Global Constraints in Distributed Constraint Satisfaction and Optimization

Christian Bessiere, Ismel Brito, Patricia Gutierrez, Pedro Meseguer

Global Constraints in Distributed Constraint Satisfaction and Optimization

Christian Bessiere[1] Ismel Brito[2] Patricia Gutierrez[2] Pedro Meseguer[2]
[1] University of Montpellier
bessiere@lirmm.fr
[2] IIIA - CSIC, Universitat Autònoma de Barcelona
ismel@iiia.csic.es, patricia@iiia.csic.es, pedro@iiia.csic.es

Abstract

Global constraints are an essential component in the efficiency of centralized constraint programming. We propose to include global constraints in distributed constraint satisfaction and optimization problems (DisCSPs and DCOPs). We detail how this inclusion can be done, considering different representations for global constraints (direct, nested, binary). We explore the relation of global constraints with local consistency (both in the hard and soft cases), in particular for generalized arc consistency (GAC). We provide experimental evidence of the benefits of global constraints on several benchmarks, both for distributed constraint satisfaction and for distributed constraint optimization.

1 Introduction

With the rise of Internet, there are more and more opportunities to solve problems in a distributed form. Distribution implies that different problem parts are handled by different agents, and these parts cannot be joined in a single agent for a centralized solving. This new setting requires the adaptation of existing solving strategies or the generation of new ones. Often, problems are solved by message passing [1].

The same trend is observed in constraint programming. A strong motivation for distributed constraint solving is privacy. Constraints contain information that agents may desire to keep hidden from other agents, which could be seen as competitors. Usually, it is assumed that a constraint is known by the agents involved in it, but not by the other agents [2]. Distributed constraint reasoning appears as a natural extension of the usual centralized approach to constraint reasoning, keeping its solving capabilities but removing the implicit assumption that every detail of the instance is known by the solving agent. In the last years a number of distributed algorithms have been proposed: ABT [2], AFC [3] ADOPT [4], NCBB [5], among others.

In centralized constraint reasoning, global constraints have been an essential component of the efficiency of constraint solvers [6]. A global constraint C is a class of constraints that all have the same specific semantics but that can involve any (unbounded) number of variables. The standard example is the *all-different* constraint, that requires that all the involved variables must take a different value. You can apply this constraint to sets of variables of any size. Each application is considered as a constraint instance. The exploitation of the semantics associated with a global constraint allows to design specialized propagators able to reach local consistency levels (typically generalized arc consistency), usually with lower complexity than generic propagators. Current constraint solvers include a list of global constraints, with propagators already implemented, available for the user to model and solve her problem [7].

Often, it is implicitly assumed that distributed constraint reasoning precludes the use of global constraints. The usual assumption is that an agent knows the constraint with each of its neighbors separately, and nothing else [2, 4]. These constraints are obviously binary. However, this interpretation is too restrictive because there are distributed applications for which it is natural to use global constraints. For example, let us consider a distributed meeting scheduling problem where agent a_1 is trying to find an appointment with agents a_2 and a_3. Agent a_1 may easily infer that there is an *all-equal* constraint between a_1, a_2 and a_3.

When adding global constraints in distributed reasoning we obtain several benefits. First, the expressivity of distributed constraint reasoning is enhanced because there are relations among variables with a particular meaning that cannot be expressed as a conjunction of binary relations (there are global constraints that are not binary decomposable). Second, the solving process can be done more efficiently. Local consistency can be enforced more efficiently when global constraints are involved [6]. Hence, assuming a solving strategy maintaining some kind of local consistency, the inclusion of global constraints improves the efficiency of the solving process, requiring less computational resources.

Once the interest of global constraints in distributed constraint reasoning is accepted, another question naturally follows: since some global constraints can be decomposed in simpler constraints, what is more efficient, to leave the global constraint as it was initially posted or to decompose it? If several decompositions are possible, which offers the best performance? We provide some answers to these questions, exploring two kinds of decompositions (binary [8] and nested for contractible constraints [9]) against the global constraint without decomposition, in two contexts: complete distributed search with / without unconditional GAC maintenance [10].

Previous paragraphs have introduced what we consider as the two main contributions of this paper: (i) the inclusion of global constraints in distributed constraint reasoning, and (ii) the exploration of different representations for global constraints in a distributed context, looking for the most efficient one. To perform this task, we use state-of-the-art distributed constraint solving algorithms, which have already been combined with some form of local consistency [10, 11, 12].

All what we have said equally applies to *hard* global constraints (global constraints which should mandatorily be satisfied, satisfaction case) and to *soft* global constraints, (global constraints which should be satisfied *as much as possible*, optimization case). Although the satisfaction case can be seen as a special case of the more general optimization one, the former uses different algorithmic techniques. For this reason, in the following, after giving motivation (Section 2) and background for this work (Section

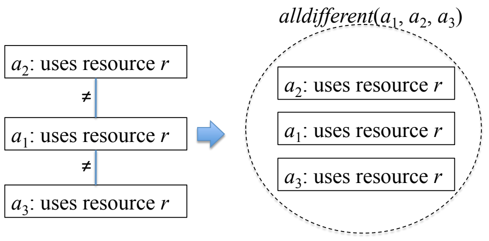


Figure 1: Job-shop scheduling problem, where agents a_1, a_2, a_3 share the same resource r. To determine its starting time, agent a_1 knows it must be different from a_2 and a_3. But because they use the same resource r, a_1 can deduce that there is a global *all-different* constraint among them.

3), we treat separately the inclusion of hard global constraints in distributed constraint satisfaction (Section 4) and the inclusion of soft global constraints in distributed optimization (Section 5). In Section 6 we terminate with some conclusions of this work.

2 Motivation

2.1 Satisfaction Case

In distributed constraint satisfaction (DisCSP) agents cooperate to find a global solution. It is assumed that each agent knows a part of the problem (the variables owned by the agent and the constraints in which they are involved) but no agent knows the whole problem. Often it is also implicitly assumed that constraints are binary, under the idea that an agent relates with another agent independently of how it relates with others. However, this view precludes the use of global constraints. In some situations global constraints naturally appear in the distributed context.

First, a global constraint appears in DisCSP when an agent can infer the existence of a global constraint from the task it performs and from the existence of some other constraints (usually binary). For example, consider the distributed job-shop scheduling problem. If agent a_1 is trying to set up the starting time of its task which uses resource r, with other two agents a_2 and a_3 which also use the same resource, a_1 may deduce that there is an *all-different* constraint between a_1, a_2 and a_3. This example appears in Figure 1.

Second, a global constraint appears in DisCSP when the model designer informs the agents of the existence of that global constraint. This information is needed to properly accomplish the intended task. For example, let us consider the task of a PC configuration, which consists in selecting a motherboard and a number of options according to the user desires. A motherboard for PC contains the CPU plus slots for memory cards, slots for disks (hard disks and DVDs), and slots for any other device. Configuring a PC means deciding which motherboard to choose and which devices are connected to the motherboard slots, satisfying the user specification. A natural constraint model of this problem contains variables for the motherboard and its slots. A solution means selecting the values for these variables, such that all constraints are satisfied. There are a number of technical constraints: only 1 motherboard, binary constraints between the motherboard and the other PC elements (memories, disks, etc.) ensuring compatibility, at least one hard disk is needed (to perform system bootstrapping). In addition, there are a number of global constraints among variables representing elements of the same type (or connecting to the same slot type) establishing the desired minimum and maximum number of elements according to the user specification (*atleast* and *atmost* constraints). Other global constraints may exist on the total desired capacity of a particular element and on the maximum budget (*sum* constraint).

Following with this example, let us consider a multi-agent context. There are several agents providers of motherboards $(C_1, ..., C_p)$, memories $(M_1, ..., M_q)$, disks $(D_1, ..., D_r)$, and other devices $(O_1, ..., O_s)$. We group agents in four sorts, c, m, d, o, for providers of motherboards, memories, disks and other devices. Each agent has four vectors of variables *id*$[1...K_s]$, *type*$[1...\ K_s]$, *capacity*$[1...K_s]$, *price*$[1...K_s]$. K_s allows enough elements for the maximum number of slots of sort s in any motherboard; for motherboard providers these vectors have 1 element only. The obvious meaning of these vectors is: *id*$[i]$ is an item of type *type*$[i]$ with capacity *capacity*$[i]$ and price *price*$[i]$. This is ensured by a quaternary constraint on *id*$[i]$, *type*$[i]$, *capacity*$[i]$ and *price*$[i]$, for all i in $1..K_s$. There is a special value $empty$ for *id*$[i]$ and *type*$[i]$, which forces *capacity*$[i]$ and *price*$[i]$ to be zero. Typically, id contains the item identifier, according to the provider catalog, while $type$ indicates the type of device or connection, with the following values: *motherboard, memory_card, hard_disk, DVD, microphone, speakers, printer, ethernet, USB*. In $capacity$ it is stored the processor speed, the memory card size or the hard disk capacity. For *type = DVD*, *capacity* is 0.

We use the following global constraints in the modelling: $atleast\ [t, v](Vars)$ (value v has to appear at least t times in the assignment of the variables in $Vars$), $atmost[t, v](Vars)$ (value v has to appear at most t times in the assignment of $Vars$), $exactly[t, v](Vars) = atleast[t, v](Vars) \wedge atmost[t, v](Vars)$, $sum(Vars) \leq bound$ (the sum of $Vars$ is lower than or equal to $bound$), $sum(Vars) \geq bound$ (the sum of $Vars$ is greater than or equal to $bound$).

Let us consider the following PC specification: *CPU speed $\geq$ 2.5 GHz; memory between 8GB and 16GB; at most 2 hard disks; total disk storage $\geq$ 1TB; 1 DVD; at least 1 ethernet connection; at least 1 printer connection; at least 2 USB; maximum budget \$500.* This specification is translated into the following constraints:

1. Unary constraint on *capacity*[1] variable of motherboard providers:
capacity[1] $\geq$ 2.5GHz.
2. Only one motherboard is required:
$exactly[1, motherboard]$($type$ vars of $C_1, ..., C_p$).
3. Binary compatibility constraints: between motherboard providers and providers of any other element.
4. Assuming that memory cards for any provider are standard with the same size (4GB), memory specifications can be translated into number of cards among all memory providers:
$atleast[2, memory\_card]$($type$ vars of $M_1, ..., M_q$)
$atmost[4, memory\_card]$($type$ vars of $M_1, ..., M_q$)

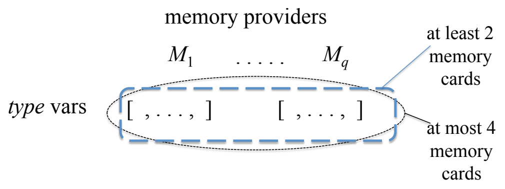


Figure 2: Memory configuration in a multi-agent context, as explained in the text. $M_1, ..., M_q$ are memory provider agents. Each has a vector of $type$ variables, containing the memory cards to be used in the configuration. Following user specification, there are two global constraints, $atleast[2, memory\_card](type$ vars of $M_1, ..., M_q), atmost[4, memory\_card](type$ vars of $M_1, ..., M_q)$. Since these constraints are not binary decomposable, there is no set of binary constraints on the problem variables with the same meaning.

5. Constraints on the number of hard disks:
$atleast[1, hard\_disk](type$ vars of $D_1, ..., D_r)$
$atmost[2, hard\_disk](type$ vars of $D_1, ..., D_r)$
6. Exactly one DVD: $exactly[1, DVD](type$ vars of $D_1, ..., D_r)$
7. Total hard disk storage:
$sum(capacity$ vars of $D_1, ..., D_r) \geq$ 1TB
8. At least one ethernet connection:
$atleast[1, ethernet](type$ vars of $O_1, ..., O_s)$
9. At least one printer connection:
$atleast[1, printer](type$ vars of $O_1, ..., O_s)$
10. At least two USB ports: $atleast[2, USB](type$ vars of $O_1, ..., O_s)$
11. Budget limited to \$500:
$sum(price$ vars of all providers$) \leq$ \$500

Constraints 2 and 4–11 are global constraints on the number, capacity or price of the PC components.

A final example comes from the sensor network application reported in [13]. There are a set of sensors and a set of mobiles. The goal is to track each mobile with 3 sensors, under some constraints of visibility (between sensors and mobiles) and compatibility (among sensors). In the model of [13], agents are mobiles, containing three variables for the three tracking sensors. If global constraints are allowed it becomes possible to provide a more natural model of this problem with sensors as agents, under an *atleast* global constraint requiring at least 3 sensors to track each mobile.

