*ECAI 2010*
*H. Coelho et al. (Eds.)*
*IOS Press, 2010*

*doi:10.3233/978-1-60750-606-5-67*



# BnB-ADOPT$^+$ with Several Soft Arc Consistency Levels

**Patricia Gutierrez** and **Pedro Meseguer** [1]

**Abstract.** Distributed constraint optimization problems can be solved by BnB-ADOPT$^+$, a distributed asynchronous search algorithm. In the centralized case, local consistency techniques applied to constraint optimization have been shown very beneficial to increase performance. In this paper, we combine BnB-ADOPT$^+$ with different levels of soft arc consistency, propagating unconditional deletions caused by either the enforced local consistency or by distributed search. The new algorithm maintains BnB-ADOPT$^+$ optimality and termination. In practice, this approach decreases substantially BnB-ADOPT$^+$ requirements in communication cost and computation effort when solving commonly used benchmarks.

## 1 INTRODUCTION

There is an increasing interest in solving *constraint optimization problems* (COP) in a distributed form. Often it occurs that different problem elements are distributed among autonomous agents, and they cannot be grouped into a single agent for privacy or for other reasons (for example, consider distributed meeting scheduling [7] or sensor networks applications [1]). In this case, we talk about *distributed COP* (DCOP). To solve them, distributed algorithms are needed, to achieve an optimal solution without joining all problem elements into a single agent. Since they are based on message passing, communication costs have to be included when evaluating them.

ADOPT [6] is an asynchronous distributed search algorithm for DCOP solving. It has been improved in BnB-ADOPT [8], which changed the original best-first strategy for depth-first, obtaining better performance. This algorithm has also been improved removing some redundant messages in BnB-ADOPT$^+$ [2], which is currently one of the most performant asynchronous distributed search algorithms for DCOP solving to optimality.

In the centralized case, COPs are often formulated using soft constrains [5]. The standard search solving algorithm is *branch-and-bound* (BnB). Maintaining some local consistency on soft constraints during BnB search causes substantial improvements in performance [4, 3]. Taking inspiration from this fact, we have explored local consistency maintenance of soft constraints when solving DCOPs. Notice that local consistencies are conceptually equal in the centralized/distributed cases. However, maintaining local consistencies during distributed search requires different techniques than in the centralized case, where all problem elements are available to the single agent performing the search. Maintaining local consistencies keeps the optimality and termination of asynchronous distributed search.

Specifically, we have taken BnB-ADOPT$^+$ as asynchronous distributed search algorithm, on top of which we maintain AC$^*$ and FDAC$^*$ versions of soft arc consistency. Then, we present the new algorithms BnB-ADOPT$^+$-AC$^*$ and BnB-ADOPT$^+$-FDAC$^*$. They achieve spectacular reductions in communication and computation if compared with the original BnB-ADOPT$^+$ on several benchmarks.

This paper is organized as follows. In section 2 we telegraphically describe the concepts used in the rest of the paper (we assume some familiarity with BnB-ADOPT and soft arc consistency versions). We present our approach in section 3, discussing some differences with the centralized case. We introduce the new algorithms with some detail in section 4, and their experimental evaluation in section 5. Finally, we conclude in section 6.

## 2 PRELIMINARIES

**COP.** A binary *Constraint Optimization Problem* (COP) is defined by $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of variables; $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a collection of finite domains; $D_i$ is the initial domain of $x_i$; $\mathcal{C}$ is a set of unary and binary soft constraints represented as cost functions; $C_{ij} \in \mathcal{C}$ specifies the cost of every combination of values of $var(C_{ij}) = (x_i, x_j)$, $C_{ij} : D_i \times D_j \mapsto N \cup \{0, \infty\}$. The cost of a complete tuple is the addition of all individual cost functions evaluated on that particular tuple. This definition assumes the weighted model of soft constraints [5]. An *optimal solution* is a complete tuple with minimum cost.

**Soft Arc Consistency.** Let be a binary COP: $(i, a)$ means $x_i$ 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, $C_\phi$ is a zero-ary cost function that represents a necessary global cost of any complete assignment. As [3], we consider the following local consistencies (variables are totally 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 $a \in D_i$ s.t. $C_i(a) = 0$; a COP is NC$^*$ if every variable is NC$^*$.
- *Arc consistency*$^*$: $(i, a)$ is arc consistency (AC) wrt. cost function $C_{ij}$ if there is $b \in D_j$ s.t. $C_{ij}(a, b) = 0$; $b$ is a *support* of $a$; $x_i$ is AC if all its values are AC wrt. 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) wrt. cost function $C_{ij}$, $j > i$, if there is $b \in D_j$ 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 wrt. 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$^*$.

