Connecting ABT with Arc Consistency [⋆]

Ismel Brito and Pedro Meseguer

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

Abstract. ABT is the reference algorithm for asynchronous distributed constraint satisfaction. When searching, ABT produces nogoods as justifications of deleted values. When one of such nogoods has an empty left-hand side, the considered value is eliminated unconditionally, once and for all. This value deletion can be propagated using standard arc consistency techniques, producing new deletions in the domains of other variables. This causes substantial reductions in the search effort required to solve a class of problems. We also extend this idea to the propagation of conditional deletions, something already proposed in the past. We provide experimental results that show the benefits of the proposed approach, especially considering communication cost.

1 Introduction

In recent years, there is an increasing interest for solving problems in which information is distributed among different agents. Most of the work in constraint reasoning assumes centralized solving, so it is inadequate for problems requiring a true distributed resolution. This has motivated the new Distributed CSP (*DisCSP*) framework, where constraint problems with elements (variables, domains, constraints) distributed among automated agents which cannot be centralized for different reasons (prohibitive translation costs or security/privacy issues) are modelled and solved.

When solving a *DisCSP* instance, all agents cooperate to find a globally consistent solution. To achieve this, agents assign their variables and exchange messages on these assignments, which allows them to check their consistency with respect to problem constraints. Several synchronous and asynchronous solving algorithms have been proposed [12, 13, 8, 2, 5]. While synchronous algorithms are easier to understand and implement, asynchronous ones are more robust. If some agents disconnect, an asynchronous algorithm is still able to provide a solution for the connected part, while this is not true in general for synchronous ones. Asynchronous algorithms exhibit a high degree of parallelism, but the information exchanged among agents is less up to date than in synchronous ones.

ABT [12, 13] is the reference algorithm for asynchronous distributed constraint solving, playing a role similar to backtracking algorithm in the centralized case. Several ideas to improve its efficiency and privacy have been proposed.

[⋆] Supported by the Spanish project TIN2006-15387-C03-01.

In this paper we study the idea of propagating value deletions in ABT. When searching, ABT produces nogoods as justifications of deleted values. When one of such nogoods has an empty left-hand side, the considered value is eliminated unconditionally, once and for all. This value deletion can be propagated using standard arc consistency techniques, producing new deletions in the domains of other variables. This causes substantial reductions in the search effort required to solve a class of problems, especially on communication cost. We extend this idea to the propagation of conditional deletions.

The idea of including consistency maintenance in ABT is not new. It was proposed by [9, 10]. However, the specialization to unconditional value deletions (nogoods with empty left-hand side) is new. Previous experimental results [9, 10], considered AAS [8] with bound consistency. Here, we provide experimental results for ABT with directional and full (both directions) arc consistency on a set of random *DisCSP* instances.

This paper is organized as follows. First, we recall the *DisCSP* definition and the *ABT* description. Then, we present the idea of propagating unconditional deletions, in the ABT-UAC algorithm. We extend this idea to conditional deletions, in the ABT-DAC algorithm. We present experimental results for both approaches on random *DisCSP* instances. Finally, we extract some conclusions and directions for further research.

2 Preliminaries

2.1 Distributed Constraint Satisfaction

A *Constraint Satisfaction Problem* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ involves a finite set of variables $\mathcal{X}$, each taking values in a finite domain, and a finite set of constraints $\mathcal{C}$. A constraint on a subset of variables forbids some combinations of values that these variables can take. A *solution* is an assignment of values to variables which satisfies every constraint. Formally,

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables;
- $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of finite domains; $D(x_i)$ is value set for x_i;
- $\mathcal{C}$ is a finite set of constraints. A constraint C_i on the ordered subset of variables $var(C_i) = (x_{i_1}, \ldots, x_{i_{r(i)}})$ specifies the relation $prm(C_i)$ of the *permitted* combinations of values for the variables in $var(C_i)$, $prm(C_i) \subseteq \prod_{x_{i_k} \in var(C_i)} D(x_{i_k})$. An element of $prm(C_i)$ is a tuple $(v_{i_1}, \ldots, v_{i_{r(i)}}),\ v_{i_k} \in D(x_{i_k})$.

A *Distributed Constraint Satisfaction Problem* (*DisCSP*) is a *CSP* where variables, domains and constraints are distributed among automated agents. Formally, a finite *DisCSP* is defined by a 5-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ are as before, and

- $\mathcal{A} = \{1, \ldots, p\}$ is a set of p agents,
- $\phi : \mathcal{X} \rightarrow \mathcal{A}$ is a function that maps each variable to its agent.

Each variable belongs to one agent. The distribution of variables divides $\mathcal{C}$ in two disjoint subsets, $\mathcal{C}_{intra} = \{C_i | \forall x_j, x_k \in var(C_i), \phi(x_j) = \phi(x_k)\}$, and $\mathcal{C}_{inter} = \{C_i | \exists x_j, x_k \in var(C_i), \phi(x_j) \neq \phi(x_k)\}$, called intraagent and interagent constraint sets, respectively. An intraagent constraint C_i is known by the agent owner of $var(C_i)$, and it is unknown by the other agents. Usually, it is considered that an interagent constraint C_j is known by every agent that owns a variable of $var(C_j)$ [13].

