Asynchronous Maintenance of Arc Consistency

Imade Benelallam, Saida Hammoujan, Redouane Ezzahir and El Houssine Bouyakhf

LIMIARF, Faculty of sciences, Rabat, Mohammed the fifth University, Agdal, Morocco
imade.benelallam@ieee.org, hammoujan@ieee.org, ezzahir@lirmm.fr boughakhf@fsr.ac.ma

Abstract

Connecting backtracking search with arc consistency is one of the most powerful techniques for solving distributed constraint satisfaction problems (DisCSPs) [11]. Recently, it has been considered that to efficiently introduce asynchronisms in a distributed search, the best choice is the limited local consistency achieved by forward checking [22]. However, in centralized approaches [1, 5], it has been shown that maintaining arc consistency (which is a more pruningful local consistency) during searching, outperforms forward checking on hard and large constraint networks.

In this paper, we propose a new distributed constraint propagation algorithm, called Asynchronous Maintenance of Arc Consistency (AMAC), which is broadly based in our recent contribution: based-nogood AFC algorithm (AFC-ng) [21]. AMAC algorithm exploits consistency maintenance concurrently, in a way that lets potential inconsistencies caused by value deletions to be propagated. Implementation issues associated with the algorithm are described in detail and its correctness is proved. Simulation studies on benchmark problems show the effectiveness of the proposed ideas. AMAC algorithm has significantly better performance than other DisCSPs algorithms: ABT-dac, ABT-uac and AFC-ng [11, 22, 23]

Mathematics Subject Classification: Artificial intelligence (AI), Operational research

Keywords: Distributed Constraints Satisfaction Problems, Multi-Agents systems, Constraints propagation, Arc-consistency
1 Introduction

The notion of constraint is so naturally present in our current life, such as distributed resource allocation problems, distributed scheduling problems, Sensor Networks, etc. The concept of Constraints Satisfaction Problems CSPs, indicates all these problems, defined by constraints, and consisting of looking for a solution satisfying them. Most of the problems met daily can be distributed where the information of the problem can’t be collected in a single site, but it has to distribute them between the participating structures (Agents), because each of them tends to keep these information (constraints, values, variables) as private as possible. The notion of DisCSPs is mainly built of agents, each holding its local constraints network, that are connected by constraints among variables of different agents. Agents assign their values, attempting to generate a locally consistent assignment of its variables that is also consistent with all constrained agents [8, 13]. In order to solve this problem, agents check the value assignments to their variables for local consistency and exchange messages between the other agents, to check consistency of their proposed assignments.

The first distributed search algorithm was asynchronous backtracking ABT [14, 15], which still the most investigated [13, 17] after nearly ten years. However, in the last few years, several new approaches of asynchronous search algorithms have been proposed. In this paper we will focus more accurately on asynchronous forward-checking algorithm AFC [22].

The basic goal in the present work is to propose a new distributed algorithm for solving DisCSPs [11], called Asynchronous Maintenance of Arc Consistency (AMAC), that connects AFC-ng algorithm that we have proposed in [21] with arc consistency. In our approach, every agent keeps the triplet : assigned variables, its domains and its nogood stores. When receiving a new assignment or a relevent nogood, a better explanation is then provided for each value-deletion. The use of nogoods to list the conflicting assignments having higher priority, allows a better learning approach to prune unfeasible sub-problems. These deletions are then propagated to all agents with lower priority than the conflicting lists (nogood), producing new deletions in the domains of other variables, and so on.

This paper is organized as follows: First, we recall the DisCSP definition and AFC-ng algorithm description. Then, we present the idea of propagating arc-consistency in AMAC algorithm. Next, we present experimental results for this approach on random DisCSP instances. Finally, we extract some conclusions and directions for further research.
2 Preliminaries

2.1 Distributed Constraints Satisfaction Problems

In a centralized approach a Constraint Satisfaction Problem \((X, D, C)\) involves a finite set of variables \(X\), each taking values in a finite domain \(D\), and a finite set of constraints \(C\) over variables, that forbids some combinations of assignments on these variables. Formally speaking, a CSP can be described such that [3]:

- \(X = \{x_1, ..., x_n\}\) is a set of \(n\) variables;
- \(D = \{d_1, ..., d_n\}\) is a collection of finite domains such that each variable \(x_i\) takes values in \(d_i\);
- \(C\) is a finite set of constraints, every constraint \(c_i\) is in turn a restriction over a subset of variables \(\overline{c_i} = \{x_{i_1}, ..., x_{i_r}\}\) called its scope, in other words, a constraint \(c_i\) is a set of forbidden/permitted tuples over \(\overline{c_i}\).

