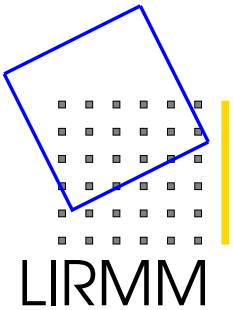
Nogood-Based Asynchronous Forward-Checking Algorithms

Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, El Houssine Bouyakhf



Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier

LIRMM, University Montpellier 2, CNRS

TECHNICAL REPORT

Nogood-Based Asynchronous Forward-Checking Algorithms

Mohamed Wahbi[1,3] **Redouane Ezzahir**[2]
Christian Bessiere[1] **El Houssine Bouyakhf**[3]

[1]**LIRMM/CNRS, University of Montpellier, France**
[2]**ENSA Agadir, University Ibn Zohr, Morroco**
[3]**LIMIARF/FSR, University Mohammed V Agdal, Morocco**

April 2012 R.R.LIRMM RR-12013

161, Rue Ada - 34095 Montpellier Cedex 5 - France
Tél: +33 (0)4 67 41 85 85 - Fax: +33 (0)4 67 41 85 00

Abstract

We propose two asynchronous algorithms for solving Distributed Constraint Satisfaction Problems (DisCSPs). The first algorithm, AFC-ng, is a nogood-based version of Asynchronous Forward Checking (AFC). Besides its use of nogoods as justification of value removals, AFC-ng allows simultaneous backtracks going from different agents to different destinations. The second algorithm, Asynchronous Forward-Checking Tree (AFC-tree), is based on the AFC-ng algorithm and is performed on a pseudo-tree ordering of the constraint graph. AFC-tree runs simultaneous search processes in disjoint problem subtrees and exploits the parallelism inherent in the problem. We prove that AFC-ng and AFC-tree only need polynomial space. We compare the performance of these algorithms with other DisCSP algorithms on random DisCSPs and instances from real benchmarks: sensor networks and distributed meeting scheduling. Our experiments show that AFC-ng improves on AFC and that AFC-tree outperforms all compared algorithms, particularly on sparse problems.

1 Introduction

Constraint programming is an area in computer science that has gained increasing interest in recent years. Constraint programming is based on its powerful framework named *constraint satisfaction problem* (CSP). CSP is a general framework that can formalize many real world combinatorial problems such as resource allocation, car sequencing, natural language understanding, machine vision, etc. A constraint satisfaction problem consists in looking for solutions to a constraint network, that is, a set of assignments of values to variables that satisfy the constraints of the problem. These constraints represent restrictions on values combinations allowed for constrained variables.

There exist applications that are of a distributed nature. In this kind of applications the knowledge about the problem, that is, variables and constraints, is distributed among physical distributed agents. This distribution is mainly due to privacy and/or security requirements: constraints or possible values may be strategic information that should not be revealed to other agents that can be seen as competitors. Several applications in multi-agent coordination are of such kind. Examples of applications are sensor networks [15, 2], military unmanned aerial vehicles teams [15], distributed scheduling problems [27, 18], distributed resource allocation problems [23], log-based reconciliation [9], Distributed Vehicle Routing Problems [16], etc. Therefore, the distributed framework *distributed constraint satisfaction problems* (DisCSP) is used to model and solve this kind of problems.

A DisCSP is composed of a group of autonomous agents, where each agent has control of some elements of information about the whole problem, that is, variables and constraints. Each agent owns its local constraint network. Variables in different agents are connected by constraints. Agents must assign values to their variables so that all constraints are satisfied. Hence, agents assign values to their variables, attempting to generate a locally consistent assignment that is also consistent with constraints between agents [30, 28]. To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages to check consistency of their proposed

assignments against constraints that contain variables that belong to other agents.

Several distributed algorithms for solving DisCSPs have been developed in the last two decades. The first complete asynchronous search algorithm for solving DisCSPs is Asynchronous Backtracking (ABT) [29, 28, 4]. ABT is an asynchronous algorithm executed autonomously by each agent in the distributed problem. Synchronous Backtrack (SBT) is the simplest DisCSP search algorithm. SBT performs assignments sequentially and synchronously. In SBT, only the agent holding a current partial assignment (CPA) performs an assignment or backtrack [31].

Extending SBT, Meisels and Zivan (2007) proposed the Asynchronous Forward-Checking (AFC). Besides assigning variables sequentially as is done in SBT, agents in AFC perform forward checking (FC [13]) asynchronously. The key here is that each time an agent succeeds to extend the current partial assignment (by assigning its variable), it sends the CPA to its successor and copies of this CPA to all agents whose assignments are not yet on the CPA. When an agent receives a copy of the CPA, it performs the forward checking phase. In the forward checking phase all inconsistent values with assignments on the received CPA are removed. The forward checking operation is performed asynchronously where comes the name of the algorithm. When an agent generates an empty domain as a result of a forward checking, it informs all agents with unassigned variables on the (inconsistent) CPA. Afterwards, one of these agents will receive the CPA and will backtrack. Thereby, only one backtrack can be generated for a given CPA. Meisels and Zivan have shown in [21] that AFC is computationally more efficient than ABT. However, due to the manner in which the backtrack operation is performed, AFC does not draw all the benefit it could from the asynchronism of the FC phase.

In this work, we present two asynchronous algorithms for solving DisCSPs. The first one is based on Asynchronous Forward Checking (AFC) and uses nogood as justifications of value removals. We call it Nogood-Based Asynchronous Forward Checking (AFC-ng). Unlike AFC, AFC-ng allows concurrent backtracks to be performed at the same time coming from different agents having an empty domain to different destinations. As a result, several CPAs could be generated simultaneously by the destination agents. Thanks to the timestamps integrated in the CPAs, the *strongest* CPA coming from the highest level in the agent ordering will eventually dominate all others. Interestingly, the search process with the strongest CPA will benefit from the computational effort done by the (killed) lower level processes. This is done by taking advantage from nogoods recorded when processing these lower level processes.

The second algorithm we propose is based on AFC-ng and is named Asynchronous Forward-Checking Tree (AFC-tree). The main feature of the AFC-tree algorithm is using different agents to search non-intersecting parts of the search space concurrently. In AFC-tree, agents are prioritized according to a pseudo-tree arrangement of the constraint graph. The pseudo-tree ordering is build in a preprocessing step. Using this priority ordering, AFC-tree performs multiple AFC-ng processes on the paths from the root to the leaves of the pseudo-tree. The agents that are brothers are committed to concurrently find the partial solutions of their variables. Therefore, AFC-tree exploits the potential

speed-up of a parallel exploration in the processing of distributed problems [11]. A solution is found when all leaf agents succeed in extending the CPA they received. Furthermore, in AFC-tree privacy may be enhanced because communication is restricted to agents in the same branch of the pseudo-tree.

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

2 Background

2.1 Basic definitions and notations

The *distributed constraint satisfaction problem* (DisCSP) has been formalized in [30] as a tuple $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{A}$ is a set of a agents $\{A_1, \ldots, A_a\}$, $\mathcal{X}$ is a set of n variables $\{x_1, \ldots, x_n\}$, where each variable x_i is controlled by one agent in $\mathcal{A}$. $\mathcal{D}^0 = \{D^0(x_1), \ldots, D^0(x_n)\}$ is a set of n domains, where $D^0(x_i)$ is the initial set of possible values to which variable x_i may be assigned. During search, values may be pruned from the domain. At any node, the set of possible values for variable x_i is denoted by $D(x_i)$ and is called the current domain of x_i. Only the agent who is assigned a variable has control on its value and knowledge of its domain. $\mathcal{C}$ is a set of constraints that specify the combinations of values allowed for the variables they involve. In this paper, we assume a binary distributed constraint network where all constraints are binary constraints (they involve two variables). A constraint $c_{ij} \in \mathcal{C}$ between two variables x_i and x_j is a subset of the Cartesian product of their domains ($c_{ij} \subseteq D^0(x_i) \times D^0(x_j)$). The connectivity between the variables can be represented with a constraint graph G, where vertices represent the variables and edges represent the constraints [10].

For simplicity purposes, we consider a restricted version of DisCSP where each agent controls exactly one variable ($a = n$). Thus, we use the terms agent and variable interchangeably and we identify the agent ID with its variable index. Furthermore, all agents store a unique total order $\prec$ on agents. Thus, agents appearing before an agent $A_i \in \mathcal{A}$ in the total order are the higher agents (predecessors) and conversely the lower agents (successors) are agents appearing after A_i. For sake of clarity, we assume that the total order is the lexicographic ordering $[A_1, A_2, \ldots, A_n]$. Each agent maintains a counter, and increments it whenever it changes its value. The current value of the counter *tags* each generated assignment.