AC$^*$/DAC$^*$ can be reached forcing supports/full supports to NC$^*$ values and pruning values not NC$^*$. 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 $C_\phi$. 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 pro-

[1] IIIA, CSIC, Campus UAB, 08193 Bellaterra, Spain. {patricia|pedro}@iiia.csic.es

jection over the value. The systematic application of these operations does not change the optimum cost and maintains an optimal solution [3]. 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 support/full support of a value of a neighbor variable. So, a deleted value in one variable might cause further deletions in other variables. The AC*/DAC* check must be performed until no further values are deleted.

**DCOP.** A *Distributed Constraint Optimization Problem* (DCOP) is defined by $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \alpha)$, where $\mathcal{X}, \mathcal{D}, \mathcal{C}$ 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 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, for any pair of agents.

**BnB-ADOPT**. BnB-ADOPT [8] is a reference algorithm for DCOP. It is a depth-first version of ADOPT [6], showing a better performance. As ADOPT, it arranges agents in a DFS tree. Each agent holds a context, which is a set of assignments involving the agent's ancestors, and will be updated with message exchange. Messages are VALUE$(i, j, val, th)$, $–i$ informs child or pseudochild $j$ that it has taken value $val$ with threshold $th$–, COST$(k, j, context, lb, ub)$ $–k$ informs parent $j$ that with $context$ its bound are $lb$ and $ub$–, and TERMINATE$(i, j)$, $–i$ informs child $j$ that 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.

**BnB-ADOPT$^+$**. BnB-ADOPT$^+$ [2] is a version of BnB-ADOPT that saves most of redundant VALUE and COST messages, keeping optimality and termination. BnB-ADOPT$^+$ causes substantial reductions in communication costs with respect to BnB-ADOPT.

## 3 BnB-ADOPT$^+$ + SOFT ARC CONSISTENCY

Here we present our contribution combining distributed search (BnB-ADOPT$^+$) and maintaining some kind of soft arc consistency for DCOP solving. Due to the distributed setting this combination requires some care. In a naive approach, each time an agent needs information of other agent this would generate two messages (request and response) which could cause a serious degradation in performance. In our approach, we try to keep the number of exchanged messages as low as possible, introducing the required elements to enforce the selected soft arc consistency in existing BnB-ADOPT$^+$ messages, keeping their meanings for distributed search.

Let us consider a DCOP instance, where agents are arranged in a DFS tree and each agent executes BnB-ADOPT$^+$. Let us consider a generic agent $self$ that takes value $v$. After sending VALUE messages, $self$ receives COST messages from its children. A COST message contains the lower bound computed by BnB-ADOPT$^+$, with the context (variable, value) pairs on which this lower bound was computed. We consider COST messages whose context is simply the $self$ agent with its actual value $v$. If the sum of the lower bounds of these COST messages exceeds $\top$ (the lowest unacceptable cost), the value $v$ of $self$ can be deleted. To see this, it is enough to realize that the lower bound is computed assuming (variable, value) pairs of context: if this is simply $(self, v)$, the actual cost of $v$ does not depend on the value of any other agent, so if it exceeds $\top$ it can be deleted.

This reasoning is valid for any agent. In addition, some extra pruning can be done at the agent located at the $root$ of the DFS tree ($D_{root} = \{a, b, \ldots\}$). Let us assume that initially $root$ takes value $a$. After a while, $root$ knows $cost(a) = lb(a) = ub(a) = \top_1$, and it decides to change its assignment to $b$. After exchanging some messages, $root$ knows $cost(b) = lb(b) = ub(b) = \top_2$. If $\top_1 > \top_2$ then value $a$ can be removed from $D_{root}$ because $cost(a) > cost(b)$. Just removing $a$ will cause no effect in BnB-ADOPT$^+$, because it will not consider $a$ again as possible value for $root$. However, if we inform constrained agents that $a$ is no longer in $D_{root}$, this may cause some values of other agents to become unfeasible so they can be deleted.

In these two cases, deletions are unconditional because they do not depend on values of other agents. These deletions can be further propagated in the same way, decreasing the size of the search space. Any deletion caused by propagation of unconditional deletions is also unconditional. To propagate these deletions to other agents we need to maintain some kind of soft arc consistency during search.