A tuple \(S\) is a set of values corresponding to a set of variables \(\text{var}(S) \subseteq X\). The pair \(<x_i, v_i>\) means that variable \(x_i\) takes value \(v_i \in d_i\). Solving a constraint network means finding a tuple \(S\) such that \(\text{var}(S) = X\) and for all \(c_i \in C, S \downarrow_{\overline{c_i}} \in c_i\). We denote \(S \downarrow_X\) the projection of tuple \(S\) over the set of variables \(X\).

A Distributed Constraint Satisfaction Problem (DisCSP) is a CSP where variables, domains and constraints are distributed among several autonomous agents. Formally, a DisCSP is defined by a Quintuple \((X, D, C, A, \varphi)\), where \(X, D\) and \(C\) are defined as above, and:

- \(A = 1, ..., p\) is a set of \(p\) agents,
- \(\varphi : X \rightarrow A\) is a function that maps each variable to its agent.

The distribution of variables divides \(C\) in two disjoint subsets, \(c_{\text{intra}}\) called intra-agent constraints, which bind variables belonging to the same agent, these constraints are unknown by all the other agents, and \(c_{\text{inter}}\) called inter-agent constraints, that connect variables belonging to different agents. In a no confidential issue, the inter-agent constraint \(c_{\text{inter}}\) is known by every agent participating in \(c_{\text{inter}}\).

In a distributed algorithm, all the agents cooperate to find a coherent solution. The agents exchange messages containing the assignment of values of their variables, which satisfies every constraints. In this paper, it is assumed that the delay of a message is finite. For a given pair of agents, messages are delivered in the order they were sent and for simplicity, we assume that each agent owns exactly one variable.
2.2 Based-nogood AFC

The AFC-ng algorithm [21] combines the advantage of assigning values consistent with all former assignments and of propagating the assignments forward asynchronously. Assignments in AFC-ng are performed by multiple agents at a time, but only the shortest CPA is maintained in depth. The backtrack operation is generated for a failure in a future variable. Agents assign their variables only when they hold the current partial assignment (CPA) that sends from the nearest higher-order agent, and attempt to extend into a complete solution by assigning their variables on it. This algorithm runs on each agent exchanging only two main messages types:

- CPA\_MSG (Current Partial Assignment): a message that carries the currently consistent partial assignment. A CPA is composed of triplets of the form \(<x_i, v_i, counter_i>\) where \(x_i\) is the variable owned by the agent \(A_i\) and \(v_i\) is the value that was assigned to \(x_i\) by \(A_i\). Each CPA contains a timestamp which is an array of counters, a single counter for each agent. The partial assignment in a CPA is maintained in the order the assignments were made by the agents.

- BACK\_MSG: a message that contains a new CPA and a resolved nogood. It’s sent to the last agent assigned in the Rhs(nogood).

Forward checking is performed as follows: every agent that assigns its variable in the CPA sends copies of that CPA, in messages CPA\_MSG, to all unassigned agents. Only the next agent has the privilege to assign its variable in it. Agents that receive CPA\_MSG update their variables domains, by removing all no viable values. An agent that generates an empty domain as a result of a forward-checking operation, sends a new CPA in a backtrack message within a resolved nogood. The receiver of the message BACK\_MSG generates a new CPA and continues the search. The old CPA is detected as obsolete and discarded using the following method for time-stamping CPAs:

- The timestamp is an array of counters, a single counter for each agent.
- agent increments its counter when it performs an assignment.
- When two CPAs are compared, the more updated is the one whose timestamp is larger lexicographically.