Definition 1 *An **assignment** for an agent $A_i \in \mathcal{A}$ is a tuple (x_i, v_i, t_i), where v_i is a value from the domain of x_i and t_i is the tag value. When comparing two assignments, the most up to date is the one with the greatest tag t_i.*

Definition 2 *A **current partial assignment** CPA is an ordered set of assignments* $\{[(x_1, v_1, t_1), \ldots, (x_i, v_i, t_i)] \mid x_1 \prec \cdots \prec x_i\}$. *Two CPAs are **consistent** if they do not disagree on any variable value.*

Definition 3 *A **timestamp** associated with a CPA is an ordered list of counters* $[t_1, t_2, \ldots, t_i]$ *where* t_j *is the tag of the variable* x_j. *When comparing two CPAs, the **strongest** one is that associated with the lexicographically greater timestamp. That is, the CPA with greatest value on the first counter on which they differ, if any, otherwise the longest one.*

Definition 4 *The **AgentView** of an agent* $A_i \in \mathcal{A}$ *stores the most up to date assignments received from higher priority agents in the agent ordering. It has a form similar to a CPA and is initialized to the set of empty assignments* $\{(x_j, empty, 0) \mid x_j \prec x_i\}$.

During search agents can infer inconsistent sets of assignments called nogoods.

Definition 5 *A **nogood** ruling out value* v_k *from the initial domain of variable* x_k *is a clause of the form* $x_i = v_i \wedge x_j = v_j \wedge \ldots \rightarrow x_k \neq v_k$, *meaning that the assignment* $x_k = v_k$ *is inconsistent with the assignments* $x_i = v_i, x_j = v_j, \ldots$. *The* left hand side (lhs) *and the* right hand side (rhs) *are defined from the position of* $\rightarrow$.

The current domain $D(x_i)$ of a variable x_i contains all values from the initial domain $D^0(x_i)$ that are not ruled out by a nogood. When all values of a variable x_k are ruled out by some nogoods ($D(x_i) = \emptyset$), these nogoods are resolved, producing a new nogood (ng). Let x_j be the lowest variable in the left-hand side of all these nogoods and $x_j = v_j$. The $lhs(ng)$ is the conjunction of the left-hand sides of all nogoods except $x_j = v_j$ and $rhs(ng)$ is $x_j \neq v_j$.

2.2 Asynchronous Forward-Checking

Asynchronous Forward-Checking (AFC) incorporates the idea of the forward-checking (FC) algorithm for centralized CSP [13]. However, agents perform the forward checking phase asynchronously [20, 21]. As in synchronous backtracking, agents assign their variables only when they hold the current partial assignment (***cpa***). The ***cpa*** is a unique message (token) that is passed from one agent to the next one in the ordering. The ***cpa*** message carries the partial assignment (CPA) that agents attempt to extend into a complete solution by assigning their variables on it. When an agent succeeds in assigning its variable on the CPA, it sends this CPA to its successor. Furthermore, copies of the CPA are sent to all agents whose assignments are not yet on the CPA. These agents perform the forward checking asynchronously in order to detect as early as possible inconsistent partial assignments. The forward-checking process is performed as follows. When an agent receives a CPA, it updates the domain of its variable, removing all values that are in conflict with assignments on the received CPA. Furthermore, the shortest CPA producing the inconsistency is stored as justification of the value deletion.

When an agent generates an empty domain as a result of a forward-checking, it initiates a backtrack process by sending ***not_ok*** messages. ***not_ok*** messages carry the shortest 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 receives the ***not_ok*** message, it checks if the CPA carried in the received message is consistent with its AgentView. If it is the case, the receiver stores the ***not_ok***, otherwise, the ***not_ok*** is discarded. When an agent holding a ***not_ok*** receives a CPA on a ***cpa*** message from its predecessor, it sends this CPA back in a ***backcpa*** message. When multiple agents reject a given assignment by sending ***not_ok*** messages, only the first agent that will receive a ***cpa*** message from its predecessor and is holding a relevant ***not_ok*** message will eventually backtrack. After receiving a new ***cpa*** message, the ***not_ok*** message becomes obsolete when the CPA it carries is no longer a subset of the received CPA.

The manner in which the backtrack operation is performed is a major drawback of the AFC algorithm. The backtrack operation requires a lot of work from the agents. An improved backtrack method for AFC was described in Section 6 of [21]. 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 timestamp mechanism (see Definition 3).

3 Nogood-based Asynchronous Forward Checking

The nogood-based Asynchronous Forward-Checking (AFC-ng) is based on AFC. AFC-ng tries to enhance the asynchronism of the forward checking phase. The two main features of AFC-ng are the following. First, it uses the nogoods as justification of value deletions. 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) [14]. 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 [4].

Second, each time an agent A_i generates an empty domain it no longer sends ***not_ok*** messages. It resolves the nogoods ruling out values from its domain, producing a new nogood ng. ng is the conjunction of lhs of all nogoods stored by A_i. Then, A_i sends the resolved nogood ng in a ***ngd*** (backtrack) message to the lowest agent in ng. 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 reassignment of the destination agents then happen simultaneously and generate several CPAs. However, the strongest CPA coming from

the highest level in the agent ordering will eventually dominate all others. Agents use the timestamp (see Definition 3) to detect the strongest CPA. Interestingly, the search process of higher levels with stronger CPAs can use nogoods reported by the (killed) lower level processes, so that it benefits from their computational effort.

3.1 Description of the algorithm

AFC-ng agents execute the pseudo-code shown in Figs. 1 and 2. Each agent A_i stores a nogood per removed value in the NogoodStore. The other values not ruled out by a nogood form $D(x_i)$, the current domain of x_i. Agent A_i calls procedure `AFC-ng` in which it initializes its AgentView (line 1) by setting counters to zero (line 10). The AgentView contains a consistency flag that represents whether the partial assignment it holds is consistent. If A_i is the initializing agent IA (the first agent in the agent ordering), it initiates the search by calling procedure `Assign()` (line 3). Then, a loop considers the reception and the processing of the possible message types.

When calling `Assign()` A_i tries to find an assignment, which is consistent with its AgentView. If A_i fails to find a consistent assignment, it calls procedure `Backtrack()` (line 15). If A_i succeeds, it increments its counter t_i and generates a CPA from its AgentView augmented by its assignment (line 13). Afterwards, A_i calls procedure `SendCPA(CPA)` (line 14). If the CPA includes all agents assignments (A_i is the lowest agent in the order, line 16), A_i reports the CPA as a solution of the problem and marks the *end* flag true to stop the main loop (lines 17-18). Otherwise, A_i sends forward the CPA to every agent whose assignments are not yet on the CPA (line 19). So, the next agent on the ordering (successor) will try to extend this CPA by assigning its variable on it while other agents will perform the forward-checking phase asynchronously to check its consistency.

Whenever A_i receives a ***cpa*** message, procedure `ProcessCPA` is called (line 7). A_i checks its AgentView status. If it is not consistent and the AgentView is a subset of the received CPA, this means that A_i has already backtracked, then A_i does nothing (line 20). Otherwise, if the received CPA is stronger than its AgentView, A_i updates its AgentView and marks it consistent (lines 22-23). Procedure `UpdateAgentView` (lines 45-47) sets the AgentView and the NogoodStore to be consistent with the received CPA. Each nogood in the NogoodStore containing a value for a variable different from that on the received CPA will be deleted (line 47). Next, A_i calls procedure `Revise` (line 24) to store nogoods for values inconsistent with the new AgentView or to try to find a better nogood for values already having one in the NogoodStore (line 50). A nogood is better according to the *HPLV* heuristic if the lowest variable in the body (*lhs*) of the nogood is higher. If A_i generates an empty domain as a result of calling `Revise`, it calls procedure `Backtrack` (line 25), otherwise, A_i calls procedure `CheckAssign` to check if it has to assign its variable (line 26). In `CheckAssign(`*sender*`)`, A_i calls procedure `Assign` to try to assign its variable only if sender is the predecessor of A_i (i.e., CPA was received from the predecessor, line 27).

When every value of A_i's variable is ruled out by a nogood (line 25), the procedure `Backtrack` is called. These nogoods are resolved by computing a new nogood ng

procedure `AFC-ng()`

1. `InitAgentView();`
2. $end \leftarrow$ false; $AgentView.Consistent \leftarrow$ true;
3. **if** $(A_i = IA)$ **then** `Assign();`
4. **while** $(\neg end)$ **do**
5. $\quad msg \leftarrow$ `getMsg();`
6. $\quad$ **switch** $(msg.type)$ **do**
7. $\quad\quad$ ***cpa*** : `ProcessCPA(`msg`);`
8. $\quad\quad$ ***ngd*** : `ProcessNogood(`msg`);`
9. $\quad\quad$ ***terminate*** : `ProcessTerminate(`msg`);`

procedure `InitAgentView()`

10. **foreach** ($x_j \prec x_i$) **do** $AgentView[j] \leftarrow \{(x_j, empty, 0)\}$;

procedure `Assign()`

11. **if** $(D(x_i) \neq \emptyset)$ **then**
12. $\quad v_i \leftarrow$ `ChooseValue()` ; $t_i \leftarrow t_i$+1;
13. $\quad$ CPA $\leftarrow \{AgentView \cup (x_i, v_i, t_i)\}$;
14. $\quad$ `SendCPA(`CPA`);`
15. **else** `Backtrack();`

procedure `SendCPA(`CPA`)`

16. **if** (`size(`CPA`)` $= n$) **then** /* A_i is the last agent in the total ordering */
17. $\quad$ `broadcastMsg:` ***terminate***(CPA);
18. $\quad$ $end \leftarrow$ true ;
19. **else** **foreach** $(x_k \succ x_i)$ **do** `sendMsg:` **cpa**(CPA) **to** x_k;

Figure 1: Nogood-based AFC algorithm running by agent A_i (Part 1).

