Asynchronous aggregation and consistency in distributed constraint satisfaction

Marius-Călin Silaghi [a,*] , Boi Faltings [b]

[a] *Computer Sciences Department, Florida Institute of Technology (FIT), Melbourne, FL 32901, USA*

[b] *Artificial Intelligence Laboratory, Swiss Federal Institute of Technology (EPFL), 1015 Lausanne, Switzerland*

Abstract

Constraint Satisfaction Problems (CSP) have been very successful in problem-solving tasks ranging from resource allocation and scheduling to configuration and design. Increasingly, many of these tasks pose themselves in a distributed setting where variables and constraints are distributed among different agents.

A variety of asynchronous search algorithms have been proposed for addressing this setting. We show how two techniques commonly used in centralized constraint satisfaction, *value aggregation* and *maintaining arc consistency* can be applied to increase efficiency in an asynchronous, distributed context as well, and report on experiments that quantify the gains.

1 Introduction

Constraint Satisfaction Problems (CSP) have wide applicability to problem-solving tasks ranging from resource allocation and scheduling to configuration and design. A constraint satisfaction problem (CSP) is given by:

- a set of n *variables* $X = \{x_1, ..., x_n\}$,
- a set of n *domains*, $D = \{d_1, ..., d_n\}$, for the variables,
- a set of t *constraints*, $C = \{c_1 = (x_i, x_j, ...), ...,c_t\}$, each of which is a subset of the set of variables, linked with a relation, and

* Corresponding author.
Email addresses: Marius.Silaghi@cs.fit.edu (Marius-Călin Silaghi), Boi.Faltings@epfl.ch (Boi Faltings).

- a set of t *relations*, $R = \{r_1, ..., r_t\}$. r_i gives the allowed value combinations for the corresponding constraint c_i.

A *solution* to a CSP is an assignment of values from the corresponding domains to each variable such that for all constraints, the combination of assigned values is allowed by the corresponding relation. Many combinatorial problems, such as resource allocation, scheduling and planning can be modeled as CSPs.

Distributed constraint satisfaction problems (DisCSPs) arise when constraints and/or variables are controlled by a set of independent but communicating agents. In the common definition of DisCSP [41], variables are distributed among agents so that each variable can only be assigned values by a single agent. A DisCSP is thus obtained from a CSP by adding:

- a set of m *agents* $A = \{A_0, .., A_m\}$
- an ownership mapping $M : X \cup C \rightarrow \mathcal{P}(A)$ that assigns each variable or constraint to the subset of agents that own it (in [41], the subset of agents owning any given variable is supposed to contain exactly one agent). $\mathcal{P}(A)$ is a common notation for the set of subsets of A.

The value of a variable can only be set by its owner. For simplicity, one often assumes that each agent A_i owns exactly one variable x_i. If in some application an agent owns several variables, the same agent can take multiple roles in the protocol.

Constraint satisfaction problems are often solved by backtrack search. Protocols have been proposed for carrying out such backtrack search as message exchanges between agents. In particular, we are interested in asynchronous protocols where agents can proceed independently without explicit synchronization. Asynchronism gives the agents more freedom in the way they can contribute to search, allowing them to enforce individual policies (on privacy, computation, etc.). It also increases both parallelism and robustness. In particular, robustness is improved by the fact that the search can still detect unsatisfiability even in the presence of withdrawn or crashed agents. Similarly, the information an agent provided before withdrawing may be sufficient to prove a solution. Distributed solutions also provide a certain security against manipulation by a centralized solving agent. Several *Asynchronous Search* (AS) algorithms have been developed that allow solving such problems by exchanging messages about variable assignments and conflicts with constraints (called nogoods) [41,23,1,4].

In centralized settings, the efficiency of backtrack search can be improved significantly using a combination of techniques:

- aggregation of variable values into larger meta-values to reduce constraint checks on individual values,
- consistency techniques to prune values that could not be part of any solution, and to detect failure early.

