Maintaining Soft Arc Consistencies in BnB-ADOPT$^+$ During Search*

P. Gutierrez[1], J.H.M. Lee[2], K.M. Lei[2], T.W.K. Mak[3], P. Meseguer[1]

[1] IIIA - CSIC, Universitat Autònoma de Barcelona, 08193 Bellaterra Spain
{patricia,pedro}@iiia.csic.es
[2] Department of Computer Science and Engineering,
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
{jlee,kmlei}@cse.cuhk.edu.hk
[3] NICTA Victoria Laboratory & University of Melbourne, VIC 3010, Australia
Terrence.Mak@nicta.com.au

Abstract. Gutierrez and Meseguer show how to enforce consistency in BnB-ADOPT$^+$ for distributed constraint optimization, but they consider unconditional deletions only. However, during search, more values can be pruned conditionally according to variable instantiations that define subproblems. Enforcing consistency in these subproblems can cause further search space reduction. We introduce efficient methods to maintain soft arc consistencies in every subproblem during search, a non trivial task due to asynchronicity and induced overheads. Experimental results show substantial benefits on three different benchmarks.

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) have been applied in modeling and solving a substantial number of multiagent coordination problems, such as meeting scheduling [1], sensor networks [2] and traffic control [3]. Several distributed algorithms for optimal DCOP solving have been proposed: ADOPT [4], DPOP [5], BnB-ADOPT [6], NCBB [7] and others.

BnB-ADOPT$^+$-AC/FDAC [8] incorporate consistency enforcement during search into BnB-ADOPT$^+$ [9], obtaining substantial efficiency improvements. Enforcing consistency allows to prune some values, making the search space smaller. This previous work considers unconditional deletions only so as to avoid overhead in handling assignments and backtracking. However, values that could be deleted conditioned to some assignments will not be pruned with this strategy, so that search space reduction opportunities are missed. In this paper, we propose an efficient way to maintain soft

* We are grateful to the anonymous referees for their constructive comments. The work of Lei and Lee was generously supported by grants CUHK413808, CUHK413710 and CUHK413713 from the Research Grants Council of Hong Kong SAR. The work of Gutierrez and Meseguer was partially supported by the Spanish project TIN2009-13591-C02-02 and Generalitat de Catalunya 2009-SGR-1434. The work of Gutierrez, Lee and Meseguer was also jointly supported by the CSIC/RGC Joint Research Scheme grants S-HK003/12 and 2011HK0017. The work of Mak was performed while he was at CUHK.

arc consistencies, considering any kind of deletions resulting from enforcing consistency in asynchronous distributed constraint solving, something that—to the best of our knowledge—has not been explored before.

A search-based constraint solving algorithm forms subproblems of the original problem by assignments. We maintain soft arc consistencies in each subproblem, so that variable assignments during search are also considered in consistency enforcement. As a result, we can explore more value pruning opportunities and thus further reduce the search space. Gutierrez and Meseguer introduce an extra copy of cost functions in each agent, so that search and consistency enforcement are done asynchronously. Our contribution goes further maintaining soft arc consistencies in each subproblem during search, so that (i) search and consistency enforcement are done asynchronously, introducing some extra copies of cost functions; (ii) the induced overhead caused by backtracking and undoing assignments and deletions is minimized. The asynchronicity requirement and different cost measurements require us to introduce novel techniques over those used in centralized CP. Experimentally, we show the benefits of our proposal on benchmarks usually unamenable to solvers without consistency.

2 Preliminaries

DCOP. A DCOP is defined by $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \alpha \rangle$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of variables; $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of finite domains for $\mathcal{X}$; $\mathcal{C}$ is a set of cost functions; $\mathcal{A} = \{1, ..., n\}$ is a set of n agents and $\alpha : \mathcal{X} \rightarrow \mathcal{A}$ maps each variable to one agent. We use binary and unary cost functions only, which produce non-negative costs. The cost of a complete assignment is the sum of all unary and binary cost functions evaluated on it. An *optimal solution* is a complete assignment with minimum cost. Each agent holds exactly one variable, so that variables and agents can be used interchangeably. Agents communicate through messages, which are never lost and delivered in the order they were sent, for any agent pair.

DCOPs can be arranged in a *pseudo-tree*, where nodes correspond to variables and edges correspond to binary cost functions. There is a subset of edges, called tree-edges, that form a rooted tree. The remaining edges are called back-edges. Variables involved in the same cost function appear in the same branch. Tree edges connect parent-child nodes. Back-edges connect a node with its pseudo-parents and pseudo-children.

BnB-ADOPT and BnB-ADOPT$^+$. BnB-ADOPT [6] is an algorithm for optimal DCOP solving. It uses the communication framework of ADOPT [4] (agents are arranged in a pseudo-tree), but it changes the search strategy to depth first branch-and-bound. It shows improvements over ADOPT. Each agent holds a context, as a set of assignments involving some of the agent's ancestors that is updated with message exchanges. Message types are: VALUE, COST and TERMINATE. A BnB-ADOPT agent executes this loop: it reads and processes all incoming messages and assigns its value. Then, it sends a VALUE to each child or pseudochild and a COST to its parent. BnB-ADOPT$^+$ [9] is a version of BnB-ADOPT that prevents from sending most redundant messages, keeping optimality and termination. It substantially reduces communication.

Soft Arc Consistency. Let (i, a) represents x_i taking value a, $\top$ is the lowest unacceptable cost, C_{ij} is the binary cost function between x_i and x_j, C_i is the unary cost

function on x_i values, C_ϕ is a zero-ary cost function (lower bound of the cost of any solution). We consider the following local consistencies [10, 11]:

- *Node Consistency* (NC): (i, a) is NC if $C_\phi + C_i(a) < \top$; x_i is NC if all its values are NC and $\exists b \in D_i$ s.t. $C_i(b) = 0$. P is NC if every variable is NC.
- *Arc Consistency* (AC): (i, a) is AC w.r.t. C_{ij} if $\exists b \in D_j$ s.t. $C_{ij}(a, b) = 0$; b is a *support* of a; x_i is AC if all its values are AC w.r.t. every binary cost function involving x_i; P is AC if every variable is AC and NC.
- *Directional Arc Consistency* (DAC): (i, a) is DAC w.r.t. C_{ij}, $j > i$, if $\exists b \in D_j$ s.t. $C_{ij}(a, b) + C_j(b) = 0$; b is a *full support* of a; x_i is DAC if all its values are DAC w.r.t. every C_{ij}; P is DAC if every variable is DAC and NC.
- *Full DAC* (FDAC): P is FDAC if it is DAC and AC.