(line 28). ng is the conjunction of the left hand sides of all nogoods stored by A_i in its NogoodStore. If the new nogood ng is empty, A_i terminates execution after sending a ***terminate*** message to all agents in the system meaning that the problem is unsolvable (lines 29-31). Otherwise, A_i updates its AgentView by removing assignments of every agent that is placed after the agent A_j owner of `rhs(`ng`)` in the total order (lines 33-34). A_i also updates its NogoodStore by removing obsolete nogoods (line 37). Obsolete nogoods are nogoods inconsistent with the AgentView or containing the assignment of x_j (the right hand side of ng) (line 36). Finally, A_i marks its AgentView as inconsistent, removes its last assignment (line 38) and it backtracks by sending one ***ngd*** message to agent A_j (the right hand side of ng) (line 39). The ***ngd*** message carries the generated nogood (ng). A_i remains in an inconsistent state until receiving a stronger CPA holding

procedure `ProcessCPA(`*msg*`)`

20. **if** ($\neg AgentView.Consistent \wedge AgentView \subset msg.CPA$) **then return** ;
21. **if** ($msg.CPA$ is stronger than $AgentView$) **then**
22. `UpdateAgentView(`*msg.CPA*`)` ;
23. $AgentView.Consistent \leftarrow$ true ;
24. `Revise()` ;
25. **if** ($D(x_i) = \emptyset$) **then** `Backtrack()`;
26. **else** `CheckAssign(`*msg.Sender*`)`;

procedure `CheckAssign(`*sender*`)`

27. **if** (`predecessor(`A_i`)` $=$ *sender*) **then** `Assign()` ;

procedure `Backtrack()`

28. $ng \leftarrow$ `solve(`$NogoodStore$`)` ;
29. **if** ($ng = empty$) **then**
30. `broadcastMsg:` ***terminate***$(\emptyset)$;
31. $end \leftarrow$ true ;
32. **else**
33. **foreach** ($x_k \succ x_j$) **do** /* Let x_j denote the variable on `rhs(`ng`)` */
34. $AgentView[k].value \leftarrow empty$;
35. **foreach** ($nogood \in NogoodStore$) **do**
36. **if** ($\neg$`Consistent(`$nogood$`,`$AgentView$`)` $\vee\ x_j \in nogood$) **then**
37. `remove(`$nogood$`,`$NogoodStore$`)` ;
38. $AgentView.Consistent \leftarrow false$; $v_i \leftarrow empty$;
39. `sendMsg:` ***ngd***(ng) **to** A_j ;

procedure `ProcessNogood(`*msg*`)`

40. **if** (`Consistent(`$msg.nogood$`,`$AgentView$`)`) **then**
41. `add(`$msg.nogood$, $NogoodStore$`)` ; /* according to the HPLV [14] */
42. **if** (`rhs(`$msg.nogood$`).`$value = v_i$) **then** $v_i \leftarrow empty$; `Assign()` ;

procedure `ProcessTerminate(`*msg*`)`

43. $end \leftarrow$ true; $v_i \leftarrow empty$;
44. **if** ($msg.CPA \neq \emptyset$) **then** $solution \leftarrow msg.CPA$;

procedure `UpdateAgentView(`*CPA*`)`

45. $AgentView \leftarrow CPA$; /* update values and tags */
46. **foreach** ($ng \in NogoodStore$) **do**
47. **if** (¬`Consistent(`ng`,`$AgentView$`)`) **then** `remove(`ng`,`$NogoodStore$`)`;

procedure `Revise()`

48. **foreach** ($v \in D^0(x_i)$) **do**
49. **if** (v is ruled out by $AgentView$) **then**
50. store the best nogood for v; /* according to the HPLV [14] */

Figure 2: Nogood-based AFC algorithm running by agent A_i (Part 2).

at least one agent assignment with counter higher than that in the AgentView of A_i.

When a ***ngd*** message is received by an agent A_i, it checks the validity of the received nogood (line 40). If the received nogood is consistent with the AgentView, this nogood is a valid justification for removing the value on its rhs. Then if the value on the rhs of the received nogood is already removed, A_i adds the received nogood to its NogoodStore if it is better (according to the HPLV heuristic) than the current stored nogood. If the value on the rhs of the received nogood belongs to the current domain of x_i, A_i simply adds it to its NogoodStore. If the value on the rhs of the received nogood equals v_i, the current value of A_i, A_i dis-instantiates its variable and calls the procedure `Assign` (line 42).

`ProcessTerminate` procedure is called when an agent receives a ***terminate*** message. It marks *end* flag true to stop the main loop (line 43). If the attached CPA is empty then there is no solution. Otherwise, the solution of the problem is retrieved from the CPA (line 44).

4 Asynchronous Forward-Checking Tree

In this section, we show how to extend our AFC-ng algorithm to the *Asynchronous Forward-Checking Tree (AFC-tree)* algorithm using a pseudo-tree arrangement of the constraint graph. To achieve this goal, agents are ordered a priori in a pseudo-tree such that agents in different branches of the tree do not share any constraint. AFC-tree does not address the process of ordering the agents in a pseudo-tree arrangement. Therefore, the construction of the pseudo-tree is done in a preprocessing step. Now, it is known from centralized CSPs that the performance of the search procedures tightly depends on the variable ordering. Thus, the task of constructing the pseudo-tree is important for a search algorithm like AFC-tree.

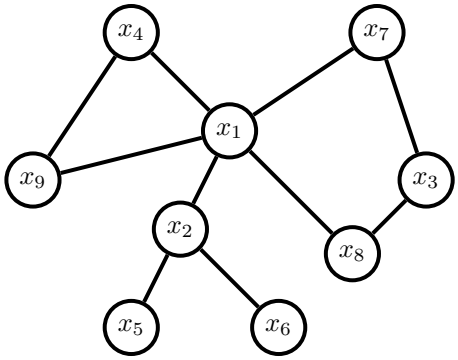
4.1 Pseudo-tree ordering

Any binary DisCSP can be represented by a *constraint graph* $G = (X_G, E_G)$, whose vertices represent the variables and edges represent the constraints. Therefore, $X_G = \mathcal{X}$ and for each constraint $c_{ij} \in \mathcal{C}$ connecting two variables x_i and x_j there exists an edge $\{x_i, x_j\} \in E_G$ linking vertices x_i and x_j.

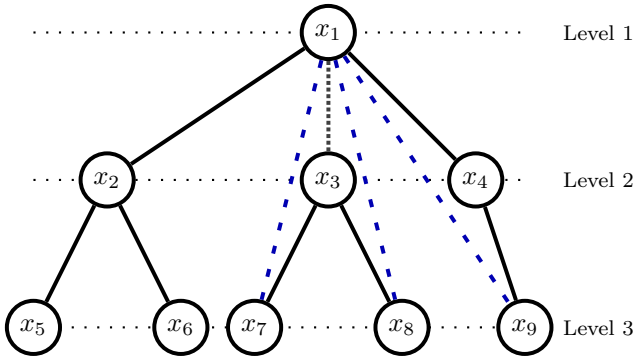
The concept of *pseudo-tree* arrangement of a constraint graph has been introduced first by Freuder and Quinn in [11]. The purpose of this arrangement is to perform search in parallel on independent branches of the pseudo-tree in order to improve search in centralized constraint satisfaction problems.

Definition 6 *A **pseudo-tree** arrangement $T = (X_T, E_T)$ of a graph $G = (X_G, E_G)$ is a rooted tree with the same set of vertices as G ($X_G = X_T$) such that vertices in different branches of T do not share any edge in G.*

Fig. 3(a) shows an example of a constraint graph G of a problem involving 9 variables $\mathcal{X} = X_G = \{x_1, \ldots, x_9\}$ and 10 constraints $\mathcal{C} = \{c_{12}, c_{14}, c_{17}, c_{18}, c_{19}, c_{25},$



(a) *a constraint graph G.*



(b) *a pseudo-tree arrangement T.*

Figure 3: Example of a pseudo-tree arrangement of a constraint graph.

c_{26}, c_{37}, c_{38}, $c_{49}\}$. An example of a pseudo-tree arrangement T of this constraint graph is illustrated in Fig. 3(b). Notice that G and T have the same vertices ($X_G = X_T$). However, a new (dotted) edge ($\{x_1, x_3\}$) linking x_1 to x_3 is added to T where $\{x_1, x_3\} \notin E_G$. Moreover, edges $\{x_1, x_7\}$, $\{x_1, x_8\}$ and $\{x_1, x_8\}$ belonging to the constraint graph G are not part of T. They are represented in T by dashed edges to show that constrained variables must be located in the same branch of T even if there is not an edge linking them.

