Maintaining Arc Consistency Asynchronously in Synchronous Distributed Search

Mohamed Wahbi[1,2] wahbi@lirmm.fr
Redouane Ezzahir[3] red.ezzahir@gmail.com
Christian Bessiere[1] bessiere@lirmm.fr
El Houssine Bouyakhf[2] bouyakhf@fsr.ac.ma

[1]LIRMM
University of Montpellier
Montpellier, France

[2]LIMIARF
University Mohammed V–Agdal
Rabat, Morocco

[3]ENSA Agadir
University Ibn Zohr
Agadir, Morroco

***Abstract*—We recently proposed Nogood-Based Asynchronous Forward Checking (AFC-ng), an efficient and robust algorithm for solving Distributed Constraint Satisfaction Problems (DisCSPs). AFC-ng performs an asynchronous forward checking phase during synchronous search. In this paper, we propose two new algorithms based on the same mechanism as AFC-ng. However, instead of using forward checking as a filtering property, we propose to maintain arc consistency asynchronously (MACA). The first algorithm we propose, MACA-del, enforces arc consistency thanks to an additional type of messages, deletion messages. The second algorithm, MACA-not, achieves arc consistency without any new type of message. We provide a theoretical analysis and an experimental evaluation of the proposed approach. Our experiments show the good performance of MACA algorithms, particularly those of MACA-not.**

***Index Terms*—constraint reasoning, distributed constraint solving, maintaining arc consistency**

I. Introduction

Constraint satisfaction problems (CSPs) can formalize many real world combinatorial problems such as resource allocation, car sequencing, machine vision, etc. A constraint satisfaction problem consists in looking for solutions to a constraint network, that is, finding a set of assignments of values to variables that satisfy the constraints of the problem. These constraints specify admissible value combinations.

Many backtrack search algorithms were developed for solving constraint satisfaction problems. Typical backtrack search algorithms try to build a solution to a CSP by interleaving variable instantiation with constraint propagation. Forward Checking (FC) [10] and Maintaining Arc Consistency (MAC) [18] are examples of such algorithms. In the 80's, FC was considered as the most efficient search algorithm. In the middle 90's, several works have empirically shown that MAC is more efficient than FC on hard and large problems [4, 9].

Sensor networks [12, 1], distributed resource allocation problems [16, 17] or distributed meeting scheduling [24, 14] are real applications of a distributed nature, that is, knowledge is distributed among several entities. These applications can be naturally modeled and solved by a CSP process once the knowledge about the problem is delivered to a centralized solver. However, in such applications, gathering the whole knowledge into a centralized solver is undesirable. In general, this restriction is mainly due to privacy and/or security reasons. The cost or the inability of translating all information to a single format may be another reason. Thus, a distributed model allowing a decentralized solving process is more adequate. The *distributed constraint satisfaction problem* (DisCSP) has such properties.

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. Therefore, agents attempt to generate a locally consistent assignment (assignment of values to their variables), that is also consistent with constraints between agents [26, 25].

Many distributed algorithms for solving DisCSPs have been designed in the last two decades. 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 backtracks [27]. Meisels and Zivan (2007) extended SBT to Asynchronous Forward Checking (AFC), an algorithm in which the FC algorithm [10] is performed asynchronously [15]. In AFC, whenever an agent succeeds to extend the current partial assignment, it sends the CPA to its successor and it sends copies of this CPA to the other unassigned agents in order to perform FC asynchronously.

In a recent work [8], we proposed the Nogood Based Asynchronous Forward Checking algorithm (AFC-ng), which is an improvement of AFC. Unlike AFC, AFC-ng uses nogoods as justification of value removals and allows several simultaneous backtracks coming from different agents and going to different destinations. AFC-ng was shown to outperform AFC.

Although, many studies incorporated FC successfully in distributed CSPs [5, 15, 8], MAC has not yet been well investigated. The only tentatives to include arc consistency maintenance in distributed algorithms were done on the Asynchronous Backtracking (ABT) algorithm. Silaghi et al. (2001) introduced the Distributed Maintaining Asynchronously Consistency for ABT, (*DMAC-ABT*), the first algorithm able to maintain arc consistency in distributed CSPs [20]. DMAC-ABT considers consistency maintenance as a hierarchical nogood-based inference. Brito and Meseguer (2008) proposed ABT-uac and ABT-dac, two algorithms that connect ABT with arc consistency [6]. ABT-uac propagates unconditionally deleted values to enforce an amount of full arc consistency. ABT-dac propagates conditionally and unconditionally deleted values using directional arc consistency. ABT-uac shows minor improvement in communication load and ABT-dac is harmful in many instances.

In this work, we propose two new synchronous search algorithms based on the same mechanism as AFC-ng. However, instead of maintaining forward checking asynchronously on agents not yet instantiated, we propose to maintain arc consistency asynchronously on these future agents. We call this new scheme MACA, for *maintaining arc consistency asynchronously*. As in AFC-ng, only the agent holding the current partial assignment (CPA) can perform an assignment. However, unlike AFC-ng, MACA attempts to maintain the arc consistency instead of performing only FC. The first algorithm we propose, MACA-del, enforces arc consistency thanks to an additional type of messages, deletion messages (***del***). Hence, whenever values are removed during a constraint propagation step, MACA-del agents notify other agents that may be affected by these removals, sending them a ***del*** message. ***del*** messages contain all removed values and the nogood justifying their removal.The second algorithm, MACA-not, achieves arc consistency without any new type of message. We achieve this by storing all deletions performed by an agent on domains of its neighboring agents, and sending this information to these neighbors within the CPA message.