In summary, an agent may be related by one or several global constraints together with a subset of the agents. This information can be deduced by the agent, or provided by an external source. Although binary constraints are the obvious choice in distributed constraint reasoning, some global constraints can also be used for modelling and solving these problems. In some cases, global constraints are needed for expressivity requirements. By exploiting global constrains some efficiency improvements can

$x_1 = \{a\}$
$x_2 = \{a, b\}$
$x_3 = \{a, b\}$

x_1	x_2	x_3	μ_{var}
a	a	a	2
a	a	b	1
a	b	a	1
a	b	b	1

soft-all-different$_{x_1,x_2,x_3}(a, a, a) = 2$

x_1	x_2	μ_{var}
a	a	1
a	b	0

x_1	x_3	μ_{var}
a	a	1
a	b	0

x_2	x_3	μ_{var}
a	a	1
a	b	0
b	a	0
b	b	1

soft-all-different$_{x_1,x_2}(a, a)$ + *soft-all-different*$_{x_1,x_3}(a, a)$ + *soft-all-different*$_{x_2,x_3}(a, a) = 3$

Figure 3: (Top) *soft-all-different* soft global constraint with μ_{var} violation measure; (bottom) its binary decomposition

be achieved.

2.2 Optimization Case

A soft global constraint is a global constraint plus a violation measure that defines the costs of value assignments based on the semantics of the global constraint and the amount of violation. For example, the *soft-all-different(T)* is associated with the violation measure μ_{var}, defined as the number of variables in T that have to change their value to satisfy that all values are different. Another violation measure for *soft-all-different* is μ_{dec}, defined as the number of pairs of variables in T with the same value [14].

As in the satisfaction case, binary soft constraints is a common assumption. However, not every soft constraint can be decomposed into an equivalent set of binary ones. For example, consider the *soft-all-different* soft global constraint in Figure 3 defined over variables x_1, x_2, x_3. The cost of every value tuple is defined by the violation measure μ_{var}. Observe that the tuple $\{x_1 = a, x_2 = a, x_3 = a\}$ has a different cost in the global formulation —involving all variables— and in the binary formulation. Hence, this constraint is not binary decomposable with the violation measure μ_{var}.[1] In general, most soft global constraints are not binary decomposable, so working with their global formulations is crucial for their effective inclusion in DCOPs.

3 Background

In this section we introduce all the necessary material on distributed reasoning and global constraints, both in the case of satisfaction problems and in the case of opti-

[1]Notice, however, that the *soft-all-different* constraint is binary decomposable with the violation measure μ_{dec} [14].

mization problems.

3.1 Satisfaction

CSP. A *Constraint Satisfaction Problem* (CSP) consists in finding solutions to a constraint network. A *constraint network* is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of finite domains such that $D(x_i)$ is the value set for x_i, and $\mathcal{C}$ is a finite set of constraints. A constraint $C(T) \in \mathcal{C}$ on the ordered subset of variables $T = (x_{i_1}, \ldots, x_{i_{r(i)}})$ specifies the set of tuples on $x_{i_1}, \ldots, x_{i_{r(i)}}$ allowed by $C(T)$. The set of allowed tuples can be defined by a table in extension, or by any Boolean function. A *solution* is an assignment of values to variables which satisfies every constraint. Solving a CSP is NP-complete.
Generalized Arc Consistency. Given a constraint $C(T)$, a pair (x_i, a), $x_i \in T$, $a \in D(x_i)$ is *generalized arc consistent* (GAC) with respect to $C(T)$ if there exists a tuple t on T such that the projection $t[x_i]$ of t on x_i is a, t is allowed by $C(T)$, and for every $x_j \in T, j \neq i$, $t[x_j] \in D(x_j)$; t is a *support* of a with respect to $C(T)$. Variable x_i is GAC if all its values are GAC with respect to every constraint involving x_i; a CSP is GAC if every variable is GAC.
Global Constraints. A *global constraint* captures a relation with a specific semantics that can apply on any number of variables [6]. C is a class of constraints defined by a Boolean function f_C whose arity is not fixed. Constraints with different arities can be defined by the same Boolean function. For instance, *all-different*(x_1, x_2, x_3) and *all-different* (x_1, x_4, x_5, x_6) are two instances of the *all-different* global constraint, where $f_{\textit{all-different}}(T)$ returns true iff $x_i \neq x_j, \forall x_i, x_j \in T$. In [9], Maher has defined the property of *contractibility*. A global constraint C is *contractible* iff for any tuple t on $x_{i_1}, \ldots, x_{i_{p+1}}$, if t satisfies $C(x_{i_1}, \ldots, x_{i_{p+1}})$ then the projection $t[x_{i_1}, \ldots, x_{i_p}]$ of t on $x_{i_1}, \ldots, x_{i_p}$ satisfies $C(x_{i_1}, \ldots, x_{i_p})$ [9].

In [8], Bessiere and Van Hentenryck defined the property of *decomposability without extra variables*. A global constraint C is *binary-decomposable without extra variables* iff for any instance $C(T)$ of C, there exists a set S of binary constraints involving only variables in T such that the solutions of S are the solutions of $C(T)$ [8]. S is called a *binary decomposition* of $C(T)$.
DisCSP. A *Distributed Constraint Satisfaction Problem* (DisCSP) is a 5-tuple ($\mathcal{X}$, $\mathcal{D}$, $\mathcal{C}$, $\mathcal{A}$, ϕ), where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ form a CSP whose variables, domains and constraints are distributed among automated agents. In addition, $\mathcal{A} = \{1, \ldots, p\}$ is a set of p agents, and $\phi : \mathcal{X} \rightarrow \mathcal{A}$ is a function that maps each variable to its agent. We make the usual assumption that each agents owns exactly one variable, so agents and variables can be used interchangeably. We also assume that a constraint $C(T)$ among several variables is known by every agent that owns a variable of T [2]. The set of constraints known by agent i is written $\mathcal{C}_i$. As in the centralized case, a *solution* is an assignment of values to variables satisfying every constraint. DisCSPs are solved by the coordinated action of agents, which communicate through messages. It is assumed that the delay of a message is finite but random. For each pair of agents, message delivery follows the sending order.
ABT. *Asynchronous Backtracking* (ABT) [2] is the reference algorithm for DisCSPs, with a role similar to backtracking in centralized CSPs. ABT is purely asynchronous,

each agent makes its own decisions, informs other agents about them, and no agent has to wait for the others' decisions. An ABT agent computes a global consistent solution (or detects that no solution exists) in finite time; its correctness and completeness have been proved [2, 15].

ABT requires a total order among agents, inducing a direction in the constraints. A binary constraint causes a directed link between the two constrained agents: the value-sending agent, from which the link starts, to the constraint-evaluating agent, at which the link ends. Each ABT agent keeps its own agent view and nogood store. The agent view of a generic agent $self$ is the set of values that $self$ believes are assigned to higher priority agents (connected to $self$ by incoming links). Its nogood store keeps nogoods as justifications of inconsistent values. A *nogood* is a conjunction of assignments $x_i = a \land x_j = b \land \ldots x_k = c \land x_p = d$ that is inconsistent, that is, it violates at least one constraint. Often a nogood is written in directed form, as $x_i = a \land x_j = b \land \ldots x_k = c \Rightarrow x_p \neq d$, where x_p should be considered after all other nogood variables in the search tree. The left-hand side (lhs) of the nogood is the expression that appears at the left side of the implication, while the right-hand side (rhs) appears at the right of the implication. Agents exchange four types of messages:

- OK?(i, j, val): i informs j that it has taken value val;
- NGD(j, i, ng): j informs i that it has detected nogood ng that involves i as the last agent in the order;
- ADDL(i, j): i asks j to set up a link from j to i;
- STOP(i, j): i informs j that the empty nogood has been generated and there is no solution.

Agents inform of their assignments by sending OK? messages to lower priority agents, which try to accommodate their assignments to the values taken by higher priority agents. If this is not possible, the lower priority agent sends a NGD message to the lowest higher priority agent in the new nogood generated [2]. If an unconnected agent appears in the nogood, it is requested to set up a new link using the ADDL message.

When ABT starts, each agent assigns its variable, and sends OK? messages containing its assignment to its neighboring agents with lower priority. When $self$ receives an OK? message, $self$ updates its agent view with the new assignment, removes nogoods which are inconsistent with this new assignment and checks the consistency of its own current assignment with the updated agent view.

When $self$ receives a NGD message, it is accepted if the contained nogood is consistent with $self$'s agent view (values of the common variables in the nogood and in $self$'s agent view are the same). Otherwise, $self$ discards the nogood as obsolete. If the nogood is accepted, the nogood store is updated, causing $self$ to search for a new consistent value (since the received nogood forbids its current value). If an unconnected agent i appears in the nogood, it is requested to set up a new link with $self$, by the message ADDL (sent from $self$ to i). From this point on, $self$ will receive the values taken by i. When $self$ cannot find any value consistent with its agent view, either because of the original constraints or because of the received nogoods, new nogoods are generated from its agent view and each one is sent to the lowest agent in it, by NGD messages.

This operation causes backtracking. There are several forms of how new nogoods are generated. In [15], when an agent has no consistent value, it resolves its nogoods following a procedure described in [16]. In this paper we consider this version. For a more detailed description, the reader is addressed to the original source [2] (or consult [15]).

3.2 Optimization

COP. A *Constraint Optimization Problem* (COP) consists in finding optimal solutions to a cost function network. A *cost function network* is defined by $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where: $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of variables. $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of finite domains such that $D(x_i)$ is the value set for x_i. $\mathcal{C}$ is a finite set of cost functions, where every cost function $C(T) \mapsto N \cup \{\infty\}$ on the ordered subset of variables $T = (x_1, \ldots, x_r)$ specifies the costs of every combination of values on T. When a cost function $C(T)$ is evaluated on a value tuple t we follow the notation: $C(T)(t)$. For example, cost function $C(x_1, x_2)$ evaluated on the tuple (a, b) is denoted $C(x_1, x_1)(a, b)$.

The cost of a tuple t is calculated aggregating all individual cost functions evaluated on t. Considering $\top$ as the lowest unacceptable cost, a *solution* is a tuple t containing a complete variable assignment with cost lower than $\top$. An *optimal solution* is a solution with minimum cost. In some cases, $\top$ is assumed unbounded and it is not explicitly stated.

Soft Global Constraints. A soft global constraint C is a class of soft constraints whose arity is not fixed, which is determined by a hard global constraint plus a violation measure μ. Soft global constraints with different arities can be defined by the same class. For instance, *soft-all-different*(x_1, x_2, x_3) and *soft-all-different*(x_1, x_4, x_5, x_6) are two instances of the *soft-all-different* soft global constraint. A soft global constraint C with measure μ is *contractible* iff μ is a non-decreasing function [17].[2] A soft global constraint C with violation measure μ admits a *binary decomposition* iff for any instance $C(x_1, \ldots, x_p)$ of C, there exists a set S of binary soft constraints involving only variables $x_1, \ldots, x_p$ such that for any value tuple t on $x_1, \ldots, x_p$, $\sum_{C(x_i,x_j)\in S} C(x_i, x_j)(t[x_i, x_j]) = \mu(t)$. For example, consider the following soft global constraints:

- *soft-all-different*(T). This soft global constraint expresses that all variable values in T should be different. Costs are defined by violation measures μ_{var} and μ_{dec} [14]: μ_{var} is the number of variables in T that have to change their values to satisfy that all values are different, while μ_{dec} is the number of pairs of variables with the same value. The *soft-all-different* constraint is contractible and binary decomposable with measure μ_{dec}, and contractible but not binary decomposable with measure μ_{var} .

- *soft-at-most[k,v]*(T). This soft global constraint expresses that at most k variables in T should take value v. Costs are defined by violation measure μ_{var},

[2]Function f on a sequence is non-decreasing if $f(\mathbf{a}) \leq f(\mathbf{b})$, for every sequence $\mathbf{a}$ and $\mathbf{b}$ such that $\mathbf{a}$ is a prefix of $\mathbf{b}$ [17].

which is the number of variables in T that have to change to satisfy this condition. The *soft-at-most[k,v]* constraint with μ_{var} is contractible but not binary decomposable.

Often, soft constraints are implemented by cost functions (assuming the weighted model of soft constraints [18]). A cost function maps each possible value tuple of its variables into natural numbers including zero and ∞. Completety permitted tuples have zero cost, completely forbidden tuples have ∞ cost, and partially permitted/forbidden tuples have intermediate costs. In this model, the goal is to find the assignment of values to all variables with minimum aggregated cost, where aggregation is ordinary addition. From now on, we use cost functions to implement soft constraints.

Soft Arc Consistency. Let us consider a COP: (x_i, a) means x_i taking value a, $\top$ is the lowest unacceptable cost, $C(x_i)$ is the unary cost function on x_i values, C_ϕ is a zero-ary cost function that represents a lower bound of the cost of any solution. As [19, 20], we consider the following local consistencies:

- *Node Consistency**: (x_i, a) is node consistent* (NC*) if $C_\phi + C(x_i)(a) < \top$; x_i is NC* if all its values are NC* and there is $a \in D(x_i)$ s.t. $C(x_i)(a) = 0$; a problem is NC* if every variable is NC*.