Maintaining soft arc consistency in the distributed case has some differences with the centralized case. They are summarized next:

- *Pruning condition.* In the centralized case, a value $a \in D_i$ can be removed if it is not NC*, that is, if $C_i(a) + C_\phi \geq \top$. However, in the distributed case $a$ can be removed only if $C_i(a) + C_\phi > \top$, as explained in the following. In both cases, $\top$ is an upper bound ($\geq$) of the optimum cost. In the distributed case, BnB-ADOPT$^+$ terminates leaving at each agent an assignment that belongs to a solution with the optimum cost (optimal solution). Pruned values will not be in their domains when BnB-ADOPT$^+$ terminates. If we prune a value when its cost equals $\top$, we might remove a value that belongs to an optimal solution. For this reason, we can

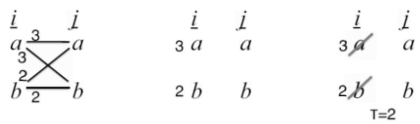


**Figure 1.** Left: Simple example with two agents $i$ and $j$ and two values per variable. Binary costs are indicated, unary costs are zero. The optimum cost is 2 and there are two optimal solutions $(i, b)(j, a)$ and $(i, b)(j, b)$. Center: $i$ projects $C_{ij}$ on its unary costs. No link between two values of different agents means a zero binary cost. Right: if $\top = 2$, pruning $v$ with $cost(v) = \top$ causes to lose value $(i, b)$ which is part of the two optimal solutions. In fact, no value remains for $i$.

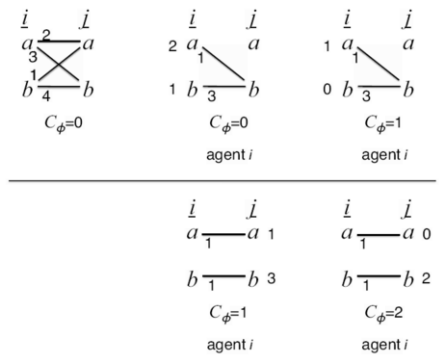


**Figure 2.** (UP) Left: Simple example with two agents $i$ and $j$ and two values per variable. Center: $i$ projects $C_{ij}$ on its unary costs. Right: $i$ projects unary costs on $C_\phi$. (DOWN) Center: $j$ projects $C_{ij}$ on its unary costs, without considering previous projection of $i$ (this is incorrect). Right: $j$ projects unary costs on $C_\phi$, causing an incorrect increment.

**BnB-ADOPT$^+$ messages:**
VALUE($sender$, $destination$, $value$, $threshold$)
COST($sender$, $destination$, $context[]$, $lb$, $ub$)
STOP($sender$, $destination$)
**BnB-ADOPT$^+$-AC$^*$ messages:**
VALUE($sender$, $destination$, $value$, $threshold$, $\top$, $C_\phi$)
COST($sender$, $destination$, $context[]$, $lb$, $ub$, $subtreeContr$)
STOP($sender$, $destination$, $emptydomain$)
DEL($sender$, $destination$, $value$)
**BnB-ADOPT$^+$-FDAC* messages:** those of BnB-ADOPT$^+$-AC$^*$ plus
UCO($sender$, $destination$, $vectorOfExtensions$)

**Figure 3.** Messages of BnB-ADOPT$^+$, BnB-ADOPT$^+$-AC$^*$ and BnB-ADOPT$^+$-FDAC$^*$.

only prune when the value cost exceeds $\top$. An example appears in Figure 1 (in the centralized case, the only agent executing the solving procedure stores the complete "best solution" found as search progresses; a value of the optimal solution can be pruned from its domain, because that solution was stored somewhere; when the algorithm terminates, that solution will be recalled).

- *Legal representation of cost functions.* In the centralized case, all cost functions are known and manipulated by a single agent, the one in charge of COP solving. This agent keeps a single copy of each cost function, where every update is accumulated. In the distributed case, a cost function $C_{ij}$ between agents $i$ and $j$ is known by both agents, which initially share the same representation of $C_{ij}$. Operations to maintain soft arc consistency modify this representation. Since each agent operates differently, after a while agents could have a different representation of $C_{ij}$. Both agents must maintain a *legal representation* of $C_{ij}$ during the soft arc consistency operations. Otherwise, the same cost can be counted twice when projecting unary costs on $C_\phi$, as shown in Figure 2, causing $C_i(a) + C_\phi$ to become an invalid lower bound for $a$. To maintain a legal representation, $i$ has to simulate the action of $j$ on its $C_{ij}$ representation, and vice versa. In some cases, $i$ has also to send a message to $j$.