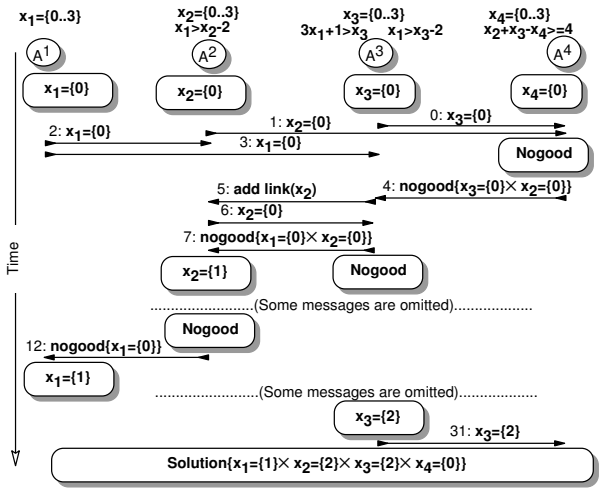


Fig. 1. Simplified trace of an asynchronous search process (for simplicity shown messages are drawn as if delivered instantaneously). Each agent A_i is associated with a variable x_i, a set of constraints involving this variable and states represented by boxes. A state shows either the assignment chosen for the owned variable or a conflicting situation (a nogood is just being inferred). The arrows represent messages. Each message is prefixed by a number.

In this paper, we show how protocols for asynchronous search can be adapted to include these techniques as well, and report on the efficiency gains that can be empirically observed on random problems.

2 Basic Asynchronous Search Algorithm

This section introduces the basic asynchronous search algorithm into which we are going to integrate the techniques we are proposing. It is the original ABT algorithm of Yokoo et.al. [41]. We use the example shown in Figure 1 to illustrate the concepts. We restrict our description to the case with unbounded nogood recording ([41]) and where each agent has exactly one variable.

In this framework, each agent is responsible for maintaining the value of one variable. It has a link toward any agent that owns (i.e. knows) a constraint involving that variable. Agents are arranged in a fixed priority order $\succ$, and we assume that $A_i \succ A_j$ (A_i has higher priority than A_j) iff $i < j$ (total order among agents). A constraint is enforced by the agent which has the lowest priority among those that are responsible for the variables in the constraint.

In our example, there are four agents, A_1, A_2, A_3, A_4 who control the variables x_1, x_2, x_3, x_4, with identical domains d_1=d_2=d_3=d_4=$\{0, 1, 2, 3\}$. Agents have the following constraints:

- A_1: $c_1(x_1, x_3)$, $r_1 = 3x_1 + 1 > x_3$

- A_2: $c_2(x_1, x_2)$, $r_2 = x_1 > x_2 - 2$
- A_3: $c_3(x_1, x_3)$, $r_3 = x_1 > x_3 - 2$
- A_4: $c_4(x_2, x_3, x_4)$, $r_4 = x_2 + x_3 - x_4 \geq 4$

In order to solve this problem with conventional AS techniques, we first need to assign a priority to each agent, then move certain constraints to the agent with the lower priority. As A_{i+1} has precedence over A_i, so that for example A_1's constraint c_1 has to be communicated to A_3 which will be responsible for its enforcement. Every constraint-evaluating agent will create communication links with the agents controlling variables in that constraint.

At every moment in the search, every agent maintains an *agent view* that describes its local view of the search space. The agent view consists of:

- the current assignments it knows for its own variable as well as for all variables of agents with higher priority that it has a constraint with,
- the currently valid *nogoods* involving its own variable. An agent can either choose to store all nogoods that it has ever received, entailing potentially exponential growth in memory requirements, or only those that are valid given the assignments in its agent view. In the latter case, nogoods might need to be rederived.

Each agent will start by randomly assigning to its variable a value from its domain (0 in our example). As detailed in Algorithm 1, when a value has to be chosen for its variable, the local search space for each agent is determined by its local constraints along with the restrictions imposed by the other agents via **ok?** and **nogood** messages. When an agent assigns a value v to its variable x, it sends an **ok?**(x=v) message to all the lower-priority agents having a link with it. These agents then evaluate their constraints on that variable. If these constraints are satisfied by the new assignment, given all the known values for the other variables, they do nothing, otherwise they try a new value for their variable. If any of them finds no available value, it generates a **nogood** message, sent to the lowest priority agent generating a culprit assignment. The agent receiving this **nogood** message will then have to incorporate the information in its local search space and change the faulty assignment or generate other nogoods, accordingly. Hence, constraints are always evaluated by lower-priority agents and values always changed by higher priority ones.