- *Generalized Arc Consistency**: (x_i, a) is generalized arc consistency (GAC) wrt. a non-unary cost function $C(T)$, if there exists a value tuple t on T such that $(x_i, a) \in t$ and $C(T)(t)$ = 0; x_i is GAC if all its values are GAC wrt. every cost function involving x_i; a problem is GAC* if every variable is GAC and NC*.

In the following we refer to NC* and GAC* as NC and GAC, without asterisk. GAC can be reached by shifting costs from the problem and deleting values not NC. Cost are shifted with *equivalent preserving transformations* in the following way: first projecting the minimum cost from non-unary cost functions to unary costs functions, and then projecting the minimum cost from unary cost functions into C_ϕ. After projection, node inconsistent values are deleted. When a value is deleted in x_i, GAC is rechecked on every variable that x_i is constrained with, so a deleted value might cause further deletions. The systematic application of these operations (projection and deletion of node inconsistent values) does not change the optimum (for details on projections and optimality, see [19]).

DCOP. A *Distributed Constraint Optimization Problem* (DCOP) [4] is defined by $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \alpha)$ where $\mathcal{X}, \mathcal{D}$ and $\mathcal{C}$ define a COP and: $\mathcal{A} = \{a_1, \ldots, a_p\}$ is a set of agents; $\alpha : \mathcal{X} \rightarrow \mathcal{A}$ maps each variable to one agent.

Agents communicate and coordinate while looking for the optimal solution through messages. It is assumed that: messages are never lost; messages are delivered in the same order they were sent; only one variable is mapped to each agent, so we use the terms variable and agent interchangeably.

In many cases, these problems are solved using a branch-and-bound schema, usually enhanced with sophisticated methods to improve lower bound computation (maintaining some forms of local consistency at each node). This facilitates pruning of the current branch and removal of future values, which improves performance.

BnB-ADOPT$^+$. BnB-ADOPT [21] is a reference algorithm for optimal DCOP solving. Agents are arranged in a depth-first search (DFS) pseudo-tree. A DFS pseudo-tree is an arrangement of the constraint graph with the following conditions: (1) There is a subset of edges, called tree-edges, that form a rooted tree covering all variables; the remaining edges are called back-edges. (2) Variables involved in the same cost function appear in the same branch of that tree. BnB-ADOPT asynchronously performs a depth-first-branch-and-bound search until an optimal solution is found. Agents may have a parent, children (connected by tree edges of the pseudo-tree), pseudo-parent and pseudo-children (connected by back-edges of the pseudo-tree) [4]. Each agent $self$ holds a $context$ that is updated with message exchange. The $context$ holds a set of assignments involving $self$ ancestors. Agents exchange the following messages:

- VALUE(i, j, val, th), –agent i informs child or pseudo-child j that it has taken value val with threshold[3] th–,
- COST$(k, j, context, lb, ub)$ –agent k informs parent j that with $context$ its bound are lb and ub–,
- TERMINATE(i, j), –agent i informs child j that agent i terminates–

A BnB-ADOPT agent executes the following loop: it reads and processes all incoming messages and assigns a value. Then, it sends a VALUE to each child or pseudo-child and a COST to its parent. When BnB-ADOPT terminates, each agent has assigned the optimum value for its variable. We use the BnB-ADOPT$^+$ version [22], which saves redundant messages. BnB-ADOPT$^+$ can also be generalized to handle non-binary constrains. It is required that each global cost function is evaluated by the last of its agents in the partial ordering of the pseudo-tree, while other agents have to send their values to the evaluator [4, 21]. For more details, see [21, 22].

4 Global Constraints in DisCSPs

As argued before, there are cases where it is really natural to use global constraints in distributed constraint reasoning. In that case, this raises the question of how to handle a global constraint in distributed reasoning.

4.1 Representing Global Constraints

We consider three different ways to model the inclusion of a global constraint in a DisCSP.

The first way to represent a global constraint, that we call the *direct representation*, is when the instance $C(T)$ of the global constraint C is posted in the DisCSP as a single constraint that allows all tuples on T satisfying C. In this representation, each agent $self$ in T simply includes $C(T)$ in its constraint set $\mathcal{C}_{self}$.

[3]The definition and usage of thresholds in BnB-ADOPT is complex and goes beyond this short summary. The interested reader may consult [21].

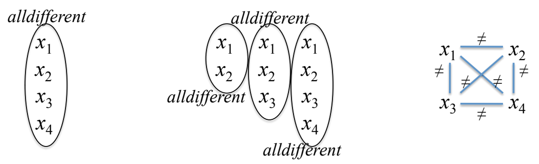


Figure 4: Representations considered for the global constraint *all-different*(x_1, x_2, x_3, x_4): (left) direct representation, (center) nested representation, (right) binary representation.

The second way we propose to represent a global constraint, called the *nested representation*, is applicable to all contractible global constraints. The nested representation of a global constraint $C(T)$ with $T = (x_{i_1}, \dots, x_{i_p})$ is the set of constraints $\{C(x_{i_1}, \dots, x_{i_j}) \mid j \in 2 \dots p\}$. For instance, the nested representation of *all-different* (x_1, x_2, x_3, x_4) is the set $S = \{$*all-different*(x_1, x_2), *all-different* (x_1, x_2, x_3), *all-different*$(x_1, x_2, x_3, x_4)\}$. Since *all-different* is contractible, the set of solutions of S is exactly the same as the set of solutions of the original constraint. The idea behind the nested representation is to use some knowledge about the semantics of the global constraint $C(T)$ to provide a model where the handling of the constraint can be more distributed. In this representation, each agent *self* in T adds all constraints of the nested representation of $C(T)$ that involve x_{self} to its constraint set $\mathcal{C}_{self}$.

The third way we propose to represent a global constraint, called the *binary representation*, is applicable to all global constraints that are binary decomposable. The binary representation of a global constraint $C(T)$ is the set of constraints of its binary decomposition. For instance, the binary representation of *all-different*(x_1, x_2, x_3, x_4) is the set $S = \{x_1 \neq x_2, x_1 \neq x_3, x_1 \neq x_4, x_2 \neq x_3, x_2 \neq x_4, x_3 \neq x_4\}$. Since *all-different* is binary decomposable, the set of solutions of S is exactly the same as the set of solutions of the original constraint. In this representation, each agent *self* in T includes all constraints of the binary decomposition of $C(T)$ that involve x_{self} in its constraint set $\mathcal{C}_{self}$. The three representations for the *all-different*(x_1, x_2, x_3, x_4) global constraint appear in Figure 4.

4.2 Searching with Global Constraints

In this section, we use ABT as the basic search algorithm for DisCSP solving. It is worth noting that the basic ABT algorithm, originally proposed for binary constraints, can be easily generalized to handle constraints of any arity. Let $C(x_i, x_j, x_k)$ be a ternary constraint. The last agent in the ABT ordering among the agents in the constraint is in charge of evaluating $C(x_i, x_j, x_k)$ when it is totally instantiated, while the others have to send their values to that evaluating agent [23]. In the following, we assume that this simple idea has been incorporated into ABT, so it can handle constraints of any arity. Therefore, we can run ABT on any of the three representations presented in Section 4.1.

In the direct representation, the instance $C(T)$ of the global constraint C is posted in the DisCSP as a single constraint. Thus, each agent *self* in T includes $C(T)$ in its constraint set $\mathcal{C}_{self}$. If agent *self* is the agent of T with lowest priority in the ABT order, it will be the one in charge of evaluating $C(T)$. In this case, links are put between the other agents in T and *self*.

In the nested representation, the global constraint $C(T)$, with $T = (x_{i_1}, \ldots, x_{i_p})$, is represented by the set of constraints $S = \{C(x_{i_1}, \ldots, x_{i_j}) \mid j \in 2 \ldots p\}$. Thus, each agent *self* in T includes all constraints of S that involve x_{self} in its constraint set $\mathcal{C}_{self}$. Thanks to these extra constraints that are posted in addition to $C(T)$, constraint handling can be done in a more distributed way. One could think that the order of variables in T must coincide with the ABT priority order. This is not needed to guarantee the correctness of the nested representation (although it is desirable for efficiency), as explained in the following. Suppose a contractible global constraint $C(x_3, x_4, x_2, x_1)$ in a DisCSP where the ABT agent ordering is, from first to last, x_1, x_2, x_3, x_4. If the order of the variables does not matter in C (for example, in the *all-different* constraint) we can reorder the scope as (x_1, x_2, x_3, x_4), following the ABT agent ordering. The nested representation will contain the constraints $C(x_1, x_2)$, $C(x_1, x_2, x_3)$, and $C(x_1, x_2, x_3, x_4)$, which is the best distribution for handling the constraint (evaluators of the new constraints are intermediate agents in T because any intermediate agent is the lowest priority agent of one of the constraints in the set S). If the order of the variables matters in C (for example, in the *increasing-by-one* constraint, which is satisfied iff each variable in C is one more than the previous variable in the scope of C), we cannot reorder the scope of C. The nested representation will then contain the constraints $C(x_3, x_4)$, $C(x_3, x_4, x_2)$, and $C(x_3, x_4, x_2, x_1)$, which will all be evaluated by the agent x_4 only, which is the lowest priority agent in each scope of the constraints of the nested representation. Observe that for the (few) contractible constraints in which the order of the variables in T matters, the fact that the ordering in T is different from the total ordering of the agents does not cause any problem: the nested representation remains correct and applicable in the same way. Previous discussion considers static variable ordering. In the case of ABT with dynamic variable ordering [24], the same arguments apply: At each time, one agent in the scope of each constraint will be the one with lowest priority in the current ABT agent ordering. That agent is in charge of evaluating that constraint. Changing dynamically the evaluator agent of each constraint may have impact in efficiency, but the nested representation remains correct. In addition, nogoods can be sent more directly to the real cause of failure.

In the binary representation, the global constraint $C(T)$ is represented by the set of constraints of its binary decomposition. Thus, each agent *self* in T includes all constraints of the binary decomposition of C that involve x_{self} in its constraint set $\mathcal{C}_{self}$.

These three ways of representing a global constraint are equivalent from the semantic point of view. So they produce the same solutions. However, they may cause different ABT executions.

Let us consider the *all-different*(x_1, x_2, x_3, x_4) of Figure 4 with the following domains $D(x_1) = \{a, c, d\}$, $D(x_2) = D(x_3) = \{a, b\}$, $D(x_4) = \{a, b, c, d\}$ and lets analyze how ABT runs on the three representations (since ABT may perform different executions, we follow one possible execution for each representation). ABT ordering is x_1, x_2, x_3, x_4 and values are chosen lexicographically. The ABT trace on the three representations appears in Figures 5-6.

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	$\mathbf{x_4}$
$x_1 \leftarrow a$ **ok?** to x_4	$x_2 \leftarrow a$ **ok?** to x_4	$x_3 \leftarrow a$ **ok?** to x_4	$x_4 \leftarrow a$
			ngd to x_3 $x_1 = a \wedge x_2 = a \Rightarrow x_3 \neq a$ $x_4 \leftarrow a$
		$x_3 \leftarrow b$ **ok?** to x_4 **addl** to x_1,x_2	
			ngd to x_3 $x_1 = a \wedge x_2 = a \Rightarrow x_3 \neq b$ $x_4 \leftarrow a$
		ngd to x_2 $x_1 = a \Rightarrow x_2 \neq a$ $x_3 \leftarrow a$ **ok?** to x_4	
	addl to x_1 $x_2 \leftarrow b$ **ok?** to x_3, x_4		**ngd** to x_3 $x_1 = a \wedge x_2 = a \Rightarrow x_3 \neq a$ $x_4 \leftarrow a$
		ngd obsolete **ok?** to x_4	
			ngd to x_3 $x_1 = a \wedge x_2 = b \Rightarrow x_3 \neq a$ $x_4 \leftarrow a$
		$x_3 \leftarrow b$ **ok?** to x_4	
			ngd to x_3 $x_1 = a \wedge x_2 = b \Rightarrow x_3 \neq b$ $x_4 \leftarrow a$
		ngd to x_2 $x_1 = a \Rightarrow x_2 \neq b$ $x_3 \leftarrow a$ **ok?** to x_4	
	ngd to x_1 $\Rightarrow x_1 \neq a$ $x_2 \leftarrow a$ **ok?** to x_3, x_4		**ngd** to x_3 $x_1 = a \wedge x_2 = b \Rightarrow x_3 \neq a$ $x_4 \leftarrow a$
$x_1 \leftarrow c$ **ok?** to x_2, x_3, x_4		**ngd** obsolete **ok?** to x_4	
			ngd to x_3 $x_1 = c \wedge x_2 = a \Rightarrow x_3 \neq a$ $x_4 \leftarrow a$
		$x_3 \leftarrow b$ **ok?** to x_4	
			$x_4 \leftarrow d$

Figure 5: Trace of ABT in the example, with three representations of *all-different*: direct.