A *solution* of a *DisCSP* is an assignment of values to variables satisfying every constraint. *DisCSP*s are solved by the coordinated action of agents, which communicate by exchanging messages. It is assumed that the delay of a message is finite but random. For a given pair of agents, messages are delivered in the order they were sent. For simplicity, we assume that each agent owns exactly one variable, and the agent number is the variable index ($\forall x_i \in \mathcal{X}, \phi(x_i) = i$). Furthermore, we assume that all constraints are binary. A constraint C_{ij} indicates that it binds variables x_i and x_j.

2.2 Asynchronous Backtracking

ABT [12, 13] is the reference algorithm for asynchronous distributed constraint solving, with a role similar to backtracking in the centralized case. An *ABT* agent makes its own decisions, informs other agents about them, and no agent has to wait for the others' decisions. The algorithm computes a global consistent solution (or detects that no solution exists) in finite time; its correctness and completeness have been proved [13, 2]. *ABT* requires constraints to be directed. A binary constraint causes a directed link between the two constrained agents: the value-sending agent, from which the link starts, and the constraint-evaluating agent, at which the link ends. To make the network cycle-free, there is a total order among agents, which is followed by the directed links.

Each ABT agent keeps its own agent view and nogood store. The agent view of $self$, a generic agent, is the set of values that $self$ believes are assigned to higher priority agents (connected to $self$ by incoming links). Its nogood store keeps nogoods as justifications of inconsistent values. Agents exchange four message types:

- **ok?**: a high priority agent informs lower priority ones about its assignment.
- **ngd**: a lower priority agent inform a higher priority one of a new nogood.
- **addl**: a lower priority agent requests a higher priority one to set up a link.
- **stop**: the empty nogood has been generated. There is no solution.

When the algorithm starts, each agent assigns its variable, and sends the assignment to its neighboring agents with lower priority. When $self$ receives an assignment, $self$ updates its agent view with the new assignment, removes inconsistent nogoods and checks the consistency of its current assignment with the updated agent view.

When $self$ receives a nogood, it is accepted if the nogood is consistent with $self$'s agent view (for the variables in the nogood, their values in the nogood and in $self$'s agent view are equal). Otherwise, $self$ discards the nogood as obsolete. If the nogood is accepted, the nogood store is updated, causing $self$ to search for a new consistent value (since the received nogood forbids its current value). If an unconnected agent i appears in the nogood, it is requested to set up a new link with $self$. From this point on, $self$ will receive i values. When $self$ cannot find any value consistent with its agent view, either because of the original constraints or because of the received nogoods, new nogoods are generated from its agent view and each one sent to the closest agent involved in it. This operation causes backtracking. There are several forms of how new nogoods are generated. In [2], when an agent has no consistent values, it resolves its nogoods following a procedure described in [1]. In this paper we consider this last version. The ABT code is in Figure 1 for the $self$ agent. Γ_0^+ and Γ_0^- are the sets of agents initially constrained with $self$ which are above and below it in the agent ordering.

```
**procedure** ABT()
$\Gamma = \Gamma_0^- \cup \Gamma_0^+; \Gamma^- = \Gamma_0^-; \Gamma^+ = \Gamma_0^+; myValue \leftarrow$ empty; $end \leftarrow$ false; CheckAgentView();
**while** ($\neg end$) **do**
  $msg \leftarrow$ getMsg();
  **switch**($msg.type$)
  $Ok?$:ProcessInfo($msg$); $Ngd$:Conflict($msg$); $Stop$: $end \leftarrow$ true; $AddL$:SetLink($msg$);

**procedure** CheckAgentView()
**if** $\neg$consistent($myValue, myAgentView$) **then**
  $myValue \leftarrow$ ChooseValue();
  **if** ($myValue$) **then for each** $k \in \Gamma^+$ **do** sendMsg:$Ok?(k, myValue)$; **else** Backtrack();

**procedure** ProcessInfo($msg$)
Update($myAgentView, msg.assig$);  CheckAgentView();

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

**procedure** SetLink($msg$)
add($msg.sender, \Gamma^+$);  sendMsg:$Ok?(msg.sender, myValue)$;

**procedure** CheckAddLink($msg$)
**for each** ($var \in$ lhs($msg.nogood$))
  **if** ($var \notin \Gamma^-$) **then** sendMsg:$AddL(var, self)$;  add($var, \Gamma^-$);
                      Update($myAgentView, msg.nogood[var]$);

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

**function** ChooseValue()
**for each** $v \in D(self)$ not eliminated by $myNogoodStore$ **do**
  **if** consistent($v, myAgentView$) **then return** ($v$); **else** add($nogood(v), myNogoodStore$);
**return** (empty);

**procedure** Update($myAgentView, newAssig$)
add($newAssig, myAgentView$);
**for each** $ng \in myNogoodStore$ **do**
  **if** $\neg$Coherent(lhs($ng$), $myAgentView$) **then** remove($ng, myNogoodStore$);

**function** Coherent($nogood, agents$)
**for each** $var \in nogood \cup agents$ **do if** $nogood[var] \neq myAgentView[var]$ **then return** false;
**return** true;
```

Fig. 1. The ABT algorithm for asynchronous backtracking search.