This paper is organized as follows. Section II gives the necessary background. Sections III describes the MACA-del and MACA-not algorithms. Theoretical analysis and correctness proofs are given in Section IV. Section V presents an experimental evaluation of our proposed algorithms against other algorithms. Finally, we will conclude the paper in Section VI.

II. Background

A. Basic Definitions and Notations

The *distributed constraint satisfaction problem* (DisCSP) has been formalized in [26] 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. $\mathcal{C}$ 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, i.e., $c_{ij} \subseteq D^0(x_i) \times D^0(x_j)$.

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 priority agents (*predecessors*) and conversely the lower priority agents (*successors*) are those appearing after A_i. For sake of clarity, we assume that the total order on agents $\prec$ 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 *compatible* 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 $\{(x_j, empty, 0) \mid x_j \prec x_i\}$.

Based on the constraints of the problem, 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 $x_i = v_i \wedge \ldots \wedge x_j = v_j \rightarrow x_k \neq v_k$, meaning that the assignment $x_k = v_k$ is inconsistent with the assignments $x_i = v_i, \ldots, x_j = v_j$. 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 (i.e., $D(x_i) = \emptyset$), 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 $x_j = v_j$. $lhs\,(ng)$ is the conjunction of the left-hand sides of all nogoods except $x_j = v_j$ and $rhs\,(ng)$ is $x_j \neq v_j$.

B. Maintaining Arc Consistency

Constraint propagation is a central feature of efficiency for solving CSPs [2]. The oldest and most commonly used technique for propagating constraints is arc consistency (AC).

Definition 6: A constraint c_{ij} is ***arc consistent*** iff $\forall\ v_i \in D(x_i), \exists\ v_j \in D(x_j)$ such that (v_i, v_j) is allowed by c_{ij} and $\forall\ v_j \in D(x_j), \exists\ v_i \in D(x_i)$ such that (v_i, v_j) is allowed by c_{ij}. A constraint network is arc consistent iff all its constraints are arc consistent.

The Maintaining Arc Consistency (MAC) algorithm [18] alternates exploration steps and constraint propagation steps. That is, at each step of the search, a variable assignment is followed by a filtering process that corresponds to enforcing arc consistency.

For implementing MAC in a distributed CSP, each agent A_i is assumed to know all constraints in which it is involved and the agents with whom it shares a constraint (aka its neighboring agents). These agents and the constraints linking them to A_i form the local constraint network of A_i, denoted by $CSP(i)$.

Definition 7: The ***Local Constraint Network*** $CSP(i)$ of an agent $A_i \in \mathcal{A}$ consists of all constraints involving x_i and all variables of these constraints.

In order to allow agents to maintain arc consistency in distributed CSPs, our proposed approach consists in enforcing arc consistency on the local constraint network of each agent. Basically, each agent A_i stores locally copies of all variables in $CSP(i)$. We also assume that each agent knows the neighborhood it has in common with its own neighbors, without knowing the constraints relating them. That is, for each of its neighbors A_k, an agent A_i knows the list of agents A_j such that there is a constraint between x_i and x_j and a constraint between x_k and x_j.

III. Maintaining Arc Consistency Asynchronously

In the nogood-based Asynchronous Forward Checking (AFC-ng), the forward checking phase aims at anticipating the backtrack. Nevertheless, we do not take advantage of the value removals caused by FC if it does not completely wipe out the domain of the variable. One can investigate these removals by enforcing arc consistency. This is motivated by the fact that the propagation of a value removal, for an agent A_i, may generate an empty domain for a variable in its local constraint network. Thus, we can detect an earlier dead-end and then anticipate as soon as possible the backtrack operation.

In synchronous search algorithms for solving DisCSPs, agents sequentially assign their variables. Thus, agents perform the assignment of their variable only when they hold the current partial assignment, CPA. We propose an algorithm in which agents assign their variable one by one following a total ordering on agents. Hence, whenever an agent succeeds in extending the CPA by assigning its variable on it, it sends the CPA to its successor to extend it. Copies of this CPA are also sent to the other agents whose assignments are not yet on the CPA in order to *maintain arc consistency asynchronously*. Therefore, when an agent receives a copy of the CPA, it maintains arc consistency in its local constraint network. To enforce arc consistency on all variables of the problem, agents communicate information about value removals produced locally with other agents. We propose two methods to achieve this. The first method, namely MACA-del, uses a new type of messages (***del*** messages) to share this information. The second method, namely MACA-not, includes the information about deletions generated locally within ***cpa*** messages.

A. *Enforcing AC using **del** messages*

In MACA-del, each agent A_i maintains arc consistency on its local constraint network, $CSP(i)$, whenever a domain of a variable in $CSP(i)$ is changed. Changes can occur either on the domain of A_i or on another domain in $CSP(i)$. In MACA-del on A_i, only removals on $D(x_i)$ are externally shared with other agents. The propagation of the removals on $D(x_i)$ is achieved by communicating to other agents the nogoods justifying these removals. These removals and their associated nogoods are sent to neighbors via ***del*** messages.