Direct representation: Variables x_1, x_2, x_3 assign the first values in their domains, informing repeatedly to x_4, which backtracks repeatedly to x_3, this to x_2 and this to x_1, until value a is removed unconditionally from $D(x_1)$. Then, x_1 takes value c and informs x_4, x_2 and x_3. They all have value a. Then, x_4 will backtrack on x_3 which will change its value to b and will inform x_4, which will take value d. At this point, all constraints are satisfied, the network is quiescent, a solution (c, a, b, d) has been found.

Nested representation: Variables x_1, x_2, x_3 assign the first values in their domains. Variable x_1 informs to variables x_2, x_3 and x_4. Variable x_2 informs to variables x_3 and x_4. Variable x_3 informs to variable x_4. This causes that x_3 and x_4 start sending

$x_1 \leftarrow a$ **ok?** to x_2, x_3, x_4	$x_2 \leftarrow a$ **ok?** to x_3, x_4	$x_3 \leftarrow a$ **ok?** to x_4	$x_4 \leftarrow a$
	$x_2 \leftarrow b$ **ok?** to x_3, x_4	**ngd** to x_2 $x_1 = a \Rightarrow x_2 \neq a$ $x_3 \leftarrow a$ **ok?** to x_4	**ngd** to x_3 $x_1 = a \wedge x_2 = a \Rightarrow x_3 \neq a$ $x_4 \leftarrow a$
	ngd obsolete **ok?** to x_3	**ngd** obsolete **ok?** to x_4 **ngd** to x_2 $x_1 = a \Rightarrow x_2 \neq b$ $x_3 \leftarrow a$ **ok?** to x_4	**ngd** to x_3 $x_1 = a \wedge x_2 = b \Rightarrow x_3 \neq a$ $x_4 \leftarrow a$
	ngd to x_1 $\Rightarrow x_1 \neq a$ $x_2 \leftarrow a$ **ok?** to x_3, x_4	**ngd** to x_2 $x_1 = a \Rightarrow x_2 \neq b$ $x_3 \leftarrow a$ **ok?** to x_4	**ngd** to x_3 $x_1 = a \wedge x_2 = b \Rightarrow x_3 \neq a$ $x_4 \leftarrow a$
$x_1 \leftarrow c$ **ok?** to x_2, x_3, x_4	**ngd** obsolete **ok?** to x_3	**ngd** obsolete **ok?** to x_4	**ngd** to x_3 $x_1 = a \wedge x_2 = a \Rightarrow x_3 \neq a$ $x_4 \leftarrow a$
		ngd obsolete **ok?** to x_4 $x_3 \leftarrow b$ **ok?** to x_4	**ngd** to x_3 $x_1 = c \wedge x_2 = a \Rightarrow x_3 \neq a$ $x_4 \leftarrow a$
		ngd obsolete **ok?** to x_4	$x_4 \leftarrow d$

$x_1 \leftarrow a$ **ok?** to x_2, x_3, x_4	$x_2 \leftarrow a$ **ok?** to x_3, x_4	$x_3 \leftarrow a$ **ok?** to x_4	$x_4 \leftarrow a$
	$x_2 \leftarrow b$ **ok?** to x_3, x_4	$x_3 \leftarrow b$ **ok?** to x_4	$x_4 \leftarrow b$
		ngd to x_2 $x_1 = a \Rightarrow x_2 \neq b$ $x_3 \leftarrow b$ **ok?** to x_4	$x_4 \leftarrow c$
	ngd to x_1 $\Rightarrow x_1 \neq a$ $x_2 \leftarrow a$ **ok?** to x_3, x_4		
$x_1 \leftarrow c$ **ok?** to x_2, x_3, x_4			
			$x_4 \leftarrow d$

Figure 6: Trace of ABT in the example, with three representations of *all-different*: nested (top), binary (bottom).

backtracking messages to x_2 and x_3 respectively. At some point, x_2 backtracks to x_1 removing unconditionally value a. Then, x_1 takes value c, x_2 and x_3 had already value a, x_3 changes to b and x_4 takes value d. The network is quiescent, the same solution (c, a, b, d) has been found. Differences with the direct representation come from the number of constraints (1 in direct, 3 in nested) which determine the number of backtracking operations that could be performed simultaneously. In this sense, the nested representation does more work in parallel than the direct one.

Binary representation: All variables assign value a and inform their lower priority agents in the binary constraints. Variables x_2 and x_3 change to b, and x_4 changes to c. There is a backtrack message from x_3 to x_2, which takes value a, and another backtrack from x_2 to x_1, which now takes value c. At this point x_4 takes value d. Again, the network is quiescent, the same solution (c, a, b, d) has been found. Differences with the direct and nested representation come from the number of constraints (1 in direct, 3 in nested, 6 in binary). This indicates the number of concurrent backtracks performed and also the number of variables which could adjust its value (the last variable in each constraint). In addition, the binary representation allows to backtrack to the real culprit of the inconsistency (to change x_1 value from a to c binary needs 2 backtrack only, while nested needs 7 and direct needs 10).

4.3 Propagating Global Constraints

Independently of the way a global constraint is included into ABT, this algorithm can be enhanced by maintaining some form of local consistency during search. This was already investigated in [10], where limited/full forms of arc consistency (AC) were maintained during ABT execution for binary DisCSPs. While in [10] a limited form of AC causing unconditional deletions and full AC causing conditional deletions were considered, in this paper we maintain a limited form of GAC that causes unconditional deletions only. Clearly, this limited GAC, that from now on we call UGAC, is less powerful than full GAC. However, maintaining full GAC in the distributed context would cause a substantial load of extra communication which could overcome the benefits of domain pruning.

The basic idea is as follows: if during ABT execution agent $self$ receives a NGD message justifying the removal of value v with a nogood with an empty left-hand side (see [2, 16, 10] for details), v can be unconditionally deleted from $D(x_{self})$. A deletion on $D(x_{self})$ is propagated maintaining UGAC on the constraints connecting x_{self} to other variables, which may cause further deletions. Since the initial deletion is unconditional, deletions caused by the propagation are also unconditional.

To maintain UGAC during ABT search, a number of modifications is needed over the basic ABT algorithm. They are:

- The domain of variables constrained with $self$ by constraints in $\mathcal{C}_{self}$ has to be represented in $self$.
- Only the agent owner of a variable can modify its domain (i.e., only $self$ can modify $D(x_{self})$). If agent i deduces that a value has to be deleted in $D(x_j)$, it does nothing because that deduction will be done in j at some point.
- There is a new type of message, DEL, to notify of value deletions. DEL$(self, k, v)$ –informing that $self$ removes v from $D(x_{self})$ – is sent from $self$ to every agent k constrained with it.
- All constraints are made GAC before ABT starts, by a suitable preprocess. Its pseudocode appears in Figure 7.

These changes do not modify ABT correctness and completeness. Regarding correctness, the only UGAC action is to remove values that will not be in any solution, so correctness is maintained. Regarding completeness, we consider two cases: (i) removing a value that is not currently assigned, and (ii) removing a value that is currently assigned. Case (i) trivially maintains completeness. Case (ii) also maintains completeness, because the asynchronous ABT allows agents to change their value at any time.

```
procedure GAC-preprocess()  /* quiescence: all constraints are GAC */
  for each $C(T) \in \mathcal{C}_{self}$ do GAC($self, C(T)$);
  while ($\neg end$) do  /*if end is true, empty domain detected */
    $msg \leftarrow$ getMsg();
    switch($msg.type$)
      $Del$: ValueDeletedPre($msg.sender, msg.value$);
      $Stop$: $end \leftarrow$ true;

procedure ValueDeletedPre($j, a$)  /* $j$ has deleted value $a$*/
  $D(x_j) \leftarrow D(x_j) - \{a\}$;
  for each $C(T) \in \mathcal{C}_{self}$ s.t. $x_j \in T$ do GAC($self, C(T)$);

procedure GAC($self, C(T)$)
  if revise($x_{self}, C(T)$) then
    if $D(x_{self}) = \emptyset$ then sendMsg:$Stop(system)$;
    else $Delval \leftarrow$ set of deleted values in $D(x_{self})$ by revise($x_{self}, C(T)$)
      for each $v \in Delval$ and $x_k \in \bigcup_{T|C(T)\in\mathcal{C}_{self}}, k \neq self$ do
        sendMsg:$Del(self, k, v)$;
```

Figure 7: GAC preprocess

Differences between ABT-UGAC and ABT are (see Figures 8-9):

- `ABT-UGAC`. It uses the DEL message, which notifies that a value has been deleted in some domain. If *self* receives that message, it calls the `ValueDeleted` procedure.
- `Conflict`. If *self* accepts a NGD message containing a nogood with empty left-hand side, it calls the `DeleteValue` procedure.
- `ValueDeleted`(j, a). Agent j has deleted value a from $D(x_j)$ and sent a DEL message to *self*. Agent *self* registers this in its $D(x_j)$ copy, and enforces AC on all the constraints including *self* and x_j in their scope. If the value of *self* is deleted in this process, the `CheckAgentView` procedure is called (looking for a new compatible value; if none exists it backtracks). Deletions in $D(x_{self})$ are propagated.
- `DeleteValue`(a, j). Agent *self* must delete its currently assigned value a because a nogood with empty left-hand side has been received from agent j. Value a is deleted from $D(x_{self})$. If, as a consequence of a's deletion, $D(x_{self})$ becomes empty, there is no solution so a STOP message is produced. Otherwise, a's deletion is notified to all agents constrained with *self* except j via DEL messages, and the procedure `CheckAgentView` is called.
- `Backtrack`. After *self* computes and sends a new nogood, it checks if its left-hand side is empty. If so, *self* knows that the value that forbids the new nogood will be removed in the domain of the variable that appears in the right-hand side of the new nogood. Then *self* calls `ValueDeleted`, as if it had received a DEL message.

```
    procedure ABT-UGAC()  /* ABT with unconditional GAC */
        $\Gamma = \Gamma_0^- \cup \Gamma_0^+; \Gamma^- = \Gamma_0^-; \Gamma^+ = \Gamma_0^+;$
        $myValue \leftarrow$ empty; $end \leftarrow$ false; CheckAgentView();
        while ($\neg end$) do
          $msg \leftarrow$ getMsg();
          switch($msg.type$)
          $Ok?$:ProcessInfo($msg$); $AddL$:SetLink($msg$);
          $Ngd$:Conflict($msg$); $Stop$: $end \leftarrow$ true;
new       $Del$: ValueDeleted($msg.sender, msg.value$);

    procedure Conflict($msg$)
        if Coherent($msg.nogood, \Gamma^- \cup \{self\}$) then
new       if lhs($msg.nogood$) = empty then
new         DeleteValue($myValue, msg.sender$);
          else
            CheckAddLink($msg$);
            add($msg.nogood, myNogoodStore$);
            $myValue \leftarrow$ empty;
            CheckAgentView();
        else if Coherent($msg.nogood, self$) then
          sendMsg:$Ok?$($msg.sender, myValue$);
```

Figure 8: The ABT-UGAC algorithm (part 1); only procedures with new lines (*new*) with respect to ABT are shown. Γ^- and Γ^+ are higher and lower priority agents connected to *self*; Γ_0^- and Γ_0^+ are those sets at the beginning of execution.

The difference that a global constraint instance $C(T)$ makes with respect to any other generic constraint is when executing the GAC procedure. Since $C(T)$ has a well-defined semantics, it is often possible to generate specific propagators able to achieve the required level of local consistency polynomially whereas the generic propagator based on table lookups is exponential in the size of T. In particular, we have implemented the popular flow-based propagator for the *all-different* constraint, following [25], which reduces propagation to matching theory.

Considering the running example of Figure 4 with the domains $D(x_1) = \{a, c, d\}$, $D(x_2) = D(x_3) = \{a, b\}$, $D(x_4) = \{a, b, c, d\}$, after applying GAC-preprocess on the direct or nested representations the following values are deleted: $(x_1, a)(x_4, a)(x_4, b)$. However, this preprocess does not filter any value in the binary representation. From this on, ABT-UGAC on the direct and nested representations has a relatively similar performance: the first value of x_1 and x_2 are part of the solution, a backtracking from x_4 causes x_3 to change to value b (direct) while x_3 is able to adjust its value to b (nested). ABT-UGAC on the binary representation does the same execution as described in Section 4.2.