3 Propagating Unconditional Deletions

During search, ABT produces nogoods in $self$ as result of the reception of **ok?** and **ngd** messages. A *nogood* is a conjunction of individual assignments, which has been found inconsistent, either because the initial constraints or because searching all possible combinations. For instance, the following nogood,

$$x_1 = a \wedge x_2 = b \wedge x_3 = c$$

means that these three assignments cannot happen simultaneously because they cause an inconsistency: either they violate a constraint or any extension including the remaining variables violates a constraint. Often, a nogood is written in *directed form*,

$$x_1 = a \wedge x_2 = b \Rightarrow x_3 \neq c$$

meaning that x_3 cannot take value c because of the values of x_1 and x_2. In a directed nogood, "$\Rightarrow$" separates the *left-hand side* (`lhs`) from the *right-hand side* (`rhs`). Since variables are ordered at each branch of the search tree, it is useful to write nogoods in a directed form, where the last variable in the branch order appears in the `rhs`. A nogood is a necessary *justification* to eliminate a value. In the previous example, the directed nogood is a justification to eliminate value c of $D(x_3)$. A nogood is *active* if the assignments in its `lhs` hold. To assure polynomic space usage, ABT only keeps one active nogood per eliminated value. As soon as a nogood becomes no active, it is removed (and the corresponding eliminated value is again available).

When all values of $D(self)$ are eliminated by some nogood, the justifying nogoods are resolved generating a new nogood ng (see [1] for a detailed description of this process). ng is sent to var, the variable that appears in $\mathtt{rhs}(ng)$ (which always has the form $var \neq val$). This means that val can be eliminated from $D(var)$, conditioned to the assignments of $\mathtt{lhs}(ng)$. It may happen that $\mathtt{lhs}(ng)$ is empty. In this case, val can be deleted from $D(var)$ unconditionally, once and for all. After removal, val will never be available again: no matter which new assignments may be explored in the future, the empty $\mathtt{lhs}(ng)$ always holds.

An unconditional deletion may generate further unconditional deletions in other domains, if it happens that the initial deletion causes a constraint (or a subset of constraints) to become locally inconsistent, and the corresponding local consistency is enforced afterwards. In this paper, we only consider arc consistency, although other local consistencies removing single values could also be analyzed [6]. We assume that domains are initially arc consistent (if not, this can be easily done by a preprocess, explained below). If value a of variable x_i is deleted, this has to be notified to all agents connected with i. These agents will check their constraints with i to enforce arc consistency after a deletion (for instance, using the popular `revise-2001` function [3]). If more deletions occur, they are propagated in the same way, until reaching a fix point.

Values deleted in this way are removed once and for all. Let us assume three sequentially constrained agents i, j and k, $i < j < k$, connected by two constraints C_{ij} and C_{jk}, such that j receives a nogood from k eliminating unconditionally value b. If it happens that b was the only support for value $a \in D(x_i)$, after the deletion of b in $D(x_j)$, a must be deleted from $D(x_i)$ because a will not be in any solution. Value b will never be available again, so a would never have a support and its deletion is unconditional.

3.1 ABT-UAC

The idea of propagating unconditional deleted values can be included in ABT, producing the new algorithm ABT-UAC. It exploits the idea that a constraint C_{ij} is known by both agents i and j. ABT-UAC requires the some minor changes with respect to ABT:

- In addition to its own domain, the domain of every variable constrained with $self$ is also represented in $self$. Assuming that a constraint between $self$ and j does not contain irrelevant values, domain computation can be done by projecting the constraint on x_j. This constraint will be arc consistent after the preprocess.
- A new message type, **del**, is required. When $self$ deletes value a in $D(self)$, it sends a **del** message to every agent initially constrained with it, except the agent that sent the message that caused a deletion. When $self$ receives a **del** message, it registers that the message value has been deleted from the domain of sender, and it enforces arc consistency on the constraint between $self$ and sender. If, as result of this enforcing, some value is deleted in $D(self)$ it is propagated as above.

Including the propagation of unconditionally deleted values does not changes the semantic of original ABT messages. It is worth noting that ABT-UAC keeps the good ABT properties, namely correctness, completeness and termination: since we are eliminating values which are unconditionally arc inconsistent, their removal will not cause to miss any solution. If the value assigned to $self$ is found to be arc inconsistent, it is removed and another value is tried for $self$. Any value removal is propagated to agents initially constrained with $self$.

It is mentioned above that initial domains are assumed to be arc consistent. If not, this can be easily done by a preprocess depicted in Figure 2, executed on each agent. First, it initially enforces arc consistency between $self$ and each constrained agent.

```
procedure AC-preprocess()
  compute $\Gamma_0 = \Gamma_0^- \cup \Gamma_0^+$; $end \leftarrow$ false; init structures of revise-2001
  for each $j \in \Gamma_0$ do AC($self, j$);
  while ($\neg end$) do
    $msg \leftarrow$ getMsg();
    switch($msg.type$)
      $Del$: ValueDeletedPre($msg.sender, msg.value$);          $Stop$: $end \leftarrow$ true;

procedure ValueDeletedPre($j, a$)
  $D(j) \leftarrow D(j) - \{a\}$;   AC($self, j$);

procedure AC($self, j$)
  if revise-2001($self, j$) then
    if $D(self) = \emptyset$ then sendMsg:$Stop(system)$;
    else $DEL$ is the set of deleted values in $D(self)$ by the last revise-2001($self, j$) call
        for each $v \in DEL$ and $k \in \Gamma_0, k \neq j$ do sendMsg:$Del(self, v)$;
```

Fig. 2. The AC algorithm for preprocessing DisCSP.