Figure 1 shows a simplified trace of message passing obtained for our example using the asynchronous backtracking algorithm described in [41]. Each agent starts by assigning the value 0 to its variable. Agent A_1 then sends an **ok?** message to A_2 and A_3 and agents A_2 and A_3 both send **ok?** messages to A_4. Agents A_2 and A_3 both find the value received from A_1 to be compatible with their constraints. Hence, they do not react. However, A_4's constraint is violated and this agent returns a **nogood** message (4) to A_3.

when received *(***ok?**,$\langle x_j, d_j \rangle$*)* **do**
- add $\langle x_j, d_j \rangle$ to **agent_view**;
- **check_agent_view**;

when received *(***nogood**,A_j,$\neg N$*)* **do**
- **when** $\langle x_k, d_k \rangle$*, where* x_k *is not connected, is contained in* $\neg N$
  - send **add-link** to A_k;
  - add $\langle x_k, d_k \rangle$ to **agent_view**;
- put $\neg N$ in *nogood-list*;
- *old_value* $\leftarrow$ *current_value*;
- **check_agent_view**;
- **when** old_value $=$ current_value
  - send (**ok?**,$\langle x_i,$ *current_value*$\rangle$) to A_j;

procedure check_agent_view do
- **when agent_view** *and* current_value *are not consistent*
  - **if** *no value in* D_i *is consistent with* **agent_view then**
    - **backtrack**;
  - **else**
    - select $d \in D_i$ where **agent_view** and d are consistent;
    - *current_value* $\leftarrow$ d;
    - send (**ok?**,$\langle x_i, d \rangle$) to lower priority agents in *outgoing links*;

procedure backtrack do
- *nogoods* $\leftarrow \{V \mid V =$ inconsistent subset of **agent_view**$\}$;
- **when** *an empty set is an element of* nogoods
  - broadcast to other agents that there is no solution;
  - terminate this algorithm;
- **for every** $V \in$ nogoods **do**
  - select $\langle x_j, d_j \rangle$ where x_j has the lowest priority in V;
  - send (**nogood**,A_i,V) to A_j;
  - remove $\langle x_j, d_j \rangle$ from **agent_view**;
- **check_agent_view**;

Algorithm 1: Procedures of A_i for receiving messages in ABT.

3 Value aggregation: from ABT to AAS

Asynchronous Aggregation Search (AAS) is an extension of ABT where constraints can be private data of some agents and several agents are allowed to simultaneously propose instantiations for the same shared variable. Coupled with the fact that AAS allows aggregating ranges of tuples, we obtain efficiency gains over the existing asynchronous backtracking algorithms. The evaluation is done using three different implementations, based respectively on full, partial and no nogood recording.

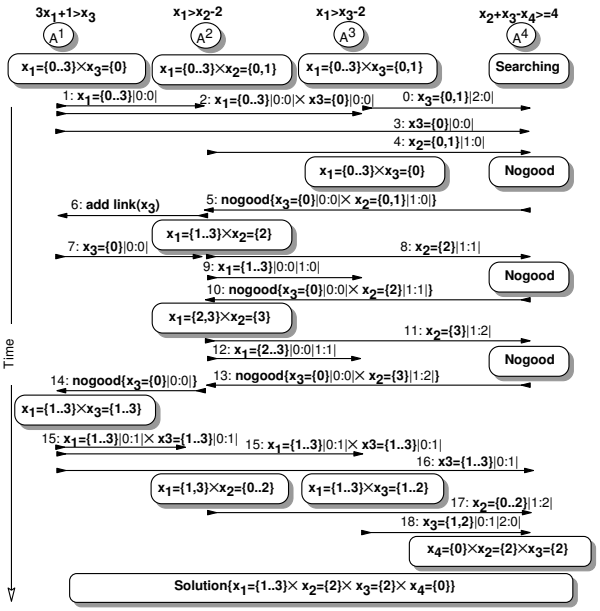


Fig. 2. Trace of a search with AAS. The states of the agents can be represented by the current solution to the local CSP defined by their constraints. The pairs $|a, b|$ included in the messages are used for message ordering.

