Asynchronous Inter-Level Forward-Checking for DisCSPs

Redouane Ezzahir[1,2], Christian Bessiere[1], Mohamed Wahbi[1,2], Imade Benelallam[2], and El Houssine Bouyakhf[2]

[1] LIRMM/CNRS, U. of Montpellier 2, France
bessiere@lirmm.fr, ezzahir@lirmm.fr, wahbi@lirmm.fr
[2] LIMIARF/FSR, U. of Mohammed V Agdal, Morroco
imade.benelallam@ieee.org, bouyakhf@fsr.ac.ma

Abstract. We propose two new asynchronous algorithms for solving Distributed Constraint Satisfaction Problems (DisCSPs). The first algorithm, AFC-ng, is a nogood-based version of Asynchronous Forward Checking (AFC). The second algorithm, Asynchronous Inter-Level Forward-Checking (AILFC), is based on the AFC-ng algorithm and is performed on a pseudo-tree ordering of the constraint graph. AFC-ng and AILFC only need polynomial space. We compare the performance of these algorithms with other DisCSP algorithms on random DisCSPs in two kinds of communication environments: Fast communication and slow communication. Our experiments show that AFC-ng improves on AFC and that AILFC outperforms all compared algorithms in communication load.

1 Introduction

Distributed Constraint Satisfaction Problems (*DisCSPs*) is a general framework for solving distributed problems. DisCSPs have a wide range of applications in multi-agent coordination, such as distributed resource allocation problems [1], distributed scheduling problems [2], sensor networks [3], and log-based reconciliation [4].

DisCSPs are composed of agents, each holding its local constraint network. Variables in different agents are connected by constraints. Agents assign values to their variables, attempting to generate a locally consistent assignment that is also consistent with all constraints between agents [5, 6]. To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages among them to check consistency of their proposed assignments against constraints among variables that belong to different agents.

Several efficient distributed algorithms for solving DisCSPs have been developed in the last decade. Synchronous Backtrack (SBT) is the simplest DisCSP search algorithm that performs assignments sequentially and synchronously. Only the agent holding a Current Partial Assignment (*CPA*) performs an assignment or backtrack [7]. The first complete asynchronous search algorithm for DisCSPs is the Asynchronous Backtracking ABT [8, 5, 9]. In ABT, agents perform assignments asynchronously and send out messages to constraining agents, informing them about their assignments. Due to the asynchronous nature of agents operations, the global assignment state at any particular time during the run of asynchronous backtracking is in general inconsistent. Nogoods are used to prevent the construction of globally inconsistent solutions. Another promising algorithm for DisCSPs is the Asynchronous Forward-Checking (AFC) algorithm [10, 11]. This algorithm is based on the forward checking (FC) algorithm for CSPs, but performs forward checking asynchronously.

In this paper, we present two new asynchronous algorithms for solving DisCSPs. The first one is based on Asynchronous Forward Checking (AFC) and uses nogood recording. We call it Nogood-Based AFC (AFC-ng). The second one is based on AFC-ng and is performed on a pseudo-tree ordering of the constraint graph. We call it Asynchronous Inter-Level Forward-Checking (AILFC).

This paper is organized as follows. Section 2 gives the necessary background on DisCSPs. Sections 3 and 4 describe the algorithms AFC-ng and AILFC. Correctness proofs are given in Section 5. Section 6 presents an experimental evaluation of our proposed algorithms against three other well-known distributed algorithms. Section 7 summarizes several related works and we conclude the paper in Section 8.

2 Background

2.1 Distributed Constraint Satisfaction Problems

The Distributed Constraint Satisfaction Problem *DisCSP* has been formalized in [6] as a tuple $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{A}$ is a set of agents $\{A_1, \ldots, A_k\}$, $\mathcal{X}$ is a set of variables $\{x_1, \ldots, x_n\}$, where each x_i is controlled by one agent in $\mathcal{A}$. $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of domains, where $D(x_i)$ is a finite set of values to which variable x_i may be assigned. Only the agent who is assigned a variable has control on its value and knowledge of its domain. $\mathcal{C}$ is a set of binary constraints that specify the combinations of values allowed for the two variables they involve. A constraint $c_{ij} \in \mathcal{C}$ between two variables x_i and x_j is a subset of the Cartesian product $D(x_i) \times D(x_j)$.

For simplicity purposes, we consider a restricted version of DisCSP where each agent owns exactly one variable. We identify the agent number with its variable index. We also consider that the total order among agents that is used by a search algorithm is the lexicographic ordering $x_i \prec x_j$ if $i < j$.

We assume that communication between two agents is not necessarily generalized FIFO (aka causal order) channels [12]. Thus, all agents maintain their own counter, called Ctr, and increment it whenever they change their value. The current value of the counter *tags* each generated assignment. An *assignment* for an agent $A_i \in \mathcal{A}$ is a tuple (x_i, v_i, Ctr_i) where v_i is a value from the domain of x_i and Ctr_i is the tag value.

A *nogood* ng for value c for variable x_k is a clause of the form $x_i = a \wedge x_j = b \wedge \ldots \Rightarrow x_k \neq c$, meaning that the assignment $x_k = c$ (i.e., the right hand side $Rhs(ng)$ of ng) is inconsistent with the assignments $x_i = a, x_j = b, \ldots$ (i.e., the left hand side $Lhs(ng)$ of ng). When every value of a variable x_k is ruled out by a nogood, these nogoods are resolved computing a new nogood $newNg$. Let x_j be the lowest variable in the left-hand side of the nogoods, with $x_j = b$. $Lhs(newNg)$ is the conjunction of the left-hand sides of all nogoods except $x_j = b$. $Rhs(newNg)$ is $x_j \neq b$.

Definition 1 (Current Partial Assignment (CPA)). *Given an agent $A_i \in \mathcal{A}$, a CPA is an ordered set of assignments* $\{(x_1, v_1, Ctr_1), \ldots, (x_{i-1}, v_{i-1}, Ctr_{i-1}) \mid x_1 \prec \ldots \prec x_{i-1} \prec x_i\}$.