In the distributed case, it is usually assumed that each agent knows about (i) its variable and (ii) the cost functions it has with other agents. Assumption (ii) implies that it also knows about the domain of variables it is constrained with (assuming that cost functions do not contain irrelevant values). To enforce any soft arc consistency, we explicitely require that if agent $i$ is connected 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 according the local consistency enforced. An agent neither can know nor update unary costs of other agents. Some soft arc consistencies require that agents have to be ordered. We take the order of agents in each branch of the DFS tree used by BnB-ADOPT$^+$. Observe that, although it is not a total order, agents in separate branches do not share cost functions, so for enforcing soft arc consistency it is enough with the ordering that agents have in DFS branches.

## 4 BnB-ADOPT$^+$ AND AC$^*$/FDAC$^*$

Distributed search can cause unconditional value deletions. These value deletions can be propagated maintaining soft arc consistency during distributed search. This idea can be easily included in BnB-ADOPT$^+$. Since there are several soft arc consistencies, this approach generates new algorithms depending on the selected soft arc consistency to be maintained. Here we present the connection of BnB-ADOPT$^+$ with AC$^*$ and FDAC$^*$. It is not difficult to prove than, no matter maintaining AC$^*$ or FDAC$^*$, the new algorithms keep the optimality and termination properties of BnB-ADOPT [8].

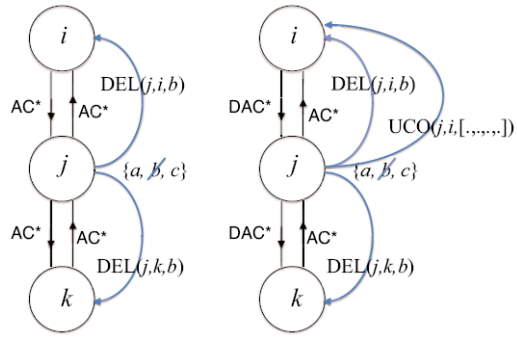


**Figure 4.** Three agents $i, j, k$ in the same branch of the DFS tree. (Left) Maintaining AC$^*$: Cost functions are AC$^*$ in both senses; deleting value $b$ in $D_j$ causes to send two DEL messages to $i$ and $k$ to restore AC$^*$. (Right) Maintaining FDAC$^*$: Cost functions are FDAC$^*$ (DAC$^*$ in one sense and AC$^*$ in the other); deleting value $b$ in $D_j$ causes to send two DEL messages to $i$ and $k$ to restore AC$^*$, plus one UCO message to the higher agent $i$ to restore DAC$^*$.

### 4.1 BnB-ADOPT$^+$-AC$^*$

BnB-ADOPT$^+$-AC$^*$ performs distributed search and maintains AC$^*$ level of soft arc consistency. If $i$ and $j$ are two neighbor agents, $i < j$, AC$^*$ is maintained from $i$ to $j$ and from $j$ to $i$, as shown in Figure 4 (left). Communication between agents is done by message passing. The semantic of original BnB-ADOPT$^+$ messages remains unchanged. New elements are included in these messages, they appear in Figure 3. BnB-ADOPT$^+$-AC* requires some minor changes with respect to BnB-ADOPT$^+$:

