Enforcing Soft Local Consistency on Multiple Representations for DCOP Solving

Patricia Gutierrez and Pedro Meseguer

IIIA, Institut d'Investigació en Intel.ligència Artificial
CSIC, Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Spain.
{patricia|pedro}@iiia.csic.es

Abstract. Connecting soft arc consistency with distributed search in DCOP solving has been very beneficial for performance. However, including higher levels of soft arc consistency breaks usual privacy requirements. To avoid this issue, we propose to keep different representations of the same problem on each agent, on which soft arc consistencies are enforced respecting privacy. Deletions caused in one representation can be legally propagated to others. Experimentally, this causes significant benefits.

1 Introduction

Recently, a number of methods for solving Distributed Constraint Optimization Problems (DCOP) have appeared [7–10] possibly as consequence of the raise of multiagent technology. In this problems, variables are distributed into several agents and the task of interest is to find a global optimal assignation in a distributed way.

Distributed search solves DCOP by exploring the search space using messages. Lately, the BnB-ADOPT$^+$ algorithm has been enhanced with some forms of soft arc consistency maintenance (specifically, AC* and FDAC*) [4], which have been shown very beneficial for performance, saving an important number of exploratory messages. Moving into the next soft arc consistency level, EDAC*, we have found that its enforcement breaks the privacy requirements usually assumed in the distributed setting. With the double aim of improving as much as possible distributed search performance while respecting agent privacy, we present the following approach. To enforce FDAC* agents must be ordered. At each agent, we propose to keep several representations of the other agents (each representation originally corresponds to a different order), from which some values can be deleted. Interestingly, these deletions can be legally propagated among representations. In some sense, this remembers the channeling constraints idea [1], where different modelings of the same problem coexist in the solving process and pruning in one model is propagated to other models by the channeling constraints. Experimentally, our approach causes significant savings on two benchmarks.

This paper is organized as follows. After recalling basic concepts, we present the idea of connecting BnB-ADOPT$^+$ with soft arc consistency, showing that EDAC* connection breaks privacy requirements. To avoid this issue, we propose maintainig multiple representations. Experimentally, we obtain significant benefits on two benchmarks.

2 Preliminaries

COP. A binary *Constraint Optimization Problem* (COP) is defined by $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, 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 (x_i takes values in $D(x_i)$), and $\mathcal{C}$ is a set of unary and binary cost functions; $C \in \mathcal{C}$ specifies the cost of every combination of values of $var(C)$, $C : \prod_{x_i \in var(C)} D(x_i) \mapsto N \cup \{0, \infty\}$. The cost of a complete tuple is the addition of all individual cost functions evaluated on that particular tuple. An *optimal solution* is a complete tuple with minimum cost. This definition assumes the weighted model of soft constraints [6].

Soft Arc Consistency. Two COP instances defined over the same set of variables are *equivalent* if they define the same cost distribution on complete assignments. Considering a single COP: (i, a) means the value a of variable x_i, $\top$ 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 that represents a necessary global cost of any complete assignment. As [5, 2], we consider the following local consistencies for the weighted model (variables connected by a cost function are ordered):

- *Node Consistency**: (i, a) is node consistent* (NC*) if $C_\phi + C_i(a) < \top$; x_i is NC* if all its values are NC* and there is $a \in D_i$ such that $C_i(a) = 0$; a COP is NC* if every variable is NC*.
- *Arc consistency**: (i, a) is arc consistency (AC) with respect to cost function C_{ij} if there is $b \in D_j$ s.t. $C_{ij}(a, b) = 0$; b is a *simple support* of a; x_i is AC if all its values are AC with respect to every binary cost function involving x_i; a COP is AC* if every variable is AC and NC*.
- *Directional arc consistency**: (i, a) is directional arc consistent (DAC) with respect to cost function C_{ij}, $j > i$, if there is $b \in D_j$ such that $C_{ij}(a, b) + C_j(b) = 0$; b is a *full support* of a; x_i is DAC if all its values are DAC with respect to every C_{ij}, $j > i$; a COP is DAC* if every variable is DAC and NC*.
- *Full DAC**: a COP is FDAC* if it is DAC* and AC*.
- *Existential arc consistency**: Variable x_i is existential arc consistent (EAC) if there is at least one value $a \in D_i$ such that $C_i(a) = 0$ and it has a full support in every cost function C_{ij}; a COP is EAC* if every variable is NC* and EAC.
- *EDAC**: a COP is EDAC* if it is FDAC* and EAC*.

AC*/DAC* can be reached forcing simple/full supports to NC* values and pruning values not NC*. Simple supports can be forced on every value 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 these operations (projection and extension) produces equivalent instances, so they do not change the minimum cost nor the optimal solutions of the original instance [5]. When we prune a value from x_i to ensure AC*/DAC*, we need to recheck AC*/DAC* on every variable that x_i is constrained with, since the deleted value could be the simple/full support.