Agent A_i stores nogoods for its removed values. They are stored in $NogoodStore[x_i]$. But in addition to nogoods stored for its own values, A_i needs to store nogoods for values removed from variables x_j in $CSP(i)$. Nogoods justifying the removals of values from $D(x_j)$ are stored in $NogoodStore[x_j]$. Hence, the NogoodStore of an agent A_i is a vector of several NogoodStores, one for each variable in $CSP(i)$.

The pseudo code of MACA-del, executed by each agent A_i, is shown in Fig. 1. Agent A_i starts the search by calling procedure `MACA-del()`. In procedure `MACA-del()`, A_i calls `Propagate()` to enforce arc consistency (line 1) in its local constraint network, i.e., $CSP(i)$. Next, if A_i is the initializing agent IA (the first agent in the agent ordering), it initiates the search by calling procedure `Assign()` (line 2). Then, a loop considers the reception and the processing of the possible message types.

When calling procedure `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 14). If A_i succeeds, it increments its counter t_i and generates a CPA from its AgentView augmented by its assignment (lines 11 and 12). Afterwards, A_i calls procedure `SendCPA(CPA)` (line 13). If the CPA includes all agents assignments (A_i is the lowest agent in the order, line 15), A_i reports the CPA as a solution of the problem and marks the *end* flag true to stop the main loop (line 16). Otherwise, A_i sends forward the CPA to all agents whose assignments are not yet on the CPA (line 17). So, the next agent on the ordering (successor) will try to extend this CPA by assigning its variable on it while other agents will maintain arc consistency asynchronously.

Whenever A_i receives a ***cpa*** message, procedure `ProcessCPA()` is called (line 6). The received message will be processed only when it holds a CPA stronger than the AgentView of A_i. If it is the case, A_i updates its AgentView (line 19) and then updates the NogoodStore of each variable in $CSP(i)$ to be compatible with the received CPA (line 20). Afterwards, A_i calls function `Propagate()` to enforce arc consistency on $CSP(i)$ (line 21). If arc consistency wiped out a domain in $CSP(i)$ (i.e., $CSP(i)$ is not arc consistent), A_i calls procedure `Backtrack()` (line 21). Otherwise, A_i checks if it has to assign its variable (line 22). A_i tries to assign its variable by calling procedure `Assign()` only if it received the ***cpa*** from its predecessor.

```
procedure MACA-del()
1.  $end \leftarrow$ false;   Propagate() ;
2.  if ( $A_i = IA$ ) then  Assign() ;
3.  while ( $\neg end$ ) do
4.      $msg \leftarrow$ getMsg();
5.      switch ( $msg.type$ ) do
6.          cpa  : ProcessCPA($msg$);
7.          ngd  : ProcessNogood($msg$);
8.          del  : ProcessDel($msg$);
9.          stp  : $end \leftarrow true$;
procedure Assign()
10. if ( $D(x_i) \neq \emptyset$ ) then
11.     $v_i \leftarrow$ ChooseValue() ; $t_i \leftarrow t_i$+1 ;
12.     CPA $\leftarrow \{AgentView \cup (x_i, v_i, t_i)\}$ ;
13.     SendCPA(CPA) ;
14. else  Backtrack() ;
procedure SendCPA(CPA)
15. if ( size(CPA) $= n$ ) then
16.     broadcastMsg:stp $\langle$CPA$\rangle$ ;   $end \leftarrow$ true ;
17. else  foreach ( $x_k \succ x_i$ ) do   sendMsg:cpa $\langle$ CPA $\rangle$ to $A_k$ ;
procedure ProcessCPA($msg$)
18. if ( $msg.CPA$ is stronger than the $AgentView$ ) then
19.     $AgentView \leftarrow msg.CPA$ ;
20.     Remove all nogoods incompatible with $AgentView$ ;
21.     if ( $\neg$Propagate() ) then  Backtrack() ;
22.     else if ( $msg.sender = A_{i-1}$ ) then  Assign() ;
function Propagate()
23. if ( $\neg$AC($CSP(i)$) ) then  return false ;
24. else if ( $D(x_i)$ was changed ) then
25.     foreach ( $x_j \in CSP(i)$ ) do
26.         $nogoods \leftarrow$ get nogoods from $NogoodStore[x_i]$ that
            are relevant to $x_j$ ;
27.         sendMsg:del $\langle nogoods \rangle$ to $A_j$ ;
28. return true ;
procedure ProcessDel($msg$)
29. Let $A_k$ be the agent that sent $msg$;
30. foreach ( $ng \in msg.nogoods$ ) do
31.     if ( Compatible($ng$, $AgentView$) ) then
32.         add($ng$, $NogoodStore[x_k]$) ;
33. if ( $D(x_k) = \emptyset \wedge x_i \in NogoodStore[x_k]$ ) then
34.     $newNogood \leftarrow$ solve($NogoodStore[x_k]$) ;
35.     add($newNogood$, $NogoodStore[x_i]$) ;
36.     Assign() ;
37. else if ( $D(x_k) = \emptyset \vee \neg$Propagate() ) then
38.     Backtrack();
procedure Backtrack()
39. Let $x_k$ be the variable within the empty domain;
40. $ng \leftarrow$ solve($NogoodStore[x_k]$) ;            /* $D(x_k) = \emptyset$ */
41. if ( $ng = empty$ ) then
42.     broadcastMsg:stp $\langle \emptyset \rangle$ ;
43.     $end \leftarrow$ true ;
44. else                    /* Let $x_j$ be the variable on the $rhs(ng)$ */
45.     sendMsg:ngd $\langle ng \rangle$  to $A_j$ ;
46.     from ( $l = j$ to $i$-1 ) do
47.         $AgentView[l].value \leftarrow empty$ ;
48.     Remove all nogoods incompatible with $AgentView$ ;
procedure ProcessNogood($msg$)
49. if ( Compatible($lhs$($msg.nogood$ ), $AgentView$) ) then
50.     add($msg.nogood$, $NogoodStore[x_i]$) ;
51.     if ( $rhs(msg.nogood).value = v_i$ ) then  Assign();
52.     else if ( $\neg$Propagate() ) then  Backtrack() ;
```