- A new message type, DEL, is required. When $self$ deletes value $a$ in $D_{self}$, it sends a DEL message to every agent constrained with it. This is depicted in Figure 4 (left). When $self$ receives a DEL message, it registers that the message value has been deleted from the domain of sender, and it enforces AC* on the constraint between $self$ and sender. If, as result of this enforcing, some value is deleted in $D_{self}$, it is propagated.
- VALUE messages include $\top$ and $C_\phi$. The initial $\top$ is passed as parameter and $root$ propagates it downwards, informing the agents of the lowest unacceptable cost. As search progresses, $root$ may discover lower values for $\top$, which are propagated in the same way. Contributions to $C_\phi$ are propagated upwards in COST messages and aggregated in $root$, building $C_\phi$, a lower bound of the instance global cost (no matter which values are assigned). Then, $root$ propagates $C_\phi$ downwards in VALUE messages.
- COST messages include the subtree contribution of each agent to the global $C_\phi$. Each agent adds its own contribution with the subtree contributions of all its children, and the result is included in the next COST message sent to its parent. All these contributions are finally added in $root$, forming the global $C_\phi$, which is propagated downwards in VALUE messages.

```
procedure AC*-preprocess(⊤)
  initialize;
  AC*();
  while ¬end ∧ ¬quiescence do
    msg ← getMsg();
    switch(msg.type)
      DEL: ProcessDelete(msg);    STOP: ProcessStop(msg);

procedure AC*()
  for each i ∈ neighbors(self) do
    if i < self then
      AC*-one-way(self, i);
1     AC*-one-way(i, self);
    else
      AC*-one-way(i, self);
2     AC*-one-way(self, i);

procedure AC*-one-way(i, j); /* after execution, AC* from i to j holds */
  FromBinaryToUnary(i, j),
  if i = self then
    PruneDomainSelf();
    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);
3     AC*-one-way(j, self);
    FromUnarySelfToC_φ();
    if a = myValue then myValue ← argmin_{v∈D_self} LB(v);

procedure ProcessDelete(msg)
  D_sender ← D_sender − {msg.value};
4 AC*-one-way(self, sender);

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

**Figure 5.** The preprocess code for enforcing AC*.

We assume that cost functions are initially AC*. If not, they are made AC* by preprocess of Figure 5. A quick description follows:

- AC-preprocess*. It receives the initial $\top$ and performs AC*. Then, it performs a receiving loop of DEL or STOP messages that ends when an empty domain has been detected ($end$ is $true$) or when there are no more messages ($quiescence$ is $true$).
- AC*(). For each binary cost function in which $self$ is involved, it enforces AC* with the following assumption: it projects first on the lower agent and then on the higher agent. It is worth noting that executing AC*-one-way($i, self$) does not change unary costs of $self$ values, but modifies the representation of $C_{i,self}$ in $self$ in the same way agent $i$ does.
- AC*-one-way($i, j$). It enforces AC* property from $i$ to $j$.
- FromBinaryToUnary($i, j$). It projects binary costs $C_{ij}$ on unary costs. It updates unary costs when the first argument is $self$.
- FromUnarySelfToC$_\phi$(). It projects $self$ unary costs on $myContribution$, which accumulates $self$ contribution to $C_\phi$.
- PruneDomainSelf(). Checks for deletion every value in $D_{self}$.

```
1  DAC*-one-way(i);
2  do nothing /* */
3  if j < self then DAC*-one-way(j); else AC*-one-way(j, self);
4  if self > sender AC*-one-way(self, sender);