In AAS, each agent maintains valuation proposals for the set of variables in which it is involved. Thus, A_1 maintains valuation proposals for x_1 and x_3, A_2 for x_1 and x_2, A_3 for x_1 and x_3, and A_4 for all of x_2, x_3 and x_4 (see Figure 2). AAS differs from ABT in the fact that the valuation proposals are not just individual assignments, but sets of aggregated alternative assignments [17] to different variables. More precisely, an **ok?** proposal is a list of domains, one for each involved variable, which represent all the tuples of their Cartesian product. The proposal **ok?**$(x_1 = \{0..3\}, x_2 = \{0, 1\})$, for example, will say that all the tuples of the Cartesian product $\{0..3\}\times\{0, 1\}$ satisfy the sending agent given its current agent-view. Similarly, the result of the search is no longer a list of individual assignments, but a set of domains whose Cartesian product contains only solutions.

Figure 2 illustrates the behavior of AAS on our small example. Agent A_1 first selects the valuation proposals $\{x_1 = \{0..3\}\}, \{x_3 = \{0\}\}$, whose Cartesian product satisfies it, and sends an **ok?** message with the needed parts of this information (as defined by existing links) to A_2, A_3 and A_4 who manage constraints sharing variables with A_1. The algorithm now works in exactly the same manner as ABT, except that messages refer to sets of alternative valuations, called aggregates. Agents also reason in terms of sets of alternative (partial) solutions represented with Cartesian products rather than solely about isolated partial assignments. More specifically, A_4 finds that the addition to its initial constraint of the unary constraints $\{x_2 \in \{0, 1\}\}$ and $\{x_3 \in \{0\}\}$ defined by its agent view leads to an insatisfiable problem. This is the same as saying that no extension of any combination in the Cartesian product $\{x_2 = \{0, 1\}\} \times \{x_3 = \{0\}\}$ defined by its agent view is compatible with its constraint. It therefore generates a **nogood** for these received proposals causing A_2 to

select the next proposal (defining another Cartesian product). Note that since this change selects a subrange of the values allowed by the knowledge of A_2 for x_1, it is not necessary to verify this change with A_1. If it were not possible to find such a subrange, a **nogood** would be generated and sent to A_1 in order to try another Cartesian-product there.

There are several ways in which the agents can build the aggregations. Aggregation algorithms guaranteeing a complete and non-redundant covering of the solution space determined by local constraints are given in [17,14,32]. The choice of exactly which of these algorithms should be used depends also on other decisions. For example, if one decides to restrict the domain representations to ranges, then the algorithms in [32] may be preferred, otherwise the algorithms in [17] are the right choice. Actually, the applications we target with our techniques (i.e. negotiation) cannot allow global policies to be enforced at this level, but they are dictated by private considerations of the agent. For this reason we considered that it was not warranted to work on optimizing aggregation policies, but just to enable agents to use aggregations.

In the following we only assume that the aggregation technique we use terminates in a finite time, that it is sound (i.e. all the elements of the Cartesian product it returns are solutions), and that it is complete (i.e. it returns a solution whenever it exists). For the reader that wants a simple example, Chronological Backtracking is such a technique where each Cartesian product contains a single element.

3.1 AAS Algorithms

In this section we will present three versions of a distributed backtrack search algorithm based on aggregation. We start by giving the necessary definitions. Similarly to the ABT algorithm of [41], the agents are assigned priorities. We assume that the agent A_i has priority over another agent A_j if $i < j$. An agent A_i is *interested* in all variables that it controls or that are involved in constraints that it enforces. It has a *link* with every agent that controls a variable it is interested in. When two agents A_i and A_j are related by a link and $i < j$, thus $A_i \succ A_j$, the link is directed from the higher priority agent A_i to A_j. A_i is called the *predecessor* of A_j and A_j is called the *successor* of A_i. The *end agents* are those without incoming links. For tests/bootstrap purposes we employ a *system agent*, a special agent that receives the subscriptions of the agents for the search. It decides the order of the agents, initializes the links and announces the termination of the search.