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

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

new procedure ValueDeleted($j, a$)
new     $D(j) \leftarrow D(j) - \{a\}$;  AC($self, j$);
new     if $myValue \notin D(self)$ then $myValue \leftarrow$ empty;  CheckAgentView();

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

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

Fig. 3. New lines/procedures of ABT-UAC with respect to ABT.

Second, value deletions are propagated as described above, until reaching quiescence, when ABT-UAC execution begins. Value deletions in the preprocessing phase are unconditional. Differences between ABT-UAC and ABT appear in Figure 3. They are,

- ABT-UAC. It includes the Del message, which notifies that a value has been deleted in some domain. Upon reception, the ValueDeleted procedure is called.
- Conflict. After accepting a Ngd message with empty lhs, the DeleteValue procedure is called.
- ValueDeleted(j, a). Agent j has deleted value a of its domain. $self$ registers this in its $D(j)$ copy, and enforces AC on the constraint between $self$ and j. If the value of $self$ is deleted in this process, the CheckAgentView procedure is called (looking for a new compatible value; if none exists performs backtracking). Any deletion in $D(self)$ is propagated.

- `DeleteValue`(a, j). Agent $self$ must delete its currently assigned value a because a nogood with empty `lhs` has been received from agent j. Value a is deleted from $D(self)$. If, as consequence of a's deletion, $D(self)$ becomes empty, there is no solution so a $Stop$ message is produced. Otherwise, a's deletion is notified to all agents constrained with $self$ except j via Del messages, and the procedure `CheckAgentView` is called.
- `Backtrack`. After $self$ computes and sends a $newNogood$, it checks if its `lhs` is empty. If so, $self$ knows that the value that forbids $newNogood$ will be removed in the domain of the variable that appears in `rhs`$(newNogood)$. Therefore, $self$ calls `ValueDeleted`, as if it would had received a Del message.

3.2 Example

A simple example of the benefits of this approach appears in Figure 4. It is a graph coloring instance, with seven agents and the indicated domains. This instance has no solution (realize that there are two available values a, b for the clique formed by the agents 5, 6 and 7, mutually connected). We assume that agents are ordered lexicographically and values are tried in the order indicated for each domain. Agent 1 assigns $x_1 \leftarrow a$, to discover after a while that there is no solution with this assignment. A nogood with empty `lhs` will reach agent 1, forbidding value a. From this point on, ABT and ABT-UAC behave differently.

ABT behavior is summarized in Figure 5, while ABT-UAC behavior is summarized in Figure 6. Since tracing asynchronous algorithms is difficult, we assume that all messages sent in a time period are read in the next time period. The main difference between ABT and ABT-UAC is that the latter propagates unconditional deletions via **del** messages. As consequence of that, it detects two empty domains at period 4 (D_4 and D_5), so there is no solution. For this, it exchanges 23 messages. ABT performs just search and it requires 25 messages to deduce that the instance has no solution.

It is worth noting that the detection of empty domains by ABT-UAC is done by the unique action of **del** messages, and **ok?** and **ngd** messages are useless.

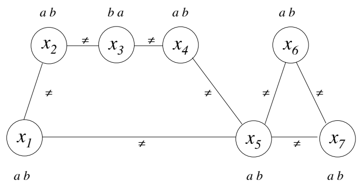

Fig. 4. Instance of graph coloring with 7 agents, each holding a variable. Domains are indicated.

t/a	1	2	3	4	5	6	7
1	$x_1 \leftarrow b$ 1 **ok?** to x_2 1 **ok?** to x_5	$x_2 \leftarrow a$ 1 **ok?** to x_3	$x_3 \leftarrow b$ 1 **ok?** to x_4	$x_4 \leftarrow a$ 1 **ok?** to x_5	$x_5 \leftarrow a$ 2 **ok?** to x_6, x_7	$x_6 \leftarrow a$ 1 **ok?** to x_7	$x_7 \leftarrow a$
2					$x_1 = b \Rightarrow x_5 \neq b$ $x_4 = a \Rightarrow x_5 \neq a$ 1 **ngd** to x_4 $x_5 \leftarrow a$ 2 **ok?** to $x_6 x_7$	$x_6 \leftarrow b$ 1 **ok?** to x_7	$x_7 \leftarrow b$
3				$x_1 = b \Rightarrow x_4 \neq a$ $x_3 = b \Rightarrow x_4 \neq b$ 1 **ngd** to x_3 $x_4 \leftarrow b$ 1 **ok?** to x_5			$x_5 = a \Rightarrow x_7 \neq a$ $x_6 = b \Rightarrow x_7 \neq b$ 1 **ngd** to x_6 $x_7 \leftarrow b$
4			$x_1 = b \Rightarrow x_3 \neq b$ $x_2 = a \Rightarrow x_3 \neq a$ 1 **ngd** to x_2 $x_3 \leftarrow a$ 1 **ok?** to x_4			$x_5 = a \Rightarrow x_6 \neq a$ $x_5 = a \Rightarrow x_6 \neq b$ 1 **ngd** to x_5 $x_6 \leftarrow a$ 1 **ok?** to x_7	
5		$x_1 = b \Rightarrow x_2 \neq a$ $x_1 = b \Rightarrow x_2 \neq b$ 1 **ngd** to x_1 $x_2 \leftarrow a$ 1 **ok?** to x_3			$x_1 = b \Rightarrow x_5 \neq b$ $\Rightarrow x_5 \neq a$ 1 **ngd** to x_1 $x_5 \leftarrow a$ 2 **ok?** to x_6, x_7		
6	$\Rightarrow x_1 \neq a$ $\Rightarrow x_1 \neq b$ empty nogood stop				$x_6 \leftarrow b$ 1 **ok?** to x_7		