procedure DAC*-one-way(i)
  P[a] ← min_{b∈D_self}{C_i,self(a, b) + C_self(b)};
  E[b] ← max_{a∈D_i}{P[a] − C_i,self(a, b)};
  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);
  PruneDomainSelf();
  FromUnarySelfToC_φ();
  for each i ∈ neighbors(self) do
    if i < self then DAC*-one-way(self, i);
```

**Figure 6.** Replacing lines **1, 2, 3, 4** of Figure 5 for the ones indicated here, we obtain the preprocess code for enforcing FDAC*. When a UCO message arrives, ProcessUnaryCosts($msg$) is called.

- DeleteValue($a$). $self$ removes value $a$ from $D_{self}$. If $D_{self} = \emptyset$, there is no acceptable solution, so $self$ sends STOP messages to all its neighbors, indicating that the process terminates. Otherwise, for all neighbors $j$, a DEL message is sent notifying $a$ deletion and AC*-one-way($j, self$) is executed. Observe that this causes no change in $self$ unary costs, which are projected on $C_\phi$. If the deleted value was the current value, a new value is selected.
- ProcessDelete($msg$). $self$ received a DEL message: $sender$ has deleted value $a$ from $D_{sender}$. $self$ registers this in its $D_{sender}$ copy and enforces AC* from $self$ to $sender$.
- ProcessStop(). $self$ received a STOP message. If caused by an empty domain, $self$ resends the STOP message to all its neighbors, except $sender$. In any case, $self$ records its reception in $end$.

The BnB-ADOPT$^+$-AC* process code is not given here for space reasons. It is based on BnB-ADOPT$^+$[2]. In addition to the normal BnB-ADOPT$^+$ operation, it includes the following actions to maintain AC*. When $self$ receives a VALUE message, the local copies of $\top$ and $C_\phi$ are updated if the values contained in the received message are better (lower for $\top$, higher for $C_\phi$). If $\top$ or $C_\phi$ changed, $D_{self}$ is tested for possible deletions (because elements of the deletion condition have changed). When $self$ receives a COST message from a child $c$, $self$ records $c$ subtree contribution to $C_\phi$. In the Backtrack procedure, when $self$ changes value, $D_{self}$ is tested for possible deletions. When $self$ receives a DEL message, the procedure ProcessDelete($msg$) that appears in Figure 5 is called.

## 4.2 BnB-ADOPT$^+$-FDAC*

BnB-ADOPT$^+$-FDAC* performs distributed search and maintains FDAC* level of soft arc consistency. If $i$ and $j$ are two neighbor agents, $i < j$, DAC* is maintained from $i$ to $j$ and AC* from $j$ to $i$, as shown in Figure 4 (right). As indicated in Figure 3, in addition to the messages required for BnB-ADOPT$^+$-AC*, it requires the new UCO (unary costs) message. When $self$ enforces DAC* on a cost function with a higher agent $i$, $self$ sends a UCO message to $i$ with

the minimum contribution of $self$ unary costs for $i$ to project on $i$ unary costs (following [3]). This is depicted in Figure 4 (right). It is worth noting that this DAC$^*$ enforcing does not eliminates previous AC$^*$ enforcing on the same pair of agents, theorem 2 of [3]; we always enforce AC$^*$ before enforcing DAC$^*$ on $C_{ij}$. The vector of extensions is the $E[b]$ computed in the procedure DAC$^*$-one-way$(i)$ in Figure 6. Upon reception, the $i$ agent will perform the extension of these unary costs into the binary cost function, the projection of the binary costs into the unary ones and these on $C_\phi$, checking its domain for possible deletions and restoring the DAC$^*$ condition from $i$ towards higher neighbors.

We assume that cost functions are initially FDAC$^*$. If not, they can be made FDAC$^*$ by the preprocess depicted in Figure 6, where lines **1, 2, 3, 4** replace the corresponding ones in Figure 5. A summary description of this code follows:

- **1** Instead of AC$^*$, $self$ enforces DAC$^*$ with higher agent $i$.
- **2** $self$ does nothing because to enforce DAC$^*$ with a lower agent, $self$ has to wait for the UCO message.
- **3** $self$ enforces either AC$^*$ or DAC$^*$, depending on the relative order between $j$ and $self$.
- **4** $self$ enforces AC$^*$ with the higher agent $sender$.
- DAC$^*$-one-way$(i)$. $self$ starts enforcing DAC$^*$ on $C_{i,self}$ by performing the required operations on its representation of $C_{i,self}$ and sending a UCO message to $i$.
- FromUnarySelfToBinary$(i, vector)$. $self$ adds in $C_{i,self}$ the costs in $vector$ that will be sent to $i$, subtracting them from $C_{self}$ unary costs.
- ProcessUnaryCosts$(msg)$. $self$ receives the UCO message and extends its costs into $C_{self,sender}$. It projects costs from $C_{self,sender}$ on its unary costs and these on $C_\phi$. $self$ tries to prune its domain and enforces DAC$^*$ with any other higher agent $i$ constrained with it.

The BnB-ADOPT$^+$-FDAC$^*$ process code is not given here for space reasons. Basically it is the BnB-ADOPT$^+$-AC$^*$ code, plus the reception and process of the new UCO message. This process is done by ProcessUnaryCosts$(msg)$ that appears in Figure 6.