Fig. 1. MACA-del algorithm running by agent A_i.

When calling function Propagate(), A_i restores arc consistency on its local constraint network according to the assignments on its AgentView (line 23). In our implementation we used *AC-2001* [3] to enforce arc consistency but any generic AC algorithm can be used. MACA-del requires from the algorithm enforcing arc consistency to store a nogood for each removed value. When two nogoods are possible for the same value, we select the best with the *Highest Possible Lowest Variable* heuristic [11]. If enforcing arc consistency on $CSP(i)$ has failed, i.e., a domain was wiped out, the function returns false (line 23). Otherwise, if the domain of x_i was changed (i.e., there are some deletions to propagate), A_i informs its constrained agents by sending them ***del*** messages that contain nogoods justifying these removals (lines 26-27). Finally, the function returns true (line 28). When sending a ***del*** message to a neighboring agent A_j, only nogoods such that all variables in their left hand sides have a higher priority than A_j will be communicated to A_j. Furthermore, all nogoods having the same left hand side are factorized in one single nogood whose right hand side is the set of all values removed by this left hand side.

Whenever A_i receives a ***del*** message, it adds to the NogoodStore of the sender, say A_k, (i.e., $NogoodStore[x_k]$) all nogoods compatible with the AgentView of A_i (lines 30-32). Afterward, A_i checks if the domain of x_k is wiped out (i.e., the remaining values in $D(x_k)$ are removed by nogoods that have just been received from A_k) and x_i belongs to the NogoodStore of x_k (i.e., x_i is already assigned and its current assignment is included in at least one nogood removing a value from $D(x_k)$) (line 33). If it is the case, A_i removes its current value by storing the resolved nogood from the NogoodStore of x_k (i.e., $newNogood$) as justification of this removal (line 34-35), and then calls procedure Assign() to try an other value (line 36). Otherwise, when $D(x_k)$ is wiped-out (x_i is not assigned) or if a dead-end occurs when trying to enforce arc consistency, A_i has to backtrack and thus it calls procedure Backtrack() (line 38).

Each time a dead-end occurs on a domain of a variable x_k in $CSP(i)$ (including x_i), the procedure Backtrack is called. The nogoods that generated the dead-end are resolved by computing a new nogood ng (line 40). ng is the conjunction of the left hand sides of all these nogoods stored by A_i in $NogoodStore[x_k]$. If the generated nogood ng is empty, A_i terminates execution after sending a ***stp*** message to all agents in the system meaning that the problem is unsolvable (lines 41-43). Otherwise, A_i backtracks by sending a ***ngd*** message to agent A_j, the owner of the variable on the right hand side of ng (line 45). Next, A_i updates its AgentView in order to keep only assignments of agents that are placed before A_j in the total ordering (lines 46-47). A_i also updates the NogoodStore of all variables in $CSP(i)$ by removing nogoods incompatible with its new AgentView (line 48).

Whenever a ***ngd*** message is received, A_i checks the validity of the received nogood (line 49). If the received nogood is compatible with its AgentView, A_i adds this nogood to its NogoodStore (i.e. $NogoodStore[x_i]$, line 50). Then, A_i

checks if the value on the right hand side of the received nogood equals its current value. If it is the case, A_i calls the procedure `Assign()` to try another value for its variable (line 51). Otherwise, A_i calls function `Propagate()` to restore arc consistency. When a dead-end was generated in its local constraint network, A_i calls procedure `Backtrack()` (line 52).

B. *Enforcing AC without an additional kind of message*

In the following, we show how to enforce arc consistency without an additional kind of messages. In MACA-del, global consistency maintenance is achieved by communicating to constrained agents (agents in $CSP(i)$) all values pruned from $D^0(x_i)$. This may generate many ***del*** messages in the network, and then result in a communication bottleneck. In addition, many ***del*** messages may lead agents to perform more effort to process them. In MACA-not, communicating the removals produced in $CSP(i)$ is delayed until the agent A_i wants to send a ***cpa*** message. When sending the ***cpa*** message to a lower priority agent A_k, A_i attaches nogoods justifying value removals from $CSP(i)$ to the ***cpa*** message. But it does not attach all of them because some variables are irrelevant to A_k (not connected to x_k by a constraint). MACA-not shares with A_k all nogoods justifying deletions on variables not yet instantiated that share a constraint with both A_i and A_k (i.e., variables in $\{CSP(i) \cap CSP(k)\} \setminus vars(CPA)$). Thus, when A_k receives the ***cpa*** it receives also deletions performed in $CSP(i)$ that can lead him to more arc consistency propagation.