Definition 2 (AgentView). *The agent view of an agent $A_i \in \mathcal{A}$ stores the newest assignments received from agents that precede A_i in the ordering $\prec$. It has a form similar to a* CPA *and is initialized to the set of empty assignments* $\{(x_j, \emptyset, 0) \mid i \neq j\}$.

Definition 3 (Time-stamp). *A time-stamp is an ordered list of counters* $\langle Ctr_1, Ctr_2, \ldots, Ctr_k\rangle$*. When comparing (lexicographically) two time-stamps, the most up to date is one which is lexicographically greater, that is, the one with greatest value on the first counter on which they differ, if any, otherwise the longest one.*

2.2 Asynchronous Forward-Checking (AFC)

AFC is based on the Forward-Checking (FC) algorithm for CSPs but it performs the forward checking phase asynchronously [10, 11]. As in synchronous backtracking, agents assign their variables only when they hold the current partial assignment (CPA). The CPA is a unique message that is passed from one agent to the next one in the ordering. The CPA carries the partial assignment that agents attempt to extend into a complete solution by assigning their variables on it. Forward checking is performed as follows. Every agent that sends the CPA to its successor also sends copies of the CPA to all agents whose assignments are not yet on the CPA. Agents that received CPAs update domains of their variables, removing all values that are in conflict with assignments on the received CPA.

An agent that generates an empty domain as a result of a forward-checking operation initiates a backtrack by sending *Not_OK* messages which carry the inconsistent partial assignment which caused the empty domain. *Not_OK* messages are sent to all agents with unassigned variables on the (inconsistent) CPA. When an agent holding a *Not_OK* receives a CPA, it sends this CPA back in a backtrack message. When multiple agents reject a given assignment by sending *Not_OK* messages, only the first agent that will receive a CPA and is holding a relevant *Not_OK* message will eventually backtrack. After receiving a new CPA, the *Not_OK* message becomes obsolete when the CPA it carries is no longer a subset of the received CPA.

An improved backtrack method for AFC was described in Section 6 of [11]. Instead of just sending *Not_OK* messages to all agents unassigned in the CPA, the agent who detects the empty domain can itself initiate a backtrack operation. It sends a backtrack message to the last agent assigned in the inconsistent CPA in addition to the *Not_OK* messages to all agents not instantiated in the inconsistent CPA. The agent who receives a backtrack message generates (if it is possible) a new CPA that will dominate older ones thanks to the time-stamp mechanism (see Definition 3).

3 Nogood-based AFC

The nogood-based Asynchronous Forward-Checking (AFC-ng) is based on AFC but it tries to enhance the asynchronism of the forward phase. The two main features of

```
procedure Start()
 1: InitMyAgentView();
 2: $end \leftarrow false$; $myAgentView.Consistent \leftarrow true$;
 3: if($self = IA$) then Assign();
 4: while($\neg end$)
 5:   msg $\leftarrow$ getMsg();
 6:   switch($msg.type$)
 7:     CPA      : ProcessCPA($msg$);
 8:     BackCPA  : ProcessBackCPA($msg$);
 9:     Terminate : ProcessTerminate($msg$);

procedure InitMyAgentView()
10: $myAgentView \leftarrow \{(x_j, \emptyset, 0) \mid x_j \prec self\}$;

procedure Assign()
11: if($\exists v \in myInitialDomain, \nexists ng \in myNogoodStore \mid Rhs(ng) = v$) then
12:   $myValue \leftarrow$ ChooseValue(); /*not eliminated by myNogoodStore*/
13:   $myCtr \leftarrow myCtr$+1; $CPA \leftarrow myAgentView \cup \{(self, myValue, myCtr)\}$;
14:   SendCPA($CPA$);
15: else Backtrack();

procedure SendCPA(CPA)
16: $next \leftarrow$ getNextAgent();
17: if($next = nil$) then BroadcastMsg:Terminate($myAgentView$); $end \leftarrow true$;
18: else for each $x_j \succ self$ do sendMsg:CPA($CPA, next$) to $x_j$;
```

Fig. 1. Nogood-based AFC algorithm running by agent self (Part 1)

AFC-ng are the following. First, an agent finding an empty domain no longer sends *Not_OK* messages. It resolves the nogoods attached to its values and sends the backtrack message to the lower agent in the resolved nogood. Hence, multiple backtracks may be performed at the same time coming from different agents having an empty domain. These backtracks are sent concurrently by these different agents to different destinations. The re-assignments of the destination agents then happen simultaneously and generate several CPAs. However, the CPA coming from the highest level in the search tree will eventually dominate all others. Interestingly, the search process with the new CPA of highest level can use nogoods reported by the (killed) lower level processes, so that it benefits from their computational effort. Second, each time an agent performs a forward-check, it revises its *initial* domain, (including values already removed by a stored nogood) in order to store the best nogoods for removed values (one nogood per value). When comparing two nogoods eliminating the same value, the nogood with the *highest possible lowest variable* involved is selected (HPLV heuristic) [13]. As a result, when an empty domain is found, the resolvent nogood contains variables as high as possible in the ordering, so that the backtrack message is sent as high as possible, thus saving unnecessary search effort [9].

Description of the algorithm

We call $self$ the variable that points to the agent itself. An AFC-ng agent $self$ executes the code shown in Figures 1 and 2. The data structure. $myInitialDomain$ contains all values of the initial domain of $self$. $self$ stores a nogood per removed value in $myNogoodStore$. $self$ calls the procedure Start() in which $self$ initiates its agent view (line 1) by setting counters to zeros (line 10). The agent view contains a consistency flag that represents whether the partial assignment it holds is consistent. If $self$ is the initializing agent (IA), it initiates the search by calling procedure assign() (line 3). All agents performing the main loop wait for messages, and process received messages according to their types (line 4-9).