From a pseudo-tree arrangement of the constraint graph we can define:

- A *branch* of the pseudo-tree is a path from the root to some leaf (e.g., $\{x_1, x_4, x_9\}$).
- A leaf is a vertex that has no child (e.g., x_9).
- The *children* of a vertex are its descendants connected to it through tree edges (e.g., $\texttt{children}(x_1) = \{x_2, x_3, x_4\}$).
- The *descendants* of a vertex x_i are vertices belonging to the subtree rooted at x_i

(e.g., `descendants`(x_2) $=\{x_5, x_6\}$ and `descendants`$(x_1)=\{\mathcal{X} \setminus x_1\}$).

- The *linked descendants* of a vertex are its descendants constrained with it together with its children, (e.g., `linkedDescendants`$(x_1) = \{x_2, x_3, x_4, x_7, x_8, x_9\}$).
- The *parent* of a vertex is the ancestor connected to it through a tree edge (e.g., `parent`$(x_9) = \{x_4\}$, `parent`$(x_3) = \{x_1\}$).
- A vertex x_i is an *ancestor* of a vertex x_j if x_i is the parent of x_j or an ancestor of the parent of x_j.
- The *ancestors* of a vertex x_i is the set of agents forming the path from the root to x_i's parent (e.g., `ancestors`$(x_8) = \{x_1, x_3\}$).

The construction of the pseudo-tree can be processed by a centralized procedure. First, a *system agent* must be elected to gather information about the constraint graph. Such system agent can be chosen using a leader election algorithm like that presented in [1]. Once, all information about the constraint graph is gathered by the system agent, it can perform a centralized algorithm to build the pseudo-tree ordering. A decentralized modification of the procedure for building the pseudo-tree was introduced by Chechetka and Sycara in [7]. This algorithm allows the distributed construction of pseudo-trees without needing to deliver any global information about the whole problem to a single process.

Whatever the method (centralized or distributed) for building the pseudo-tree, the obtained pseudo-tree may require the addition of some edges not belonging to the original constraint graph. In the example presented in Fig. 3(b), a new edge linking x_1 to x_3 is added to the resulting pseudo-tree T. The structure of the pseudo-tree will be used for communication between agents. Thus, the added link between x_1 and x_3 will be used to exchange messages between them. However, in some distributed applications, the communication might be restricted to the neighboring agents (i.e., a message can be passed only locally between agents that share a constraint). The solution in such applications is to use a *depth-first search tree (DFS-tree)*. DFS-trees are special cases of pseudo-trees where all edges belong to the original graph.

We present in Fig. 4 a simple distributed algorithm for the distributed construction of the DFS-tree named `DistributedDFS` algorithm. The `DistributedDFS` is similar to the algorithm proposed by Cheung in [8]. The `DistributedDFS` algorithm is a distribution of a DFS traversal of the constraint graph. Each agent maintains a set $Visited$ where it stores its neighbours which are already visited (line 2). The first step is to design the root agent using a leader election algorithm (line 1). An example of leader election algorithm was presented by Abu-Amara in [1]. Once the root is designed, it can start the distributed construction of the DFS-tree (procedure `CheckNeighbourhood` call, line 3). The designed root initiates the propagation of a ***token***, which is a unique message that will be circulated on the network until "visiting" all the agents of the problem.

When an agent x_i receives the ***token***, it marks all its neighbours included in the received message as visited (line 6). Next, x_i checks if the token is sent back by a child.

procedure `distributedDFS()`

1. Select the root via a leader election algorithm ;
2. $Visited \leftarrow \emptyset$; $end \leftarrow false$;
3. **if** (x_i *is the elected root*) **then** `CheckNeighbourhood()` ;
4. **while** ($\neg end$) **do**
5. $msg \leftarrow$ `getMsg()`;
6. $Visited \leftarrow Visited \cup$ (`neighbours`$(x_i) \cap msg.DFS$) ;
7. **if** ($msg.Sender \in$ `children`(x_i)) **then**
8. `descendants`$(x_i) \leftarrow$ `descendants`$(x_i) \cup msg.DFS$;
9. **else**
10. `parent`$(x_i) \leftarrow msg.Sender$;
11. `ancestors`$(x_i) \leftarrow msg.DFS$;
12. `CheckNeighbourhood()`;

procedure `CheckNeighbourhood()`

13. **if** (`neighbours`$(x_i) = Visited$) **then**
14. `sendMsg`: ***token***(`descendants`$(x_i) \cup \{x_i\}$) **to** `parent`(x_i) ;
15. $end \leftarrow true$;
16. **else**
17. select x_j in `neighbours`$(x_i) \backslash Visited$;
18. `children`$(x_i) \leftarrow$ `children`$(x_i) \cup x_j$;
19. `sendMsg`: ***token***(`ancestors`$(x_i) \cup \{x_i\}$) **to** x_j ;

Figure 4: The distributed depth-first search construction algorithm.

If it is the case, x_i sets all agents belonging to the subtree rooted at message sender (i.e., its child) as its descendants (lines 7-8). Otherwise, the ***token*** is received for the first time from the parent of x_i. Thus, x_i marks the sender as its parent (line 10) and all agents contained in the token (i.e., the sender and its ancestors) as its ancestors (line 11). Afterwards, x_i calls the procedure `CheckNeighbourhood` to check if it has to pass on the ***token*** to an unvisited neighbour or to return back the ***token*** to its parent if all its neighbours are already visited.

The procedure `CheckNeighbourhood` checks if all neighbours are already visited (line 13). If it is the case, the agent x_i sends back the ***token*** to its parent (line 14). The ***token*** contains the set DFS composed by x_i and its descendants. Until this point the agent x_i knows all its ancestors, its children and its descendants. Thus, the agent x_i terminates the execution of `DistributedDFS` (line 15). Otherwise, agent x_i chooses one of its neighbours (x_j) not yet visited and designs it as a child (lines 17-18). Afterwards, x_i passes on to x_j the ***token*** where it puts the ancestors of the child x_j (i.e., `ancestors`$(x_i) \cup \{x_i\}$) (line 19).

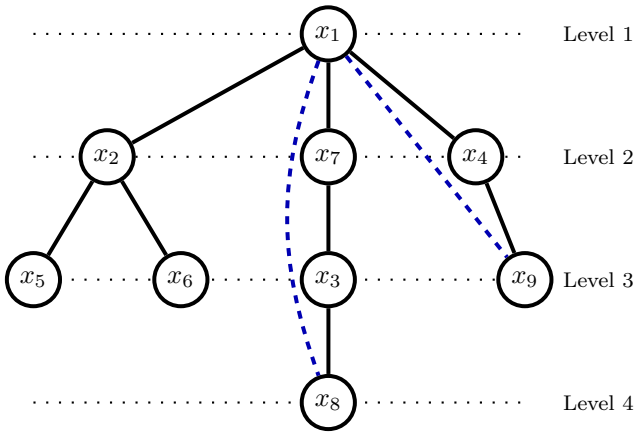


Figure 5: A DFS-tree arrangement of the constraint graph in Fig. 3(a).

Consider for example the constraint graph G presented in Fig. 3(a). Fig. 5 shows an example of a DFS-tree arrangement of the constraint graph G obtained by performing distributively the `DistributedDFS` algorithm. The `DistributedDFS` algorithm can be performed as follows. First, let x_1 be the elected root of the DFS-tree (i.e., the leader election algorithm elects the most connected agent). The root x_1 initiates the DFS-tree construction by calling procedure `CheckNeighbourhood` (line 3). Then, x_1 selects from its unvisited neighbours x_2 to be its child (lines 17-18). Next, x_1 passes on the ***token*** to x_2 where it put itself to be the ancestor of the receiver (x_2) (line 19). After receiving the ***token***, x_2 updates the set of its visited neighbours (line 6) by marking x_1 (the only neighbour included in the ***token***) visited. Afterwards, x_2 sets x_1 to be its parent and puts $\{x_1\}$ to be its set of ancestors (lines 10-11). Next, x_2 calls procedure `CheckNeighbourhood` (line 12). Until this point, x_2 has one visited neighbour (x_1) and two unvisited neighbours (x_5 and x_6). For instance, let x_2 chooses x_5 to be its child. Thus, x_2 sends the ***token*** to x_5 where it sets the DFS set to $\{x_1, x_2\}$. After receiving the ***token***, x_5 marks its single neighbour x_2 as visited (line 6), sets x_2 to be its parent (line 10), sets $\{x_1, x_2\}$ to be its ancestors ans sends the ***token*** back to x_2 where it puts itself. After receiving back the ***token*** from x_5, x_2 adds x_5 to its descendants and selects the last unvisited neighbour (x_6) to be its child and passes the ***token*** to x_6. In a similar way, x_6 returns back the ***token*** to x_2. Then, x_2 sends back the ***token*** to its parent x_1 since all its neighbours have been visited. The ***token*** contains the descendants of x_1 ($\{x_2, x_5, x_6\}$) on the subtree rooted at x_2. After receiving the ***token*** back from x_2, x_1 will select an agent from its unvisited neighbours $\{x_4, x_7, x_8, x_9\}$. Hence, the subtree rooted at x_2 where each agent knows its ancestors and its descendants is build without delivering any global information. The other subtrees respectively rooted at x_7 and x_4 are build in a similar manner. Thus, we obtain the DFS-tree shown in Fig. 5.