Figure 1 presents the main procedures of the AFC-ng algorithm (see [21] for a detailed description of this algorithm). We note that we have made some minor changes on these procedures, in order to be adapted to AMAC extensions.
Asynchronous maintenance of arc-consistency

procedure Start()
1. myAgentView ← \{(A_j, ∅, 0) | A_j ≺ self\};
2. end ← false; myAgentView.Consistent ← true;
3. if (self = IA) then assign();
4. while (¬end)
5. msg ← getMsg();
6. switch (msg.type)
7. CPA_MSG: ProcessCPA(msg.CPA, msg.next);
8. Back_MSG: ProcessBackCPA(msg.CPA, msg.Nogood);
9. Terminate: ProcessTerminate(msg.CPA);

procedure assign()
10. if (myDomain ̸= ∅) then
11. myValue ← nextValue(); myCounter ← myCounter+1;
12. CPA ← myAgentView ∪ \{< self, myValue, myCounter >\};
13. if (self is the last agent) then BroadcastMsg: Terminate (myAgentView); end ← true;
14. next ← getNextAgent();
15. for each A_j ≻ self do SendMsg: CPA_MSG (CPA, next) to A_j;
16. else backtrack();

procedure backtrack()
17. Nogood ← resolveNogoods();
18. if (Nogood = empty) then broadcast(stop); terminate execution;
19. for each (A_j ≻ Rhs(Nogood)) and (A_j ∈ myAgentView) do
20. myAgentView.Value[A_j] ← Unknown;
21. for each ng ∈ myNogoodStore do
22. if (A_j ∈ Lhs(ng)) then remove(ng, myNogoodStore);
23. myAgentView.consistent ← False; Adjust CPA according to myAgentView;
24. SendMsg: Back_MSG (CPA, Nogood) to Rhs(Nogood);

ProcessCPA(CPA, next)
25. if (¬myAgentView.consistent) and (myAgentView ⊂ CPA) then return;
26. level ← compareTimeStamp(CPA);
27. if (level > 0) then
28. myAgentView.consistent ← True; updateMyAgentView(CPA, level);
29. filterInitialDomain();
30. if (myDomain = ∅) then backtrack();
31. elseif (self = next) then assign();

ProcessBackCPA(CPA, Nogood)
32. if (Lhs(Nogood) = ∅) then delete(RhsValue(Nogood));
33. if (RhsValue(Nogood) = myValue) then assign();
34. else
35. if (¬myAgentView.consistent and myAgentView ⊂ CPA) then return;
36. level ← compareTimeStamp(CPA);
37. if (level = 0 and myValue = RhsValue(Nogood)) then
38. myValue ← empty; add(Nogood, myNogoodStore); assign();

Figure 1: Nogood-based AFC algorithm running by agent self
3 Asynchronous Maintenance of AC

The main contribution in this paper is the proposed approach that propagate arc consistency in AFC-ng [21], namely, Asynchronous Maintenance of Arc Consistency algorithm (AMAC). One of the important goal in search algorithms is to detect earlier every backtracks in order to speed up the search and prune infeasible sub-problems. The AMAC algorithm is a great enhancement of the AFC-ng protocol; which maintains a full arc consistency during the search. The key idea in AMAC is to track the inconsistencies until there is no inconsistent value in the network, while sending forward the CPA. In AMAC algorithm, every new assignment is checked asynchronously by unassigned variables, and arc consistency is then propagated concurrently in both directions according to the left hand side of the resolved nogood.

One of the main features of AMAC algorithm is that it needs only a polynomial space requirements to store nogoods. Each agent in AMAC keeps a nogood store for all its neighbours (e.i. it represents the domains of constrained agents). If the nogood that starts propagation is no longer active, propagation must be undone, but if \( Lhs(Nogood) = \emptyset \), \( Nogood \) is always active, so propagation effects remain. Running on each agent, AMAC algorithm requires only one additional message regarding to AFC-ng algorithm, namely DEL\_MSG:

- DEL\_MSG: Message which contains a removing value and its explanation (nogood). Agent \( A_i \) informs its neighbours \( A_j \) that the value \( v_i \) has been deleted from \( d_i \). Agent \( A_j \) has less priority than the agent in the nogood corresponding to the value \( v_i \). Upon reception, \( A_j \) deletes value \( v_i \) from its \( d_i \) representation and stores its corresponding nogood, next it enforces AC on constraint \( C(j \rightarrow i) \). If, a consequence of AC enforcement, value \( v_j \) of \( A_j \) is deleted, this is propagated in the same way.