We present in Fig. 2 the pseudo-code of the MACA-not algorithm. Only procedures that are new or different from those of MACA-del in Fig. 1 are presented. Function `Propagate()` no longer sends ***del*** messages, it just maintains arc consistency on $CSP(i)$ and returns true iff no domain was wiped out.

In procedure `SendCPA(CPA)`, when sending a ***cpa*** message to an agent A_k, A_i attaches to the CPA the nogoods justifying the removals from the domains of variables in $CSP(i)$ constrained with A_k (lines 13-17, Fig. 2).

Whenever agent A_i receives a ***cpa*** message, procedure `ProcessCPA()` is called (line 6). The received message will be processed only when it holds a CPA stronger that the AgentView of A_i. If it is the case, A_i updates its AgentView (line 19) and then updates the NogoodStores to be compatible with the received CPA (line 20). Next, all nogoods contained in the received message are added to the NogoodStore (lines 21-23). Obviously, nogoods are added to the NogoodStore referring to the variable in their right hand side (i.e., ng is added to $NogoodStore[x_k]$ if x_k is the variable in $rhs(ng)$). Afterwards, A_i calls function `Propagate()` to restore arc consistency in $CSP(i)$ (line 24). If a domain of a variable in $CSP(i)$ wiped out, A_i calls procedure `Backtrack()` (line 24). Otherwise, A_i checks if it has to assign its variable (line 25). A_i tries to assign its variable by calling procedure `Assign()` only if it received the ***cpa*** from its predecessor, i.e., A_{i-1}.

```
procedure MACA-not()
1.  $end \leftarrow$ false;  Propagate() ;
2.  if ( $A_i = IA$ ) then  Assign() ;
3.  while ( $\neg end$ ) do
4.     $msg \leftarrow$ getMsg();
5.     switch ( $msg.type$ ) do
6.        cpa  : ProcessCPA($msg$);
7.        ngd  : ProcessNogood($msg$);
8.        stp  : $end \leftarrow true$;
procedure SendCPA(CPA)
9.  if ( size(CPA) $= n$ ) then
10.    broadcastMsg:stp ⟨CPA⟩ ;
11.    $end \leftarrow$ true ;
12. else
13.    foreach ( $x_k \succ x_i$ ) do
14.       $nogoods \leftarrow \emptyset$;
15.       foreach ( $x_j \in \{CSP(i) \cap CSP(k)\} \mid x_j \succ x_i$ ) do
16.          $nogoods \leftarrow nogoods \cup$ getNogoods($x_j$) ;
17.       sendMsg:cpa ⟨$CPA$, $nogoods$⟩ to $A_k$ ;
procedure ProcessCPA($msg$)
18. if ( $msg.CPA$ is stronger than the $AgentView$ ) then
19.    $AgentView \leftarrow CPA$ ;
20.    Remove all nogoods incompatible with $AgentView$ ;
21.    foreach ( $ng \in msg.nogoods$ ) do
22.       let $x_k$ be the variable in the $rhs(ng)$;
23.       add($ng$, $NogoodStore[x_k]$) ;
24.    if ( ¬Propagate() ) then  Backtrack() ;
25.    else if ( $msg.sender = A_{i-1}$ ) then  Assign();
function Propagate()
26. return AC($CSP(i)$) ;
```

Fig. 2. New lines/procedures for MACA-not with respect to MACA-del.

IV. Theoretical Analysis

We demonstrate that MACA is sound, complete and terminates, with a polynomial space complexity.

Lemma 1: MACA is guaranteed to terminate.

Proof: (Sketch) The proof is close to the one given in [23]. It can easily be obtained by induction on the agent ordering that there will be a finite number of new generated CPAs (at most nd, where n the number of variables and d is the maximum domain size), and that agents can never fall into an infinite loop for a given CPA. ■

Lemma 2: MACA cannot infer inconsistency if a solution exists.

Proof: Whenever a stronger CPA or a ***ngd*** message is received, MACA agents update their NogoodStore. In MACA-del, the nogoods contained in ***del*** are accepted only if they are compatible with AgentView (lines 30-32, Fig. 1). In MACA-not, the nogoods included in the ***cpa*** messages are compatible with the received CPA and they are accepted only when the CPA is stronger than AgentView (line 18, Fig. 2). Hence, for every CPA that may potentially lead to a solution, agents only store valid nogoods. 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. ■

Theorem 1: MACA is correct.