4.2 The AFC-tree algorithm

The AFC-tree 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, AFC-tree 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. Among all agents that receive the CPA, children perform 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, agents who are in disjoint subtrees may assign their variables simultaneously. AFC-tree thus exploits the potential speed-up of a parallel exploration in the processing of distributed problems. The degree of asynchronism is enhanced.

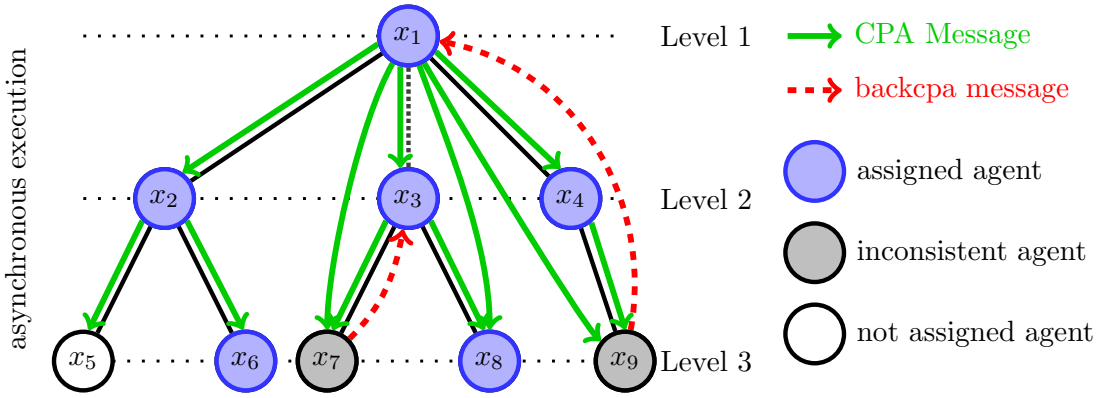


Figure 6: An example of the AFC-tree execution.

An execution of AFC-tree on a sample DisCSP problem is shown in Fig. 6. At time t_1, the root x_1 sends copies of the CPA on ***cpa*** messages to its linked descendants. Children x_2, x_3 and x_4 assign their values simultaneously in the received CPAs and then perform concurrently the AFC-tree algorithm. Agents x_7, x_8 and x_9 only perform a forward-checking. At time t_2, x_9 finds an empty domain and sends a ***ngd*** 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 ***ngd*** message. For the CPA that propagates on the path (x_1, x_2, x_6), x_6 successfully assigned its value and initiated a solution detection. The same thing is going to happen on the path (x_1, x_2, x_5) when x_5 (not yet instantiated) will receive the CPA from its parent x_2. When x_1 receives the ***ngd*** message from x_9, it initiates a new search process by sending a new copy of the CPA which will dominate all other CPAs where x_1 is assigned its old value. This new CPA generated by x_1 can then take advantage from efforts done by the obsolete CPAs. Consider for instance the subtree rooted at x_2. If the value of x_2 is consistent with the value of x_1 on the new

CPA, all nogoods stored on the subtree rooted at x_2 are still valid and a solution is reached on the subtree without any nogood generation.

In AFC-ng, a solution is reached when the last agent in the agent ordering receives the CPA and succeeds in assigning its variable. In AFC-tree, the situation is different because a CPA can reach a leaf agent 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 [6], but AFC-tree uses a different mechanism that allows to detect solutions before quiescence. AFC-tree uses an additional type of messages called ***accept*** that informs 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 [25].

The mechanism of solution detection is as follows: whenever a leaf node succeeds in assigning its value, it sends an ***accept*** 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 A_i receives ***accept*** messages from all its children that are all consistent with each other, all consistent with A_i's AgentView and with A_i's value, A_i builds an ***accept*** message being the conjunction of all received ***accept*** messages plus A_i's value-assignment. If A_i is the root a solution is found, and A_i broadcasts this solution to all agents. Otherwise, A_i sends the built ***accept*** message to its parent.

4.3 Description of the algorithm

We present in Fig. 7 only the procedures that are new or different from those of AFC-ng in Figs. 1 and 2. In `InitAgentView`, the AgentView of A_i is initialized to the set `ancestors(`A_i`)` and t_j is set to 0 for each agent (x_j) in `ancestors(`A_i`)` (line 11). The new data structure storing the received ***accept*** 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, A_i sends copies of the CPA only to its linked descendants (`linkedDescendants(`A_i`)`, lines 13-14). When the set `linkedDescendants(`A_i`)` is empty (i.e., A_i is a leaf), A_i calls the procedure `SolutionDetection` to build and send an ***accept*** message. In `CheckAssign(`*sender*`)`, A_i assigns its value if the CPA was received from its parent (line 16) (i.e., if *sender* is the parent of A_i).

In `ProcessAccept(`*msg*`)`, when A_i receives an ***accept*** message from its *child* for the first time, or the CPA contained in the received ***accept*** message is stronger than that received before, A_i stores the content of this message (lines 17-18) and calls the `SolutionDetection` procedure (line 19).

In `SolutionDetection`, if A_i is a leaf (i.e., `children(`A_i`)` is empty, line 20), it sends an ***accept*** message to its parent. The ***accept*** message sent by A_i contains its AgentView incremented by its assignment (lines 20-21). If A_i is not a leaf, it calls function `BuildAccept` to build an accept partial solution PA (line 23). If the returned partial solution PA is not empty and A_i is the root, PA is a solution of the problem. Then, A_i broadcasts it to other agents including the system agent and sets the *end* flag

procedure `AFC-tree()`

1. `InitAgentView();`
2. $end \leftarrow$ false; $AgentView.Consistent \leftarrow$ true;
3. **if** $(A_i = IA)$ **then** `Assign();`
4. **while** $(\neg end)$ **do**
5. $\quad msg \leftarrow$ `getMsg();`
6. $\quad$ **switch** $(msg.type)$ **do**
7. $\quad\quad$ ***cpa*** : `ProcessCPA(`msg`);`
8. $\quad\quad$ ***ngd*** : `ProcessNogood(`msg`);`
9. $\quad\quad$ ***terminate*** : `ProcessTerminate(`msg`);`
10. $\quad\quad$ ***accept*** : `ProcessAccept(`msg`);`

procedure `InitAgentView()`

11. **foreach** ($A_j \in$ `ancestors(`A_i`)`) **do** $AgentView[j] \leftarrow \{(x_j, empty, 0)\}$;
12. **foreach** $(child \in$ `children(`A_i`)`$)$ **do** $Accept[child] \leftarrow \emptyset$;

procedure `SendCPA(`CPA`)`

13. **if** (`children(`A_i`)` $\neq \emptyset$) **then**
14. $\quad$ **foreach** $(desc \in$ `linkedDescendants(`A_i`)`$)$ **do** `sendMsg:` ***cpa***(CPA) **to** $desc$
15. **else** `SolutionDetection()` ;

procedure `CheckAssign(`$sender$`)`

16. **if** (`parent(`A_i`)`$=sender$) **then** `Assign()` ;

procedure `ProcessAccept(`msg`)`

17. **if** ($msg.CPA$ is stronger than $Accept[msg.Sender]$) **then**
18. $\quad$ Accept$[msg.Sender] \leftarrow msg.CPA$;
19. $\quad$ `SolutionDetection()` ;

procedure `SolutionDetection()`

20. **if** (`children(`A_i`)` $= \emptyset$) **then**
21. $\quad$ `sendMsg:` ***accept***$(AgentView \cup \{(x_i, v_i, t_i)\})$ **to** `parent(`A_i`)` ;
22. **else**
23. $\quad PA \leftarrow$ `BuildAccept()` ;
24. $\quad$ **if** $(PA \neq \emptyset)$ **then**
25. $\quad\quad$ **if** $(A_i = root)$ **then** `broadcastMsg:` ***terminate***(PA); $end \leftarrow$true ;
26. $\quad\quad$ **else** `sendMsg:` ***accept***(PA) **to** `parent(`A_i`)` ;

function `BuildAccept()`

27. $PA \leftarrow AgentView \cup \{(x_i, v_i, t_i)\}$;
28. **foreach** $(child \in$ `children(`x_i`)`) **do**
29. $\quad$ **if** $(Accept[child] = \emptyset \vee \neg$ `Consistent(`$PA, Accept[child]$`)`$)$ **then** **return** $\emptyset$;
30. $\quad$ **else** $PA \leftarrow PA \cup$ Accept$[child]$;
31. **return** PA;

Figure 7: New lines/procedures of AFC-tree with respect to AFC-ng.

to true (line 25). Otherwise, A_i sends an ***accept*** message containing PA to its parent (line 26).