The Asynchronous Maintenance of Arc-Consistency algorithm is broadly based on AFC-ng algorithm. Each agent sends the CPA to next agents whose assignments are not yet on the CPA. Agents which receive CPA behaves like in AFC-ng, update their domains by removing all values incompatible with CPA, then send DEL\_MSG in order to enforce arc-consistency with constrained agents. The propagation of arc-consistency allows agents to detect inconsistent partial assignment and launch backtracks as soon as possible. An agent that produces a dead-end initialises a backtrack by sending a BACK\_MSG. In order to introduce revise-2001 function into AMAC’s agent, it’s important to note that we have to include some minor changes with respect to AFC-ng algorithm. Assuming that the agent \( self \) knows its neighbours, the domain of every variable constrained with \( self \) is also represented in \( self \), thus domain computation can be done by projecting constraints on the constrained variables. These constraints will be arc-consistent after the preprocessing.
3.1 Algorithm description

procedure start()
1. initPropagation(); *
2. ...;
10. DEL_MSG: ProcessDEL(msg.Nogood, msg.Sender); *

procedure initPropagation()
11. compute neighbourhoods; init structures of revise2001; *
12. for each \( A_j \in \text{my neighbours} \) do propagateAC(self, \( A_j \)); *

ProcessCPA(\( CPA, next \))
13. if ((\( \neg \text{myAgentView.consistent} \)) and \( \text{myAgentView} \subset CPA \)) then return;
14. level ← compareTimeStamp(CPA);
15. if (level > 0) then
16. myAgentView.consistent ← True;
17. updateMyAgentView(CPA, level); filterAndPropagate(); *
18. if (myDomain = \( \emptyset \)) then backtrack();
19. else (self = next) then assign();

ProcessBackCPA(\( CPA, Nogood, A_j \))
20. if (Lhs(Nogood) = \( \emptyset \)) then deleteValue(getRhsValue(Nogood), BACK_MSG.sender); *
21. if (getRhsValue(Nogood) = myValue) then assign();
22. else
23. if (\( \neg \text{myAgentView.consistent} \) and \( \text{myAgentView} \subset CPA \)) then return;
24. level ← compareTimeStamp(CPA);
25. if ((level = 0) and (myValue = getRhsValue(Nogood))) then
26. myValue ← empty; add(Nogood, nogoodStore[self]);
27. SendMsg : DEL_MSG (Nogood, self) to agents \( \succ lb(Lhs(Nogood)) \) except \( A_j \); assign(); *

ProcessDEL(Nogood, \( A_j \))
28. if (Lhs(Nogood) \( \subset \) myAgentView) then *
29. if (Rhs(Nogood) \( \subset \) myAgentView) then *
30. for each (self \( \in \) myAgentView) and (self \( \succ \) getRhsVar(Nogood)) do *
31. myAgentView[i] ← null; *
32. updateNogoodStore(); myAgentView.consistent ← False; return; *
33. if (Lhs(Nogood) = \( \emptyset \)) then *
34. domains[j] ← domains[j] − getRhsValue(Nogood); *
35. else add(Nogood, nogoodStore[j]); *
36. propagateAC(self, \( A_j \)); *

Figure 2: The AMAC algorithm (Part 1)

Look now what is going on in AMAC’s agent in figures 2 and 3. Here we underline only the affected and new instructions (indicated with an asterisk). First, before starting assignment by the initializer agent, every agent initializes propagation of arc-consistency with its neighbors through the function initPropagation() (line 1), the arc-consistency is then enforced continuously even during the search phase. If this initial review generates a domain wipeout through propagateAC() (line 44), the agent sends a stop message to all officers.
If this revision causes a deletion of at least one value, the agent sends a nogood with empty Lhs in DEL_MSG message to all agent below the last variable in Lhs of nogood \(lv(Lhs(nogood))\) (lines 47-48). If the value of \(self\) is deleted in this process, the assign() procedure is called (looking for a new compatible value, if none exists performs backtracking) (line 46). The CPA in AMAC is sent forward as in AFC-ng, when an agent \(self\) receives the CPA_MSG, instead of calling the procedure filterInitialDomain() it calls the procedure filterAndPropagate() (line 17), hence \(self\) revises its initial domain. Revising initial domain allows agents to store for every removed value the better nogood using HPLV heuristic (line 39), and send a DEL_MSG for every removed value and send in DEL_MSG nogood for every removed value returned by the function \(nogoodOf(v)\) (lins 41-42). The case where the agent receives the message BACK_MSG with a carried nogood, the inconsistent value is then removed unconditionally, when a nogood with empty Lhs has been accepted (line 20), or conditionally when a nogood with non-empty Lhs has been accepted (lines 26-27). In order to enforce arc-consistency, when DEL_MSG is received, the carried Nogood is accepted, if it is coherent with the agent view of \(self\) (line 28), otherwise, the message is considered as obsolete. Next, the agent \(self\) checks if the culprit assignment \(Rhs(Nogood)\) is a sub-set of the agent view (line 29), this means that \(Rhs(Nogood)\) has already or will backtrack, hence, \(self\) updates its agent view, sets its flag to false (lines 30-32), and does nothing. If the \(Lhs(Nogood)\) is empty (line 33) the value is deleted unconditionally from domains[j], otherwise, the non-empty \(Lhs(Nogood)\) is added. Arc-consistency is enforced within the procedure propagateAC(), which is based on revise 2001 [1]. When a value \(v\) is deleted since all values initially consistent with \(v\) in the domain of a constrained variable have been eliminated, a DEL_MSG is sent to all agents with less priority than the last agent in the Lhs of the generated nogoods, except agent who sends the DEL_MSG (line 39).