Proof: The argument for soundness is close to the one given in [23]. The fact that agents only forward consistent partial solution on the ***cpa*** messages at only one place in procedure `Assign()` (line 13, Fig. 1), implies that the agents receive only consistent assignments. A solution is found by the last agent only in procedure `SendCPA(CPA)` at (line 16, Fig. 1 and line 10, Fig. 2). At this point, all agents have assigned their variables, and their assignments are consistent. Thus, MACA is sound. Completeness comes from the fact that MACA is able to terminate and does not report inconsistency if a solution exists (Lemmas 1 and 2). ■

Theorem 2: MACA is polynomial in space.

Proof: On each agent, MACA stores one nogood of size at most n per removed value in its local constraint network. The local constraint network contains at most n variables. Thus, the space complexity of MACA is in $O(n^2 d)$ on each agent where d is the maximal initial domain size. ■

Theorem 3: MACA messages are polynomially bounded.

Proof: The largest messages for MACA-del are ***del*** messages. In the worst-case, a ***del*** message contains a nogood for each value. Thus, the size of ***del*** messages is in $O(nd)$. In MACA-not, the largest messages are ***cpa*** messages. The worst-case is a ***cpa*** message containing a CPA and one nogood for each value of each variable in the local constraint network. Thus, the size of a ***cpa*** message is in $O(n + n^2 d) = O(n^2 d)$. ■

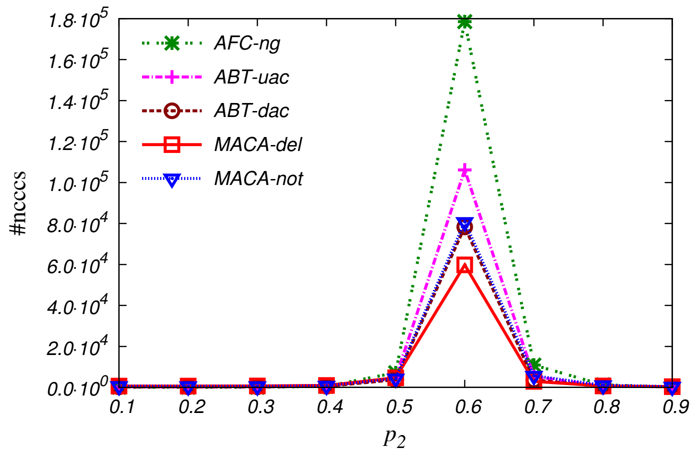
V. Experimental Results

In this section we experimentally compare MACA algorithms to ABT-uac, ABT-dac [6] and AFC-ng [8]. These algorithms are evaluated on uniform random binary DisCSPs. All experiments were performed on the DisChoco 2.0 platform[1] [22], in which agents are simulated by Java threads that communicate only through message passing. All algorithms are tested on the same static agents ordering (lexicographic ordering) and the same nogood selection heuristic (*HPLV*) [11]. For ABT-dac and ABT-uac we implemented an improved version of Silaghi's solution detection [19] and counters for tagging assignments.

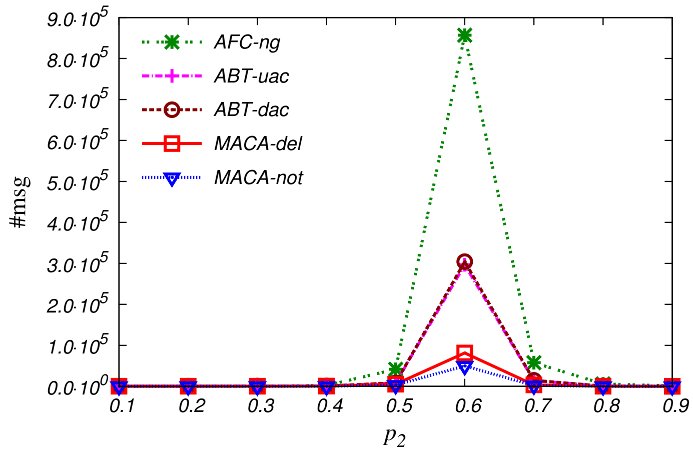
We evaluate the performance of the algorithms by communication load [13] and computation effort. Communication load is measured by the total number of exchanged messages among agents during algorithm execution (#*msg*), including those of termination detection (system messages). Computation effort is measured by the number of non-concurrent constraint checks (#*ncccs*) [28]. #*ncccs* is the metric used in distributed constraint solving to simulate the computation time.

The algorithms are tested on uniform random binary 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 is the network connectivity defined as the ratio of existing binary constraints, and p_2 is the constraint tightness defined as the ratio of forbidden value pairs. We solved instances of two classes of constraint networks: sparse networks $\langle 20, 10, 0.25, p_2\rangle$ and dense ones $\langle 20, 10, 0.7, p_2\rangle$. We vary the constraint tightness (i.e., p_2) from 0.1 to 0.9 by steps of 0.1. When we vary the constraint tightness, problems go through a phase transition [7, 21]. Before the phase transition, problems are relatively easy to solve. After the phase transition, it is very easy to prove that problems are unsolvable. At the phase transition, problems are in general difficult to solve or to prove unsolvable. By comparing algorithms at the phase transition we highlight their differences on difficult problems. For each pair of fixed density and tightness (p_1, p_2) we generated 100 instances. The average over the 100 instances is reported.