DCOP. A *Distributed Constraint Optimization Problem* (DCOP) is defined by a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \alpha \rangle$, where $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ define a COP, $\mathcal{A} = \{1, \ldots, p\}$ is a set of p agents,

and $\alpha : \mathcal{X} \rightarrow \mathcal{A}$ maps each variable to one agent. We assume that each agent holds exactly one variable (so the terms variables and agents can be used interchangeably) and cost functions are unary and binary only. Agents communicate through messages, which could be delayed but never lost, and they are delivered in the order they were sent.

BnB-ADOPT / BnB-ADOPT$^+$. BnB-ADOPT [10] is a reference algorithm for DCOP. It is a depth-first version of ADOPT [7], showing a better performance. As ADOPT, it arranges agents in a DFS tree (agent ancestors are its parent and pseudoparents, agent descendants are its children and pseudochildren). Each agent holds a context, which is a set of assignations involving the agent's ancestors, and will be updated with message exchange. BnB-ADOPT uses three message types: VALUE(i, j, val, th) – i informs descendant j that i takes value val with threshold th– COST$(k, j, context, lb, ub)$ –k informs parent j that its bound are lb/ub in $context$– and TERMINATE(i, j) –i informs child j that i terminates–. A BnB-ADOPT agent executes the following loop: it reads and processes all incoming messages and takes value. Then, it sends a VALUE to each child or pseudochild and a COST to its parent. Here, we assume that the reader has some familiarity with BnB-ADOPT code.

BnB-ADOPT$^+$ [3] is a version of BnB-ADOPT that saves most of redundant messages, causing substantial reductions in communication costs with respect to the original algorithm. BnB-ADOPT$^+$ keeps the optimality and termination of BnB-ADOPT. Basically, it stores the last VALUE and COST messages sent to each destination, and it checks if some VALUE or COST messages to be sent at the current iteration might be redundant. If messages are found redundant, they are not sent in most of the cases.

3 Connecting Distributed Search with Soft Arc Consistency

Combining search with soft arc consistency brings substantial benefits to search performance. Taking BnB-ADOPT$^+$ as the distributed search algorithm, its combination with AC* and FDAC* soft arc consistency levels [4] has provided very good results. The resulting algorithms maintain BnB-ADOPT$^+$ optimality and termination, improving its performance: soft arc inconsistent values are removed from their domains, making smaller the search space, which causes substantial reductions in the search effort.

Initially, the proposed level of soft arc consistency is assured in a preprocess step, and during execution it is enforced every time soft arc consistency is broken. The AC*/FDAC* condition could be violated during execution if one of the following cases occur: a value is deleted (a support could be lost), a new $\top$ is found, or C_ϕ increments. AC*/FDAC* is enforced implementing the projection and extension operators for the distributed case. In the next two sections we summarize existing approaches to connect BnB-ADOPT$^+$ with AC* and FDAC* soft arc consistency levels [4]. In addition, we discuss its combination with EDAC*, the next soft arc consistency level.

3.1 Connecting BnB-ADOPT$^+$ with AC*

AC* requires that every value of every variable has a simple support on any other variable it is constrained with. AC* maintenance involves projecting binary costs into unary ones, and unary costs on C_ϕ.

As consequence of distributed search and AC* maintenance, some deletions may occur. In distributed search, NC* property is relaxed to allow equality with $\top$, $C_\phi + C_i(a) \leq \top$ because otherwise it could be pruned an optimal solution. In the centralized case this is not a problem, because each time a new solution is found it is stored. In the distributed case, however, this is not possible since there is no special agent holding the whole solution. So a value is pruned when its cost exceeds $\top$.

To propagate value deletions, and inform of $\top$ and C_ϕ some information has to be exchanged between agents: the subtree contribution of each agent to the C_ϕ ($subtreeContribution$), the global C_ϕ, and the global $\top$. This information is included in the original BnB-ADOPT$^+$ messages: VALUE and COST. A new message type is added: DEL to notify value deletions. The message structure appears in Figure 1.

It is assumed that cost functions are initially AC*. If not, they are made AC* by the preprocess that appears in Figure 2. The execution logic of the original BnB-ADOPT$^+$ algorithm remains mostly the same, with the following minor changes:

1. When $self$ receives a VALUE message, updates its local copy of $\top$ and C_ϕ. This values are later used in the NC* pruning. When $self$ sends VALUE messages, if $self = root$, it calculates the global C_ϕ as the sum of contributions from all its children pluds its own local contribution, and it is sent downwards.
2. When $self$ receives a COST message from a child c, it records the contribution of c to the global C_ϕ ($subtreeContribution$). Then, when $self$ sends a COST message, it calculates its own $subtreeContribution$ as the contributions of its children plus its own local contribution, and this is sent upwards.
3. After an agents reads and process all incoming messages, it checks if any of its values can be unconditionally deleted. If so, the value is deleted and AC* is reinforced. A value can be deleted either because it is found not NC*, or because its lower bound exceeds the $\top$ of the problem (to assure unconditional deletions the lower bound should aggregate only costs not involved with any higher agent).