When calling assign() $self$ tries to find an assignment, which is consistent with its agent view. If $self$ succeeds, it increments its counter Ctr, generates a CPA from its agent view augmented by $self$ assignment (line 13), and then sends forward the CPA to every agent whose assignments are not yet on the CPA, or reports a solution, when the CPA includes all agents assignments (line 17). Before sending any CPA, $self$ attaches to every CPA message the ID of his successor (line 18). Only if the receiver ID equals that attached to the CPA message, the receiver performs an assignment (line 26). When $self$ fails to find a consistent assignment, it calls procedure Backtrack() (line 15).

Agents use time-stamps to detect and discard obsolete CPAs. Function Compare-TimeStamp($view$, CPA) returns the index $splitlevel$ of the first counter on which $view$ and CPA differ if CPA is newest (see Definition 3) or contains $view$ (line 48). If $view$ is newest, it returns -1. When $view$ and CPA are identical or when CPA is included in $view$ CompareTimeStamp returns 0.

Whenever $self$ receives a CPA, procedure ProcessCPA() is called. $self$ checks its agent view status. If it is not consistent and the agent view is a subset of the received CPA, this means that $self$ has already backtracked, then $self$ does nothing (line 19). Otherwise, $self$ compares the time-stamp of its agent view with the one of the received CPA by calling CompareTimeStamp (line 20). If the received CPA is newest, $self$ updates its agent view and marks it consistent (lines 21-22). Procedure Update-MyAgentView (lines 41-43) sets the agent view and the nogood store to be consistent with the received CPA. Each nogood in the nogood store containing a value for a variable different from that received in the CPA will be deleted (line 43). Next, $self$ calls procedure FC_ReviseInitialDomain() (in line 23) to store nogoods for values inconsistent with the new agent view or to try to find a better nogood for values already having one in the nogood store (line 46). A nogood is better according to the *HPLV* heuristic if the lowest variable in the body of the nogood is higher.

When every value of $self$'s variable is ruled out by a nogood (line 24), the procedure Backtrack is called. These nogoods are resolved by computing a new nogood $newNg$ (line 27). If the new nogood is empty, $self$ terminates execution after sending a $Terminate$ message to all agents in the system meaning that problem is unsolvable (line 28). Otherwise, $self$ updates its agent view by removing assignments of every agent that is strictly greater than the last agent (Rhs($newNg$)) in the $newNg$. $self$ also updates its nogood store by removing obsolete nogoods. Finally it marks its agent view as inconsistent and it initiates a backtrack procedure by sending one $BackCPA$ message to the lower priority agent (Rhs($newNg$)) involved in the $newNg$ (line 34).

The $BackCPA$ message carries the $newNg$ and the inconsistent CPA containing assignments of all agents smaller than or equal to $Rhs(newNg)$ in the agent ordering

```
**ProcessCPA**($msg$)
19: if($\neg myAgentView.Consistent \wedge myAgentView \subset msg.CPA$) then return;
20: $splitlevel \leftarrow$ CompareTimeStamp($myAgentView, msg.CPA$);
21: if($splitlevel > 0$) then
22:     UpdateMyAgentView($msg.CPA, splitlevel$); $myAgentView.Consistent \leftarrow true$;
23:     FC_ReviseInitialDomain();
24:     if($\forall v \in myInitialDomain, \exists ng \in myNogoodStore \mid Rhs(ng) = v$) then Backtrack();
25:     else CheckAssign($msg.Next$)

procedure CheckAssign($next$)
26: if($next = self$) then Assign();

procedure Backtrack()
27: $newNg \leftarrow$ solve($myNogoodStore$);
28: if($newNg = empty$) then BroadcastMsg:Terminate($\emptyset$); $end \leftarrow true$; return;
29: for each $x_j \succ$ Rhs($newNg$) do
30:     $myAgentView.Value$[$x_j$]$\leftarrow unknown$ ;
31:     for each $ng \in myNogoodStore$ do
32:         if($x_j \in Lhs(ng)$) then remove($ng, myNogoodStore$);
33: $myAgentView.Consistent \leftarrow false$; $myValue \leftarrow empty$; $CPA \leftarrow myAgentView$;
34: SendMsg:BackCPA($CPA, newNg$) to Rhs($newNg$);

ProcessBackCPA($msg$)
35: if($\neg myAgentView.Consistent \ \wedge \ myAgentView \subset msg.CPA$) then return;
36: $splitlevel \leftarrow$ CompareTimeStamp($myAgentView, msg.CPA$);
37: if($splitlevel = 0 \ \wedge \ myValue =$ RhsValue($msg.Nogood$)) then
38:     add($msg.Nogood, myNogoodStore$); $myValue \leftarrow empty$; Assign();

ProcessTerminate($msg$)
39: $end \leftarrow true$ ; $myValue \leftarrow empty$;
40: if($msg.CPA \neq \emptyset$) then $myValue \leftarrow msg.CPA.$Value[$self$];

procedure UpdateMyAgentView($CPA, splitlevel$)
41: for each $j \geq splitlevel$ do $myAgentView$[$j$]$\leftarrow CPA$[$j$]; /* update value and Ctr */
42: for each $ng \in myNogoodStore$ do
43:     if $Lhs(ng)$ is inconsistent with $myAgentView$ then remove($ng, myNogoodStore$);

procedure FC_ReviseInitialDomain()
44: for each $v \in myInitialDomain$ do
45:     if($\neg$Consistent($v, myAgentView$)) then
46:         store the best nogood for $v$; /* according to the HPLV heuristic*/

function CompareTimeStamp($view, CPA$)
47: from $j \leftarrow 1$ to size($CPA$) do
48:     if ($Ctr(CPA[j]) > Ctr(view[j])$) then return $j$;
49:     else if ($Ctr(CPA[j]) < Ctr(view[j])$) then return $-1$;
50: return $0$;
```

Fig. 2. Nogood-based AFC algorithm running by agent self (Part 2)

(lines 29-30). $self$ remains in an inconsistent state until receiving a new CPA holding at least one agent assignment with counter higher than that in the agent view of $self$ (lines 21-22).