## 5 EXPERIMENTAL RESULTS

We evaluate the efficiency of BnB-ADOPT$^+$-AC*/FDAC* by a discrete event simulator. Performance is evaluated in terms of communication cost (messages exchanged), computation effort (non-concurrent constraint checks), considering also the number of iterations (synchronous cycles; in a cycle every agent reads all its incoming messages, processes them and sends all its outgoing messages) the simulator must perform until the solution is found. We tested our algorithms on unstructured instances with binary random DCOPs, and on structured distributed meeting scheduling datasets.

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 number of cost functions (this is done to introduce some variability among tuple costs; using a unique type of cost function causes that all tuples look pretty similar from an optimization view). Results appear in Table 1 (a), averaged over 50 instances.

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 [9] with different hierarchical scenarios and domain 10: case A (8 variables), case B (10 variables), case C (12 variables) and case D (12 variables). Results appear in Table 1 (b), averaged over 30 instances.

For each problem, we calculate an initial $\top$ to have prune opportunities on the AC* and FDAC* preprocess. This is done in the following way. Each leaf agent choose the best value with local information, and informs its parent of the selected value and its cost. Parents receive this information from children and choose their own best value regarding local information, and also inform their parents accumulating the cost of the partial solution. When all agents have chosen their value, we have a complete solution (likely not the optimal one) which is an upper bound of the optimum problem cost. So $root$ calculates the cost of this complete solution and inform this cost downwards. This cost is considered the initial $\top$ of the problem. With this preprocess we are able to calculate a $\top$ different from $\infty$, requiring only two messages per each agent: one from child to parent informing the partial solution cost, and one from parent to children informing of the global initial $\top$.

On random DCOPs, BnB-ADOPT$^+$-AC*/FDAC* showed clear benefits on communication costs with respect to BnB-ADOPT$^+$. Maintaining AC* level (BnB-ADOPT$^+$-AC*) the number of exchanged messages is divided by a factor from 3 to 10. Notice that this reduction is obtained generating only very few DEL messages. In addition, including the FDAC* level (BnB-ADOPT$^+$-FDAC*) enhances this reduction, dividing the number of BnB-ADOPT$^+$ exchanged messages by a factor from 5 to 27. Notice that maintaining the higher FDAC* level increases slightly the number of DEL messages (this is because more deletions have been generated) and only very few UCO messages are added. In contrast, important savings are obtained compared to AC*. In general, including few DEL and UCO messages and performing extra local computation to enforce soft arc consistency allows BnB-ADOPT$^+$-AC*/FDAC* to obtain large reductions in VALUE and COST messages. This is because values that will not be in any optimal solution – which would be discovered by distributed search– are sooner removed by soft arc consistency, so agents will need to assign less values (consequently they will generate less VALUE messages) when testing the optimum assignment for each context. If less VALUEs are generated, less COSTs will be sent in response. We assume the usual case where communication time is higher than computation time, then the total elapsed time is dominated by the communication time, and reducing the number of messages causes an important effect in performance.

We also observe a clear decrement in the number of cycles of BnB-ADOPT$^+$-AC*/FDAC* (divided by a factor from 3 to 17), combined with a decrement in the number of messages per cycle with respect to the original BnB-ADOPT$^+$. Assuming that processing each message type requires approximately the same time, the combination of these two effects 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 very beneficial for agent performance.

Notice that although agents need to perform more local computation to maintain local consistency, the number of non-concurrent constraint checks (NCCCs) also shows important reductions. This is the combination of two opposite trends: agents are doing more work enforcing soft arc consistency and processing new DEL and UCO messages, but less work processing less VALUE and COST mes-

(a) Random DCOPs

| $p_1$ | #Msgs | #VALUE | #COST | #DEL | #UCO | #Cycles | #NCCC | #Deletions |
|---|---|---|---|---|---|---|---|---|
| | 354,415 | 119,007 | 235,399 | **0** | **0** | 34,222 | 7,117,237 | 0 |
| | 35,011 | 18,489 | 16,298 | 196 | **0** | 4,962 | 705,548 | 66 |
| 0.3 | **28,502** | **15,564** | **12,542** | 223 | 145 | **3,935** | **575,709** | **76** |
| | 5,743,888 | 1,430,183 | 4,313,695 | **0** | **0** | 547,566 | 132,071,219 | 0 |
| | 622,241 | 296,087 | 325,902 | 225 | **0** | 114,147 | 20,979,964 | 60 |
| 0.4 | **209,142** | **105,150** | **103,496** | 268 | 199 | **31,519** | **5,693,774** | **72** |
| | 9,680,458 | 2,693,821 | 6,986,627 | **0** | **0** | 918,093 | 251,286,055 | 0 |
| | 2,469,241 | 1,092,284 | 1,376,669 | 260 | **0** | 503,722 | 131,834,779 | 55 |