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



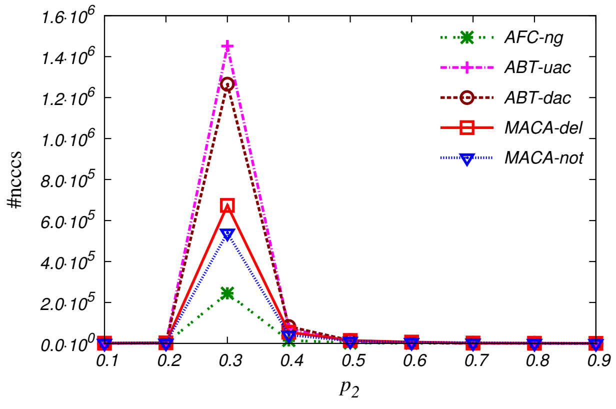
(a) #*ncccs performed on sparse random DisCSPs*



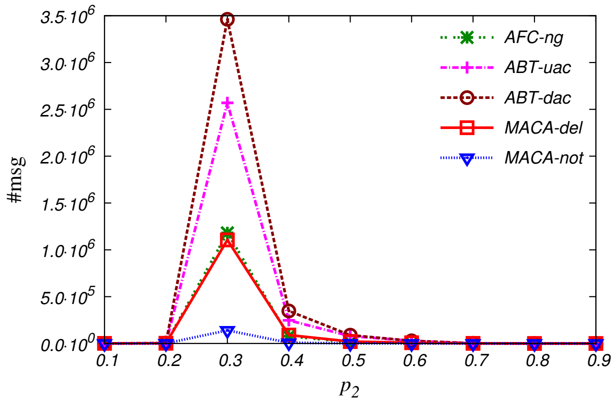
(b) *Number of messages sent on sparse random DisCSPs*

Fig. 3. The #*ncccs* performed and total number of messages sent for solving sparse uniform binary random DisCSPs problems where $\langle n = 20, d = 10, p_1 = 0.25\rangle$.

In Fig. 3, we present the performance of the algorithms on the sparse instances ($p_1 = 0.25$). Concerning the computational effort (Fig. 3(a)), algorithms enforcing an amount of arc consistency are better than AFC-ng, which only enforces forward checking. Among these algorithms MACA-del is the fastest one. MACA-not behaves like ABT-dac, which is better than ABT-uac. Concerning the communication load (Fig. 3(b)), algorithms performing an amount of arc consistency improve on AFC-ng by an even larger scale than for computational effort. ABT-uac and ABT-dac require almost the same number of exchanged messages. Among the algorithms maintaining an amount of arc consistency, the algorithms with a synchronous behavior (MACA algorithms) outperform those with an asynchronous behavior (ABT-dac and ABT-uac) by a factor of 6. It thus seems that on sparse problems, maintaining arc consistency in synchronous search algorithms provides more benefit than in asynchronous ones. MACA-not exchanges slightly fewer messages than MACA-del at the complexity peak.



(a) *#ncccs performed on dense random DisCSPs*



(b) *Number of messages sent on dense random DisCSPs*

Fig. 4. The *#ncccs* performed and total number of messages sent for solving dense uniform binary random DisCSPs problems where $\langle n = 20,\ d = 10,\ p_1 = 0.70\rangle$.

In Fig. 4, we present the performance of the algorithms on the dense instances ($p_1 = 0.7$). Concerning the computational effort (Fig. 4(a)), the first observation is that asynchronous algorithms are less efficient than those performing assignments sequentially. Among all compared algorithms, AFC-ng is the fastest one on these dense problems. This is consistent with results on centralized CSPs where FC had a better behavior on dense problems than on sparse ones [4, 9]. As on sparse problems, ABT-dac outperforms ABT-uac. Conversely to sparse problems, MACA-not outperforms MACA-del. Concerning the communication load (Fig. 4(b)), on dense problems, asynchronous algorithms (ABT-uac and ABT-dac) require a large number of exchanged messages. MACA-del does not improve on AFC-ng because of a too large number of exchanged ***del*** messages. On these problems, MACA-not is the algorithm that requires the smallest number of messages. MACA-not improves on synchronous algorithms (AFC-ng and MACA-del) by a factor of 11 and on asynchronous algorithms (ABT-uac and ABT-dac) by a factor of 40.

From these experiments we can conclude that in synchronous algorithms, maintaining arc consistency is better than maintaining forward checking in terms of computational effort when the network is sparse, and is always better in terms of communication load. We can also conclude that maintaning arc consistency in synchronous algorithms produces much larger benefits than maintaining arc consistency in asynchronous algorithms like ABT.

VI. Conclusion

We have proposed two new synchronous search algorithms for solving DisCSPs. These are the first attempts to maintain arc consistency during synchronous search in DisCSPs. The first algorithm, MACA-del, enforces arc consistency thanks to an additional type of messages, deletion messages. The second algorithm, MACA-not, achieves arc consistency without any new type of message. Despite the synchronicity of search, these two algorithms perform the arc consistency phase asynchronously. Our experiments show that maintaining arc consistency during synchronous search produces much larger benefits than maintaining arc consistency in asynchronous algorithms like ABT. The communication load of MACA-del can be significantly lower than that of AFC-ng, the best synchronous algorithm to date. MACA-not shows even larger improvements thanks to its more parsimonious use of messages.

