|I|, |J|]$ have been emptied in the first step. Thus, these instantiations become complete $k$-consistent supports.

### Tuple consistency

Tuple consistency, proposed by Dehani et al. [2013], is an extension of simple arc consistency. Instead of defining a property on the cost (variable, cost function) pairs, it defines it on pairs of (scope, cost function), where scope is a subset of variables inside the scope of the cost function. In other words, simple arc consistency guarantees the extensibility of values on cost functions while tuple consistency guarantees the extensibility of tuples on cost functions. For a cost function $c_S$, it requires that any partial tuple over scope $S' \subset S$ must be extended on a tuple over $S$ that has zero-cost in $c_S$.

**Definition 2.23 (Tuple consistency)** *Given a cost function $c_S$ and a scope $S' \subset S$. A tuple $\tau'_{S'} \in \ell(S')$ is tuple consistent (TC) iff there exist a tuple $\tau_S \in \ell(S)$ such that $\tau_S[S'] = \tau'_{S'}$ and $c_S(t) = 0$. Such a tuple $\tau_S$ is called a tuple consistency support for $\tau'_{S'}$ on $c_S$. The scope $S'$ is tuple consistent iff every tuple of $\ell(S')$ is tuple consistent. A WCSP is tuple consistent iff every subset of variables is tuple consistent.*

Notice that $k$-consistency handles sub-problems of $k$ variables, for a fixed integer $k$, regardless whether $k$ variables of the sub-problems are connected to each other by a $k$-ary cost function or not. Conversely, tuple consistency handles sub-problems of variables that are connected to each other by a cost function. These sub-problems have a size that changes according to the arity of the cost functions.

Indeed, TC support can be called simple arc support for tuples that are values. The property TC defined for scopes of size 1 i.e., variables, is exactly simple AC. Thus, in binary WCSP, TC becomes simple AC, and both are related to 2-complete consistency as analyzed in the previous section. In ternary WCSPs, we have 3-complete consistency > TC > simple AC, where > means the stronger relation. TC is stronger than simple AC because the property it defines for scopes of size 1 is already simple AC. TC is weaker than complete 3-consistency because complete 3-consistency takes into account unary and binary costs in the support for $c_\varnothing$ (empty scope) and binary costs in the support for values (tuples of size 1) but TC does not, despite both consistencies define the same features for pairs of values (tuples of size 2).

In order to enforce TC supports, Dehani et al. [2013] use an Equivalent Preserving Transformation which projects costs from cost functions $c_S$ to tuples of smaller scopes $S' \subset S$. This transformation can create new cost functions over scopes $S'$. In WCSPs having cost functions of large arities, TC can create cost functions for all subset of variables and thus TC is not practical. Therefore, a limited variant of TC, denoted as $TC_r$, has been also proposed by guaranteeing TC only for tuples of size smaller than $r$. A WCSP is $TC_r$ iff every scope $S'$ such that $|S'| < r$ is TC. $TC_1$ is simple AC.

### Cyclic consistency

Cyclic consistency [Cooper, 2004] is a limited version of Optimal soft arc consistency [Cooper et al., 2007] that relaxes the traditional SAC operations used in soft arc consistencies by allowing negative costs. A cyclic consistency operation is a set of relaxed SAC operations used for a cycle of variables $(i_1, i_2, \dots, i_r)$ where $i_1 < i_2 < \dots < i_r < i_1$. It transforms WCSPs into valid WCSPs without negative costs. Relaxed SAC operations are only applied on two consecutive variables in the cycle. Each variable can receive costs from the previous and send costs to the next one in the cycle. The amount of shifted costs in each individual relaxed SAC operation can be negative but no negative cost is created after applying all the relaxed SAC operations of the cyclic consistency operation.

A variable $i$ is cyclic consistent if there is no cyclic consistency operation that can increase the unary cost of all values $a \in D(i)$ such that $c_i(a) = 0$. A WCSP is cyclic consistent if every variable is consistent. Cyclic consistency is optimal in the sense that when a WCSP is cyclic consistent, there is no set of traditional SAC operations when applied on cycles of variables that can increase the lower bound $c_\varnothing$.

### Enhanced arc consistencies for ternary WCSPs

In order to avoid the oscillation of EDAC in ternary WCSPs, [Sánchez et al., 2008] redefined the full arc supports for ternary cost functions by using not only unary but also binary costs. The definition of the simple arc support (for both binary and ternary cost functions) as well as the full arc support for binary cost functions are not changed.

**Definition 2.24 (full arc support)** *For a ternary cost function $c_{ijk}$, a pair of values $(j, b), (k, c)$ is a full arc support of a value $(i, a)$ iff* $c_{ijk}(a,b,c)+c_{ij}(a,b)+c_{ik}(a,c)+c_{jk}(b,c)+c_j(b) + c_k(c) = 0$

Hereafter, the newly defined full support is called enhanced full arc support to distinguish from the original one [Lee and Leung, 2012]. At the same time, we use notation + to denote soft arc consistencies which use the enhanced full arc support instead of the original one, that is DAC$^+$, FDAC$^+$, EAC$^+$, EDAC$^+$.

A ternary WCSP $P$ is DAC$^+$ w.r.t an order "$<$" of variables iff every value of every variable has a full support in every binary cost function $c_{ij}$ such that $j > i$ and in every ternary cost function $c_{ijk}$ such that $j > i, k > i$. $P$ is FDAC$^+$ if every variable is DAC$^+$ on cost functions $c_{ij}, c_{ijk}$ such that $i < j, i < k$ and AC on the other ones. $P$ is EAC$^+$ if there exists at least one value per variable that has full support on every cost function. $P$ is EDAC$^+$ if it is FDAC$^+$ and EAC$^+$.

**Example 2.10** *Consider the WCSP in 2.8(a). It is FDAC but not EAC because of variable $x_2$. Sub-figures 2.8(b),(c) demonstrate the enforcement of full arc support for value $(x_2, a)$ in $c_{12}$ in order to enforce $P$ for EAC. The extension of unary cost $c_1(a)$ on $c_{12}$ during this enforcement makes $(x_2, b)$ fully supported by $(x_1, a)$ in the ternary cost function $c_{123}$ (Sub-figure(c)). The resulting problem is thus EAC but it is now not FDAC because of variable $x_1$ in the cost function $c_{12}$. Enforcing FDAC converts the problem back to the original problem in Sub-figure 2.8(a). Therefore, enforcing EDAC oscillates. However, the problem in Sub-figure 2.8(c) is not EAC$^+$ because $(x_2, b)$ has no enhanced full support in $c_{123}$ due to the presence of the binary cost $c_{12}(a, b)$. As shown in the next example, enforcing EAC$^+$ in $P$ will lead to a problem EDAC$^+$ with an increased lower bound.*

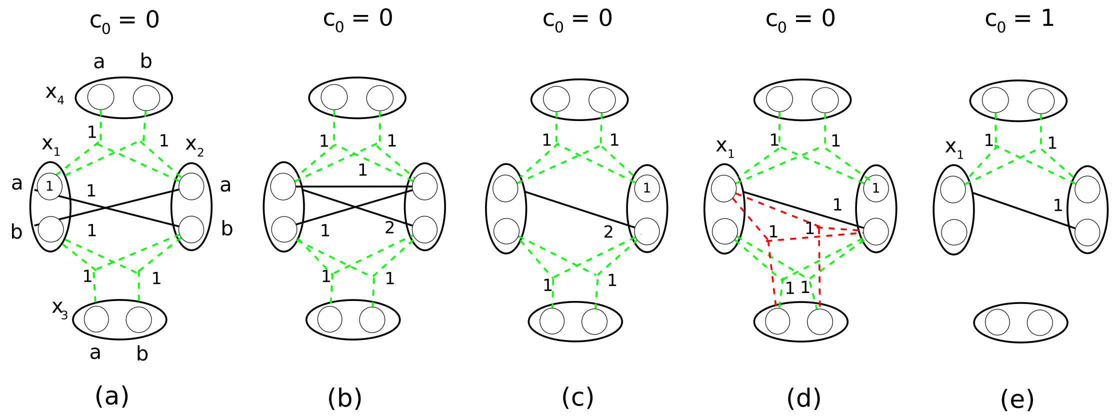


*Figure 2.8: Enforcing enhanced EDAC*

Sánchez et al. [2008] proposed an algorithm to enforce enhanced full arc supports for ternary cost functions. The idea is to extend unary and ternary costs (respectively on binary and ternary cost functions) involved in the scope of ternary cost functions in such a way that a maximum projection can be performed on inconsistent values. Precisely it computes for each value $(i, a)$ in each cost function $c_{ijk}$ the maximum amount of cost that can be projected on $(i, a)$. In order to archive this cost projection, it firstly extends unary costs from $c_j$ and $c_k$ respectively on $c_{ij}$ and $c_{ik}$ and then extends binary costs to $c_{ijk}$. These extensions are properly computed in the sense that weaker extensions cannot lead to the maximum projection on $c_i$ while stronger cannot lead to a larger projection on $c_i$. This enforcement does not break arc consistency at binary and ternary level.

**Example 2.11** *Consider again the example 2.10. Enforcing EAC$^+$ requires to enforce enhanced full supports. Enforcing enhanced full support for $(x_2, b)$ in $c_{123}$ requires firstly to extend the binary cost $c_{12}(a, b)$ on $c_{123}$ an amount of cost 1 (Sub-figure 2.8(d)) and then to project from $c_{123}$ on $(x_2, b)$ an amount of cost 1. Finally, enforcing NC for $x_2$ allows to increase $c_\varnothing$ by 1 (Sub-figure 2.8(e)).*

### 2.2.7 Virtual arc consistency

The common feature of all soft arc consistencies introduced in previous section such as AC, DAC, FDAC, EDAC, is that the consistency enforcement is performed by applying a chaotic sequence of EPTs until a fixpoint i.e., a corresponding arc consistency closure, is established. In this section, we will present a consistency enforced in a different way. Virtual arc consistency, proposed in [Cooper et al., 2008, 2010], is enforced in WCSP $P$ by applying a planned sequence of EPTs which is defined through a classical CSP, called Bool($P$). The application of each such sequence of EPTs will always lead to an increase of the lower bound $c_\varnothing$ when the problem is not virtual arc consistent.

**Definition 2.25** *Given a CFN $P = (X, D, C, m)$, the CSP Bool($P$) $= (X, D, \overline{C}, 1)$ is such that $\exists \overline{c}_S \in \overline{C}$ iff $\exists c_S \in C$, $S \neq \varnothing$ and $\overline{c}_S(\tau_S) = 1 \Leftrightarrow c_S(\tau_S) \neq 0$.*

Bool($P$) is therefore a CSP whose solutions are exactly all complete tuples having cost $c_\varnothing$ in $P$.

**Definition 2.26** *A CFN $P$ is virtual arc consistent (VAC) iff the arc consistent closure of the CSP Bool($P$) is non-empty.*

If $P$ is not VAC, i.e., the arc consistency closure of Bool($P$) is non-empty, enforcing AC on Bool($P$) will lead to a domain wipe-out. In this case, it has been shown and proved in [Cooper et al., 2008, 2010] that there exists a sequence of EPTs which leads to an increase of $c_\varnothing$ when applied on $P$. To exploit this property, VAC enforcing uses an iterative process. Each iteration consists of 3 phases. **Phase 1** is an instrumented AC enforcing in Bool($P$) in order to determine whether the WCSP $P$ is VAC or not. If $P$ is VAC, i.e., no wipe-out occurs, the algorithm stops. Otherwise, it proceeds to Phase 2. **Phase 2** aims at defining the minimal set of necessary value deletions that lead to the wipe-out. At the same time, it computes the minimal sequence of EPTs and the amount of cost that can be moved to $c_\varnothing$ via the wiped-out variable. **Phase 3** applies EPTs defined in Phase 2 to increase $c_\varnothing$.

#### Algorithm enforcing VAC

**Phase 1** For enforcing AC in CSP Bool($P$), Phase 1 may use a coarse grained algorithm such as AC3 or AC2001 with an extra maintained data-structure during the enforcing, as described in Algorithm 2.9. The instrumented version records every deletion in a dedicated data-structure denoted as killer. When a value $(i, a)$ lacks a valid support on a cost function $\overline{c}_S$, we set killer((i,a)) = $S$ and we delete the value (line 5, 6). This deleted value is also stored in queue $Q$ that will be used in Phase 2. The revise propagation queue $Q_{AC}$, initialized to contain all variables $X$, contains variables having reduced domains (their neighbors need to be rechecked for AC).

If no domain wipe-out occurs, $P$ is VAC and we stop. Otherwise, when a wipe-out is detected in variable $i_0$, it is important to realize that the resulting problem is not yet the arc consistent closure of Bool($P$). There may still be values which have no valid support and are not deleted because the corresponding domain has not been revised. These "pending for revision" domains are still represented in the propagation queue $Q_{AC}$. The remaining values either have a valid support or have been deleted and have a non empty associated killer. Such a problem will be called a *justified partial* AC–closure of Bool($P$). It cannot have larger domains than Bool($P$) and all deletions are properly justified by their killer.

**Algorithme 2.9 :** VAC iteration - Phase 1: Instrumented AC

```
1  Procedure Revise($i, S$)
2      change ← false;
3      foreach $a \in D(i)$ do
4          if $\nexists \tau \in \ell(S)$ s.t. $(\tau[i] = a) \wedge (\tau \in c_S)$ then
5              delete $a$ from $D(i)$;
6              killer$[i, a] \leftarrow S$;
7              $Q$.push$(i, a)$;
8              change ← true;
9      return change;
10 Procedure Instrumented-AC()
11     $Q_{AC} \leftarrow \{(i, S) | c_S \in C, i \in S\}$;
12     while $Q_{AC} \neq \varnothing$ do
13         $(i, S) \leftarrow P$.pop();
14         if Revise$(i, S)$ then
15             if $D(i) = \varnothing$ then return $i$;
16             else $Q_{AC} \leftarrow Q_{AC} \cup \{(j, S') | c_{S'} \in C, S' \neq S, \{i, j\} \subseteq S', j \neq i\}$;
17     return 0;
```

**Phase 2** The second phase of VAC is described in Algorithm 2.10. It first identifies the subset of EPTs that are necessary to produce the wipe-out. This is achieved by tracing back the propagation history defined by killer and the list of deleted values $Q$, in reverse order, from the wiped-out variable up to non-zero costs (line 9). For a deleted value, the Boolean $M(i, a) =$ true means that the deletion of $(i, a)$ is needed to explain the wipe-out. All the deleted values which are necessary to explain the final wipe-out are stored in reverse chronological order in a queue $R$ that will be used in Phase 3 later (line 13). Phase 2 then computes the maximum possible increase achievable in $c_\varnothing$, denoted as $\lambda$, and the set of EPTs to apply to $P$ in order to achieve this increase. As shown in [Cooper et al., 2010], all these EPTs move an amount of cost which is a multiple of $\lambda$. These amounts are stored in two arrays of integers: $k(j, b)$ and $k_S(j, b)$ store the number of quantum $\lambda$ that needs to be respectively projected on $(j, b)$ and extended from $(j, b)$ to $c_S$. The integer $k(S, t)$ stores the total number of quanta $\lambda$ requested on a tuple having a positive cost $c_S(\tau) \neq 0$. It is important to note that these cost moves follow a simple law of conservation (similar to flow algorithms). For any value $(j, b)$ which is not a source of cost $(c_j(b) = 0)$, the amount of cost that arrives in $(j, b)$ by Project is exactly the amount of cost that leaves $(j, b)$ by Extend (See [Cooper et al., 2010, page 465]).

$$\forall (j, b) \ s.t. \ c_j(b) = 0, k(j, b) = \sum_{c_{ij} \in C} k_{ij}(j, b) \tag{2.1}$$

Each value $(i, a)$ deleted by $c_S$ that is necessary for the wipe-out (line 11) will get costs from $c_S$ via all tuples $\tau_S$ involving $a$ (line 14), where tuples of positive costs use directly their costs to feed $(i, a)$ (hence $\lambda$ and $k_S$ are immediately updated, line 16, 17) while tuples of zero costs have to, recursively, get costs from an involved invalid value $(j, \tau[j])$ to feed $(i, a)$. If the actually defined amount of cost $k_S(j, \tau[j])$ that $(j, \tau[j])$ will extend on $c_S$, hence on $\tau$, is still not enough for $(i, a)$ (line 20), this amount needs to be recomputed (line 22).

**Algorithme 2.10 :** VAC iteration - Phase 2 and 3: Computing $\lambda$ and Applying EPTs

```
Procedure Phase2
    initialize all $k$, $k_S$ to 0, $\lambda$ to $\infty$;
    $i_0 \leftarrow$ Instrumented-AC ();
    if $i_0 = 0$ then return;
    foreach $a \in Di_0$ do
        $k(i_0, a) \leftarrow 1, M(i_0, a) \leftarrow$ true;
        if $c_{i_0}(a) \neq 0$ then
            $M(i_0, a) \leftarrow$ false, $\lambda \leftarrow \min\{\lambda, c_{i_0}(a)\}$;
    while $Q \neq \varnothing$ do
        $(i, a) \leftarrow Q$. pop();
        if $M(i, a)$ then
            $S \leftarrow$ killer$[i, a]$;
            $R$.push$(i, a)$;
            foreach $\tau \in \ell(S)$ s.t $\tau[i] = a$ do
                if $c_S(\tau) \neq 0$ then
                    $k(S, t) \leftarrow k(S, t) + k(i, a)$;
                    $\lambda \leftarrow \min\{\lambda, \frac{c_S(\tau)}{k(S,t)}\}$;
                else
                    Let $j \in S, j \neq i, j$ be a variable that invalidates $\tau$ in Bool$(P)$;
                    if $k(i, a) > k_S(j, \tau[j])$ then
                        $k(j, \tau[j]) \leftarrow k(j, \tau[j]) + k(i, a) - k_S(j, \tau[j])$;
                        $k_S(j, \tau[j]) \leftarrow k(i, a)$;
                        if $c_j(\tau[j]) = 0$ then $M(j, \tau[j]) \leftarrow$ true ;
                        else $\lambda \leftarrow \min\{\lambda, \frac{c_j(\tau[j])}{k(j,\tau[j])}\}$ ;

Procedure Phase3
    while $R \neq \varnothing$ do
        $(j, b) \leftarrow R$.pop();
        $S \leftarrow$ killer$[j, b]$;
        foreach $i \in S, i \neq j, a \in D(i)$ s.t $k_S(i, a) \neq 0$ do
            Extend$(i, a, S, \lambda \times k_S(i, a))$;
            $k_S(i, a) \leftarrow 0$;
        Project$(S, j, b, \lambda \times k(j, b))$;
    UnaryProject $(i_0, \lambda)$;
```

This also means that the total cost extended from value $(j, \tau[j])$ changes (line 21). The value $(j, \tau[j])$ in turn either needs costs from somewhere if it has an empty cost (line 23) or uses its cost otherwise (line 24). $\lambda$ is computed as the minimum of all ratios between the positive costs that will be used and the numbers of demands requested to these costs (line 8, 17, 24). This computation ensures that there will be no negative cost when applying EPTs in Phase 3.

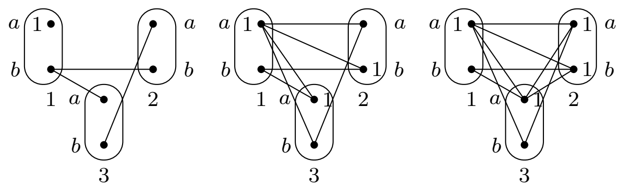
**Phase 3** The last phase of VAC is described in detail in Algorithm 2.10. It modifies the original CFN by applying the EPTs defined by the data-structures $k$ and $k_S$ identified in the previous phase on all the deleted values that have been stored in $R$. A value $(j, b)$ deleted by $\bar{c}_S$ will receive a cost of $k(j, b) \times \lambda$ by Project from $c_S$ (line 32). This requires to first extend a cost $k_S(i, a) \times \lambda$ from the invalid supports $(i, a)$ to $c_S$ so that tuple $\tau$ has positive cost to provide to $(j, b)$ (line 30). The result of this phase is a new problem $P'$, equivalent to $P$ but with an increased lower-bound $c_\varnothing$ (line 33).

### Example

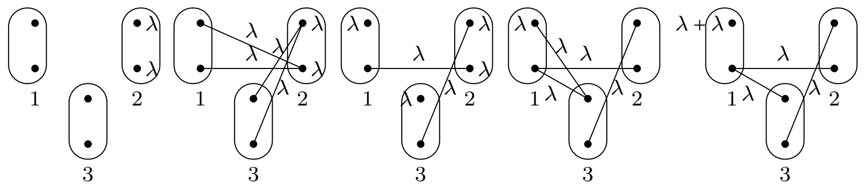
Consider the WCSP presented in the leftmost sub-figure in Figure 2.9(a). This problem has a positive unary cost $c_1(a) = 1$ represented by the number 1 beside the vertex $(1, a)$ and three positive binary costs $c_{12}(b, b) = c_{13}(b, a) = c_{23}(a, b) = 1$ represented as edges connecting the two corresponding values. The problem Bool$(P)$ can be represented as a WCSP with $m = 1$, where zero costs represent consistent elements (values and pair of values) while non-zero costs represent inconsistent elements. Thus, Bool$(P)$ has the same representation as $P$, by the leftmost sub-figure. In such a representation, propagating inconsistencies means adding "virtual' numbers beside vertex and edges. During enforcing arc consistency, as described in the middle sub-figure, the inconsistency of $(i, a)$ is firstly propagated to binary cost functions $c_{12}, c_{23}$ and then to values $(2, b), (3, a)$. Next, the inconsistency of value $(3, a)$ is propagated to $c_{23}$ and then to value $(2, a)$ (right-hand sub-figure in Figure 2.9(a)). A wipe-out is created in the domain of variable 3 and this means that the problem is not VAC.

Every step in Phase 2 is described in (Figure 2.9(b)). The task of Phase 2 is to determine which positive costs to be used in order to move an amount of cost $\lambda$ to $c_\varnothing$ via the wiped-out variable 2 (first sub-figure). Value $(2, a)$ needs a cost of $\lambda$ from $c_{23}$ (because of which it was deleted) and $(2, b)$ needs the same amount of cost from $c_{12}$ (second sub-figure). Pairs of values $((1, b), (2, b))$ and $((3, b), (2, a))$ have already positive binary costs in the WCSP $P$ for providing to $(2, b)$ and $(2, a)$ respectively. Conversely, having an empty binary cost, pairs of values $((1, a), (2, b))$ and $((3, a), (2, a))$ respectively have to get a cost of $\lambda$ from values $(1, a)$ and $(3, a)$ (third sub-figure) where value $(1, a)$ is available to be used because of its positive unary cost while value $(3, a)$ has no cost so needs to get costs from its killer $c_{31}$ (fourth sub-figure). In $c_{13}$, the pair of value $((1, a), (3, a))$ has no cost and needs an amount of cost $\lambda$ from $(1, a)$. Globally, value $(1, a)$ requires 2 demands of costs of $\lambda$ while pairs of values $((1, b), (2, b))$ and $((3, b), (2, a))$ only one (last sub-figure). The value of $\lambda$ cannot exceed $\min\{1/2, 1/1, 1/1\} = 1/2$ and this means that we can increase $c_\varnothing$ by 1/2. Such a fractional amount of cost could be managed using a decimal or rational representation of costs.

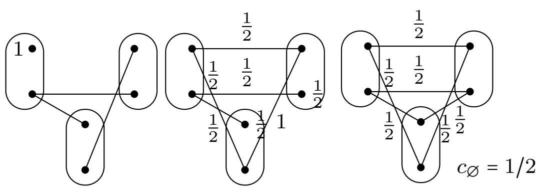
Phase 3 modifies the WCSP problem by moving costs between cost functions as defined in



(a) Phase 1: Establishing arc consistency in Bool($P$) (left-most figure, same representation as WCSP $P$)



(b) Phase 2: tracing back costs of $\lambda$ from variable 2 until non-zero costs in the original WCSP $P$



(c) Phase 3: Applying the corresponding soft arc consistency operations to $P$

*Figure 2.9: An example of establishing VAC in a WCSP [Cooper et al., 2010]*

Phase 2. First, value $(1, a)$ extends a cost of 1/2 to $c_{12}$ and then $c_{12}$ projects a cost of $\lambda$ on $(2, b)$. Similarly, value $(1, a)$ extends a cost of 1/2 to $c_{13}$ and then $c_{13}$ projects a cost of $\lambda$ on $(3, a)$ (middle sub-figure in Figure 2.9(c)). Next, value $(3, a)$ extends a costs of 1/2 to $c_{23}$ and then $c_{23}$ projects the same amount of cost on $(2, a)$. Both values of variable 2 have a unary cost equal to 1/2 and we can move a cost of 1/2 from this variable to $c_\varnothing$ (right-hand sub-figure in Figure 2.9(c)).

VAC is stronger than EDAC in the sense that if a WCSP is virtual arc consistent then establishing EDAC cannot increase the lower bound because no sequence of EPTs can increase $c_\varnothing$. This is exemplified in the previous example. The resulting WCSP (right-hand sub-figure in Figure 2.9(c)) is VAC and also EDAC.

# Chapter 3

# Dynamic virtual arc consistency

## 3.1 Introduction

Among soft arc consistencies for WCSPs, VAC is an attractive choice for Branch-and-Bound search because it can provide good lower bounds for search and it has an acceptable time complexity compared with optimal soft arc consistency (OSAC [Cooper et al., 2007, 2010]). More precisely, VAC is stronger than simple soft arc consistencies such as AC, DAC, FDAC, EDAC while it is experimentally faster than OSAC. However, the time complexity of VAC is still expensive compared with simple soft arc consistencies. Therefore, improving the efficiency in time of VAC is one of our targets.

VAC is characterized by its iterative behavior where each iteration incrementally modifies the WCSP $P$ and therefore the classic CSP Bool($P$) which is derived from $P$ by forbidding values and tuples that have costs greater or equal to zero (or any given threshold). The next iteration of VAC will enforce once again classical AC on a slightly restricted version of Bool($P$). This situation, where AC is iteratively enforced on incrementally modified versions of a constraint network, has been previously considered in Dynamic Arc Consistency algorithms [Bessière, 1991; Barták and Surynek, 2005; Mouhoub, 2003; Quéva et al., 2010] for Dynamic CSPs [Dechter and Dechter, 1988]. Suitably integrating the idea of DnAC into VAC can improve the efficiency of enforcing AC in Phase 1 and therefore the global efficiency of VAC

Moreover, the efficiency of maintaining VAC during search, just like its iterative behavior, can be accelerated by exploiting the incrementality of the changes because of branching operations. Indeed, whether an iteration of VAC has just been executed or a decision has been taken by Branch-and-Bound search, maintaining VAC requires to repeatedly enforce standard AC on the hardened version of an incrementally modified version of the problem. As a result, DnAC also can be integrated into VAC for efficiently handling the kind of changes caused by branching operations during search. Similarly to enforcing arc consistency in CSPs, revising heuristics can be introduced within Phase 1 of VAC in order to improve practical efficiency of enforcing VAC.

From this, a new method to enforce VAC has been proposed by integrating DnAC into VAC. Instead of rebuilding the CSP Bool($P$) from scratch at each new iteration during enforcing VAC or at each new node during search, the new method only handles a part of Bool($P$) that is modified according to the changes in $P$. This allows to save enforcing effort done in previous iterations or parent nodes. Inside each iteration of VAC, Bool($P$) is maintained incrementally but when the search makes a branching operation, it is either maintained incrementally or rebuilt from scratch. The former method is called "full dynamic VAC" while the latter is called "normal dynamic VAC".

This chapter consists of 3 main sections. Section 3.2 focuses on the integration of DnAC into VAC for handling incremental changes in WCSP $P$ caused by EPTs during successive iterations of VAC. We first characterize modifications in Bool($P$) that can happen between two successive iterations of VAC. Then, we propose an algorithm enforcing VAC which only modifies Phase 1 of VAC by using DnAC for updating Bool($P$). The new method still executes Phase 2, 3 in the same way as the static VAC. Finally, we will prove the correctness of our algorithm and analyze its space and time complexity.

The next section aims at integrating DnVAC into VAC for handling incremental changes between the problems considered at adjacent nodes in the search tree. We also characterize modifications in Bool($P$) according to such changes in $P$ caused by branching operations. Then, an algorithm for updating Bool($P$) in this situation will be introduced. Moreover, this section introduces some revising heuristics that can be applied within Phase 1 of VAC, in order to accelerate the enforcement of VAC.

In Section , our experimentation of static and dynamic VAC will be presented. We will compare the efficiency of static VAC and the two versions of dynamic VAC, normal and full dynamic VAC, when being maintained during search. From the obtained experimental results, the main criteria that influences the performance of dynamic VAC will be indicated.

## 3.2 Maintaining VAC dynamically during successive iterations

### 3.2.1 Specification of changes in Bool($P$)

In traditional Dynamic CSPs, DnAC algorithms are applied after each constraint removal or addition. In the case of VAC, the situation is more complex because a series of modifications of Bool($P$) occurs during Phase 3 through applications of different EPTs where each EPT leads to both increase and decrease of costs in $P$ and as a result can lead to both removal and restoration of values and tuples in the CSP Bool($P$).

A simple call to $\mathsf{Project}(c_{ij}, i, a, \alpha)$ can be decomposed in 1) an increase of cost of the unary cost function $c_i(a)$ and 2) a decrease of costs in the binary cost function $c_{ij}$. If a previously zero cost $c_i(a)$ becomes non-zero, the associated value $(i, a)$ is removed from Bool($P$) and this corresponds to a restriction. Conversely, if the non-zero cost of a pair $(a, b)$ reaches zero, this previously forbidden pair in $\overline{c}_{ij}$ becomes authorized and this corresponds to a relaxation.

Similarly, a simple call to $\mathsf{Extend}(c_{ij}, i, a, \alpha)$ can be decomposed in 1) a decrease of cost of the unary cost function $c_i(a)$ and 2) an increase of costs in the binary cost function $c_{ij}$. If a previously non-zero cost $c_i(a)$ reaches zero, the previously forbidden value $(i, a)$ becomes authorized in Bool($P$) and this corresponds to a relaxation. Conversely, if the zero cost of a previously authorized pair $(a, b)$ becomes non-zero, this pair becomes forbidden in $\overline{c}_{ij}$ and this corresponds to a restriction.

Therefore, instead of applying a DnAC algorithm inside each Project, Extend and UnaryProject operation, a better approach consists in applying DnAC principles only after Phase 3 to avoid useless restorations/deletions of values by DnAC.

Each iteration of VAC transforms the current CFN $P$ into a modified problem $P'$ with cost functions $c'_i$ and $c'_{ij}$. After Phase 2, it is already possible to compute the values of $c'_i$ and $c'_{ij}$ because they are defined by a known sequence of applications of Project, Extend and UnaryProject on $c_i$ and $c_{ij}$. For example, if $i$ is not the wiped-out variable, we have for any value $a$:

$$c'_i(a) = c_i(a) + k(i,a).\lambda - \sum_{c_{ij} \in C} k_{ij}(i,a).\lambda$$

A similar computation can be done for any value $a$ of the wiped-out variable for which a cost of $\lambda$ shifted to $c_\varnothing$ must be taken into account.

$$c'_i(a) = c_i(a) + k(i,a).\lambda - \sum_{c_{ij} \in C} (k_{ij}(i,a).\lambda) - \lambda$$

For a pair of values $((i,a),(j,b))$, if one of the two values is not deleted or the deletion of one value is not necessary for the wipe-out, $c_{ij}$ is not changed. Otherwise, we have:

$$c'_{ij} = c_{ij} + (k_{ij}(i,a) + k_{ij}(j,b) - k(ij,ab)).\lambda$$

where $k(ij,ab)$ and $k_{ij}(j,b)$ are the simplified notations for $k(S,\tau)$ and $k_S(j,b)$ in the case of binary cost functions $S = \{i,j\}$, $\tau = (a,b) \in D_i \times D_j$ which respectively store the total number of quantum $\lambda$ that the pair of values $(i_a, j_b)$ receive from every where and from value $(i,b)$.

We now show that the global effect of all EPTs on Bool$(P)$ in Phase 3 is a set of relaxations only, at the unary and binary levels.

**Property 1** *Following Phase 2, we know that:*

*a)* $\forall(i,a)$*:* $c'_i(a) \leq c_i(a)$.

*b)* $\forall(i,a)$ *and* $(j,b)$*: if* $c'_{ij}(a,b) \neq c_{ij}(a,b)$ *then* $(i,a)$ *or* $(j,b)$ *is deleted in the current justified partial AC–closure of Bool(P).*

**Proof 1** *a) In VAC, the only operation that may increase unary costs is the* Project *operation. However, according to equation 2.1 in subsection 2.2.7, any value* $(i,a)$ *that receives cost by* Project *will later* Extend *the same amount of cost (to other binary cost functions or to* $c_\varnothing$*). Hence, unary costs cannot increase.*

*b) The only way for a binary cost* $c_{ij}(a,b)$ *to change is by a* Project *from* $c_{ij}$ *or an* Extend *onto it. However, Phase 3 of VAC applies* Project *and* Extend *to values extracted from the queue* $R$ *of deleted values (built by Phase 2). Therefore when the cost of a pair* $(a,b)$ *changes, either* $(i,a)$ *or* $(j,b)$ *must have been deleted.*

**Corollary 1** *The EPTs applied in Phase 3 of VAC, transforming Bool*$(P)$ *into Bool*$(P')$*, generate only the following types of relaxations:*

*a) values* $(i,a)$ *that become authorized* $(c_i(a) > c'_i(a) = 0)$.

*b) pairs* $((i,a),(j,b)$ *that become authorized* $(c_{ij}(a,b) > c'_{ij}(a,b) = 0)$.

**Proof 2** *a) From Property 1(a), we know that unary costs may only decrease. Some may therefore go from a non-zero cost to a zero cost. Therefore the corresponding value re-appears in Bool*$(P')$. *This can be considered as the retraction of a unary constraint.*

*b) From Property 1(b), the costs of pairs may either increase or decrease. If a binary cost* $(a,b)$ *increases from zero to non-zero, this cannot destroy a valid support because either of the 2 values is deleted in the current partial closure. The support cannot be valid. If the cost of* $(a,b)$ *decreases however, it may create a new support for* $a$ *or* $b$.

### 3.2.2 Algorithm

During successive iterations of VAC, no restriction can be created in Bool$(P)$. Therefore, the DnAC algorithm used can be specialized for relaxations, as described in Algorithm 3.1. This procedure is performed right after Phase 2, before Phase 3, in order to update Bool$(P)$ according to the changes in $P$ that will occur in Phase 3 because of EPTs. Since Bool$(P)$ is maintained after Phase 3 of each iteration, so $\overline{D}(i)$ mentioned in the following represents the domain of variable $i$ in the final justified partial AC−closure obtained after Phase 1. The restoration protocol consists of 3 stages: initialization, propagation, filtering as in AC/DC−2. It is executed based on killer which is equivalent to the justification system used in AC/DC−2. Similarly to AC/DC, for each variable $i$, it uses a list *restored*$[i]$ to store values that have just been restored in Bool$(P)$ and are waiting for being propagated.

The **initialization stage** scans all the deleted values in the queue $R$ to identify which values should be restored (line 2). Note that each value in $R$ was deleted due to the loss of support on some constraint and its deletion is necessary for the wipe-out. Thus, the wiped-out variable $i_0$ is processed separately because some values in $D(i_0)$ could be deleted by itself because of its previous unary cost (line 7). As Corollary 1 shows, there are 2 possible cases: (1) when a value $(i,a)$ becomes authorized $c_i(a) > c'_i(a) = 0$, it will be restored (line 5), (2) when a new valid support appears for a value $(j,b)$ by satisfying $(c_{ij}(a,b) + c_i(a)) > (c'_{ij}(a,b) + c'_i(a)) = 0$ and killer $(j,b) = i$, $(j,b)$ will be restored (line 6). The restoration of values is executed by Procedure Restore at line 8. When a value $(i,a)$ is restored, it is stored in array *restored*$[i]$ and variable $i$ is kept in a list $RL$ for future propagation.

The **propagation stage** (line 12) propagates value restorations to direct neighbors of the variables whose domain has been extended, as in AC/DC−2. The propagation queue $RL$ contains variables having extended domains. Each such variable $i$ can restore a value $(j,b)$ in a neighbor variable $j$ if it was deleted due to $c_{ij}$ (line 16) and is now supported by a restored value in $i$ (line 17). After propagating all restored values, the *restored* list is emptied (line 19) to avoid re-propagating values which have been already propagated.

The **filtering stage** must eliminate the restored values $(i,a)$ which are not arc consistent on some constraint $\overline{c}_{ij}$ and properly set the associated killer $(i,a)$ to $j$. This is precisely what is achieved by the Phase 1 of VAC. Hence, we integrated this stage into Phase 1 by adding the neighbor variables of variables having restored values into the revision propagation queue $Q_{AC}$ used in Phase 1 of VAC (line 20).

In summary, in order to maintain VAC dynamically, the DynVAC algorithm performs Phase 1, 2, 3 in the same way as the static VAC algorithm, except for the extra work, done by Algorithm 3.1, that is inserted between Phase 2 and Phase 3 of VAC for updating Bool($P$).

**Algorithme 3.1 :** Algorithm updating Bool($P$) during successive iterations of VAC

```
Procedure Initialization
    foreach $(j,b) \in R$ do
        $i \longleftarrow$ killer $[j,b]$ ;
        foreach $a \in D(i) - \overline{D}(i)$ do
            if $(c_i(a) > 0) \wedge (c'_i(a) = 0)$ then Restore$(i,a)$ ;
            if $b \notin \overline{D}(j) \ \wedge \ c'_i(a) = 0 \ \wedge \ c'_{ij}(a,b) = 0$ then Restore$(j,b)$ ;
    foreach $a \in Di_0$ s.t. $c_{i_0}(a) > 0 \ \wedge \ c'_{i_0}(a) = 0$ do Restore$(i_0,a)$ ;
Procedure Restore$(i,a)$
    add $a$ into $\overline{D}(i)$ and restored$[i]$;
    add $i$ into $RL$;
    killer $[i,a] \leftarrow$ nil;
Procedure Relax−Propagation
    while $RL \neq \varnothing$ do
        $i \leftarrow RL$.pop();
        foreach $c_{ij} \in C$ do
            foreach $b \in D(j) - \overline{D(j)}$ s.t. killer $[j,b]= i$ do
                if $\exists a \in$ restored$[i]$ s.t. $c'_{ij}(a,b) = 0$ then
                    Restore$(j,b)$;
        restored$[i] \leftarrow \varnothing$;
        $Q_{AC} \leftarrow Q_{AC} \cup \{j \mid \overline{c}_{ij} \in \overline{C}\}$;
```

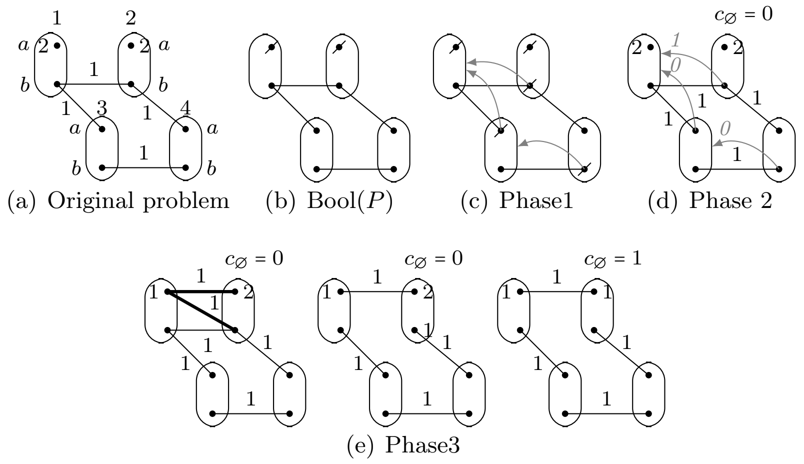
### 3.2.3 Example

The essential gain of DynVAC compared to VAC lies in the fact that the list of variables to revise during Phase 1 is not reset to the full set of variables $X$ at each iteration but is instead maintained along all iterations, avoiding useless repeated filtering. We illustrate this on a small example.

Consider the binary CFN in Figure 3.1(a). Each variable has two values $a$ and $b$ represented as vertices. Non-zero unary costs are displayed beside values. An edge between two vertices indicates that the corresponding pair has a non-zero binary cost. Zero costs are not represented. In Bool($P$) (Figure 3.1(b)), forbidden values are shown as crossed-out and edges represent *forbidden* pairs.

Suppose that the revision order in Phase 1 is ($c_{13}$, $c_{34}$, $c_{12}$, $c_{24}$). After revising $c_{13}$, $c_{34}$, $c_{12}$, $(3,a)$, $(4,b)$ and $(2,b)$ have been deleted from Bool($P$) respectively. Phase 1 stops because variable 2 has wiped-out (Figure 3.1(c)). The gray arrows represent the state of the killer data-structure for removed values, pointing to the variable that offered no valid support.