AC/DAC can be reached by forcing supports/full supports to NC values and pruning values that are not NC. Supports can be forced by projecting the minimum cost from its binary cost functions to its unary costs, and then projecting the minimum unary cost into C_ϕ. Full supports can be forced in the same way, but first it is needed to extend from the unary costs of neighbors to the binary cost functions the minimum cost required to perform in the next step the projection over the value. The systematic application of *projection* and *extension* does not change the optimum cost [10, 11]. When we prune a value from x_i, we need to recheck AC/DAC on every variable that x_i is constrained with, since the deleted value could be the support/full support of a value of a neighbor variable. So, a deleted value in one variable might cause further deletions in others. The AC/DAC check must be done until no further values are deleted.

BnB-ADOPT$^+$ and Soft Arc Consistencies. BnB-ADOPT$^+$ has been combined with AC and FDAC [8]. Search is based on BnB-ADOPT$^+$, maintaining the same data and communication structure. Soft arc consistencies are enforced on a copy of the original cost functions, limited to unconditional deletions. This combination has caused a number of modifications in the original algorithm, both in messages and in computation.

Regarding messages, (i) COST messages include $subtreeContr$ that aggregates the costs of unary projections to C_ϕ made on every agent; (ii) VALUE messages include $\top$ and C_ϕ; (iii) a new DEL message is added to inform of value deletions; when received, neighbors recheck AC/FDAC, which may lead to further deletions; (iv) a new UCO message is added when FDAC is enforced, to inform the unary costs needed for enforcing DAC; when received, agents enforce DAC with any other higher constrained agents and recheck FDAC, which may lead to further deletions.

Regarding computation, each agent holds one copy of constrained agents' domains and related binary cost functions for consistency enforcement. Handling value deletions require some extra effort. Only the agent owner of a variable can modify its domain.

3 Maintaining Soft Arc Consistencies

We enforce AC and FDAC asynchronously in all subproblems during search by utilizing additional copies of variable domains and cost functions in each agent. To explain our *Maintaining AC* (MAC) and *Maintaining FDAC* (MFDAC) algorithms, we first outline an agent classing scheme based on the position of an agent in the problem structure. The

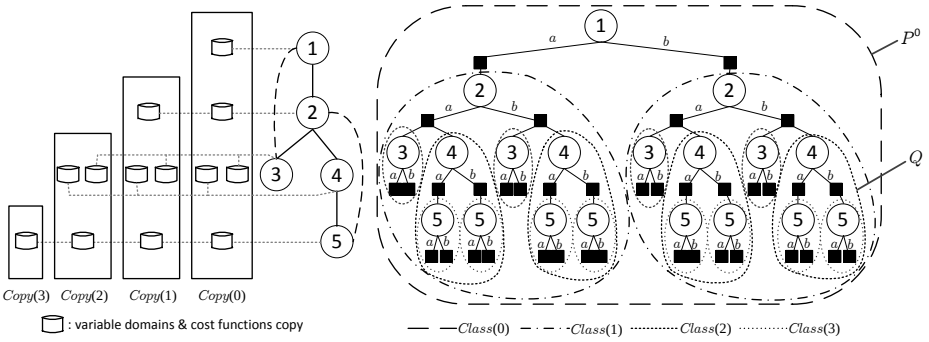

Fig. 1. Left: The pseudo-tree of a DCOP with five variables, the variable domains and cost functions copies they maintain. Right: Search tree (a/b domains), subproblems and classes of subproblems. Subproblems at the same depth belong to same class.

scheme governs the required number of copies of variable domains and cost functions. Second, we provide the information of messages in our methods and an overview of the changes in the overall message handling mechanism after adopting our new methods in BnB-ADOPT$^+$. Third, we provide methods for reinitializing variable domains and cost functions copies in an agent when the context of a subproblem changes. Such reinitialization is needed since conditional deletions are no longer valid. Thus, consistency enforcement has to start from scratch again using the new context. Fourth, we propose a new message type and the handling mechanism for backtracking, when an agent arrives at the empty domain within a subproblem. This means that the assignments of some ancestor agents cannot lead to the optimal solution and should be pruned. Fifth, we reduce costs by transferring deletions from subproblems to inner subproblems. Sixth, we present an ordering scheme and asynchronous messaging mechanism to ensure that the two separate copies of the same cost function stored in the two constrained agents are identical even in the presence of simultaneous consistency operations. Finally, we describe how we ensure optimality and termination after introducing the new methods.

3.1 Classes of Subproblems

In BnB-ADOPT$^+$ [9], all agents are organized in a pseudo-tree (Fig. 1 Left). The variable ordering of the corresponding AND-OR search tree [6] (Fig. 1 Right) follows the (partial) order defined in the pseudo-tree. When an agent is assigned a value, the descendant agents together with the current assignments form a subproblem. Notations: P^0 is the original DCOP; P is a subproblem of P^0; T^0 is a pseudo-tree that defines the variable ordering in P^0; d_j is the *depth* of agent j in T^0 as the distance from the root node to j excluding back-edges; $vars(P)$ is the set of variables of P; $depth(P)$ is the smallest depth among all variables in $vars(P)$; $ancestors(P)$ is the set of *ancestor variables* satisfying (1) they are in $vars(P^0)$ but not in $vars(P)$, (2) they have depths smaller than $depth(P)$, and (3) they are constrained with at least one variable in $vars(P)$; $context(P)$ is the variable assignments of $ancestors(P)$; $context_j$, the *context* of agent j, is the set $\{(i, a, t)|i = j$ or i is an ancestor of j, a is agent i value

which is assigned to i at timestamp $t\}$. Two contexts are *compatible* if no agent takes on different values in the two contexts. Every *subproblem* P of P^0 is uniquely identified by $(depth(P), ancestors(P), context(P), vars(P))$.

Fig. 1 Right illustrates the search tree and subproblems of a DCOP with 5 agents. Each circular node is the root node of a subproblem and there are 19 such subproblems (including the original problem) in the example. The original problem P^0 is $(0, \emptyset, \emptyset, \{1, 2, 3, 4, 5\})$ and $(2, \{1, 2\}, \{(1, b), (2, b)\}, \{4, 5\})$ (labeled Q in the figure) is the subproblem of P^0 after instantiating agent 1 and agent 2 to value b. We define a class of subproblems as follows. A subproblem P of P^0 is of *Class* d if $depth(P) = d$. We further define $Class(d) = \{P | depth(P) = d\}$.