Fig. 5. Trace of ABT in the example, after discarding value a for x_1.

time/agent	1	2	3	4	5	6	7
1	$D_1 = \{\cancel{a} b\}$ 2 **del** to x_2, x_5 $x_1 \leftarrow b$ 2 **ok?** to x_2, x_5	$x_2 \leftarrow a$ 1 **ok?** to x_3	$x_3 \leftarrow b$ 1 **ok?** to x_4	$x_4 \leftarrow a$ 1 **ok?** to x_5	$x_5 \leftarrow a$ 2 **ok?** to x_6, x_7	$x_6 \leftarrow a$ 1 **ok?** to x_7	$x_7 \leftarrow a$
2					$D_5 = \{a\ \cancel{b}\}$ 2 **del** to x_6, x_7 $x_1 = b \Rightarrow x_5 \neq b$ $x_4 = a \Rightarrow x_5 \neq a$ 1 **ngd** to x_4 $x_5 \leftarrow a$ 2 **ok?** to $x_6 x_7$	$x_6 \leftarrow b$ 1 **ok?** to x_7	$x_7 \leftarrow b$
3				$x_1 = b \Rightarrow x_4 \neq a$ $x_3 = b \Rightarrow x_4 \neq b$ 1 **ngd** to x_3 $x_4 \leftarrow b$ 1 **ok?** to x_5		$D_6 = \{\cancel{a} b\}$ 1 **del** to x_7	$D_7 = \{\cancel{a} b\}$ 1 **del** to x_6 $x_5 = a \Rightarrow x_7 \neq a$ $x_6 = b \Rightarrow x_7 \neq b$ 1 **ngd** to x_6 $x_7 \leftarrow b$
4			$x_1 = b \Rightarrow x_3 \neq b$ $x_2 = a \Rightarrow x_3 \neq a$ 1 **ngd** to x_2 $x_3 \leftarrow a$ 1 **ok?** to x_4			$D_6 = \emptyset$ stop	$D_7 = \emptyset$ stop

Fig. 6. Trace of ABT-UAC in the example, after discarding value a for x_1.

4 Propagating Any Deletion

The idea of propagating unconditional deletions can be extended to propagate any deletion, including conditional ones. A value is conditionally deleted when the reason for its removal is a nogood with a non-empty `lhs`. This value remains deleted as long as its justifying nogood is active. When this nogood becomes no active, it has to be removed and the value is available again. Propagating any deletion means that any deleted value and its justifying nogood of any agent has to be sent to any other agent constrained with it. This was already proposed in [9, 10]. Agents have to store the received nogoods while they are active, but the space complexity remains polynomial [9]. As in the previous case, ABT propagating any deletion remains sound, complete and terminates.

When a value $a \in D(self)$ is removed, we differentiate between,

- Unconditional deletion. Value a is removed when,
  1. a nogood with empty `lhs` has been accepted, or
  2. all values initially consistent with a in the domain of a constrained variable have been unconditionally eliminated.

  Then, a is eliminated from $D(self)$ once and for all ($D(self) \leftarrow D(self) - \{a\}$).
- Conditional deletion. Value a is removed when,
  1. $self$ produces a nogood with non-empty `lhs` for a when looking for a consistent value for x_{self}; it is the justification for the conditional a deletion, or
  2. a nogood with non-empty `lhs` has been accepted; this nogood is the justification for the conditional a deletion, or
  3. all values initially consistent with a in the domain of a constrained variable have been eliminated, and this removal is conditional for at least one of these values. The nogood of a deletion is the conjunction of the `lhs` of the nogoods of the conditionally removed values which were initially consistent with a.

Nogoods justifying deletions of values in $D(self)$ have to be sent to constrained agents, which will enforce arc consistency in their constraints with $self$. This may produce further deletions in the domains of those agent variables, which have to be propagated, etc. Each time a value is conditionally deleted, a nogood is added that justifies its deletion. When this nogood is no longer active, it has to be removed and the deleted value becomes available again. Because of that, if an agent stores a nogood, it must have direct link with all agents owners of the variables that appear in its `lhs`, to be notified if one of these variables changes its value (which could render the nogood no active). To perform propagation, $self$ has to send all nogoods of its values to all agents constrained with $self$. If ng is a nogood to propagate, it is sent to all constrained agents that are below the last variable (`lv`) in the static agent ordering of ABT agents, that appears in `lhs`(ng). The reason is clear: if ng is sent to agent $k <$ `lv`(`lhs`(ng)), agent k has no way to determine if ng is active or not, because there are variables in `lhs`(ng) which are below k (so their values will never be sent to k).