In addition, in the current implementation:

- BnB-ADOPT$^+$ works with the original cost functions, while AC* is enforced on a copy of them. When a value is deleted, this is included in the original cost functions.
- There is a pre-preprocess to compute an initial $\top$ to pass to `AC*-preprocess(`$\top$`)`.

BnB-ADOPT$^+$:
VALUE($sender$, $destination$, $value$, $threshold$)
COST($sender$, $destination$, $context[]$, lb, ub)
STOP($sender$, $destination$)
BnB-ADOPT$^+$ with AC* maintenance:
VALUE($sender$, $destination$, $value$, $threshold$, $\top$, C_ϕ)
COST($sender$, $destination$, $context[]$, lb, ub, $subtreeContribution$)
STOP($sender$, $destination$, $emptydomain$)
DEL($sender$, $destination$, $value$)
BnB-ADOPT$^+$ with FDAC* maintenance: BnB-ADOPT$^+$with AC* maintenance plus
UCO($sender$, $destination$, $vectorOfExtensions$)

Fig. 1. Messages of original BnB-ADOPT$^+$ and BnB-ADOPT$^+$ maintaining AC* and FDAC*.

```
procedure AC*-preprocess(⊤)
  initialize;
  AC*-pre();
  while ¬end ∧ ¬quiescence do
    while input queue is not empty do
      msg ← getMsg();
      switch(msg.type)
        DEL: ProcessDelete(msg);
        STOP: ProcessStop(msg);
    PruneDomainSelf();

procedure AC*-pre()
  for each i ∈ neighbors(self) do
    if i < self then
      FromBinaryToUnary(self, i),
      FromBinaryToUnary(i, self),
    else
      FromBinaryToUnary(i, self),
      FromBinaryToUnary(self, i),
    FromUnarySelfToC_φ();

procedure FromBinaryToUnary(i, j)
  for each a ∈ D_i do
    v ← argmin_{b∈D_j} {C_ij(a, b)}; α ← C_ij(a, v);
    for each b ∈ D_j do C_ij(a, b) ← C_ij(a, b) − α;
    if i = self then C_i(a) ← C_i(a) + α;

procedure FromUnarySelfToC_φ()
  v ← argmin_{a∈D_self} {C_self(a)}; α ← C_self(v);
  myContribution ← myContribution + α;
  for each a ∈ D_self do C_self(a) ← C_self(a) − α;

procedure PruneDomainSelf()
  for each a ∈ D_self do if C_self(a) + C_φ > ⊤ then DeleteValue(a);

procedure DeleteValue(a)
  D_self ← D_self − {a};
  if D_self = ∅ then
    for each j ∈ neighbors(self) do sendMsg:(STOP, self, j, true);
    end ← true;
  else
    for each j ∈ neighbors(self) do
      sendMsg:(DEL, self, j, a);
      FromBinaryToUnary(j, self);
    FromUnarySelfToC_φ();
    if a = myValue then myValue ← argmin_{v∈D_self} LB(v);

procedure ProcessDelete(msg)
  D_sender ← D_sender − {msg.value};
  FromBinaryToUnary(self, msg.sender),
  FromUnarySelfToCφ(),

procedure ProcessStop(msg)
  if (msg.emptyDomain = true) then
    for each j ∈ neighbors(self), j ≠ sender do sendMsg(STOP, self, j, true);
  end ← true;
```

Fig. 2. AC* preprocess.

3.2 Connecting BnB-ADOPT+ with FDAC*

In the same way as section 3.1, BnB-ADOPT+ is connected with FDAC*. Given a variable ordering, FDAC* implies that every value of every variable has a simple support

```
procedure FDAC*-preprocess(⊤)
  initialize;
  AC*-pre();
  while ¬end ∧ ¬quiescence do
    while input queue is not empty do
      msg ← getMsg();
      switch(msg.type)
        DEL: ProcessDelete(msg);
        UCO: ProcessUnaryCosts(msg);
        STOP: ProcessStop(msg);
    PruneDomainSelf();
    DAC*-pre();

procedure DAC*-pre()
  for each i ∈ neighbors(self) do
    if (self < i)
      for each a ∈ D_i do P[a] ← min_{b∈D_self} {C_{i,self}(a, b) + C_self(b)};
      for each b ∈ D_j do E[b] ← max_{a∈D_i} {P[a] − C_{i,self}(a, b)};
    if E ≠ [0,...,0] then
      sendMsg(UCO, self, i, E);
      FromUnarySelfToBinary(i, E);
      FromBinaryToUnary(i, self);

procedure FromUnarySelfToBinary(i, vector)
  for each b ∈ D_self do
    for each a ∈ D_i do C_{i,self}(a, b) ← C_{i,self}(a, b) + vector[b];
    C_self(b) ← C_self(b) − vector[b];

procedure ProcessUnaryCosts(msg)
  for each b ∈ D_sender do
    for each a ∈ D_self do C_{self,sender}(a, b) ← C_{self,sender}(a, b) + msg.vector(b); /* extension */
  FromBinaryToUnary(self, sender);
  FromUnarySelfToCφ();
```

Fig. 3. FDAC* preprocess. Missing procedures appear in Figure 2.

with higher variables in the ordering (AC*) and a full support with lower variables in the ordering (DAC*). In addition to the projection operator, maintaining FDAC* requires the extension operator, which extends unary costs into binary ones.