When a $BackCPA$ message is received, $self$ checks the validity of received $BackCPA$ using agent view status and time-stamp (lines 35-36). If $BackCPA$ is accepted (line 37), $self$ removes its last assignment, adds attached nogood to its nogood store, and calls the procedure assign() (line 38).

ProcessTerminate procedure is called when an agent receives a $Terminate$ message. It marks end flag $true$ to stop the main loop (line 39). If attached CPA is empty then there is no solution. Otherwise, agent solution is retrieved from the CPA (line 40).

4 Asynchronous Inter Level Forward-Checking

A DisCSP can be represented by a constraint graph $G = (X, E)$, whose nodes represent the variables and edges represent the constraints (that is, $X = \mathcal{X}$ and $\{x_i, x_j\} \in E \Leftrightarrow c_{ij} \in \mathcal{C}$). The graph can be re-arranged to form a pseudo-tree [14]. A *pseudo-tree* $G_{PT} = (X, r, E, U)$ for the graph G is defined by a root node $r \in X$ and a directed tree $T = (X, U)$ rooted in r such that for any edge $\{x_i, x_j\} \in E$, x_i and x_j are not in different branches of T. For any arc $(x_i, x_j) \in U$, the node x_i is the parent of the node x_j. If x_i is the parent of x_j, then x_j is a child of x_i. A node x_i is an ancestor of a node x_j if x_i is the parent of x_j or an ancestor of the parent of x_j. A node x_j is a descendant of a node x_i if x_i is an ancestor of x_j. A leaf is a node that has no child. In our implementation, the pseudo-tree is built by a DFS traversal of the graph. Thus, we have $U \subseteq E$.

The AILFC algorithm is based on AFC-ng performed on a pseudo-tree ordering of the constraint graph (built in a preprocessing step). Agents are prioritized according to the pseudo-tree ordering in which each agent has a single parent and various children. Using this priority ordering, AILFC performs multiple AFC-ng processes on the paths from the root to the leaves. The root initiates the search by generating a CPA, assigning its value on it, and sending CPA messages to its linked descendants (including its children) that share a constraint with it. Each child that receives a copy of the CPA performs AFC-ng on the sub-problem restricted to its ancestors (agents that are assigned in the CPA) and the set of its descendants. Therefore, instead of giving the privilege of assigning to only one agent, all agents who are in disjoint subtrees may assign their variables simultaneously. So, the Inter-Level Forward Checking is performed asynchronously on each path from the root to any leaf. AILFC thus exploits the potential speed-up of a parallel exploration in the processing of distributed problems.

An execution of AILFC on a sample DisCSP problem is shown in Figure 3. At time t_1, the root x_1 sends copies of the CPA on messages to its linked descendants (including its children). Children x_2, x_3 and x_4 assign their values simultaneously in the received CPAs and then perform concurrently the AILFC algorithm. Agents x_7, and x_9 only perform a forward- checking. At time t_2, x_9 finds an empty domain and sends a $BackCPA$ message to x_1. At the same time, other CPAs propagate down through the other paths. For instance, a CPA has propagated down from x_3 to x_7 and x_8. x_7 detects an empty domain and sends a nogood to x_3 attached on a $BackCPA$ message.

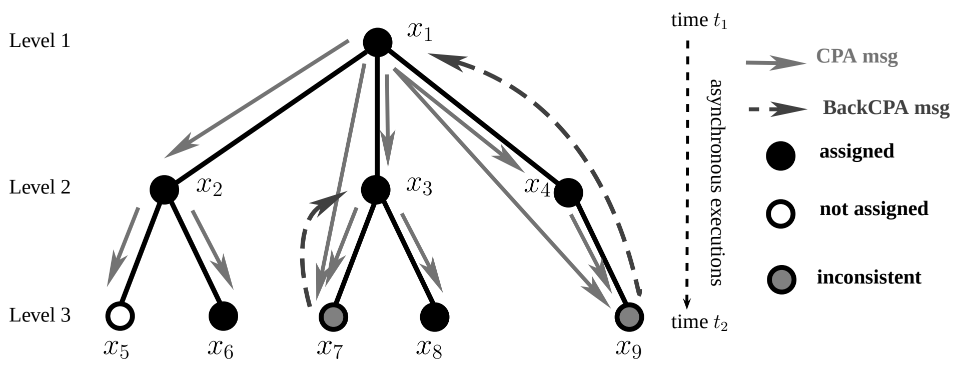


Fig. 3. An example of the AILFC execution

For the CPA that propagates on the path (x_1, x_2, x_5) (resp. (x_1, x_2, x_6)), x_5 (resp. x_6) successfully assigned its value and initiated a solution detection. However, when x_1 receives the $BackCPA$ from x_9, it initiates a new search process by sending a new copy of the CPA which will kill any CPA where x_1 is assigned its old value.

In AFC-ng, a solution is reached when the last agent receives the CPA and succeeds in assigning its variable. In AILFC, the situation is different because a CPA can reach a leaf without being complete. When all agents are assigned and no constraint is violated, this state is a global solution and the network has reached quiescence, meaning that no message is traveling through it. Such a state can be detected using specialized snapshot algorithms [15], but AILFC uses a different mechanism that allows to detect solutions before quiescence. AILFC uses an additional type of message called $Accepted$ that inform parents of the acceptance of their CPA. Termination can be inferred earlier and the number of messages required for termination detection can be reduced. A similar technique of solution detection was used in the AAS algorithm [16].

The mechanism of solution detection is as follows: whenever a leaf node succeeds in assigning its value, it sends an $Accepted$ message to its parent. This message contains the CPA that was received from the parent incremented by the value-assignment of the leaf node. When a non-leaf agent $self$ receives $Accepted$ messages from all its children that are all compatible with each other, all compatible with $self$'s agent view and with $self$'s value, $self$ builds an $Accepted$ message being the conjunction of all received $Accepted$ messages plus $self$'s value-assignment. If $self$ is the root a solution is found, and $self$ broadcasts this solution to all agents. Otherwise, $self$ sends the built $Accepted$ to its parent.