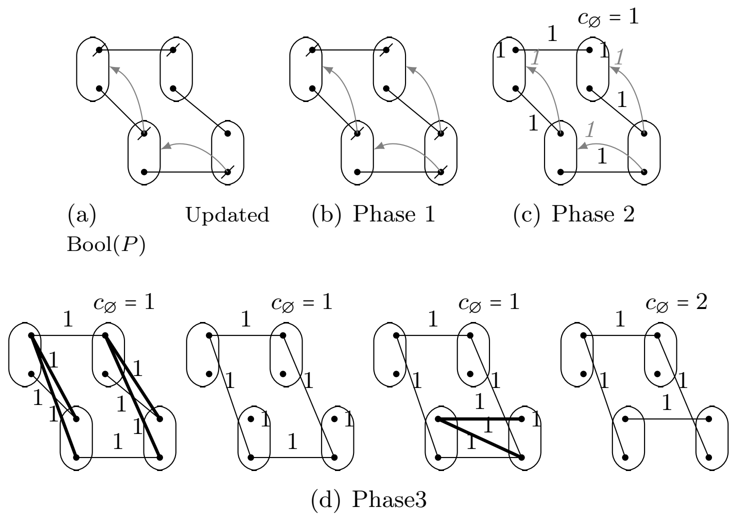
A removed value without any justification arrow means that the value is deleted because of its own positive unary cost.



(a) Original problem (b) Bool($P$) (c) Phase1 (d) Phase 2

(e) Phase3

*Figure 3.1: DynVAC: Iteration 1*



(a) Updated Bool($P$) (b) Phase 1 (c) Phase 2

(d) Phase3

*Figure 3.2: DynVAC: Iteration 2*

In Phase 2 (Figure 3.1(d)), the deletion of $(2,b)$ alone is sufficient for the wipe-out. It uses the non-zero costs $c_{12}(b,b)$ and $c_1(a)$ to provide $c_\varnothing$ with a maximal amount of cost $\lambda = 1$. The numbers in italic associated with gray arrows precisely indicate the corresponding value of $k(i,a)$. Applying identified EPTs, Phase 3 (Figure 3.1(e)) transforms $P$ into an equivalent problem $P'$ with $c'_\varnothing = 1$. Extended costs are shown in bold.

VAC enforcing continues because $P'$ is still not VAC. To update Bool($P$) in Figure 3.1(c), we consider $((1,a),(2,a),(2,b))$ for restoration because only $c_{12}$ has been modified by EPTs in Phase 3. Only $(2,b)$ is restored because it has a zero cost and a support $(b,b)$ on $c_{12}$. This restoration does not lead to further restorations. The constraints of the updated Bool($P$) are directly defined by $P'$. In fact, the updated result in Figure 3.2(a) is already a justified partial AC–closure of Bool($P'$) with two extra deleted values $(3,a)$ and $(4,b)$ and the associated killer. The next Phase 1 starts from this updated Bool($P$). $(4,a)$ is removed after revising $c_{24}$ and variable 4 wipes out (Figure 3.2(b)). Phase 2 and Phase 3 perform as in the previous iteration. The final problem (Figure 3.2(d)) with $c''_{\varnothing} = 2$ is VAC.

### 3.2.4 Correctness of the algorithm

In this section, we use dedicated notations: given two CSPs $P = (X, D, C, 1)$ and $P' = (X, D', C, 1)$, we write $P \subseteq P'$ if every domain $D(i) \in D$ is included in the corresponding domain $D'(i) \in D'$. We will say that $(i,a) \in P$ if $a$ is present in $D(i)$ in $P$, and that $(i,a) \notin P$ otherwise.

The DynVAC algorithm is correct if the result it provides to the next iteration $(t{+}1)$ of VAC is a justified partial AC–closure of Bool($P^{t+1}$), where $P^t$ and $P^{t+1}$ (with cost functions $c^t$ and $c^{t+1}$) are respectively the initial WCSPs on which VAC is enforced at the iterations $t$ and $t+1$. This result is entirely defined by 1) the values restored by the relaxation initialization and propagation, 2) the contents of the killer data-structure and of the $Q_{AC}$ queue given to Phase 1.

We denote by $Bool_{JPAC}(P^t)$ the justified partial AC–closure of Bool($P^t$) obtained at the end of the phase 1 of iteration $t$ and by $\text{Bool}_{_{Dyn}}(P^t)$ the CSP obtained after value restoration by DynVAC in the new Phase 3, with the constraints of Bool($P^{t+1}$). These different WCSPs and CSPs are represented in Figure 3.3.

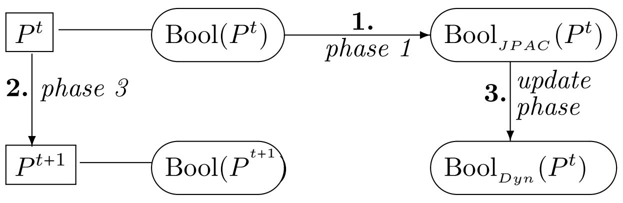

$$\begin{tikzcd}
\boxed{P^t} \arrow[r, dash] \arrow[d, "\textbf{2.}\ \textit{phase 3}"] & \text{Bool}(P^t) \arrow[r, "\textbf{1.}", "\textit{phase 1}"'] & \text{Bool}_{_{JPAC}}(P^t) \arrow[d, "\textbf{3.}"', "\textit{update phase}"] \\
\boxed{P^{t+1}} \arrow[r, dash] & \text{Bool}(P^{t+1}) & \text{Bool}_{_{Dyn}}(P^t)
\end{tikzcd}$$

*Figure 3.3: The different CFNs and CSPs built in one iteration of DynVAC*

In order for DynVAC to be correct, the problem $\text{Bool}_{_{Dyn}}(P^t)$ and the associated killer data-structure must satisfy the following property to be a *justified partial AC–closure* of Bool($P^{t+1}$):

(1) $\text{Bool}_{_{Dyn}}(P^t) \subseteq \text{Bool}(P^{t+1})$. Indeed, $\text{Bool}_{_{Dyn}}(P^t)$ should not authorize any value which is not authorized in Bool($P^{t+1}$).

(2) $\forall (i,a)$ s.t. $(i,a) \in \text{Bool}(P^{t+1})$ and $(i,a) \notin \text{Bool}_{_{Dyn}}(P^t)$, $(i,a)$ can be deleted by enforcing AC in Bool($P^{t+1}$) using the sequence of EPTs represented by the associated killer structure in $\text{Bool}_{_{Dyn}}(P^t)$.

(3) For any other value $(i,a)$ in $\text{Bool}_{_{Dyn}}(P^t)$, either it is arc consistent in $\text{Bool}_{_{Dyn}}(P^t)$ or all variables neighboring to $i$ in $Q_{AC}$ will be revised.

**Proof 3** *(1) By definition of Bool($P$), $(i,a) \notin$ Bool($P^{t+1}$) $\Leftrightarrow c^{t+1}(i,a) > 0$. By Corollary 1, this means that $c^t(i,a) > 0$ and $(i,a) \notin$ Bool($P^t$). Since AC enforcing can only delete values, $(i,a) \notin Bool_{JPAC}(P^t)$. Then $(i,a)$ cannot be restored because of its strictly positive cost. Thus, $(i,a)$ is absent from $Bool_{_{Dyn}}(P^t)$.*

*(2) A value $(i,a) \notin Bool_{Dyn}(P^t)$ must have been deleted in $Bool_{JPAC}(P^t)$ with* killer$(i,a) = j$ *and is not restorable. Either it is a direct deletion because $(i,a)$ had no valid support in Bool$(P^t)$, i.e. $\forall b \in D(j), c^t_j(b) + c^t_{ij}(a,b) > 0$. Since $(i,a)$ has not been restored, this means that $\forall b \in D(j), c^{t+1}_j(b) + c^{t+1}_{ij}(a,b) > 0$ and $(i,a)$ can be deleted immediately in Bool$(P^{t+1})$ by revising for AC for $i$ w.r.t $j$. Otherwise, $(i,a)$ has valid supports in Bool$(P^t)$ but they have been previously deleted in Phase 1 and not restored. Therefore, the same argument as above applies inductively. Since AC is a sequential process and the number of values is finite, ultimately the previous case must apply and the value $(i,a)$ can therefore be deleted in Bool$(P^{t+1})$ too by enforcing AC in it.*

*(3) Any other value $(i,a)$, i.e. $(i,a) \in Bool_{Dyn}(P^t)$, is either restored by DynVAC (by Procedure* Restore *at line 8, Algorithm 3.1) and in this case $Q_{AC}$ has been updated to enforce revision of $i$ (line 20 ) or else it was present in $Bool_{JPAC}(P^t)$, a justified partial AC–closure of Bool$(P^t)$. Then $(i,a)$ had all neighbors in $Q_{AC}$ and still has them. Else, it has valid supports on all constraints. By Corollary 1, these valid supports are preserved.*

### 3.2.5 Complexity

DynVAC uses an additional data structure *restored* in comparison to static VAC. The maximum total size of this data structure is $nd$ because each value is restored at most once. DynVAC has therefore the same space complexity as VAC, in $O(ed + nd)$ or $O(ed)$.

The update initialization stage has a $O(nd)$ time complexity because there are at most $n \times d$ values to be restored (line 5,6 Algorithm 3.1). The worse-case time complexity of the propagation stage is $O(ed^2)$ because each restored value is propagated exactly once and each pair of values involved in a constraint is thus tested at most once (line 16,17 Algorithm 3.1). So, the time complexity to update Bool$(P)$ is $O(ed^2)$, which does not modify the complexity of Phase 3 of VAC.

An iteration of DynVAC has therefore a $O(ed^2)$ time complexity, similarly to VAC, as long as an optimal AC algorithm is used in Phase 1. Although DynVAC does not have an improved asymptotic complexity compared to VAC, experimental tests presented in the next section will show important speedups in practice.

## 3.3 Maintaining VAC dynamically during search

### 3.3.1 Specification of changes in Bool$(P)$

The standard way to maintain VAC during search is to rebuild the CN Bool$(P)$ at each new node and then use this new Bool$(P)$ to enforce VAC in the CFN $P$. If DynVAC has been shown to enhance VAC efficiency for pre-processing CFNs at the root node, DynVAC has never been maintained during search.

Similarly, the simplest way to maintain Dynamic VAC during search consists in enforcing Dynamic VAC at each node. However, maintaining Dynamic VAC during search offers new opportunities for incrementality. In this section, we show that VAC can be incrementally maintained not only between successive iterations of VAC inside a node, but also during search between nodes, by incrementally maintaining an AC–closure of Bool($P$) between search nodes.

Between two consecutive nodes, the CN Bool($P$) is only modified according to the changes in $P$ caused by branching operations. The branches out of a node can be the assignment of a variable $(i = a)$ or a domain restriction $(i \neq a)$, $(i < a)$ or $(i > a)$ that respectively remove value $a$, values before and after $a$ from the domain $D(i)$. We denote all these different cases as $P_{|i}$. We expect that maintaining AC with a Dynamic AC algorithm in such a slightly modified Bool($P$) will be beneficial compared to a cold restart. Suppose that branching operations transform the current CFN $P$ into $P_{|i=a}, P_{|i\neq a}, P_{|i>a}$ or $P_{|i<a}$. After a branching operation, $P_{|i}$ will have modified domains and also possibly modified cost functions. These will be respectively denoted as $D'$ and $c'_i$ or $c'_{ij}$. To enforce VAC incrementally between nodes, we need to compute the AC–closure of Bool($P_{|i}$) from the AC–closure of Bool($P$) using only the modification from $P$ to $P_{|i}$.

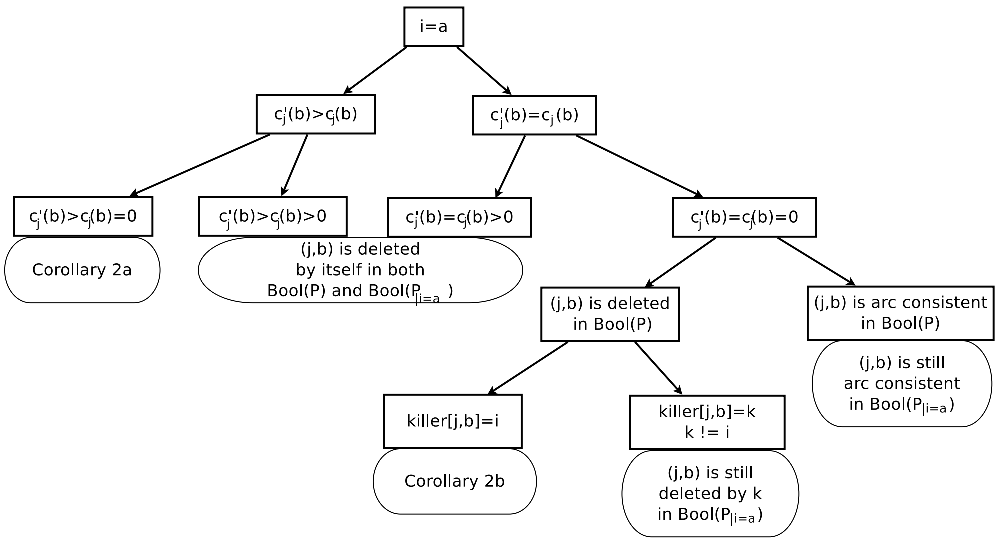

i=a
c'j(b)>cj(b)
c'j(b)=cj(b)
c'j(b)>cj(b)=0
c'j(b)>cj(b)>0
c'j(b)=cj(b)>0
c'j(b)=cj(b)=0
Corollary 2a
(j,b) is deleted by itself in both Bool(P) and Bool(P|i=a )
(j,b) is deleted in Bool(P)
(j,b) is arc consistent in Bool(P)
(j,b) is still arc consistent in Bool(P|i=a)
killer[j,b]=i
killer[j,b]=k k != i
Corollary 2b
(j,b) is still deleted by k in Bool(P|i=a)

*Figure 3.4: All possible situations that can happen after a variable assignment*

When a variable domain $D(i)$ is restricted by a branching operation of the form $(i \neq a)$, $(i < a)$ or $(i > a)$ that does not reduce the domain to a singleton, the removal of values leads to a new domain $D'(i)$ and thus a new domain, denoted as $\overline{D'}(i)$, in the currently computed AC–closure of Bool($P_{|i}$). Some values of the neighboring variables $j$ may have lost their support on $\overline{c'}_{ij}$ in Bool($P_{|i}$) and thus need to be checked for AC. Similarly to the case of constraint restriction, this can be naturally achieved by DnAC by (1) adding the neighbor variables of $i$ in the propagation queue $Q_{AC}$ and (2) restarting propagation to remove all values which are arc inconsistent in Bool($P_{|i}$) due to this domain restriction.

When instead, an assignment is performed (or equivalently the domain is reduced to a singleton), solvers exploit this very specific situation to directly eliminate the variable $i$ from the problem. Therefore, the modifications caused by a variable assignment $(i = a)$ are more complex than for other domain reductions. Indeed, when the domain of a variable $i$ is restricted to a single value $a$, the unary cost $c_i(a)$ of $(i, a)$ is immediately projected on $c_\varnothing$. Then binary costs $c_{ij}(a, b)$ also are projected on neighboring values $(j, b)$ of $i$ and the binary cost functions $c'_{ij}$ are removed from the CFN $P_{|i=a}$.

**Property 2** *The CFN* $P_{|i=a} = (X, D', C', m)$ *which is obtained from* $P = (X, D, C, m)$ *by* *a variable assignment $(i = a)$ satisfies the following properties:*

*a) $c'_i(a) = 0$.*

*b) for every variable $j$, there is no cost function $c'_{ij}$ connecting $i$ and $j$.*

*c) for every value $(j, b)$ such that $\exists c_{ij}$ in $P$, we have $c'_j(b) = c_j(b) + c_{ij}(a, b)$.*

Figure 3.4 lists all the situations that can happen after a variable assignment $(i = a)$ for the values $(j, b)$ of a neighbor variable $j$. It shows that, except for the two cases that will be considered later in Corollary 3, the deleted/non deleted status of $(j, b)$ remains unchanged in Bool($P_{|i=a}$). To show this, we first prove the following Corollary of Property 2:

**Corollary 2** *The CN Bool($P_{|i=a}$) satisfies the following properties:*

*a) for every variable $j \neq i$, $j$ is arc consistent with $i$.*

*b) $(i, a)$ is an arc consistent value in Bool($P_{|i=a}$).*

**Proof 4** *a) The constraints of Bool($P_{|i=a}$) are directly defined from the cost functions of $P_{|i=a}$. From Property 2b), we know that there does not exist any $c'_{ij}$ in $P_{|i=a}$ where $j \neq i$. Therefore, there also does not exist any constraint $\overline{c'}_{ij}$ in Bool($P_{|i=a}$). As a result, $j$ is arc consistent with variable $i$.*

*b) In Bool($P_{|i=a}$), $(i, a)$ cannot be deleted because of its positive cost since $c'_i(a) = 0$ according to Property 2a). Moreover, $(i, a)$ cannot be killed by any other variable $j$ because no constraint links $i$ to other variables in Bool($P_{|i=a}$).*

Considering the various cases where the status of $(j, b)$ does not change, most are straightforward. The only non obvious case is when $c'_j(b) = c_j(b) = 0$ and $(j, b)$ is arc consistent in Bool($P$). This implies that $(j, b)$ is arc consistent with every variable $k \neq i$ in Bool($P$). Because the variable assignment does not change cost functions $c_{jk}$, $(j, b)$ will still be arc consistent with $k$ in Bool($P_{|i=a}$). In addition, Corollary 2a) indicates that $(j, b)$ is also arc consistent with $i$ in Bool($P_{|i=a}$). Therefore, being arc consistent with every variable and having a zero unary cost, $(j, b)$ is an arc consistent value in Bool($P_{|i=a}$) (and it has the same status as in Bool($P$)).

Then, as shown in Figure 3.4, there are two cases where the status of values may change compared to the status in Bool($P$), requiring to update Bool($P_{|i=a}$). This is proved in Corollary 3. In all other cases, as we saw, the status of values in Bool($P_{|i=a}$) remains the same as in Bool($P$).

**Corollary 3** *Each variable assignment $(i = a)$ in $P$ can generate both:*

*a) the removal of values $(j, b)$ in Bool($P_{|i=a}$) such that there exists $c_{ij}$ in $P$ and $c'_j(b) > c_j(b) = 0$.*

*b) the restoration of values $(j, b)$ in Bool($P_{|i=a}$) which were deleted in Bool($P$) by $\overline{c}_{ij}$ and such that $c'_j(b) = 0$.*

**Proof 5** *a) From Property 2c), unary costs of values $(j, b)$ that are neighbor with $i$ in $P$ may only increase or remain unchanged in $P_{|i=a}$. Some may therefore go from a zero to a non-zero cost. In this case, the corresponding values are deleted in Bool($P_{|i=a}$). From the dynamic AC point of view, this can be considered as a restriction, with the addition of a unary constraint on $j$ (i.e., domain restriction).*

*b) If $(j, b)$ is a value removed in Bool$(P)$ because of $\bar{c}_{ij}$ (*$\mathsf{killer}(j, b) = i$*), this implies that $c_j(b) = 0$ (otherwise, we would have* $\mathsf{killer}(j, b) = j$*). According to Property 2c), the unary cost of $(j, b)$ in $P_{|i=a}$ can remain unchanged, i.e., $c'_j(b) = 0$, if $c_{ij}(a, b) = 0$. Such a value with zero cost cannot be killed by itself in Bool$(P_{|i=a})$. Furthermore, $i$ is no longer a reason to delete $(j, b)$ because as Corollary 2a) states, $(j, b)$ is arc consistent with $i$. Hence, $(j, b)$ can become viable in Bool$(P_{|i=a})$. From the dynamic AC point of view, this corresponds to the relaxation of a unary constraint on $j$ (domain relaxation) and $(j, b)$ needs to be considered as restorable.*

In summary, the change in a cost function network $P$ caused by variable assignments can lead to both the restoration and the removal of values in the CN Bool$(P)$, while domain restrictions only lead to the removal of values. For these two cases of branching operations during search, DnAC can be used for maintaining a *justified partial* AC–closure of such modified Bool$(P)$. This will be showed in an algorithm presented in the next section.

### 3.3.2 Algorithm

**Algorithme 3.2 :** Algorithm updating Bool$(P)$ w.r.t branching operations during search

```
1  Procedure restrict($i, a$, operation)
2      switch operation do
3          case operation is ">"
4              remove values smaller or equal to $a$ from $D(i)$ and $\overline{D}(i)$
5          case operation is "<"
6              remove values greater or equal to $a$ from $D(i)$ and $\overline{D}(i)$
7          otherwise remove $a$ from $D(i)$ and $\overline{D}(i)$;
8      $Q_{AC} \leftarrow \{i\}$;
9      Instrumented-AC();
10 Procedure assign($i, a$)
11     $D(i) = \overline{D}(i) = \{a\}$;
12     UnaryProject($i$);
13     foreach $c_{ij}$ do
14         foreach $b \in D(j)$ do
15             if $c_j(b) = 0$ and $c_j(b) + c_{ij}(a, b) > 0$ then
16                 remove $b$ from $\overline{D}(j)$;
17                 add $j$ into $Q_{AC}$;
18             $c_j(b) \leftarrow c_j(b) + c_{ij}(a, b)$;
19             if $c_j(b) = 0$ and killer$(j, b) = i$ then
20                 restore$(j, b)$;
21         remove $c_{ij}$ and $\bar{c}_{ij}$;
22     Relax–Propagation() ;
23     Instrumented-AC();
```

Branching operations in search can provoke two processes of DnAC: restriction (value removals) and relaxation (value restorations). Therefore, DnAC needs to use two dedicated queues $Q_{AC}$ and $RL$, the former for propagating value removals and the second for value restorations. Algorithm 3.2 presents two procedures: "restrict" and "assign" for handling respectively the domain restriction and the variable assignment in $D(i)$.

Procedure restrict$(i, a,$ operation$)$ at line 1 is used to update $P$ and Bool$(P)$ when restricting the domain $D(i)$ except for the restriction to a singleton value which is considered as a variable assignment. As analyzed in the previous section, there is only value removal for this case. Variables neighbor to $i$ can loose supports in $D(i)$ and thus need to be revised for AC. This naturally can be done by an instrumented arc enforcing procedure (line 9) with an initial propagation queue $Q_{AC}$ containing $i$ - the variable with reduced domain (line 8).

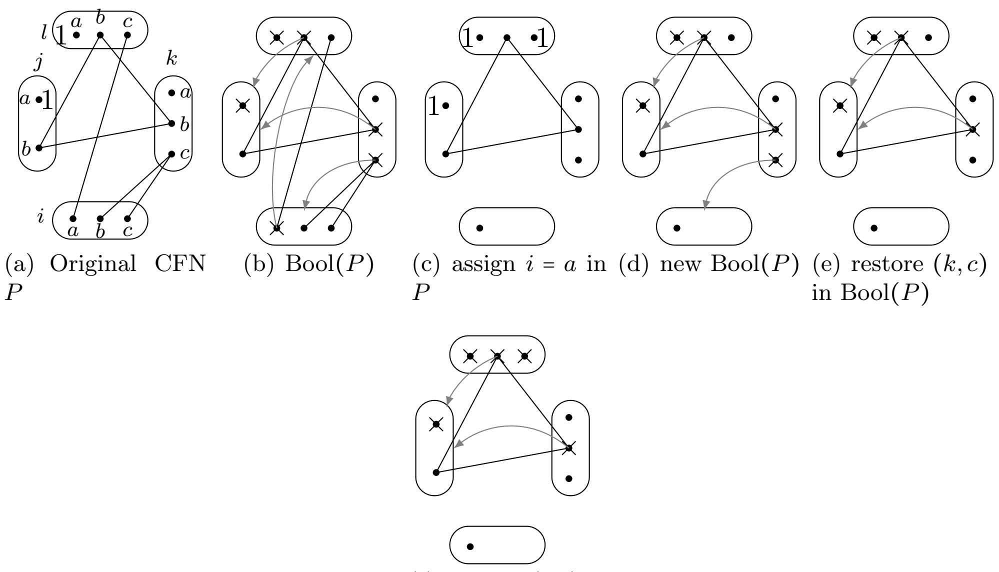
Procedure assign$(i, a)$ at line 10 is used to update $P$ and Bool$(P)$ when assigning $(i = a)$. The unary cost of $(i, a)$ is projected on $c_\varnothing$ (line 12). For every variable $j$ neighbor to $i$, binary costs of pairs of values $((i, a), (j, b))$ are projected on $c_j(b)$ (line 18). New values of non-zero cost (line 15) will be removed from Bool$(P)$ (line 16). The removal of $(j, b)$ can further lead to the removal of other values. Thus, $j$ is pushed into queue $Q_{AC}$ for the future revision for AC (line 17). Conversely, if $(j, b)$ has been removed by $i$ and still has a zero cost (line 19), it is restorable and will be restored (line 20). After defining all values to be removed and restored because of the value assignment for $i$, the algorithm will do the propagation for value restorations and then for value removals (lines 22 and 23 respectively). Note that the propagation for value restorations is performed by Procedure Relax−Propagation() defined in Algorithm 3.1 where the condition $c'_{ij}(a, b) = 0$ at line 17 is replaced by $c_{ij}(a, b) = 0$. The reason for this replacement is that the new CSP $P$ after branching operations have been redefined before launching the propagation of value restorations. We can thus use directly new costs $w$ of the newly defined problem $P$.

When the search backtracks, the AC−closure of Bool$(P)$ can be simply rebuilt via the restoration of the justification system "killer" through trailing. Indeed, if killer$(i, a) = nil$, this means that $(i, a)$ was consistent in the old Bool$(P)$. In this case, $(i, a)$ will be set as an available value in $\overline{D}(i)$. Conversely, if killer$(i, a) = j$ or $i$, this means that $(i, a)$ was removed because of constraint $\overline{c}_{ij}$ or $\overline{c}_i$. In this case, $(i, a)$ will be removed from $\overline{D}(i)$.

### 3.3.3 Example

Consider the CFN $P$ in Figure 3.5(a) Suppose that $P$ is the CFN obtained at some node of the tree search and is therefore virtual arc consistent. The AC−closure of Bool$(P)$ for this node is represented in Figure 3.5(b). Both $P$ and Bool$(P)$ are represented in the same way as in Figure 3.1 in Example 3.2.3. Now we will show how Bool$(P)$ is updated in two cases: a domain restriction $(i \neq a)$ and a variable assignment $(i = a)$.

**Domain restriction.** In the case $(i \neq a)$, DynVAC will propagate the domain reduction of $i$ to neighbor variables $k, l$. We observe that $(i, a)$ was also deleted in Bool$(P)$ of the parent node. It did not support any value. Thus, the removal of $(i, a)$ does not lead to any support loss. The propagation procedure stops here. We obtain a new Bool$(P)$, which is identical to the old one in Figure 3.5(b). In this favorable case, DynVAC has almost nothing to do to construct Bool$(P_{|i \neq a})$.



(a) Original CFN $P$ (b) Bool($P$) (c) assign $i = a$ in $P$ (d) new Bool($P$) (e) restore $(k, c)$ in Bool($P$)

(f) remove $(l, c)$ from Bool($P$)

*Figure 3.5: Updating Bool($P$) in the case of a variable assignment*

**Variable assignment.** Consider now the case of a variable assignment $(i = a)$. Only the pair $((i, a), (l, c))$ has positive cost and this binary cost will be projected to $(l, c) : c'_l(c) = 1$. Then cost functions $c_{ik}, c_{il}$ are removed from $P$. The network of $P_{|i=a}$ is presented in Figure 3.5(c). Similarly, following assignment, values $(i, b), (i, c)$ and constraint $\overline{c}_{ik}, \overline{c}_{il}$ will be removed from Bool($P$) and value $(i, a)$ will be set as consistent in Bool($P_{|i=a}$) (Figure 3.5(d)). In the neighborhood of $i$, only value $(k, c)$ is restorable because it was removed due to $\overline{c}_{ij}$ but $\overline{c}_{ij}$ has just been removed from Bool($P$). Thus, $(k, c)$ is restored (Figure 3.5(e)). The restoration of $(k, c)$ does not make any other value restorable. At the same time, in the neighborhood of $i$, only value $(l, c)$ has an increased unary cost. $(l, c)$ will be deleted in Bool($P_{|i=a}$) because of its positive cost (Figure 3.5(f)). The domain of $l$ becomes empty and VAC can project unary costs of 1 to $c_\varnothing$. We observe that the result of enforcing AC on the sub-problem of Bool($P$) defined by variables $j, k, l$ (for constraints $\overline{c}_{jl}, \overline{c}_{jk}, \overline{c}_{kl}$) is preserved and that a variable wipe-out is detected early.

### 3.3.4 Revision heuristics

One of the possible methods for improving the efficiency of enforcing AC in classical CSPs is to use revision heuristics and this idea can be applied in Phase 1 of VAC in order to improve the efficiency of DynVAC. Phase 1 of VAC terminates as soon as a variable domain in Bool($P$) is wiped-out if the problem is not virtual arc consistent or all variables in the propagation queue $Q_{AC}$ have been revised otherwise. We observe that:

- In general, there are more chances for variables of small domain size than those of large domain size to become wiped-out when being enforced for AC. Thus, revising such variables of small domain size first may lead a wipe-out earlier.
- When processing a variable $j$ popped out from the propagation queue $Q_{AC}$, all neighbors of $j$ have to be checked for AC w.r.t $j$ and one of these neighbors can become

wiped-out. Processing variables having more neighbors, i.e. large degree; has more chances to lead to a wipe-out.

- In Phase 3 of VAC, a variable $j$ provides its unary costs for a variable $i$ that is neighbor to $j$ if some values in $i$ were deleted because of $i$ in Phase 1 and these deletions are necessary for the wipe-out. The larger unary costs in $D(i)$ are, the larger costs can be moved to the neighbors of $i$. According to this rule, we hope that more costs can be moved to the wiped-out variable $i_0$ and thus to $c_\varnothing$.

From the observations above, we have proposed to use revision heuristics during enforcing AC in Phase 1 as follow:

- domain size (domsize): this heuristic selects first in $Q_{AC}$ the variable having the smallest current domain size in Bool$(P)$. Then, constraints are processed in ascending order of the domain sizes of the opposite variables.
- maximum unary cost (maxcost): this heuristic selects first in $Q_{AC}$ the variable having the greatest maximum unary cost in its domain. Then, constraints are processed in descending order of the maximum unary cost in the domain of the opposite variables.
- degree: this heuristic selects first in $Q_{AC}$ the variable having most neighbors and then constraints are processed in descending order of the number of neighbors (known as the size of the neighborhood) of the opposite variables.

## 3.4 Experimentation

### 3.4.1 VAC implementation

In this section, we will compare the efficiency of VAC when maintained during search in three different ways:

- Static VAC (VAC): Bool$(P)$ is always rebuilt from scratch whenever a new iteration of VAC or a branching operation is performed.
- Normal Dynamic VAC (norDVAC): Bool$(P)$ is maintained incrementally inside each node of the search (between VAC iterations) but it is rebuilt from scratch when the search makes a branching operation.
- Full DynVAC (fulDVAC): Bool$(P)$ is incrementally maintained both inside each node of the search and when a branching operation is performed.

As described in [Cooper et al., 2010], we remind the reader that when VAC is enforced in practice, iterations are stopped when the increase of the lower bound $c_\varnothing$ becomes less than a threshold $\varepsilon$. This is called VAC$_\varepsilon$. Furthermore, to accelerate VAC enforcing, [Cooper et al., 2010] suggest to try to collect largest costs first by using relaxations of Bool$(P)$ that only forbids tuples with sufficiently large costs, above a threshold $\theta$. This is denoted as Bool$(P)_\theta$. The set of all non-zero binary costs $c_{ij}$ of the problem are sorted in a fixed number $k$ of groups. The smallest costs of each group define a sequence of thresholds $(\theta_1, \theta_2, \ldots, \theta_k)$. Starting from $\theta_1$, VAC iterations are performed at a fixed threshold until there is no more wipe-out. Then the algorithm is performed at the next threshold $\theta_{i+1}$ in the sequence. After the last $\theta_k$, a geometric strategy, defined by $\theta_{i+1} = \theta_i/2$, is used such that the algorithm stops when $\theta_i$ is smaller than a given value $T$. The value $T$ used at the root node is $\varepsilon$ while during search, a large value $\varepsilon_s$ is used. This aims to maximally increase $c_\varnothing$ at the root and to avoid taking too much time to collect very small costs for $c_\varnothing$ during search.

In VAC and Normal DynVAC, when the search makes a branching operation (i.e., goes down in the search tree) or when the search backtracks, $\theta$ is reinitialized to $\theta_1$ in order to rebuild Bool($P$) from scratch. In Full DynVAC however, in order to incrementally maintain DynVAC during search between nodes, $\theta$ is incrementally updated when the search makes a branching operation and is restored by "trailing" when the search backtracks. It is noticed that after branching operations, new large costs may appear and this may again allow to increase $c_\varnothing$ by a large amount of cost. However, the $\theta$ inherited from the parent node may be too small to collect large costs to $c_\varnothing$ first. Costs can be split and this lead to too many iterations of VAC that each only increases $c_\varnothing$ by a small amount of costs. In order to avoid this problem, in our implementation of Full DynVAC, when the lower bound cannot be improved by more than $\varepsilon_s$ for 5 first successive iterations after a branching operation, $\theta$ will be reinitialized to $\theta_1$; i.e., Bool($P$) is rebuilt.

### 3.4.2 Set of benchmarks

In order to evaluate the efficiency of VAC algorithms when maintained during search, we use a large set of benchmarks that are collected from different resources (all in WCSP format):

- MaxCSP, planning, celar (Radio Link Frequency Assignment problems Cabon et al. [1999]), tag08 (bioinformatics Tag-SNP identification problems Sanchez et al. [2009]), warehouse (uncapacited warehouse location problems Kratica et al. [2001]), bep and ProteinDesign are extracted from the Cost Function Library (CFLib)[1]
- GeomSurf-7 are collected from the Computer Vision and Pattern Recognition (CVPR) OpenGM2 benchmark[2]
- Coloring consists of unsatisfiable binary CN instances with constraints defined in extension representing graph coloring problems.
- ImageAlignment and ProteinFolding are taken from the 2011 Probabilistic Inference Challenge.
- CPD are instances of computational protein design problems [Traoré et al., 2013; Allouche et al., 2014a].

Following the result of previous DynVAC experimentations for pre-processing in [Nguyen et al., 2013] that showed that DynVAC does not improve VAC on problems with Boolean domains, we excluded problems with only Boolean variables. The resulting very wide set of benchmarks represents a total of more than 1,000 instances coming from various application areas.

In our experiments, we set $\varepsilon = 1$, $\varepsilon_s = 1000$ for problems with very large optimums ($[10^6 \dots 10^{12}]$) such as tag08, warehouse, CPD, GeomSurf-7, and $\varepsilon = 1$, $\varepsilon_s = 100$ for prob-

[1]https://mulcyber.toulouse.inra.fr/scm/viewvc.php/trunk/?root=costfunctionlib
[2]http://hci.iwr.uni-heidelberg.de/opengm2

lems with large optimums ($[10^4 \dots 10^6]$) such as ImageAlignment, ProteinDesign, ProteinFolding. For the remaining problems, we use $\varepsilon = 0.0001$, $\varepsilon_s = 0.1$. The same thresholds are used in all algorithms. All experiments are executed on the same hardware using AMD Opteron(tm) 6176 processors.

### 3.4.3 Experimental results

Table 3.6 reports the mean run-time (in seconds) for solving the above different categories of benchmarks by enforcing static VAC, normal DynVAC and full DynVAC. Each line corresponds to a category of benchmarks where the number of instances (#inst) and the mean values of problem size ($n$: number of variables, $d$: domain size, $e$: number of cost functions) are given. We also report the mean graph densities per category (dens). The graph density of an instance is defined as the ratio of its number of cost functions with the number of edges in a complete graph.

The three following columns reports results for our three algorithms. Each box here contains two numbers. The italic number in the bottom represents the number of instances solved by the algorithm in less than one hour while the number in the top represents the mean value of the run-time. This mean is computed only over problems that are solved in less than 1 hour by all three variants of VAC. The best results are in bold. The 10th and 11th columns give the speedups obtained by full DynVAC compared respectively to normal DynVAC and VAC.

#### Comparing VAC and normal DynVAC

First, we compare the efficiency of normal DynVAC and VAC. The 12th column of the result table shows that normal DynVAC outperforms VAC on only 3 of 13 categories of benchmarks while it gets slower otherwise. It provides small speed-ups by a factor of 1.73, 1.22 and 1.08 on categories planning, bep and celar respectively where it solves one instance less than VAC on celar. This result means that normal DynVAC has not significantly improved performance over VAC, as expected.

To explain this behavior, we need to know the similarity as well as the difference between normal DynVAC and VAC. After each branching operation, both normal DynVAC and VAC have to restart from scratch with a reinitialized Bool($P$). Bool($P$) must be AC-filtered with all possible thresholds from $\theta_1$ in both variants. After branching operations, there may be few or no new large costs to increase $c_\varnothing$. In other words, almost all nodes are virtual arc consistent, as their parent nodes. In such nodes, each iteration of normal DynVAC and of VAC do the same Phase 1 in the sense that all variables in Bool($P$) have to be checked for AC. In this case, normal DynVAC has no benefit over VAC, it can even get slower because of the maintenance of the particular data structures of Bool($P$) used in normal DynVAC such as value domains, arc supports,...

#### Comparing normal DynVAC and full DynVAC

Fortunately, normal DynVAC can be improved by full DynVAC. The 14th column of the result table shows that full DynVAC outperforms normal DynVAC on all categories of benchmarks and is more efficient than VAC in most cases.

*Figure 3.6: Average solving time per category of benchmarks when maintaining variants of VAC in search.*

| | | | | | | Run-time | | | | | Speed-up of | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | # inst | $\overline{n}$ | $\overline{d}$ | $\overline{e}$ | $\overline{dens}$ | VAC | NorDV | FulDV | DomNor | DomFul | NorDV /VAC | FulDV /VAC | FulDV /NorDV | DomNor /NorDV | DomFul /FulDV |
| coloring | 22 | 120 | 4 | 1203 | 0,112 | **16,7** | 26,35 | 18,64 | 40,88 | 23,36 | 0,63 | 0,9 | 1,41 | 0,64 | 0,8 |
| | | | | | | 17 | 17 | 17 | 17 | 17 | | | | | |
| GeomSurf-7 | 300 | 505 | 7 | 2140 | 0,011 | 6,2 | 9,36 | **1,08** | 27,7 | 1,56 | 0,66 | 5,74 | 8,67 | 0,34 | 0,69 |
| | | | | | | 300 | 300 | 300 | 300 | 299 | | | | | |
| maxcsp | 340 | 31 | 10 | 129 | 0,213 | **58,76** | 78,75 | 58,99 | 100,32 | 76,58 | 0,75 | 1 | 1,34 | 0,78 | 0,77 |
| | | | | | | 340 | 340 | 340 | 340 | 340 | | | | | |
| planning | 76 | 240 | 14 | 14746 | 0,307 | 20,22 | 11,69 | **9,56** | 16,7 | 12,74 | 1,73 | 2,11 | 1,22 | 0,7 | 0,75 |
| | | | | | | 75 | 75 | 76 | 76 | 76 | | | | | |
| Matching | 4 | 19 | 19 | 166 | 0,441 | 104,03 | 367,4 | 281,99 | **100,72** | 146,97 | 0,28 | 0,37 | 1,3 | 3,65 | 1,92 |
| | | | | | | 4 | 3 | 3 | 4 | 4 | | | | | |
| celar | 46 | 137 | 36 | 1346 | 0,532 | 364,27 | 337,31 | **290,78** | 379,56 | 403,85 | 1,08 | 1,25 | 1,16 | 0,89 | 0,72 |
| | | | | | | 38 | 37 | 38 | 36 | 35 | | | | | |
| tag08 | 82 | 185 | 62 | 8003 | 0,179 | **14,09** | 31,42 | 27,39 | 14,47 | 14,87 | 0,45 | 0,51 | 1,15 | 2,17 | 1,84 |
| | | | | | | 81 | 80 | 81 | 81 | 81 | | | | | |
| Image Alignment | 10 | 191 | 70 | 1819 | 0,123 | 18,58 | 23,74 | **3,23** | 24,48 | 3,13 | 0,78 | 5,76 | 7,36 | 0,97 | 1,03 |
| | | | | | | 10 | 10 | 10 | 10 | 10 | | | | | |
| warehouse | 57 | 216 | 78 | 18739 | 0,237 | 164,26 | 289,3 | **90,49** | 257,7 | 83,83 | 0,57 | 1,82 | 3,2 | 1,12 | 1,08 |
| | | | | | | 54 | 51 | 55 | 50 | 56 | | | | | |
| Protein Design | 10 | 13 | 123 | 97 | 0,585 | **6,21** | 11,22 | 11,17 | 11,06 | 9,47 | 0,55 | 0,56 | 1 | 1,01 | 1,18 |
| | | | | | | 9 | 9 | 9 | 9 | 9 | | | | | |
| CPD | 35 | 50 | 148 | 1561 | 0,527 | **39,3** | 46,79 | 46,72 | 45,3 | 52,46 | 0,84 | 0,84 | 1 | 1,03 | 0,89 |
| | | | | | | 24 | 24 | 24 | 24 | 24 | | | | | |
| bep | 4 | 18 | 168 | 160 | 0,5 | 3,2 | 2,62 | **2,56** | 2,58 | 2,78 | 1,22 | 1,25 | 1,02 | 1,02 | 0,92 |
| | | | | | | 3 | 3 | 3 | 3 | 2 | | | | | |
| Protein Folding | 21 | 486 | 267 | 2291 | 0,293 | 37,3475 | 65,3985 | **34,292** | 104,7105 | 45,1735 | 0,57 | 1,09 | 1,91 | 0,62 | 0,76 |
| | | | | | | 21 | 21 | 21 | 21 | 20 | | | | | |

Compared to normal DynVAC, Full DynVAC shows very significant speed-ups on the categories GeomSur-7, ImageAlignment and warehouse (speed-ups of 8.7, 7.4 and 3.2 respectively). For the other categories (coloring, maxcsp, planning, matching, celar, tag08, ProteinFolding), the speed-up of full DynVAC goes from a factor 1.91 to a factor 1.15. Finally, it has almost the same efficiency on three categories (Protein Design, CPD and bep).