Fig. 1 Right also illustrates the classes of subproblems of the DCOP. There are four classes of subproblems: $Class(0)$ involves the original problem only. $Class(1)$ includes two subproblems $(0, \{1\}, \{(1, a)\}, \{2, 3, 4, 5\})$ and $(0, \{1\}, \{(1, b)\}, \{2, 3, 4, 5\})$, $Class(2)$ includes eight subproblems in which four are rooted at node 3 and the other four are rooted at node 4. All $Class(2)$ subproblems hold the assignment information of agents 1 and 2 (their context). $Class(3)$ includes eight subproblems which are all rooted at node 5 and hold assignment information of agents 1, 2 and 4.

In BnB-ADOPT$^+$-AC/FDAC [8], search and consistency enforcement are done asynchronously: an extra copy of each cost function is used for consistency enforcement and they do not interfere with the original copy used for search. We use the same idea for MAC and MFDAC: we include extra copies of variable domains and cost functions for enforcing consistency in different subproblems, but not a copy for each subproblem. Each agent i of depth d_i will hold one copy $Copy(d)$ for each class $Class(d)$ of subproblems where $d \leq d_i$. For instance, in Fig. 1 Left, agents keep the following copies of cost functions and domains: agent 1 one copy, agent 2 two copies, agent 3 and 4 three copies, and agent 5 four copies. Then, each agent i will hold $d_i + 1$ copies of variable domains and cost functions and the space complexity of each agent is $O(dhm^2)$ where d is the agent's depth, h is the pseudo-tree's height and m is the maximum domain size of agents. These copies will play a key role in reinitializing domains and cost functions when conditional deletions are no longer valid in a context change.

3.2 Maintaining Consistencies in All Subproblems: an Overview

To maintain soft arc consistencies in every subproblem, extra operations and information exchanges are needed. The major additional operations include (1) reinitialization, (2) backtracking to the culprit when an empty domain is detected and (3) transferring

AC/FDAC	**MAC/MFDAC**
VALUE(*src*,*dest*,*value*,*threshold*,$\top$,C_ϕ)	VALUE(*src*,*dest*,*value*,*threshold*,$\top$,C_ϕ[],*context*)
COST(*src*,*dest*,*lb*,*ub*,*reducedContext*,*subtreeContr*)	COST(*src*,*dest*,*lb*,*ub*,*context*,*subtreeContr*[])
DEL(*src*,*dest*,*value*,*ACC*|*DACC*)	DEL(*src*,*dest*,*depth*,*values*[],*context*,*ACC*[]|*DACC*[])
UCO*(*src*,*dest*,*vectorOfExtensions*,*ACC*)	UCO**(*src*,*dest*,*depth*,*vectorOfExtensions*,*context*,*ACC*)
	BTK(*src*,*dest*,*targetDepth*,*context*)
* Only in FDAC	** Only in MFDAC

Table 1. Messages of AC, FDAC, MAC and MFDAC. New fields are underlined. DEL messages contain ACC or DACC depending on the AC or FDAC consistency level enforced.

```
procedure ProcessVALUE(msg)
    do the work as in BnB-ADOPT+
    Reinitialize(msg.src,msg.context)
    update ⊤ and C_φ if applicable

procedure ProcessCOST(msg)
    do the work as in BnB-ADOPT+
    Reinitialize(msg.src,msg.context)
    aggregate C_φ from msg.subtreeContr
    update C_φ if applicable

procedure ProcessDEL(msg)
    Reinitialize(msg.src,msg.context)
    d ← msg.depth
    vars ← set of variables i in context_self where d_i ∈ [0, msg.depth − 1]
    if values of vars in msg.context are compatible with those in context_self then
        for d' = d → d_self do
            delete msg.values[] from msg.src's domain in Copy(d')
            undo disordered operations in Copy(d') if necessary
            perform projection in Copy(d')
            update ACC counter if necessary

procedure ProcessUCO(msg)
    Reinitialize(msg.src,msg.context)
    d ← msg.depth
    vars ← set of variables i in context_self where d_i ∈ [0, msg.depth − 1]
    if values of vars in msg.context are compatible with those in context_self then
        if ACC_{self→msg.src} = msg.ACC then
            perform extension in Copy(d)
            update DACC counter if necessary
```

Fig. 2. Pseudocode for handling VALUE, COST, DEL and UCO messages

deletions to subproblems. Reinitialization is needed for ensuring the correctness of the algorithm. Backtracking to the culprit and transferring deletions to subproblems are not necessary for correctness but they can improve performance. Besides, to ensure the agents maintain the same cost functions in each copy, Gutierrez and Meseguer [12] proposed to include two new messages to synchronize deletions. However, these messages introduce an extra overhead and slow down the consistency enforcement. We propose a new method to allow agents to undo and reorder some of their operations in order to ensure identical cost functions copies.

Consistency enforcement in each subproblem is similar to that of Gutierrez and Meseguer [8], in which consistency is only enforced in the copy for the original problem. In our case, consistency is enforced in the copy for every class of subproblems at the same time. Extra information is embedded in the existing messages (TERMINATE message same as the one in BnB-ADOPT$^+$ and UCO only in FDAC and MFDAC) and only one new message type (BTK) is added. Table 1 summarizes the information per type. Fig. 2 shows the pseudocode for handling these messages (pseudocode for BTK appears in Section 3.4).

When an agent receives a VALUE or COST message, it first performs the BnB-ADOPT$^+$ process, and then it checks for reinitialization. When an agent receives a DEL message, it does the following steps (1) reinitialization checking, (2) compatibility checking, (3) value deletions, (4) maintaining identical cost function copies, (5) projections and (6) update projection counter. Similarly, when an agent receives an UCO message, it checks for reinitialization first and then performs the extension and extension counter update. After an agent i has processed a VALUE, COST, DEL or UCO message, AC/DAC may be re-enforced in $Copy(d)$ where $d \in [1, d_i]$ if (1) $Copy(d)$ is

reinitialized, (2) a better $\top$ or C_ϕ is found in $Copy(d)$, (3) some values are deleted in $Copy(d)$, and (4) some unary costs are increased in $Copy(d)$ (only apply for DAC).