The procedure deleteValue() has the role of removing unconditionally the value \(v\) and propagate this deletion via DEL_MSG to the other constrained agents (line 42), if the domain of \(self\) is empty unconditionally (line 41) a stop message is then broadcasted to inform agents that the problem has no solution.

filterAndPropagate() has the role of filtering the domain of \(self\) and storing the better discovered nogoods, for each value recently removed, a DEL_MSG is sent to all agents with less priority than the last agent in the Lhs of the generated nogoods.

As in AFC-ng an agent uses the procedure assign() whether the CPA was the most recent received by message CPA_MSG or BACK_MSG. If the agent can instantiate its variable respecting the constraints, a solution is found if \(A_{i}\) is the lowest priority, otherwise the agent increments its counter and then sends the CPA to the following agent, and a copy to non-instantiated
agents. Otherwise, the agent changes the content of its AgentView highlighting the shortest inconsistent partial assignment and then begin the process of backtracking.

\textbf{procedure} \texttt{filterAndPropagate()}
\begin{algorithmic}
\State \textbf{for each} \((v \in myInitialDomain)\) \textbf{do} \texttt{*}
\State \hspace{1em} \textbf{if} \((-\text{Consistent}(v, myAgentView))\) \textbf{then} \texttt{*}
\State \hspace{2em} \text{store the better nogood for } v \text{ in } \text{nogoodStore}[self]; \texttt{*}
\State \hspace{1em} \text{del} \leftarrow \text{the newest removed values}; \texttt{*}
\State \hspace{1em} \textbf{for each} \((v \in \text{del})\) \textbf{do} \texttt{*}
\State \hspace{2em} \texttt{SendMsg : DEL\_MSG (nogoodOf(v), self) to agents } \succ \text{lv(Lhs(nogoodOf(v)))}; \texttt{*}
\end{algorithmic}

\textbf{procedure} \texttt{propagateAC(self, }A_j\text{)}
\begin{algorithmic}
\State \textbf{if} \((\text{revise}_{2001}(self, j))\) \textbf{then} \texttt{*}
\State \hspace{1em} \textbf{if} \((\text{myDomains is empty unconditionally})\) \textbf{then} \texttt{BroadcastMsg : Terminate (\emptyset); end} \leftarrow \text{true}; \texttt{*}
\State \hspace{1em} \textbf{else} \texttt{del} \leftarrow \text{the newest removed values in } \text{domains}[self] \text{ by } \text{revise}_{-2001}; \texttt{*}
\State \hspace{1em} \textbf{if} \((\text{myValue} \neq \text{empty} \land \text{myValue} \notin \text{myDomain})\) \texttt{then} \texttt{myValue} \leftarrow \text{empty}; \texttt{assign(); *}
\State \hspace{1em} \textbf{for each} \((v \in \text{del})\) \textbf{do} \texttt{*}
\State \hspace{2em} \texttt{SendMsg : DEL\_MSG (nogoodOf(v), self) to agents } \succ \text{lv(Lhs(nogoodOf(v)))} \text{ except } A_j; \texttt{*}
\end{algorithmic}