Propagating any deletion has a clear drawback: the *huge* number of messages that should be exchanged. This large number of **del** messages may overcome the benefits of propagation, which are reduction of the search effort (because there are less available values) which causes a reduction in the number of **ok?** and **ngd** messages. To mitigate this drawback, we suggest to propagate any deletion directionally, following the ABT static order of agents. If a value is deleted in $self$, the **del** message goes to agents above $self$ in the ordering. If agent $j, j < self$ is constrained with $self$, upon the reception of **del** message, j enforces arc consistency in the constraint between j and $self$.

Assuming that the instance is initially arc consistent, directional arc consistency is maintained. These ideas are implemented in the ABT-DAC algorithm. This algorithm presents the following changes with respect to ABT-UAC:

- Agent $self$, in addition to store the nogoods for the values of $D(self)$, it has to store the nogoods for values of other agents. Because of that, $myNogoodStore$ becomes a vector indexed by agent, $myNogoodStore[k]$.

- Message **del** contains a nogood (instead of a pair (variable,value)).

Differences of ABT-DAC with ABT-UAC appear in Figure 7. In that code, when value a is unconditionally deleted, it is removed from its domain ($D \leftarrow D - \{a\}$). When a is conditionally deleted, a nogood justifying its deletion is added to the nogood store. Checking for empty domain ($D = \emptyset$) means if all values of D have been unconditionally removed. New lines are explained in the following,

```
procedure ValueDeleted($msg$)
new  if Coherent ($msg.nogood, \Gamma^- \cup \{self\}$) then
new    if lhs($msg.nogood$) = empty then $D(msg.sender) \leftarrow D(msg.sender) - \{a\}$;
new    else add($msg.nogood, myNogoodStore[msg.sender]$); CheckAddLink($msg$);
new    DAC($self, msg.sender$);
       if $myValue \notin D(self)$ then $myValue \leftarrow$ empty;  CheckAgentView();

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

procedure Backtrack()
new  $newNogood \leftarrow$ solve($myNogoodStore[self]$);
     if ($newNogood$ = empty) then $end \leftarrow$ true;  sendMsg:$Stop(system)$;
     else sendMsg:$Ngd(newNogood)$; Update($myAgentView$,rhs($newNogood$) $\leftarrow$ ukn);
new  ValueDeleted($newNogood$);

new  procedure DAC($self, j$)
new  if revise-2001($self, j$) then
new    if $D(self) = \emptyset$ then sendMsg:$Stop(system)$; /* empty by unconditional deletions */
new    else $DEL$ is the set of deleted values in $D(self)$ by the last revise-2001($self, j$) call
new      for each $a \in DEL, ng(a)$ justifies deletion, $k \in \Gamma_0^-, k >$ lv(lhs($ng(a)$)) do
new        sendMsg:$Del(ng(a))$;

function ChooseValue()
new  for each $v \in D(self)$ not eliminated by $myNogoodStore[self]$ do
       if consistent($v, myAgentView$) then return ($v$);
new    else add($nogood(v), myNogoodStore[self]$);
new        for each $k \in \Gamma_0^- k >$ lv (lhs($nogood(v)$)) do sendMsg:$Del(nogood(v))$;
     return (empty);

procedure Update($myAgentView, newAssig$)
     add($newAssig, myAgentView$);
new  for each $k \in \Gamma^+ \cup \{self\}$ do
new    for each $ng \in myNogoodStore[k]$ do
new      if $\neg$Coherent(lhs($ng$), $myAgentView$) then remove($ng, myNogoodStore[k]$);
```

Fig. 7. New lines/procedures for ABT-DAC, with respect to ABT-UAC. Deletions are directionally propagated with DAC.

- `ValueDeleted`(msg). Agent $msg.sender$ has deleted a value of its domain with $msg.nogood$. First, $self$ checks if this message is up to date, by comparing `lhs`($msg.nogood$) with its $agentView$. If the message is accepted, then it differentiates between unconditional or conditional deletion. In the first case, the value is removed from $D(msg.sender)$. Otherwise, the nogood is stored and analyzed for possible new links. in both cases, directed arc consistency between $self$ and $msg.sender$ is enforced. If the value of $self$ is deleted in this process, the `CheckAgentView` procedure is called (looking for a new compatible value; if none exists performs backtracking). Any deletion in $D(self)$ is directionally propagated.
- `DeleteValue`(a, j). The only difference with the same procedure of ABT-UAC is in the last line, where an unconditional deletion is propagated. The new format of **del** message requires to form a directed nogood.
- `Backtrack`. Differences are in the first and last lines. In the first line, because $myNogoodStore$ is now a vector, $myNogoodStore[self]$ is solved, instead of $myNogoodStore$. In the last line, the `ValueDeleted` procedure is always called.
- `DAC`$(self, j)$. Arc consistency is enforced in the constraint between $self$ and j. If, as result of this enforcing, $D(self)$ becomes empty (all values have been unconditionally deleted), the problem has no solution, stop. Otherwise, any deletion is propagated to agents above $self$ and below `lv(lhs)` of the nogood sent.
- `ChooseValue`(). A new value for $self$, consistent with $myAgentView$, is fetched. If a particular value is not consistent with $myAgentView$ (because constraints with other agents), a nogood justifying its conditional deletion is computed, stored and directionally propagated.
- `Update`$(myAgentView, newAssig)$. $myNogoodStore[k]$, now a vector indexed by agent, causes the only difference. To remove nogoods which may become no active by $newAssig$, all stored nogoods of agents below $self$ should be checked.