To explain these differences, we have to remember what makes full DynVAC different from normal DynVAC. The difference between full DynVAC and normal DynVAC comes from the way they update Bool$(P)$ after a branching operation. In full DynVAC, only the variables of Bool$(P)$ that are directly linked to the branching variable (its neighborhood) are modified and restorations are propagated to the other variables before arc consistency is rechecked on the modified variables. In normal DynVAC, the whole arc consistency process is redone from scratch on the reinitialized Bool$(P)$. It is therefore expected that the graph density of the instance solved will have a strong impact on efficiency. If the graph density is low, very few variables will appear in the neighborhood of the modified variable and full DynVAC should save a lot of work compared to normal Dyn VAC.

Another factor for efficiency is the domain size. If we restore a few values incrementally in a domain that was initially large, the cost of re-propagating AC can be much lower than if resetting the whole domain from scratch. If we restore a few values in a domain that was initially small, AC will not be significantly faster than if resetting the whole initial domain.

Overall, it is therefore expected that the greatest savings will be obtained by full DynVAC on problems with low graph densities and large domains. If we analyze the characteristics of the categories of problems on which full DynVAC behaves very well, we effectively observe that GeomSur-7 has a very low graph density (0,011 –lowest density of the set of benchmarks), ImageAlignment has a low density (0.123) and a large domain size (70), and warehouse has a medium density (0.237) but is characterized by a large domain size (78). Along the same line, by observing the categories on which full DynVAC is only slightly faster than normal DynVAC, we see that they are characterized by a high graph density (0.527 for CPD and 0.5 for bep).

This impact of density and domain size on the performance of full DynVAC seems to be confirmed on the other categories of problems. Coloring, for instance, has a very low density (0.112), and we would expect a very good performance of full DynVAC. However, the moderately improved performance of full DynVAC on this coloring category (only 1.4 times faster than normal DynVAC) can be explained by the very small mean domain size (4).

### Comparing full DynVAC and VAC

We now compare the efficiency of full DynVAC to the efficiency of static VAC. We observe that full DynVAC is better than static VAC on 7 of the 13 categories. The categories GeomSur-7, ImageAlignment and warehouse, where full DynVAC was significantly better than normal DynVAC, remain very favorable to full DynVAC when compared to static VAC. This tends to show that the criteria of density and domain size remain `true` when comparing to static VAC. We also observe that category planning is another example where full DynVAC is significantly faster than static VAC (by a factor 2.11).

Overall, if we compare the efficiency of the two variants of DynVAC to static VAC we observe that when normal DynVAC is faster than static VAC then full DynVAC is even faster. This however happens for only two categories: planning and bep, whereas full DynVAC is faster on 7 categories.

In summary, full DynVAC has a performance stronger than normal DynVAC when maintained in search. It can outperform static VAC on many problems that are not dense and having large domain size.

#### Using heuristics inside variants of DynVAC

In this section, we would use the domain-based revision heuristic inside two variants of DynVAC. The motivation for the usage of this heuristic is that it has been proved to be the best for CSPs [Wallace and E.C.Freuder, 1992] and [Boussemart et al., 2004a]. We denote by DomNor and DomFul the normal and full DynVAC when using this heuristic. The corresponding results are presented in the 10th, 11th columns of the result table. The last two columns of the table give the speed-ups obtained when using the heuristic.

From these last two columns, we observe that the heuristics allows to improve the performance of two variants of DynVAC on only 3 or 4 of 13 categories of benchmarks respectively while it slows down them on almost cases.

Precisely, the heuristic provides speed-ups of factors 3.65, 2.17, 1.12 to normal DynVAC and speed-ups of factors 1.92, 1.84, 1.08 to full DynVAC on categories Matching, tag08 and warehouse. Moreover, full DynVAC gets 1.18 time faster thanks to the heuristic. Among these categories, it is observed that two variants of DynVAC are slower than VAC on Matching and tag08, and the usage of heuristic lead to performances comparable to VAC on such categories.

For other cases, using the domain-based heuristic makes both two variants of DynVAC slower on 6 of 13 categories (coloring, GeomSur7, maxcsp, planning, celar and Protein-Folding). This can be explained by the computation of the ordering heuristic that is not useful inside nodes of the search tree when they are virtual arc consistent.

In summary, incrementally maintaining DynVAC during search is in general very beneficial in comparison with restarting from scratch after each branching operation and graph density and domain size are the main criteria that have a visible impact on the performance of full DynVAC. Moreover, full DynVAC can outperform static VAC on more than haft of the set of benchmarks. The domain-based revision heuristic seems not appropriate for two variants of DynVAC in the sense that it allows to accelerate them on only a few of problems while it slows them down in most cases.

## 3.5 Conclusions

In this chapter, we have introduced Dynamic VAC, an incremental approach for enforcing VAC in CFNs. It combines the idea of dynamic arc consistency algorithms with the iterative VAC algorithm in order to efficiently maintain arc consistency in the CSP Bool($P$) during VAC enforcing. By correctly specifying the incremental modifications in the CSP Bool($P$) caused by EPTs during successive iterations of VAC, the new method can easily maintain arc consistency in Bool($P$) by dynamic arc consistency and at the same time can save the work of enforcing arc consistency in Bool($P$) done in previous iterations.

Then we show that Dynamic VAC can be also maintained during search in two ways: either Dynamic VAC can be just maintained at each node of the search tree or the incrementality of Dynamic VAC can be exploited also to account for the small problem changes that occur following branching operations in the search tree. The former method has to rebuild the CSP Bool($P$) from scratch for each new node as done in the static VAC, while the second one uses again the idea of dynamic arc consistency to maintain arc consistency in Bool($P$).

By exploiting both incremental changes caused by branching operations as well as incremental changes of EPTs during successive iterations of VAC, the new fully incremental method outperforms both the direct application of dynamic VAC and the usual maintenance of static VAC on a variety of problems. This is especially `true` for problems having small graph densities and large domains.

The incompletely incremental method which ignores the incremental changes caused by branching operations can outperform the static VAC on a small set of problems. This method does not save the work done in parent nodes and thus has a performance weaker than the fully incremental method. The application of the domain-based revision heuristic for enforcing AC in the CSP Bool($P$) does not significantly improve the performance of two methods maintaining dynamic VAC in search.

There are still more opportunities to accelerate algorithms maintaining VAC during search. Indeed, VAC does not require to enforce AC on Bool($P$) but only to detect if the AC−closure is empty or not. Therefore, it is sufficient to identify a single viable value in each domain to conclude. This simplified problem has been solved using so-called Lazy AC algorithms [Schiex et al., 1996] that could also be injected in VAC algorithms to increase their efficiency.

# Chapter 4

# Soft high order consistencies

## 4.1 Introduction

The efficiency of Branch-and-Bound search when applying a local consistency depends on two factors: pruning power and time complexity of enforcing the consistency, where the pruning power is directly correlated to the quality of the lower bound provided by the consistency. The total time of the search for solutions is the combination of the time for enforcing the local consistency (defined by the enforcing time complexity) and the time for visiting the search space (defined by the pruning power). The enforcing time complexity should be put into balance with pruning power. Therefore, studying soft domain-based consistencies which are stronger than soft arc consistencies in terms of pruning power is a possible approach to speed up the search.

Indeed, strong soft consistencies can be defined for WCSPs by extending hard high order consistencies used for CSPs such as RPC, PIC, maxRPC, NIC, SAC, $k$-inverse consistency,... Among them, the group of triangle-based consistencies consisting of RPC, PIC, maxRPC are most interesting because they have a pruning power strong enough and their computational cost is much cheaper than the others by only dealing with subnetworks of three variables.

From hard arc consistency, many soft variants have been proposed for WCSPs such as (simple) AC, DAC, FDAC, FAC, EAC, EDAC, VAC. These consistencies are distinguished by the fact that (1) unary costs can be taken into account in the evaluation of arc supports and (2) consistencies are applied to either all domain values or only a value per domain. Based on the idea of extending hard arc consistency to soft arc consistencies, 6 soft variants for each hard consistency RPC, PIC, maxRPC have been proposed: the simple, directional, full directional, existential, existential directional and virtual one.

In addition to arc consistencies, soft RPCs, PICs, maxRPCs guarantee the extensibility without extra cost of arc supports on third variables, at values called "witnesses". The extensibility of a value to a triangle of variables is evaluated by the combined cost involving binary and ternary costs. By using or ignoring unary costs in this evaluation of extensibility, we have proposed 2 levels, called "simple" and "full" witnesses, similarly to simple and full arc supports. The simple, directional, full directional, existential, existential directional variants of RPC, PIC, maxRPC are based on the notion of simple and full supports and witnesses, while the virtual variant is based on the satisfaction of hard RPC, PIC, maxRPC in CSPs Bool($P$) of WCSPs $P$.

Soft RPCs, PICs, maxRPCs allow to improve the lower bound $c_\varnothing$ compared to soft arc consistencies by exploiting the combined costs involving unary, binary and ternary terms inside triangles of variables. To do this, beside soft arc consistency operations used in soft arc consistencies, soft RPCs, PICs, maxRPCs need extra operations for shifting costs inside triangles of variables.

The new consistencies keep the same property of corresponding hard RPC, PIC, maxRPC consistencies in terms of the pruning power which is represented by the quality of lower bound and by the capacity of increasing unary costs. The stronger relation among hard AC, PRC, PIC, maxRPC [Debruyne and Bessière, 1997b] is preserved for soft ACs, RPCs, PICs, maxRPCs. For each softness level, the soft maxRPC consistency is the strongest and the soft AC is the weakest. The advantage of the new consistencies is to provide strong lower-bounds for Branch-and-Bound search. However, enforcing these consistencies can create new ternary cost functions connecting variables of triangles and this will change the network structure of WCSPs.

This chapter consists of 4 sections. Section 4.2 presents the definitions of new soft RPCs, PICs, maxRPCs. The next section focus on comparing these consistencies in terms of the pruning power. In this section, a method for comparing soft consistencies is proposed by extending the "stronger relation" used for hard consistencies. A global relation between soft consistencies will be generalized. Section 4.4 will show how costs are shifted between cost functions to enforce the new consistencies and then introduce some algorithms for enforcing them. In Section 4.5 some experimental results of PIC, maxRPC obtained when used for pre-processing and for search will be introduced.

## 4.2 Definitions of soft high order consistencies

Based on the six soft arc consistencies AC, DAC, FDAC, EAC, EDAC, VAC extended from hard arc consistency, we have generalized 6 so-called "softness" levels for extending each hard domain based consistency to soft consistencies. A soft consistency at:

- "Non-directional" or "simple" level defines a property $A$ for every domain value $(i, a)$ in every sub-problem involving $i$. For example, the property $A$ defined by soft arc consistency for every value in every cost function (i.e., sub-problem containing the variables of the cost function) is that "having simple support".
- "Directional" level defines a property $B$ stronger than $A$ for every domain value $(i, a)$ in every sub-problem involving $i$ w.r.t an order $<$ on the variables. For example, the property $B$ defined by directional arc consistency for every value $(i, a)$ in every cost function $c_S$ such that $i$ is the smallest variable in $S$ is that "having full support".
- "Full directional" level defines the property $B$ for every domain value in every sub-problem involving $i$ w.r.t $<$; and the property $A$ in other sub-problems.
- "Existential" level defines a property $C$ (stronger or equal to $B$) for only a value per domain in every sub-problem involving $i$.
- "Existential directional" level combines the properties defined by the "existential" level and by the "full directional" one.
- "Virtual" level define a property WCSPs $P$ via the hard consistency in CSPs Bool($P$).

In this section, 6 soft consistencies defined at 6 softness levels above for each hard RPC, PIC, maxRPC consistency are presented. These new consistencies are based on the notion of simple and full arc supports that are used in soft arc consistencies. Moreover, they are also based on the notion of "simple and full" witnesses to express the extensibility of pairs of values on third variables. A pair of values is called simply extensible on a variable if there exists a value on that variable such that the 3-values tuple has a zero combined cost, consisting of only binary and ternary costs, not unary costs. A pair of values is called fully extensible on a variable if there exists a value on that variable such that the 3-values tuple has an zero combined cost, consisting of unary, binary and ternary costs. Such values are called respectively the simple and full witnesses of the pair of values on the third variable.

From this section, if we say that a variable $i$ is linked to a variable $j$, this means that there exists a binary cost function $c_{ij}$ between them.

**Definition 4.1 (Witness and extensibility on a variable)** *For 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 $c_k(c) + c_{ik}(a, c) + c_{jk}(b, c) + c_{ijk}(a, b, c) = 0$.*
- *$(i_a, j_b)$ is simply extensible on variable $k$ if there exists a simple witness on $k$ for $(i_a, j_b)$.*
- *$(i_a, j_b)$ is fully extensible on variable $k$ if there exists a full witness on $k$ for $(i_a, j_b)$.*

In all the examples of this chapter, WCSPs are represented by graphs where (1) arcs between two vertices indicate that the corresponding pairs of values have non-zero binary costs and (2) vertices surrounded by a red circle mean that the corresponding values are inconsistent with a given soft consistency.

**Example 4.1** *Consider the WCSP(a) in Figure 4.1. $(k_2, b)$ is a simple witness for the pair of values $(i_a, j_a)$ on $k_2$ but it is not a full witness. Thus, $(i_a, j_a)$ is simply extensible but is not full extensible on $k_2$. Moreover, $(i_a, j_a)$ is neither simply nor fully extensible on $k_1$.*

**Definition 4.2 (Extensibility)** *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 variable $k$ linked both to $i$ and $j$.*
- *fully extensible if it is fully extensible on every variable $k$ linked both to $i$ and $j$.*
- *directionally-fully extensible if it is fully extensible on every $k > i$ linked both to $i$ and $j$.*
- *semi-fully extensible if it is simply extensible on every variable $k < i$ and is fully extensible on every $k > i$ such that $k$ is 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. Consider the example in Figure 4.1 to better understand the different extensibility of pairs of values.

If a pair of values $(i_a, j_b)$ is not extensible, there exists a variable $k$ such that for every $c \in D(k)$, the combined cost (involving binary and ternary costs with or without unary costs) of tuple $(i_a, j_b, k_c)$ is positive.

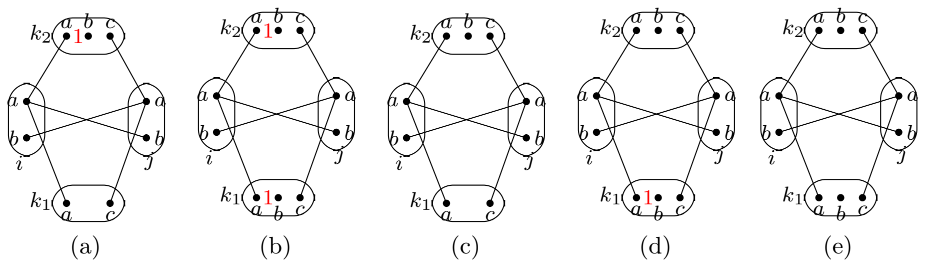


*Figure 4.1: Example of different extensibilities of the pair of values $(i_a, j_a)$. $k_1 < i < j < k_2$. In WCSP(a), $(i_a, j_a)$ is not simply extensible on $k_1$. In WCSP(b), $(i_a, j_a)$ is simply extensible (on both $k_1, k2$) but is not directionally-fully extensible (because it is not fully extensible on $k_2$). In WCSP(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 WCSP(d), $(i_a, j_a)$ is semi-fully extensible (fully extensible on $k_1$ and simply extensible on $k_2$) but is not fully extensible (because it is not fully extensible on $k_1$). In WCSP(e), $(i_a, j_a)$ is fully extensible (on both $k_1, k_2$).*

Let $\alpha$ be the smallest among the combined costs of all tuples $(i_a, j_b, k_c)$. The sub-problem over 3 variables $\{i, j, k\}$ is equivalent to a WCSP in which the binary cost $c_{ij}(a, b)$ is increased by $\alpha$ while the combined cost, excluding $c_{ij}$, of every tuple $(i_a, j_b, k_c)$ is decreased by $\alpha$. This means that there exists an equivalence preserving transformation to increase the binary cost $c_{ij}(a, b)$ and this can possibly break soft arc consistencies of $c_{ij}$. This property will be used in every soft RPC, PIC, maxRPC consistency. The notations ACs, RPCs, PICs, maxRPCs are used to briefly indicate all AC, RPC, PIC, maxRPC consistencies.

### 4.2.1 Soft restricted path consistencies

In addition to soft arc consistencies, soft RPC consistencies check the extensibility only for pairs of values which will make a value soft arc inconsistent if their binary cost increases. The idea of soft restricted path consistencies is to guarantee that every unique (simple or full) arc support of every value in every binary cost function is (simply, directionally-fully, semi-fully, fully) extensible on every third variables. If a value $(i, a)$ has only one simple support $(j, b^*)$ in a binary cost function $c_{ij}$ and this support $(i_a, j_{b^*})$ is not extensible on some third variable $k$, this means that 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 an EPT.

**Definition 4.3 (RPC)** *A WCSP is restricted path consistent (RPC) if it is arc consistent and $\forall i, \forall a \in D(i), \forall c_{ij}$ on which $(i, a)$ has only one simple arc support $b \in D(j)$, $(i_a, j_b)$ is simply extensible.*

**Example 4.2** *Consider the WCSP in Figure 4.1(a). This problem is EDAC but not RPC because of value $(i, a)$. The unique support $(i_a, j_a)$ of this value on $c_{ij}$ is not simply extensible on $k_1$. Conversely, the problem in Figure 4.1(b) is RPC. Both $(i_a, j_a)$ (the unique simple arc support of $(i, a), (j, a)$ in $c_{ij}$) and $(i_b, j_b)$ (the unique simple arc support of $(i, b), (j, b)$ in $c_{ij}$) are simply extensible on $k_1$ and $k_2$ at simple witnesses $(k_1, b)$ and $(k_2, b)$ respectively.*

RPC is also called non-directional RPC. RPC guarantees the simple extensibility for every binary arc support which is the unique simple arc support for the corresponding values and this property is applied for both sides of binary cost functions. The following consistency, called directional RPC, guarantees the directional-full extensibility for every binary arc support which is the unique full arc support for the corresponding values. This property is applied for only one side of each binary cost function.

**Definition 4.4 (DRPC)** *A WCSP is directional restricted path consistent (DRPC) with respect to an order "<" on the variables if it is directional arc consistent with respect to <, and* $\forall i, \forall a \in D(i), \forall c_{ij}$ *such that* $i < j$ *on which* $(i,a)$ *has only one full arc support* $b \in D(j)$, $(i_a, j_b)$ *is directionally-fully extensible.*

**Example 4.3** *Consider the WCSP(b) in Figure 4.1. It is not DRPC because* $(i,a)$ *has only one full arc support* $(i_a, j_a)$ *in* $c_{ij}$ *but* $(i_a, j_a)$ *is not fully extensible in* $k_2 > i$*. Conversely, the WCSP(c) in Figure 4.1 is DRPC because both* $(i_a, j_a)$ *(the unique full arc support of* $(i,a)$ *in* $c_{ij}$*) and* $(i_b, j_b)$ *(the unique full arc support of* $(i,b)$ *in* $c_{ij}$*) are fully extensible on* $k_2 > i$ *at* $(k_2, b)$*. Variable* $k_1 < i$ *is not concerned by DRPC for* $i$*.*

**Definition 4.5 (FDRPC)** *A WCSP is full directional restricted path consistent (FDRPC) with respect to an order "<" on the variables if it is full directional arc consistent with respect to <, and* $\forall i, \forall a \in D(i), \forall c_{ij}$ *such that 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 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.*

Indeed, FDRPC ensures DRPC on one side of each binary cost function and RPC on the other side.

**Example 4.4** *Consider the WCSP(c) in Figure 4.1. It is not FDRPC because of value* $(i,a)$*. Its unique full support* $(i_a, j_a)$ *is not simply extensible on* $k_1$*. Conversely, the WCSP(d) in Figure 4.1 is FDRPC.*

ERPC has been proposed based on the idea of EAC which requires the existence of a special value per domain that satisfies a property stronger than FDAC. While RPC, DRPC, FDRPC require that every domain value must satisfy a same property, ERPC requires that each domain has at least a value which has full arc supports in every cost function such that every unique full binary support is fully extensible.

**Definition 4.6 (ERPC)** *A WCSP is existential restricted path consistent (ERPC) if for every variable* $i$*, there exists a value* $a \in D(i)$ *such that*

- $c_i(a) = 0$
- $i_a$ *has a full arc support in every cost function and*
- *for every cost function* $c_{ij}$ *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* $i$*.*

**Example 4.5** *Consider the WCSP(d) in Figure(d) in Figure 4.1. It is ERPC where* $(i,b)$*,* $(j,b)$*,* $(k_1,a)$*,* $(k_2,a)$ *are ERPC supports for variables* $i, j, k_1$ *and* $k_2$*.*

EDRPC guarantees (1) the property of FDRPC for every domain value and (2) the existence of an ERPC support per domain.

**Definition 4.7 (EDRPC)** *A WCSP is existential directional restricted path consistent (FDRPC) with respect to an order "<" on the variables if it is existential restricted path consistent and full directional restricted path consistent with respect to <.*

**Example 4.6** *Consider the WCSP(d) in Figure 4.1. This problem is EDRPC because it is ERPC (example 4.5) and FDRPC (example 4.4).*

Similarly to the idea of VAC, the virtual RPC exploits the relation between WCSPs $P$ and their CSPs Bool($P$) in such a way that virtual RPC is established in $P$ via the establishment of hard RPC in Bool($P$).

**Definition 4.8 (VRPC)** *A WCSP $P$ is virtual restricted path consistent (VRPC) if the restricted path consistency closure of the CSP Bool($P$) is non-empty.*

The definition below expresses the relation between all soft consistencies above with hard RPC.

**Definition 4.9 (Soft consistencies associated with hard RPC)** *The soft RPC, DRPC, FDRPC, ERPC, EDRPC and VRPC are soft consistencies associated with hard RPC.*

### 4.2.2 Soft path inverse consistencies

In addition to arc consistencies, soft path inverse consistencies guarantee the extensibility of domain values on triangles of variables. For all triangles $(i, j, k)$ sharing two variables $i, j$ with constraint $c_{ij}$, PIC requires that for each $k$, 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 4.10 (PIC)** *A WCSP is path inverse consistent (PIC) if it is arc consistent and $\forall i, \forall a \in D(i), \forall c_{ij}, \forall k$ linked both to $i$ and $j$, there exists a simple arc support $b \in D(j)$ such that $(i_a, j_b)$ is simply extensible on $k$.*

**Definition 4.11 (DPIC)** *A WCSP is directional path inverse consistent (DPIC) with respect to an order "<" on the variables if it is directional arc consistent with respect to <, and $\forall i, \forall a \in D(i), \forall c_{ij}$ such that $i < j, \forall k$ linked both to $i$ and $j$ such that $i < k$, there exists a full arc support $b \in D(j)$ such that $(i_a, j_b)$ is fully extensible on $k$.*

**Definition 4.12 (FDPIC)** *A WCSP is full directional path inverse consistent (FDPIC) with respect to an order "<" on the variables if it is full directional arc consistent with respect to <, and $\forall i, \forall a \in D(i), \forall c_{ij}, \forall k$ linked both to $i$ and $j$, if $i > j$ or $i > k$, there exists a simple arc support $b \in D(j)$ such that $(i_a, j_b)$ is simple extensible on $k$; otherwise if $i < j$ and $i < k$, there exists a full arc support $b \in D(j)$ such that $(i_a, j_b)$ is fully extensible on $k$.*

**Definition 4.13 (EPIC)** *A WCSP is existential path inverse consistent (EPIC) if for every variable $i$, there exists a value $a \in D(i)$ such that*

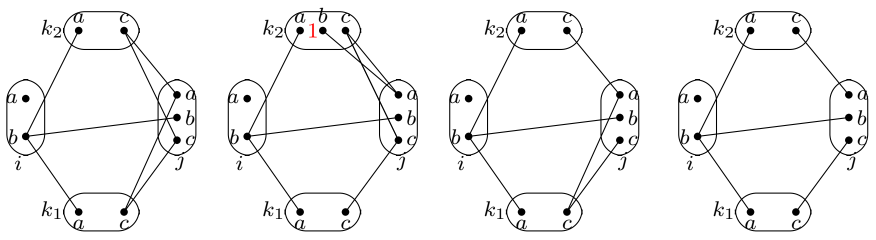
- $c_i(a) = 0$
- *$i_a$ has a full arc support in every cost function and*
- *for every cost function $c_{ij}$, for every $k$ linked both to $i$ and $j$, there exists a full arc support $b \in D(j)$ such that $(i_a, j_b)$ is fully extensible on $k$.*

**Definition 4.14 (EDPIC)** *A WCSP is existential directional path inverse consistent (EDPIC) with respect to an order "<" on the variables if it is existential path inverse consistent and full directional path inverse consistent with respect to <.*

Consider WCSPs in Figure 4.2 to distinguish the soft variants of PIC.

**Definition 4.15 (VPIC)** *A WCSP is virtual path inverse consistent (VPIC) if the path inverse consistent closure of the CSP Bool$(P)$ is non-empty.*

**Definition 4.16 (Soft consistencies associated with hard PIC)** *The soft PIC, DPIC, FDPIC, EPIC, EDPIC and VPIC are soft consistencies associated with hard PIC.*



(a) not PIC (b) PIC not DPIC (c) DPIC not FDPIC (d) EDPIC

*Figure 4.2: Example of soft PIC consistencies. $k_1 < i < k_2 < j$ and $\exists\ c_{ij}, c_{ik_1}, c_{ik_2}, c_{jk_1}, c_{jk_2}$. The WCSP(a) is not PIC because value $(i,b)$ is not simply extensible to triangle $(i,j,k_1)$. The WCSP(b) is PIC but is not DPIC because value $(i,b)$ is not fully extensible to triangle $(i,j,k_2)$ with $i < j, i < k_2$. The WCSP(c) is DPIC (because every value in $D(i)$ can be fully extended to $(i,j,k_2)$ only which is concerned by DPIC for $i$) but it is not FDPIC (because value $(i,b)$ is not simply extensible to triangle $(i,j,k_1)$). The WCSP(d) is FDPIC where every variable is simply extensible to 2 triangles and $i$ is fully extensible to $(i,j,k_2)$. The WCSP(d) is also EPIC where $(i,a),(j,a),(k_1,a),(k_2,a)$ are respectively EPIC supports of $i,j,k_1,k_2$.*

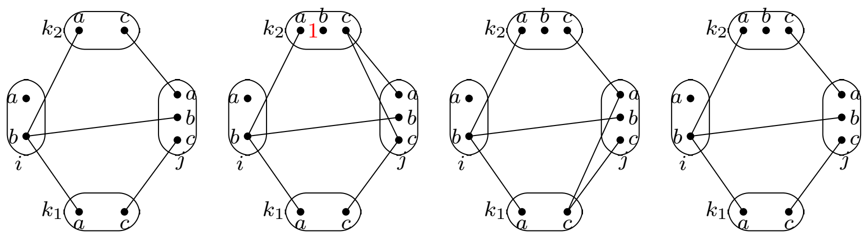
### 4.2.3 Soft max-restricted path consistencies

The idea of soft max-restricted path consistencies (soft maxRPCs) is to check the existence of an extensible arc support for each value in each binary cost function whatever the number of arc supports the value has. In constrast to soft PICs, maxRPCs require the extensibility of the same arc support for each value in each binary cost function at the same time on all third variables. If a value $(i,a)$ has no such extensible arc support in some binary cost function $c_{ij}$, the binary cost of every arc support of $(i,a)$ in $c_{ij}$ can increase by equivalence preserving transformations and then $(i,a)$ will no longer be arc consistent.

As usual, the simplest version of soft max-restricted path consistencies is soft maxRPC, which checks the simple extensibility for simple arc supports.

**Definition 4.17 (maxRPC)** *A WCSP is max-restricted path consistent (maxRPC) if it is arc consistent and $\forall i, \forall a \in D(i), \forall c_{ij}$ there exists a simple arc support $b \in D(j)$ such that $(i_a, j_b)$ is simply extensible.*

**Example 4.7** *Consider the WCSP(a) in Figure 4.3 which is exactly the WCSP(d) in Figure 4.2 that has been proved EDPIC. This problem is not maxRPC because of value $(i,b)$. This value has 2 arc supports $(j,a)$ and $(j,c)$ in $c_{ij}$, but none is simply extensible on both $k_1, k_2$ ($(i_b, j_a)$ is not simply extensible on $k_2$ while $(i_b, j_c)$ is not simply extensible on $k_1$). Consider the WCSP(b) in Figure 4.3. This problem is maxRPC, $(i,b)$ has an arc support in $c_{ij}$, $(i_b, j_a)$, which is simply extensible on both $k_1$ and $k_2$ at respectively $(k_1,c)$ and $(k_2,b)$).*



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

*Figure 4.3:* ***Example of soft maxRPCs.*** *$k_1 < i < k_2 < j$ and $\exists\ c_{ij}, c_{ik_1}, c_{ik_2}, c_{jk_1}, c_{jk_2}$. The WCSP(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 WCSP(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 WCSP(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 $(i,j,k_1)$ is not concerned by DmaxRPC for $i$). The WCSP(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 WCSP(d) is both FDmaxRPC and EmaxRPC where $(i,a), (k_1,a), (j,a), (k_2,a)$ are respectively EPIC supports of variables $i, k_1, k_2, j$.*

**Definition 4.18 (DmaxRPC)** *A WCSP is directional max-restricted path consistent (DmaxRPC) with respect to an order "<" on the variables if it is directional arc consistent with respect to <, and $\forall i, \forall a \in D(i), \forall c_{ij}$ such that $i < j$, there exists a full arc support $b \in D(j)$ such that $(i_a, j_b)$ is directionally-fully extensible.*

**Example 4.8** *Consider the WCSP(b) in Figure 4.3. It is not DmaxRPC because of value $(i,b)$. Both full arc supports $(i_b, j_a)$ and $(i_b, j_c)$ of $(i,b)$ in $c_{ij}$ are not fully extensible in $k_2$. Conversely, the WCSP(c) in Figure 4.3 is DmaxRPC where only triangle $(i,j,k)$ of $i$ is concerned by DmaxRPC because $k_1 < i < k_2 < j$.*

**Definition 4.19 (FDmaxRPC)** *A WCSP is full directional max-restricted path consistent (FDmaxRPC) with respect to an order "<" on the variables if it is full directional arc consistent with respect to <, and $\forall i, \forall a \in D(i), \forall c_{ij}$ (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.*

FDmaxRPC ensures DmaxRPC on one side of each binary cost function and maxRPC on the other side.

**Example 4.9** *Consider the WCSP(c) in Figure 4.3. It is not FDmaxRPC because value $(i,b)$ has no arc support in $c_{ij}$ that is simply extensible on $k_1$. Conversely, the WCSP(d) in Figure 4.3 is FDmaxRPC where the full supports $(i_a, j_a)$ of $(i,a)$ and $(i_b, j_a)$ of $(i,b)$ $c_{ij}$ are fully extensible (on both $k_1, k_2$).*

**Definition 4.20 (EmaxRPC)** *A WCSP is existential max-restricted path consistent (EmaxRPC) if for every variable $i$, there exists a value $a \in D(i)$ such that*

- $c_i(a) = 0$
- *$i_a$ has a full arc support in every cost function and*
- *for every cost function $c_{ij}$ there exists a full arc support $b \in D(j)$ such that $(i_a, j_b)$ is*

*fully extensible.*

**Example 4.10** *Consider the WCSP(a) in Figure 4.3. This problem is EmaxRPC (despite the fact that it is not maxRPC) where values $(i, a), (j, a), (k_1, a), (k_2, a)$ are respectively the EmaxRPC supports of variables $i, j, k_1$ and $k_2$. Similarly, the WCSP(d) in Figure 4.3 is also EmaxRPC.*

**Definition 4.21 (EDmaxRPC)** *A WCSP is existential directional max-restricted path consistent (EDmaxRPC) with respect to an order $<$ on the variables if it is existential max-restricted path consistent and full directional max-restricted path consistent with respect to $<$.*

**Definition 4.22 (VmaxRPC)** *A WCSP $P$ is virtual max-restricted path consistent (VmaxRPC) if the max-restricted path consistency closure of the CSP Bool$(P)$ is non-empty*

**Definition 4.23 (Soft consistencies associated with hard maxRPC)** *The soft consistencies maxRPC, DmaxRPC, FDmaxRPC, EmaxRPC, EDmaxRPC and VmaxRPC are said to be associated with hard maxRPC.*

## 4.3 Comparison between soft domain consistencies

Similarly to hard consistencies, soft consistencies can be classified in soft domain-based consistencies and soft constraint-based consistencies. The former defines properties for unary costs and the latter for $k$-ary costs with $k \geq 2$. The virtual consistencies such as VAC, VRPC, VPIC and VmaxRPC are not classified in either of these two forms because they defines properties for all costs of WCSPs $P$ via the classical CSPs Bool$(P)$. All ACs, RPCs, PICs and maxRPCs except for the virtual consistencies are soft domain-based consistencies (also briefly called domain consistencies). In this document, we are only interested in soft domain based consistencies and virtual consistencies of hard domain based consistencies.

In classic CSPs, hard consistencies are enforced by eliminating inconsistent elements: values for domain-filtering consistencies and tuples for constraint-filtering ones. The efficiency of a hard consistency is represented by its pruning power: the more values and tuples are pruned, the more powerful the consistency is. In WCSPs, soft consistencies are enforced by shifting costs between cost functions of different arities in order to (1) increase the lower bound $c_\varnothing$ (e,g,. VAC) or (2) move costs from cost functions of higher arity to cost functions of lower arity, that is, from a cost function of scope $S$ to a cost function of scope $S' \subset S$, where $|S'| = 1$ for soft domain consistencies. The final result of cost movements is to remove elements (values for soft domain consistencies and tuples for soft constraint consistencies) that become too costly or to increase the lower bound $c_\varnothing$. Thus, the power of virtual consistencies is evaluated by the quality of the lower bound $c_\varnothing$ and the power of soft domain consistencies is further evaluated by the capacity of increasing unary costs.

### 4.3.1 Stronger relation

To compare the efficiency of soft domain consistencies and virtual consistencies, we define two relations: "stronger" $\geq$ and "stronger in terms of lower bounds" $\geq_{c_\varnothing}$, based on the idea of the stronger relation used for hard domain consistencies [Debruyne and Bessière, 1997b]. A soft consistency $A$ is called stronger than a soft consistency $B$ if for every problem which already satisfies $A$, the weaker consistency $B$ cannot improve it in terms of increasing unary costs and $c_\varnothing$. For a given WCSP $P$ and a soft consistency $A$, let $c_\varnothing[P]$ denote the lower bound of $P$ and $A(P)$ a problem (which is not necessarily unique) obtained after enforcing $A$ in $P$.

**Definition 4.24 (Stronger relation)** *Given two soft consistencies $A$ and $B$,*

- *$A$ is stronger than $B$, noted by $A \geq B$, iff for every WCSP $P$ that satisfies $A$, $B(P) = P$.*
- *$A$ is stronger than $B$ in terms of lower bound, noted by $A \geq_{c_\varnothing} B$, iff for every WCSP $P$ that satisfies $A$, $c_\varnothing[B(P)] = c_\varnothing[P]$*

The relation $\geq$ means that if a WCSP $P$ satisfies $A$ and $A \geq B$, then $P$ also satisfies $B$. The relation $\geq_{c_\varnothing}$ means that if a WCSP $P$ satisfies $A$ and $A \geq_{c_\varnothing} B$, then $B$ cannot increase $c_\varnothing$ of $P$. Enhancing the lower bound is only one effect of soft consistency enforcement beside increasing unary costs (domain based consistencies), binary costs (higher order consistencies), etc. Thus, the relation $\geq$ subsumes $\geq_{c_\varnothing}$. Indeed, $\geq_{c_\varnothing}$ is a relaxation of $\geq$. The relation $\geq$ is used on two soft domain consistencies while $\geq_{c_\varnothing}$ is used on two virtual consistencies or for a virtual consistency and a soft domain consistency.

**Proposition 4.1** *Given two soft consistencies $A$ and $B$. If $A \geq B$ then $A \geq_{c_\varnothing} B$.*

**Proof 6** *The proof is trivial. Because $A \geq B$, $B(P) = P$ for every $P$ that satisfies $A$. So we have $c_\varnothing[B(P)] = c_\varnothing[P]$ and thus $A \geq_{c_\varnothing} B$.*

Based on the stronger relation, the strictly stronger relation has been proposed to express the strict dominance of the stronger consistency in the sense that (1) the weaker consistency cannot do better than the stronger one in WCSPs which already satisfy the stronger, (2) while the stronger consistency does better than the weaker one for at least one WCSP which already satisfies the weaker consistency.

**Definition 4.25 (Strictly stronger relation)** *Given two soft consistencies $A$ and $B$.*

- *$A$ is strictly stronger than $B$, noted $A > B$, iff $A \geq B$ and $\exists$ a WCSP $P$ 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 WCSP $P$ such that $P$ satisfies $B$ and $c_\varnothing[A(P)] > c_\varnothing[P]$*

To show that a soft consistency $A$ is not stronger or not stronger in terms of lower bounds than $B$, it is enough to show that there exists a WCSP $P$ in which $A$ holds and $B$ does better than $A$. This means that $B$ still can modify (i.e., increase unary costs or the lower bound) some problem which is already $A$. Two consistencies $A$ and $B$ are incomparable iff $A$ is not stronger than $B$ and $B$ is not stronger than $A$.

**Definition 4.26 (Incomparable relation)** *Given two soft consistencies $A$ and $B$,*

- *$A$ and $B$ are incomparable, noted $A \not\gtrless B$, iff $A \not\geq B$ and $B \not\geq A$*
- *$A$ and $B$ are incomparable in terms of lower bound, noted $A \not\gtrless_{c_\varnothing} B$, if $A \not\geq_{c_\varnothing} B$ and $A \not\geq_{c_\varnothing} B$*

**Proposition 4.2** *Given two soft consistencies $A$ and $B$. If $A \not\gtrless_{c_\varnothing} B$ then $A \not\gtrless B$.*

**Proof 7** *Suppose that $A \not\succ_{c_\varnothing} B$, we have $A \ngeq_{c_\varnothing} B$ and $B \ngeq_{c_\varnothing} A$. Because $\geq$ implies $\geq_{c_\varnothing}$, so $\ngeq_{c_\varnothing}$ implies $\ngeq$. Thus, we have $A \ngeq B$ and $B \ngeq A$. This means that $A \not\succ B$.*

Similarly to the stronger and strictly stronger relations for hard consistencies, our relations for soft consistencies also have the transitivity property.