In `BuildAccept`, if an accept partial solution is reached. A_i generates a partial solution PA incrementing its AgentView with its assignment (line 27). Next, A_i loops over the set of ***accept*** messages received from its children. If at least one *child* has never sent an ***accept*** message or the ***accept*** message is inconsistent with PA, then the partial solution has not yet been reached and the function returns empty (line 29). Otherwise, the partial solution PA is incremented by the ***accept*** message of *child* (line 30). Finally, the accept partial solution is returned (line 31).

5 Correctness Proofs

Theorem 1 *The spatial complexity of AFC-ng (resp. AFC-tree) is polynomially bounded by $O(nd)$ per agent.*

Proof. 1 In AFC-ng, the size of nogoods is bounded by n, the total number of variables. In AFC-tree, the size of nogoods is bounded by h ($h \leq n$), the height of the pseudo-tree. Now, on each agent, AFC-ng (resp. AFC-tree) only stores one nogood per removed value. Thus, the space complexity of AFC-ng is in $O(nd)$ on each agent. AFC-tree also stores its set of descendants and ancestors, which is bounded by n on each agent. Therefore, AFC-tree has a space complexity in $O(hd + n)$. □

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

Proof. 2 We prove by induction on the agent ordering that there will be a finite number of new generated CPAs (at most d^n, where d is the size of the initial domain and n the number of variables.), and that agents can never fall into an infinite loop for a given CPA. The base case for induction ($i = 1$) is obvious. The only messages that x_1 can receive are ***ngd*** messages. All nogoods contained in these ***ngd*** messages have an empty lhs. Hence, values on their rhs are removed once and for all from the domain of x_1. Now, x_1 only generates a new CPA when it receives a nogood ruling out its current value. Thus, the maximal number of CPAs that x_1 can generate equals the size of its initial domain (d). Suppose now that the number of CPAs that agents $x_1, \ldots, x_{i-1}$ can generate is finite (and bounded by d^{i-1}). Given such a CPA on $[x_1, \ldots, x_{i-1}]$, x_i generates new CPAs (line 13, Fig. 1) only when it changes its assignment after receiving a nogood ruling out its current value v_i. Given the fact that any received nogood can include, in its lhs, only the assignments of higher priority agents ($[x_1, \ldots, x_{i-1}]$), this nogood will remain valid as long as the CPA on $[x_1, \ldots, x_{i-1}]$ does not change. Thus, x_i cannot regenerate a new CPA containing v_i without changing assignments on higher priority agents ($[x_1, \ldots, x_{i-1}]$). Since there are a finite number of values on the domain of variable x_i, there will be a finite number of new CPAs generated by x_i (d^i). Therefore, by induction we have that there will be a finite number of new CPAs (d^n) generated by AFC-ng.

Let *cpa* be the strongest CPA generated in the network and A_i be the agent that generated *cpa*. After a finite amount of time, all unassigned agents on *cpa* ($[x_{i+1}, \ldots, x_n]$) will receive *cpa* and thus will discard all other CPAs. Two cases occur. First case, at least one agent detects a dead-end and thus backtracks to an agent A_j included in *cpa* (i.e., $j \leq i$) forcing it to change its current value on *cpa* and to generate a new stronger CPA. Second case (no agent detects dead-end), if $i < n$, A_{i+1} generates a new stronger CPA by adding its assignment to *cpa*, else ($i = n$), a solution is reported. As a result, agents can never fall into an infinite loop for a given CPA and AFC-ng is thus guaranteed to terminate. □

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

Proof. 3 Whenever a stronger CPA or a ***ngd*** message is received, AFC-ng agents update their NogoodStore. 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 *AFC-ng is correct.*

Proof. 4 The argument for soundness is close to the one given in [21, 22]. The fact that agents only forward consistent partial solution on the CPA messages at only one place in procedure `Assign` (line 14, Fig. 1), implies that the agents receive only consistent assignments. A solution is reported by the last agent only in procedure `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. Completeness comes from the fact that AFC-ng is able to terminate and does not report inconsistency if a solution exists (Lemmas 1 and 2). □

Theorem 3 *AFC-tree algorithm is correct.*

Proof. 5 AFC-tree agents only forward consistent partial assignments (CPAs). Hence, leaf agents receive only consistent CPAs. Thus, leaf agents only send ***accept*** message holding consistent assignments to their parent. Since a parent builds an ***accept*** message only when the ***accept*** messages received from all its children are consistent with each other and all consistent with its own value, the ***accept*** message it sends contains a consistent partial solution. The root broadcasts a solution only when it can build itself such an ***accept*** message. Therefore, the solution is correct and AFC-tree is sound.

From Lemma 1 we deduce that the AFC-tree agent of highest 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 AFC-tree. AFC-tree performs multiple AFC-ng processes on the paths of the pseudo-tree from the root to the leaves. Thus, from Lemma 2, AFC-tree inherits the property that an empty nogood cannot be inferred if the network is satisfiable. As AFC-tree terminates, this ensures its completeness. □

6 Experimental Evaluation

In this section we experimentally compare AFC-ng and AFC-tree to two other algorithms: AFC and ABT. Algorithms are evaluated on three benchmarks: uniform binary random DisCSPs, distributed sensor-mobile networks and distributed meeting scheduling problems. All experiments were performed on the DisChoco 2.0 platform[1] [26], in which agents are simulated by Java threads that communicate only through message passing. All algorithms are tested on the same static agents ordering using the *dom/deg* heuristic [3] and the same nogood selection heuristic (*HPLV*) [14]. For ABT we implemented an improved version of Silaghi's solution detection [24] and counters for tagging assignments.

We evaluate the performance of the algorithms by communication load [17] and computation effort. Communication load is measured by the total number of exchanged messages among agents during algorithm execution (*#msg*), including those of termination detection (system messages). Computation effort is measured by the number of non-concurrent constraint checks (*#ncccs*) [32]. *#ncccs* is the metric used in distributed constraint solving to simulate the computation time.

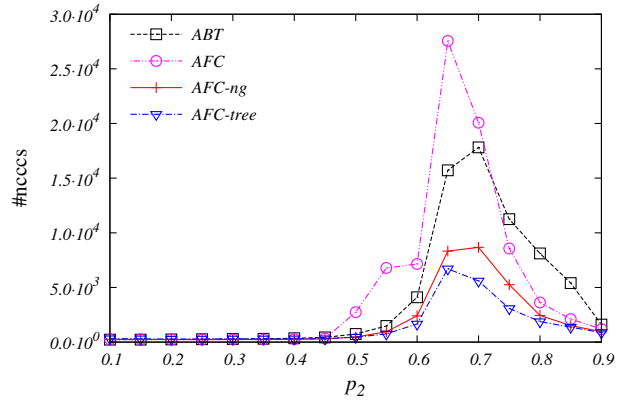
6.1 Uniform binary random DisCSPs

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 is the number of values in each of the domains, 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 instances of two classes of constraints graphs: sparse graphs $\langle 20, 10, 0.2, p_2\rangle$ and dense ones $\langle 20, 10, 0.7, p_2\rangle$. We vary the tightness from 0.1 to 0.9 by steps of 0.05. For each pair of fixed density and tightness (p_1, p_2) we generated 25 instances, solved 4 times each. We report average over the 100 runs.

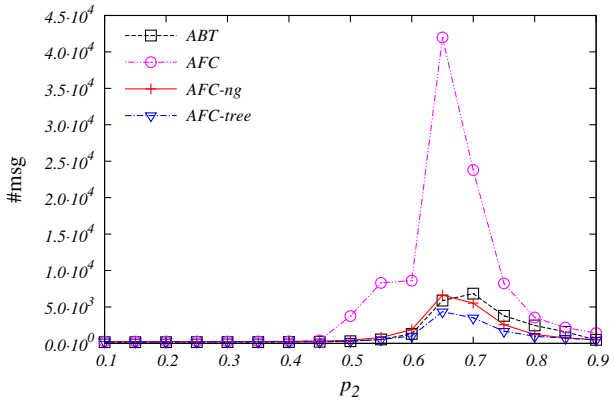
Fig. 8 presents performance of AFC-tree, AFC-ng, AFC, and ABT running on the sparse instances ($p_1 = 0.2$). In terms of computational effort (Fig. 8(a)), we observe that at the complexity peak, AFC is the less efficient algorithm. It is better than ABT (the second worst) only on instances to the right of the complexity peak (over-constrained). On the most difficult instances, AFC-ng improves AFC the performance of standard AFC by a factor of 3.5 and outperforms ABT by a factor of 2. AFC-tree takes advantage of the pseudo-tree arrangement to improve the speed-up of AFC-ng. Concerning communication load (Fig. 8(b)), AFC dramatically deteriorates compared to all other algorithms. AFC-ng improves AFC by a factor of 7. AFC-ng exchanges slightly fewer messages than ABT in the over-constrained area whereas AFC-tree has the smallest communication load on all problems.