Description of the algorithm

A preprocessing step before starting the AILFC algorithm is performed to convert the constraint graph into a pseudo-tree. $Children(self) \subset \mathcal{A}$ is the set of children of agent $self$ in the pseudo-tree, $Desc(self)$ is the set of its descendants and $linkedDesc(self) \subset Desc(self)$ is the set of its descendants (including its children) that are constrained with $self$. $Parent(self) \in \mathcal{A}$ is the parent of agent $self$ and $Ancestors(self) \subset \mathcal{A}$ is the set of its ancestors (including its parent).

In Figure 4, we present only the procedures that are new or different from those of AFC-ng in Figures 1 and 2. In InitMyAgentView(), the agent view of $self$ is initialized

```
procedure Start()
..:
10:     Accepted  : ProcessAccepted($msg$);

procedure InitMyAgentView()
11: $myAgentView \leftarrow \{(x_j, \emptyset, 0) \mid x_j \in$ Ancestors($self$)$\}$;
12: for each $child \in$ children($self$) accepted[$child$]$\leftarrow \emptyset$;  /*For Solution Detection*/

procedure SendCPA($CPA$)
13: if(children($self$)= $\emptyset$) then
14:   SolutionDetection();
15: else  for each $desc \in$ linkedDesc($self$)do sendMsg:CPA($CPA, self$) to $desc$;

procedure CheckAssign($ancestor$)
16: if(Parent($self$)= $ancestor$) then Assign();

procedure SolutionDetection()
17: if(children($self$) = $\emptyset$) then
18:   SendAccepted($myAgentView \cup \{(self, myValue, myCtr)\}$, $self$) to Parent($self$);
19: else $PA \leftarrow$ BuildAccepted();
20: if($PA \neq \emptyset$)then
21:   if($self = root$) then Broadcast(Terminate, $PA$); $end \leftarrow true$;
22:   else SendAccepted($PA$, $self$) to Parent($self$);

ProcessAccepted($msg$)
23: if(accepted[$msg.Sender$]=$\emptyset\vee$CompareTimeStamp($msg.CPA$,accepted[$msg.Sender$])$>$0)then
24:   accepted[$msg.Sender$] $\leftarrow msg.CPA$;
25:   SolutionDetection();

function BuildAccepted()
26: $PA \leftarrow myAgentView \cup \{(self, myValue, myCtr)\}$;
27: for each $child \in$ children($self$) do
28:   if(accepted[$child$]= $\emptyset \vee \neg$Compatible($PA$,accepted[$child$])) return $\emptyset$;
29:   else $PA \leftarrow PA \cup$ accepted[$child$];
30: return $PA$
```

Fig. 4. New lines/procedures of AILFC with respect to AFC-ng.

to the set $Ancestors(self)$. Ctr is set to 0 for each agent in $Ancestors(self)$ (line 11). The new data structure storing the received $Accepted$ messages is initialized to the empty set (line 12). In SendCPA(CPA), instead of sending copies of the CPA to all agents not yet instantiated on it, $self$ sends copies of the CPA only to its linked descendants ($linkedDesc(self)$) (line 15). When the set $linkedDesc(self)$ is empty (i.e., $self$ is a leaf), $self$ calls the procedure SolutionDetection to build and send an $Accepted$ message. In CheckAssign($ancestor$), $self$ assigns its value if the CPA was received from its parent (line 16) (i.e., if $ancestor$ is the parent of $self$).

In SolutionDetection(), if $self$ is a leaf ($Children(self)$ is empty), it sends an $Accepted$ message to its parent. The $Accepted$ message sent by $self$ contains its agent view incremented by its assignment (lines 17-18). If $self$ is not a leaf, it calls the BuildAccepted() procedure to build an accepted partial solution PA (line 19). If the returned partial solution PA is not empty and $self$ is the root, PA is a solution of the problem. Then, $self$ broadcasts it to other agents including the system agent and sets the end flag to $true$ (line 21). Otherwise, $self$ sends an $Accepted$ message containing PA to its parent (line 22).

In ProcessAccepted(msg), when $self$ receives an $Accepted$ message from its $child$ for the first time, or when msg is newer than that received before (lines 23-24), $self$ stores the content of this message and calls the SolutionDetection procedure (line 25).

In BuildAccepted(), if an accepted partial solution is reached. $self$ generates a partial solution PA incrementing its agent view with its assignment (line 26). Next, $self$ loops over the set of $Accepted$ messages received from its children. If at least one $child$ has never sent an $Accepted$ message or the $Accepted$ message is incompatible with PA, then the partial solution has not yet been reached and the function returns empty (lines 27-28). Otherwise, the partial solution PA is incremented by the $Accepted$ message of $child$ (line 29). Finally, the accepted partial solution is returned (line 30).

5 Correctness Proofs

Theorem 1. *AFC-ng is sound, complete, and terminates.*

The argument for soundness is close to the one given in [11, 17]. The fact that agents only forward consistent partial solution on the CPAs messages at only one place in function assign() (line 14), implies that the agents receive only consistent assignments. A solution is reported by the last agent only in function SendCPA(CPA) at line 17. At this point, all agents have assigned their variables, and their assignments are consistent. Thus the AFC-ng algorithm is sound.

For completeness, we need to show that AFC-ng is able to terminate and does not report inconsistency if a solution exists.

Lemma 1. *AFC-ng is guaranteed to terminate.*

For sake of clarity, we assume that the order in which AFC-ng assigns the variables is the lexicographic ordering $X_1, X_2, \ldots, X_n$. We define the total order o on CPAs as follows. Let I_1 be an assignment on $X_1, \ldots, X_{k_1}$, I_2 be an assignment on $X_1, \ldots, X_{k_2}$, and s be the smallest index on which I_1 and I_2 differ. $I_1 \prec_o I_2$ if and only if $s = k_1 + 1$ or the value $I_1[s]$ is chosen before the value $I_2[s]$ by the value ordering heuristics on variable X_s given the CPA $I_1[1..s-1]$.