4.4 Experimental Results

To evaluate the impact of global constraints in DisCSPs, we compare ABT with and without UGAC on random DisCSPs created as follows. We first generate random binary instances using model B [26], and then add some global constraints. In model B,

```
new  procedure ValueDeleted($j, a$)   /* $j$ has deleted $a$ from $D(x_j)$ */
new  $D(x_j) \leftarrow D(x_j) - \{a\}$;
new  for each $C(T) \in \mathcal{C}_{self}$ s.t. $j \in T$ do GAC($x_{self}, C(T)$);
new  if $myValue \notin D(x_{self})$ then
new    $myValue \leftarrow$ empty; CheckAgentView();

new  procedure DeleteValue($a, j$)   /* self deletes $a$ from $D(x_{self})$ */
new  $D(x_{self}) \leftarrow D(x_{self}) - \{a\}$;
new  if $D(x_{self}) = \emptyset$ then sendMsg:$Stop(system)$;
new  else
new    for each $k \in \Gamma_0, k \neq j$ do sendMsg:$Del(self, k, a)$;
new    CheckAgentView();

procedure Backtrack()
     $newNogood \leftarrow$ solve($myNogoodStore$);
     if ($newNogood =$ empty) then $end \leftarrow$ true; sendMsg:$Stop(system)$;
     else
       sendMsg:$Ngd(newNogood)$;
       Update($myAgentView$,rhs($newNogood$) $\leftarrow$ unknown);
new  if lhs($newNogood$) $=$ empty then
new    ValueDeleted(rhs($newNogood$));
     else CheckAgentView();
```

Figure 9: The ABT-UGAC algorithm (part 2); only procedures with new lines (*new*) with respect to ABT are shown. Γ^- and Γ^+ are higher and lower priority agents connected to *self*; Γ_0^- and Γ_0^+ are those sets at the beginning of execution.

a random binary CSP class is characterized by $\langle n, d, p_1, p_2 \rangle$, where n is the number of variables, d is the domain size of each variable, p_1 is the problem connectivity defined as the ratio of existing constraints and p_2 is the constraint tightness expressed as the ratio of forbidden value pairs. Hence, every instance contains $p_1 \cdot n(n-1)/2$ binary constraints and each of them has $p_2 \cdot d^2$ forbidden value pairs. From a binary instance, we can generate two types of benchmarks: the *all-different* benchmark and the *atmost* benchmark. In the *all-different* benchmark, each binary instance includes 2 *all-different* constraints, each involving 5 randomly chosen variables. We also performed this experiment with 10 —instead of 2— *all-different* constraints per instance, obtaining similar results. Direct, nested and binary representations are used with these *all-different*. In the *atmost* benchmark, each binary instance includes 10 *atmost[k,v]* constraints, each involving from 3 to 10 randomly chosen variables. The value v whose occurrences are bounded is also randomly chosen in the set of values occurring in domains and its maximum number of occurrences k is 1 or 2. Only direct and nested representations are used on these *atmost* constraints because *atmost* is not binary decomposable. In both benchmarks, since the order in which variables appear in the global constraint does not matter, we assumed the ABT priority order.

For the *all-different* benchmark, we have performed two experiments: sparse $\langle 20, 5, 0.2, p_2 \rangle$ and dense $\langle 20, 5, 0.7, p_2 \rangle$, where p_2 varies between 0.1 and 0.9 in steps of 0.1. For each experiment, we evaluate performance as the number of messages exchanged and the number of non-concurrent constraint checks (NCCCs) [27], considering the three representations. UGAC enforcing uses generic table lookups when testing binary con-

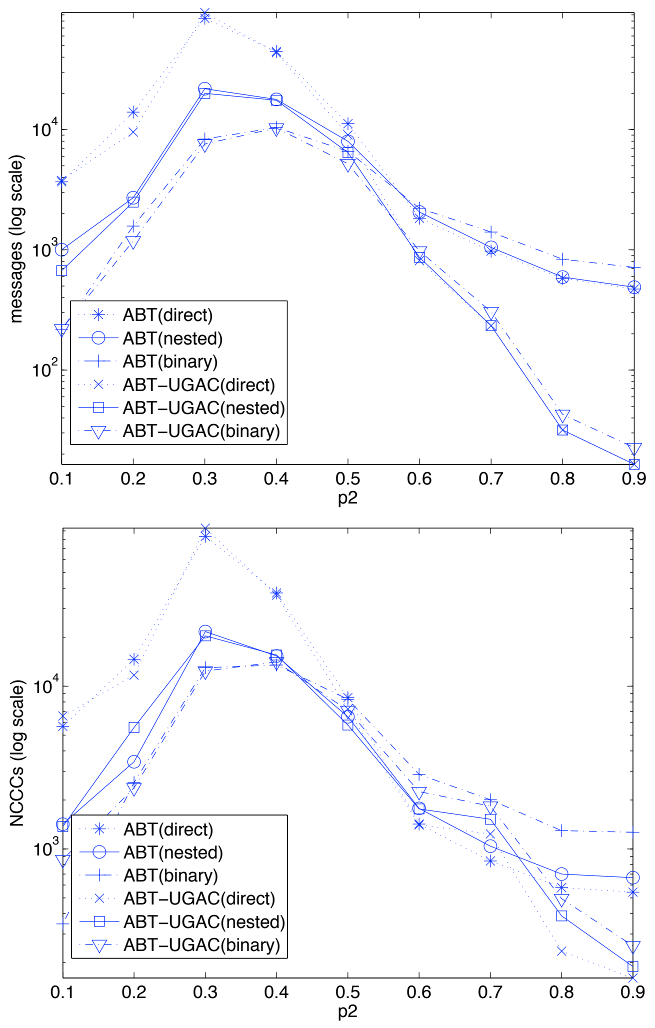
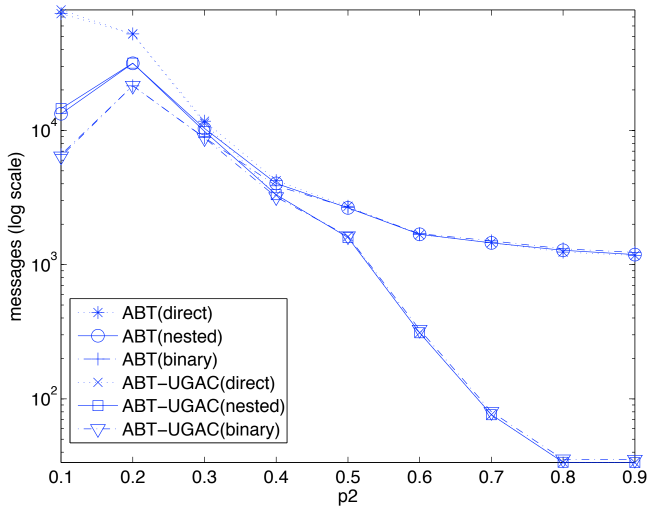


Figure 10: Results in #messages and NCCCs for the *all-different* benchmark described in the text. $p_1 = 0.2$. Observe that for large values of p_2 some lines superpose. Top plot: ABT(direct) with ABT(nested), ABT-UGAC(direct) with ABT-UGAC(nested).

straints and it executes a special propagator when testing the global *all-different* constraints (as described in [25]). Each time this propagator is called it computes a maximum matching in a graph; then, the NCCC counter increases in the number of nodes of that graph.

Results in number of exchanged messages and NCCCs appear in Figures 10-11, averaged on 100 instances per each p_2. Since sparse ($p_1 = 0.2$) and dense ($p_1 = 0.7$) instances show similar results, we discuss them jointly. Considering #messages of sparse



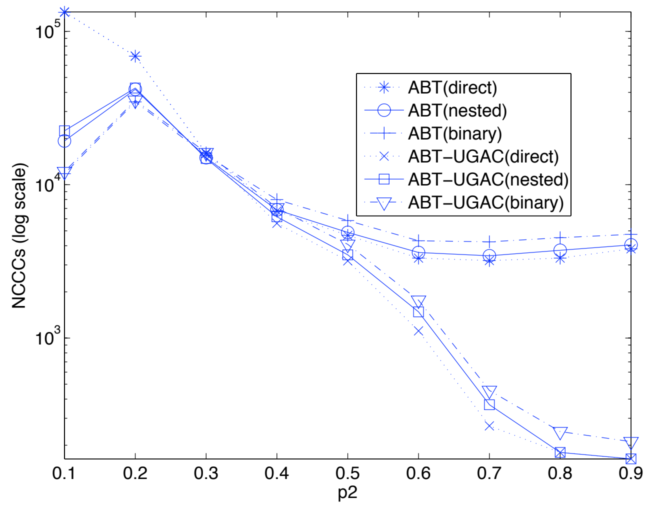
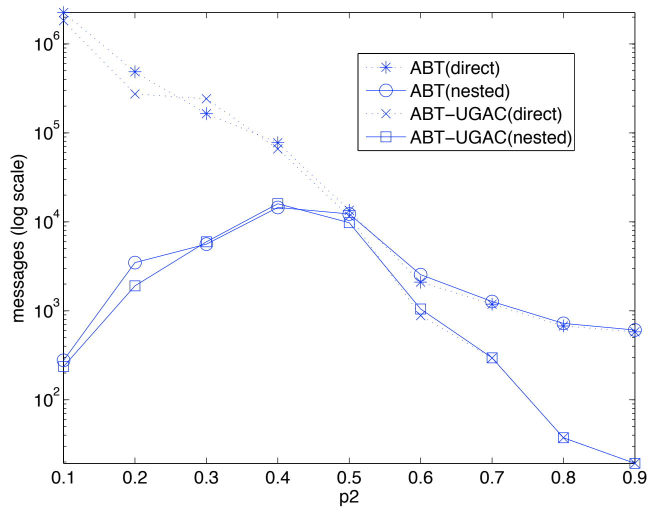


Figure 11: Results in #messages and NCCCs for the *all-different* benchmark described in the text. $p_1 = 0.7$. Observe that for large values of p_2 some lines superpose. ABT(direct) with ABT(nested) with ABT(binary), ABT-UGAC(direct) with ABT-UGAC(nested) with ABT-UGAC(binary).

instances and $p_2 < 0.5$ (dense instances and $p_2 < 0.3$, resp.) we observe that the curve of plain ABT with a particular global constraint representation follows closely the curve of ABT-UGAC with the same representation. This shows that maintaining UGAC when constraints are loose does not pay off and it is the type of representation that makes the difference in efficiency. The most efficient representation is binary, followed by nested and finally direct representation. The direct representation causes in-



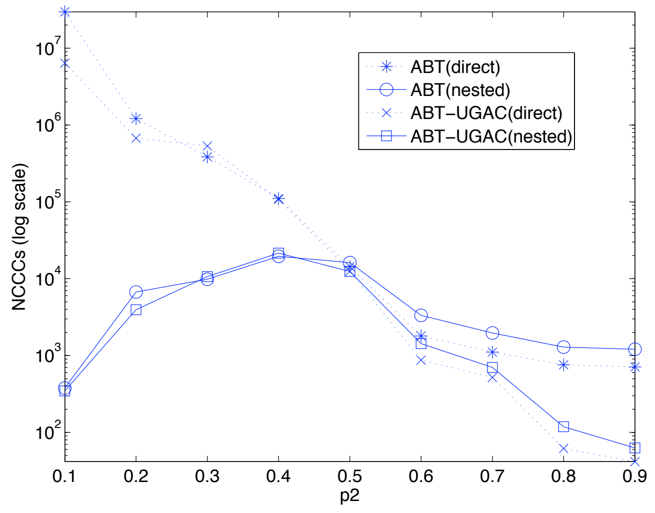
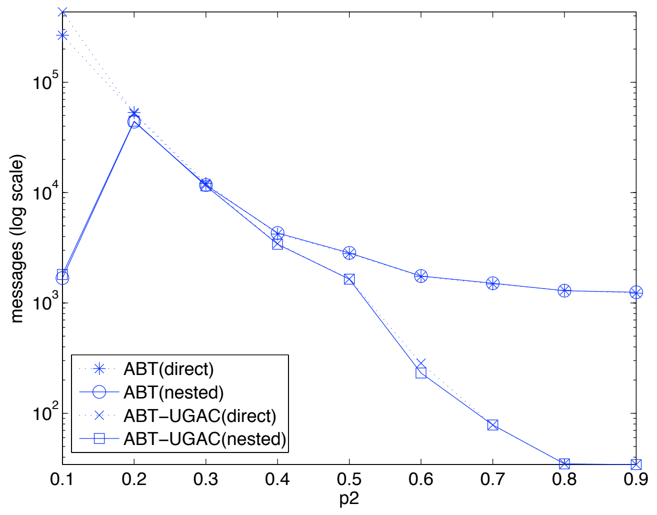


Figure 12: Results in #messages and NCCCs for the *atmost* benchmark described in the text. $p_1 = 0.2$. Observe that for large values of p_2 some lines superpose. ABT(direct) with ABT(nested), ABT-UGAC(direct) with ABT-UGAC(nested).

efficient chronological backtracking (which causes many useless messages —see ABT trace in Figure 5), and nested representation implies sending extra OK? messages. In this setting where not much pruning occurs, the binary decomposition appears as the most efficient, because although it sends many OK? messages, it performs backtracking directly to the culprit.