Fig. 9 presents the results on the dense instances ($p_1 = 0.7$). When comparing the computational effort (Fig. 9(a)), the results obtained show that ABT dramatically deteriorates compared to synchronous algorithms. AFC-ng and AFC-tree show a small im-

[1] http://www.lirmm.fr/coconut/dischoco/



(a)



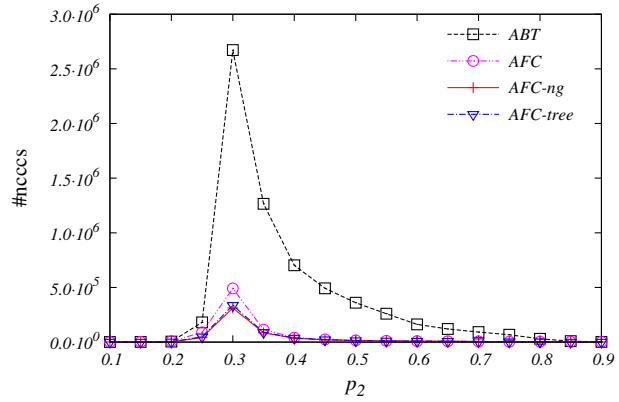
(b)

Figure 8: Total number of messages sent and $\#ncccs$ performed on sparse problems ($p_1 = 0.2$)

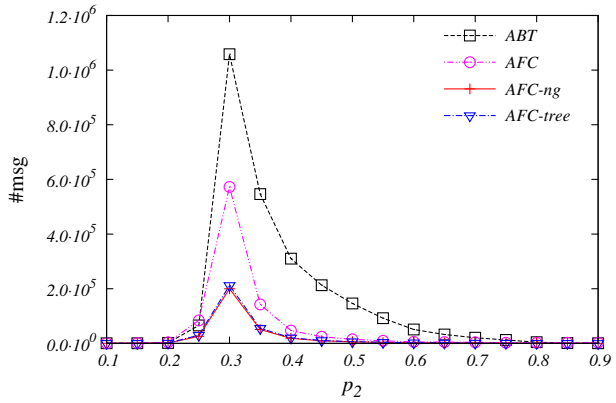
provement compared to AFC. Regarding the number of exchanged messages (Fig. 9(b)), ABT is again significantly the worst. AFC-ng and AFC-tree outperform AFC by a factor 3. On these dense graphs, AFC-tree behaves like AFC-ng because it does not benefit from the pseudo-tree arrangement, which is like a chain-tree in such graphs.

6.2 Distributed sensor-mobile problems

The *distributed sensor-mobile problem* (SensorDisCSP) [2] is a benchmark based on a real distributed problem. It consists of n sensors that track m mobiles. Each mobile must be tracked by 3 sensors. Each sensor can track at most one mobile. A solution must satisfy visibility and compatibility constraints. The visibility constraint defines the set of sensors to which a mobile is visible. The compatibility constraint defines the



(a)



(b)

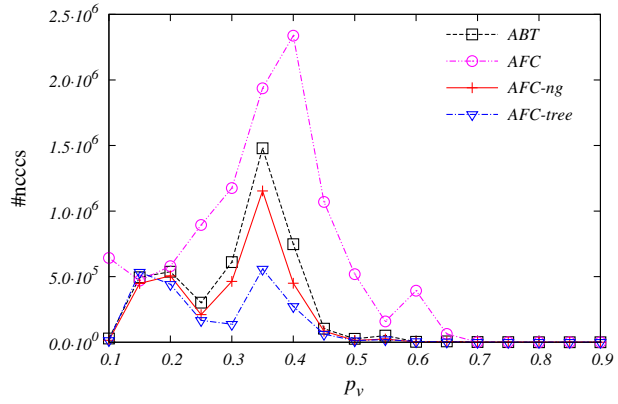
Figure 9: Total number of messages sent and $\#ncccs$ performed on dense problems ($p_1 = 0.7$)

compatibility among sensors.

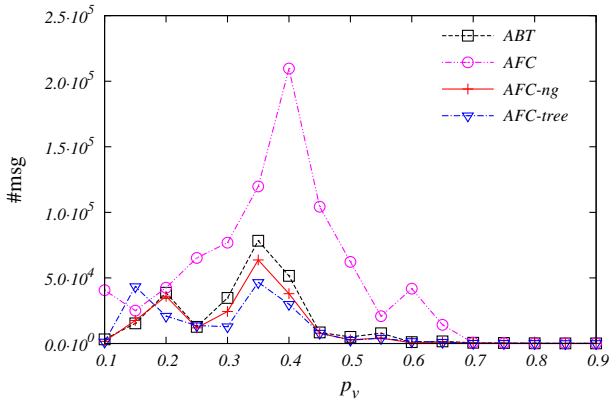
We encode SensorDisCSP in DisCSP as follows. Each agent represents one mobile. There are three variables per agent, one for each sensor that we need to allocate to the corresponding mobile. The domain of each variable is the set of sensors that can detect the corresponding mobile. The intra-agent constraints between the variables of one agent (mobile) specify that the three sensors assigned to the mobile must be distinct and pairwise compatible. The inter-agent constraints between the variables of different agents specify that a given sensor can be selected by at most one agent. In our implementation of the DisCSP algorithms, this encoding is translated to an equivalent formulation where we have three virtual agents for every real agent, each virtual agent handling a single variable.

Problems are characterized by $\langle n, m, p_c, p_v \rangle$, where n is the number of sensors, m is the number of mobiles, each sensor can communicate with a fraction p_c of the sensors that are in its sensing range, and each mobile can be tracked by a fraction p_v of the sensors having the mobile in their sensing range. We present results for the class $\langle 25, 5, 0.4, p_v \rangle$, where we vary p_v from 0.1 to 0.9 by steps of 0.05. Again, for each pair (p_c, p_v) we generated 25 instances, solved 4 times each, and averaged over the 100 runs.

Fig. 10 presents the performance of AFC-tree, AFC-ng, AFC, and ABT obtained on these SensorDisCSP with $\langle n = 25, m = 5, p_c = 0.4 \rangle$. When comparing the computational effort, (Fig. 10(a)), ABT outperforms AFC whereas AFC-ng outperforms both. AFC-tree outperforms all the compared algorithms. Concerning communication load (Fig. 10(b)), the ranking of algorithms is similar to that on computational effort, though differences tend to be smaller between ABT, AFC-ng and AFC-tree. AFC-tree



(a)



(b)

Figure 10: Total number of messages sent and *#ncccs* performed on instances where $p_c = 0.4$

remains the best on all problems except for a single point ($p_v = 1.5$), where it is slightly dominated by the other algorithms.

6.3 Distributed meeting scheduling problems

The *Distributed Meeting Scheduling Problem* (DMSP) is a truly distributed benchmark where agents may not desire to deliver their personal information to a centralized agent to solve the whole problem [27, 19]. The DMSP consists of a set of n agents having a personal private calendar and a set of m meetings each taking place in a specified location. Each agent knows the set of the k among m meetings he/she must attend. It is assumed that each agent knows the travelling time between the locations where his/her meetings will be held. The travelling time between two meetings m_i and m_j is denoted by $TravellingTime(m_i, m_j)$. Solving the problem consists in satisfying the following constraints: (i) all agents attending a meeting must agree on when it will occur, (ii) an agent cannot attend two meetings at same time, (iii) an agent must have enough time to travel from the location where he/she is to the location where the next meeting will be held.

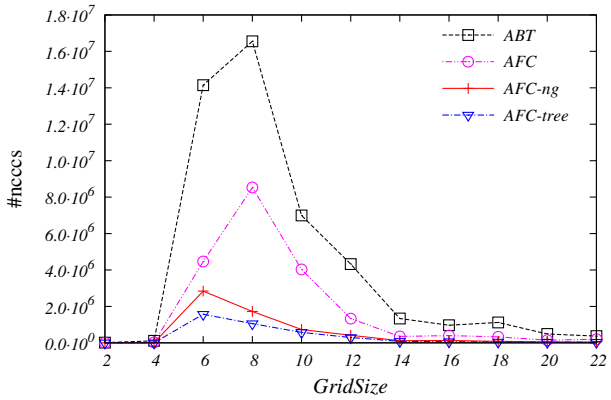
We encode the DMSP in DisCSP as follows. Each DisCSP agent represents a real agent and contains k variables representing the k meetings to which the agent participates. These k meetings are selected randomly among the m meetings. The domain of each variable contains the $d \times h$ slots where a meeting can be scheduled. A slot is one hour long, and there are h slots per day and d days. There is an equality constraint for each pair of variables corresponding to the same meeting in different agents. This equality constraint means that all agents attending a meeting must schedule it at the same slot (constraint (i)). There is an *arrival-time* constraint between all variables/meetings belonging to the same agent. The arrival-time constraint between two variables m_i and m_j is $|m_i - m_j| - duration > TravellingTime(m_i, m_j)$, where *duration* is the duration of every meeting. This arrival-time constraint allows us to express both constraints (ii) and (iii). We place meetings randomly on the nodes of a uniform grid of size $g \times g$ and the travelling time between two adjacent nodes is 1 hour. Thus, the travelling time between two meetings equals the Euclidean distance between nodes representing the locations where they will be held. For varying the tightness of the arrival-time constraint we vary the size of the grid on which meetings are placed. Problems are characterized by $\langle n, m, k, d, h, g \rangle$, where n is the number of agents, m is the number meetings, k is the number of meetings/variables per agent, d is the number of days and h is the number of hours per day, and g is the grid size. The duration of each meeting is one hour. In our implementation of the DisCSP algorithms, this encoding is translated to an equivalent formulation where we have k (number of meetings per agent) virtual agents for every real agent, each virtual agent handling a single variable. We present results for the class $\langle 20, 9, 3, 2, 10, g \rangle$ where we vary g from 2 to 22 by steps of 2. Again, for each g we generated 25 instances, solved 4 times each, and averaged over the 100 runs.