It may occur that a **del** message includes a nogood more up to date than the agent view of the receiving agent. In that case, [9] used a time-stamp system to determine which message was earlier, and how to update correctly the agent view. We take a simpler approach here: if this happens, the **del** message is considered obsolete and discarded (see `ValueDeleted`).

5 Experimental Results

We experimentally evaluate the performance of ABT-UAC and ABT-DAC algorithms with respect to ABT on uniform binary random *DisCSP*. A binary random *DisCSP* class is characterized by $\langle n, d, p_1, p_2\rangle$, where n is the number of variables, d the number of values per variable, p_1 the network *connectivity* defined as the ratio of existing constraints, and p_2 the constraint *tightness* defined as the ratio of forbidden value pairs. The constrained variables and the forbidden value pairs are randomly selected [11]. A problem class will be referred to as a $\langle n, d, p_1, p_2\rangle$ network. Each agent is assigned one variable. Neighboring agents are connected by constraints.

Using this model, we have tested random instances on four sets of experiments. The first three ones consist of instances of 16 agents and 8 values per agent, considering

	alg	**#ok?**	**#ngd**	**#addl**	**#del**	**#msg**	**#del-val**	**#obsolete**	**nccc**
$p_1 = .2$	ABT	4,866	1,829	27	0	6,722	0	446	5,689
	ABT-UAC	1,566	573	19	154	2,312	66	142	2,502
	ABT-DAC	2,983	1,132	26	1,938	6,080	22	653	57,785
$p_1 = .5$	ABT	28,055	8,070	39	0	36,164	0	2,702	39,923
	ABT-UAC	27,487	7,902	39	125	35,553	19	2,642	40,039
	ABT-DAC	28,042	8,094	39	11,281	47,456	3	5,138	345,933
$p_1 = .8$	ABT	53,400	14,999	19	0	68,418	0	5,909	98,330
	ABT-UAC	52,304	14,670	19	805	67,798	76	5,783	101,948
	ABT-DAC	53,238	14,932	19	21,484	89,674	7	11,049	670,803

Table 1. Results of ABT, ABT-UAC and ABT-DAC on random instances of 16 agents, 8 values per agent and sparse, medium and dense connectivities.

	alg	**#ok?**	**#ngd**	**#addl**	**#del**	**#msg**	**#del-val**	**#obsol**	**nccc**
$p_1 = 1$	ABT	267,046	153,909	0	0	420,955	0	100,632	557,242
	ABT-UAC	44,312	25,378	0	45,640	115,331	1,130	16,277	1,708,130
	ABT-DAC	66,596	38,270	0	124,931	229,797	499	65,762	131,596,664
$p_1 = .7$	ABT	1,191,937	468,069	301	0	1,660,307	0	321,994	1,876,059
	ABT-UAC	1,130,235	443,916	301	47,954	1,622,407	1,587	305,807	4,535,344
	ABT-DAC	210,963	87,129	275	277,351	575,718	52	151,288	391,778,623

Table 2. Results of ABT, ABT-UAC and ABT-DAC on random instances of 50 agents, 50 values per agent and two connectivities.

three connectivity classes, sparse ($p_1 = .2, p_2 = .7$), medium ($p_1 = .5, p_2 = .4$) and dense ($p_1 = .8, p_2 = .3$) while the four experiment considers instances of 50 agents and 50 values per agent, considering two dense connectivity classes, ($p_1 = 1, p_2 = .875, p_1 = .7, p_2 = .8$). Tightness were selected at the complexity peak, where the differences among algorithms are more explicit. Tables 1 and 2 present the results of the algorithms according to two parameters: the communication cost, in terms of the number of messages, and the computation effort, in terms of the number of non concurrent constraint checks ($nccc$) [7], respectively. In addition to these parameters, we also report the number messages sent for each message type, the number of unconditionally deleted values and the number obsolete messages (obsolete **ngd** for ABT and ABT-UAC, obsolete **ngd** plus obsolete **del** for ABT-DAC). All parameters are averaged over 50 executions.

Table 1 presents the first three sets of experiments for random instances of 16 agents and 8 values per agent. The upper set corresponds to the sparse instances. Regarding the number of messages, we observe that ABT-UAC and ABT-DAC always dominate the standard ABT. ABT-UAC is the algorithm that shows the best results, reducing approximately three times the number of **ok?**, **ngd** and total messages sent. As expected, ABT-UAC sends a lower number of arc consistent messages than ABT-DAC. However, ABT-UAC discards more unconditional arc inconsistent values than ABT-DAC (since ABT-UAC and ABT-DAC perform different propagations of different deletions, they may cause different numbers of removed values). Regarding message obsolescence, results show that agents in ABT-UAC are better informed about others' assignments than agents in ABT.

The second and third sets of experiments in the same table correspond to medium and dense connected binary instances of 16 agents and 8 values. In both cases the tightness of the constraints is low at the complexity peak. Therefore, there is a little propagation to reach an arc consistent state. Although the impact of maintaining arc consistent domains on ABT is minor, ABT-UAC is always more economic than ABT with respect to the total number of messages. In contrast, ABT-DAC sends more messages than ABT.