**Property 3 (Transitivity)** *Given three soft consistencies $A, B, C$.*

*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 $A \geq B$ and $B \geq_{c_\varnothing} C$ then $A \geq_{c_\varnothing} C$*

**Proof 8** *a. Let $P$ be a WCSP 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, for every WCSP $P$ which satisfies $A$, $C(P) = P$, i.e., $A \geq C$.*

*b. Because $>$ implies $\geq$, we have $A \geq B$ and $B \geq C$. So $A \geq C$ from the property (a). On the other hand, because $A > B$, there exists a WCSP $P^*$ such that $P^*$ satisfies $B$ and $A(P^*) \neq P^*$. Because $P^*$ satisfies $B$ and $B \geq C$, $C(P^*) = P$, i.e., $P^*$ also satisfies $C$. Thus there exists $P^*$ which is $C$ and $A(P^*) \neq P^*$. So $A > C$.*

*c. Let $P$ be a WCSP that satisfies $A$. Because $A \geq B$ and $P$ satisfies $A$, $B(P) = P$, i.e., $P$ also satisfies $B$. Because $B \geq_{c_\varnothing} C$ and $P$ satisfies $B$, $c_\varnothing[C(P)] = c_\varnothing[P]$. Thus, for every WCSP $P$ which satisfies $A$, $c_\varnothing[C(P)] = c_\varnothing[P]$. i.e., $A \geq_{c_\varnothing} C$.*

### 4.3.2 Relation graph

The graph 4.4 summarizes the strictly stronger relations among all variants of soft ACs, RPCs, PICs and maxRPCs where continuous arrows present the $>$ relation and dashed arrows present the $>_{c_\varnothing}$ relation. The soft consistencies presented in the same row are soft consistencies associated with a same hard consistency while the soft consistencies in the same column are called soft consistencies of the same "softness" level, e,g,. DAC, DRPC, DPIC, DmaxRPC are soft directional consistencies. If there exists a directed path from a consistency $A$ to $B$ in the graph, consisting of continuous arrows, it means that $A > B$. This presents the transitive property of the strictly stronger relation. If there exists a directed path from $A$ to $B$, consisting of continuous and dashed arrows, it means that $A >_{c_\varnothing} B$. Conversely, if there does not exist any directed path between $A$ and $B$, they are incomparable.

First, we consider the relation between virtual consistencies and other soft consistencies. This relation is indicated by the following theorems. Let $\overline{A}$ be a hard consistency and VA is the soft virtual consistency defined via $\overline{A}$ in CSPs Bool$(P)$. The following theorem will give a relation between virtual consistencies VA and soft consistencies $A$ associated to hard consistencies $\overline{A}$.

**Theorem 4.1** *Given a hard consistency $\overline{A} \in \{AC, RPC, PIC, maxRPC\}$, the virtual consistency VA and a soft consistency $A \neq$ VA associated with $\overline{A}$. VA $>_{c_\varnothing} A$*

**Proof 9** *First, to prove that VA $\geq_{c_\varnothing} A$, we will use a proof by contradiction. Suppose that there exists a WCSP $P$ which $P$ is VA and $A$ still can increase the lower bound of $P$ from a variable $x_\varnothing$. All values and tuples whose costs have been necessary for increasing the lower bound $c_\varnothing$ 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 supposition is false. This means that for every P which satisfies VA, A cannot increase the lower bound of P: VA $\geq_{c_\varnothing}$ A. Secondly, Figure 4.12 shows a problem which satisfies every soft consistency associated with hard AC, RPC, PIC, maxRPC, but does not satisfy VAC, VRPC, VPIC and VmaxRPC. In summary, VA > A for every associated hard consistency $\overline{A} \in \{$ AC,RPC,PIC,maxRPC$\}$*

Virtual consistencies can be compared through the comparison of associated hard consistencies. The following theorem will show that the stronger relation of hard consistencies is preserved in the case of virtual soft consistencies: the stronger the hard consistency is, the stronger its virtual soft version is.

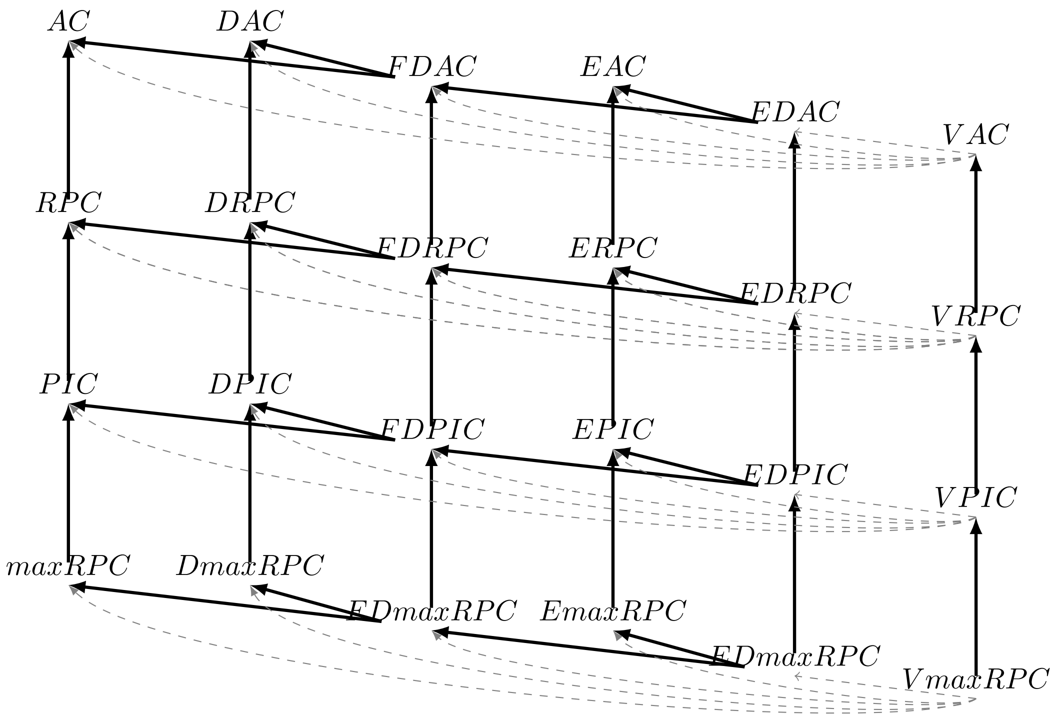


*Figure 4.4: Hasse diagram of relations between soft consistencies*
$A \longrightarrow B : A > B$
$A \dashrightarrow B : A >_{c_\varnothing} B$
$A \longrightarrow B \longrightarrow C$ implies $A \longrightarrow C$
$A \longrightarrow B \dashrightarrow C$ implies $A \dashrightarrow C$

**Theorem 4.2** *Given two hard consistencies $\overline{A}$, $\overline{B} \in \{$AC,RPC,PIC,maxRPC$\}$ and two virtual consistencies VA, VB respectively associated with $\overline{A}$, $\overline{B}$. If $\overline{A} > \overline{B}$ then*

*a. VA > VB*
*b. VA $>_{c_\varnothing}$ B for every soft consistency $B \neq VB$ associated with $\overline{B}$.*

**Proof 10**

*a. Firstly, we prove that $VA \geq VB$. Let P be a WCSP 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 $VA > VB$. Because hard maxRPC > hard PIC > hard RPC > hard AC, we have to prove that VmaxRPC*

*> VPIC > VRPC > VAC. Figure 4.5 shows a WCSP P which satisfies VAC but not VRPC (because the CSP Bool(P) is AC but not RPC with a wipe-out at j): VRPC can increase $c_\varnothing$ by 1, i.e., can modify P. Figure 4.6 shows a WCSP P which satisfies VRPC but not VPIC (because the CSP Bool(P) is RPC but not PIC with a wipe-out at i): PIC can increase $c_\varnothing$ by 1, i.e., can modify P. Figure 4.7 shows a WCSP P which satisfies VPIC but not VmaxRPC (because the CSP Bool(P) is PIC but not maxRPC with wipe-out at i): maxRPC can increase $c_\varnothing$ by 1, i.e., can modify P.*

b. *From Theorem 4.2(a), we have VA > VB. From Theorem 4.1, we have $VB >_{c_\varnothing} B$ which implies that $VB \geq_{c_\varnothing} B$. From Property 3(c), we have $VA \geq_{c_\varnothing} B$. Now, we will prove that $VA >_{c_\varnothing} B$. Because $VB >_{c_\varnothing} B$, there exists a WCSP P such that P is B and VB can still increase the lower bound $c_\varnothing[P]$. 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, $c_\varnothing[VA(P)] > c_\varnothing[P]$ and P satisfies B.*

Theorem 4.2(a) is reformulated as VmaxRPC > VPIC > VRPC > VAC.

Now, the relation between soft domain consistencies sharing the same column and then the same row is considered.

### Vertical comparison

Each column in Graph 4.4 consists of 4 soft consistencies at the same softness level associated with hard consistencies AC, RPC, PIC, maxRPC. The following theorem will show that at each "softness" level, the soft maxRPC is strictly stronger than the soft PIC,the soft PIC is strictly stronger than the soft RPC,and the soft RPC is strictly stronger than the soft AC.

**Theorem 4.3 (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 11** *Firstly, we will prove the stronger relation ≥ between consistencies. According to the definition of soft RPCs, PICs and maxRPCs at non-directional level, maxRPC implies PIC, PIC implies RPC and RPC implies AC. Therefore maxRPC ≥ PIC ≥ RPC ≥ AC. The proof is the same for other softness level, except for virtual consistencies. According to Theorem 4.2, we have VmaxRPC ≥ VPIC ≥ VRPC ≥ VAC. Now, to prove the strictly stronger relation between consistencies, we will show a WCSP in which the weaker consistency is satisfied while the stronger is not.*

a. *Figure 4.5 shows a WCSP which satisfies AC but does not satisfy RPC. Figure 4.6 shows a WCSP which satisfies RPC but does not satisfy PIC. Figure 4.7 shows a WCSP which satisfies PIC but does not satisfy maxRPC. Thus maxRPC > PIC > RPC > AC.*

b-e. *The proof is similar to that for (a) by using Figures 4.5, 4.6 and 4.7.*

### Horizontal comparison

Each row in Graph 4.4 consists of six soft consistencies associated with a same hard consistency where the virtual consistency is strictly stronger in terms of lower bound than every one (according to Theorem 4.1). The following theorem will show that for the five remaining soft consistencies associated with every hard consistency:

- The existential directional consistency is strictly stronger than both the existential and the full directional ones
- The full directional consistency is strictly stronger than both the non-directional and the directional ones
- Other pairs of consistencies whose relationship is not implied by transitivity are incomparable: in general, every directional consistency is incomparable with every non-directional one, every existential consistency is incomparable with every non-directional, directional and full directional one.

**Theorem 4.4 (Horizontal comparison)** *Consider 2 hard consistencies $\overline{X}, \overline{Y} \in \{$ AC, RPC, PIC, maxRPC $\}$. Let $X, DX, FDX, EX, EDX$ be the non-directional, directional, full directional, existential, existential directional consistency of $\overline{X}$, and $Y, DY, FDY$ be the non-directional, directional, full directional consistency of $\overline{Y}$.*

*a. (column 2-1): $X \not\geq DY$*
*b. (column 3-1): $FDX > X$*
*c. (column 3-2): $FDX > DX$*
*d. (column 4-1,2,3): $EX \not\geq Y, DY, FDY$*
*e. (column 5-3): $EDX > FDX$*
*f. (column 5-4): $EDX > EX$*

**Proof 12**

*a. (column 2-1): $X \not\geq DY$. Figure 4.8 shows a problem which satisfies AC, RPC, PIC and maxRPC, but does not satisfy DAC, DRPC, DPIC and DmaxRPC because of $i_a$. Conversely, Figure 4.9 shows a WCSP which satisfies DAC, DRPC, DPIC and DmaxRPC, but does not satisfy AC, RPC, PIC and maxRPC. Thus, every non-directional consistency is incomparable with every directional consistency.*

*b (column 3-1): $FDX > X$. We have that FDmaxRPC $\geq$ maxRPC, FDPIC $\geq$ PIC, FDRPC $\geq$ RPC, FDAC $\geq$ AC from the definition of these full directional consistencies. Figure 4.8 shows a problem which is maxRPC, PIC, RPC, AC but is not FDmaxRPC, FDPIC, FDRPC, FDAC. Thus, FDmaxRPC $>$ maxRPC, FDPIC $>$ PIC, FDRPC $>$ RPC, FDAC $>$ AC.*

*c (column 3-2): $FDX > DX$. We have that FDmaxRPC $\geq$ DmaxRPC, FDPIC $\geq$ DPIC, FDRPC $\geq$ DRPC, FDAC $\geq$ DAC from the definition of these full directional consistencies. Figure 4.9 shows a problem which is DmaxRPC, DPIC, DRPC, DAC but is not FDmaxRPC, FDPIC, FDRPC, FDAC. Thus, FDmaxRPC $>$ maxRPC, FDPIC $>$ PIC, FDRPC $>$ RPC, FDAC $>$ AC.*

*d. (column 4-3,2,1): $EX \not\geq Y, DY, FDY$. Figure 4.10 shows a problem which satisfies the full directional AC/RPC/PIC/maxRPC, but does not satisfy the existential ones. Conversely, the problem in figure 4.11 satisfies the existential AC/RPC/PIC/-*

*maxRPC but does not satisfy the non-directional, directional and full directional ones. Thus, every existential consistency is incomparable with every non-directional, directional and full directional one.*

*e,f. (column 5-3, 5-4): $EDX > FDX$ and $EDX > EX$. The proof is trivial based on the definitions.*

### Diagonal comparison

Now, we will show that for any other pair of consistencies which is not covered by Theorem 4.3 or 4.4, the consistencies are incomparable. To prove that two consistencies $A$ and $B$ are incomparable, it is enough to show 2 WCSPs such that one satisfies $A$ but not $B$, and the other satisfies $B$ but not $A$.

- FDAC $\ngtr$ RPC,PIC,maxRPC, DRPC,DPIC,DmaxRPC.
  - Figure 4.5: FDAC holds but RPC,PIC,maxRPC, DRPC,DPIC,DmaxRPC do not.
  - Figure 4.8: RPC,PIC,maxRPC hold but FDAC does not.
  - Figure 4.9: DRPC,DPIC,DmaxRPC hold but FDAC does not.
- FDRPC $\ngtr$ PIC,maxRPC, DPIC,DmaxRPC.
  - Figure 4.6: FDRPC holds but PIC,maxRPC, DPIC,DmaxRPC do not
  - Figure 4.8: PIC,maxRPC hold but FDRPC does not.
  - Figure 4.9: DPIC,DmaxRPC hold but FDRPC does not.
- FDPIC $\ngtr$ maxRPC, DmaxRPC.
  - Figure 4.7: FDPIC holds but maxRPC, DmaxRPC do not
  - Figure 4.8: PIC,maxRPC hold but FDRPC does not
  - Figure 4.9: DPIC,DmaxRPC hold but FDRPC does not
- EDAC $\ngtr$ (E/FD/D/-)(RPC/PIC/maxRPC)
  - Figure 4.5: is EDAC but is not (E/FD/D/-)(RPC/PIC/maxRPC)
  - Figure 4.10: is not EDAC but is (FD/D/-)(RPC/PIC/maxRPC)
  - Figure 4.11: is not EDAC but is E(RPC/PIC/maxRPC)
- EDRPC $\ngtr$ EPIC,EmaxRPC, FDPIC,FDmaxRPC, DPIC,DmaxRPC, PIC,maxRPC.
  - Figure 4.6: is EDRPC but is not EPIC,EmaxRPC, FDPIC, FDmaxRPC, DPIC, DmaxRPC, PIC, maxRPC.
  - Figure 4.10: is not EDRPC but is FDPIC,FDmaxRPC, DPIC,DmaxRPC, PIC,maxRPC
  - Figure 4.11: is not EDRPC but is EPIC,EmaxRPC.
- EDPIC $\ngtr$ EmaxRPC, FDmaxRPC, DmaxRPC, maxRPC.
  - Figure 4.7: is EDPIC but is not EmaxRPC, FDmaxRPC, DmaxRPC, maxRPC.
  - Figure 4.10: is not EDPIC but is FDmaxRPC, DmaxRPC, maxRPC.
  - Figure 4.11: is not EDPIC but is EmaxRPC but is not EDPIC.
- VAC $\ngtr_{c_\varnothing}$ (ED/E/FD/D/-)(RPC/PIC/maxRPC)
  - Figure 4.5: is VAC but is not (ED/E/FD/D/-)(RPC/PIC/maxRPC).
  - Figure 4.12: is not VAC but is (ED/E/FD/D/-)(RPC/PIC/maxRPC).
- VRPC $\ngtr_{c_\varnothing}$ (ED/E/FD/D/-)(PIC/maxRPC)
  - Figure 4.6: is VRPC but is not (ED/E/FD/D/-)(PIC/maxRPC).
  - Figure 4.12: is not VRPC but is (ED/E/FD/D/-)(PIC/maxRPC).
- VPIC $\ngtr_{c_\varnothing}$ (ED/E/FD/D/-)maxRPC
  - Figure 4.7: is VPIC but is not (ED/E/FD/D/-)maxRPC
  - Figure 4.12: is not VPIC but is (ED/E/FD/D/-)maxRPC

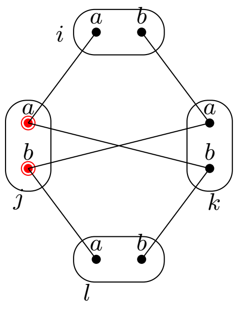


*Figure 4.5: A WCSP which satisfies all arc consistencies but does not satisfy any soft RPC (hence does not satisfy any soft PIC, maxRPC). $j < k < i < l$. The problem does not satisfy any soft RPC because of variable $j$ (the unique support $(j_a, k_a)$ of $(j, a)$ in $c_{jk}$ is not simply extensible on $i$ and the unique support $(j_b, k_b)$ of $(j, b)$ is not simply extensible on $l$.)*

| | | | | | |
|---|---|---|---|---|---|
| AC | DAC | FDAC | EAC | EDAC | VAC |
| ~~RPC~~ | ~~DRPC~~ | ~~FDRPC~~ | ~~ERPC~~ | ~~EDRPC~~ | ~~VRPC~~ |
| ~~PIC~~ | ~~DPIC~~ | ~~FDPIC~~ | ~~EPIC~~ | ~~EDPIC~~ | ~~VPIC~~ |
| ~~maxRPC~~ | ~~DmaxRPC~~ | ~~FDmaxRPC~~ | ~~EmaxRPC~~ | ~~EDmaxRPC~~ | ~~VmaxRPC~~ |

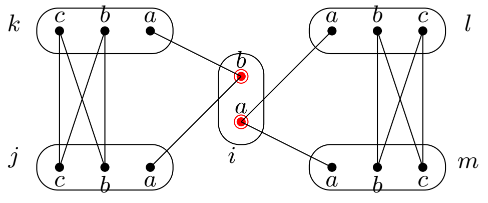


*Figure 4.6: A WCSP which satisfies all RPC consistencies but does not satisfy any PIC consistency. $i < j < k < l < m$. Every value of $i$ satisfies RPC consistencies because it has more than 2 full (hence simple) arc supports in $c_{ik}, c_{ij}, c_{il}, c_{im}$. The problem does not satisfy any PIC consistency because of variable $i$ (value $(i, a)$ is not normally (hence not fully) extensible to triangle $\Delta_{ilm}$ while $(i, b)$ is not simply (hence not fully) extensible to triangle $\Delta_{ijk}$)*

| | | | | | |
|---|---|---|---|---|---|
| RPC | DRPC | FDRPC | ERPC | EDRPC | VRPC |
| ~~PIC~~ | ~~DPIC~~ | ~~FDPIC~~ | ~~EPIC~~ | ~~EDPIC~~ | ~~VPIC~~ |
| ~~maxRPC~~ | ~~DmaxRPC~~ | ~~FDmaxRPC~~ | ~~EmaxRPC~~ | ~~EDmaxRPC~~ | ~~VmaxRPC~~ |

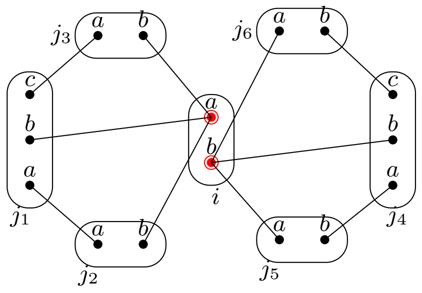


*Figure 4.7: A WCSP which satisfies all PIC consistencies but does not satisfy any maxRPC consistency. $i < j_1 < j_2 < j_3 < j_4 < j_5 < j_6$. There are only zero unary costs in this problem, thus simple and full supports (or witnesses) are identical. The problem is EDPIC since both $(i,a),(i,b)$ can 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 $\Delta_{ij_1j_3}$; this is the same for value $(i,b)$ in $c_{ij_4}$).*

| | | | | | |
|---|---|---|---|---|---|
| RPC | DRPC | FDRPC | ERPC | EDRPC | VRPC |
| PIC | PIC | FDPIC | EPIC | EDPIC | VPIC |
| ~~maxRPC~~ | ~~DmaxRPC~~ | ~~FDmaxRPC~~ | ~~EmaxRPC~~ | ~~EDmaxRPC~~ | ~~VmaxRPC~~ |

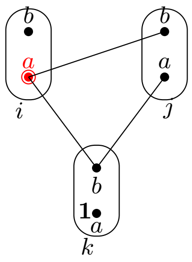


*Figure 4.8: A WCSP which is non-directional consistent but is not directional consistent. $i < j < k$. The problem is not DAC because value $(i,a)$ has no full arc support in $c_{ik}$. Therefore, it does not satisfy FDAC, EDAC, FDRPC, EDRPC, FDPIC, EDPIC, FDmaxRPC, EDmaxRPC. However, the problem is maxRPC (hence PIC, RPC) because it is AC and every domain value is normally extensible to the triangle.*

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

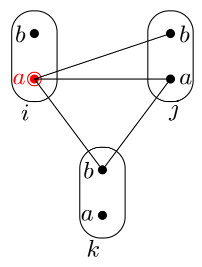


*Figure 4.9: A WCSP which is directional consistent ($i > j > k$) but is non-directional inconsistent. The problem is not AC because $(i, a)$ has no arc support in $c_{ij}$. However, the problem is DAC because every value of $j$ and $k$ has full arc support in $c_{ji}, c_{ki}$. Moreover, the problem is DmaxRPC (hence DPIC, DRPC) because every value of $j$ and $k$ can be fully extended on the triangle (in the triangle $\Delta_{ijk}$, only the smallest variable $k$ and $c_{ki}, c_{kj}$ are interested by high order directional consistencies).*

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

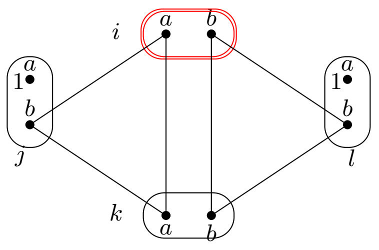


*Figure 4.10: A WCSP which is full directional consistent but is existential inconsistent ($l < j < k < i$). The problem is not EAC (hence not ERPC, EPIC, EmaxRPC) because of value $i$ ($i_a$ has no full support in $c_{ij}$ while $i_b$ has no full support in $c_{il}$). However, the problem is FDmaxRPC (hence FDPIC, FDRPC) because it is FDAC and every value of $i, k$ can be normally extended to both 2 triangles and every value of $j, l$ can be fully extended to $\Delta_{jik}$ and $\Delta_{lik}$ respectively.*

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

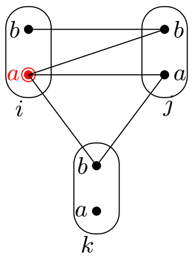


*Figure 4.11: A WCSP which is existential consistent but is not full directional consistent. $i > j > k$. The problem is not AC (hence is not RPC, PIC, maxRPC) because of value $(i,a)$ (has no arc support in $c_{ij}$) and is not DAC (hence is not DRPC, DPIC, DmaxRPC) because of value $(j,b)$ (has no 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~~ |
| ~~RPC~~ | ~~DRPC~~ | ~~FDRPC~~ | ERPC | ~~EDRPC~~ |
| ~~PIC~~ | ~~DPIC~~ | ~~FDPIC~~ | EPIC | ~~EDPIC~~ |
| ~~maxRPC~~ | ~~DmaxRPC~~ | ~~FDmaxRPC~~ | EmaxRPC | ~~EDmaxRPC~~ |

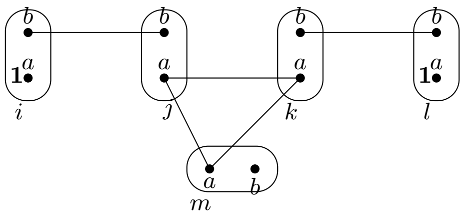


*Figure 4.12: A WCSP which is existential directional but is not virtual consistent $l < i < j < k < m$. The problem is not VAC (hence not VRPC, VPIC, VmaxRPC) because AC makes Bool(P) wiped-out at $j$ or $k$. Conversly, the problem is EDmaxRPC where variables $j, m, k$ are FDmaxRPC in $\Delta_{ijk}$ and $i_b, j_a, k_a, l_b, m_a$ are EmaxRPC supports of variables.*

| | | | | | |
|---|---|---|---|---|---|
| AC | DAC | FDAC | EAC | EDAC | ~~VAC~~ |
| RPC | DRPC | FDRPC | ERPC | EDRPC | ~~VRPC~~ |
| PIC | DPIC | FDPIC | EPIC | EDPIC | ~~VPIC~~ |
| maxRPC | DmaxRPC | FDmaxRPC | EmaxRPC | EDmaxRPC | ~~VmaxRPC~~ |

## 4.4 Enforcing algorithms

Two algorithms enforcing EDPIC and EDmaxRPC will be introduced in this thesis. RPCs have not been implemented because of the costly maintainance of the number of arc supports per value in each cost function in WCSPs. This number can both increase and decrease during enforcing RPCs because moving costs between cost functions of different arities can iteratively break or create arc supports for values.

High order consistencies are only enforced inside triangles of variables that are connected two-by-two by binary cost functions regardless of whether or not variables are connected by ternary cost functions. A triangle over three variables $i, j, k$, is noted $\Delta_{ijk}$. The notations $\Delta_{ijk}, \Delta_{ikj}, \Delta_{jki}, \Delta_{jik}, \Delta_{kij}, \Delta_{kji}$ represent the same triangle over $i, j, k$. The common idea for enforcing supports and witnesses is to move costs of triangles $\Delta_{ijk}$ (consisting of binary costs $c_{ij}, c_{ik}, c_{jk}$, ternary costs if $c_{ijk}$ exists, and possibly unary costs $c_j, c_k$) to inconsistent values of $i$. The two following notations express such a combined cost of triangles.

- $\Delta_{ijk}(a,b,c) = c_{ij}(a,b) + c_{jk}(b,c) + c_{ik}(a,c) + c_{ijk}(a,b,c)$: denotes the sum of binary and ternary costs involved in the instantiation $(i_a, j_b, k_c)$, where $c_{ijk}(a,b,c) = 0$ if $c_{ijk}$ does not exist.
- $\Lambda_{ijk}(a,b,c) = c_{ik}(a,c) + c_{jk}(b,c) + c_{ijk}(a,b,c)$: denotes the same amount of cost as $\Delta_{ijk}(a,b,c)$ but excluding $c_{ij}(a,b)$.

### 4.4.1 Equivalence Preserving Tranformations

In order to move costs from a triangle $\Delta_{ijk}$ to a unary cost function $c_i$, binary costs involved in the triangle are firstly extended to the ternary cost function $c_{ijk}$ and then ternary costs are projected to $c_i$. Thus, beside soft arc consistency operations defined for enforcing arc consistencies, we need extra EPTs for shifting cost between ternary and binary cost functions as well as between ternary and unary cost functions. We have renamed SAC operations to better distinguish them from the new operations. Algorithm 4.1 presents all the basic operations that will be used in our algorithms enforcing EDPIC and EDmaxRPC.

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

SAC operations Extend1To2 and Project2To1 are renamed and represented here because they need to store extra information (queues of variables $P, S$). The queues of variables $P, S, T$ will be used in our algorithms enforcing high order consistencies. They will be explained later and thus can be ignored for the moment.

**Algorithme 4.1 :** Elementary operations

```
1  Procedure Extend1To2($i, a, c_{ij}, \alpha$)
2      // precondition: $c_i(a) \geq \alpha$
3      foreach $b \in D(j)$ do $c_{ij}(a,b) \leftarrow c_{ij}(a,b) + \alpha$;
4      $c_i(a) \leftarrow c_i(a) - \alpha$;
5      $T \leftarrow T \cup \{c_{ij}\}$;
6  Procedure Extend2To3($i, a, j, b, c_{ijk}, \alpha$)
7      // precondition: $c_{ij}(a,b) \geq \alpha$
8      foreach $c \in D(k)$ do $c_{ijk}(a,b,c) \leftarrow c_{ijk}(a,b,c) + \alpha$;
9      $c_{ij}(a,b) \leftarrow c_{ij}(a,b) - \alpha$
10 Procedure Project3To1($c_{ijk}, i, a, \alpha$)
11     // precondition: $\forall b \in D(j), c \in D(k), c_{ijk}(a,b,c) \geq \alpha$
12     foreach $b \in D(j), c \in D(k)$ do $c_{ijk}(a,b,c) \leftarrow c_{ijk}(a,b,c) - \alpha$;
13     if $c_i(a) = 0 \wedge \alpha > 0$ then
14         $P \leftarrow P \cup \{i\}$;
15         $S \leftarrow S \cup \{i\}$;
16     $c_i(a) \leftarrow c_i(a) + \alpha$;
17 Procedure Project3To2($c_{ijk}, i, a, j, b, \alpha$)
18     // precondition: $\forall c \in D(k), c_{ijk}(a,b,c) \geq \alpha$
19     foreach $c \in D(k)$ do $c_{ijk}(a,b,c) \leftarrow c_{ijk}(a,b,c) - \alpha$;
20     $c_{ij}(a,b) \leftarrow c_{ij}(a,b) + \alpha$;
21 Procedure Project2To1($c_{ij}, i, a, \alpha$)
22     // precondition: $\forall b \in D(j), c_{ij}(a,b) \geq \alpha$
23     foreach $b \in D(j)$ do $c_{ij}(a,b) \leftarrow c_{ij}(a,b) - \alpha$;
24     if $c_i(a) = 0 \wedge \alpha > 0$ then
25         $P \leftarrow P \cup \{i\}$;
26         $S \leftarrow S \cup \{i\}$;
27     $c_i(a) \leftarrow c_i(a) + \alpha$;
28 Procedure isSmallest($i, \Delta_{ijk}$)
29     return $((i < j) \wedge (i < k))$;
30 Procedure PruneVars()
31     foreach $a \in D(i)$ do
32         if $c_i(a) + c_\varnothing \geq m$ then
33             $D(i) \leftarrow D(i) - \{a\}$;
34             $Q \leftarrow Q \cup \{i\}$;
```

### 4.4.2 Enforcing soft path inverse consistencies

#### Enforcing PIC supports

RPCs and maxRPCs are defined for variables $i$ in triangles sharing a same pair of variables (involving $i$ and another variable) while PIC consistencies are simply defined for variables $i$ in triangles involving $i$. Thus, PIC consistencies are symmetric w.r.t triangles: supports and witnesses are considered equal and will be combined. A PIC support of a value $i_a$ on a triangle $\Delta_{ijk}$ is a pair of values $(j_b, k_c)$ such that $\Delta_{ijk}(a,b,c) = 0$. A full PIC support of $i_a$ on $\Delta_{ijk}$ is a pair of values $(j_b, k_c)$ such that $\Delta_{ijk}(a,b,c) + c_j(b) + c_k(c) = 0$.

The enforcement for PIC supports is based on the idea of enforcing EDAC for ternary cost functions proposed in [Sánchez et al., 2008]. Unary and binary costs are taken into account in the definition of full arc supports. Thus, a full arc support is exactly a full PIC support when the three variables are connected by a ternary cost function, and one-by-one by a binary cost function. Full arc supports are enforced by:

(1) extending non-zero unary costs to binary cost functions involved in the ternary cost function,

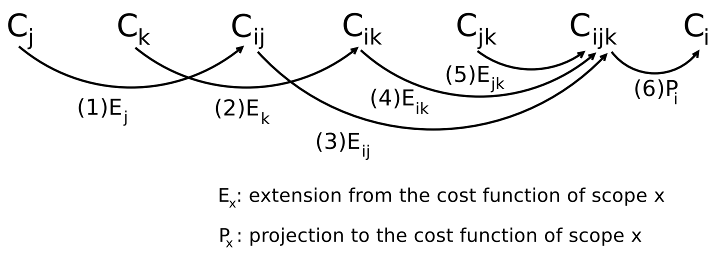
(2) extending non-zero binary costs to the ternary cost function and

(3) projecting ternary costs to values of the variable to be enforced.

Based on this principle, we proposed algorithms for enforcing PIC supports.

**Enforcing simple PIC supports** Simple PIC supports are enforced by Procedure findPICSupport in Algorithm 4.2. To create a simple PIC support for a value $i_a$ on $\Delta_{ijk}$, binary and ternary costs involved in $\Delta_{ijk}$ are moved to $i_a$ in such a way that there is an instantiation $(i_a, j_b, k_c)$ whose ternary and binary costs decrease to 0. The order in which costs are moved between cost functions to enforce simple PIC support is presented in Figure 4.13. Firstly, binary costs $c_{ij}, c_{ik}, c_{jk}$ are extended on ternary cost function $c_{ijk}$ by the procedure Extend2To3 (line 10–12). Then, ternary costs $c_{ijk}$ are projected on $i_a$ by Procedure Project3To1 (line 10). The maximum possible cost projected on each value $a \in D(i)$, stored in $P_i[a]$, is computed based on available binary and ternary costs (line 3). Binary cost extensions $E_{ij}, E_{ik}, E_{jk}$ are computed based on $P_i[a]$, the ternary and binary costs on two other sides of the triangle (line 5–9). Each extension is strong enough in the sense that a stronger extension cannot lead to a projection on $i_a$ greater than $P_i[a]$. This extension is also minimum in the sense that a weaker extension would result in negative costs. The last condition guarantees that for each binary cost extension $E_{ij}(a,b), E_{ik}(a,c)$ or $E_{jk}(b,c)$, there exists a value in the remaining variable $k_c, j_b$ or $i_a$ respectively such that the final resulting ternary cost $c_{ijk}(a,b,c) + E_{ij}(a,b) + E_{ik}(a,c) + E_{jk}(b,c) - P_i[a] = 0$. Therefore, 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.

**Enforcing full PIC supports** Full PIC supports are enforced by Procedure findFullPICSupport in Algorithm 4.2 in a way similar to Procedure findPICSupport. The difference is that unary costs of $j$, $k$ are extended on binary functions $c_{ij}$ and $c_{ik}$ respectively, by Procedure Extend1To2, in order to create full PIC supports with zero unary costs



C_j
C_k
C_ij
C_ik
C_jk
C_ijk
C_i
(1)E_j
(2)E_k
(3)E_ij
(4)E_ik
(5)E_jk
(6)P_i
E_x: extension from the cost function of scope x
P_x: projection to the cost function of scope x

*Figure 4.13: The order of cost movements for enforcing simple or full PIC supports, where unary cost extensions are not included in the enforcement of simple PIC supports.*

(line 22, 23). After that, binary and ternary costs are moved to $i_a$ in the same way as for enforcing simple PIC supports (line 24). The order in which costs are moved to enforce full PIC supports is also described in Figure 4.13. 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 $E_j, E_k$ (line 17,19,21). Similarly to binary cost extensions, unary cost extensions are strong enough to lead to a cost projection $P_i[a]$ without creating negative costs. This condition ensures that for any unary cost extension $E_j[b], E_k[c]$, there exists a value $a \in D(i)$ such that the final resulting 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 $T$ for enforcing PIC supports for related values later.

**Example 4.11** *Consider the Problem in Figure 4.14(a) with 4 variables $i, j, l, k$ and 5 binary cost functions $c_{ij}, c_{ik}, c_{il}, c_{jk}, c_{jl}$ such that $i < j < k < l$. Each variable has 2 values $a, b$. Binary costs are represented by edges (red continuous line) and ternary costs are represented by hyper edges (blue dashed lines for $c_{ijk}$ and green dashed lines for $c_{ijl}$). The absence of (hyper)edges indicates a zero cost. The number beside values, edges and hyper-edges present the corresponding unary, binary and ternary costs respectively. The initial problem is FDAC but not FDPIC because value $(i, a)$ cannot be fully extended on $\Delta_{ijk}$. Now, consider enforcing full PIC supports for the values of variable $i$.*

*Procedure* findFullPICSupport$(i, j, k)$ *computes the amounts of cost for projections/extensions: $P_i[a] = E_j[b] = 1$. Other projection/extension costs are zero. After extending a cost of 1 from $j_b$ on $c_{ij}$, it will call Procedure* findPICSupport$(i, j, k)$ *and compute the amounts of cost projections/extensions as follows:*

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

*The resulting problem, presented in the sub-figure 4.14(d) is still not FDPIC because value $(i, b)$ cannot be fully extended on triangle $\Delta_{ijl}$. Then Procedure* findFullPICSupport$(i, j, l)$ *computes the following projection/extension costs:*

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

*The final problem, presented in the sub-figure 4.14(g) is FDPIC.*

Notice that enforcing both simple and full PIC supports can create new ternary functions among 3 variables of triangles due to binary extensions, e.g., ternary cost functions $c_{ijk}, c_{ijl}$ are created during enforcing. If a binary cost needs to be extended to a ternary cost function $c_{ijk}$ and if $c_{ijk}$ does not exist, $c_{ijk}$ will be created and initialized with empty costs for every

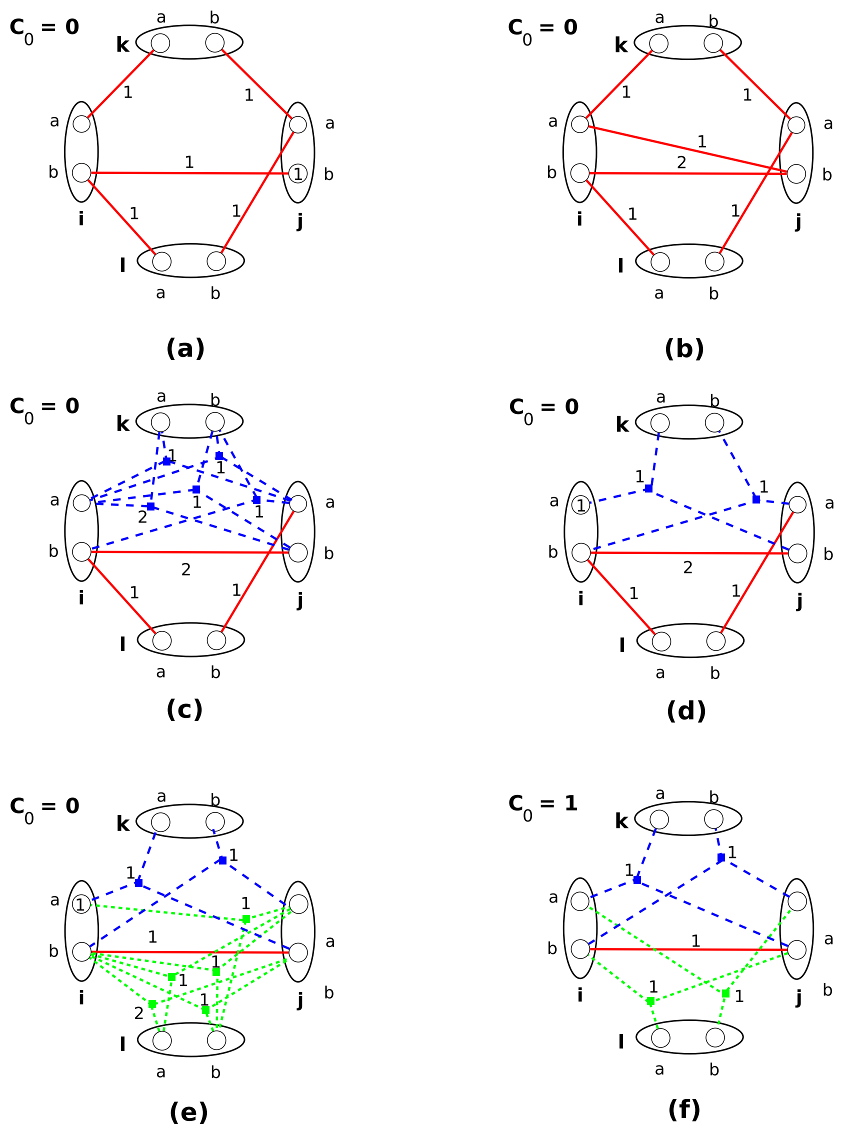


*Figure 4.14: Cost evolution in a WCSP during the enforcement of full PIC supports (a) original problem with 5 binary cost functions $c_{ij}, c_{ik}, c_{il}, c_{jk}, c_{jl}, i < j < k < l\}$. It is FDAC but not FDPIC because of variable $i$ where $(i,a)$ and $(i,b)$ cannot be fully extended on $\Delta_{ijk}$ and $\Delta_{ijl}$ 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.*

tuples over $(i, j, k)$. Therefore, enforcing PIC consistencies can change the structure of the network by adding new ternary cost functions. This is a difference between enforcing soft PIC consistencies and hard PIC.