3.3 Reinitialization

When enforcing consistencies in a subproblem P (excluding the original problem), the conditional deletions generated depend on the variable assignments information ($context(P)$) that P holds. These conditional deletions may not occur in other subproblems in $Class(depth(P))$, when the variable assignments have changed. Therefore, conditionally deleted values have to be recovered when values of ancestor agents change. When an agent $i \in ancestors(P)$ of a subproblem P in $Class(d)$ changes its value, the context no longer matches that of P. Search should be now switched to another $P' \in Class(d)$ such that $context(P')$ matches the new value of agent i and other existing assignments. In addition, the copies of cost functions owned by the agents in $vars(P')$ should be reset using the corresponding copies from upper classes and updated with the variable assignments in $context(P')$. Otherwise, the search algorithm will search for solution based on obsolete value pruning information and may result in suboptimal solution. This rationale justifies our next rule.

Rule 1 When an agent i changes its value, all agents $j \in vars(P)$ where $P \in \{P' | i \in ancestors(P')\}$ should reinitialize $Copy(d)$ where $d_i < d \leq d_j$ to be the corresponding subproblem based on the updated context. The reinitialization in j is done in a top-down sequence as follows. For $d = d_i + 1$ to d_j: (1) $Copy(d) = Copy(d-1)$; (2) Transform each binary cost function C_{jk} where $k \in ancestors(P')$ to unary cost functions C_j by assigning each k to value a where $(k, a) \in context(P')$.

Next we describe how to implement Rule 1 in BnB-ADOPT$^+$ with MAC and MF-DAC. Rule 1 affects an agent when there is a context change. We use VALUE, COST, DEL and UCO messages to carry the context information.

Receiving a VALUE message always signifies a context change in i's parent or pseudo-parent. Thus, agent i always performs reinitialization before deciding whether to change its own value. Receiving a COST message from any of its children may cause a context change. Agent i should always first compare the timestamps of the child's context and i's own context. If the child's context is older than or equal to i's context, i performs nothing. Otherwise, there is a context change and i will perform reinitialization before performing other BnB-ADOPT$^+$ operations. When receiving a DEL or UCO message, an agent performs similar checking before proceeding to consistency enforcement operations—if any—. Strictly speaking, reinitialization in an agent is needed only when handling VALUE and COST messages to ensure correctness of the solving result; skipping the reinitialization step for DEL and UCO messages will only miss pruning opportunities, and thus losing efficiency.

Agents need to have the context of *all* ancestors to check for context changes and to do reinitialization. In our VALUE, COST, DEL and UCO messages we include the context of the sender agent, instead of the agent's reduced context as used in BnB-ADOPT$^+$, which does not necessarily contain the information of all ancestors.

Fig. 3 shows how to reinitialize, and `Reinitialize`(src,$context_{src}$) is called whenever an agent $self$ receives a VALUE, COST, DEL or UCO message from src

```
procedure Reinitialize(src,context_src)
    mindepth = ∞
    for d = d_src → 0 do
        if d >= d_self then continue
        var ← variable of depth d in context_self
        if Time(context_src, var) > Time(context_self, var) then
            context_self(var) ← context_src(var)
            mindepth ← d + 1
    if mindepth ≠ ∞ then
        for d' = mindepth → d_self do
            Copy(d') ← Copy(d' − 1)
            var' ← variable of depth (d' − 1) in context_self
            TransformBinaryToUnary(Copy(d'), var', context_self)

function Timestamp(context, var) return t where (var, a, t) ∈ context

procedure TransformBinaryToUnary(Copy, var, context)
    if self is constrained with var and (var, a, t) ∈ context then
        for each b ∈ D_self do
            Copy.C_self(b) ← Copy.C_self(b) + Copy.C_self,var(b, a)
```

Fig. 3. Pseudocode for performing reinitialization

(as shown in Fig. 2 and 3). When $self$ receives $context_{src}$, it first checks whether the variable assignments that src holds are the latest information by comparing the timestamps of each variable assignment in $context_{src}$ and $context_{self}$. If the information in $context_{src}$ is more updated, $self$ updates $context_{self}$ according to $context_{src}$. If $self$'s context is updated, it has to perform reinitialization starting from the class of subproblems $Class(mindepth)$ where $mindepth$ is the smallest depth d_i and agent i's context has been changed in $context_{self}$. The operations for reinitializing $Copy(d)$ are described in Rule 1. We do not reinitialize the subproblem from the original problem but by duplicating from $Copy(d-1)$. Thus the works done in the current subproblem of $Class(d-1)$ will not have to be repeated in $Class(d)$.

3.4 Backtracking

Enforcing consistencies in a subproblem P can lead to an empty domain in some agent of P. In this case, $context(P)$ is inconsistent and it should be changed. Upon backtracking, the current assigned value a to the parent, say j, of the root of P should be changed: value a is removed from D_j, and agent j can then pick another value from D_j. This justifies our next rule.

Rule 2 If an agent i obtains an empty domain in the subproblem P during consistency enforcement, the agent $j \in ancestors(P)$ with $d_j = depth(P) - 1$ can delete its value a from its domain in $Copy(d_j)$, where $(j, a) \in context(P)$, provided that $context_j$ is compatible with $context(P)$.

We add a new message BTK to notify backtrackings. When agent i obtains an empty domain in P, i sends a message BTK(i,k,$depth(P) - 1, context(P)$) to its parent k. The BTK message is sent to the parent agent for propagation because agents can only communicate with constrained agents but the targeted agent may not be a constrained agent. Therefore, this message is propagated up the pseudo-tree until it reaches agent $j \in ancestors(P)$ where $d_j = depth(P) - 1$.

```
procedure ProcessBTK(msg)
    if d_self ≠ msg.targetDepth then
        sendMsg:(BTK, self, parent, msg.targetDepth, msg.context)
    else
        if msg.context is compatible with context_self then
            DeleteValue(Copy(msg.targetDepth), a) where (self, a, t) ∈ msg.context
```

Fig. 4. Pseudocode for handling the BTK message

Fig. 4 shows how to handle an incoming BTK message. When an agent other than j receives a BTK message, it forwards the message to its parent. When j receives that message, j checks whether the attached context is compatible with its own context. If yes, it knows that its current assignment $(j, a) \in context(P)$ will not lead to an optimal solution and it deletes a from $Copy(d_j)$. Otherwise, j ignores the message.

3.5 Transferring Deletions to Subproblems

Redundant deletions may appear in embedded subproblems. It is easy to see that if P' is a subproblem of P, the values deleted in P can also be deleted in P'. We can transfer the deletions in P to P' and no need to send out redundant information of these deletions for P'. Transferring deletions to subproblems not just avoid redundant DEL messages, it may also increase the chance of reducing more search space. Since the consistency enforcement in different subproblem is different, the suboptimal values found in P may not be found in P'. If we transfer these suboptimal values from P to P', more pruning opportunities may be found in P'.