To prove the lemma we prove that AFC-ng performs a finite number of backtrack steps. In AFC-ng, several backtracks can be performed simultaneously as they are generated concurrently by different agents to different destinations. The re-assignments of destination agents then happen simultaneously, generating several CPAs. However, the CPA at the highest level in the search hierarchy tree will eventually dominate all others thanks to its greater time-stamp (see line 21 in Figure 2). Thus, every backtrack step may be represented by the backtrack at the highest level. The agent X_i who has received that backtrack of highest level has to replace its previous assignment v_i in the CPA by a new one v_i' because the backtrack message contains a nogood rejecting value v_i. If v_i was not the first value chosen by X_i since it has received the current CPA from

X_{i-1} then we know that all other values v_j preferred to v_i were ruled out by a nogood at the time v_i was chosen. Now, the CPA on $X_1, \ldots, X_{i-1}$ has not changed since then, otherwise this would not be the highest backtrack. As a result, the nogoods rejecting values v_j preferred to v_i are still valid and v'_i is necessarily the *next* preferred value in the heuristic order. By definition of the order o, the new CPA obtained is greater than the previous one according to o because it has not changed on $X_1, \ldots, X_{i-1}$ and v'_i is less preferred than v_i. Since o is a total order and since there are a finite number of variables and a finite number of values per variable, there will be a finite number of new CPAs generated. Now, each backtrack of highest level generates a new CPA. Thus, AFC-ng performs a finite number of backtracks.

Lemma 2. *AFC-ng cannot infer inconsistency if a solution exists.*

Whenever a newer CPA or a $BackCPA$ message is received, AFC-ng agent updates its nogood store. Hence, for every CPA that may potentially lead to a solution, agents only store valid nogoods. In addition, every nogood resulting from a CPA is redundant with regard to the DisCSP to solve. Since all additional nogoods are generated by logical inference when a domain wipe-out occurs, the empty nogood cannot be inferred if the network is satisfiable. This mean that AFC-ng is able to produce all solutions.

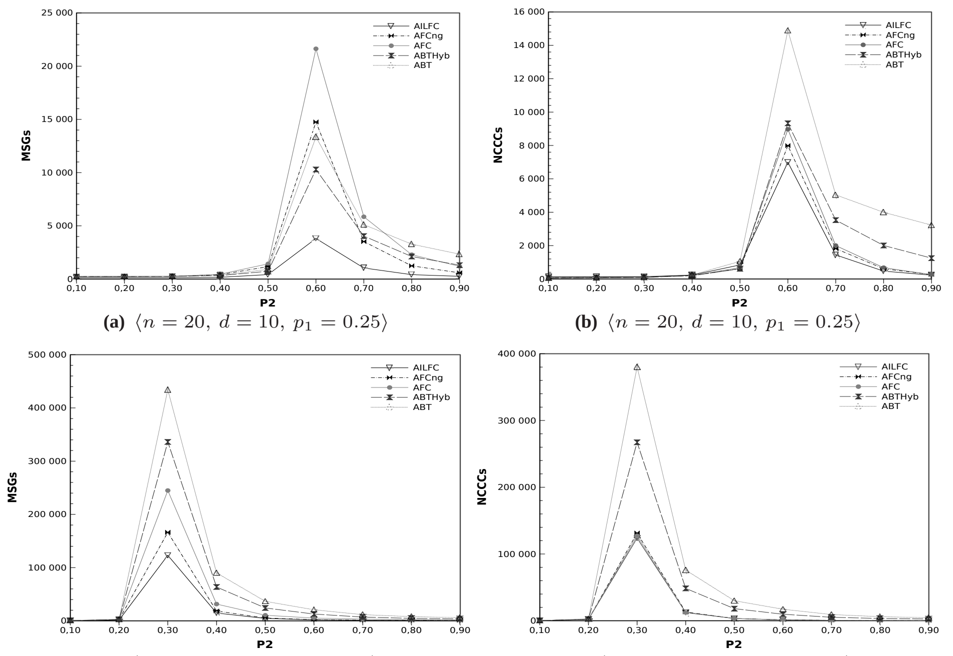
Theorem 2. *AILFC algorithm is sound, complete, and terminates.*

AILFC agents only forward consistent partial assignments (CPAs). Hence, leaf agents receive only consistent CPAs. Thus, leaf agents send Accepted message only holding consistent assignments to their parent. Since a parent builds an $Accepted$ message only when the $Accepted$ messages received from all its children are compatible with each other and all compatible with its own value, the $Accepted$ message it sends contains a consistent partial solution. The root broadcasts a solution only when it can build itself such an $Accepted$ message. Therefore, the solution is correct and AILFC is sound.

AILFC performs multiple AFC-ng processes on the paths of the pseudo-tree from the root to the leaves. Thus, it inherits the completeness property of AFC-ng (empty nogood cannot be inferred if the network is satisfiable (see Lemma 2). It also appears that the agent of high priority cannot fall into an infinite loop. By induction on the level of the pseudo-tree no agent can fall in such a loop, which ensures the termination of AILFC.

6 Experimental Evaluation

In this section we compare experimentally AFC-ng and AILFC to three other algorithms: AFC, ABT, and ABT-Hyb [18]. Algorithms are tested on the same static agents ordering using *max-degree* heuristic and the same nogood selection heuristic (*HPLV*). For ABT and ABT-Hyb we implemented an improved version of Silaghi's solution detection [12] and counters for tagging assignments. This allows to better treat non-causal order channels [12]. All experiments were performed on the DisChoco platform [19] in which agents are simulated by Java threads that communicate only through message passing.