References

[1] R. Béjar, C. Domshlak, C. Fernández, C. Gomes, B. Krishnamachari, B. Selman, and M. Valls, “Sensor networks and distributed CSP: communication, computation and complexity,” *Artificial Intelligence*, vol. 161, pp. 117–147, 2005.

[2] C. Bessiere, “Constraint Propagation,” in *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. New York, NY, USA: Elsevier, 2006, ch. 3.

[3] C. Bessiere and J.-C. Régin, “Refining the basic constraint propagation algorithm,” in *Proceedings of IJCAI’01*, 2001, pp. 309–315.

[4] C. Bessiere and J.-C. Régin, “MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems,” in *Proceedings of CP’96*, 1996, pp. 61–75.

[5] I. Brito and P. Meseguer, “Distributed Forward Checking,” in *Proceeding of CP’03*, Ireland, 2003, pp. 801–806.

[6] ——, “Connecting ABT with Arc Consistency,” in *CP*, 2008, pp. 387–401.

[7] P. Cheeseman, B. Kanefsky, and W. M. Taylor, “Where the really hard problems are,” in *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 1*, ser. IJCAI’91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 331–337.

[8] R. Ezzahir, C. Bessiere, M. Wahbi, I. Benelallam, and E.-H. Bouyakhf, “Asynchronous inter-level Forward-Checking for DisCSPs,” in *Proceedings of CP’09*, 2009, pp. 304–318.

[9] S. A. Grant and B. M. Smith, “The phase transition behaviour of maintaining arc consistency,” in *Proceedings of the 12th European conference on Artificial intelligence*, ser. ECAI’96, 1996, pp. 175–179.

[10] R. M. Haralick and G. L. Elliott, “Increasing tree search efficiency for constraint satisfaction problems,” *Artificial Intelligence*, vol. 14, no. 3, pp. 263–313, 1980.

[11] K. Hirayama and M. Yokoo, “The Effect of Nogood Learning in Distributed Constraint Satisfaction,” in *Proceedings of ICDCS’00*, 2000, pp. 169–177.

[12] H. Jung, M. Tambe, and S. Kulkarni, “Argumentation as Distributed Constraint Satisfaction: Applications and Results,” in *Proceedings of AGENTS’01*, 2001, pp. 324–331.

[13] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann Series, 1997.

[14] R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, and P. Varakantham, “Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling,” in *Proceedings of AAMAS’04*, 2004.

[15] A. Meisels and R. Zivan, “Asynchronous Forward-Checking for DisCSPs,” *Constraints*, vol. 12, no. 1, pp. 131–150, 2007.

[16] A. Petcu and B. Faltings, “A value ordering heuristic for distributed resource allocation,” in *Proceeding of CSCLP’04*, Feb 2004, pp. 86–97.

[17] P. Prosser, C. Conway, and C. Muller, “A constraint maintenance system for the distributed resource allocation problem,” *Intelligent Systems Engineering*, vol. 1, no. 1, pp. 76–83, oct 1992.

[18] D. Sabin and E. Freuder, “Contradicting conventional wisdom in constraint satisfaction,” in *Proceedings of CP’94*, vol. 874, 1994, pp. 10–20.

[19] M.-C. Silaghi, “Generalized dynamic ordering for asynchronous backtracking on DisCSPs,” in *DCR workshop, AAMAS-06*, 2006.

[20] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings, “Consistency maintenance for ABT,” in *Proceedings of CP’01*, 2001, pp. 271–285.

[21] B. M. Smith, “The phase transition and the mushy region in constraint satisfaction problems,” in *Proceedings of the 11th European conference on Artificial intelligence*, ser. ECAI’94, 1994, pp. 100–104.

[22] M. Wahbi, R. Ezzahir, C. Bessiere, and E.-H. Bouyakhf, “DisChoco 2: A Platform for Distributed Constraint Reasoning,” in *Proceedings of DCR’11*, 2011, pp. 112–121. [Online]. Available: http://www.lirmm.fr/coconut/dischoco/

[23] M. Wahbi, R. Ezzahir, C. Bessiere, and E. H. Bouyakhf, “Nogood-Based Asynchronous Forward-Checking Algorithms,” LIRMM, Tech. Rep. 12013, Apr. 2012. [Online]. Available: http://hal-lirmm.ccsd.cnrs.fr/lirmm-00691197

[24] R. J. Wallace and E. C. Freuder, “Constraint-based multi-agent meeting scheduling: effects of agent heterogeneity on performance and privacy loss,” in *Proceeding of the workshop on DCR’02*, 2002, pp. 176–182.

[25] M. Yokoo, “Algorithms for distributed constraint satisfaction problems: A review,” *Autonomous Agents and Multi-Agent Systems*, vol. 3, no. 2, pp. 185–207, 2000.

[26] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara, “The distributed constraint satisfaction problem: Formalization and algorithms,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 10, pp. 673–685, 1998.

[27] R. Zivan and A. Meisels, “Synchronous vs asynchronous search on DisCSPs,” in *Proceedings of EUMA’03*, 2003.

[28] ——, “Message delay and DisCSP search algorithms,” *Annals of Mathematics and Artificial Intelligence*, vol. 46, no. 4, pp. 415–439, 2006.