When an agent deletes values in subproblem P, $depth(P) = d$, it can also apply the deletions to subproblems P' where $depth(P') > d$. A DEL message is labeled by d. When other agents receive that message, they apply the deletions to all the subproblems P' s.t. $depth(P') \geq d$. The pseudocode of transferring deletions to subproblems when receiving a DEL message is covered in the `ProcessDEL()` procedure in Fig. 2.

3.6 Keeping Cost Functions Copies Identical

Each of the two agents constrained by a cost function holds a separate copy of the cost function for consistency enforcement. It is thus of paramount importance to ensure the two copies being identical but this task is made difficult by the asynchronous nature of the search algorithm. Fig. 5 gives a simple example of simultaneous deletions [12] in constrained agents i and j, which cause projections from C_{ij} to C_i in agent j and C_j in agent i respectively. The asynchronous nature of message exchanges can result in the projections/extensions performed in different order and thus different C_{ij} copies in agents i and j respectively.

Gutierrez and Meseguer [12] propose to include two new messages to synchronize deletions but the overhead is high. by allowing one of the two agents to undo and reorder the operations. With this *Undo Mechanism* we keep the asynchronicity and avoid extra messages. We give preference to one of the two agents. The operations will be done in the order of the preferred agent, while the non-preferred one must undo the operations that do not follow that order.

Let us consider two constrained agents i and j, and the cost function between them C_{ij}; i and j each holds a copy of it, denoted by C_{ij}^i and C_{ij}^j respectively. Both agents

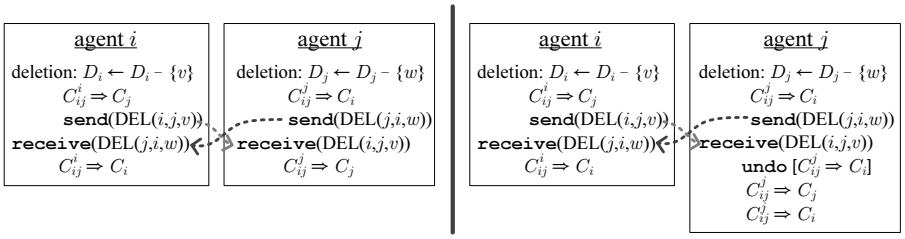

Fig. 5. Left: issue by two simultaneous deletions Right: proposed solution when maintaining AC.

maintain AC. The projection from C_{ij} to C_i has to be done on both C^i_{ij} and C^j_{ij}. $C^i_{ij} \Rightarrow C_i$ represents i performs projection from C_{ij} to C_i on i's copy and $C^j_{ij} \Rightarrow C_i$ represents j performs projection from C_{ij} to C_i on j's copy. If value v is deleted from D_i and value w from D_j simultaneously, both i and j will process these deletions (which imply each agent projecting from C_{ij} to each other) and they will send DEL messages to each other (Fig. 5 Left). If i is the preferred agent, upon receipt of the DEL message from j, it performs $C^i_{ij} \Rightarrow C_i$ and updates C^i_{ij}. However, when j receives the DEL message from i, if j realizes that it has done more projections $C^j_{ij} \Rightarrow C_i$ than the agent i, then it has to undo some of these projections, until both have done the same number of projections. The proposed solution appear in Fig. 5 Right. The same ordering of operations in both agents is achieved as follows. Agent i keeps a counter $ACC_{j\rightarrow i}$ to record the number of projections $C^i_{ij} \Rightarrow C_i$ (and $DACC_{j\rightarrow i}$ to record the number of extensions from agent j to i in FDAC/MFDAC cases). These counter and stack are stored in the copy of each class of subproblems. Agent j keeps a stack $P^j_{j\rightarrow i}$ that records each projection operation $C^j_{ij} \Rightarrow C_i$. The operations of the Undo Mechanism on C_{ij} between agents i and j for AC and MAC are:
Agent i:

- When there is a value deletion, perform projection $C^i_{ij} \Rightarrow C_j$. Attach $ACC^i_{j\rightarrow i}$ in a DEL message and send it to j. Then, reset $ACC^i_{j\rightarrow i}$ to zero.
- When i receives a DEL message from j, perform projection $C^i_{ij} \Rightarrow C_i$ and increment $ACC^i_{j\rightarrow i}$ by 1.

Agent j:

- When there is a value deletion, perform projection $C^j_{ij} \Rightarrow C_i$. Push this projection in the stack $P^j_{j\rightarrow i}$. Send the DEL message to i.
- When j receives a DEL message from i, pop and undo $|P^j_{j\rightarrow i}| - ACC^i_{j\rightarrow i}$ number of projection records from the stack $P^j_{j\rightarrow i}$, where $|P^j_{j\rightarrow i}|$ is the size of the stack $P^j_{j\rightarrow i}$, and clear the stack. Then, the DEL message is processed, projecting $C^j_{ij} \Rightarrow C_j$. If there is at least one pop/undo performed, then perform projection $C^j_{ij} \Rightarrow C_i$.

To maintain FDAC between two constrained agents i and j, DAC is maintained in one direction (e.g. j to i) and AC in the other (e.g. i to j). In FDAC, preference should be given to agent i if AC is enforced from C_{ij} to C_j since the enforcement of DAC from j

to i is ensured under the assumption that i is AC w.r.t. C_{ij} [10] (in AC, any agent i or j may be preferred). Due to space limits, we skip the details for FDAC and MFDAC.

3.7 Optimality and Termination

Enforcing MAC and MFDAC during BnB-ADOPT$^+$ search maintains the optimality and termination properties of BnB-ADOPT$^+$, as we see next.

Projections and extensions to maintain MAC and MFDAC are done on a copy of the cost functions. In this way, the search process is based on the unmodified original copy of the cost functions. The only changes with respect to the BnB-ADOPT$^+$ operations come from the fact that inconsistent values discovered by local consistency enforcement are removed from the domain of agents.

Termination is justified as follows. BnB-ADOPT$^+$ always terminates [6, 9] and the only change that BnB-ADOPT$^+$-MAC introduces is AC enforcement after variable assignments. AC enfocement terminates, because the number of agents involved is finite and their domains are also finite. When enforcing AC in a particular subproblem, after a finite amount of time all subproblem variables become AC (possibly after some value deletions) reaching a fixpoint.