### Enforcing EDPIC

EDPIC can be enforced in binary and ternary WCSPs by Procedure enforceEDPIC in Algorithm 4.3. This procedure makes the use of 4 propagation queues $Q, P, S$ and $T$ for storing variables which have had some change in value domain or in unary cost function.

- If $i \in Q$, it means that some value of $D(i)$ has been deleted (Algorithm 4.1, Procedure PruneVars, line 34). Neighbors of $i$ may have lost their simple PIC support and need to be revised for PIC consistency.
- If $i \in P$, it means that some value of $D(i)$ has increased its cost from 0 (Algorithm 4.1, Procedure Project3To1 at line 14 and Project2To1 at line 25). Neighbor variables lower than $i$ may have lost their full PIC support and need to be checked for DPIC.
- With the same content as $P$, the queue $S$ is used to efficiently build the propagation queue $R$ that will be used for enforcing EPIC i.e., for propagating the loss of EPIC supports. $S$ contains variables $i$ such that the unary cost of some value in $D(i)$ has been increased from 0 (Algorithm 4.1, Procedure Project3To1 at line 15 and Procedure Project2To1 at line 26), while $R$ contains variables that need to be checked for EPIC. If $i \in S$, the value in $D(i)$ that has increased its unary cost may be the existential support of $i$ and thus $i$ needs to be checked for EPIC. On the other hand, the EPIC support of neighboring variables $j$ may be fully supported by this value. Thus, neighboring variables $j$ also need to be checked for EPIC. In summary, all variables of $S$ and their neighbors are pushed into $R$ to be checked for EPIC (Algorithm 4.3, Procedure enforceEDPIC line 4).
- If $c_{ij} \in T$, it means the binary cost function $c_{ij}$ has been modified because of an unary cost extension from a greater variable between $i$ and $j$ to $c_{ij}$, where some binary cost of $c_{ij}$ may have increased from 0 (Procedure Extend1To2, Algorithm 4.1, line 5). So, $i$, $j$ and their common neighbors may have lost simple PIC support and need to be revised for PIC. Note that this modification in binary costs does not lead to the loss of full PIC supports because full PIC supports only contain values of zero costs from which unary cost extensions to binary cost function cannot occur. Thus, DPIC and EPIC are preserved and do not need to be rechecked after this modification in binary costs.

EDPIC can be enforced by simply enforcing EPIC, DPIC and PIC simultaneously. Procedure enforceEDPIC consists of four inner-while loops and one for-loop to enforce respectively EPIC, DPIC, PIC and NC.

- The first while-loop (line 5-7) aims to enforce EPIC. It firstly 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 $i$, it does not care about triangles in which $i$ is not the smallest, because DPIC takes care of them. In other words, EPIC is only responsible for triangles on which the considered variable is not the smallest. That's why the property of EPIC is only checked for such triangles (Algorithm 4.2, line 26).

**Algorithme 4.2 :** Algorithms enforcing PIC supports

```
Procedure findPICSupport($i, \Delta_{ijk}$)
    foreach $a \in D(i)$ do
        $P_i[a] \leftarrow \min_{b \in D(j), c \in D(k)} \Delta_{ijk}(a,b,c)$;
    foreach $a \in D(i), b \in D(j)$ do
        $E_{ij}[a,b] \leftarrow \max_{c \in D(k)} \{P_i[a] - c_{ijk}(a,b,c) - c_{ik}(a,c) - c_{jk}(b,c)\}$;
    foreach $a \in D(i), c \in D(k)$ do
        $E_{ik}[a,c] \leftarrow \max_{b \in D(j)} \{P_i[a] - c_{ijk}(a,b,c) - c_{jk}(b,c) - E_{ij}(a,b)\}$;
    foreach $b \in D(j), c \in D(k)$ do
        $E_{jk}[b,c] \leftarrow \max_{a \in D(i)} \{P_i[a] - c_{ijk}(a,b,c) - E_{ij}(a,b) - E_{ik}(a,c)\}$;
    foreach $a \in D(i), b \in D(j)$ do Extend2To3($i, a, j, b, c_{ijk}, E_{ij}[a,b]$);
    foreach $a \in D(i), c \in D(k)$ do Extend2To3($i, a, k, c, c_{ijk}, E_{ik}[a,c]$);
    foreach $b \in D(j), c \in D(k)$ do Extend2To3($j, b, k, c, c_{ijk}, E_{jk}[b,c]$);
    foreach $a \in D(i)$ do Project3To1($c_{ijk}, i, a, P_i[a]$);
    ProjectUnary($i$);

Procedure findFullPICSupport($i, \Delta_{ijk}$)
    foreach $a \in D(i)$ do
        $P_i[a] \leftarrow \min_{b \in D(j), c \in D(k)} \{\Delta_{ijk}(a,b,c) + c_j(b) + c_k(c)\}$;
    foreach $b \in D(j)$ do
        $E_j[b] \leftarrow \max_{a \in D(i), c \in D(k)} \{P_i[a] - \Delta_{ijk}(a,b,c) - c_k(c)\}$;
    foreach $c \in D(k)$ do
        $E_k[c] \leftarrow \max_{a \in D(i), b \in D(j)} \{P_i[a] - c_{ijk}(a,b,c) - E_j[b]\}$;
    foreach $b \in D(j)$ do Extend1To2($j, b, c_{ji}, E_j[b]$) ;
    foreach $c \in D(k)$ do Extend1To2($k, c, c_{ki}, E_k[c]$) ;
    findPICSupport($i, \Delta_{ijk}$);

Procedure findEPICSupport($i$)
    $\alpha \leftarrow \min_{a \in D(i)} \{c_i(a) + \sum_{\Delta_{ijk}, i>j \text{ or } i>k} \min_{b \in D(j), c \in D(k)} \{\Delta_{ijk}(a,b,c) + c_j(b) + c_k(c)\}\}$ ;
    if $\alpha > 0$ then
        foreach $\Delta_{ijk}$ do
            if $\neg$isSmallest($i, \Delta_{ijk}$) then findFullPICSupport($i, \Delta_{ijk}$);
            $R \leftarrow R \cup \bigcup_{\Delta_{ijk}} \{j, k\}$;
        UnaryProject($i, \alpha$);
```

**Algorithme 4.3 :** Algorithm enforcing EDPIC

```
Procedure enforceEDPIC()
    $S \leftarrow P \leftarrow Q \leftarrow X$;  $T \leftarrow \varnothing$;
    while $Q \neq \varnothing$ or $P \neq \varnothing$ or $S \neq \varnothing$ or $T \neq \varnothing$ do
        $R \leftarrow S \cup \bigcup_{i \in S, \Delta_{ijk}} \{j, k\}$;
        while $R \neq \varnothing$ do
            $i \leftarrow R$.popmin();
            findEPICSupport($i$);
        while $P \neq \varnothing$ do
            $j \leftarrow P$.popmax();
            foreach $\Delta_{ijk}$ do
                if isSmallest($i, \Delta_{ijk}$) then findFullPICSupport($i, \Delta_{ijk}$);
                if isSmallest($k, \Delta_{ijk}$) then findFullPICSupport($k, \Delta_{ijk}$);
        while $Q \neq \varnothing$ do
            $j \leftarrow Q$.popmin();
            foreach $\Delta_{ijk}$ do
                if ¬isSmallest($i, \Delta_{ijk}$) then findPICSupport($i, \Delta_{ijk}$);
                if ¬isSmallest($k, \Delta_{ijk}$) then findPICSupport($k, \Delta_{ijk}$);
        while $T \neq \varnothing$ do
            $c_{ij} \leftarrow T$.pop();
            foreach $\Delta_{ijk}$ do
                if ¬isSmallest($i, \Delta_{ijk}$) then findPICSupport($i, \Delta_{ijk}$);
                if ¬isSmallest($j, \Delta_{ijk}$) then findPICSupport($j, \Delta_{ijk}$);
                if ¬isSmallest($k, \Delta_{ijk}$) then findPICSupport($k, \Delta_{ijk}$);
        PruneVars()
```

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 triangles in which $i$ is not the smallest variable (Algorithm 4.2, line 29). The EPIC supports of neighboring variables of $i$ can also be destroyed due to new values of non-zero cost made by the enforcement of full PIC supports on $i$. Thus, all neighbors of $i$ are pushed back to $R$ to be checked for EPIC later (Algorithm 4.2, line 30).

- DPIC is enforced by the second while-loop at line 8. For a variable $j \in P$, only its neighboring variables that are connected to $j$ by a triangle $\Delta_{ijk}$ (line 10) and is smallest variable among $i, j, k$ (line 11, 12) are considered for checking for DPIC. This condition guarantees that the enforcement for DPIC is performed in one direction w.r.t the DAC order in such a way that unary costs are only extended from greater variables to smaller ones. Moreover, this condition ensures the termination of the algorithm (proved later).

- PIC is enforced by two while-loops at lines 13 and 18. For a variable $j \in Q$, every neighboring variable of $i$ is checked for PIC. For a pair of variables $(i, j) \in T$, $i, j$ and every variable connected to both $i$ and $j$ are checked for PIC. Simple PIC supports are enforced in the reverse direction of the DAC order (line 16 – 17, line 21– 23), i.e., simple PIC supports are only enforced for a variable in a triangle in which the variable is not the smallest one.

### Enforcing other PIC consistencies

By reducing Algorithm 4.3 which enforces EPIC, we can obtain 4 algorithms for enforcing other consistencies EPIC, FDPIC, DPIC and PIC.

- EPIC: the algorithm enforcing EPIC keeps only the first while-loop of Algorithm 4.3 (lines 4– 7). However, to search for a PIC support for a variable $i$, Procedure findEPICSupport in Algorithm 4.2 is now responsible for all triangles involving $i$, regardless of whether $i$ is the smallest variable among 3 variables of triangles or not. The condition ¬isSmallest at line 26 (for computing the EPIC property, i.e., $\alpha$) and at line 29 must be removed from this procedure.
- FDPIC: the algorithm enforcing FDPIC removes the first loop concerning EPIC at line 5 from Algorithm 4.3 and keeps the rest.
- DPIC: the algorithm enforcing DPIC only keeps the while-loop at line 8 of Algorithm 4.3.
- PIC: the algorithm enforcing PIC only keeps the while-loop at line 13 of Algorithm 4.3. It does not use the while-loop at line 18 because the extensions of unary costs on binary cost functions never occur when enforcing simple PIC supports, i.e with PIC.

Each of these algorithms also use Procedure PruneVars at line 24 of Algorithm 4.3 to enforce NC. Notice that the while-loop at line 18 of Algorithm 4.3, that propagates the increase in binary costs, is only used for FDPIC and EDPIC. The reason is that the increase of binary costs caused by unary cost extensions can only lead to the loss of simple PIC supports but cannot lead to the loss of full PIC supports, because full PIC supports only involve zero unary costs which cannot be used for extensions. Because simple PIC supports are used by PIC, FDPIC and EDPIC, these consistencies must be revised after the increase in binary costs caused by unary cost extensions. However unary cost extensions do not occur with PIC, only FDPIC and EDPIC have to be revised.

### Complexity

In this section, we analyze the complexity of the algorithms enforcing PIC, DPIC and EPIC, and prove the termination of the algorithms enforcing FDPIC and EDPIC.

#### Algorithm enforcing PIC

In the PIC algorithm (corresponding to the loop at line 13 of Algorithm 4.3), a triangle $\Delta_{ijk}$ is processed whenever $i, j$ or $k$ are in $Q$. A variable is pushed into $Q$ only when one of its values have been removed. Thus, each variable is pushed into $Q$ at most $d$ times and each triangle is processed at most $3d$ times. A call to findPICSupport takes $O(d^3)$ in time, the total time complexity of the PIC algorithm is $O(cd^4)$ where $c$ is the number of triangles.

#### Algorithm enforcing DPIC

$P$ is a priority queue in which variables are arranged with respect to the DAC order. When a variable $j$ is popped out from $P$, all variables before (greater than) it have been processed. Only variables smaller than $j$ are processed and can increase its unary costs. Thus, $j$ and bigger variables are never pushed into $P$ again. In other words, each variable is pushed into $P$ once and the algorithm DPIC (corresponding to the loop at line 8, Algorithm 4.3) traverses $P$ once. A triangle $\Delta_{ijk}$ with $i < j, i < k$ is processed at most 2 times when one of the two bigger variables $j$ or $k$ is popped from $P$. Because the time for processing a triangle, done by findFullPICSupport, is $O(d^3)$, the DPIC algorithm has a complexity of $O(cd^3)$.

#### Algorithm enforcing FDPIC

In order to prove the termination of our algorithm enforcing FDPIC, we will compute an upper bound on its time complexity based on the features of domain consistencies as well as those of full directional algorithms. For a domain consistency, every algorithm enforcing it stops when all the values are consistent. Whenever there is an inconsistent value (not satisfying the consistency), an unary cost projection on this value will be created and the enforcing procedure will continue to revise its neighbors. Thus, the time complexity of a domain consistency can be computed by the total number of unary cost projections created during the enforcement process. FDPIC is a domain consistency, so we will compute this number of unary cost projections.

In a full directional consistency, values can both receive or send costs via unary cost projections and extensions respectively, where the maximum total cost that a value can extend cannot exceed the sum of cost that the value owns and receives. Unary costs can be extended to either $c_{\varnothing}$ or functions of higher arities where the second extensions are caused only by the directional consistency. To simplify the computation of unary cost projections which depends on unary cost extensions, in the first time, we do not take into account the costs extended to $c_\varnothing$ and then we will take into account this amount of costs in the total number of unary cost projections. Let $\mathrm{Pr}_{ij}(a), \mathrm{Ex}_{ik}(a)$ the accumulated sum of costs that a value $(i, a)$ gets from $j$ and sends to a smaller variable $k$ until the enforcing procedure stops. The final cost of $(i, a)$ is $c_i(a) + \sum_j Pr_{ij}(a) - \sum_{k,k<i} Ex_{ik}(a)$. If the cost of $(i, a)$ reaches the totally inconsistent cost $m$, it will be deleted. Thus, we have:

$$c_i(a) + \sum_j \mathrm{Pr}_{ij}(a) - \sum_{k,k<i} \mathrm{Ex}_{ik}(a) < m \tag{4.1}$$

$\mathrm{Ex}_{ik}(a)$ is the amount of cost that $c_{ki}$ lacks to project an amount of costs $P_{ki}(b)$ on every value $(k, b)$ of $k$ (from binary $c_{ki}$ or from some ternary function $c_{kij}$). We have:

$$\mathrm{Ex}_{ik}(a) \le \max_{b\in D(k)}\{\mathrm{Pr}_{ki}(b)\} \tag{4.2}$$

From (4.1) and (4.2), we have:

$$\sum_j \mathrm{Pr}_{ij}(a) < m + \sum_{k,k<i} \mathrm{Ex}_{ik}(a) < m + \sum_{k,k<i} \max_{b\in D(k)}\{\mathrm{Pr}_{ki}(b)\} \tag{4.3}$$

Now, we will prove by contradiction that for every variable $i$ and every value $(i, a)$: $\sum_j \mathrm{Pr}_{ij}(a) < 2^{i-1}m$ by analyzing variables from the smallest to the highest ones. Consider the smallest variable 1: it does not extend its costs since unary costs only can be extended to smaller variables. Thus $\sum_{k,k<1} \mathrm{Ex}_{1k}(a) = 0$ and we have:

$$\sum_j \mathrm{Pr}_{1j}(a) < m + 0 = 2^0 m \tag{4.4}$$

Suppose that for variable $i - 1$, we have:

$$\sum_j \mathrm{Pr}_{(i-1)j}(a) < 2^{i-2} m \tag{4.5}$$

Consider variable $i$, from (4.3) we have $\sum_j \mathrm{Pr}_{ij}(a) < m + \sum_{k,k<i} \max_{b\in D(k)}\{\mathrm{Pr}_{ki}(b)\}$. Moreover, we have:

$Pr_{1i}(b) < \sum_j Pr_{1j}(b) < 2^0 m$ from (4.3), thus $Pr_{1i}(b) < 2^0 m$

...

$Pr_{(i-1)i}(b) < \sum_j Pr_{(i-1)j}(b) < 2^{i-2} m$ from (4.3), thus $Pr_{(i-1)i}(b) < 2^{i-2} m$

Thus, we have:

$$\sum_j Pr_{ij}(a) < m + 2^0 m + 2^1 m + \ldots + 2^{i-2} m = 2^{i-1} m \tag{4.6}$$

In conclusion, for every variable $i$ and every value $(i, a)$: $\sum_j \mathrm{Pr}_{ij}(a) < 2^{i-1}m$. The sum of unary cost projections, denoted by Pr is:

$$Pr = \sum_{i,j,a\in D(i)} Pr_{ij}(a) < d.(2^0 + 2^1 + \ldots 2^{n-1})m < (2^n - 1)dm \tag{4.7}$$

Now, we need take into account the amount of costs projected on $c_\varnothing$. Extending a cost of 1 from a variable to $c_\varnothing$ consumes an amount of cost $d$ in the sum of unary costs of this variable. But $c_\varnothing$ cannot increase by more than $m$. Thus, the maximum amount of unary costs extended to $c_\varnothing$ is $dm$. From (4.7), we have:

$$Pr < (2^n - 1)dm + dm = 2^n dm \tag{4.8}$$

Every cost extension (from unary to binary and from binary to ternary) is always associated to/followed by unary cost projections and all are performed in Procedure findPICSupport or findFullPICSupport. Each projection operation and its associated cost extensions take $O(d^3)$ time. Thus the time complexity of our FDPIC algorithm is in $O(2^n d^4 m)$ and it terminates. It is likely that this asymptotic bound is not tight.

### Algorithm enforcing EPIC

Similarly to queue $P$ used for DPIC, queue $R$ is also a priority queue in which variables are arranged with respect to the DAC order, from the smallest one. When a variable $i$ is popped out from $R$, all variables $j$ before $i$ (i.e., smaller than $i$) and all triangles involving $i$ have been processed. Enforcing EPIC support for $i$ can increase some unary costs in $c_i$ but this increase does not lead to the loss of EPIC supports of variables $j$ smaller than $i$. Suppose that $(j,b)$ is the EPIC support of $j$ that was found when revising EPIC for $j$, and $(i_a,k_c)$ is the full PIC support of $(j,b)$ in $\Delta_{ijk}$. Because $(j,b)$ is the EPIC support of $j$, i.e., $c_j(b) = 0$, $(j_b,k_c)$ is also a full PIC support of $(i,a)$. Thus, Procedure findFullPICSupport (called from Procedure findEPICSupport Algorithm 4.2) does not project any cost on $(i,a)$. This means that enforcing EPIC support for $i$ cannot increase the unary cost of values in $D(i)$ by which the EPIC support of smaller variables is fully supported. In other words, variables smaller than $i$ are still EPIC and never pushed into $R$ again. Thus, each variable is pushed into $R$ at most once. Moreover, it is noticed that whenever EPIC is violated (i.e., $\alpha > 0$ at line 27 in Algorithm 4.2), $c_\varnothing$ will increase. Because EPIC cannot be violated more than $m$ times, line 29 in Algorithm 4.2 cannot be performed more than $m$ times. In summary, line 29 in Algorithm 4.2 is performed at most $O(\max(n,m))$ times. The time complexity for enforcing EPIC is thus $O(\max(n,m)c'd^3) = O(\max\{cd^3, mc'd^3\})$ where $c'$ is the maximum number of triangles for each variable and $c$ is the total number of triangles in the problem.

### Algorithm enforcing EDPIC

(a) The EDPIC algorithm iteratively performs EPIC and FDPIC code blocks corresponding to the loop at line 5 and the loops at lines 8, 13, 18 (Algorithm 4.3) until there is no more cost movement. Whenever EPIC is violated due to a variable $i$, $c_\varnothing$ increases and $i$ will be pushed into $P$ that activates the FDPIC blocks. Thus FDPIC blocks cannot be activated more than $O(m)$ times. The total number of unary cost projections created by the FDPIC part cannot exceed $O(m \times 2^n dm) = O(2^n dm^2)$. The time complexity of the FDPIC part cannot exceed $O(2^n d^4 m^2)$.

(b) In a single EPIC loop, each variable is never pushed into $R$ twice because variables are enforced for EPIC w.r.t the DAC order from the smallest ones. When a variable $i$ is processed, only variables greater than $i$ can be pushed into $R$. Thus, each variable is enforced for EPIC at most once, leading to the fact that each triangle is checked for EPIC at most once. Because *findEPICSupport* takes $O(c'd^3)$ in time, the time complexity of a single EPIC loop is $O(cd^3)$. The EPIC loop is activated whenever there has been a unary cost projection from 0. Thus, the total of time that EPIC loop is activated by FDPIC blocks cannot exceed the number of unary cost projections caused by FDPIC, that is $O(2^n dm^2)$. In summary, the time complexity of the EPIC part cannot exceed $O(2^n cd^4 m^2)$.

From (a) and (b), we conclude that the time complexity of the whole EDPIC algorithm cannot exceed $O(2^n cd^4 m^2)$. Thus, this proves that EDPIC algorithm terminates. This bound is likely not tight.

### 4.4.3 Enforcing soft max-restricted path consistencies

#### Enforcing maxRPC supports and witnesses

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 `maxRPCSupport`$[i, a, j]$ and the witness for this support on a variable $k$ is stored in `maxRPCWitness`$[i, a, j, k]$. In our algorithm enforcing EDmaxRPC, we use a parameter named `fullLevel`, where `fullLevel` = `false` indicates the semi-fully extensible arc supports (used by FDmaxRPC) and `fullLevel` = `true` the fully extensible ones (used by EmaxRPC). We will use the following functions, as described in Algorithm 4.4:

- $\curlywedge_{ij}^{k}(a, b, \text{wit})$: searches for a witness in $D(k)$ for a pair of values $(i_a, j_b)$, and returns the combined cost, consisting of $c_{ik}, c_{jk}$, of the tuple involving this witness and the pair of values. This combined cost presents the maximum cost that can be projected on the pair of values from the binary cost functions $c_{ik}, c_{jk}$ of the triangle $\Delta_{ijk}$.

- $\ddot{\curlywedge}_{ij}^{k}(a, b, \text{wit}, \texttt{fullLevel})$: does the same work as $\curlywedge$, but takes into account the unary cost $c_k$ of witnesses in the case of (1) fully extensible arc supports (`fullLevel`=`true`) or (2) semi-fully extensible arc supports on triangles w.r.t DAC order $(i < k)$.

- $\curlywedge\!\!\curlywedge_{ij}(a, b)$: computes the maximum sum of costs that can be projected on the pair of values $(i_a, j_b)$ from all triangles $\Delta_{ijk}$ sharing $i, j$.

- $\ddot{\curlywedge\!\!\curlywedge}_{ij}(a, b, \texttt{fullLevel})$: does the same work as $\curlywedge\!\!\curlywedge$, but takes into account the unary costs of witnesses $c_k$ on extra variables $k$ according to `fullLevel` and the order between $i$ and $k$ as mentioned in $\ddot{\curlywedge}_{ij}^{k}$.

**Simple maxRPC supports** are enforced by Procedure findmaxRPCSupport in Algorithm 4.5. The main idea to enforce a simple maxRPC support for a value $(i, a)$ on $c_{ij}$ is to move costs from 2 sides $c_{ik}, c_{jk}$ of all triangles $\Delta_{ijk}$ to $c_{ij}$ via $c_{ijk}$ (line 23 – 25) and finally project costs from $c_{ij}$ to $(i, a)$ (line 26) in such a way that there exists a value $b \in D(j)$ and a value $c \in D(k)$ for each triangle $\Delta_{ijk}$ such that the binary and ternary costs involved in the tuple $(i_a, j_b, k_c)$ decrease to 0. The maximum cost that a pair of values $(i_a, j_b)$ can receive from all triangles sharing $c_{ij}$ is computed by Function $\curlywedge\!\!\curlywedge_{ij}$. The cost that pairs of values $(i_a, j_b)$, $\forall b \in D(j)$, can receive plus their available binary cost will define the maximum cost $P_i$ that can be projected to $(i, a)$ (line 14). This allows to

**Algorithme 4.4 :** Elementary functions used for enforcing maxRPC supports

```
Function $\curlywedge\!\!\curlywedge_{ij}(a, b)$
    var wit;
    return $\sum_k \{\curlywedge^k_{ij}(a, b, \text{wit})\}$;
Function $\ddot{\curlywedge\!\!\curlywedge}_{ij}(a, b, \texttt{fullLevel})$
    // only called when fullLevel = true or $i < j$
    var wit;
    return $\sum_k \{\ddot{\curlywedge}^k_{ij}(a, b, \text{wit}, \texttt{fullLevel})\}$;
Function $\curlywedge^k_{ij}(a, b, \text{wit})$
    wit ← $\text{argmin}_{c \in D(k)}\{\Lambda_{ijk}(a, b, c)\}$;
    return $\Lambda_{ijk}(a, b, \text{wit})$;
Function $\ddot{\curlywedge}^k_{ij}(a, b, \text{wit}, \texttt{fullLevel})$
    if fullLevel or $i < k$ then
        wit ← $\text{argmin}_{c \in D(k)}\{\Lambda_{ijk}(a, b, c) + c_k(c)\}$;
        return $\Lambda_{ijk}(a, b, \text{wit}) + c_k(\text{wit})$;
    else
        wit ← $\text{argmin}_{c \in D(k)}\{\Lambda_{ijk}(a, b, c)\}$;
        return $\Lambda_{ijk}(a, b, \text{wit})$;
```

compute the amount of cost that needs to be projected on each pair $(i_a, j_b)$ for such a projection to be achieved on $i_a$.

The real cost $P_{ij}[a, b]$ that a triangle $\Delta_{ijk}$ provides to $(i_a, j_b)$ is the minimum of what is needed for this pair of values $P_i - c_{ij}(a, b)$ and what can be provided for it by $\Delta_{ijk}$ (line 18). This condition guarantees that $c_{ij}$ has enough costs to make a unary cost projection $P_i$ on $i_a$ without resulting in negative costs. Moreover, if more costs are 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 25), 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 $c_{ijk}$ (line 24, 23). 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 $c_{ijk}$ (line 20, 22).

**Full maxRPC supports** According to the definition of FDmaxRPC and EmaxRPC, there are two levels for full maxRPC supports. The full support used for EmaxRPC is defined for every value $(i, a)$ in every cost function $c_{ij}$ and on every triangle $\Delta_{ijk}$ regardless whether $i < j$ or $i > j$, $i < k$ or $i > k$. This support exploits the unary cost of the supporting and witnessing values in all cases.

Conversely, the full maxRPC support used for FDmaxRPC is weaker. It is defined for every value $(i, a)$ in only cost functions $c_{ij}$ such that $i < j$ and on every triangle $\Delta_{ijk}$ regardless whether $i < k$ or $i > k$. This support always exploits the unary cost of the supporting values and only exploits the unary cost of the witnessing values in the case that $i < k$.

**Algorithme 4.5 :** Algorithms to enforce maxRPC supports

```
Procedure findFullmaxRPCSupport($i, a, j,$ fullLevel)
    // condition: $i < j$ or fullLevel = true
    $P_i \leftarrow \min_{b \in D(j)}\{c_j[b] + c_{ij}(a,b) + \ddot{\curlywedge\!\!\curlywedge}_{ij}(a,b,\texttt{fullLevel})\}$;
    foreach $b \in D(j)$ do
        $E_j \leftarrow P_i - \ddot{\curlywedge\!\!\curlywedge}_{ij}(a,b,\texttt{fullLevel}) - c_{ij}(a,b)$;
        Extend1To2($j, b, c_{ij}, E_j$);
    foreach $\Delta_{ijk}$ s.t(fullLevel $\wedge$ $\neg$isSmallest($i, \Delta_{ijk}$))
    or ($\neg$fullLevel $\wedge$ ($i < k$)) do
        foreach $c \in D(k)$ do
            $E_k \leftarrow \min(c_k[c], \max_{b \in D(j)}\{P_i - \Delta_{ijk}(a,b,c)\})$;
            Extend1To2($k, c, c_{ik}, E_k$);
    findmaxRPCSupport($i, a, j$);

Procedure findmaxRPCSupport($i, a, j$)
    $P_i \leftarrow \min_{b \in D(j)}\{c_{ij}(a,b) + \curlywedge\!\!\curlywedge_{ij}(a,b)\}$ ;
    var wit[][];
    foreach $\Delta_{ijk}$ do
        foreach $b \in D(j)$ do
            $P_{ij}[a,b] \leftarrow \min\{P_i - c_{ij}(a,b),\ \curlywedge^{k}_{ij}(a,b,wit[b][k])\}$ ;
        foreach $c \in D(k)$ do
            $E_{ik}[a,c] \leftarrow \min\{c_{ik}(a,c),\ \max_{b \in D(j)}\{P_i - c_{ijk}(a,b,c) - c_{ij}(a,b) - c_{jk}(,b,c)\}\}$ ;
        foreach $b \in D(j), c \in D(k)$ do
            $E_{jk}[b,c] \leftarrow \min\{c_{jk}(b,c),\ P_i - c_{ijk}(a,b,c) - c_{ij}(a,b) - E_{ik}[a,c]\}$ ;
        foreach $b \in D(j), c \in D(k)$ do Extend2To3($j, b, k, c, c_{ijk}, E_{jk}[b,c]$) ;
        foreach $c \in D(k)$ do Extend2To3($i, a, k, c, c_{ijk}, E_{ik}[a,c]$) ;
        foreach $b \in D(j)$ do Project3To2($c_{ijk}, i, a, j, b, P_{ij}[a,b]$) ;
    Project2To1($c_{ij}, i, a, P_i$);
    ProjectUnary($i$);
    maxRPCSupport[$i, a, j$] $\leftarrow$argmin$\{P_i\}$;
    foreach $\Delta_{ijk}$ do maxRPCWitness[$i, a, j, k$] $\leftarrow wit[k]$;

Procedure findEmaxRPCSupport ($i$)
    fullLevel $\leftarrow$ true;
    $\alpha \leftarrow \min_{a \in D(i)}\{c_i(a) + \sum_{c_{ij}} \min_{b \in D(j)}\{c_{ij}(a,b) + \ddot{\curlywedge\!\!\curlywedge}_{ij}(a,b,\texttt{fullLevel})\}\}$;
    if $\alpha > 0$ then
        foreach $c_{ij}$ do
            foreach $a \in D(i)$ do
                findFullmaxRPCSupport($i, a, j,$ fullLevel);
            $R \leftarrow R \cup \bigcup_{c_{ij}}\{j\}$;
        UnaryProject($i, \alpha$) ;
```

The parameter `fullLevel` allows to design a same procedure (findFullmaxRPCSupport in Algorithm 4.5) for enforcing these two levels of full supports, where `fullLevel = true` for the full supports used in FDmaxRPC and `fullLevel = false` for the full supports used in EmaxRPC. The unary cost of witnesses in $D(k)$ is taken into account either in the full support of EmaxRPC (`fullLevel = true`) or in the full support of FDmaxRPC (`fullLevel = false`) when $i < k$. This explains the condition at line 12 in Procedure ⅄̈ of Algorithm 4.4 and the condition at line 8 in Procedure findFullmaxRPCSupport of Algorithm 4.5.

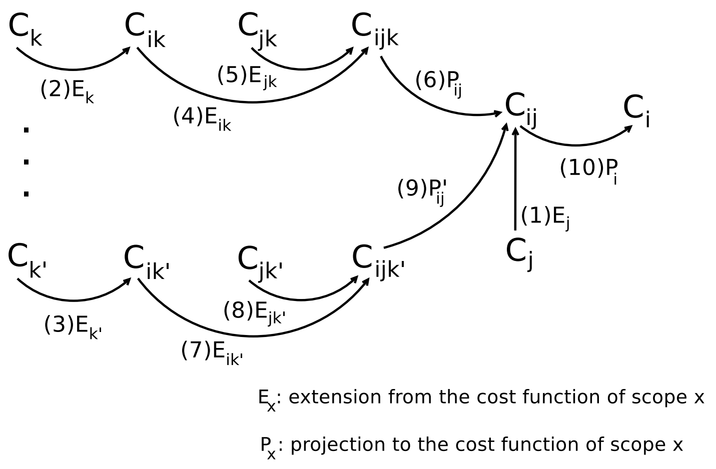


*Figure 4.15: The order of cost movements for enforcing full maxRPC supports*

In Procedure findFullmaxRPCSupport, the idea to enforce a full maxRPC support for value $(i, a)$ in a cost function $c_{ij}$ is to extend unary costs from $j$ to $c_{ij}$ (line 6) and from third variables $k$ to $c_{ik}$ (line 11). Then, costs are moved in the same way as enforcing simple maxRPC support in Procedure findmaxRPCSupport (line 12). 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 witnessing values via ⅄̈ (line 3). In order to achieve this unary projection, each value $j_b, k_c$ needs to extend respectively on $c_{ij}$ and $c_{ik}$ a amount of cost $E_j$, $E_k$ (line 5 and 10).

The order in which costs are moved when enforcing full maxRPC supports is described in Figure 4.15 where the flows indicate the direction of cost movements and the numbers under the flows indicate the order in which the corresponding cost movements are performed. The flow of moving costs for enforcing simple maxRPC supports is similar to Figure 4.15 by removing the unary cost extensions.

**Example 4.12** *Consider the problem in Figure 4.16(a) which is presented in the same way as the Figure 4.11 in the example 4.14. This problem is FDPIC but not FDmaxRPC because $i_a$ has no full AC support in $c_{ij}$ which can be extended on both $\Delta_{ijk}$ and $\Delta_{ijl}$: $(i_a, j_a)$ can be extended on $\Delta_{ijl}$ but not on $\Delta_{ijk}$ while $(i_a, j_c)$ can be extended on $\Delta_{ijk}$ but not on $\Delta_{ijl}$. The positive projection/extension costs computed by Procedure* findFullmaxRPCSupport$(i, a, j)$ *are: $P_i = 2, E_j[b] = 1$. The procedure extends a cost of 1 from $j_b$ on $c_{ij}$ and then calls* findmaxRPCSupport$(i, a, j)$ *which computes the following positive projections/extension costs:*

$$P_i = E_{jk}[a, b] = P_{ij}[a, a] = E_{jl}[c, b] = E_{jl}[c, b] = P_{ij}[a, a] = 2.$$

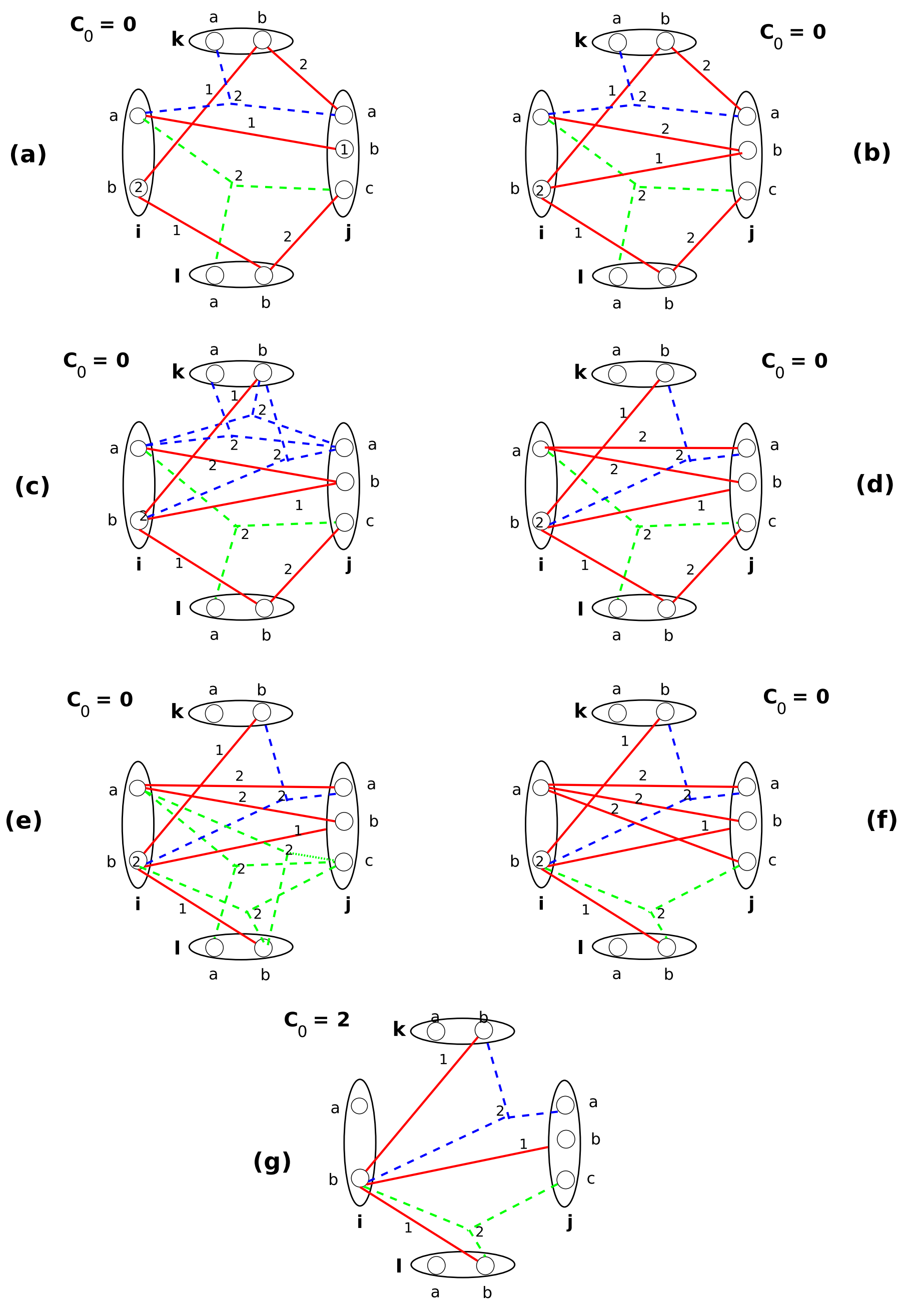


*Figure 4.16: Cost evolution in a WCSP during the enforcing of full maxRPC supports (a) original problem with 5 binary cost functions $c_{ij}, c_{ik}, c_{il}, c_{jk}, c_{jl}$ and 2 ternary functions $c_{ijk}, c_{ijl}$, $i < \{j,k,l\}$. It is FDPIC but not FDmaxRPC due to $i_a$ (no full maxRPC support 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 final problem presented in the sub-figure (g) is FDmaxRPC.*

**Enforcing maxRPC witnesses** Let $j$ be a variable whose domain has been reduced or such that some unary cost projections occurred on values of zero costs in $D(j)$. This can break the witnesses for simple or full maxRPC supports of adjacent variables $i$ in some $c_{ij}$. The check and search for new witnesses is performed by Algorithm 4.6.

Procedure findWitness_remove $(i, k, j)$ handles the case of domain reduction of $D(j)$. The procedure checks first whether or not the current support of a value $i_a$ on $c_{ik}$ is still available (line 16). If not, a new support for $a_i$ needs to be searched for. Depending on the relation between $i$ and $k$, the procedure will search for a simple or a full maxRPC support (line 25, 26). Conversely, if the current support of $i_a$ in $c_{ij}$ is still available but the current witness of $i_a$ in $c_{ij}$ on $\Delta_{ijk}$ is not available (line 18), the procedure tries to search for another witness (line 19). If there does not exist any witness for the current support of $i_a$ in $c_{ik}$, another simple or full support for $i_a$ needs to be searched for according to $i > k$ (line 22) or $i < k$ respectively (line 23).

Procedure findWitness_project $(i, k, j)$ handles the case that some unary costs $c_j$ have increased from 0. The search for witnesses is performed in the same way as done in findWitness_remove except for the fact that unary costs are taken into account to check the availability of supports and witnesses (lines 4, 6) and to check whether there exists or not another witness for replacing the current unavailable one (by Procedure $\ddot{\lambda}_{ij}^{k}$ at line 7). The parameter "`fullLevel`" used in the call to $\ddot{\lambda}_{ij}^{k}$ and in the call to findFullmaxRPCSupport is set to *semiLevel*, because Procedure findWitness_project is only activated by DmaxRPC, i.e., only called in the while-loop enforcing DmaxRPC at line 7 of Algorithm 4.7.

**Enforcing EDmaxRPC**

EDmaxRPC is enforced by Procedure enforceEDmaxRPC in Algorithm 4.7. It consists of 4 inner-while loops that handle the same propagation queues $S, P, Q, T$ used in the EDPIC enforcement algorithm.

- The inner while at line 14 enforces maxRPC by propagating domain reductions stored in the queue $Q$. For a variable $j \in Q$, some values of the neighboring variables greater than $j \in Q$ may have lost their simple maxRPC support and thus a new simple support needs to be searched for for such values (line 17- 19). Moreover, the deleted values in $j$ could have been the witness for:
  - the simple witnesses for simple maxRPC supports of neighboring values $i_a$ in $c_{ik}$ (of course $i > k$).
  - the full witnesses for full maxRPC supports of neighboring values $i_a$ in $c_{ik}$ (of course $i < k$) in the case that $i > j$.