| 0.5 | **1,287,015** | **571,312** | **715,069** | 339 | 266 | **223,496** | **66,967,310** | **71** |
| | 7,813,885 | 2,333,830 | 5,480,046 | **0** | **0** | 685,061 | 189,161,842 | 0 |
| | 2,299,184 | 1,022,889 | 1,275,964 | 303 | **0** | 359,408 | 113,987,882 | 54 |
| 0.6 | **1,317,743** | **624,336** | **692,658** | 392 | 329 | **173,822** | **51,705,844** | **71** |

(b) Distributed Meeting Scheduling

| | #Msgs | #VALUE | #COST | #DEL | #UCO | #Cycles | #NCCC | #Deletions |
|---|---|---|---|---|---|---|---|---|
| | 35,767 | 14,021 | 21,739 | **0** | **0** | 4,427 | 690,786 | 0 |
| | 5,818 | 2,461 | 3,157 | 177 | **0** | 1,306 | 220,040 | 43 |
| A | **5,325** | **2,198** | **2,808** | 199 | 97 | **1,167** | **210,358** | **49** |
| | 69,453 | 28,821 | 40,623 | **0** | **0** | 7,150 | 801,384 | 0 |
| | 11,474 | 4,924 | 6,369 | 153 | **0** | 2,585 | 313,254 | 44 |
| B | **10,207** | **4,317** | **5,591** | 180 | 90 | **2,326** | **297,964** | **52** |
| | 13,862 | 6,907 | 6,944 | **0** | **0** | 1,278 | 157,995 | 0 |
| | 3,155 | 1,655 | 1,257 | 209 | **0** | 325 | **48,447** | 74 |
| C | **2,990** | **1,493** | **1,126** | 224 | 113 | **295** | 53,717 | **80** |
| | 20,386 | 9,457 | 10,917 | **0** | **0** | 1,733 | 141,816 | 0 |
| | 3,507 | 1,708 | 1,557 | 208 | **0** | 532 | **57,412** | 74 |
| D | **3,196** | **1,474** | **1,327** | 235 | 125 | **462** | 61,559 | **84** |

**Table 1.** Experimental results of BnB-ADOPT$^+$ (first row) compared to BnB-ADOPT$^+$-AC* (second row) and BnB-ADOPT$^+$-FDAC* (third row)

sages. This combination turns out to be very beneficial, saving computational effort for all cases tested. In some cases, reduction reaches up to one order of magnitude.

For the meeting scheduling instances, we also obtain clear benefits maintaining AC*, enhanced by FDAC*. For the stronger FDAC* level (BnB-ADOPT$^+$-FDAC*) messages are divided by a factor from 4 to 6, cycles are divided by a factor from 3 to 4 and there are significant savings in NCCCs. To obtain these results, very few DEL and UCO messages are needed, and the extra computational effort required to maintain AC* or FDAC* is effectively balanced by the decrement on VALUE and COST messages.

So, maintaining soft arc consistency (BnB-ADOPT$^+$-AC$^*$/FDAC$^*$) proved to be clearly beneficial for the instances tested. The propagation of deletions contributes to diminish the search effort, decreasing the number of COST and VALUE messages exchanged. Also, 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, searching for more pruning opportunities. In the worst case, maintaining FDAC$^*$ our approach divides the number of messages required to reach an optimal solution by a factor of 3, substantially decreasing the number of cycles and the computational effort at each agent.

## 6 CONCLUSION

In this work we have connected BnB-ADOPT$^+$ with some forms of soft arc consistency in the weighted case, aiming at detecting and pruning values which would not be in the optimal solution, with the final goal of improving search efficiency. These deletions are unconditional and do not rely on any previous variable assignment. The transformations introduced (extending unary costs into binary ones, projecting binary costs into unary ones, projecting unary costs into $C_\phi$, and pruning values not NC$^*$) assure that the optimum (and any optimal solution) of the transformed problem remains the same as the original instance. According to experimental results, propagation of unconditional deletions provides substantial benefits for the benchmarks tested. New messages DEL and UCO have been introduced. However, the increment in the number of messages due to the generation of new DEL and UCO messages has been largely compensated by the decrement in the number of COST and VALUE messages used to solve the problem. BnB-ADOPT$^+$-AC$^*$/FDAC$^*$ has been proved to be very beneficial with respect to BnB-ADOPT$^+$, not only in communication cost but also in computation effort.

## ACKNOWLEDGEMENTS

This work is partially supported by the project TIN2009-13591-C02-02. We want to thank the referees for their constructive comments.

## REFERENCES

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

[2] P. Gutierrez and P. Meseguer, 'Saving messages in BnB-ADOPT', *Proc. AAAI-10*, (2010).

[3] J. Larrosa and T. Schiex, 'In the quest of the best form of local consistency for weighted CSP', *Proc. of IJCAI-03*, (2003).

[4] J. Larrosa and T. Schiex, 'Solving weighted csp by maintaining arc consistency', *Artificial Intelligence*, **159**, 1–26, (2004).

[5] P. Meseguer, F. Rossi, and T. Schiex, *Handbook of Constraint Programming. Chapter 9, Soft Constraints.*, Elsevier, 2006.

[6] 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).

[7] R. Wallace and E. Freuder, 'Constraint-based reasoning and privacy/efficiency tradeoffs in multi-agent problem solving', *Artificial Intelligence*, **161**, 209–227, (2005).

[8] W. Yeoh, A. Felner, and S. Koenig, 'Bnb-adopt: An asynchronous branch-and-bound DCOP algorithm', *Proc. of AAMAS-08*, 591–598, (2008).

[9] Z. Yin. USC dcop repository. Meeting scheduling and sensor net datasets, http://teamcore.usc.edu/dcop, 2008.