Optimality is justified as follows. In the case of unconditional deletions, deleted values are suboptimal values which will not be present in the optimal solution, so it is completely legal to remove them. In the case of conditional deletions, deleted values are values proved inconsistent conditioned to the current assignment of ancestor agents. They are properly restored using a reinitialization mechanism when the assignments of ancestors change. Operation is as follows. An agent may change its assigned value, selecting another one from its domain, only after it receives a VALUE or COST message. Reinitialization is done whenever an agent receives a VALUE or COST message and there is context change. Thus, reinitialization is guaranteed to be performed before any agent changes its value, so that no obsolete value deletions will be considered. Then, in both cases all solutions potentially optimal are visited. Next we detail these operations, showing they do not affect optimality and termination.

In MAC (both unconditional and conditional deletions), we perform projections over the cost functions (projections from binary to unary cost functions, and from unary to C_ϕ). Projection is an equivalence preserving transformation [11]. Its application maintains the optimum cost and the set of optimal solutions. In our approach (distributed context), we assure identical copies of any binary cost function in the two involved agents: cost projections are performed in the same order in the two agents (Section 3.6). Therefore, costs cannot be duplicated when projections are performed inside each agent (equivalence is preserved) or when costs are propagated to other agents. Since each agent contributes to C_ϕ projecting on its unary cost functions, we can conclude that projections of different agents into C_ϕ does not duplicate costs. Proving that a value a of variable x_i is not NC involves its unary cost $C_i(a)$ and C_ϕ. Since we have seen that, neither $C_i(a)$ nor C_ϕ contains duplicated costs, the NC detection is correct and a's deletion is legal. Because of the NC definition, the first found optimal solution can never be pruned, since the cost of their values will never reach $\top$.

In the case of conditional deletions, the reinitialization mechanism (Section 3.3) ensures the correctness of values deletions in different copies. For each copy, projections

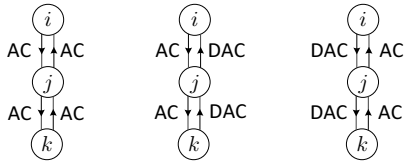

Fig. 6. Directions of enforcing AC/FDAC consistencies

and deletions are performed conditioned to the ancestor assignments. For example, in $Copy(0)$, projections are performed contemplating no previous assignments, and only unconditional deletions are detected; in $Copy(1)$, inconsistent values are discovered and deleted conditioned to the first-level ancestor's assignment; in $Copy(2)$, inconsistent values are deleted conditioned to the first and second level agent's assignment, and so on. Each time an agent of depth d changes its variable assignment, the $Copy(d)$ of descendant agents are restored to $Copy(d-1)$. In this way, all modifications that were performed according to the previous variable assignment are undone.

Regarding values pruned by backtracking messages, the justification of its correctness is as follows. When an empty domain is found in $Copy(d)$ in one agent, we have discovered that the current assignment of the ancestor at depth d is inconsistent, and so it must be removed. This is implemented by sending a BTK message to that ancestor. Note that only BTK messages containing a compatible context are accepted in the ancestor. In this way, it is assured that the ancestor agent changes its value if the empty domain of the descendant agent was generated considering a compatible context. Otherwise, either the descendant or the ancestor is missing one or several messages that will properly update their contexts. Upon receipt of these messages, proper actions, depending on the missing messages, will be taken by the ancestor/descendant.

Regarding MFDAC, in addition to projections, we have to take into account extensions, another equivalence preserving transformation [10]. Our approach (distributed case) is correct, since each agent can extend its own unary costs only. So no cost duplication may occur. The process is done in such a way that the copies of any binary cost function are kept identical in the two involved agents. From this point on, only projections are done, and arguments from previous paragraphs apply.

4 Experimental Results

We evaluate the efficiency of BnB-ADOPT$^+$-MAC/MFDAC (abbrev as MAC/MFDAC) by comparing to BnB-ADOPT$^+$-AC/FDAC (abbrev as AC/FDAC). For AC and MAC algorithms, AC is enforced in both directions of each binary cost function. The direction of DAC enforcement matters in FDAC and MFDAC algorithms. Fig. 6 shows the direction of AC and DAC enforcement between agents, where i (j) is the parent or pseudo-parent of j (k). For FDAC algorithm, we use the direction as shown in Fig. 6 Middle. DAC is enforced bottom-up so that the unary costs are pushed upward so as to hopefully increase the opportunities of pruning more values in upper agents (pruning values in upper agents is more preferred because BnB-ADOPT$^+$ is a depth-first search algorithm). For MFDAC, we evaluate both directions: MFDAC1 uses the direction shown in Fig. 6 Middle and MFDAC2 uses the direction in Fig. 6 Right. We

evaluate both because of the possible tradeoff between backtracking and direct pruning in upper agents. With MFDAC1, unary costs will float upward and increase the opportunities of pruning values directly in upper agents. However, MFDAC2 pushes the unary costs downward and increases the opportunities of reaching empty domains in lower agents, which can possibly increase the pruning opportunities in upper agents.

Our simulator runs in cycles, during which every agent reads its incoming messages, performs computation and sends its outcoming messages. Without delays, a message sent in a cycle is delivered in the next cycle. To make a more realistic evaluation, a random delay of $[0, 50]$ cycles is introduced to each message in our experiments. Besides, we have an extensive number of instances over three benchmarks. Since AC is too slow to generate results for hard or large-scale problems within a reasonable time, we set a 2×10^8 NCCCs limit in our simulator. One can expect that setting the NCCCs limit is to our *disadvantage* since MAC/MFDAC can improve even more on harder or larger-scale problems (normally taking bigger effort to solve but some of these problems are skipped because of the NCCCs limit). Three measures of performance are thus compared: (1) the number of messages to evaluate the communication cost, (2) the number of non-concurrent constraint checks (NCCCs) to evaluate the computation effort, and (3) the number of instances that can be solved within the 2×10^8 NCCCs limit to evaluate the general efficiency of each algorithm. In addition, we assume that each randomly delayed cycle costs 100 NCCCs and it is counted in the total NCCCs accordingly.

We test our algorithms on three sets of benchmarks: binary random DCOPs [8], Soft Graph Coloring Problems (SGCP) and Radio Link Frequency Assignment Problem (RLFAP) [13]. We run 50 instances for each parameter setting. Results are reported in Tables 2, 3 and 4. The columns show (from left to right) the problem, algorithm, the number of instances that can be solved within 2×10^8 NCCCs limit, the number of commonly solved instances (the number of messages and NCCCs are averaged over this number), total number of messages, number of VALUE, COST, DEL, BTK and UCO messages, and NCCCs. The best results for each measure are highlighted in bold for each parameter setting.