(a) $\langle n = 20,\ d = 10,\ p_1 = 0.25\rangle$ **(b)** $\langle n = 20,\ d = 10,\ p_1 = 0.25\rangle$

(c) $\langle n = 20, d = 10, p_1 = 0.75\rangle$ **(d)** $\langle n = 20,\ d = 10,\ p_1 = 0.75\rangle$

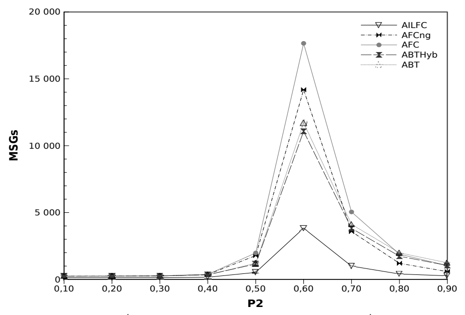
Fig. 5. Total number of messages sent and NCCCs on fast communication

The algorithms are tested on uniform binary random DisCSPs which are characterized by $\langle n,\ d,\ p_1,\ p_2\rangle$, where n is the number of agents/variables, d the number of values per variable, p_1 the network connectivity defined as the ratio of existing binary constraints, and p_2 the constraint tightness defined as the ratio of forbidden value pairs. We solved 100 instances of two classes of constraints graph: sparse graph $\langle 20, 10, 0.25, p_2\rangle$ and dense graph $\langle 20, 10, 0.75, p_2\rangle$. We vary the tightness from 0.10 to 0.90 by steps of 0.10.

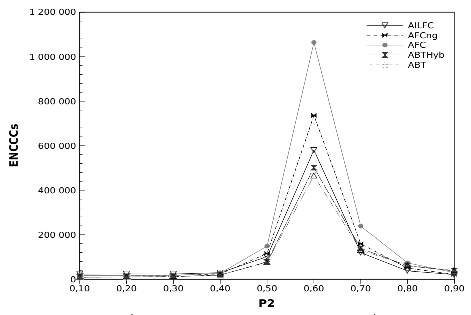
We evaluate the algorithms performance by the average of total messages sent [20] (including system messages) and the average of Equivalent Non-Concurrent Constraint Checks (ENCCCs) [21]. ENCCCs are a weighted sum of processing and communication time. We simulate two scenarios of communication: fast communication (where message delay is null and ENCCCs reduce to standard NCCCs), and slow communication with uniform random message delay (where the cost of the delay is between 500 and 1000 constraint checks.)

Fast communication

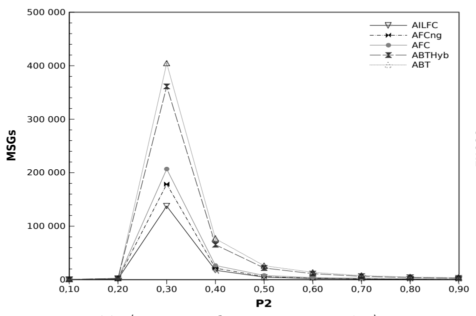
Figure 5 presents performance of AILFC, AFC-ng, AFC, ABT and ABT-hyb running on a fast communication environment. The figure shows that in both types of constraint graphs (sparse and dense), AILFC has the lowest communication load (#MSGs). Concerning NCCCs, AILFC is the fastest algorithm on sparse graphs (Fig. 5(b)). On dense



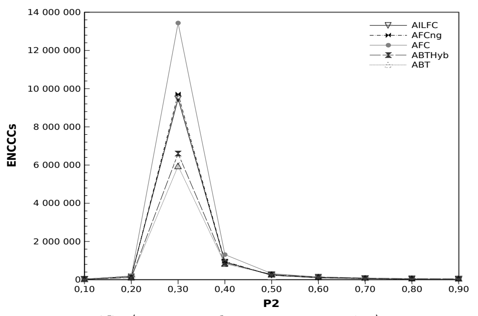
(a) $\langle n = 20,\ d = 10,\ p_1 = 0.25\rangle$



(b) $\langle n = 20,\ d = 10,\ p_1 = 0.25\rangle$



(c) $\langle n = 20,\ d = 10,\ p_1 = 0.75\rangle$



(d) $\langle n = 20,\ d = 10,\ p_1 = 0.75\rangle$

Fig. 6. Total number of messages sent and ENCCCs on slow communication

graphs AILFC behaves like AFC and AFC-ng. Comparing AFC-ng with AFC, Fig. 5 shows that they perform the same number of NCCCs but AFC-ng exchanges less messages than AFC. Comparing AFC-ng with ABT and ABT-hyb, Fig. 5 shows that in both types of constraint graphs AFC-ng is faster than ABT-hyb and ABT. However, on sparse graphs, Fig 5(a) shows that AFC-ng sends more messages than ABT-hyb and ABT.

Slow communication

In Figure 6 we report experimental results with slow communication. The figure shows that AILFC is again the best algorithm in terms of number of messages. Concerning ENCCCs, as in the fast communication environment, AILFC is faster than or equal to AFC and AFC-ng depending on whether the graph is sparse or dense. The comparison of AFC and AFC-ng shows a pattern close to the one observed with fast communication: AFC-ng is better, or slightly better, both in terms of messages and ENCCCs. The main difference between fast and slow communication is the performance of ABT and ABT-hyb. Whereas they remain expensive in terms of messages, they become the best algorithms in terms of ENCCCs, with a slight advantage to ABT. This confirms that in slow communication environment, the more the algorithm is asynchronous, the better it is.

Discussion

A first observation on these experiments is that ABT, ABT-hyb on one side, and AFC, AFC-ng on the other side, show quite opposite patterns. If message passing is not an issue, ABT and ABT-hyb are good choices with slow communication whereas AFC and AFC-ng are good when communication is fast. A second observation is that AILFC is always better than or equivalent to AFC-ng, which is better than or equivalent to AFC, both in terms of messages and amount of processing (ENCCCs). If limiting the communication load is important, AILFC is the best among all both for fast and slow communication. AILFC benefits both from running separate search processes in disjoint problem subtrees, which pays off when a graph is sparse, and from using the same mechanism as AFC-ng, which pays off when agents are highly connected (dense graphs).