Considering the three experiments, propagating deletions algorithms send **del** messages, which cause to delete some values. These deletions cause to diminish the search effort, decreasing the number of **ok?** and **ngd** messages exchanged. When the number of saved **ok?** and **ngd** messages is larger than the number of **del** messages, propagation pays off and causes an overall message decrement. However, if the number of saved **ok?** and **ngd** messages is smaller than the number of **del** messages, propagation is harmful. In the sparse class, both ABT-UAC and ABT-DAC are beneficial, while for the medium and dense classes only ABT-UAC is beneficial while ABT-DAC is harmful. In these two classes, the number of **ok?** and **ngd** is practically the same for ABT and ABT-DAC, so the effect of propagation is practically unnoticed. In terms of $nccc$, propagating deletions algorithms are clearly more costly than ABT, since they perform full or directed arc consistency, which implies more constraint checks. ABT-DAC is always more costly than ABT-UAC because it performs more effort, propagating also conditional deletions.

Since the decrement in the number of messages caused by ABT-UAC in the medium and dense connectivity classes of 16 agents and 8 values is minor, one might think that the proposed approach is not beneficial on any medium or dense classes. To evaluate this hypothesis, we have performed the fourth set of experiments for random instances of 50 agents and 50 values per agent, with $p_1 = 1$ and $p_1 = 0.7$. Results appear in Table 2. Regarding communication cost, results of $p_1 = 1$ show a significant improvement of ABT-UAC with respect to ABT: the number of messages it sends is 3.6 times lower than ABT. We can note larger gains in the number of messages for each ABT message type. We observe that ABT-DAC also needs lower number of messages than ABT, even when it discards less than the half of the arc inconsistent values that ABT-UAC. Results for $p_1 = .7$ show that the winner here is ABT-DAC, requiring a number of messages that divides by 2.9 the number required by ABT. ABT-UAC shows some minor improvements. Regarding computation effort, once again $nccc$ reflect the high local effort that agents must pay in order to have consistent domains.

6 Conclusions

From this work we can extract some conclusions. According to experimental results, propagation of unconditional deletions is not harmful, and it provides substantial benefits for some problem instances, reducing substantially ABT communication requirements among agents. Directional propagation of any deletion provides a less clear picture: it can be harmful in some instances, but also beneficial in others. More experimental work is needed to assess their relative importance in different problem classes.

As future work, many ideas remain to be explored. On one hand, it has to be found the right degree of arc consistency when propagating any deletion. In more general terms, other local consistencies that remove individual values [6] could replace arc consistency in the proposed approach; it remains to analyze how this can be done and their cost. On the other hand, the proposed approach could be combined with other strategies that improve ABT efficiency, like dynamic variable ordering [14, 15], or the hybrid ABT version [4]. Finally, privacy deserves a special mention. Obviously, the proposed approach is less private than ABT, since deleted values (and the reasons for their deletion) are exchanged among agents. How privacy could be improved inside the framework of the proposed approach is an open question for further research.

Acknowledgements

Authors sincerely thank reviewers for their comments; they helped us to make a better paper.

References

1. Baker A. B. The hazards of fancy backtracking. *Proc. of AAAI-94*, 288–293, 1994.
2. Bessiere C., Brito I., Maestre A., Meseguer P. The Asynchronous Backtracking without adding links: a new member in the ABT family. *Artifical Intelligence* **161**, 7–24, 2005.
3. Bessiere C., Regin J.C. Refining the basic constraint propagation algorithm. *Proc. IJCAI-01*, 309–315, 2001.
4. Brito I., Meseguer P. Improving ABT performance by adding synchronization points. *Recent Advances in Constraints* in press, 2008.
5. Dechter R. and Pearl J. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, **34**, 1–38, 1988.
6. Debruyne R., Bessiere C. Domain filtering consistencies. *JAIR*, 14:205–230, 2001.
7. Meisels A., Kaplansky E., Razgon I., Zivan R. Comparing Performance of Distributed Constraint Processing Algorithms. *AAMAS Workshop on Distributed Constr. Reas.*, 86–93, 2002.
8. Silaghi M.C., Sam-Haroud D., Faltings B. Asynchronous Search with Aggregations. *Proc. of AAAI-00*, 917–922, 2000.
9. Silaghi M.C., Sam-Haroud D., Faltings B. Consistency Maintenance for ABT. *Proc. of CP-01*, 271–285, 2001.
10. Silaghi M.C., Sam-Haroud D., Faltings B. Asynchronous consistency and maintenance in distributed constraint satisfaction. *Artificial Intelligence*, **161**, 25–53, 2005.
11. Smith B. Phase Transition and the Mushy Region in Constraint Satisfaction Problems. *In Proc. of the 11th ECAI*, 100–104, 1994.
12. Yokoo M., Durfee E., Ishida T., Kuwabara K. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. In *Proc. of the 12th. DCS*, 614–621, 1992.
13. Yokoo M., Durfee E., Ishida T., Kuwabara K. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. Know. and Data Engin.*, **10**, 673–685, 1998.
14. Zivan R., Meisels A. Dynamic ordering for asynchronous backtracking on DisCSPs. *Constraints*, **11**, 179-197, 2006.
15. Zivan R., Zazone M., Meisels A. Min-domain ordering for asynchronous backtracking. *Proc. of CP-07*, 758–772, 2007.