\textbf{procedure} \texttt{deleteValue(v, }A_j\text{)}
\begin{algorithmic}
\State \texttt{myDomains} \leftarrow \text{myDomains} \setminus \{v\}; \texttt{*}
\State \textbf{if} \((\text{myDomains is empty unconditionally})\) \textbf{then} \texttt{BroadcastMsg : Terminate (\emptyset); end} \leftarrow \text{true}; \texttt{*}
\State \textbf{else} \texttt{SendMsg : DEL\_MSG (nogoodOf(v), self) to agents } \succ \text{lv(Lhs(nogoodOf(v)))} \text{ except } A_j; \texttt{*}
\end{algorithmic}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{The AMAC algorithm (Part 2)}
\end{figure}

4 Algorithms correctness

In this section, we concentrate on proving correctness of AMAC algorithm that is immediately involved by the correctness of AFC-ng. To show this, we would need to prove the soundness, completeness and termination:

\textbf{Theorem 4.1} AFC-ng is sound, complete, and terminates.

The argument for soundness is close to the one given in [22, 25]. The fact that agents only forward consistent partial solution on the CPAs messages at only one place in function \texttt{assign()} \ (line 15), implies that the agents receive only consistent assignments. A solution is reported by the last agent only at line 13. At this point, all agents have assigned their variables, and their assignments are consistent. Thus the AFC-ng algorithm is sound.

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

\textbf{Lemma 4.2} AFC-ng is guaranteed to terminate.
For sake of clarity, we assume that the order in which AFC-ng assigns the variables is the lexicographic ordering $X_1, X_2, \ldots, X_n$. We define the total order $o$ on CPAs as follows. Let $I_1$ be an assignment on $X_1, \ldots, X_{k_1}$, $I_2$ be an assignment on $X_1, \ldots, X_{k_2}$, and $s$ be the smallest index on which $I_1$ and $I_2$ differ. $I_1 \prec_o I_2$ if and only if $s = k_1 + 1$ or the value $I_1[s]$ is chosen before the value $I_2[s]$ by the value ordering heuristics on variable $X_s$ given the CPA $I_1[1..s-1]$.

To prove the lemma we prove that AFC-ng performs a finite number of backtrack steps. In AFC-ng, several backtracks can be performed simultaneously as they are generated concurrently by different agents to different destinations. The re-assignments of destination agents then happen simultaneously, generating several CPAs. However, the CPA at the highest level in the search hierarchy tree will eventually dominate all others thanks to its greater time-stamp (see line 27 in Figure 1). Thus, every backtrack step may be represented by the backtrack at the highest level. The agent $X_i$ who has received that backtrack of highest level has to replace its previous assignment $v_i$ in the CPA by a new one $v'_i$ because the backtrack message contains a nogood rejecting value $v_i$. If $v_i$ was not the first value chosen by $X_i$ since it has received the current CPA from $X_{i-1}$ then we know that all other values $v_j$ preferred to $v_i$ were ruled out by a nogood at the time $v_i$ was chosen. Now, the CPA on $X_1, \ldots, X_{i-1}$ has not changed since then, otherwise this would not be the highest backtrack. As a result, the nogoods rejecting values $v_j$ preferred to $v_i$ are still valid and $v'_i$ is necessarily the next preferred value in the heuristic order. By definition of the order $o$, the new CPA obtained is greater than the previous one according to $o$ because it has not changed on $X_1, \ldots, X_{i-1}$ and $v'_i$ is less preferred than $v_i$. Since $o$ is a total order and since there are a finite number of variables and a finite number of values per variable, there will be a finite number of new CPAs generated. Now, each backtrack of highest level generates a new CPA. Thus, AFC-ng performs a finite number of backtracks.

**Lemma 4.3** AFC-ng cannot infer inconsistency if a solution exists.

Whenever a newest CPA or a BackCPA message is received, AFC-ng agent updates its NogoodStore. Therefore, for every CPA that may potentially lead to a solution, agents only store valid nogoods. In addition, every nogood resulting from a CPA message 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.

A central fact which can be established at once is that the correctness of AMAC algorithm is broadly based on AFC-ng. The correction of AMAC includes the safety, the completeness and the ending.
Theorem 4.4 AMAC is sound, complete, and terminates.