7 Other Related Work

In [18, 7] the performance of asynchronous (ABT), synchronous (Synchronous Conflict BackJumping (SCBJ)), and hybrid approaches (ABT-Hyb) was studied. It is shown that ABT-Hyb improves over ABT and that SCBJ requires less communication effort than ABT-Hyb. In Interleaved Asynchronous Backtracking (IDIBT) [22], agents participate in multiple processes of asynchronous backtracking. Each agent keeps a separate *AgentView* for each search process in IDIBT. The number of search processes is fixed by the first agent in the ordering. The performance of concurrent asynchronous backtracking [22] was tested and found to be ineffective for more than two concurrent search processes [22]. Dynamic Distributed BackJumping (DDBJ) was presented in [17]. It is an improved version of the basic AFC. It combines the concurrency of an asynchronous dynamic backjumping algorithm, and the computational efficiency of the AFC algorithm, coupled with the *possible conflict heuristics* of dynamic value and variable ordering. As in DDBJ, AFC-ng performs several backtracks simultaneously. However, AFC-ng should not be confused with DDBJ. DDBJ is based on dynamic ordering and requires additional messages to compute ordering heuristics. In AFC-ng, all agents that received a $BackCPA$ message continue search concurrently. Once a more up to date CPA is received by an agent, all nogoods already stored can be kept if consistent with that CPA.

8 Conclusion

Two new complete, asynchronous algorithms are presented. The first algorithm, Nogood-Based Asynchronous Forward Checking (AFC-ng), is an improvement on AFC. The second, Asynchronous Inter-Level Forward-Checking (AILFC), is based on AFC-ng and is performed on a pseudo-tree re-arrangement of the constraint graph. Experiments ran on random DisCSPs show that AFC-ng improves AFC both in fast and slow communication environments. Experiments show that AILFC is the more robust algorithm in both communication types. In particular, it is the best in terms of messages sent. In slow communication environments, the performance of algorithms that perform variable assignments sequentially deteriorates. This is observed for AFC and AFC-ng, and, less significantly for ABT-Hyb, when compared to ABT.

References

1. Petcu, A., Faltings, B.: A value ordering heuristic for distributed resource allocation. In: Proceeding of CSCLP04, Lausanne, Switzerland (2004)
2. Wallace, R.J., Freuder, E.: Constraint-based multi-agent meeting scheduling: effects of agent heterogeneity on performance and privacy loss. In: Proceeding of the 3rd workshop on distributed constrait reasoning, DCR-02, Bologna (2002) 176–182
3. Fernandez, C., Bejar, R., Krishnamachari, B., Gomes, K.: Communication and computation in distributed CSP algorithms. In: Proceeding of CP-02, Ithaca, NY, USA (2002) 664–679
4. Chong, Y.L., Hamadi, Y.: Distributed log-based reconciliation. In: ECAI. (2006) 108–112
5. Yokoo, M.: Algorithms for distributed constraint satisfaction problems: A review. Autonomous Agents & Multi-Agent Systems **3** (2000) 198–212
6. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: Distributed constraint satisfaction problem: Formalization and algorithms. IEEE Trans. on Data and Kn. Eng. **10** (1998) 673–685
7. Zivan, R., Meisels, A.: Synchronous vs asynchronous search on DisCSPs. In: Proceeding of 1st European Workshop on Multi Agent System, EUMAS, Oxford (2003)
8. Yokoo, M., Durfee, E.H., Ishida, T., , Kuwabara., K.: Distributed constraint satisfaction for formalizing distributed problem solving. In: IEEE Intern. Conf. Distrb. Comp. Sys. (1992) 614–621
9. Bessiere, C., Maestre, A., Brito, I., Meseguer, P.: Asynchronous backtracking without adding links: a new member in the ABT family. Artificial Intelligence **161:1-2** (2005) 7–24
10. Meisels, A., Zivan, R.: Asynchronous forward-checking for distributed CSPs. In Zhang, W., ed.: Frontiers in Artificial Intelligence and Applications, IOS Press (2003)
11. Meisels, A., Zivan, R.: Asynchronous forward-checking for DisCSPs. Constraints **12**(1) (2007) 131–150
12. Silaghi, M.C.: Generalized dynamic ordering for asynchronous backtracking on DisCSPs. In: DCR workshop, AAMAS-06, Hakodate, Japan (2006)
13. Maestre, A., Bessiere, C.: Improving asynchronous backtracking for dealing with complex local problems. In: ECAI. (2004) 206–210
14. Freuder, E.C., Quinn, M.J.: Taking advantage of stable sets of variables in constraint satisfaction problems. In: In IJCAI85. (1985) 1076–1078
15. Mani, K.C., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1) (1985) 63–75
16. Silaghi, M.C., Faltings, B.: Asynchronous aggregation and consistency in distributed constraint satisfaction. Artificial Intelligence **161:1-2** (2005) 25–54
17. Nguyen, T., Sam-Hroud, D., Faltings, B.: Dynamic distributed backjumping. In: Proceeding of 5th workshop on DCR-04, Toronto (2004)
18. Brito, I., Meseguer, P.: Synchronous, asynchronous and hybrid algorithms for DisCSP. In: Workshop on DCR-04, CP-2004, Toronto (2004)
19. Ezzahir, R., Bessiere, C., Belaissaoui, M., Bouyakhf, E.H.: Dischoco: a platform for distributed constraint programming. In: Proceeding of Workshop on Distributed Constraint Reasoning of IJCAI-07. (2007) 16–21
20. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Series (1997)
21. Chechetka, A., Sycara, K.: No-commitment branch and bound search for distributed constraint optimization. In: Proc. of AAMAS '06, New York, NY, USA, ACM (2006) 1427–1429
22. Hamadi, Y.: Interleaved backtracking in distributed constraint networks. Int.. J. of AI Tools **11** (2002) 167–188