On this class of meeting scheduling benchmarks AFC-tree and AFC-ng continue to perform well. They are close to each other, with a slight gain for AFC-tree. They are both significantly better that ABT and AFC, both for computational effort (Fig. 11(a))

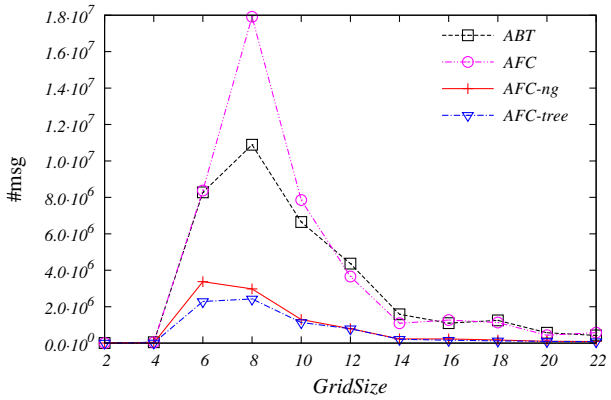
and communication load (Fig. 11(b)). Concerning the computational effort, ABT is the slowest algorithm to solve such problems. AFC outperforms ABT by a factor of 2 at the peak. However, ABT requires less messages than AFC.

6.4 Discussion

A first observation on these experiments is that AFC-ng is always better than AFC, both in terms of exchanged messages and computational effort (*#ncccs*). A closer look at the type of exchanged messages shows that the backtrack operation in AFC requires exchanging a lot of ***not_ok*** messages (approximately 50% of the total number of messages sent by agents). This confirms the significance of using nogoods as justification of value removals and allowing several concurrent backtracks in AFC-ng. A second observation



(a) $\langle n = 20,\ m = 9,\ k = 3, d = 2, h = 10\rangle$



(b) $\langle n = 20,\ m = 9,\ k = 3, d = 2, h = 10\rangle$

Figure 11: Total number of messages sent and *#ncccs* performed on meeting scheduling benchmarks where the number of meeting per agent is 3 ($k = 3$).

on these experiments is that AFC-tree is almost always better than or equivalent to AFC-ng both in terms of communication load and computational effort. When the graph is sparse, AFC-tree benefits from running separate search processes in disjoint problem subtrees. When agents are highly connected (dense graphs), AFC-tree runs on a chain-tree pseudo-tree and thus mimics AFC-ng. A final observation on these experiments is that ABT performs bad in dense graphs compared to synchronous algorithms.

7 Other Related Work

In [5, 31] 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) [12], 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 [12] was tested and found to be ineffective for more than two concurrent search processes [12]. Dynamic Distributed BackJumping (DDBJ) was presented in [22]. 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 ***ngd*** message continue search concurrently. Once a stronger 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. Besides its use of nogoods as justification of value removal, AFC-ng allows simultaneous backtracks going from different agents to different destinations. Thus, it enhances the asynchronism of the forward-checking phase. The second algorithm, Asynchronous Forward-Checking Tree (AFC-tree), is based on AFC-ng and is performed on a pseudo-tree arrangement of the constraint graph. AFC-tree runs simultaneous AFC-ng processes on each branch of the pseudo-tree to exploit the parallelism inherent in the problem. Our experiments show that AFC-ng improves the AFC algorithm in terms of computational effort and number of exchanged messages. Experiments also show that AFC-tree is the most robust algorithm. It is particularly good when the problems are sparse because it takes advantage of the pseudo-tree ordering.

References

[1] Hosame H. Abu-Amara. Fault-tolerant distributed algorithm for election in complete networks. *IEEE Transactions on Computers*, 37:449–453, April 1988.

[2] Ramón Béjar, Carmel Domshlak, Cèsar Fernández, Carla Gomes, Bhaskar Krishnamachari, Bart Selman, and Magda Valls. Sensor networks and distributed csp: communication, computation and complexity. *Artificial Intelligence*, 161:117–147, 2005.

[3] Christian Bessiere and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *CP*, volume 1118 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1996.

[4] Christian Bessiere, Arnold Maestre, Ismel Brito, and Pedro Meseguer. Asynchronous backtracking without adding links: a new member in the ABT family. *Artificial Intelligence*, 161:7–24, 2005.

[5] Ismel Brito and Pedro Meseguer. Synchronous, asynchronous and hybrid algorithms for DisCSP. In *Proceeding of the fifth Workshop on Distributed Constraint Reasoning at the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, DCR'04, pages 80–94. Toronto, Canada, September 2004.

[6] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[7] Anton Chechetka and Katia Sycara. A decentralized variable ordering method for distributed constraint optimization. Technical Report CMU-RI-TR-05-18, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2005.

[8] To-Yat Cheung. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE transaction on software engineering*, 9(4):504–512, 1983.

[9] Yek Loong Chong and Youssef Hamadi. Distributed log-based reconciliation. In *Proceeding of the 17th European Conference on Artificial Intelligence ECAI'06*, pages 108–112, Riva del Garda, Italy, August 2006.

[10] Rina Dechter. Constraint networks (survey). *In S. C. Shapiro (Eds.), Encyclopedia Artificial Intelligence*, pages 276–285, 1992.

[11] Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *IJCAI 1985*, pages 1076–1078, 1985.

[12] Youssef Hamadi. Interleaved backtracking in distributed constraint networks. *International Journal of Artificial Intelligence Tools*, 11:167–188, 2002.

[13] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[14] Katsutoshi Hirayama and Makoto Yokoo. The effect of nogood learning in distributed constraint satisfaction. In *Proceedings of the The 20th International Conference on Distributed Computing Systems*, ICDCS'00, pages 169–177, Washington, DC, USA, 2000. IEEE Computer Society.

[15] Hyuckchul Jung, Milind Tambe, and Shriniwas Kulkarni. Argumentation as distributed constraint satisfaction: applications and results. In *Proceedings of the fifth international conference on Autonomous agents*, AGENTS'01, pages 324–331, New York, NY, USA, 2001.

[16] Thomas Léauté and Boi Faltings. Coordinating logistics operations with privacy guarantees. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, IJCAI'11, pages 2482–2487, Barcelona, Spain, July 16–22 2011.

[17] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series, 1997.

[18] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce, and Pradeep Varakantham. Taking dcop to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS'04, 2004.

[19] Amnon Meisels and Oz Lavee. Using additional information in DisCSP search. In *Proceeding of 5th workshop on distributed constraints reasoning*, DCR'04, Toronto, 2004.

[20] Amnon Meisels and Roie Zivan. Asynchronous forward-checking for distributed CSPs. In W. Zhang, editor, *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.

[21] Amnon Meisels and Roie Zivan. Asynchronous forward-checking for DisCSPs. *Constraints*, 12(1):131–150, 2007.

[22] Viet Nguyen, Djamila Sam-Haroud, and Boi Faltings. Dynamic distributed backjumping. In *Proceeding of 5th workshop on DCR'04*, Toronto, 2004.

[23] Adrian Petcu and Boi Faltings. A value ordering heuristic for distributed resource allocation. In *Proceeding of CSCLP04*, Lausanne, Switzerland, 2004.

[24] Marius-Calin Silaghi. Generalized dynamic ordering for asynchronous backtracking on DisCSPs. In *DCR workshop, AAMAS-06*, Hakodate, Japan, 2006.

[25] Marius-Calin Silaghi and Boi Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence*, 161:25–53, 2005.

[26] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, and El Houssine Bouyakhf. DisChoco 2: A platform for distributed constraint reasoning. In *Proceedings of the IJCAI'11 workshop on Distributed Constraint Reasoning*, pages 112–121, Barcelona, Catalonia, Spain, 2011. URL http://www.lirmm.fr/coconut/dischoco/.

[27] Richard J. Wallace and Eugene C. Freuder. 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, pages 176–182, Bologna, 2002.

[28] Makoto Yokoo. Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

[29] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the 12th IEEE Int'l Conf. Distributed Computing Systems*, pages 614–621, 1992.

[30] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, September 1998.

[31] Roie Zivan and Amnon Meisels. Synchronous vs asynchronous search on DisCSPs. In *Proceeding of 1st European Workshop on Multi Agent System, EUMAS*, Oxford, 2003.

[32] Roie Zivan and Amnon Meisels. Message delay and DisCSP search algorithms. *Annals of Mathematics and Artificial Intelligence*, 46(4):415–439, 2006.