The full supports requirement causes a new message type, the UCO message, that carries the unary costs that a lower variable is willing to offer to a higher variable. The structure of this new message appears in Figure 1. We assume that initial cost functions are FDAC* with respect to the ordering used. If not, they can be made FDAC* with the preprocess that appears in Figure 3, where first all cost functions are made AC* (both ways), and second they are made DAC* (one way: from the higher to the lower agent, the ordering is considered the same as the DFS tree). The order in which soft arc consistencies is done is relevant, because DAC* enforcing is prepared to respect previous AC* enforcing [5].

3.3 Maintaining EDAC*

In the distributed case, it is usually assumed that each agents knows its variable and the cost functions that it has with other agents. This second assumption implies that it knows the domain of the variables it is constrained with. To enforce soft arc consistencies higher than NC*, it is required that if agent i is constrained with agent j by C_{ij}, i

has to represent locally D_j. For privacy reasons, we assume that the unary costs of the values of an agent are held by itself, who knows them and updates them accordingly. An agent neither can know nor update unary costs of other agents.

Maintaining AC*/FDAC* during distributed search requires each agent to know the binary cost functions in which it is involved and the unary costs of its values. These requirements are in agreement with the privacy requirements not permitting an agent to know the unary costs of values of other agents. However when moving to EDAC* (the next soft arc consistency level) this privacy requirement is broken. EDAC* maintenance requires that at each variable there is a value with unary cost 0 which is fully supported in both directions (cost functions linking higher agents with $self$, cost functions linking $self$ with lower agents).

Let us consider two agents $i, j, i < j$ that share a cost function C_{ij}. To assure that j has a value fully supported by i, i has to extend some of its unary costs into the binary ones, which will be projected on the unary costs of j values. However, i will only extend its unary costs to j if it is sure that from this operation C_ϕ will increase (otherwise termination is not guaranteed). But this condition can only be assured if i knows the unary costs of j[1]. Therefore, aiming at EDAC* maintenance breaks the natural privacy requirements explained above, which represents a serious drawback in the distributed environment.

A possible way to partially avoid this issue, while enforcing some soft arc consistency that prunes more than FDAC*, comes from the following fact. Observe that the first variable in a FDAC* ordering satisfies the EDAC* property: for FDAC* each value has a simple/full support and there is a value with cost 0 (for NC*); since it is the first variable in the ordering, these supports have to be full supports. This suggests us an alternative way for the distributed setting: instead of having a single ordering of agents, we may have several orderings. On each ordering we enforce FDAC*, and the first variable of every ordering satisfies EDAC*. Next we show that having different representations and propagating deletions among them is legal and does not compromise the correctness of this idea.

4 Multiple Representations

It is known that with different variable orderings FDAC* maintenance may cause pruning different values (see the example at the end of the section). This fact motivates the present approach. It is unclear how to determine the *best ordering*, in the sense of the ordering that prunes most. Instead of looking for that ordering, we consider as alternative to keep *multiple orderings* $O_1, ..., O_r$ at each agent, on which FDAC* is separately enforced. Maintaining FDAC* in O_p may cause the deletion of value a of variable i: this deletion is propagated to all other orderings $O_1, ..., O_{p-1}, O_{p+1}, O_r$. Propagating value deletions among different orderings is legal as proved in the following proposition.

[1] This can be clearly seen in line 1 of function FindExistentialSupport of [2]. The expression of α involves $C_i(a)$ and $C_j(b)$, unary costs of values of x_i and x_j. While this causes no difficulties in a centralized approach, it becomes a real issue in a distributed setting.

Proposition 1. *Let us assume that enforcing FDAC* on the ordering O_1 causes to delete value (i, a), while enforcing FDAC* on the ordering O_2 causes to delete value (j, b). Then, both values can be deleted without losing any optimal solution.*

Proof. If enforcing FDAC* using ordering O_1 we delete value (i, a), this means that value a for variable i will not appear in any optimal solution of the problem. This fact derives directly from soft arc consistency, and it is independent of the ordering used. The same situation happens with ordering O_2 and value (j, b). Therefore, it is legal to remove both values independently of the ordering used. □

Since cost functions evolve depending on the ordering used, we prefer to talk about different representations instead of different orderings (clearly, each ordering defines a representation). Propagating value deletions between equivalent representations produce new equivalent representations, as proved next.