For sparse instances and $p_2 > 0.5$ (dense instances and $p_2 > 0.3$, resp.) the situation changes and ABT curves are grouped according to UGAC enforcement: maintain-



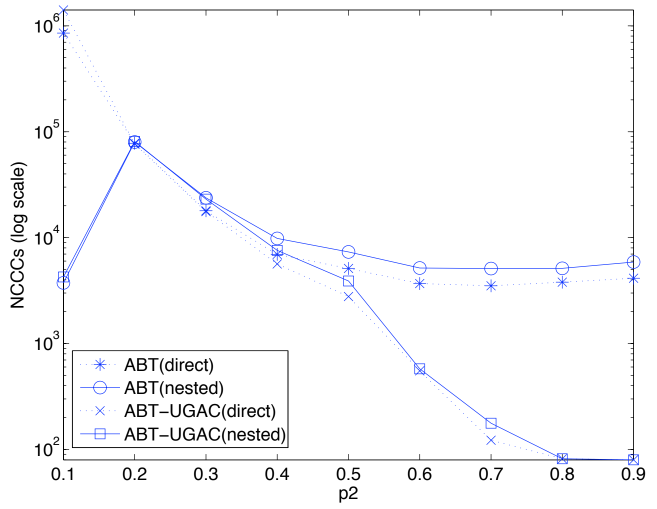


Figure 13: Results in #messages and NCCCs for the *atmost* benchmark described in the text. $p_1 = 0.7$. Observe that for large values of p_2 some lines superpose. ABT(direct) with ABT(nested), ABT-UGAC(direct) with ABT-UGAC(nested).

ing UGAC is now the most discriminant element. The analysis of this fact is simple: for medium to high tightnesses, UGAC maintenance really decreases the size of the search space, so algorithms including UGAC terminate faster and thus require less messages (much less, observe the logarithmic scale) than plain ABT. On the relative performance of the three representations for global constraints, the less efficient representation is the binary, clearly dominated by the direct and nested ones, which practically use the same number of messages. We explain this as the combined effect of two facts: a high number of constraints (a single *all-different* becomes a quadratic number of constraints in binary; a linear number of constraints in nested; one constraint in direct) and the fact that the problem has no solution. In the absence of solutions, ABT necessarily generates nogoods to prove inconsistency and agents in binary will tend to send a higher number of NGD messages because they belong to more constraints than in the other representations. Some of these NGDs are useless and become obsolete.

Regarding NCCC, the observed pattern is similar to that observed for the number of messages. Differences among representations are due to the different number of constraints existing in each representation.

Results for the *atmost* benchmark appear in Figures 12-13. We experimented with it because *atmost* is an example of non binary decomposable constraint. This explains why only the direct and nested representations are reported. Evaluation was similar to the one done on the *all-different* benchmark: two experiments: sparse $\langle 20, 5, 0.2, p_2 \rangle$ and dense $\langle 20, 5, 0.7, p_2 \rangle$, where p_2 varies between 0.1 and 0.9 in steps of 0.1, counting the number of messages exchanged and the number of non-concurrent constraint checks (NCCC) on the available representations. Results are averaged on 100 instances per p_2.

The number of exchanged messages shows a similar pattern to the one observed in the *all-different* benchmark. For sparse instances and $p_2 < 0.5$ (dense instances and $p_2 < 0.3$, resp.) we observe that UGAC has no much effect and the dominant factor is the representation of the global constraint: as in the *all-different* benchmark, the direct representation is much worse than the nested one. For sparse instances and $p_2 > 0.5$ (dense instances and $p_2 > 0.3$, resp.) the situation changes. As in the case of the *all-different* benchmark, maintaining UGAC becomes the main reason for efficiency, more important than the type of representation of the global constraints. Thus, ABT curves are closely related, while ABT-UGAC curves are also closely joined. The explanation of this fact is the same as in the *all-different* benchmark: For medium to high tightness, UGAC maintenance really decreases the size of the search space, so algorithms including UGAC terminate faster and require much less messages to explore that space than plain ABT.

Regarding NCCC, they show a pattern similar to the number of messages. Differences among representations are due to the different number of constraints existing in each representation.

5 Global Cost Functions in DCOP

In this section we consider the introduction of global cost functions in distributed constraint optimization problems. The section is organized exactly as Section 4.

5.1 Representing Global Cost Functions

We consider three different ways to model the inclusion of a global cost function in DCOPs. The user chooses one of the three representations and the solving is done on that representation.

The first way is the direct representation. An instance $C(T)$ of a global cost function C is posted as a single cost function. Only the last agent (among the agents in T) in the branch of the pseudo-tree evaluates the cost function.

In the DisCSP case, we had the nested representation when C was contractible. Contractibility is a notion also defined in the optimization case. Unfortunately, the technique of the nested representation used for DisCSPs no longer works. When we create copies of the global cost function $C(T)$ on subsets of the variables in T, costs are duplicated and the model remains no loger equivalent. Take for instance the global cost function *soft-all-different*(x_1, x_2, x_3, x_4) and its nested representation $S = \{$*soft-all-different*$(x_1, x_2),$ *soft-all-different*$(x_1, x_2, x_3),$ *soft-all-different*$(x_1, x_2, x_3, x_4)\}$. Consider the assignment $x_1 = a, x_2 = a, x_3 = b, x_4 = c$. The cost of assigning $x_1 = a, x_2 = a$ will be counted in all three cost functions in the nested representation, leading to a higher cost than that in the original *soft-all-different*(x_1, x_2, x_3, x_4) global cost function, thus losing equivalence. However, we still have the interesting property that each cost function in S taken separately produces costs that never exceed the cost of the original global cost function. This property can be used to still use the nested representation in DCOPs provided that we slightly modify the algorithms so that when an agent receives costs from several cost functions from the same nested representation, it does not sum them but takes their maximum.

Finally, the binary representation works the same as in DisCSPs. If C is binary decomposable, instead of $C(T)$, the binary decomposition of $C(T)$ is included in the problem.

5.2 Searching with Global Cost Functions

We have extended the distributed search algorithm BnB-ADOPT$^+$ [22] to support global cost functions with a direct, nested and binary representations. Some modifications of the original algorithm are needed, explained below.

In the case of the binary representation, there is no need for any modification: BnB-ADOPT$^+$ works as usual since the global cost function has been expressed as a set of binary cost functions. For the nested and direct representations, we apply the following modifications:

1. Every agent *self* keeps a global cost function set with all the global cost functions in which *self* is involved. Every global cost function $C(T)$ implicitly contains the agents involved in T (neighbors of *self*). For some cost functions, additional information can be stored. For example, for the *soft-at-most[k,v]* cost function, parameters k (number of repetitions) and v (domain value) are stored.

2. During the search process, every time *self* needs to evaluate the cost of a given value v, all local costs are aggregated. Binary cost functions are evaluated as usual, and global cost functions are evaluated according to its violation measure.

3. VALUE messages are sent to children and pseudo-children involved in any kind —binary or global— of cost functions. In the case of direct representation, VALUE messages are sent only to the deepest agent in the DFS tree involved in the global cost function (no VALUES are sent to intermediate agents because

```
(1) procedure CalculateCost($value$)
(2)   $cost = cost +$ BinaryCostWithValue($value$);
(3)   $cost = cost +$ GlobalCostWithValue($value$);
(4)   return $cost$;

(5) function BinaryCostWithValue($value$)
(6)   for each $(x_i, d_i) \in context$ do
(7)     $binaryCost = binaryCost + C(x_i, self)(d_i, value)$;
(8)   return $binaryCost$;

(9) function GlobalCostWithValue($value$)
(10)  cost =0;
(11)  for each $global \in globalCtrSet$ do
(12)    for each $(x_i, d_i) \in context$ do
(13)      if $x_i \in global.vars$ then $globalContext.add(x_i, d_i)$;
(14)    if $globalContext.size == global.vars.size$ then //self last evaluator
(15)      $cost = cost +$ EvalGlobal($global, globalContext \cup (self, value)$);
(16)    else //self is an intermediate agent in the restriction
(17)      if $DIRECT$ representation then $cost = cost + 0$;
(18)      if $NESTED$ representation then
(19)        for each $x_i \in global.vars$ do
(20)          if $lowerGlobalEvals.contain(x_i)$ then $cost = cost + 0$;
(21)          else $cost = cost +$ EvalGlobal($globalContext \cup (self, value)$);
(22)  return cost;

(23)  function EvalGlobal($global, varAssignments$)
(24)    return $global.\mu(varAssignments)$;
```

Figure 14: Aggregating costs of binary and global cost functions.

they are not able to evaluate the global cost function). In the case of nested representation, VALUE messages are sent to all children and pseudo-children involved in the global cost function, since they are all able to evaluate it.

4. COST messages include a list of all the agents that have evaluated a global cost function. This is done to prevent duplication of costs and is explained in the next paragraphs.

Figure 14 shows the pseudocode for cost aggregation of binary and global cost functions in BnB-ADOPT$^+$ (lines 1-4). Binary costs are calculated as usual, aggregating all binary costs evaluated on *self* value and the assignments of the current context (lines 5-8). Global costs are calculated in lines 9-20. Although there is no need to separate binary cost aggregation from global cost aggregation, we have presented them in separate procedures for a better understanding of the new modifications.

For every global cost function of the global cost function set (*globalCtrSet*) *self* creates a tuple with their assignment in the current context (*globalContext*, lines 11-13). If *self* is the deepest agent in the DFS tree (taking into account the variables involved in the global cost function) then *self* evaluates the cost function (lines 14-15). If *self* is an intermediate agent, it does the following. If representation is direct, *self* cannot evaluate the global cost function and cost is not incremented (line 17). If representation is nested, it requires some care, as we said when presenting the nested representation. A nested global cost function is evaluated more than once by intermediate agents and if these costs are simply aggregated duplication of costs may occur. To prevent this, COST messages include the set of agents that have evaluated their global cost functions (*lowerGlobalEvals*). When a COST message arrives, $self$ knows which agents have evaluated their cost functions and contributed to the lower bound. If some of them appear in the scope of C, then $self$ does not evaluate this cost function (lines 18-21). By doing this, the deepest agent in the DFS tree evaluating the global cost function precludes any other agent in the same branch to evaluate the cost function, avoiding cost duplication. Preference is given to the deepest agent because is the one that receives more value assignments and can perform a more informed evaluation. When bounds coming from a branch of the DFS are reinitialized (this happens under certain conditions in BnB-ADOPT, for details see [21]), the agents in the set *lowerGlobalEvals* lying on that branch are removed.

Finally, a procedure for evaluating the global cost functions according to its violation measure is presented in lines 23-24.

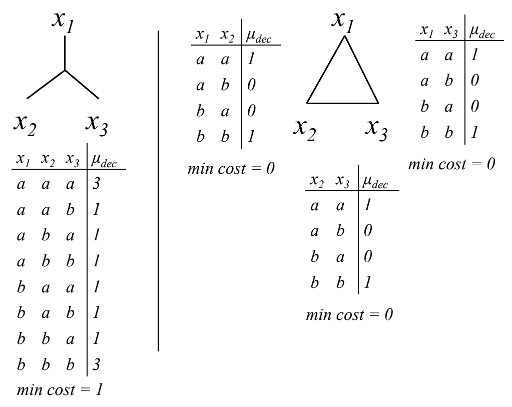
5.3 Propagating Global Cost Functions

When introducing soft local consistency, the solving process is improved. The quality of the bounds obtained as the result of applying local consistency is often better when the problem contains global constraints than when it contains an equivalent binary formulation. For example, consider the case of the *soft-all-different(*x_1, x_2, x_3*)* with violation measure μ_{dec} and its equivalent binary formulation: $\{$*soft-all-different(*x_1, x_2*)*, *soft-all-different(*x_2, x_3*)*, *soft-all-different(*x_1, x_3*)*$\}$ with the domain set $\{a, b\}$ for every variable (Figure 15). If UGAC is applied on the global formulation it can be inferred a lower bound of 1 for the optimal solution. Since there are three variables and only two domain values, any ternary tuple (with a combination of x_1, x_2, x_3 values) will cost at least 1 (Figure 15, left). However in the binary formulation we can only infer a lower bound of 0, if looking independently at every binary tuple (Figure 15, right).

Specific propagators exploiting the semantics of global cost functions has been proposed in the centralized case. These propagators allow to achieve the generalized arc consistency level more efficiently than using generic propagators. In both cases —binary or global cost functions— soft local consistency is based on *equivalent preserving transformations* where costs are shifted from binary/global cost functions to unary cost functions.