  In summary, the support of neighboring values $i_a$ in $c_{ik}$ may have lost their witness on $j$ if $i > k$ or $i > j$ (line 20) and thus a new witness need to be search for for such supports (line 21).
- The inner while loop at line 7 processes the propagation $P$ to enforce DmaxRPC. A variable $j$ is added in $P$ if some value in $D(j)$ has increased cost from 0. The

**Algorithme 4.6 :** Algorithms to enforce maxRPC witness

```
1  Procedure findWitness_project ($i, k, j$)
       // condition:  $i < j, i < k$
       // evoked by a 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 the cost function $c_{ij}$
2      foreach $a \in D(i)$ do
3          $s \leftarrow$ maxRPCSupport$[i, a, k]$;
4          if $s \in D(k)$ and $c_k(s) + c_{ik}(a, s) = 0$ then
5              $w \leftarrow$ maxRPCWitness$[i, a, k, j]$;
6              if $w \notin D(j)$ or $c_j(w) > 0$ or $\Delta_{ikj}(a, s, w) > 0$ then
7                  if $\ddot{\curlywedge}_{ik}^{j}(a, s, \text{wit}, \text{semiLevel}) = 0$ then
8                      maxRPCWitness$[i, a, k, j] \leftarrow$ wit;
9                  else
10                     findFullmaxRPCSupport$(i, a, k, semiLevel)$;
11         else
12             findFullmaxRPCSupport$(i, a, k, semiLevel)$;

13 Procedure findWitness_remove ($i, k, j$)
       // condition:  $i > j$ or $i > k$
       // evoked by the reduction of domain $D(j)$, used to search for a
          witness in $D(j)$ for the support of values of $i$ in the cost
          function $c_{ij}$
14     foreach $a \in D(i)$ do
15         $s \leftarrow$ maxRPCSupport$[i, a, k]$;
16         if $s \in D(k)$ and $c_{ik}(a, s) = 0$ then
17             $w \leftarrow$ maxRPCWitness$[i, a, k, j]$;
18             if $w \notin D(j)$ or $\Delta_{ikj}(a, s, w) > 0$ then
19                 if ($i > k$ and $\curlywedge_{ik}^{j}(a, s, \text{wit}) = 0$) or ($i < k$ and $\ddot{\curlywedge}_{ik}^{j}(a, s, \text{wit}) = 0$) then
20                     maxRPCWitness$[i, a, k, j] \leftarrow$ wit;
21                 else
22                     if $i > k$ then findmaxRPCSupport$(i, a, k)$;
23                     else findFullmaxRPCSupport$(i, a, k, semiLevel)$;
24         else
25             if $i > k$ then findmaxRPCSupport$(i, a, k)$;
26             else findFullmaxRPCSupport$(i, a, k, semiLevel)$;
```

change in unary costs only can break full supports and the witness of full supports. Neighboring variables $i$ smaller than $j$ (line 9) can have lost full supports in $c_{ij}$ and thus new full supports need to be searched for for values of $i$ (line 11). Moreover, the full supports in $c_{ik}, i < k$ (line 12) can have lost full witnesses on $j$ if $i < j$ (line 9) and thus need to be searched for new witnesses (line 13).

- The while-inner loop at line 4 enforces EmaxRPC by processing the propagation queue $R$ built on $S$. Similarly to $P$, $S$ contains variables in which there have been some unary cost increases from 0. A variable $j \in S$ may have lost its EmaxRPC support. At the same time, the EmaxRPC support of adjacent variables $i$ may have lost the full maxRPC support in $c_{ij}$. Thus, all variables in $S$ and their neighbors which need to be checked for EPIC are pushed into $R$ (line 3). The queue $S$ allows to avoid adding all the adjacent variables. A unary cost projection is performed inside Project2To1 and Project3To1.

  Procedure findEmaxRPCSupport($i$) in Algorithm 4.5 searches for a EmaxRPC support for the variable $i$. It first checks the EmaxRPC property at line 32. If there does not exist any EmaxRPC support (line 33), the procedure will search for full maxRPC supports for any value of $i$ in any cost function $c_{ij}$ by calling findFullmaxRPCSupport with the option `fullLevel = true` (line 31). EmaxRPC does not need to take care of the triangles $\Delta_{ijk}$ in which $i$ is not the smallest variable, because DPIC takes care of such triangles. In other words, the search for full maxRPC supports for EmaxRPC consistency with `fullLevel = true` is only performed on triangles for which the considered variable has the smaller DAC order (findFullmaxRPCSupport, line 8).

- The while-inner loop at line 22 enforce maxRPC by processing the propagation queue $T$. This queue contains the binary cost functions $c_{ij}$ which have been modified by unary cost extensions from the greater variable between $i$ and $j$ on $c_{ij}$. Let $i^*$ and $j^*$ be respectively the greater and the smaller variable between $i$ and $j$. The modification in binary cost $c_{ij}$:

  - cannot break the full maxRPC supports of the smaller variable $j^*$ because the full supporting values for values of $j^*$ in $D(i^*)$ have zero cost and of course, these full supporting values cannot extend cost on $c_{ij}$.
  - can break the simple maxRPC supports for the values of the greater variable $i^*$ in $c_{ij}$ and thus new supports need to be searched for for such values (line 27).
  - can break the witnesses for maxRPC supports in $c_{ik}$ (line 29, 31) or in $c_{jk}$ (line 30, 32).

An algorithm enforcing FDmaxRPC can be obtained from the EDmaxRPC algorithm by removing the loop processing EPIC at line 4. Similarly, the algorithms enforcing a single consistency maxRPC, DmaxRPC and EmaxRPC remove the global while loop at line 2 and keep only a while-loop at line 14, 7 or 4 that processes maxRPC, DmaxRPC or EmaxRPC respectively.

### Complexity

The algorithm for enforcing each consistency, maxRPC, DmaxRPC, FDmaxRPC and EmaxRPC can be inferred from Algorithm 4.7 for enforcing EDmaxRPC. Enforcing maxRPC

**Algorithme 4.7 :** Algorithm enforcing EDmaxRPC

```
Procedure enforceEDmaxRPC()
    while $S \neq \emptyset$ or $P \neq \emptyset$ $Q \neq \emptyset$ or $T \neq \emptyset$ do
        $R \leftarrow S \cup \bigcup_{i \in S, c_{ij}} \{j\}$;
        while $R \neq \emptyset$ do
            $j \leftarrow R$.popmin();
            findEmaxRPCSupport($j$);
        while $P \neq \emptyset$ do
            $j \leftarrow P$.popmax();
            foreach $c_{ij}, i < j$ do
                foreach $a \in D(i)$ do
                    findFullmaxRPCSupport($i, a, j, semiLevel$);
                foreach $\Delta_{ikj}, i < k$ do
                    findWitness_project ($i, k, j$);
        while $Q \neq \emptyset$ do
            $j \leftarrow Q$.popmin();
            foreach $c_{ij}$ do
                if $i > j$ then
                    foreach $a \in D(i)$ do
                        findmaxRPCSupport($i, a, j$);
                foreach $\Delta_{ikj}$ s.t. $i > j$ or $i > k$ do
                    findWitness_remove ($i, k, j$);
        while $T \neq \emptyset$ do
            $c_{ij} \leftarrow T$.pop();
            $i^* \leftarrow \max\{i, j\}$;
            $j^* \leftarrow \min\{i, j\}$;
            foreach $a \in D(i^*)$ do
                findmaxRPCSupport($i^*, a, j^*$);
            foreach $\Delta_{ijk}$ do
                findWitness_remove ($i, k, j$);
                findWitness_remove ($k, i, j$);
                findWitness_remove ($j, k, i$);
                findWitness_remove ($k, j, i$);
```

contains only the loop at line 14 because there is no unary cost extension caused by maxRPC. Enforcing DmaxRPC contains only the loop at line 7. Enforcing FDmaxRPC contains three loops at lines 7, 14 and 22. We will discuss the complexity of algorithms enforcing maxRPC, DmaxRPC and EmaxRPC and prove the termination of FDmaxRPC, EDmaxRPC.

Functions $\curlywedge_{ij}()$ and $\ddot{\curlywedge}_{ij}()$ have a time complexity in $O(c'd)$ where $c'$ is the maximum number of triangles sharing a binary cost function.

Thus, a constraint check for a value, done by findmaxRPCSupport$(i,a,j)$ and by findFullmaxRPCSupport$(i,a,j)$ takes $O(c'd^3)$ time. A constraint check takes $O(c'd^4)$ time.

Procedure findWitness_project $(i,k,j)$ and findWitness_remove $(i,k,j)$ have the best time complexity in $O(d^2)$ when every current support of variable $i$ in $c_{ik}$ have a witness on $k$. They have the worst time complexity in $O(c'd^4)$ when the current support of every value of $i$ in $c_{ik}$ has no witness on $j$. In this case, a new maxRPC support in $c_{ik}$ needs to be searched for for every value of $i$. This is exactly the work of a constraint check for $c_{ik}$. Thus, in order to compute the time complexity of our maxRPC algorithms, we are only interested in the number of constraint checks caused by the search for supports or for witnesses (find(Full)maxRPCSupport or findWitness_remove(project)).

**maxRPC** A binary function $c_{ij}$ is checked for maxRPC at most $2d$ times because each variable is pushed into $Q$ at most $d$ times. The total time to check for maxRPC for all the constraints is $O(ed \times c'd^4) = O(cd^5)$.

**DmaxRPC** Similarly to the DPIC algorithm, the variables in $P$ are arranged with respect to the DAC order. When a variable $j$ is popped out of $P$, only variables smaller than it are considered for checking DmaxRPC (algorithm 4.7, line 9) and can increase their unary costs. Thus, $j$ and greater variables are never pushed into $P$ again. Each variable $j$ is pushed into $P$ at most once and thus each binary function $c_{ij}$ is checked for DmaxRPC at most once (if $i < j$). A constraint check for DmaxRPC takes the same time as a constraint check for maxRPC in $O(c'd^4)$. Thus, the total time of DmaxRPC algorithm is $O(e \times c'd^4) = O(cd^4)$.

**FDmaxRPC** In FDmaxRPC, costs are only moved inside Procedure `find(Full)maxRPCSupport` where every cost movement is always accompanied by unary cost projections. Each unary cost projection and associated cost movements take $O(c'd^4)$ in time. From the equation 4.8, we have that $O(2^n dm)$ is an upper bound on the number of unary cost projections in any algorithm enforcing a full directional consistency. Thus, the total time complexity of our FDmaxRPC algorithm cannot exceed $O(2^n dm \times c'd^4) = O(2^n c'd^5 m)$ or $O(2^n cd^5 m)$. This means that the algorithm terminates. This bound is likely not tight.

**EDmaxRPC** Similarly to EDPIC, in an algorithm of EmaxRPC, the property EmaxRPC cannot be violated more than $m$ times because each time EmaxRPC is violated, $c_\varnothing$ increases. Thus, the FDmaxRPC block (consisting of 3 inner-while loops at line 7, 14, 22 in algorithm 4.7) is activated by EmaxRPC less than $m$ times. The number of unary cost projections created by FDmaxRPC part is also smaller than $O(m \times 2^n dm) = O(2^n dm^2)$. The total time complexity of the FDmaxRPC part is smaller than $O(2^n dm^2 \times c'd^4) = O(2^n c'd^5 m^2)$.

The EmaxRPC loop at line 4 is activated whenever $S \neq \varnothing$ which means that there has been a unary cost projection from 0 created in the FDmaxRPC part. So, the EmaxRPC loop cannot be activated more than the number of unary cost projections and this is smaller than $O(2^n dm^2)$. In a single EmaxRPC loop, each variable is enforced for EmaxRPC at most once because it is never pushed into $R$ twice (see the explication for EDPIC). Because the time complexity of findEmaxRPCSupport is $O(c'd^4)$, the time complexity of the EmaxRPC part cannot exceed $O(c'd^4 \times 2^n dm^2) = O(2^n c'd^5 m^2)$ or $O(2^n cd^5 m^2)$. In summary, the time complexity of the whole EDmaxRPC algorithm cannot exceed $O(2^n cd^5 m^2)$ and it terminates. This bound is likely not tight.

## 4.5 Experimentation

### 4.5.1 Benchmarks and experiments

In order to evaluate the practical interest of establishing high order consistencies (HOCs), we compared it to the default local consistency enforced in `toulbar2`: EDAC. Indeed, EDAC is still the state-of-the-art for WCSP solving (VAC being mostly useful for some very hard or specific problems). We use a set of benchmarks which have been used in the experimentation of EDAC in [Allouche et al., 2014b] for comparing the performance of the `toulbar2` solver with other solvers. This set of benchmarks is large and heterogeneous enough to facilitate the identification of favorable and unfavorable cases for the application of high order consistencies. It consists of benchmarks that are collected from different resources and that have been transformed to the WCSP format. This set of benchmarks includes cost function networks (CFN), Max-CSP, Weighted partial Max-SAT (WPMS) and Markov Random Field (MRF) problems that are grouped in classes as follows:

- CFN: contains cost function networks extracted from the Cost Function Library[1], including Combinatorial Auctions [Larrosa et al., 2008], Radio Link Frequency Assignment problems [Cabon et al., 1999], Mendelian error correction problems on complex pedigree [Sánchez et al., 2008], Computational Protein Design problems [Allouche et al., 2012], SPOT5 satellite scheduling problems [Bensana et al., 1999] and uncapacited warehouse location problems [Kratica et al., 2001].
- MRF: consists of Markov Random Field problems that are collected from the Probabilistic Inference Challenge 2011[2] and Genetic Linkage Analysis problems[Favier et al., 2011].
- WPMS: contains Max-SAT problems that are collected from the Max-SAT Evaluation[3],
- Max-CSPs: contains unsatisfiable binary CSP problems with constraints defined in extension, including BlackHole, Langford, QCP Quasi-group Completion Problem, Graph Coloring, random Composed, random 3-SAT EHI, and random Geometric.
- CVPR: contains MRF instances from the Computer Vision and Pattern Recognition (CVPR) OpenGM2 benchmark[4]

[1] https://mulcyber.toulouse.inra.fr/scm/viewvc.php/trunk/?root=costfunctionlib
[2] http://www.cs.huji.ac.il/project/PASCAL/realBoard.php
[3] http://maxsat.ia.udl.cat:81/13/benchmarks/
[4] http://hci.iwr.uni-heidelberg.de/opengm2

The set of benchmarks is described in Table 4.1. Each line corresponds to a category of benchmarks where the number of instances (#inst) and the mean values of problem size are also given. We report also the mean triangle density per category in the last column. The triangle density of an instance is defined as the ratio of its number of triangles ($c$) with the number of triangles in a complete graph ($n \times (n-1) \times (n-2)/6$ where $n$ is the number of its variables), i.e. $(6 \times c)/(n \times (n-1) \times (n-2))$.

In this thesis, we have considered three experiments of enforcing high order consistencies. First, we use HOCs during pre-processing and EDAC during search in order to know the impact of HOCs on the search via the initial quality of the lower bound: how and in which cases they can accelerate the search. Second, we define a restricted version of HOCs and enforce it during pre-processing together with EDAC during search. This experiment aims at decreasing the pre-processing time of HOCs while still providing good lower bounds for accelerating the search. Finally, we use the restricted HOCs during both pre-processing and search in order to determine whether they can accelerate the search when being maintained during search or not. We do not maintain non restricted HOCs in search because they are too costly when maintained during search.

## 4.5.2 Pre-processing by PICs and maxRPCs

In order to evaluate the performance of HOCs for pre-processing, in our first experiments, we enforce HOCs during pre-processing and maintain EDAC during search. Notice that in pre-processing, HOCs are enforced after WCSPs are enforced for EDAC. We will compare the efficiency (in terms of the number of solved problems, the running-time, the number of backtracks, the lower bound obtained after pre-processing) of HOCs in this experiment and EDAC enforced during both pre-processing and search. These experiments are implemented in the `toulbar2` solver, using the same parameters as in [Allouche et al., 2014b] `-dee=1, -l=1`.

### Number of solved problems

Table 4.2 reports the number of instances solved using either EDAC (third columns) or EDAC combined with HOCs (eight following columns) for pre-processing together with maintaining EDAC during search. Each line corresponds to a category of benchmarks. The green line gives a global evaluation on the overall set of benchmarks. It shows that in general HOCs solve less instances than EDAC but on some special categories of benchmarks such as CVPR, HOCs can solve more instances than EDAC where the stronger HOCs are, the better they are.

While EDAC cannot solve any ChineseChars instance (the same for all the other solvers reported in [Allouche et al., 2014b]) every HOC can solve a certain number of instances (at least 8 by PIC and at most 16 by EDmaxRPC). We also have tried VAC to solve the ChineseChars benchmarks but no instance can be solved by VAC in the limited time. Similarly to ChineseChars, HOCs solve up to 5% instances more than EDAC on CVPR/GeomSurf-7. Matching is the unique case of CVPR where HOCs can solve less instances than EDAC. For other categories of CVPR, HOCs solve the same number of instances as EDAC. These problems have a small triangle density and this is not sufficient to make HOCs sufficiently

| Categories | #inst | $\overline{n}$ | $\overline{d}$ | $\overline{e}$ | $\overline{r}$ | $\overline{c}$ | $\overline{c'}$ | $\overline{\text{dens}}$ |
|---|---|---|---|---|---|---|---|---|
| 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 |
| MatchingStereo | 2 | 138407 | 18 | 414477 | 2 | 8 | 8 | 2,70E-14 |
| ObjectSeg | 5 | 68160 | 6 | 203947 | 2 | 31 | 31 | 5,91E-13 |
| PhotoMontage | 2 | 469856 | 6 | 1408134 | 2 | 521 | 521 | 4,03E-14 |
| SceneDecomp | 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 |
| PackupWeighted | 99 | 9492 | 2 | 23731 | 61 | 9236 | 9236 | 6,87E-07 |
| PlanningWithPre | 29 | 14991 | 2 | 111259 | 64 | 8026 | 8026 | 1,76E-06 |
| TimeTabling | 25 | 128243 | 2 | 785222 | 21 | 40052 | 40052 | 1,58E-09 |
| Upgradeability | 100 | 18169 | 2 | 105097 | 77 | 1884 | 1884 | 1,88E-09 |
| UAI | 211 | | | | | | | |
| Grid | 21 | 3143 | 2 | 9379 | 2 | 2 | 2 | 3,74E-08 |
| ImageAlignment | 10 | 191 | 70 | 1819 | 2 | 6218 | 37 | 0,0058 |
| Linkage | 22 | 917 | 5 | 1560 | 4 | 13 | 13 | 2,23E-07 |
| ObjectDetection | 37 | 60 | 17 | 1830 | 2 | 34220 | 0 | 1 |
| ProteinFolding | 21 | 486 | 267 | 2291 | 2 | 4698 | 273 | 0,52 |
| Segmentation | 100 | 229 | 12 | 851 | 2 | 315 | 185 | 0,00016 |
| WCSP | 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 |
| ProteinDesign | 10 | 13 | 123 | 97 | 2 | 311 | 0 | 0,966 |
| SPOT5 | 20 | 385 | 4 | 6603 | 3 | 35976 | 2900 | 0,0055 |

*Table 4.1: The set of benchmarks for high order consistencies (#inst: number of instances, $\overline{n}$: mean number of variables, mean $\overline{d}$: mean domain size, mean $\overline{e}$: mean number of cost functions, mean $\overline{r}$: mean arity of cost functions, mean $\overline{c}$: mean number of triangles, mean $\overline{c'}$: mean number of triangles used by restricted high order consistencies at the root of the search tree, and mean dens: mean triangle density.)*

| problem | inst | EDAC | PIC | DPIC | FDPIC | EDPIC | maxRPC | DmaxRPC | FdmaxRPC | EdmaxRPC |
|---|---|---|---|---|---|---|---|---|---|---|
| summary | 2820 | **2053** | 1972 | 1980 | 1979 | 1979 | 1967 | 1982 | 1980 | 1981 |
| CVPR | 1453 | 1301 | 1308 | 1315 | 1315 | 1318 | 1309 | 1321 | 1318 | **1327** |
| ChineseChars | 100 | 0 | 8 | 8 | 10 | 10 | 10 | 9 | 10 | **16** |
| GeomSurf-7 | 300 | 281 | 280 | 287 | 285 | 288 | 281 | 292 | 292 | **295** |
| ColorSeg | 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| GeomSurf-3 | 300 | 300 | 300 | 300 | 300 | 300 | 300 | 300 | 300 | 300 |
| InPainting | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Matching | 4 | **4** | **4** | **4** | **4** | **4** | 2 | **4** | 0 | 0 |
| MatchingStereo | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ObjectSeg | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PhotoMontage | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SceneDecomp | 715 | 715 | 715 | 715 | 715 | 715 | 715 | 715 | 715 | 715 |
| MaxCSP | 503 | **214** | 212 | 211 | 210 | 209 | 211 | 211 | 210 | 210 |
| Coloring | 22 | 17 | **18** | **18** | 17 | 17 | **18** | **18** | **18** | **18** |
| QCP | 60 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 |
| BlackHole | 37 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Composed | 80 | 80 | 80 | 80 | 80 | 80 | 80 | 80 | 80 | 80 |
| EHI | 200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Geometric | 100 | **91** | 88 | 87 | 87 | 86 | 87 | 87 | 86 | 86 |
| Langford | 4 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| MaxSAT | 427 | **195** | 173 | 171 | 172 | 169 | 170 | 169 | 171 | 168 |
| Haplotyping | 100 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1 |
| MaxClique | 62 | **33** | 15 | 14 | 15 | 14 | 13 | 12 | 14 | 13 |
| MIPLib | 12 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| PackupWeighted | 99 | **52** | 48 | 47 | 47 | 47 | 47 | 46 | 47 | 47 |
| PlanningWithPre | 29 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| TimeTabling | 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Upgradeability | 100 | **100** | **100** | **100** | **100** | 98 | **100** | **100** | 99 | 98 |
| UAI | 211 | **142** | 130 | 132 | 130 | 131 | 130 | 130 | 130 | 130 |
| ImageAlignment | 10 | 10 | 7 | 9 | 7 | 7 | 6 | 7 | 5 | 5 |
| Linkage | 22 | 13 | 13 | 13 | 13 | 14 | 14 | 13 | **15** | **15** |
| ObjectDetection | 37 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ProteinFolding | 21 | **19** | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Segmentation | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Grid | 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| WCSP | 226 | **201** | 149 | 151 | 152 | 152 | 147 | 151 | 151 | 146 |
| Auction | 170 | **166** | 126 | 128 | 130 | 129 | 125 | 129 | 129 | 125 |
| CELAR | 16 | **12** | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 3 |
| ProteinDesign | 10 | **9** | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 |
| SPOT5 | 20 | 4 | 4 | 4 | 4 | 5 | 4 | 4 | 4 | 4 |
| Pedigree | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |

*Table 4.2: The number of instances per category solved in less 1 hour for CVPR group and 1200 seconds for other cases. The best results are in bold except for the cases where every consistency give the same result.*

different from EDAC. Among them, the categories GeomSurf-3, MatchingStereo, ObjectSeg and SceneDecomp have too few triangles of variables (8, 8, 31, 48 respectively). As we will discuss later, on these problems, HOCs provide exactly the same lower bounds as EDAC.

HOCs do not make any improvement for every category of benchmarks in the group MaxCSP of binary and Boolean problems. The tiny problems are solved by all nine consistencies while the difficult ones either cannot be solved by anyone or can be solved only by EDAC. Similarly, HOCs do not improve EDAC on the groups of categories MaxSAT, UAI and WCSP. A decrease in the number of solved instances is observed on categories MaxClique, PackupWeighted, ImageAlignment, ProteinFolding, Auction, CELAR, ProteinDesign. These problems have a large triangle density and the problems ImageAlignment, ProteinFolding, Auction, CELAR and ProteinDesign have a large domain size (70, 267, 44, 123 respectively). These two factors are the reason of the drastic increase in the time complexity of HOCs over EDAC and thus HOCs solve less instances than EDAC.

In summary, the favorable cases for the application of HOCs during pre-processing are ChineseChars and GeomSurf-7 problems. Problems having large domain sizes and large triangle densities are unfavorable cases for HOCs.

#### Lower bound

Table 4.3 reports the mean lower bounds per category of benchmarks obtained by using EDAC and HOCs for pre-processing. We use the ratio $LB/LB_0 - 1$, where $LB$ and $LB_0$ are respectively the lower bounds obtained by HOCs and by EDAC, to present the increase in the lower bounds of HOCs over EDAC. Note that this representation is relative because lower bounds vary significantly from problem to problem, i.e., from 0 to $10^{14}$. These mean increases and the accumulated mean increases are presented by Sub-figures 4.17(b) and 4.17(a). As expected, both Table 4.3 and Figure 4.17 show important contribution of HOCs to the lower bound. They also show that the increase in lower bounds is in general consistent with the strength of the HOCs.

More precisely, the largest increases in lower bounds, up to 37%, can be observed on MaxClique, 25,24% on Auction, 32,23% on SPOT5, and 5% on Upgradeability, ProteinFolding, CELAR. These problems are characterized by a very large triangle density (except for Upgradeability) that makes more opportunities for HOCs to increase lower bounds. A significant improvement is also observed on CVPR benchmarks such as ChineseChars, GeomSurf-7 (despite the fact that GeomSurf-7 has a medium triangle density, HOCs still can improve lower bounds thanks to the special characteristics of these problems).

However, HOCs do not improve EDAC lower bounds for many categories of benchmarks such as all categories of MaxCSP, GeomSurf-3, MatchingStereo, ObjectSeg of CVPR, MIPLib, PlanningWithPre, TimeTabling of MaxSAT, DBN, Linkage of UAI, Pedigree and Warehouse of WCSP. This can be explained by the excessively small number of triangles for categories GeomSurf-3, MatchingStereo and ObjectSeg (having only 8, 8 and 31 triangles respectively) whereas, in the case of GeomSurf-3, EDAC already provides lower bounds equal to optimums for 293/300 instances.

However, the quality of lower bounds are not always consistent with the strength of consistencies because locally consistent closures are not unique in WCSPs. Each enforcing

| Problems | EDAC | PIC | DPIC | FDPIC | EDPIC | maxRPC | DmaxRPC | FdmaxRPC | EdmaxRPC |
|---|---|---|---|---|---|---|---|---|---|
| MaxCSP | | | | | | | | | |
| ChineseChars($\times 10^9$) | 97190 | 97215 | 97212 | 97216 | 97217 | 97218 | 97213 | 97217 | 97222 |
| ColorSeg($\times 10^7$) | 170577 | 170587 | 170663 | 170663 | 170676 | 170590 | 170743 | 170744 | 170764 |
| GeomSurf-3 | 13213474307 | - | - | - | - | - | - | - | - |
| GeomSurf-7($\times 10^6$) | 44232 | 44277 | 44440 | 44478 | 44777 | 44315 | 44612 | 44646 | 45066 |
| InPainting-8($\times 10^6$) | 43159 | 43163 | 43237 | 43237 | 43237 | 43163 | 43251 | 43252 | 43256 |
| MatchingStereo | 1558718 | - | - | - | - | - | - | - | - |
| ObjectSeg | 3092221904136 | - | - | - | - | - | - | - | - |
| PhotoMontage | 100776 | 100879 | 100847 | 100880 | 100880 | 100879 | 100847 | 100880 | 100880 |
| SceneDecomp($\times 10^6$) | 555626 | 555630 | 555630 | 555631 | 555637 | 555630 | 555633 | 555634 | 555641 |
| MaxCSP | | | | | | | | | |
| BlackHole-7 | 0 | - | - | - | - | - | - | - | - |
| Coloring | 1 | - | - | - | - | - | - | - | - |
| Composed | 0 | - | - | - | - | - | - | - | - |
| EHI-90 | 0 | - | - | - | - | - | - | - | - |
| Geometric | 0 | - | - | - | - | - | - | - | - |
| Langford | 0 | - | - | - | - | - | - | - | - |
| QCP-25/ | 0 | - | - | - | - | - | - | - | - |
| MaxSAT | | | | | | | | | |
| Haplotyping | 8778 | 8796 | 8784 | 8796 | 8796 | 8791 | 8791 | 8791 | 8791 |
| MaxClique | 160 | 214 | 173 | 216 | 219 | 217 | 179 | 219 | 219 |
| MIPLib | 6857 | - | - | - | - | - | - | - | - |
| PackupWeighted | 2301028 | 2307218 | 2303029 | 2307106 | 2306715 | 2307621 | 2304071 | 2307241 | 2308181 |
| PlanningWithPre | 0 | - | - | - | - | - | - | - | - |
| TimeTabling | 2 | - | - | - | - | - | - | - | - |
| Upgradeability($\times 10^4$) | 40351 | 40831 | 40893 | 40986 | 42366 | 41204 | 40924 | 40971 | 42195 |
| UAI | | | | | | | | | |
| Grid | 154278 | 154756 | + | + | + | + | + | + | + |
| ImageAlignment | 136247 | 136250 | + | + | + | + | + | + | + |
| Linkage | 3874 | 3892 | + | + | + | + | + | + | + |
| ProteinFolding | 77845 | 80980 | 80129 | 80911 | 81335 | 80997 | 80647 | 80925 | 81463 |
| Segmentation | 18785 | 18864 | 18846 | 18867 | 18919 | 18872 | 18873 | 18872 | 18961 |
| WCSP | | | | | | | | | |
| Auction | 47845 | 54723 | 53674 | 55494 | 55798 | 55585 | 59679 | 59919 | 56527 |
| CELAR | 2601 | 2662 | 2690 | 2678 | 2690 | 2720 | 2693 | 2697 | 2711 |
| Pedigree | 23 | - | - | - | - | - | - | - | - |
| ProteinDesign | 190316 | 190523 | 190493 | 190498 | 190518 | 190506 | 190465 | 190468 | 190526 |
| SPOT5 | 55054 | 63981 | 66026 | 66682 | 70244 | 65333 | 72261 | 72799 | 71328 |

*Table 4.3: The mean lower bounds per category of benchmarks obtained after the pre-processing phase. These means are computed only over instances that are pre-processed by all consistencies. The symbol "-" means that HOCs provide the same lower bounds as EDAC while "+" means that HOCs provide the same lower bounds as PIC and greater than EDAC.*

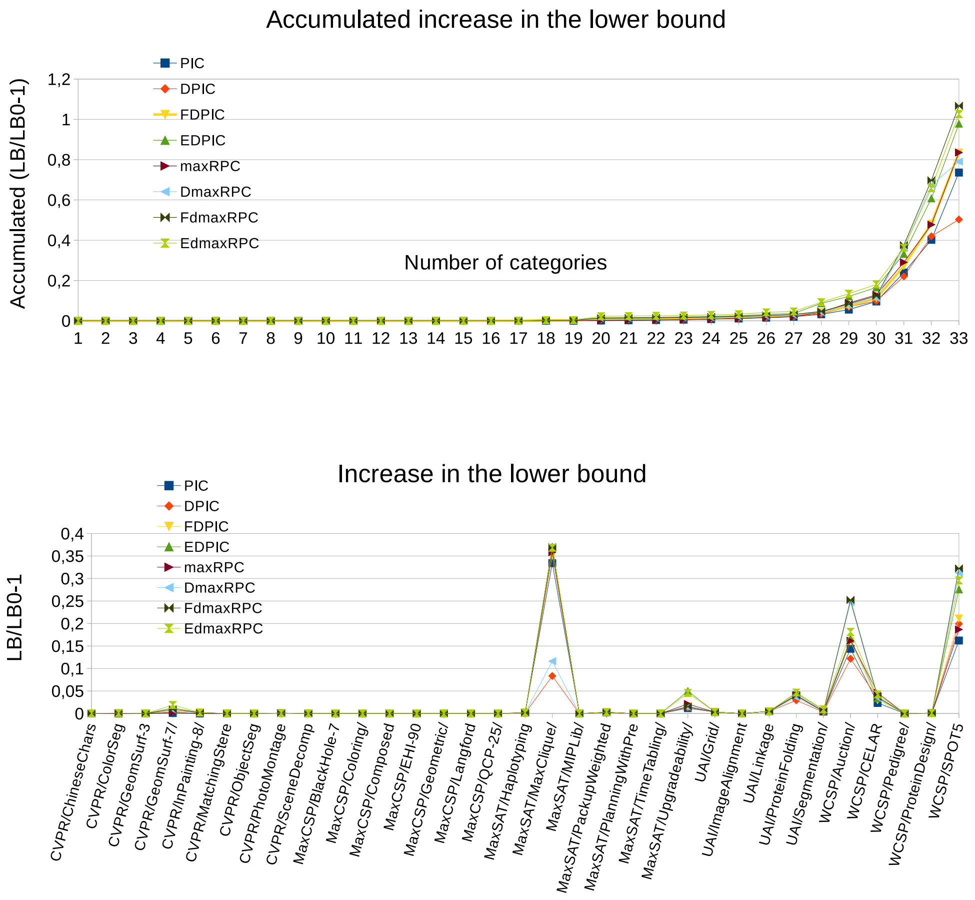


*Figure 4.17: The increase in lower bounds obtained by high order consistencies over EDAC when being used for pre-processing. This increase is evaluated by the ratio $\frac{LB}{LB_0} - 1$ where $LB$ and $LB_0$ are respectively the lower bounds obtained by a high order consistency and EDAC.*

order, defined by the propagation queue, can lead to a different closure with a different lower bound. This is visible in Table 4.3 where EDmaxRPC can provide lower bounds smaller than FDmaxRPC (for example for WCSP/Auction: $LB_{EDmaxRPC} = 56527 < LB_{FDmaxRPC} = 59919$), or FDmaxRPC can provide smaller lower bounds than DmaxRPC and maxRPC (for WCSP/CELAR $LB_{FDmaxRPC} = 2697 < LB_{maxRPC} = 2720$), and this is also observed for PIC ($LB_{FDPIC} = 2678 < LB_{DPIC} = 2690$).

In summary, HOCs can improve EDAC lower bounds on a variety of problems, especially on graphs having sufficiently large triangle densities.

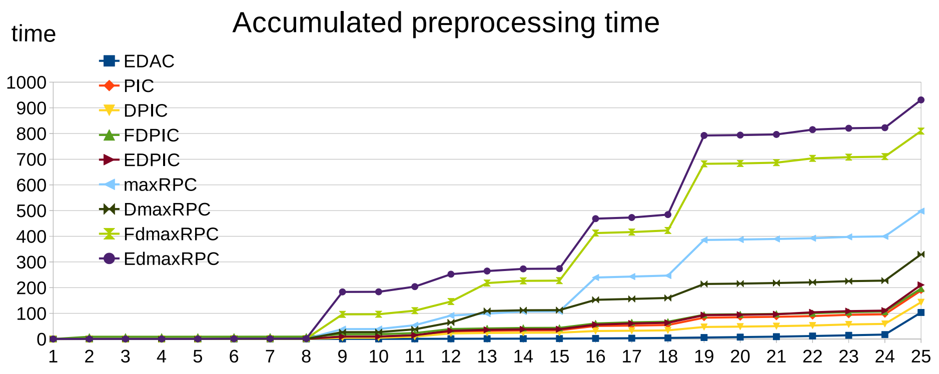
#### Pre-processing, searching and solving time

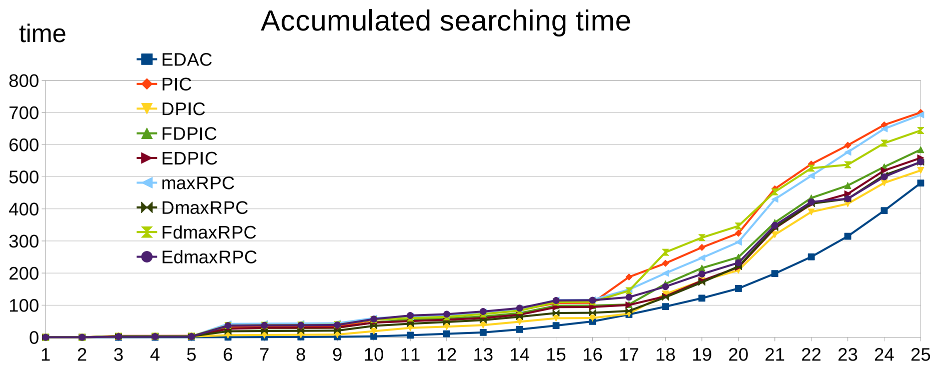
We consider 3 kinds of time measures : pre-processing, searching and solving times. The former is the time used for pre-processing problems, the second for searching for solutions, and the last for solving problems computed as the sum of the pre-processing and the searching time. Figure 4.19 shows the mean time per category of benchmarks and Figure 4.18 shows the accumulated mean time for these three kinds of time measures. They show that high order consistencies are slower than EDAC as expected and the slow-downs are consistent with the strength of high order consistencies where the magnitude of slow-downs in search time is much smaller than that in pre-processing time because the search is accelerated thanks to the increase in lower bounds.

**Pre-processing time** Compared to EDAC, HOCs slow down by a factor of 1690, 371, 241, 195, 101, 50, 34 respectively on categories **ProteinDesign**, **ImageAlignment**, MaxClique, **CELAR**, Auction, SPOT5, **BlackHole** (Sub-figure 4.19(a)). These categories have a large mean triangle density where the categories in bold also have a large mean domain size. Moreover, a large slow-down by a factor up to 198 is created on CVPR/ChineseChars (absent from Sub-figure 4.19(a)) where HOCs use 945 seconds compared to 4.85 seconds used by EDAC for pre-processing.

**Solving time** Despite the fact that HOCs are always slower than EDAC in pre-processing, Sub-figure 4.19(b),(c) shows that they can outperform EDAC in search and in total solving time on many problems. A mean speed-ups by a factor of 2 and 6 respectively is observed for EDPIC and EDmaxRPC on category GeomSurf-7 in Sub-figure 4.19(c). Even more impressive speed-ups on GeomSurf-7 and ChineseChars problems can be observed in Table 4.4: many instances cannot be solved in 1 hour by EDAC but can be solved by HOCs in less than 100s. The other speed-ups of 1.9, 3.3, 1.3 and 7.8 observed respectively on Haplotyping, MaxClique, MIPLib and ProteinFolding are not the result of a general superiority because (1) only one among 100 Haplotyping instances and 3 among 12 MIPLib instances are solved by all 9 consistencies and (2) on MaxClique, ProteinFolding, HOCs are always slower than EDAC except for one difficult instance.