Proposition 2. *Let us assume that we have two equivalent representations R_1 and R_2 of the same problem, and enforcing FDAC* on R_1 causes to remove (i, a) producing a new representation R'_1. Removing (i, a) from R_2 we obtain a new representation R'_2. Then, R'_1 and R'_2 are equivalent.*

Proof. Let us call A the set of complete tuples with value a for variable i, and $CT(R)$ the set of complete tuples generated by representation R. Then, we have

$$CT(R'_1) = CT(R_1) \setminus A$$

$$CT(R'_2) = CT(R_2) \setminus A$$

Since R_1 and R_2 are equivalent representations, they have the same cost distributions on complete tuples. Therefore, it is direct to see that R'_1 and R'_2 are equivalent representations. □

In the distributed case, the idea of multiple representations can be included in BnB-ADOPT$^+$ producing the new BnB-ADOPT$^+$-FDAC*-MR algorithm. For single order FDAC* enforcing, we maintain a single copy of the cost functions in which we enforce FDAC*, following the order in which agents appear in the DFS tree branches. Implementing r representations requires each agent holding a set of cost functions $\{C_1, C_2...C_r\}$. On all r cost functions agents enforce FDAC*. The direction of the arc consistency enforcement will be defined by the set of partial orders $\{O_1, O_2, ..O_r\}$.

Orders are generated in the following way. Initially r different agents are selected, each of them will be the first agent in one of the r orders. Each agent chooses randomly a neighbor and sends a message containing the order (at this moment the order only contains the sender agent as the first agent). When this message arrive, if the receiver is not already in the order and the order is not complete, the receiver selects if it wants to be the next agent in the order. After this, the receiver chooses another neighbor randomly and sends the order. When the order is complete, it is informed to all agents in the DFS tree.

Having different orders produces different flows of costs and as result, some values may be found node-inconsistent in some representation. Then these values are deleted from all the representations. Every time there is a deletion, the agent will need to reinforce FDAC* over the r representations. For this agents will need to store:

1. One partial order for every representation r
2. One copy of the binary and unary cost functions for every representation r
3. One C_ϕ value for every representation r. Since different projections and extensions are performed on every representation, different C_ϕ values are obtained
4. All children $subtreeContribution$ to C_ϕ for every representation r. Since different projections and extensions are performed on every representation, agents will contribute to the C_ϕ differently on every one of them

The following changes in messages are needed to maintain the previous structures:

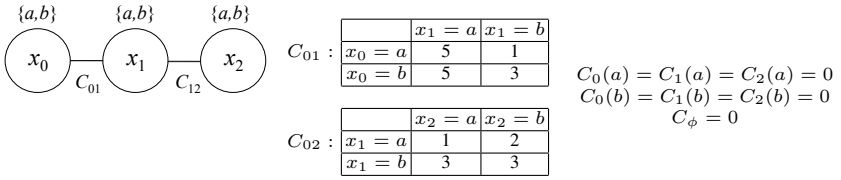
- VALUE: a vector C_ϕ[] is sent containing the C_ϕ values for every representation
- COST: a vector $subtreeContribution$[] is sent containing the subtree contribution to the C_ϕ for every representation
- UCO: a vector $vectorOfExtensions$[][] is sent containing the extensions for every representation

4.1 Example

Consider the problem in Figure 4 with $\top = 4$. If FDAC* is enforced with the order $\{x_0, x_1, x_2\}$, we get cost function C_{01} as displayed in Figure 5 (b). On the other hand, if FDAC* is enforced following the order $\{x_1, x_0, x_2\}$ value $(1, a)$ is found node inconsistent, it is deleted and we get cost function C_{01} as displayed in Figure 5 (c) and (d). Initially we do not know which is the best order to maintain FDAC*, but if we work with both orders maintaining two representations we will be able to prune inconsistent values in any of them.

For this problem, BnB-ADOPT$^+$ maintaining FDAC* with order $\{x_0, x_1, x_2\}$ requires 23 messages, while maintaining FDAC* with both orders $\{x_0, x_1, x_2\}$ and $\{x_1, x_0, x_2\}$ requieres 21 messages. We present a reduced execution trace of BnB-ADOPT+$^+$-FDAC* (Table 1, left) with order $\{x_0, x_1, x_2\}$ and BnB-ADOPT+$^+$-FDAC*-MR (Table 1, right) with both orders $\{x_0, x_1, x_2\}$ and $\{x_1, x_0, x_2\}$.