Agents in AMAC send forward only coherent partial instantiations, and the extent of CPAs remains the same as in AFC-ng, the following Lemma is held:

Lemma 4.5: AMAC extend only the coherent partial instantiations. The partial instantiations are received via a CPA and are extended and sent forward by the agent of reception.

Above, It was proved that AFC-ng is sound. Since that assign() procedure remains unchanged and the only change that has been reported in ProcessCPA procedure is the propagation of DEL_MSG (Figure 2 line 17), and the solution is announced when a CPA includes a complete and coherent instantiation. This can strictly implies the soundness of AMAC. For completeness it is only necessary to avoid the stop of the algorithm having to find the first solution.

Lemma 4.6: AMAC executes a number finished by the backtrack.

AFC-ng maintains a limited form of arc consistency while AMAC maintains full arc consistency during search. However, the number of backtrack of AMAC algorithm can never exceed the total number of AFC-ng’s backtrack, then the number of the possible backtracks is also finished for AMAC, which proves the ending of the algorithm.

5 Experimental results

There is only one way to know which DisCSP algorithm is the better in practice. You have to implement the algorithm and measure its performances. This may seem like a simple task, but there is many pitfalls. The results will be influenced by such factors as supported plate-form, compilers, and machine architecture. Also, the amount of work spent on implementing a particular algorithm is likely to affect its efficiency. We have implemented all the algorithms in our laboratory plate-form, named DisChoco [20], in which agents are simulated by threads which communicate only through message passing.

In this paper we experimentally evaluate the performance of AMAC with respect to the past extended arc-consistency distributed algorithm, namely ABT-uac and ABT-dac on uniform binary random DisCSP. A binary random DisCSP class is characterized by $\langle n, d, p_1, p_2 \rangle$, where $n$ is the number of variables (agents), $d$ the number of values per variable, $p_1$ the network connectivity defined as the ratio of existing constraints, and $p_2$ the constraint tightness defined as the ratio of forbidden value pairs.
The experimental setup included problems generated with 20 variables \((n = 20)\) and 10 values \((d = 10)\). In this paper, the experiments include DisCSPs with two different network density values \(p_1 = 0.5\) and \(p_1 = 0.8\). The value of \(p_2\) was varied between 0.1 and 0.9 in increments of 0.1, to cover all ranges of problem difficulty \([18]\). Algorithmic performance is evaluated by communication cost and computation effort. Communication cost is measured by the total number of exchanged messages (MSGs). Computation effort is measured by the number of non-concurrent constraint checks (NCCCs), results are averaged on 20 instances.

![Figure 4](image-url)  
(a)  
(b)

Figure 4: Non concurrent constraints checks and communication cost performed by AMAC on random DisCSPs problems \((p_1 = 0.5)\)

Figures 4-(a) and (b) present the results in run-time and network load of AMAC, ABT-uac, ABT-dac and AFC-ng for low density DisCSPs. AMAC outperforms ABT-uac and ABT-dac by a factor of 5 in number of NCCCs and by a factor of 2 in the total number of messages, we show also that AFC-ng gives better performance in respect to ABT-uac and ABT-dac, this due to huge number of arc consistency messages propagated in the constraint network.

Figures 5-(a) and (b) present the results of the same comparison for higher density DisCSPs. For higher density networks, the factor of improvement of AMAC over the other algorithms in run-time and network-load is increased by a large factor in favor of AMAC.

Figures 6-(a) and (b) present the results in run-time and network load of AMAC, ABT-uac and ABT-dac for different number of agents. We show that AMAC outperforms ABT-uac and ABT-dac with a significant scalability.
6 Conclusion

A new distributed propagation and search algorithm for distributed constraints networks was presented. The proposed algorithm incorporates arc-consistency. We have proved that the proposed AMAC algorithm is correct, complete and terminate. We have shown analytically and experimentally that this idea effectively enhances performances in comparison with ABT-uac and ABT-dac, because it increases both communication and computation effort. We believe that this approach could be useful for those applications with high density on DisCSPs. The main conclusion of the present study is that propagation
of arc-consistency performed in distributed search enhances the efficiency of the search process. The performance of asynchronous maintenance of arc-consistency generates a more efficient search than ABT connected with arc-consistency [11].

ACKNOWLEDGEMENTS. Words of thanks, recognition and appreciation are owed to C. Bessiere for their generosity and support to our IA team;

References
Asynchronous maintenance of arc-consistency


**Received: August, 2009**