On other categories of benchmarks, especially unfavorable cases, HOCs are still slower than EDAC in search (as in pre-processing) and thus slower in terms of the total solving time. Precisely, we can observe very large slow-downs by a factor up to 4 on MaxClique, 332 on ImageAlignment, 3 on Segmentation, 3 on Auction, 109 on CELAR, 1471 on ProteinDesign and 5 on SPOT5. These slow-downs are explained by the dramaticaly increased enforcing





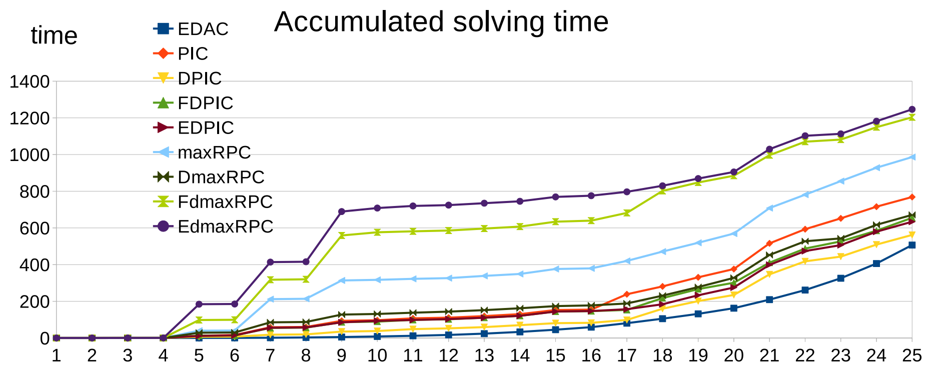


*Figure 4.18: The accumulated mean time for pre-processing, searching and solving categories of benchmarks. Only instances solved by all consistencies are taken into account to compute the mean time of each category. Each sub-figure corresponds to a type of time measure whereas each line corresponds to a consistency. Axis X represents the categories while Axis Y represents the time in seconds. For each sub-figure, benchmarks categories are arranged by increasing order in the mean time of EDAC.*

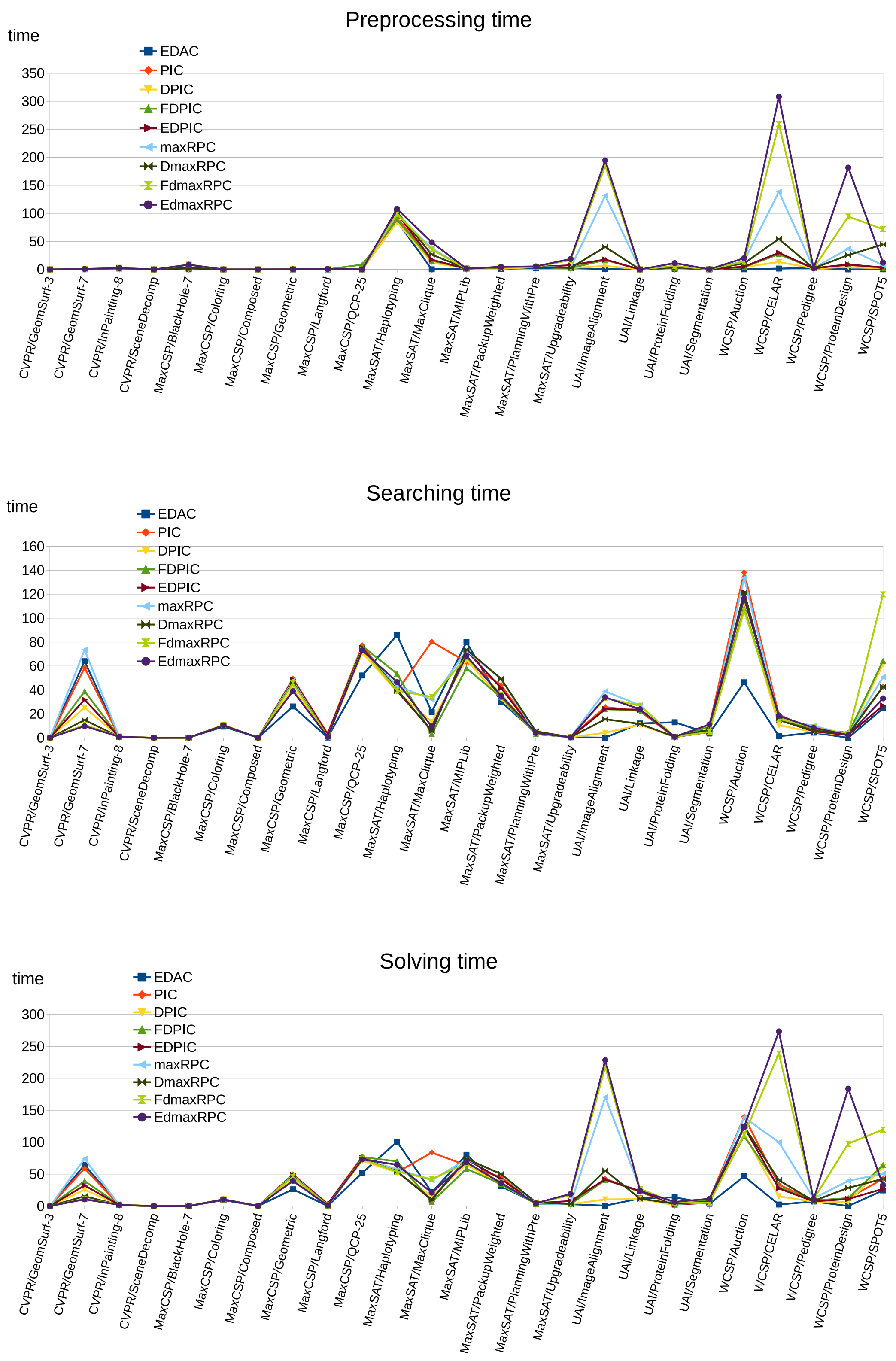


*Figure 4.19: The mean time for pre-processing, searching and solving categories of benchmarks. The mean pre-processing time per category is computed over only instances pre-processed by all consistencies. Conversely, the mean search and solving time is computed over only instances solved by all consistencies. Axis X represents benchmarks categories and Axis Y represents the time in seconds.*

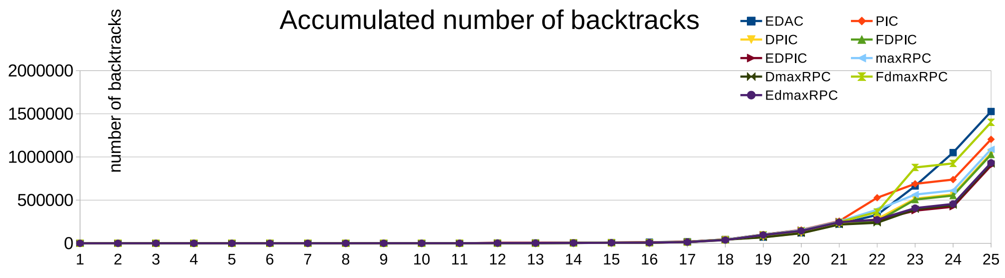
| Problem | EDAC | PIC | DPIC | FDPIC | EDPIC | maxRPC | DmaxRPC | FdmaxRPC | EdmaxRPC |
|---|---|---|---|---|---|---|---|---|---|
| ChineseChars | | | | | | | | | |
| TST_0012_88_103 | - | 195 | 575 | 34 | 41 | **21** | 119 | 27 | 44 |
| TST_0020_96_94 | - | - | 2647 | 249 | 260 | 363 | 1709 | **158** | **158** |
| TST_0024_88_126 | - | - | - | - | - | - | - | - | **662** |
| TST_0027_88_109 | - | - | - | - | - | - | - | - | **865** |
| TST_0041_88_96 | - | - | - | 3149 | 1569 | 269 | - | 760 | **114** |
| TST_0047_112_121 | - | 215 | 83 | 23 | 49 | **18** | 26 | 28 | 64 |
| TST_0052_96_107 | - | 1230 | 3564 | 1842 | 154 | 77 | 690 | 183 | **46** |
| TST_0059_104_73 | - | 130 | 93 | 37 | 32 | **11** | 28 | 20 | 33 |
| TST_0067_96_121 | - | 201 | 590 | 117 | 143 | 47 | 151 | **41** | 76 |
| TST_0070_88_96 | - | 460 | 2194 | 392 | 158 | 56 | 210 | 99 | 73 |
| TST_0084_120_115 | - | - | - | - | - | - | - | - | **416** |
| TST_0087_88_124 | - | - | - | - | - | - | - | - | **1910** |
| TST_0089_72_92 | - | - | - | - | - | - | - | - | 1148 |
| TST_0099_72_105 | - | 502 | 2012 | 112 | 101 | 30 | 347 | **65** | 75 |
| TST_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** |

*Table 4.4: The solving time (in seconds) for a subset of benchmarks. This subset contains only ChineseChars and GeomSurf-7 instances which respectively can and cannot be solved by one of consistencies in 1 hour. "–" means that the problem cannot be solved. Best results are in bold.*

time of EDAC during search because a very large number of new ternary cost functions that have been created by HOCs during pre-processing. Moreover, they combine the slow-downs in pre-processing time of HOCs over EDAC on categories MaxClique, ImageAlignment, CELAR.

**Summary** HOCs are always slower than EDAC in pre-processing but they can outperform EDAC in terms of the final solving time on favorable problems such as ChineseChars, GeomSurf-7. On problems having large densities, they get significantly slower than EDAC.

### Number of backtracks



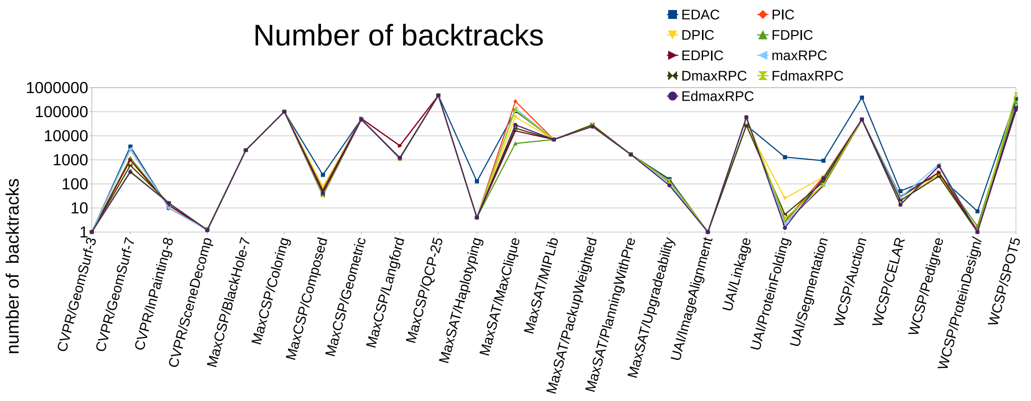


*Figure 4.20: The (accumulated) mean number of backtracks used during search. These means are computed only over problems that are solved by all consistencies.*

The mean numbers of backtracks during search per category of benchmarks is presented in Sub-figure 4.20(b). The accumulated value for these mean numbers of backtracks is presented in Sub-figure 4.20(a). They show that HOCs in general use less backtracks than EDAC as expected, where a reduction of 40% in the total number of backtracks is observed for both EDmaxRPC and EDPIC.

Sub-figure(b) shows that this reduction happens for both favorable and unfavorable cases. Precisely, HOCs reduce the number of backtracks by a factor up to 2604 respectively on ProteinFolding, 11.5 on GeomSurf-7, 8 on Auction, 5.3 on Segmentation, 3.7 on MaxClique, 5.8 on Composed, 2.5 on SPOT5, 3.9 on CELAR, 1.4 for Upgradeability.

ProteinFolding problems become nearly backtrack-free in search where EDmaxRPC and EDAC respectively use 0.5 and 1302 backtracks on average. Moreover, HOCs make problems ProteinDesign backtrack-free in search but of course very slow while EDAC does not.

Indeed, as analyzed in the previous section, HOCs can significantly improve lower bounds on these problems and this is the reason for such reduction in the number of backtracks.

#### Conclusion

High order consistencies have a good behavior on specific categories of benchmarks such as ChineseChars and GeomSurf-7. On such categories, they can solve more instances in less mean time and use less nodes and backtracks than EDAC. However, they are much slower than EDAC on problems having large triangle densities and as a result solve less instances. In both two cases, they can significantly improve lower bounds and therefore use less nodes and backtracks than EDAC. Especially, some problems can be solved backtrack-free by high order consistencies.

### 4.5.3 Pre-processing by a restricted version of PICs and maxRPCs

#### Restricted high order consistencies

In this section, we present a restricted version of high order consistencies by limiting the number of triangles to be checked for the consistencies. The goal of this limitation is to reduce the enforcing time complexity of high order consistencies and therefore to improve the number of solved problems, especially for problems having large triangle densities. For favorable categories of benchmarks, we would process all triangles in order to profit as much as possible from the advantage of high-order consistencies. Conversely, for unfavorable cases, we would enforce high order consistencies on only a subset of triangles because processing all triangles is too costly for such problems.

In order to define a threshold on the number of triangles, we have considered the relationship between the behavior of high order consistencies in the first experimentation and the characteristics of problems. We observed that unfavorable categories of benchmarks such as Matching, Geometric, MaxClique, ImageAlignment, ObjectDetection, ProteinFolding, Auction, CELAR, ProteinDesign,... have a triangle density larger than $10^{-4}$, i.e., have more than $c^* = n(n-1)(n-2)/6.10^4$ triangles. Conversely, the favorable categories of benchmarks have less than $c^*$ triangles. From these observations, we propose a restricted version for high order consistencies which processes at most $c^*$ triangles. A value of $c^* < 10$ is considered too small to make a difference between restricted high order consistencies and EDAC. In this case, restricted consistencies will be replaced by EDAC and the number of triangles used by restricted consistencies $c'$ is set to 0. Conversely, there will be two possible cases. If $c \leq c^*$, all triangles will be used by restricted high order consistencies, i.e. $c' = c$. Otherwise, only $c^*$ triangles are used for restricted consistencies, i.e. $c' = c^*$. In the second case, triangles are evaluated and classified according to the mean binary cost of triangles. The $c^*$ stronger triangles are selected for enforcing HOCs. The mean cost of a binary cost function $c_{ij}$ is computed by $(\sum_{a \in D(i), b \in D(j)} c_{ij}(a,b))/(|D(i)| \times |D(j)|)$. The mean binary cost of a triangle $\Delta_{ijk}$ is the sum of the mean cost of three binary cost functions.

In summary, compared to HOCs, the restricted HOCs are unchanged for problems having more than 10 triangles and less than $c^*$ triangles; or become weaker otherwise. They also can be considered as adaptive parameterized consistencies [Balafrej et al., 2013] with a parameter $c'/c \in [0..1]$ where $c'$ is automatically defined for each problem based on the maximum number of triangles in the corresponding complete graph. The restricted HOCs are therefore intermediate between HOCs and EDAC. We denote by HOCs$^r$s, PICs$^r$ and maxRPCs$^r$ the restricted version of HOCs, PICs and maxRPCs respectively.

The mean number of triangles (per category of benchmarks) used by restricted HOCs at the root of the search tree is represented in the last column ($c'$) of Table 4.1. $c' = 0$ means that HOCs$^r$s are equivalent to EDAC. $c' = c$ means that HOCs$^r$s are equivalent to HOCs. In this table, HOCs$^r$s are identical to HOCs for most categories of CVPR and MaxCSP while they are reduced to EDAC for categories Matching, Composed, Geometric, Langford, ObjectDetection.

*Table 4.5: The number of instances solved in less than 1200 seconds (1 hour for CVPR group) by enforcing EDAC, original and restricted HOCs during pre-processing. Each block of one or two lines corresponds to a category of benchmarks whose name and size are given in the two first columns. A white block corresponds to a single category of benchmarks where a white block with the name in italic and having only one line means that HOCs$^r$s gives the same result as HOCs. A yellow block corresponds to the group of categories presented below it in the table. The green block represents the summary on the complete set of benchmarks. Each box in the third column contains the number of instances per category of benchmarks solved by EDAC while each box of the last eight columns contains either one number (the same number of instances solved by the original and restricted HOCs) or two numbers: the top number represents the number of instances solved by original HOCs and the bottom number those solved by restricted consistencies.*

| Problems | inst | EDAC | PIC | DPIC | FDPIC | EDPIC | maxRPC | DmaxRPC | FdmaxRPC | EdmaxRPC |
|---|---|---|---|---|---|---|---|---|---|---|
| summary | 2820 | 2053 | 1972 | 1980 | 1979 | 1979 | 1967 | 1982 | 1980 | 1981 |
| | | | 2051 | 2055 | 2060 | 2058 | 2059 | 2069 | 2072 | **2074** |
| CVPR | | 1301 | 1308 | 1315 | 1315 | 1318 | 1309 | 1321 | 1318 | 1327 |
| | | | 1309 | 1311 | 1317 | 1320 | 1314 | 1322 | 1325 | **1329** |
| ChineseChars | 100 | 0 | 8 | 8 | 10 | 10 | 10 | 9 | 10 | 16 |
| | | | 9 | 7 | 9 | 10 | 14 | 10 | 13 | 15 |
| GeomSurf-7 | 300 | 281 | 280 | 287 | 285 | 288 | 281 | 292 | 292 | **295** |
| | | | 280 | 284 | 288 | 290 | 280 | 292 | 292 | 294 |
| Matching | 4 | **4** | **4** | **4** | **4** | **4** | 2 | **4** | 0 | 0 |
| | | | **4** | **4** | **4** | **4** | **4** | **4** | **4** | **4** |
| *ColorSeg* | 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MaxCSP | 503 | **214** | 212 | 211 | 210 | 209 | 211 | 211 | 210 | 210 |
| | | | **214** | **214** | **214** | **214** | **214** | **214** | **214** | **214** |
| Coloring | 22 | 17 | **18** | **18** | 17 | 17 | **18** | **18** | **18** | **18** |
| | | | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 |
| *QCP* | 60 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 |
| *BlackHole* | 37 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| *EHI* | 200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Continued on next page*

Table 4.5 – *Continued from previous page*

| Problems | inst | EDAC | PIC | DPIC | FDPIC | EDPIC | maxRPC | DmaxRPC | FdmaxRPC | EdmaxRPC |
|---|---|---|---|---|---|---|---|---|---|---|
| *Langford* | 4 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| MaxSAT | 427 | **195** | 173 | 171 | 172 | 169 | 170 | 169 | 171 | 168 |
| | | | 182 | 185 | 184 | 178 | 185 | 186 | 186 | 185 |
| Haplotyping | 100 | 1 | 1 | 1 | 1 | 1 | 1 | **2** | **2** | 1 |
| | | | 1 | 1 | 1 | 1 | **2** | **2** | **2** | **2** |
| MaxClique | 62 | **33** | 15 | 14 | 15 | 14 | 13 | 12 | 14 | 13 |
| | | | 28 | 29 | 30 | 29 | 29 | 29 | 30 | 30 |
| PackupWeighted | 99 | **52** | 48 | 47 | 47 | 47 | 47 | 46 | 47 | 47 |
| | | | 48 | 46 | 48 | 47 | 48 | 47 | 47 | 47 |
| Upgradeability | 100 | **100** | **100** | **100** | **100** | 98 | **100** | **100** | 99 | 98 |
| | | | 96 | **100** | 96 | 92 | 97 | 99 | 98 | 97 |
| MIPLib | 12 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| | | | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| UAI | 211 | **142** | 130 | 132 | 130 | 131 | 130 | 130 | 130 | 130 |
| | | | 143 | 143 | 144 | 144 | 143 | 143 | 145 | 145 |
| ImageAlignment | 10 | **10** | 7 | 9 | 7 | 7 | 6 | 7 | 5 | 5 |
| | | | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** |
| Linkage | 22 | 13 | 13 | 13 | 13 | 14 | 14 | 13 | **15** | **15** |
| | | | 14 | 13 | 14 | 14 | 14 | 14 | **15** | **15** |
| ProteinFolding | 21 | 19 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| | | | **20** | **20** | **20** | **20** | **20** | **20** | **20** | **20** |
| Segmentation | 100 | **100** | **100** | **100** | **100** | **100** | **100** | **100** | **100** | **100** |
| | | | 99 | **100** | **100** | **100** | 99 | 99 | **100** | **100** |
| WCSP | 226 | **201** | 149 | 151 | 152 | 152 | 147 | 151 | 151 | 146 |
| | | | 202 | 202 | 200 | 201 | 202 | 202 | 202 | 200 |
| Auction | 170 | 166 | 126 | 128 | 130 | 129 | 125 | 129 | 129 | 125 |
| | | | **167** | **167** | 166 | **167** | **167** | **167** | **167** | 165 |
| CELAR | 16 | **12** | 4 | 4 | 3 | 3 | 3 | 3 | 3 | 3 |
| | | | **12** | **12** | 11 | 11 | **12** | **12** | **12** | **12** |
| SPOT5 | 20 | 4 | 4 | 4 | 4 | 5 | 4 | 4 | 4 | 4 |
| | | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |

## Experimental results

In order to improve the efficiency of HOCs for pre-processing, we consider a second experiment which enforces HOCs$^r$s instead of HOCs during pre-processing and EDAC during search as done in the first experimentation. The mean number of instances (per category) solved by HOCs$^r$s is also introduced in Table 4.5, represented by the bottom number in boxes. Please notice that when $c' = c$, HOCs$^r$s in general give the same result as HOCs, but they also can give a different result because HOCs$^r$ rearrange the triangles list, according to the mean binary cost of triangles, and thus revise triangles in a different order. This table does not report the result for categories of benchmarks on which either HOCs$^r$s are replaced by EDAC ($c' = 0$) or HOCs$^r$s are identical to HOCs ($c' = c$) and give the same results as HOCs.

The green block in the table shows that in total HOCs$^r$s can solve up to 95 instances more than HOCs. Except for PIC, HOCs$^r$ also can solve more instances than EDAC (up to 23 instances). As we will analyze immediately later, this result is the combination of (1) an improvement of HOCs$^r$ over HOCs on unfavorable categories of benchmarks and (2) the same results on favorable cases.

As expected, HOCs$^r$s outperform HOCs on unfavorable categories of benchmarks such as MaxClique, ImageAlignment, ProteinFolding, Auction, CELAR which have large triangle densities and their number of triangles have been significantly reduced. HOCs$^r$s can solve a number of instances comparable to EDAC, i.e., larger than HOCs. This is also presented in Table 4.7 which reports in the 4th and 5th columns respectively the mean solving time (per category of benchmarks) taken by the HOCs and HOCs$^r$s. HOCs$^r$ outperform HOCs by a factor going from 10 to 30 on MaxClique, 16 and 145 on ImageAlignment, 7.8 and 13.9 on Auction, 3.4 and 49 on Celar. ProteinFolding is a case special. Despite the fact that HOCs$^r$s are slower than HOCs on the 10 tiniest instances of the category (Table 4.7), they outperform HOCs on 10 other hard instances and thus can solve them in less than 1200 seconds while HOCs cannot (Table 4.5). For unfavorable categories of benchmarks where the number of triangles reduces to 0 ($c' = 0$) such as Matching, Geometric, ProteinDesign, HOCs$^r$s of course give the same result as EDAC, i.e. better than HOCs.

When HOCs$^r$ are identical to HOCs in the sense of the number of triangles, HOCs$^r$ give a result similar to HOCs on almost all cases, except for ChineseChars and Upgradeability. Table 4.5 shows that HOCs$^r$ can solve slightly more or slightly less instances than HOCs on ChineseChars and less instances on Upgradeability. Table 4.7 shows speed-ups by factors up to 2 of HOCs$^r$s over HOCs on ChineseChars and slow-downs by factors up to 10 on Upgradeability. This shows the impact of the revising order of triangles as analyzed above.

In the case of favorable problems, HOCs$^r$s still keep the advantage not only on ChineseChars but also on GeomSurf-7. The number of triangles of GeomSurf-7 problems insignificantly decreases and this is not sufficient to make HOCs$^r$s sufficiently different from HOCs.

In summary, pre-processing by HOCs$^r$ allows to solve more instances in less time than HOCs on unfavorable categories of benchmarks while HOCs$^r$s still behave well on favorable categories of benchmarks similarly to HOCs and better than EDAC. On unfavorable cases, HOCs$^r$ can solve a number of instances comparable to EDAC.

### 4.5.4 Maintaining PICs$^r$ and maxRPC$^r$ during search

In our third experiments, we will enforce HOCs$^r$s not only during pre-processing but also during search. This experimentation is motivated by the good behavior of HOCs$^r$s on the overall set of benchmarks when used for pre-processing. It is expected that HOCs$^r$s when enforced during search will be more beneficial on favorable problems while providing results comparable to EDAC, i.e., better than HOCs$^r$s when enforced during pre-processing, on unfavorable cases. We decided to not maintain HOCs during search because this is too costly, especially on unfavorable problems on which the solving time of HOCs will increase by a factor related to the number of nodes.

Thus, three approaches for using HOCs implemented in this thesis are:

- first approach: enforcing HOCs during preprocessing and EDAC during search.
- second approach: enforcing HOCs$^r$s during preprocessing and EDAC during search.

- third approach: enforcing HOCs$^r$s during both preprocessing and search.

Now, we will evaluate the practical interest of maintaining HOCs$^r$s during search by comparing them to HOCs$^r$s when used for pre-processing.

### Number of solved problems

In addition to Table 4.5, Table 4.6 reports at the third lines of boxes the numbers of instances per category of benchmarks solved by maintaining HOCs$^r$s during search. This table does not report the result for categories of benchmarks on which either HOCs$^r$s are replaced by EDAC ($c' = 0$) or HOCs$^r$s are identical to HOCs ($c' = c$) and the 3 approaches enforcing HOCs give the same results.

The block of 3 green lines shows that in general, HOCs$^r$s when maintained during search has a better behavior than HOCs but worse than EDAC as well as HOCs$^r$s when enforced only during pre-processing. Precisely, when maintaining HOCs$^r$s, the total number of solved instances of the set of benchmarks decreases by a factor 1.95% compared to EDAC, decreases by a factor going from 1.17% to 3.86% compared to pre-processing by HOCs$^r$s but increases by a factor going from 0.61% to 2.58% compared to pre-processing by HOCs.

Compared to HOCs$^r$s used for pre-processing, the reduction in the number of solved instances when maintaining HOCs$^r$s during search occurs on both favorable and unfavorable categories of benchmarks, especially on the unfavorable cases. A slight reduction up to 4% is observed on GeomSurf-7 while larger reductions are observed on unfavorable and neutral categories of benchmarks, up to 26.67% on MaxClique, 13% on Auction, 17% on PackupWeighted, 8% on Upgradeability, 40% on Linkage.

Conversely, compared to HOCs used for pre-processing, the number of solved instances increases by a factor up to 93% on MaxClique and 22% on Auction and decrease by a factor up to 17% on PackupWeighted, 11% on Upgradeability, 40% on Linkage. Notice that the number of triangles used by HOCs$^r$s reduces on the two first categories MaxClique and Auction and remain unchanged on the three last cases. This means that we should not maintain HOCs (HOCs$^r$s when exploiting all triangles of problems) during search. For unfavorable benchmarks having large triangle densities, maintaining HOCs$^r$ behaves better than pre-processing HOCs and worse than pre-processing HOCs$^r$.

Compared to EDAC, HOCs$^r$s maintained in search still behave better on ChineseChars, and can behave better or worse on GeomSur7. For other cases, the number of solved instances is either unchanged or decreases (MaxClique, PackupWeighted, Linkage, Auction).

### Solving time

Figure 4.21 presents the accumulated solving time of HOCs when enforced in the three ways. Each sub-figure corresponds to a consistency and each line corresponds to an enforcing method. We observe that:

- HOCs$^r$ used for pre-processing is the fastest and HOCs$^r$ used for both pre-processing and search is the slowest.
- The difference in solving time of 3 methods are consistent with the strength of consistencies.

Precisely, compared to HOCs (first approach):

- the total average time of HOCs$^r$ when used for pre-processing (second approach) decreases by a factor going from 1.4 to 2.1 (PIC$^r$s) or from from 1.7 to 3.1 (maxRPC$^r$s).
- the total average time of HOCs$^r$ when used for both pre-processing and search (third approach) decreases (except for PIC, DPIC, FDPIC, FDmaxRPC which create very large speed-ups on hard instances ChineseChars).

Consider the last column in Table 4.7 to compare the solving time of HOCs$^r$s when enforced during pre-processing and when enforced during both pre-processing and search. It is observed that HOCs$^r$s get slower when maintained during search by a factor going from 1.18 to 3.97 on ChineseChars (except for PIC$^r$ and DPIC$^r$ that are respectively 1.23 and 3.17 times faster), from 1.02 to 9.37 on GeomSurf-7 (except for DPIC$^r$ that is faster 1.18 times faster)

On the categories of benchmarks having $c' = 0$, there is no significant difference in the solving time between the second and the third approaches enforcing HOCs$^r$. For the rest of the set of benchmarks, HOCs$^r$, especially maxRPCs$^r$, get slower when maintained during search by a factor up to 20 on Linkage, MaxClique, PlanningWithPre, PackupWeighted; up to 10 on CELAR, Segmentation, Haplotyping, Coloring; or up to 5 on SPOT5, Auction, ProteinFolding, Langford, Composed,. . .

In summary, HOCs$^r$s, especially maxRPCs$^r$, when maintained during search have a reduced performance in terms of the number of solved instances as well as the mean solving time compared to themselves when enforced only during pre-processing. This behavior occurs on both favorable and unfavorable problems.

*Table 4.6: The number of instances per category solved in less than 1200 seconds (1 hour for CVPR group). Each block of three lines corresponds to a category of benchmarks where the two first lines present the same results as in Table 4.5 and the third line gives the number of instances solved by HOCs when enforced during pre-processing and search. The blocks presented in only one line with name in italic means that HOCs give the same results in three cases: (1) HOCs for pre-processing, (2) restricted HOCs for pre-processing and (3) restricted HOCs for pre-processing and search.*

| Problems | inst | EDAC | PIC | DPIC | FDPIC | EDPIC | maxRPC | DmaxRPC | FdmaxRPC | EdmaxRPC |
|---|---|---|---|---|---|---|---|---|---|---|
| summary | 2820 | 2053 | 1972 | 1980 | 1979 | 1979 | 1967 | 1982 | 1980 | 1981 |
| | | | 2051 | 2055 | 2060 | 2058 | 2059 | 2069 | 2072 | **2074** |
| | | | 2013 | 2031 | 2030 | 2018 | 1993 | 2010 | 1992 | 1998 |
| CVPR | | 1301 | 1308 | 1315 | 1315 | 1318 | 1309 | 1321 | 1318 | **1327** |
| | | | 1309 | 1311 | 1317 | 1320 | 1314 | 1322 | 1325 | 1329 |
| | | | 1307 | 1312 | 1319 | 1317 | 1304 | 1315 | 1313 | 1313 |
| ChineseChars | 100 | 0 | 8 | 8 | 10 | 10 | 10 | 9 | 10 | **16** |
| | | | 9 | 7 | 9 | 10 | 14 | 10 | 13 | 15 |
| | | | 9 | 9 | 10 | 10 | 11 | 10 | 12 | 11 |
| GeomSurf-7 | 300 | 281 | 280 | 287 | 285 | 288 | 281 | 292 | 292 | **295** |
| | | | 280 | 284 | 288 | 290 | 280 | 292 | 292 | 294 |
| | | | 278 | 283 | 289 | 287 | 273 | 285 | 281 | 282 |

*Continued on next page*

Table 4.6 – *Continued from previous page*

| Problems | inst | EDAC | PIC | DPIC | FDPIC | EDPIC | maxRPC | DmaxRPC | FdmaxRPC | EdmaxRPC |
|---|---|---|---|---|---|---|---|---|---|---|
| Matching | 4 | 4 | 4 | 4 | 4 | 4 | 2 | 4 | 0 | 0 |
| | | | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| | | | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| MaxCSP | 503 | 214 | 212 | 211 | 210 | 209 | 211 | 211 | 210 | 210 |
| | | | 214 | 214 | 214 | 214 | 214 | 214 | 214 | 214 |
| | | | 213 | 213 | 212 | 208 | 209 | 208 | 209 | 208 |
| Coloring | 22 | 17 | **18** | **18** | 17 | 17 | **18** | **18** | **18** | **18** |
| | | | 17 | 17 | 17 | 17 | 17 | 17 | 17 | 17 |
| | | | 17 | 17 | 16 | 16 | 17 | 16 | 16 | 16 |
| QCP | 60 | **14** | **14** | **14** | **14** | **14** | **14** | **14** | **14** | **14** |
| | | | **14** | **14** | **14** | **14** | **14** | **14** | **14** | **14** |
| | | | **14** | **14** | **14** | **14** | 13 | 13 | 13 | **14** |
| *BlackHole* | 37 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| *EHI* | 200 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MaxSAT | 427 | 195 | 173 | 171 | 172 | 169 | 170 | 169 | 171 | 168 |
| | | | 182 | 185 | 184 | 178 | 185 | **186** | **186** | 185 |
| | | | 166 | 175 | 169 | 167 | 162 | 167 | 164 | 162 |
| Haplotyping | 100 | 1 | 1 | 1 | 1 | 1 | 1 | **2** | **2** | 1 |
| | | | 1 | 1 | 1 | 1 | **2** | **2** | **2** | **2** |
| | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| MaxClique | 62 | **33** | 15 | 14 | 15 | 14 | 13 | 12 | 14 | 13 |
| | | | 28 | 29 | 30 | 29 | 29 | 29 | 30 | 30 |
| | | | 24 | 27 | 24 | 26 | 23 | 23 | 22 | 22 |
| PackupWeighted | 99 | **52** | 48 | 47 | 47 | 47 | 47 | 46 | 47 | 47 |
| | | | 48 | 46 | 48 | 47 | 48 | 47 | 47 | 47 |
| | | | 41 | 39 | 42 | 40 | 41 | 39 | 40 | 40 |
| Upgradeability | 100 | **100** | **100** | **100** | **100** | 98 | **100** | **100** | 99 | 98 |
| | | | 96 | **100** | 96 | 92 | 97 | 99 | 98 | 97 |
| | | | 92 | **100** | 94 | 92 | 89 | 96 | 93 | 91 |
| MIPLib | 12 | 3 | **3** | **3** | **3** | **3** | **3** | **3** | **3** | **3** |
| | | | **3** | **3** | **3** | **3** | **3** | **3** | **3** | **3** |
| | | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| UAI | 211 | 142 | 130 | 132 | 130 | 131 | 130 | 130 | 130 | 130 |
| | | | 143 | 143 | 144 | 144 | 143 | 143 | 145 | **145** |
| | | | 139 | 140 | 140 | 138 | 137 | 140 | 137 | 137 |
| ImageAlignment | 10 | **10** | 7 | 9 | 7 | 7 | 6 | 7 | 5 | 5 |
| | | | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** |
| | | | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** |
| Linkage | 22 | 13 | 13 | 13 | 13 | 14 | 14 | 13 | **15** | **15** |
| | | | 14 | 13 | 14 | 14 | 14 | 14 | **15** | **15** |
| | | | 11 | 10 | 11 | 10 | 9 | 10 | 10 | 9 |
| ProteinFolding | 21 | 19 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| | | | **20** | **20** | **20** | **20** | **20** | **20** | **20** | **20** |
| | | | **20** | **20** | **20** | **20** | **20** | **20** | **20** | **20** |
| Segmentation | 100 | **100** | **100** | **100** | **100** | **100** | **100** | **100** | **100** | **100** |
| | | | 99 | **100** | **100** | **100** | 99 | 99 | **100** | **100** |
| | | | 98 | **100** | 99 | 98 | 98 | **100** | 98 | 98 |
| WCSP | 226 | 201 | 149 | 151 | 152 | 152 | 147 | 151 | 151 | 146 |

*Continued on next page*

Table 4.6 – *Continued from previous page*

| Problems | inst | EDAC | PIC | DPIC | FDPIC | EDPIC | maxRPC | DmaxRPC | FdmaxRPC | EdmaxRPC |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | **202**<br>188 | **202**<br>191 | 200<br>190 | 201<br>188 | **202**<br>181 | **202**<br>180 | **202**<br>180 | 200<br>178 |
| Auction | 170 | 166 | 126<br>**167**<br>154 | 128<br>**167**<br>156 | 130<br>166<br>156 | 129<br>**167**<br>154 | 125<br>**167**<br>147 | 129<br>**167**<br>146 | 129<br>**167**<br>146 | 125<br>165<br>144 |
| CELAR | 16 | **12** | 4<br>**12**<br>11 | 4<br>**12**<br>12 | 3<br>11<br>11 | 3<br>11<br>11 | 3<br>**12**<br>11 | 3<br>**12**<br>11 | 3<br>**12**<br>11 | 3<br>**12**<br>11 |
| SPOT5 | 20 | 4<br>4<br>4 | 4<br>4<br>4 | 4<br>4<br>4 | 4<br>4<br>4 | **5**<br>4<br>4 | 4<br>4<br>4 | 4<br>4<br>4 | 4<br>4<br>4 | 4<br>4<br>4 |

*Table 4.7: The mean solving time per category of benchmarks taken by HOCs for three cases. "Option1": HOCs for pre-processing and EDAC for search. "Option2": restricted HOCs for pre-processing and EDAC for search. "Option3": restricted HOCs for both pre-processing and search. Each block of 8 lines corresponds to a category of benchmarks where each line corresponds to a HOC whose name is represented in the second column. The three following columns report the mean solving time (in seconds) per category of benchmarks taken by each HOC in each case. For a HOC, only instances solved in all three cases are taken into account to compute the mean time of HOC for each case. The second last column presents the speed-ups of restricted HOCs over HOCs when they are used for pre-processing. The last column presents the speed-ups of restricted HOCs enforced in both pre-processing and searching over enforced in only pre-processing.*

| problems | consistency | option1 | option2 | option3 | $\frac{option2}{option1}$ | $\frac{option3}{option2}$ |
|---|---|---|---|---|---|---|
| CVPR/ ChineseChars | PIC | 516,56 | 596,88 | 486,55 | 1,16 | 0,82 |
| | DPIC | 924,57 | 964,91 | 304,52 | 1,04 | 0,32 |
| | FDPIC | 699,55 | 331,11 | 391 | 0,47 | 1,18 |
| | EDPIC | 303,86 | 246,23 | 976,83 | 0,81 | 3,97 |
| | maxRPC | 111,74 | 55,34 | 144,05 | 0,5 | 2,6 |
| | DmaxRPC | 539,5 | 274,37 | 455,96 | 0,51 | 1,66 |
| | FdmaxRPC | 178,28 | 90,34 | 149,42 | 0,51 | 1,65 |
| | EdmaxRPC | 129,39 | 84,62 | 240,92 | 0,65 | 2,85 |
| CVPR/ GeomSurf-7 | PIC | 51,64 | 71,53 | 80,06 | 1,39 | 1,12 |
| | DPIC | 27,33 | 45,47 | 38,4 | 1,66 | 0,84 |
| | FDPIC | 48,42 | 40,13 | 50,51 | 0,83 | 1,26 |
| | EDPIC | 51,86 | 57,58 | 58,67 | 1,11 | 1,02 |
| | maxRPC | 51,54 | 30,46 | 92,57 | 0,59 | 3,04 |
| | DmaxRPC | 23,86 | 17,9 | 93,68 | 0,75 | 5,23 |
| | FdmaxRPC | 15,75 | 20,58 | 106,31 | 1,31 | 5,17 |
| | EdmaxRPC | 24,21 | 11,6 | 108,65 | 0,48 | 9,37 |
| CVPR/ GeomSurf-3 | PIC | 0,03 | 0,03 | 0,03 | 1,02 | 1 |
| | DPIC | 0,03 | 0,03 | 0,03 | 0,97 | 1,07 |
| | FDPIC | 0,03 | 0,03 | 0,03 | 1,05 | 0,99 |
| | EDPIC | 0,03 | 0,03 | 0,03 | 0,97 | 1,1 |
| | maxRPC | 0,03 | 0,03 | 0,04 | 1,02 | 1,05 |

*Continued on next page*

Table 4.7 – *Continued from previous page*

| problems | consistency | option1 | option2 | option3 | $\frac{option2}{option1}$ | $\frac{option3}{option2}$ |
|---|---|---|---|---|---|---|
| | DmaxRPC | 0,03 | 0,03 | 0,04 | 1,01 | 1,1 |
| | FdmaxRPC | 0,03 | 0,03 | 0,04 | 0,98 | 1,15 |
| | EdmaxRPC | 0,03 | 0,03 | 0,04 | 1,01 | 1,08 |
| CVPR/ InPainting | PIC | 1,48 | 1,65 | 4,43 | 1,11 | 2,68 |
| | DPIC | 1,63 | 1,64 | 7,74 | 1,01 | 4,72 |
| | FDPIC | 1,86 | 2,32 | 8,36 | 1,25 | 3,6 |
| | EDPIC | 1,72 | 1,66 | 4,35 | 0,97 | 2,62 |
| | maxRPC | 1,96 | 1,78 | 3,17 | 0,91 | 1,78 |
| | DmaxRPC | 1,71 | 1,78 | 10,29 | 1,04 | 5,78 |
| | FdmaxRPC | 2,48 | 1,87 | 8,05 | 0,75 | 4,3 |
| | EdmaxRPC | 1,96 | 1,82 | 15,88 | 0,93 | 8,73 |
| CVPR/ SceneDecomp | PIC | 0,02 | 0,03 | 0,03 | 1,13 | 1,01 |
| | DPIC | 0,02 | 0,03 | 0,03 | 1,07 | 1,05 |
| | FDPIC | 0,03 | 0,03 | 0,03 | 1,09 | 1,04 |
| | EDPIC | 0,02 | 0,03 | 0,03 | 1,08 | 1,09 |
| | maxRPC | 0,03 | 0,03 | 0,03 | 1,08 | 1,21 |
| | DmaxRPC | 0,02 | 0,03 | 0,03 | 1,11 | 1,15 |
| | FdmaxRPC | 0,03 | 0,03 | 0,03 | 1,09 | 1,16 |
| | EdmaxRPC | 0,03 | 0,03 | 0,03 | 1,1 | 1,18 |
| MaxCSP/ Coloring | PIC | 10,43 | 9,3 | 31,72 | 0,89 | 3,41 |
| | DPIC | 10,31 | 9,4 | 34,26 | 0,91 | 3,65 |
| | FDPIC | 10,45 | 9,69 | 41,35 | 0,93 | 4,27 |
| | EDPIC | 10,23 | 9,39 | 28,37 | 0,92 | 3,02 |
| | maxRPC | 10,22 | 9,3 | 73,18 | 0,91 | 7,87 |
| | DmaxRPC | 10,31 | 9,46 | 68,69 | 0,92 | 7,26 |
| | FdmaxRPC | 10,57 | 9,51 | 62,61 | 0,9 | 6,59 |
| | EdmaxRPC | 10,41 | 9,47 | 68,49 | 0,91 | 7,23 |
| MaxCSP/ QCP | PIC | 77,34 | 52,57 | 53,14 | 0,68 | 1,01 |
| | DPIC | 71,06 | 52,4 | 52,63 | 0,74 | 1 |
| | FDPIC | 76,64 | 52,07 | 52,24 | 0,68 | 1 |
| | EDPIC | 75,2 | 51,46 | 46,66 | 0,68 | 0,91 |
| | maxRPC | 64,91 | 46,7 | 115,52 | 0,72 | 2,47 |
| | DmaxRPC | 66,65 | 45,79 | 119,01 | 0,69 | 2,6 |
| | FdmaxRPC | 65,77 | 47,07 | 121,63 | 0,72 | 2,58 |
| | EdmaxRPC | 73,09 | 53,78 | 107,54 | 0,74 | 2 |
| MaxCSP/ BlackHole-7 | PIC | 0,09 | 0,07 | 0,07 | 0,77 | 1,04 |
| | DPIC | 0,09 | 0,07 | 0,08 | 0,77 | 1,11 |
| | FDPIC | 0,09 | 0,07 | 0,08 | 0,75 | 1,07 |
| | EDPIC | 0,1 | 0,07 | 0,08 | 0,72 | 1,1 |
| | maxRPC | 0,1 | 0,07 | 0,12 | 0,72 | 1,69 |
| | DmaxRPC | 0,09 | 0,07 | 0,13 | 0,78 | 1,77 |
| | FdmaxRPC | 0,1 | 0,07 | 0,13 | 0,71 | 1,8 |
| | EdmaxRPC | 0,1 | 0,07 | 0,13 | 0,72 | 1,84 |
| MaxSAT/ Haplotyping | PIC | 52,8 | 49,23 | 209,42 | 0,93 | 4,25 |
| | DPIC | 52,74 | 48,75 | 178,07 | 0,92 | 3,65 |
| | FDPIC | 70,07 | 51,71 | 215,93 | 0,74 | 4,18 |
| | EDPIC | 54,19 | 53,66 | 411,59 | 0,99 | 7,67 |
| | maxRPC | 58,08 | 51,62 | 331,36 | 0,89 | 6,42 |
| | DmaxRPC | 54,04 | 48,86 | 319,66 | 0,9 | 6,54 |
| | FdmaxRPC | 54,15 | 52,52 | 231,41 | 0,97 | 4,41 |
| | EdmaxRPC | 64,76 | 52,84 | 416,67 | 0,82 | 7,89 |
| | PIC | 203,38 | 11,55 | 10,63 | 0,06 | 0,92 |