A short description of the execution follows. From lines 1 to 5 of Table 1 both algorithms behave in the same way: x_0 assigns value a and sends a VALUE message to x_1. Knowing that $x_0 = a$, x_1 assigns value b (best value with current information) and sends a VALUE message to x_2. Then, x_2 informs the cost of the current assignation to x_1, and finally x_1 sends a COST message to x_0. When this COST arrives to x_0, a complete solution has been found and $\top = 4$. x_0 changes value to b and sends a VALUE



C_{01} :

	$x_1 = a$	$x_1 = b$
$x_0 = a$	5	1
$x_0 = b$	5	3

C_{02} :

	$x_2 = a$	$x_2 = b$
$x_1 = a$	1	2
$x_1 = b$	3	3

$C_0(a) = C_1(a) = C_2(a) = 0$
$C_0(b) = C_1(b) = C_2(b) = 0$
$C_\phi = 0$

Fig. 4. Simple example with three variables and its initial binary, unary and zeroary cost functions.

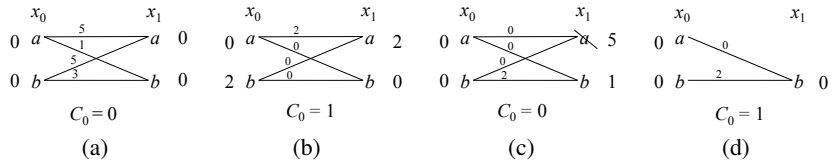


Fig. 5. Enforcing FDAC* on cost function C_{01}: (a) original instance (b) FDAC* with order $\{x_0, x_1, x_2\}$ (c) node inconsistent value with order $\{x_1, x_0, x_2\}$ (d) FDAC* in the same order.

message to x_1 that contains $\top$. Some other concurrent messages are sent, but we do not comment them because they are not relevant for the example.

From lines 6 to 9 algorithms behave differently. In BnB-ADOPT$^+$-FDAC*-MR (right), when x_0 VALUE message arrives, x_1 is able to delete value a with $\top = 4$, since this value is not node consistent in the order $\{x_1, x_0, x_2\}$. In lines 6-8 this value is deleted and neighbors are informed (notice that this deletion is not performed on BnB-ADOPT$^+$-FDAC* until lines 16-17). Now, x_1 can only take value b since value a has been deleted. In BnB-ADOPT$^+$-FDAC* (left), in line 9, agent x_1 takes value a since it

	BnB-ADOPT$^+$-FDAC*	BnB-ADOPT$^+$-FDAC*-MR
(1)	x_1 received VALUE: $x_0 = a$	x_1 received VALUE: $x_0 = a$
(2)	x_2 received VALUE: $x_1 = b$	x_2 received VALUE: $x_1 = b$
(3)	x_1 received COST: sender=x_2, UB=3 LB=3	x_1 received COST: sender=x_2, UB=3, LB=3
(4)	x_0 received COST: sender=x_1, UB=4 LB=4	x_0 received COST: sender=x_1, UB=4, LB=4
(5)	x_1 received VALUE: $x_0 = b$	x_1 received VALUE: $x_0 = b$
(6)		x_1 **delete value** a
(7)		x_2 **received DEL** $x_1 = a$
(8)		x_0 **received DEL** $x_1 = a$
(9)	x_2 **received VALUE:** $x_1 = a$	x_2 **received VALUE:** $x_1 = b$
(10)	x_0 received COST: sender=x_1, UB=6 LB=5	x_0 received COST: sender=x_1, UB=6, LB=6
(11)	x_0 delete value b	x_0 delete value b
(12)	x_1 **received COST: sender=x_2, UB=1 LB=1**	
(13)	x_1 received DEL $x_0 = b$	x_1 received DEL $x_0 = b$
(14)	x_1 received VALUE: $x_0 = a$	x_1 received VALUE $x_0 = a$
(15)	x_1 received STOP	x_1 received STOP
(16)	x_1 **delete value** a	
(17)	x_2 **received DEL** $x_1 = a$	
(1 8)	x_2 **received VALUE:** $x_1 = b$	
(19)	x_2 received STOP	x_2 received STOP
	No more messages...	No more messages...
	23 total messages	21 total messages
	9 VALUE msg , 5 COST msg, 3 DEL msg	8 VALUE msg, 4 COST msg, 3 DEL msg
	5 cycles	4 cycles
	TOTAL cost: 4	TOTAL cost: 4
	OPT. SOLUTION: $x_0 = a$; $x_1 = b$; $x_2 = a$	OPT. SOLUTION: $x_0 = a$; $x_1 = b$; $x_2 = a$

Table 1. Summary of BnB-ADOPT$^+$-FDAC* and BnB-ADOPT$^+$-FDAC*-MR execution on the example of Figure 4.

is the one that minimize the LB for the current context (LB(a) = 5 and LB(b) = 6). This VALUE message will generate the corresponding COST message of line 12.

In lines 10-11 of both algorithms, x_0 receives a COST message that inform the cost of the assignation $x_0 = b$ and decides to delete its value b. The only possible value for x_0 is now a, so x_0 assigns it and terminates. From line 13 to 15, x_1 receives DEL, VALUE and STOP messages from x_0. From line 16 to 18 of BnB-ADOPT$^+$-FDAC* (left) x_1 deletes value a and assigns value b. Finally on line 19 of both algorithms, x_2 receives a STOP message and execution is concluded.