These same technique can be applied in distributed COP. We project costs from binary and global cost functions to unary cost functions and finally project unary costs to C_ϕ. After binary/global projections are made, agents check their domains searching for inconsistent values. For this, some modifications are needed:

- The domain of neighboring agents (agents connected with $self$ by cost functions) are represented in $self$.
- A new DEL message is added to notify value deletions.
- COSTs and VALUEs contain extra information.



x_1	x_2	x_3	μ_{dec}
a	*a*	*a*	3
a	*a*	*b*	1
a	*b*	*a*	1
a	*b*	*b*	1
b	*a*	*a*	1
b	*a*	*b*	1
b	*b*	*a*	1
b	*b*	*b*	3

min cost = 1

x_1	x_2	μ_{dec}
a	*a*	1
a	*b*	0
b	*a*	0
b	*b*	1

min cost = 0

x_2	x_3	μ_{dec}
a	*a*	1
a	*b*	0
b	*a*	0
b	*b*	1

min cost = 0

x_1	x_3	μ_{dec}
a	*a*	1
a	*b*	0
b	*a*	0
b	*b*	1

min cost = 0

Figure 15: (Left) *soft-all-different*(x_1, x_2, x_3) with μ_{dec} violation measure; (right) its binary decomposition

Following the technique proposed in [11], we maintain the GAC consistency level during search, but performing only unconditional deletions, so we call it unconditional generalized arc consistency (UGAC). An agent *self* deletes a value v *unconditionally* if this value is guaranteed to be sub-optimal and does not need to be restored again during the search process. If *self* contains a value v not NC ($C(self)(v) + C_\phi > \top$) then v can be deleted unconditionally because the cost of a solution containing the assignment $self = v$ is necessarily greater than $\top$. We also detect unconditional deletions in the following way. Let us consider agent *self* executing BnB-ADOPT$^+$. Suppose *self* assigns value v and sends the corresponding VALUE messages to children and pseudo-children. As response, COST messages arrive. We consider those COST messages whose context is simply $(self, v)$. This means that the bounds informed in these COST messages only depend on *self* assignment (observe that the *root* agent always receives such COST messages). If the sum of the lower bounds contained in those COST messages exceeds $\top$, v can be deleted unconditionally because the cost of a solution containing $self = v$ is necessarily greater than $\top$.

As in [11], messages include information required to perform deletions, namely $\top$

BnB-ADOPT$^+$-UGAC messages:
VALUE($sender, destination, value, threshold, \underline{\top, C_\phi}$)
COST($sender, destination, context[], lb, ub, \underline{subtreeContr}$
$\underline{agents\ contributing\ lb}$)
TERMINATE($sender, destination, \underline{emptydomain}$)
$\underline{\text{DEL}(sender, destination, value)}$

Figure 16: Messages of BnB-ADOPT$^+$-UGAC. New parts wrt. BnB-ADOPT$^+$ are underlined.

```
(1) procedure ProjectFromAllDiffToUnary($global, v$)
(2) $graph = alldiffGraphs.get(global)$;
(3) $minCost = minCost + graph.getMinCostFlow()$;
(4) for each $x_i \in global.vars$ do
(5)   $minCost = minCost+$
      $graph.residualGraph.shortestPathWith(x_i, v)$;
(6)   if $minCost > 0$ then
(7)     $graph.getArc(x_i, v).cost = cost - minCost$;
(8)     if $x_i = self$ then $C(self)(v) = C(self)(v) + minCost$;
```

Figure 17: Projection with *soft-all-different* global cost function.

(the lowest unacceptable cost), C_ϕ (the minimum cost of any complete assignment), and the subtree contribution to C_ϕ (each node k computes the contribution to the C_ϕ of the subtree rooted at k). These three elements travel in existing BnB-ADOPT$^+$ messages (the two first in VALUE messages, the last in COST messages). In addition, a new message DEL$(self, k, v)$ is added, to notify agent k that $self$ deletes value v. The structure of these new messages appears in Figure 16. When $self$ receives a VALUE message, $self$ updates its local copies of $\top$ and C_ϕ if the values contained in the received message are better (lower $\top$ or higher C_ϕ). When $self$ receives a COST message from a child c, $self$ records c subtree contribution to C_ϕ. When $self$ receives a DEL message, $self$ removes the deleted value from its domain copy of the sender agent and performs projections from the cost functions involving the sender agent to its unary costs and to C_ϕ. When $\top$ or C_ϕ change, $D(x_{self})$ is tested for possible deletions.

This mechanism described to detect and propagate unconditional deletions is similar to the one proposed in [11], avoiding simultaneous deletions [12]. However, to reach the UGAC level, agents need to project costs not only from binary cost functions, but from global cost functions as well. In the following, we describe how to project binary and global costs specifically from the *soft-all-different* and *soft-at-most* global cost functions.

```
(1)  procedure ProjectFromAtMostToUnary($global, v$)
(2)  if $global.k = v$ and $D(x_{self}).contains(v)$ then
(3)    for each $x_i \in global.vars$ do
(4)      if $D(x_i).contains(v)$ and $D(x_i).size() = 1$ then
(5)        $singletonCounter = singletonCounter + 1$;
(6)    if $singletonCounter > global.n$ then
(7)      $cost = singletonCounter - global.n$;
(8)      if $cost > global.minCost$ then
(9)        $cost = temp$;
(10)       $cost = cost - global.minCost$;
(11)       $global.minCost = temp$;
(12)       if $global.vars[0] = self$ then
(13)         $C(self)(v) = C(self)(v) + minCost$;
```

Figure 18: Projection with *soft-all-atmost[k,v]* global cost function.

Projecting Costs with Binary Cost Functions

As in [19], we project costs from binary cost functions to unary in the following way. The projection of costs from the binary cost function $C(x_i, x_j)$ to the unary cost function $C(x_i)$ is a flow of costs defined as follows. Let α_a be the minimum cost of value a in x_i with respect to $C(x_i, x_j)$ (namely $\alpha_a = min_{b \in D(x_j)} C_{x_i,x_j}(a, b)$). The projection consists in adding α_a to $C_{x_i}(a)$ (namely, $C_{x_i}(a) = C_{x_i}(a) + \alpha_a$) and subtracting α_a from $C_{x_i,x_j}(a, b)$ (namely, $C_{x_i,x_j}(a, b) = C_{x_i,x_j}(a, b) - \alpha_a, \forall b \in D(x_j)$). This process is done for all values a in $D(x_i)$.

Every pair of agents x_i and x_j sharing a binary cost function perform projections following a fixed order (projections are done first over the higher agent in the DFS and after over the lower agent). As result of these projections binary and unary cost functions are updated in both agents.

Projecting Costs with *soft-all-different*

We follow the approach described in [20] for the centralized case, where it is able to enforce GAC on the *soft-all-different* global cost function in polynomial time, something that is exponential for a generic non-binary cost function. A graph for every *soft-all-different* is constructed following [28]. This graph is stored by the agent and updated during execution. Every time a projection operation is required, instead of exhaustively looking at all tuples of the global cost function, the minimum cost that can be projected is computed as the flow of minimum cost of the graph associated with the cost function [20]. Minimum flow cost computation is based on the successive shortest path algorithm, which searches shortest paths in the graph until no more flows can be added to the graph. Pseudocode appears in Figure 17.

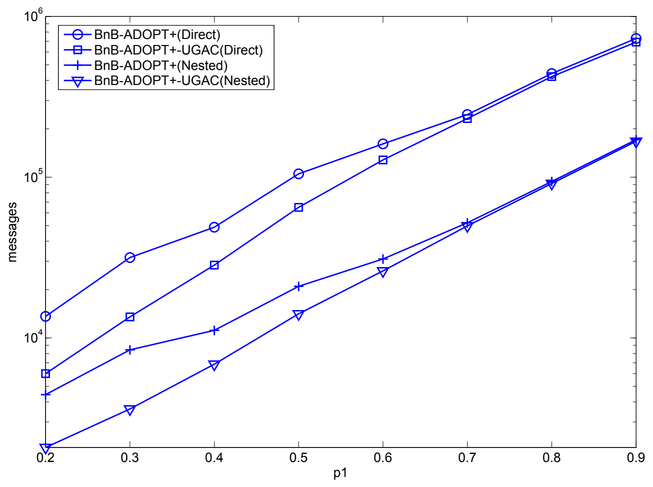
Projecting Costs with *soft-at-most*

For the *soft-at-most* global cost function we propose the following technique to project costs from the global cost function *soft-at-most[k,v](T)* to the unary cost functions $C(x_i)(v)$. Agent x_i counts how many agents in T have a singleton domain $\{v\}$. If the number of singleton domains is greater than k a minimum cost equal to the number of singleton domains minus k can be added to the unary cost $C(v)$ in one of the agents of the global cost function. We always project on the first agent of the cost function. In order to maintain equivalence, the *soft-at-most* stores this cost, that will be decremented from any future projection performed. Pseudocode is presented in Figure 18.

5.4 Experimental Results

To evaluate the impact of including global cost functions in DCOPs, we tested several random DCOPs including *soft-all-different* and *soft-at-most* global cost functions.

We consider binary random DCOPs with 10 variables and domain size of 5. The number of binary cost functions is $p_1 \cdot n(n-1)/2$, where n is the number of variables and p_1 is the network connectivity that varies in the range $[0.2, 0.9]$ in steps of 0.1. Binary costs are selected from an uniform cost distribution. Two types of binary cost functions are used, cheap and expensive. Cheap cost functions extract costs from the set $\{0, ..., 10\}$ while expensive ones extract costs from the set $\{0, ..., 1000\}$. The proportion of expensive cost functions is 1/4 of the total number of binary cost functions



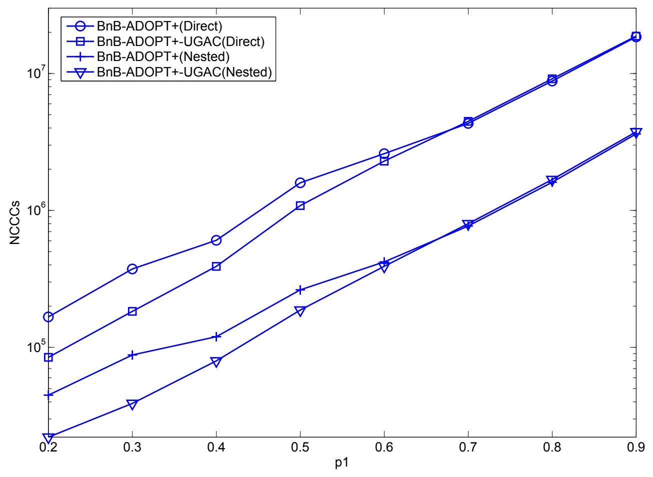
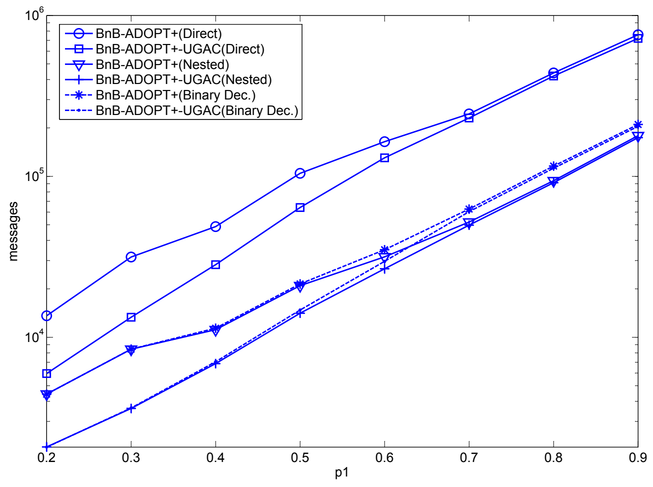


Figure 19: Experimental results of random DCOPs including *soft-all-different* global cost functions with the violation measure μ_{var}.