Binary random DCOPs [8] are characterized by $\langle n, d, p\rangle$, where n is the number of variables, d is the domain size and p is the network connectivity. We have generated random DCOP instances: $\langle n = 10, d = 10, p \in \{0.3, 0.4, 0.5, 0.6\}\rangle$. Costs are selected from a uniform cost distribution. Following Guiterrez and Meseguer [8], two types of binary cost functions are used, small and large. Small cost functions randomly extract costs from the set $\{0, ..., 10\}$ while large ones randomly extract costs from the set $\{0, ..., 1000\}$. The proportion of large cost functions is 1/4 of the total number of cost functions. Results are reported in Table 2.

Soft Graph Coloring Problems are the softened version of graph coloring problems by allowing the inequalities to return costs from the violation measure $M^2 - |v_i - v_j|^2$, where M is the maximum domain size, v_i and v_j are the values of agent i and j respectively. Each SGCP is also characterized by $\langle n, d, p\rangle$, where n is the number of variables, d is the domain size and p is the network connectivity. We evaluate four sets of instances: $\langle n \in \{6, 7, 8, 9\}, d = 8, p = 0.4\rangle$. Results are shown in Table 3.

We generate the Radio Link Frequency Assignment Problems according to two small but hard CELAR sub-instances [13], which are extracted from CELAR6. All

p	Algorithm	#instances solved within NCCCs limit	Avg. over (common instances)	#Msgs	#VALUE	#COST	#DEL	#BTK	#UCO	NCCCs
0.3	AC	**50**	50	6,802	1,619	5,099	59	0	0	5,622,762
	FDAC	**50**		4,645	1,062	3,389	117	0	53	3,857,078
	MAC	**50**		5,610	1,124	3,569	760	134	0	4,203,119
	MFDAC1	**50**		**3,656**	726	2,346	338	13	184	**2,738,511**
	MFDAC2	**50**		5,036	923	2,911	495	249	435	3,511,191
0.4	AC	47	47	56,632	11,581	44,946	79	0	0	42,210,453
	FDAC	48		39,560	8,043	31,188	195	0	105	29,477,148
	MAC	**50**		36,309	6,692	25,564	2,399	1,628	0	24,845,040
	MFDAC1	**50**		**28,493**	5,271	20,541	1,430	236	967	**19,236,541**
	MFDAC2	**50**		29,814	5,116	20,255	1,523	1,441	1,451	19,434,413
0.5	AC	35	34	106,194	20,796	85,260	106	0	0	78,603,224
	FDAC	38		75,074	14,412	60,231	247	0	152	55,129,851
	MAC	43		63,571	11,238	46,279	2,694	3,329	0	43,949,687
	MFDAC1	44		**54,564**	9,490	39,791	2,926	286	2,018	**36,699,194**
	MFDAC2	**46**		57,150	9,497	39,535	2,245	3,651	2,191	37,488,828
0.6	AC	9	9	124,222	26,839	97,268	86	0	0	91,145,921
	FDAC	16		90,850	14,867	55,465	277	0	211	51,437,525
	MAC	20		47,586	8,973	35,153	2,143	1,288	0	34,059,166
	MFDAC1	**24**		**37,697**	6,883	27,900	1,141	463	1,255	**27,122,566**
	MFDAC2	20		45,988	8,093	31,699	2,011	2,047	2,109	31,074,814

Table 2. Random DCOPs

n	Algorithm	#instances solved within NCCCs limit	Avg. over (common instances)	#Msgs	#VALUE	#COST	#DEL	#BTK	#UCO	NCCCs
6	AC	**50**	50	459	123	321	8	0	0	572,082
	FDAC	**50**		376	91	240	29	0	7	438,002
	MAC	**50**		358	81	190	67	11	0	361,607
	MFDAC1	**50**		**287**	51	127	54	9	30	**248,106**
	MFDAC2	**50**		367	71	161	70	17	40	333,807
7	AC	**50**	50	1,349	370	961	9	0	0	1,534,451
	FDAC	**50**		875	225	594	37	0	8	974,678
	MAC	**50**		888	213	507	143	14	0	841,000
	MFDAC1	**50**		**628**	127	314	95	20	51	**521,659**
	MFDAC2	**50**		883	185	437	143	27	81	733,084
8	AC	**50**	50	8,611	2,072	6,523	5	0	0	8,562,373
	FDAC	**50**		5,764	1,359	4,354	29	0	11	5,727,394
	MAC	**50**		4,955	1,044	3,166	625	109	0	4,261,463
	MFDAC1	**50**		**4,359**	905	2,942	287	61	138	3,799,575
	MFDAC2	**50**		4,695	857	2,615	613	163	437	**3,553,383**
9	AC	46	46	39,199	8,659	30,525	3	0	0	32,353,604
	FDAC	46		30,189	6,580	23,559	23	0	14	24,858,245
	MAC	**47**		23,164	4,554	15,882	2,545	170	0	17,448,119
	MFDAC1	**47**		25,738	5,265	19,124	795	69	453	19,829,021
	MFDAC2	**47**		**20,219**	3,547	12,624	2,081	493	1,461	**13,863,427**

Table 3. Soft Graph Coloring Problems

RLFAP instances are generated with parameters $\langle i, n, d\rangle$, where i is the index of the CELAR sub-instances, n is an even number of links, and d is an even number of allowed frequencies. For each instance, we randomly extract a sequence of n links from the corresponding CELAR sub-instance and fix a domain of d frequencies. If two links are restricted not to take frequencies f_i and f_j with distance less than t, we measure the costs of interference by using a binary constraint with violation measure $\max(0, t-|f_i-f_j|)$. We evaluate three sets of instances: A$\langle 0, 10, 12\rangle$, B$\langle 1, 6, 6\rangle$, and C$\langle 1, 6, 8\rangle$. Results are reported in Table 4.