Notice that, by detecting a deletion sooner (lines 6-8 for BnB-ADOPT$^+$-FDAC*-MR, and lines 16-18 in BnB-ADOPT$^+$-FDAC*) it is possible to end the execution with 2 messages less and 1 cycle less. This 2 messages saved are the ones represented in lines 9 and 12 for BnB-ADOPT$^+$-FDAC* (left).

5 Experimental Results

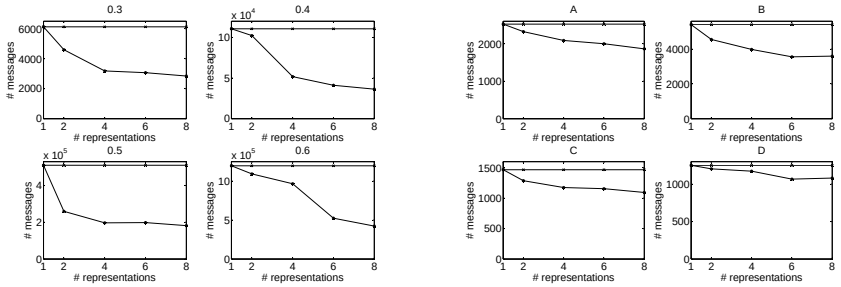
We evaluate the efficiency of BnB-ADOPT$^+$-FDAC*-MR (multiple representations) with respect to BnB-ADOPT$^+$-FDAC* (single representation) on unstructured instances with binary random DCOPs, and on structured distributed meeting scheduling.

Binary random DCOP are characterized by $\langle n, d, p_1 \rangle$, where n is the number of variables, d is the domain size and p_1 is the network connectivity. We have generated random DCOP instances: $\langle n = 10, d = 10, p_1 = 0.3, 0.4, 0.5, 0.6 \rangle$. Costs are selected from an uniform cost distribution. Two types of binary cost functions are used, small and large. Small cost functions extract costs from the set $\{0, \ldots, 10\}$ while large ones extract costs from the set $\{0, \ldots, 1000\}$. The proportion of large cost functions is 1/4 of the total cost functions (this is done to introduce some variability among tuple costs).

On the meeting scheduling formulation, variables represent meetings, domain represent time slot assigned for each meeting, and there are constraints between meetings that share participants. We present 4 cases obtained from the DCOP repository [11] with different hierarchical scenarios and domain 10: case A (8 variables), case B (10 variables), case C (12 variables) and case D (12 variables).

Figure 6 (a) and (b) shows experimental results for meeting scheduling and random problems averaged over 30 and 50 instances respectively, with a number of representations from 2 to 8. For an easy comparison, BnB-ADOPT$^+$-FDAC* results are drawn as an horizontal line. On random DCOPs, BnB-ADOPT$^+$-FDAC*-MR showed clear benefits on communication costs with respect to BnB-ADOPT$^+$-FDAC*. Maintaining from 4 to 6 representations, the number of exchanged messages is divided by a factor of at least 2. Also, the number of cycles required to reach the solution is divided by a factor from 2 to 3. For meeting scheduling instances we also observe a decrement in the number of cycles and messages exchanged, although to a smaller extent. In Figure 6 we observe that benefits in communication are unevenly distributed: the #saved messages/#representations ratio is higher in the left-half of the plots. This suggests that $n/2$ could be a good number of representations. More work is needed to substantiate this conjecture. Table 2 shows the details of the experiments maintaining 6 representations.

Assuming that processing each message type requires approximately the same time, a decrement in cycles combined with a decrement in the number of messages per cycle



(a) Random Instances (b) Meeting Scheduling

Fig. 6. Exchanged messages (y axis) when solving random instances (left) and meeting scheduling (right) with an increasing number of representations (x axis).

is an improvement indicator. Since agents need to process less information coming from their neighbors on each iteration, and they perform less iterations to reach the optimum, this combined reduction is beneficial for agent performance.

Notice that maintaining FDAC* on multiple representations has produced only a few extra DEL and UCO messages. In the case of DEL messages, this slight increment is because more deletions have been produced. In the case of UCO messages, the increment is because UCO messages are only sent if their $costOfExtension$ vector is different from zero. As we are maintaining several representations with different orders, is more probable that the extensions will be different from zero in any of the 6 representations.

The number of NCCCs increases since more projection and extensions are needed to maintain FDAC* on all representations. However, observe that this increment is not linear with respect to the number of representations maintained, it is smoothed by the fact that less messages are generated and more deletions are performed. So there are messages on the BnB-ADOPT$^+$-FDAC* algorithm that will not be needed to process with multiple representations, and also there are values that will not be needed to assign or to check for node consistency.