(this is done to introduce some variability among binary tuple costs [11]). In addition to binary cost functions, global cost functions are included. The first experiment includes 2 *soft-all-different*(T) global cost functions in every instance, where T is a set of 5 randomly chosen variables. Two types of violation measures are used for *soft-all-different*: μ_{var} and μ_{dec}. The second experiment includes 2 *soft-at-most[k,v]*(T) global cost functions in every instance, where T is a set of 5 randomly chosen variables, k (number of occurrences) is randomly chosen in the set $\{0, ..., 3\}$ and v is randomly selected in the variable domains. The violation measure used is μ_{var}. To balance binary and global costs, the cost of the *soft-all-different* and *soft-at-most* is calculated as the amount of the violation measure multiplied by 1000. Results are averaged over 50 instances for



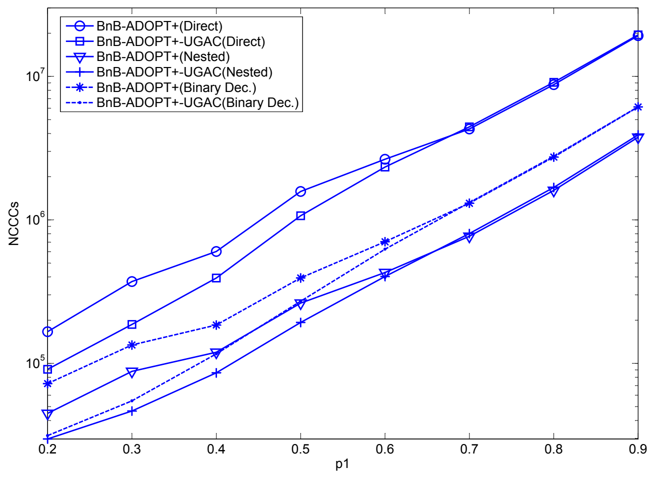


Figure 20: Experimental results of random DCOPs including *soft-all-different* global cost functions with the violation measure μ_{dec}.

each experiment.

We tested the extended versions of BnB-ADOPT$^+$ and BnB-ADOPT$^+$-UGAC able to handle global cost functions using a discrete event simulator. Computational effort is evaluated in terms of non-concurrent constraint checks (NCCCs) [27]. Network load is evaluated in terms of the number of messages exchanged. We have considered the different ways to incorporate global cost functions. For *soft-all-different* and *soft-at-most* with μ_{var} we tested on two representations: direct and nested representation because these global cost functions are not binary decomposable with violation measure μ_{var}. For *soft-all-different* with μ_{dec} we tested on the three possible representations: direct, nested and binary.

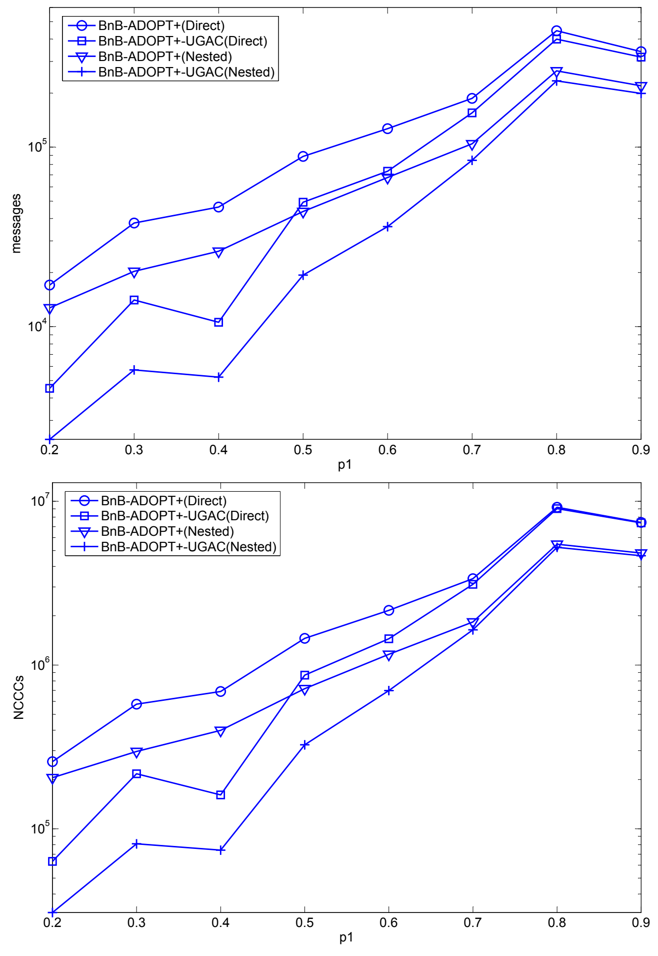


Figure 21: Experimental results of random DCOPs including *soft-at-most* global cost functions with the violation measure μ_{var}.

Specifically for UGAC enforcement, computational effort is measured as follows. For the sets including *soft-all-different* global cost functions, we use the special propagator proposed in [20]. Every time a projection operation is required, instead of exhaustively looking at all tuples of the global cost function (which would increment the NCCC counter for each tuple), we compute the minimum flow of this graph based on the successive calls to the shortest path algorithm. We assess the computational effort of the shortest path algorithm as the number of nodes of the graph where this algorithm is executed (this looks reasonable for small graphs, which is the case here). Each time the shortest path algorithm is executed, we add this number to the NCCC counter of the agent.

For the experiments including the *soft-at-most* global cost functions, every time the cost of the violation measure is computed as the number of singleton domains minus k, the NCCC counter is incremented.

Figures 19-20 contain the results of the first experiment including *soft-all-different* with violation measures μ_{var} and μ_{dec}. Figure 21 contains the results of the second experiment including *soft-at-most* with violation measure μ_{var}.

In general, we observe that the nested representation is better than the direct one in terms of messages and NCCCs, for both BnB-ADOPT^{+} and BnB-ADOPT^{+}-UGAC. In the direct representation, VALUE messages are sent only to the last agent of the global cost function whereas in the nested representation VALUE messages are sent to every agent in the scope of the cost function that is below the current one. However the early detection of dead-ends compensates by far these extra messages that are sent in the nested representation.

It is of special interest to observe the behavior of the binary decomposition. We can observe that the nested representation is consistently better than the binary decomposition, especially on high density problems. For BnB-ADOPT^{+}-UGAC this can be explained by the fact that applying UGAC on global cost functions may prune more values than applying UGAC on its binary decomposition. This was confirmed by comparing the number of DEL messages generated by every representation. For BnB-ADOPT^{+}, we see that it is also better to represent the global cost functions by the nested representation than by the binary decomposition.

Considering the impact of maintaining UGAC we observe that it really pays-off in terms of messages, especially for low and medium connectivities. As expected, UGAC maintenance sometimes requires a higher number of NCCCs because more computation must be done to make the problem UGAC.

6 Conclusion

We have introduced the use of global constraints in distributed constraint reasoning. We have proposed three different ways to represent global constraints in a distributed constraint network, depending on whether the constraint is contractible and/or binary decomposable. This approach applies similarly to DisCSPs and DCOPs. We have evaluated the performance of ABT on DisCSPs and of BnB-ADOPT^{+} on DCOPs, both with or without unconditional generalized arc consistency using different representations for global constraints. Maintaining some form of local consistency is never harmful in terms of messages and almost never in terms of NCCCs. Regarding the different representations of global constraints, the direct representation often is the less efficient one, followed by the binary representation. The nested representation seems to offer a good compromise: it is never worse than direct, and in some cases it is better than binary. This is good news because there are many more constraints that are contractible (the condition for nested representation) than constraints that are binary decomposable.

Acknowledgments

Christian Bessiere is partially supported by the French Agence Nationale de la Recherche project ANR-10-BLA-0214, and by the European Community project FP7-284715 ICON. The work of Ismel Brito, Patricia Gutierrez and Pedro Meseguer has been partially supported by the Spanish project TIN2009-13591-C02-02 and Generalitat de Catalunya 2009-SGR-1434. This work was started from a bilateral CNRS-CSIC project (CSIC number: 2010FR0040).

References

[1] Lynch, N. A. (1996) *Distributed Algorithms*. Morgan Kaufmann, New York.

[2] Yokoo, M., Durfee, E., Ishida, T., and Kuwabara, K. (1998) The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Tr. Know. Data Engin.*, **10**, 673–685.

[3] Meisels, A. and Zivan, R. (2007) Asynchronous forward-checking for discsps. *Constraints*, **12**, 131–150.

[4] Modi, P. J., Shen, W. M., Tambe, M., and Yokoo, M. (2005) Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, **161**, 149–180.

[5] Chechetka, A. and Sycara, K. P. (2006) No-commitment branch and bound search for distributed constraint optimization. *Proc. AAMAS-06*, Hakodate, Japan, 8-12 May, pp. 1427–1429. ACM Press, New York.

[6] van Hoeve, W. J. and Katriel, I. (2006) Global constraints. In van Beek, P., Rossi, F., and Walsh, T. (eds.), *Handbook of Constraint Programming*, pp. 205–244. Elsevier, Amsterdam.

[7] Michel, L., Schulte, C., and Van Hentenryck, P. (2007) Constraint programming tools. In Benhamou, F., Jussien, N., and OSullivan, B. (eds.), *Trends in Constraint Programming*, pp. 41–57. Wiley, New Jersey.

[8] Bessiere, C. and Van Hentenryck, P. (2003) To be or not to be ... a global constraint. *Proc. CP-03*, Kinsale, Ireland, 29 September - 3 October, pp. 789–794. Springer-Verlag, Berlin.

[9] Maher, M. (2009) Open contractible global constraints. *Proc. IJCAI-09*, Pasadena, USA, 11-17 July, pp. 578–583. AAAI Press, Menlo Park.

[10] Brito, I. and Meseguer, P. (2008) Connecting ABT with arc consistency. *Proc. CP-08*, Sydney, Australia, 14-18 September, pp. 387–401. Springer-Verlag, Berlin.

[11] Gutierrez, P. and Meseguer, P. (2010) BnB-ADOPT$^+$ with several soft AC levels. *Proc. ECAI-10*, Lisbon, Portugal, 16-20 August, pp. 67–72. IOS Press, Amsterdam.

[12] Gutierrez, P. and Meseguer, P. (2012) Improving BnB-ADOPT$^+$-AC. *Proc. AAMAS-12*, Valencia, Spain, 4-8 June, pp. 111–222. ACM Press, New York.

[13] Bejar, R., Domshlak, C., Fernandez, C., Gomes, C., Krishnamachari, B., Selman, B., and M., V. (2005) Sensor networks and distributed csp: Communication, computation and complexity. *Artificial Intelligence*, **161**, 117–147.

[14] Petit, T., Regin, J. C., and Bessiere, C. (2001) Specific filtering algorithms for over-constrained problems. *Proc. CP-01*, Paphos, Cyprus, 26 November - 1 December, pp. 451–463. Springer-Verlag, Berlin.

[15] Bessiere, C., Brito, I., Maestre, A., and Meseguer, P. (2005) Asynchronous backtracking without adding links: a new member in the ABT family. *Artificial Intelligence*, **161**, 7–24.

[16] Baker, A. B. (1994) The hazards of fancy backtracking. *Proc. AAAI-94*, Seattle, USA, 31 July-4 August, pp. 288–293. AAAI Press, Menlo Park.

[17] Maher, M. (2009) Soggy constraints: Soft open global constraints. *Proc. CP-09*, Lisbon, Portugal, 20-24 September, pp. 584–591. Springer-Verlag, Berlin.

[18] Meseguer, P., Rossi, F., and Schiex, T. (2006) Soft constraints. In van Beek, P., Rossi, F., and Walsh, T. (eds.), *Handbook of Constraint Programming*, pp. 279–326. Elsevier, Amsterdam.

[19] Larrosa, J. and Schiex, T. (2003) In the quest of the best form of local consistency for weighted CSP. *Proc. IJCAI-03*, Acapulco, Mexico, 9-15 August, pp. 239–244. AAAI Press, Menlo Park.

[20] Lee, J. H. M. and Leung, K. L. (2009) Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. *Proc. IJCAI-09*, Pasadena, USA, 11-17 July, pp. 559–565. AAAI Press, Menlo Park.

[21] Yeoh, W., Felner, A., and Koenig, S. (2010) BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *JAIR*, **38**, 85–133.

[22] Gutierrez, P. and Meseguer, P. (2010) Saving redundant messages in BnB-ADOPT. *Proc AAAI-10*, Atlanta, USA, 11-15 July, pp. 1259–1260. AAAI Press, Menlo Park.

[23] Brito, I. and Meseguer, P. (2006) Asynchronous backtracking for non-binary DisCSP. *ECAI-06 DCR workshop*, Riva del Garda, Italy, 28 August.

[24] Zivan, R. and Meisels, A. (2006) Dynamic ordering for asynchronous backtracking on discsps. *Constraints*, **11**, 179–197.

[25] Regin, J. C. (1994) A filtering algorithm for constraints of difference in CSPs. *Proc. AAAI-94*, Seattle, USA, 31 July-4 August, pp. 362–367. AAAI Press, Menlo Park.

[26] Palmer, E. M. (1985) *Graphical Evolution*. Wiley, New Jersey.

[27] Meisels, A., Kaplansky, E., Razgon, I., and Zivan, R. (2002) Comparing performance of distributed constraint processing algorithms. *AAMAS-02 DCR workshop*, Bologna, Italy, 16 July, pp. 86–93.

[28] van Hoeve, W. J., Pesant, G., and Rousseau, L. M. (2006) On global warming: flow-based soft global constraints. *Journal of Heuristics*, **12**, 347–373.