*Continued on next page*

Table 4.7 – *Continued from previous page*

| problems | consistency | option1 | option2 | option3 | $\frac{option2}{option1}$ | $\frac{option3}{option2}$ |
|---|---|---|---|---|---|---|
| MaxSAT/ MaxClique | DPIC | 168,79 | 13,71 | 15,15 | 0,08 | 1,11 |
| | FDPIC | 187,82 | 17,66 | 12,75 | 0,09 | 0,72 |
| | EDPIC | 128,72 | 12,74 | 27,89 | 0,1 | 2,19 |
| | maxRPC | 130,14 | 4,4 | 89,52 | 0,03 | 20,36 |
| | DmaxRPC | 103,34 | 6,75 | 10,56 | 0,07 | 1,56 |
| | FdmaxRPC | 179,55 | 10,49 | 24,03 | 0,06 | 2,29 |
| | EdmaxRPC | 107,71 | 4,34 | 77,57 | 0,04 | 17,86 |
| MaxSAT/ PackupWeighted | PIC | 21,51 | 15,45 | 18,74 | 0,72 | 1,21 |
| | DPIC | 1,59 | 1,18 | 1,6 | 0,74 | 1,36 |
| | FDPIC | 2,89 | 3,88 | 27,47 | 1,34 | 7,07 |
| | EDPIC | 1,19 | 1,43 | 8,75 | 1,21 | 6,1 |
| | maxRPC | 1,29 | 1,26 | 21,86 | 0,98 | 17,33 |
| | DmaxRPC | 1,6 | 1,26 | 2,88 | 0,79 | 2,29 |
| | FdmaxRPC | 1,26 | 1,31 | 19,03 | 1,04 | 14,52 |
| | EdmaxRPC | 1,27 | 1,26 | 11,73 | 0,99 | 9,3 |
| MaxSAT/ Upgradeability | PIC | 9,72 | 44,25 | 50,76 | 4,55 | 1,15 |
| | DPIC | 2,96 | 2,98 | 3,81 | 1,01 | 1,28 |
| | FDPIC | 10,77 | 52,16 | 41,9 | 4,84 | 0,8 |
| | EDPIC | 3,07 | 15,54 | 27,65 | 5,06 | 1,78 |
| | maxRPC | 3,29 | 33,4 | 64,47 | 10,14 | 1,93 |
| | DmaxRPC | 3,26 | 42,39 | 68,24 | 12,98 | 1,61 |
| | FdmaxRPC | 26,72 | 49,98 | 73,01 | 1,87 | 1,46 |
| | EdmaxRPC | 15,38 | 49,47 | 81,31 | 3,22 | 1,64 |
| MaxSAT/ MIPLib | PIC | 0,13 | 0,09 | 0,1 | 0,68 | 1,12 |
| | DPIC | 0,12 | 0,09 | 0,12 | 0,71 | 1,41 |
| | FDPIC | 0,11 | 0,1 | 0,11 | 0,9 | 1,11 |
| | EDPIC | 0,09 | 0,1 | 0,13 | 1,06 | 1,32 |
| | maxRPC | 0,09 | 0,09 | 0,23 | 1 | 2,56 |
| | DmaxRPC | 0,11 | 0,09 | 0,18 | 0,77 | 2,12 |
| | FdmaxRPC | 0,11 | 0,1 | 0,19 | 0,9 | 1,95 |
| | EdmaxRPC | 0,1 | 0,08 | 0,17 | 0,8 | 2,06 |
| MaxSAT/ PlanningWithPre | PIC | 3,56 | 3,12 | 27,66 | 0,88 | 8,86 |
| | DPIC | 4,16 | 3,7 | 24,71 | 0,89 | 6,69 |
| | FDPIC | 3,43 | 3,53 | 25,48 | 1,03 | 7,22 |
| | EDPIC | 4,58 | 3,82 | 34,63 | 0,83 | 9,06 |
| | maxRPC | 3,98 | 3,8 | 57,78 | 0,95 | 15,22 |
| | DmaxRPC | 5,84 | 3,4 | 53,64 | 0,58 | 15,77 |
| | FdmaxRPC | 4,29 | 4,08 | 64,18 | 0,95 | 15,74 |
| | EdmaxRPC | 4,52 | 3,88 | 72,4 | 0,86 | 18,66 |
| UAI/ ImageAlignment | PIC | 109,53 | 4,23 | 6,17 | 0,04 | 1,46 |
| | DPIC | 68,88 | 4,27 | 6,57 | 0,06 | 1,54 |
| | FDPIC | 106,51 | 4,52 | 6,61 | 0,04 | 1,46 |
| | EDPIC | 103,66 | 4,52 | 7,12 | 0,04 | 1,57 |
| | maxRPC | 299,96 | 3,27 | 6,97 | 0,01 | 2,13 |
| | DmaxRPC | 188,71 | 2,67 | 5,9 | 0,01 | 2,21 |
| | FdmaxRPC | 218,33 | 1,62 | 2,98 | 0,01 | 1,84 |
| | EdmaxRPC | 228,42 | 1,57 | 2,91 | 0,01 | 1,85 |
| UAI/ Linkage | PIC | 9,95 | 10,21 | 130,14 | 1,03 | 12,74 |
| | DPIC | 7,28 | 8,64 | 82,73 | 1,19 | 9,58 |
| | FDPIC | 10,58 | 9,74 | 155,9 | 0,92 | 16 |
| | EDPIC | 8,08 | 8,26 | 28,62 | 1,02 | 3,47 |
| | maxRPC | 7,18 | 9,36 | 97,89 | 1,3 | 10,46 |

*Continued on next page*

Table 4.7 – *Continued from previous page*

| problems | consistency | option1 | option2 | option3 | $\frac{option2}{option1}$ | $\frac{option3}{option2}$ |
|---|---|---|---|---|---|---|
| | DmaxRPC | 4,22 | 9,32 | 102,21 | 2,21 | 10,97 |
| | FdmaxRPC | 7,48 | 8,64 | 185,03 | 1,16 | 21,4 |
| | EdmaxRPC | 7,21 | 7,16 | 54,35 | 0,99 | 7,59 |
| UAI/ ProteinFolding | PIC | 3,61 | 6,86 | 23,32 | 1,9 | 3,4 |
| | DPIC | 4,65 | 9,2 | 21,89 | 1,98 | 2,38 |
| | FDPIC | 3,45 | 12,65 | 20,84 | 3,67 | 1,65 |
| | EDPIC | 3,05 | 9,95 | 48,3 | 3,26 | 4,85 |
| | maxRPC | 5,08 | 9,45 | 46,19 | 1,86 | 4,89 |
| | DmaxRPC | 4,53 | 10 | 39,35 | 2,21 | 3,93 |
| | FdmaxRPC | 5,63 | 4,88 | 10,65 | 0,87 | 2,18 |
| | EdmaxRPC | 7,78 | 17,6 | 62,61 | 2,26 | 3,56 |
| UAI/ Segmentation | PIC | 2,45 | 2,46 | 6,8 | 1 | 2,76 |
| | DPIC | 10,63 | 13,04 | 12,55 | 1,23 | 0,96 |
| | FDPIC | 6,22 | 3,5 | 14,26 | 0,56 | 4,07 |
| | EDPIC | 2,25 | 4,51 | 8,74 | 2 | 1,94 |
| | maxRPC | 2,4 | 2,4 | 16,63 | 1 | 6,93 |
| | DmaxRPC | 4,9 | 3,42 | 18,78 | 0,7 | 5,49 |
| | FdmaxRPC | 2,37 | 2,55 | 16,83 | 1,08 | 6,6 |
| | EdmaxRPC | 2,64 | 2,74 | 23,55 | 1,04 | 8,6 |
| WCSP/ Auction | PIC | 154,57 | 12,12 | 30,09 | 0,08 | 2,48 |
| | DPIC | 123,61 | 15,86 | 48,09 | 0,13 | 3,03 |
| | FDPIC | 129,39 | 15,93 | 46,97 | 0,12 | 2,95 |
| | EDPIC | 138,14 | 13,73 | 46,84 | 0,1 | 3,41 |
| | maxRPC | 120,18 | 12,53 | 61,13 | 0,1 | 4,88 |
| | DmaxRPC | 152,85 | 11,01 | 47,35 | 0,07 | 4,3 |
| | FdmaxRPC | 125,61 | 11,36 | 56,38 | 0,09 | 4,96 |
| | EdmaxRPC | 114,03 | 9,16 | 49,41 | 0,08 | 5,39 |
| WCSP/ CELAR | PIC | 33,51 | 8,16 | 20,93 | 0,24 | 2,57 |
| | DPIC | 281,73 | 5,74 | 5,81 | 0,02 | 1,01 |
| | FDPIC | 29,77 | 7,04 | 18,65 | 0,24 | 2,65 |
| | EDPIC | 27,73 | 8,09 | 20,69 | 0,29 | 2,56 |
| | maxRPC | 99,93 | 15,19 | 94,59 | 0,15 | 6,23 |
| | DmaxRPC | 40,53 | 8,75 | 37,08 | 0,22 | 4,24 |
| | FdmaxRPC | 238,89 | 22,36 | 159,32 | 0,09 | 7,13 |
| | EdmaxRPC | 273,48 | 19,75 | 195,31 | 0,07 | 9,89 |
| WCSP/ Pedigree | PIC | 10,93 | 8,91 | 12,61 | 0,82 | 1,42 |
| | DPIC | 8,51 | 9,28 | 14,20 | 1,09 | 1,53 |
| | FDPIC | 10,52 | 7,66 | 13,86 | 0,73 | 1,81 |
| | EDPIC | 9,34 | 9,66 | 18,63 | 1,03 | 1,93 |
| | maxRPC | 15,17 | 7,63 | 16,46 | 0,50 | 2,16 |
| | DmaxRPC | 10,13 | 9,48 | 17,27 | 0,94 | 1,82 |
| | FdmaxRPC | 13,51 | 8,34 | 24,99 | 0,62 | 3,00 |
| | EdmaxRPC | 13,07 | 7,69 | 17,10 | 0,59 | 2,22 |
| WCSP/ SPOT5 | PIC | 42,69 | 35,63 | 49,40 | 0,83 | 1,39 |
| | DPIC | 61,27 | 38,27 | 37,85 | 0,62 | 0,99 |
| | FDPIC | 64,39 | 36,18 | 39,91 | 0,56 | 1,10 |
| | EDPIC | 27,01 | 35,01 | 77,12 | 1,30 | 2,20 |
| | maxRPC | 50,86 | 35,83 | 49,76 | 0,70 | 1,39 |
| | DmaxRPC | 42,68 | 31,28 | 186,36 | 0,73 | 5,96 |
| | FdmaxRPC | 119,92 | 41,33 | 84,76 | 0,34 | 2,05 |
| | EdmaxRPC | 33,01 | 33,45 | 104,88 | 1,01 | 3,14 |

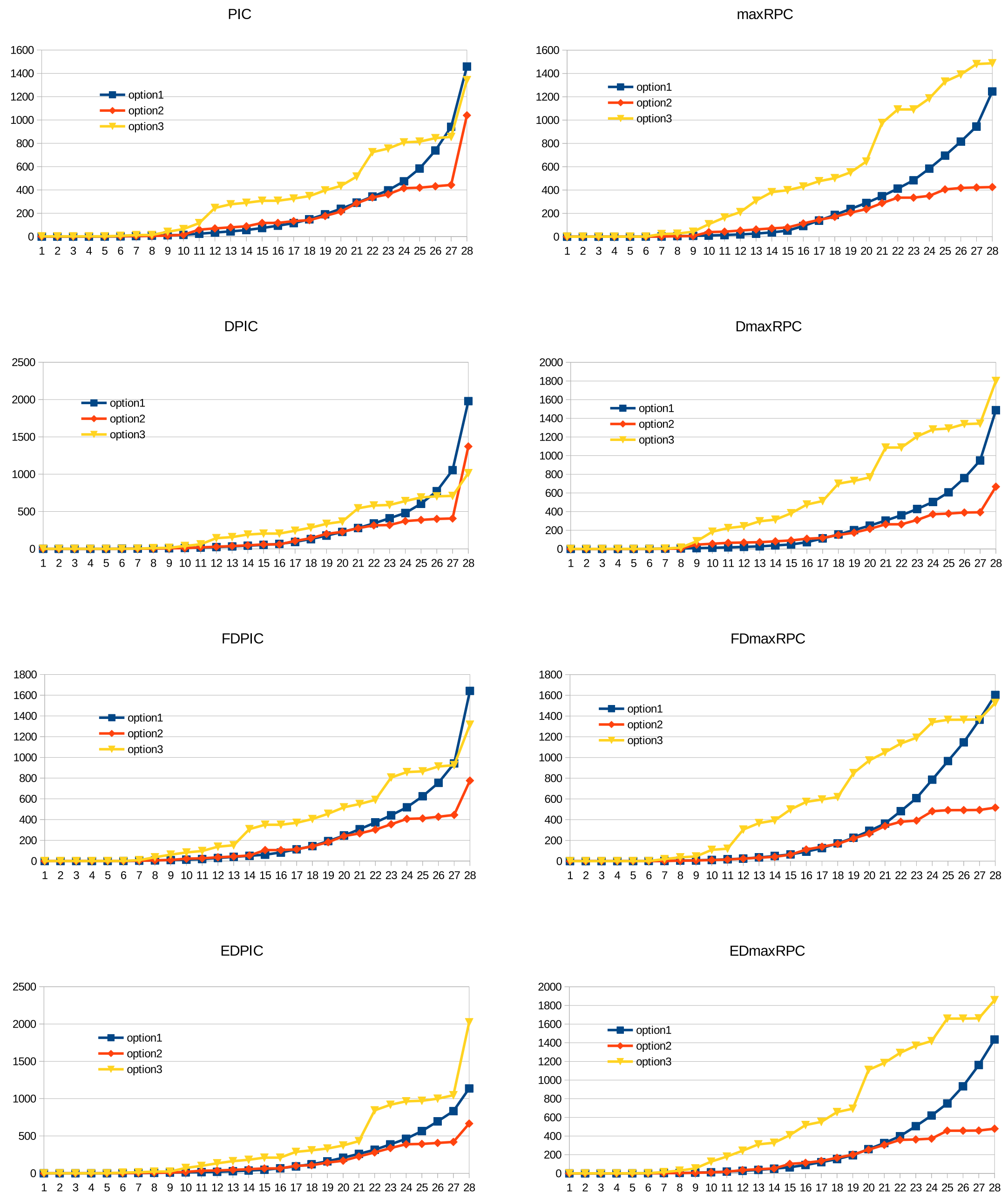


*Figure 4.21: Accumulated mean solving time taken by variants of high order consistencies. Each sub-figure corresponds to a high order consistency and the three lines in it correspond to three ways of applying the corresponding high order consistency. Blue lines (option 1) mean that high order consistencies are enforced during pre-processing and EDAC during search; red lines (option 2) for restricted high order consistencies during pre-processing and EDAC during search; and yellow lines (option 3) for restricted high order consistencies during both pre-processing and search. Axis X represents the categories and Axis Y represents the solving time (in seconds) that is accumulated on each category of benchmarks. For each consistency (i.e., each sub-figure), categories of benchmarks are arranged w.r.t the solving time of option 1.*

### 4.5.5 Conclusion

The experimental results show that high order consistencies are efficient for specific grid graphs that have a small triangle density but the are much more slower than EDAC for graphs of large triangle density. Using restricted high order consistencies for pre-processing can handle this problem in the sense that they outperform original ones on unfavorable problems and give results comparable to EDAC while they are still efficient on favorable problems. Restricted consistencies get significantly slower when maintained during search on the overall set of benchmarks.

## 4.6 Conclusions

In this section, we have proposed a group of new soft consistencies, called high order consistencies, that are an extension of hard RPC, PIC and maxRPC to WCSPs. The new consistencies are strictly stronger than EDAC in the sense that they provide lower bounds much better than EDAC. This is also shown in our experimental results. The improvement in lower bounds may allow to accelerate the search despite their costly time complexity. The experimental results show that high order consistencies, especially EDmaxRPC, are efficient on grid graphs such as ChineseChars, GeomSurf-7 which contain a lot of significant information inside triples of variables. For such kinds of problems, the improvement in lower bounds dominates the costly time complexity of high order consistencies in search. Using high order consistencies for pre-processing allows to solve more instances in less time on average than EDAC

However, high order consistencies have not a good behavior on graphs having a large triangle density. On such problems, the time complexity of high order consistencies, especially maxRPCs, dramatically increase. On the one hand, enforcing high order consistencies for all triangles is too costly. On the other hand, maintaining high order consistencies during search is also too costly. The best approach for solving WCSPs seems to be to use a restricted version of high order consistencies during pre-processing. By limiting the number of triangles to be processed, the restricted versions still have a good behavior on favorable problems while providing a result better than the original ones and comparable to EDAC on unfavorable cases.

# Chapter 5

# Conclusions and perspectives

This thesis focused on two directions for efficiently solving Weighted Constraint Satisfaction Problems: 1) improving the efficiency of enforcing the existing soft consistencies in terms of time and 2) proposing new soft consistencies that can provide strong lower-bounds for Branch-and-Bound search. In the first direction, we chose to improve the efficiency of the algorithm enforcing Virtual Arc Consistency and proposed a consistency named Dynamic Virtual Arc Consistency. In the last direction, we used the idea of hard triangle-based consistencies such as RPC, PIC, maxRPC to create new soft high order consistencies for WCSPs.

## 5.1 Dynamic Virtual Arc Consistency

### 5.1.1 Conclusions

By integrating the idea of dynamic arc consistency algorithms with the iterative VAC algorithm for dynamically enforcing classic arc consistency in the harden version of WCSPs, the hybrid method, named Dynamic VAC, can efficiently maintain VAC during iterations of VAC as well as during branching operations in the tree search.

There are two approaches for maintaining Dynamic VAC during search:

- Dynamic VAC is just maintained at each node of the search tree. When the search branches out, VAC is enforced from scratch.
- Dynamic VAC is maintained both inside nodes and during the search tree. When the search branches out, VAC is incrementally enforced from the virtual arc consistent problem of the parent node.

By exploiting both incremental changes caused by branching operations as well as incremental changes of EPTs during successive iterations of VAC, the new fully incremental method outperforms both the direct application of dynamic VAC and the usual maintenance of static VAC on a variety of problems. This is especially true for problems having small graph densities and large domains.

The incompletely incremental method which ignores the incremental changes caused by branching operations can outperform the static VAC on a small set of problems. This method does not save the work done in parent nodes and thus has a performance weaker than the fully incremental method. The application of the domain-based revision heuristic for enforcing AC in the CSP Bool($P$) does not significantly improve the performance of two methods maintaining dynamic VAC in search.

This work on Dynamic Virtual Arc Consistency, has been published in proceedings of international peer-reviewed conferences.

- H. Nguyen, T. Schiex, and C. Bessiere, **Dynamic virtual arc consistency**, in Proceedings of the 28th Annual ACM Symposium on Applied Computing. ACM, 2013, pp. 98– 103.
- H. Nguyen, T. Schiex, C. Bessiere, S. de Givry, **Maintaining Virtual Arc Consistency Dynamically During Search**, in Proc. of ICTAI'2014, Limassol, Cyprus.

### 5.1.2 Perspectives

There are still more opportunities to accelerate algorithms maintaining VAC during search. Indeed, VAC does not require to enforce AC on Bool($P$) but only to detect if the AC−closure is empty or not. Therefore, it is sufficient to identify a single viable value in each domain to conclude. This simplified problem has been solved using so-called Lazy AC algorithms [Schiex et al., 1996] that could also be injected in VAC algorithms to increase their efficiency.

## 5.2 Soft high order consistencies

### 5.2.1 Conclusions

In this thesis, we have proposed 18 soft high order consistencies for WCSPs based on the idea of hard RPC, PIC, and maxRPC used in classical CSPs. Corresponding to each hard consistency, 6 soft variants are defined for WCSPs including the simple, directional, full directional, existential, existential directional and virtual consistency.

In addition to soft arc consistencies, the new consistencies ensure the extensibility of domain values on two extra variables. They can provide lower-bounds stronger than soft arc consistencies by exploiting the combined costs involving unary, binary and ternary terms inside triangles of variables. However, enforcing soft high order consistencies can create new ternary cost functions.

As expected, the experimental results show that in general, high order consistencies provide stronger lower-bounds than soft arc consistencies but enforcing them takes longer than enforcing soft arc consistencies. The quality of lower-bounds are in general consistent with the strength of consistencies. Fortunately, high order consistencies significantly speed up soft arc consistencies on some specific problems. The unfavorable cases for the application of high order consistencies are problems having large triangle densities and large domain size.

This work on high order consistencies remains to be published.

### 5.2.2 Perspectives

We have three perspectives related to high order consistencies. It is to implement (1) VPIC and VmaxRPC; (2) RPCs; (3) new restricted high order consistencies.

#### Implementation of VPIC and VmaxRPC

Firstly, we would like to implement in the `toulbar2` solver VPIC and VmaxRPC because they are respectively the strongest form of soft PICs and maxRPCs. These potential consistencies will be more benificial on the favorable problems of Computer Vision and Pattern Recognition. The idea for enforcing VPIC and VmaxRPC is based on the algorithm enforcing VAC, that is an iterative process of three phases.

- Phase 1: enforces hard PIC and maxRPC in the hardened version of WCSPs $P$. It deletes all values that have no AC/PIC/maxRPC supports in some binary constraints and pairs of values that have no witnesses on some third variables. All the values and pairs of values deleted during the enforcement are stored. The reasons of these deletions are also stored in the killer system.
  - $\mathsf{killer}[i,a] = j$ if $(i,a)$ has no (AC/PIC/maxRPC) support in the binary cost function $c_{ij}$.
  - $\mathsf{killer}[i,a] = (j,k)$ if $(i,a)$ has no AC support in the ternary cost function $c_{ijk}$.
  - $\mathsf{killer}[i_a,j_b] = k$ if $(i_a,j_b)$ has no witness on $k$.

  The procedure stops whenever a domain is wiped-out. In this case, there exists a sequence of EPTs when applied in WCSPs increasing the lower-bound. This sequence will be defined in the next phase.
- Phase 2: identifies all the deleted values and pairs of values that are necessary for the wipe-out by tracing back the propagation history defined by the killer system and the queue of deleted elements (values and pairs of values). Similary to VAC, this phase computes (1) the maximum possible increase $\lambda$ achievable in the lower bound and (2) the costs to be moved in WCSPs for archiving this increase. They are:
  - $K_i(a)$: the number of $\lambda$ that is projected on $(i,a)$ from $\mathsf{killer}[i,a]$.
  - $K_{ij}(a,b)$: the number of $\lambda$ that is projected on $(i_a,j_b)$ from $\mathsf{killer}[i_a,j_b]$.
  - $K_{ij}(i,a)$: the number of $\lambda$ that $(i,a)$ extends to $c_{ij}$
  - $K_{ijk}(i,a,j,b)$: the number of $\lambda$ that $(i_a,j_b)$ extends to $c_{ijk}$.
- Phase 3: applies EPTs defined in Phase 2 to modify WCSPs by extending and projecting amounts of costs defined by $K$ at deleted values and pairs of values that are necessary for the wipe-out.

#### Implementation of RPCs

Secondly, Soft Restricted Path Consistencies will be also implemented in order to solve problems having large triangle density better than the stronger consistencies, that are

PICs, maxRPCs. The number of arc supports per value in binary cost functions can be maintained whenever there is an unary cost projection or extension.

- An unary cost projection from $c_{ij}$ to a value $(i, a)$ is decomposed in 1) an increase in the unary cost $c_i(a)$ and 2) a decrease in the binary cost function $c_{ij}$ a. Every adjacent value $(k, c)$, $k < i$, that are fully supported by $(i, a)$ decreases its number of full supports by 1. If the binary cost $c_{ij}(a, b)$ of a pair of values $(i_a, j_b)$ decreases to 0, the number of simple arc supports of $(i, a)$, $(j, b)$ will increase by 1.
- An unary cost extension from $(i, a)$ to $c_{ij}$ is decomposed in 1) a decrease in the unary cost $c_i(a)$ and 2) an increase in the binary cost function $c_{ij}$. If $c_i(a)$ decreases to 0, the number of full supports of every adjacent values $(k, c)$ such that $k < c$ and $c_{ik}(a, c) = 0$, will increase by 1. If the binary cost $c_{ij}(a, b)$ of a pair of values $(i_a, j_b)$ increases from 0, the number of simple arc supports of $(i, a)$, $(j, b)$ will decrease by 1.

The maitenance of the number of support seems to be expensive but it is expected that RPCs process less triangles than PICs, maxRPCs and have a much smaller time complexity on problems having large triangle density.

The procedure for searching for RPC supports and witnesses is similar to Procedures find(Full)maxRPCSupport, findWitness_project and findWitness_project used for maxRPC, except for the check for the number of arc supports.

### Implementation of new restricted high order consistencies

Finally, we would like to implement other restricted versions of high order consistencies. In this thesis, a restricted version of high order consistencies has been proposed. These restricted consistencies select to process only a limited number of triangles based on the mean binary cost inside triangles. We can define other restricted versions by using other heuristics for arranging and selecting triangles of variables as well as by defining another threshold on the number of used triangles.

The first heuristic is based on domain size: the triangle of variables that has the smallest sum of the three variable domain sizes is the most preferred. The second heuristic is based on the sum of unary costs in three variable domains of triangles: the most preferred triangle has the largest such sum of costs. It is expected that a domain of small size or a domain containing less values of zero-cost have more potential to be node inconsistent (i.e. every value in the domain has a non-zero cost) after enforcing soft consistencies.

# Bibliography

D. Allouche, C. Bessiere, P. .Boizumault, S. Givry, P. Gutierrez, S. Loudni, J. Metivier, and T. Schiex. Decomposing global cost functions. In *Proc. of AAAI*, 2012.

D. Allouche, I. André, S. Barbe, J. Davies, S. de Givry, G. Katsirelos, B. O'Sullivan, S. Prestwich, T. Schiex, and S. Traoré. Computational protein design as an optimization problem. *Artificial Intelligence*, 212:59–79, 2014a.

D. Allouche, S. de Givry, B. Hurley, G. Katsirelos, B. O'Sullivan, and T. Schiex. Une comparaison de logiciels d'optimisation sur une large collection de modèles graphiques. In *Proc. of JFPC-14*, 2014b.

P. J. Assef Chmeiss. Path-consistency: When space misses time. In A. Press, editor, *Proc. of AAAI'96*, pages 196–201, 1996.

T. Balafoutis and K. Stergiou. Exploiting constraint weights for revision ordering in arc consistency algorithms. In *Proceedings of the ECAI-08 Workshop on Modeling and Solving Problems with Constraints*, 2008a.

T. Balafoutis and K. Stergiou. On conflict-driven variable ordering heuristics. In *Proceedings of the ERCIM workshop on Constraint Solving and Constraint Logic Programming*, 2008b.

T. Balafoutis, A. Paparrizou, K. Stergiou, and T. Walsh. New algorithms for max restricted path consistency. *Constraints*, 16:372,406, 2011.

A. Balafrej, C. Bessiere, R. Coletta, and E. Bouyakhf. Adaptive parameterized consistency. In *Proc. of CP'2013*, 2013.

R. Barták and P. Surynek. An improved algorithm for maintaining arc consistency in dynamic constraint satisfaction problems. In *Proc. of the* $18^{th}$ *International FLAIRS Conference*, pages 161–166, Menlo Park, CA, USA, 2005. AAAI Press.

R. Barták and P. Surynek. An improved algorithm for maintaining arc consistency in dynamic constraint satisfaction problems. In *International Florida AI Research Society Conference*, pages 161–166, 2005.

E. Bensana, M. Lemaître, and G. Verfaillie. Earth observation satellite management. *Constraints*, 4(3):293–299, 1999.

P. Berlandier. Improving domain filtering using restricted path consistency. In *Proceedings IEEE Conference on Artificial Intelligenece and Applications (CAIA'95)*, 1995.

P. Berlandier and B. Neveu. Maintaining Arc Consistency through Constraint Retraction. In *Proc. of the 6$^{th}$ IEEE International Conference on Tools with Artificial Intelligence (TAI94)*, New Orleans, LA, 1994.

C. Bessière. Arc-consistency in dynamic constraint satisfaction problems. In *Proc. of AAAI'91*, pages 221–226, Anaheim, CA, 1991.

C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

C. Bessiere. Constraint propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.

C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proc. IJCAI'2001*, pages 309–315, 2001.

C. Bessiere, E. Freuder, and J. Regin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999.

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

F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the Constraint Satisfaction Problem. In *CP'04 Workshop on Constraint Propagation and Implementation*, pages 9–43, Toronto, Canada, Sept. 2004a.

F. Boussemart, F. Hemery, C. Lecoutre, and L. S. Lakhdar. Boosting systematic search by weighting constraints. In *16th European Conference on Artificial Intelligence(ECAI'04)*, pages 146–150, Valencia, Spain, aug 2004b.

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

A. Chmeiss. Sur la consistance de chemin et ses formes partielles. *In the Actes du Congres AFCET- RFIA*, pages 212–219, 1996.

M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual Arc Consistency for Weighted CSP. In *Proc. of AAAI'2008*, Chicago, USA, 2008.

M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174:449–478, 2010.

M. C. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3):311–342, 2003.

M. C. Cooper. Cyclic consistency: a local reduction operation for binary valued constraints. *Artificial Intelligence*, 155(1-2):69–92, 2004.

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

M. C. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2):199–227, 2004.

M. C. Cooper, S. de Givry, and T. Schiex. Optimal soft arc consistency. In *Proc. of IJCAI'2007*, pages 68–73, Hyderabad, India, Jan. 2007.

R. Debruyne. Arc-consistency in dynamic CSPs is no more prohibitive. In *Proceedings Eighth IEEE International Conference on Tools with Artificial Intelligence (ICTAI-1996)*, pages 299–306. IEEE, 1996.

R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proc. of CP'97*, number 1330 in LNCS, pages 312–326, Linz, Austria, Nov. 1997a. Springer-Verlag.

R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proc. of the 15th IJCAI*, pages 412–415, Nagoya, Aichi, Japan, 1997b.

R. Debruyne and C. Bessiere. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, pages 205–230, 2001.

R. Dechter. Tractable structures for constraint satisfaction problems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 7. Elsevier, 2006.

R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proc. of AAAI'88*, pages 37–42, St. Paul, MN, 1988.

R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, chapter 11, pages 370–425. Springer-Verlag, 1988.

D. Dehani, C. Lecoutre, and O. Roussel. Extension des cohérences wcsps aux tuples. In *Proc. of JFPC-13*, May 2013.

G. F. and I. G. Improved algorithms for max-restricted path consistency. In *Proc. of CP*, volume 2833, pages 858–862. Springer,Heidelberg, 2003.

A. Favier, S. de Givry, A. Legarra, and T. Schiex. Pairwise decomposition for combinatorial optimization in graphical models. In *Proc. of IJCAI'11*, Barcelona, Spain, 2011.

E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21: 958–966, Nov. 1978.

E. C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(14):755–761, 1985.

E. C. Freuder and C. D. Elfe. Neighborhood inverse consistency preprocessing. In *Proc. of AAAI'96*, Portland, OR, Aug. 1996.

S. W. Golomb and L. D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, Oct. 1965. ISSN 0004-5411.

C. Han and C. Lee. Comments on mohr and henderson's path consistency algorithm. *Artificial Intelligence*, 36:125–130, 1988.

R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

H. H. Hoos and E. Tsang. Local search methods. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 5. Elsevier, 2006.

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

A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

J. Larrosa. On arc and node consistency in weighted CSP. In *Proc. AAAI'02*, pages 48–53, Edmondton, (CA), 2002.

J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted CSP. In *Proc. of the 18th IJCAI*, pages 239–244, Acapulco, Mexico, Aug. 2003.

J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artif. Intell.*, 159(1-2):1–26, 2004.

J. Larrosa, S. de Givry, F. Heras, and M. Zytnicki. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *Proc. of the 19th IJCAI*, pages 84–89, Edinburgh, Scotland, Aug. 2005.

J. Larrosa, F. Heras, and S. de Givry. A logical approach to efficient max-sat solving. *Artif. Intell.*, 172(2-3):204–233, 2008.

E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

C. Lecoutre, F. Hemery, et al. A study of residual supports in arc consistency. In *IJCAI*, volume 7, pages 125–130, 2007.

J. Lee and K. Leung. Consistency techniques for flow-based projection-safe global cost functions in weighted constraint satisfaction. *Artificial Intelligence*, 43:257–292, 2012.

J. Lee and K. L. Leung. Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. In C. Boutilier, editor, *IJCAI*, pages 559–565, 2009.

J. Lee and K. L. Leung. A stronger consistency for soft global constraints in weighted constraint satisfaction. In M. Fox and D. Poole, editors, *AAAI*. AAAI Press, 2010.

C. Likitvivatanavong, Y. Zhang, J. Bowen, and E. C. Freuder. Arc consistency in mac: a new perspective. *Proceedings of CPAI*, 4:93–107, 2004.

A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.

R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28 (2):225–233, 1986.

U. Montanari. Network of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

M. Mouhoub. Arc consistency for dynamic CSPs. In *Proc. Knowledge-Based Intelligent Information and Engineering Systems (KES 2003)*, pages 393–400. Springer, 2003.

H. Nguyen, T. Schiex, and C. Bessiere. Dynamic virtual arc consistency. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 98–103. ACM, 2013.

M. Quéva, C. Probst, and L. Ricci. Maintaining arc consistency in non-binary dynamic CSPs using simple tabular reduction. In *Proceedings of the 2010 conference on STAIRS 2010: Proceedings of the Fifth Starting AI Researchers' Symposium*, pages 251–263. IOS Press, 2010.

R.Debruyne. A property of path inverse consistency leading to an optimal algorithm. In *Proc. ECAI'2000*, pages 88–92, Berlin, Germany, 2000.

M. Sánchez, S. de Givry, and T. Schiex. Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints*, 13(1-2):130–154, 2008.

M. Sanchez, D. Allouche, S. de Givry, and T. Schiex. Russian doll search with tree decomposition. In *Proc. IJCAI'09*, pages 603–608, San Diego (CA), USA, 2009.

T. Schiex. Arc consistency for soft constraints. In *Principles and Practice of Constraint Programming - CP 2000*, volume 1894 of *LNCS*, pages 411–424, Singapore, Sept. 2000.

T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: hard and easy problems. In *Proc. of the $14^{th}$ IJCAI*, pages 631–637, Montréal, Canada, Aug. 1995.

T. Schiex, J.-C. Régin, C. Gaspin, and G. Verfaillie. Lazy arc consistency. In *Proc. of AAAI'96*, Portland, OR, Aug. 1996. AAAI Press.

M. Singh. Path consistency revisited. In *ICTAI'95 (IEEE Conference of Tools for Artificial Intelligence)*, Washington D.C., 1995.

P. Surynek and R. Barták. A new algorithm for maintaining arc consistency after constraint retraction. In *Proc. Principles and Practice of Constraint Programming – CP 2004*, number 3258 in LNCS, pages 767–771, Toronto, Canada, 2004.

S. Traoré, D. Allouche, I. André, S. de Givry, G. Katsirelos, T. Schiex, and S. Barbe. A new framework for computational protein design through cost function network optimization. *Bioinformatics*, 29(17):2129–2136, 2013.

P. van Beek. Backtracking search algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 4. Elsevier, 2006.

J. Vion and R. Debruyne. Light algorithms for maintaining maxrpc during search. In *Proceedings of SARA*, 2000.

R. Wallace and E.C.Freuder. Ordering heuristic for arc consistency algorithms. In *Proc. of NCCAI'92*, pages 163–169, 1992.

W. Zhang. Phase transitions and backbones of 3-SAT and maximum 3-SAT. In *Proc. of the $7^th$ International Conference on Principles and Practice of Constraint Programming (CP-01)*, volume 2239 of *LNCS*, pages 153–167, Paphos, Cyprus, Nov. 2001. Springer.

J. Zlomek and R. Bartak. Full arc consistency in wcsp and in constraint hierarchies with finite domains. In *Proc. CP'2005*, 2005.

## Résumé

Cette thèse se focalise sur l'étude de cohérences locales fortes afin de résoudre des problèmes d'optimisation sur des réseaux de fonctions de coûts (ou réseaux de contraintes pondérées). Ces méthodes fournissent le minorant nécessaire pour des approches de type "Séparation-Evaluation". Nous étudions dans un premier temps la cohérence d'Arc virtuelle (VAC), une des plus fortes cohérences d'arcs du domaine, qui est établie via l'établissement de la cohérence d'arc dure dans une séquence de réseaux de contraintes classiques. L'algorithme itératif pour établir VAC est amélioré via l'introduction d'une incrémentalité accrue, exploitant la cohérence d'arc dynamique. La nouvelle méthode est aussi capable de maintenir VAC efficacement pendant la recherche lorsque les réseaux de contraintes pondérées sont dynamiquement modifiés par les opérations de branchement. Dans une seconde partie, nous nous intéressons à des cohérences de domaines plus fortes, inspirées de cohérences similaires dans les réseaux de contraintes classiques (cohérence de chemin inverse, réduite ou Max-réduite). Pour chaque cohérence dure, plusieurs cohérences souples ont été proposées pour les réseaux de contraintes pondérées. Les nouvelles cohérences fournissent un minorant plus fort que celui des cohérences d'arc souples en traitant les triplets de variables connectées deux à deux par des fonctions de coûts binaires. Dans cette thèse, nous étudions les propriétés des nouvelles cohérences, les implémentons et les testons sur une variété de problèmes.

**Mots clés:** CSP pondéré • Réseaux de fonctions de coûts • Cohérences locales fortes • Cohérence d'arc dynamique • Cohérence d'arc virtuelle •

## Abstract

This thesis focuses on strong local consistencies for solving optimization problems in cost function networks (or weighted constraint networks). These methods provide the lower bound necessary for Branch-and-Bound search. We first study the Virtual arc consistency, one of the strongest soft arc consistencies, which is enforced by iteratively establishing hard arc consistency in a sequence of classical Constraint Networks. The algorithm enforcing VAC is improved by integrating the dynamic arc consistency to exploit its incremental behavior. The dynamic arc consistency also allows to improve VAC when maintained VAC during search by efficiently exploiting the changes caused by branching operations. Secondly, we are interested in stronger domain-based soft consistencies, inspired from similar consistencies in hard constraint networks (path inverse consistency, restricted or Max-restricted path consistencies). From each of these hard consistencies, many soft variants have been proposed for weighted constraint networks. The new consistencies provide lower bounds stronger than soft arc consistencies by processing triplets of variables connected two-by-two by binary cost functions. We have studied the properties of these new consistencies, implemented and tested them on a variety of problems.

**Keywords:** Weighted CSP • Cost Function Networks • Strong local consistencies • High order consistencies • Dynamic arc consistency • Virtual arc consistency •