In general, by including few DEL and UCO messages and performing extra local computation to enforce FDAC* on multiple representations, BnB-ADOPT$^+$-FDAC*-MR is able to obtain important savings in communication. We assume the usual case where communication time is higher than computation time, then the total elapsed time is dominated by communication time. The flows of costs from one agent to another, implemented by UCO messages, allows an agent to pass some of their unitary costs to higher agents following different orders. This accumulations of costs on higher agents following different orders brings more pruning opportunities. In general, it is expected that the combination of multiple enforcements will be able to extend the inference benefits.

(a) Random DCOPs

p_1	#Msgs	#VALUE	#COST	#DEL	#UCO	#Cycles	#NCCC	#Deletions
	6,128	2,795	3,047	**230**	**28**	1,039	**519,112**	80
	3,068	**1,335**	**1,391**	245	69	**480**	1,778,525	**86**
	110,696	48,281	62046	**288**	**53**	17,937	**9,910,897**	78
0.4	**41,147**	**19,309**	**21,357**	311	142	**5,561**	22,063,034	**85**
	510,411	225,155	284,781	**366**	**82**	85,710	**121,453,697**	78
0.5	**198,474**	**91,506**	**106,299**	397	244	**30,659**	318,565,730	**85**
	1196935	475,416	720,975	**408**	**108**	199,971	**470,462,443**	74
0.6	**524,406**	**209,150**	**314,454**	459	314	**87,357**	1,217,511,858	**84**

(b) Distributed Meeting Scheduling

	#Msgs	#VALUE	#COST	#DEL	#UCO	#Cycles	#NCCC	#Deletions
	2,524	1,056	1,240	**200**	**5**	462	**382,676**	49
A	**2,001**	**820**	**921**	216	9	**329**	1,762,278	**53**
	5,405	2,323	2,863	**184**	**7**	1,080	**659,314**	53
B	**3,556**	**1,513**	**1,7821**	210	23	**650**	2,487,359	**61**
	1,467	697	505	**225**	**6**	125	**71,439**	80
C	**1,156**	**509**	**353**	238	21	**83**	352,795	**85**
	1,251	526	448	**234**	**8**	132	**56,447**	83
D	**1,067**	**423**	**345**	241	24	**98**	327,749	**85**

Table 2. Experimental results of BnB-ADOPT$^+$-FDAC* (first row) compared to BnB-ADOPT$^+$-FDAC*-MR (second row) maintaining 6 representations.

6 Conclusion

Maintaining soft arc consistency during distributed search has proved to be beneficial for performance with AC* and FDAC* levels. To assure FDAC* and EDAC*, a partial order among variable is needed. In this paper we discuss connecting BnB-ADOPT$^+$ with the next soft arc consistency level EDAC*. It turns out that to maintain EDAC* an agent needs to know the unary cost of neighboring agents, which violates usual privacy requirements.

To avoid this issue, we propose to maintain multiple orders among variables, and the same number of cost functions on wich we enforce FDAC* following the corresponding order. It is known that with different orderings FDAC* maintenance may detect different node inconsistent values, however if we maintain several orders we are able to detect the node inconsistent values on every order and delete them, propagating this deletions to other orders. Propagating deletions among different orders is legal and furthermore, maintaining FDAC* on multiple representations assures EDAC* on the first variable of the order. Experimental results have shown benefits in terms of number of cycles and messages exchanged.

References

1. B.M.W. Cheng, K.M.F. Choi, J.H.M. Lee, and J.C.K. Wu. Increasing constraint propagation by redundant modelling: an experience report. *Constraints*, (4):167–192, 1999.
2. S. de Givry, F. Heras, J. Larrosa, and M. Zytnicki. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. *Proc. of IJCAI-05*, pages 84–89, 2005.
3. P. Gutierrez and P. Meseguer. Saving messages in BnB-ADOPT. *Proc. AAAI-10*, 2010.

4. P. Gutierrez and P. Meseguer. BnB-ADOPT$^+$ with several soft arc consistency levels. *ECAI*, 2010, submitted.
5. J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted CSP. *Proc. of IJCAI-03*, pages 239–244, 2003.
6. P. Meseguer, F. Rossi, and T. Schiex. *Handbook of Constraint Programming. Chapter 9, Soft Constraints.* Elsevier, 2006.
7. P. J. Modi, W.M. Shen, M. Tambe, and M. Yokoo. Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005.
8. A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. *Proc. of IJCAI-05*, pages 266–271, 2005.
9. M. Silaghi and M. Yokoo. Nogood-based asynchronous distributed optimization (ADOPT-ng). *Proc. of AAMAS-06*, pages 1389–1396, 2006.
10. W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Proc. of AAMAS-08*, pages 591–598, 2008.
11. Z. Yin. USC dcop repository. Meeting scheduling and sensor net datasets, http://teamcore.usc.edu/dcop, 2008.