As we see in Tables 2, 3 and 4, MAC, MFDAC1 and MFDAC2 substantially further reduce the total number of messages and NCCCs, and be able to solve the same number or more instances within the NCCCs limit over all three benchmarks. Moreover, MAC

	Algorithm	#instances solved within NCCCs limit	Avg. over (common instances)	#Msgs	#VALUE	#COST	#DEL	#BTK	#UCO	NCCCs
A	AC	**50**	50	28,837	5,064	23,751	0	0	0	24,522,945
	FDAC	**50**		28,894	5,069	23,790	0	0	13	24,621,897
	MAC	**50**		22,840	,3802	16,540	1,954	522	0	17,447,513
	MFDAC1	**50**		**18,054**	2,937	12,000	1,606	378	1,090	13,051,121
	MFDAC2	**50**		19,233	2,888	11,711	1,861	1,250	1,501	**12,845,773**
B	AC	21	21	56,943	10,466	46,455	11	0	0	67,658,716
	FDAC	21		57,964	10,635	47,267	39	0	9	69,091,598
	MAC	**50**		29,120	4,930	21,521	1,061	1,596	0	37,861,737
	MFDAC1	**50**		**18,080**	3,228	13,900	403	433	100	**20,881,354**
	MFDAC2	**50**		25,430	4,490	19,489	541	702	197	2,937,7041
C	AC	18	18	29,385	5,505	23,853	15	0	0	34,158,516
	FDAC	18		31,133	5,814	25,250	47	0	9	36,259,302
	MAC	**50**		13,914	2,464	10,787	297	356	0	15,890,040
	MFDAC1	**50**		**11,964**	2,183	9,394	177	123	71	**14,067,760**
	MFDAC2	**50**		13,454	2,431	10,496	220	207	89	15,731,062

Table 4. Radio Link Frequency Assignment Problems

outperforms FDAC in almost all cases even when MAC is maintaining a weaker form of consistency than FDAC. Although our methods introduce overhead, i.e., increase in the number of DEL, BTK and UCO messages, the reduction in the number of VALUE and COST messages (and thus search space) outweighs the overhead. Therefore, we conclude that maintaining soft arc consistencies during search is beneficial.

We also observe that the improvement of MFDAC over AC and FDAC in random DCOPs increases as constraint density increases. More constraints in the problem implies more pruning opportunities and thus substantial smaller search space. Similar observations cannot be concluded for Soft Graph Coloring and Radio Link Frequency Assignment Problems since these problems have particular problem structures affecting the efficiency and power of consistency enforcement.

To compare the different directions of DAC enforcement, we can see MFDAC1 outperforms MFDAC2 in some instances while MFDAC2 outperforms MFDAC1 in others. For random DCOPs and Radio Link Frequency Assignment Problem, MFDAC1 performs the best in almost all instances. However, for Soft Graph Coloring Problem, MFDAC2 performs better for instances with $n = 9$ and MFDAC1 performs better on another three sets of instances. From these results we can see that the directions of DAC enforcement can affect the efficiency and the effects are problem-specific.

5 Conclusion

In this paper, we propose methods to maintaining soft arc consistencies in every subproblem during search. In order to preserve the asynchronicities of search and consistency enforcement, we propose to include extra copies (a small number) of variable domains and cost functions. Besides, we minimize the induced overhead caused by backtracking and undoing assignments and deletions by attaching information in the existing messages rather than creating new ones. We present the issues and solutions for maintaining consistencies in subproblems and ensure their correctness: (i) reinitializing variables' domains and cost functions after context changes in subproblems to ensure the search algorithm would not search on values using obsolete value pruning information, (ii) backtracking when an agent arrives at the empty domain within a subproblem

so as to prune the value in upper agents which could not lead to an optimal solution, (iii) transferring deletions from subproblems to further subproblems to avoid redundant messages, and (iv) asynchronous methods to ensure identical cost functions copies in different agents by ensuring the ordering of consistency operations between every two agents. Our experimental results show that our methods can substantially further reduce the communication and computation efforts compared to BnB-ADOPT$^+$-AC/FDAC, which only consider unconditional deletions. These results allow us to consider the proposed methods as important steps to maintain consistencies in every subproblems asynchronously during search and improve the efficiency of optimal DCOP solving. As a future work, we may go further to maintain the even stronger Existential Directional Arc Consistency (EDAC) [14] during distributed and asynchronous search, but preserving privacy is a concern [15]. The study of how DAC enforcement directions affect efficiency and the possible heuristics for such ordering is a worthwhile direction.

References

1. Maheswaran, R.T., Tambe, M., Bowring, E., Pearce, J.P., Varakantham, P.: Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In: Proc. AAMAS '04. (2004) 310–317
2. Jain, M., Taylor, M., Tambe, M., Yokoo, M.: DCOPs meet the realworld: exploring unknown reward matrices with applications to mobile sensor networks. In: Proc. IJCAI'09. (2009) 181–186
3. Junges, R., Bazzan, A.L.C.: Evaluating the performance of DCOP algorithms in a real world, dynamic problem. In: Proc. AAMAS'08. (2008) 599–606
4. Modi, P.J., Shen, W.M., Tambe, M., Yokoo, M.: ADOPT: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence **161** (2005) 149–180
5. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: Proc. IJCAI'05. (2005) 266–271
6. Yeoh, W., Felner, A., Koenig, S.: BnB-ADOPT: an asynchronous branch-and-bound DCOP algorithm. JAIR **38** (2010) 85–133
7. Chechetka, A., Sycara, K.: No-commitment branch and bound search for distributed constraint optimization. In: Proc. AAMAS'06. (2006) 1427–1429
8. Gutierrez, P., Meseguer, P.: BnB-ADOPT$^+$ with several soft arc consistency levels. In: Proc. ECAI'10. (2010) 67–72
9. Gutierrez, P., Meseguer, P.: Saving redundant messages in BnB-ADOPT. In: Proc. AAAI'10. (2010) 1259–1260
10. Larrosa, J., Schiex, T.: In the quest of the best form of local consistency for weighted CSP. In: Proc. IJCAI'03. (2003) 239–244
11. Larrosa, J., Schiex, T.: Solving weighted CSP by maintaining arc consistency. Artificial Intelligence **159**(1) (2004) 1–26
12. Gutierrez, P., Meseguer, P.: Improving BnB-ADOPT$^+$-AC. In: Proc. AAMAS '12. (2012) 273–280
13. Cabon, B., de Givry, S., Lobjois, L., Fcabon, L.L., Schiex, T.: Radio link frequency assignment. Constraints **4** (1999) 79–89
14. de Givry, S., Heras, F., Larrosa, J., Zytnicki, M.: Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In: Proc. IJCAI-05. (2005) 84–89
15. Gutierrez, P., Meseguer, P.: Enforcing soft local consistency on multiple representations for DCOP solving. In: Proc. CP 2010 workshop: Preferences and Soft Constraints. (2010) 98–113