Definition 1 (Assignment) *An assignment is a triplet (x_j, set_j, h_j) where x_j is a variable, set_j a range of values for x_j and h_j a history of the pair (x_j, set_j).*

The history provides the information necessary for a correct message ordering. It determines if a given assignment is more recent than another and will be described

in more details later. Let $a_1 = (x_j, set_j, h_j)$ and $a_2 = (x_j, set'_j, h'_j)$ be two assignments for the variable x_j. a_1 is *newer* than a_2 if h_j is more recent than h'_j.

Definition 2 *An* aggregate *is a list of assignments.*

Definition 3 (Explicit nogood) *An explicit nogood has the form $\neg V$, where V is an aggregate.*

The agents communicate using channels without message loss via:

- **ok?** messages which have as parameter an aggregate. They represent proposals of domains for a given set of variables and are sent from agents with higher priorities to agents with lower priorities. An agent sends **ok?** messages containing only domains of variables in which the target agent is interested. He does not send messages with any assignment of a variable x which does not modify the domain of the most recent assignment that he already knowns for x. If he has not just discarded a recent applicable nogood [1] , then he sends only the domains for which he proposes a new modification now. **ok?** messages are also sent as answers to **add-link** messages.
- **nogood** messages which have as parameter an explicit nogood. A **nogood** message is sent from an agent with lower priority to an agent with higher priority, namely to the agent with the lowest priority among those that have modified an assignment in the parameter.
- **add-link**(vars) messages: sent from agent A_j to agent A_i (with $j > i$). They inform A_i that A_j is interested in the variables $vars$.

Each agent A_i owns a set of local constraints. The *current solution space* of A_i, denoted as C_{A_i}, is described by the local constraints, a list of nogoods and a *view*.

Definition 4 (View) *The view of an agent A_i is an aggregate V containing received assignments for variables A_i is interested in.*

A view imposes restrictions on the original search space defined by the local constraints of an agent. It contains for each variable, the newest received assignment via incoming messages. Each assignment (x_j, set_j, h_j) found in the view of A_i defines an entailed unary constraint: $\{x_j \in set_j\}$.

Definition 5 (Entailed nogood) *Let V_1 be the view of a given agent, T be the set of tuples disabled from the original solution space by the entailed unary constraints of the assignments in V_1. We say: The nogood $V_1 \rightarrow \neg T$ is entailed by the view V_1.*

A tuple is *contained* in the current solution space of agent A_i if it satisfies the local constraints and is not contained in the explicit or entailed nogoods of C_{A_i}. The

[1] This refers to nogoods discarded, as described later, since the last instantiation, within the reset CL of AAS0

current instantiation of an agent A_i is a Cartesian product such that all its tuples are contained in C_{A_i}. The list of nogoods, respectively the view, of an agent A_i is updated by the **nogood**, respectively **ok?** messages it receives.

We now propose the following three distributed backtrack search algorithms based on aggregation:

- AAS2: is based on full nogood recording similarly to the ABT algorithm of [40].
- AAS1: proceeds similarly to dynamic backtracking [11]. It removes the nogoods depending on the instantiation of the modified variables, allowing for guarantees of polynomial space complexity.

  As at most one valid nogood is stored for each issued proposal, the exact space required is a polynomial function *in the size of the local search space of the agent*, sn^2. But this size, s, is given by the maximal number of variables that the agent can instantiate and when this number is not bounded, as in the general case considered so far, we end up with an exponential function in the number of variables, n^2d^n, where d is the domain size.

  Two solutions exist to this problem. One is to renounce to generality and to arbitrarily decide on an upper-bound in the number of variables that can be assigned by an agent (e.g. see the upper-bound of one set in [4]). Such an upper-bound brings no other difference in the algorithms and therefore will not receive additional attention in this paper. A more radical alternative is the next version.
- AAS0: is a modification of AAS1 with less nogood recording. AAS0 is a novel algorithm which merges all the nogoods maintained by each agent of AAS1 into a single nogood using the relaxation rule:

$$\frac{\begin{array}{l} V_1 \wedge V_2 \rightarrow \neg T_1 \\ V_1 \wedge V_3 \rightarrow \neg T_2 \end{array}}{\Rightarrow V_1 \wedge V_2 \wedge V_3 \rightarrow \neg(T_1 \vee T_2),}$$

  where V_1, V_2 and V_3 are aggregates of assignments generated by other agents, obtained by grouping the elements of the nogoods, such that they have no variable in common. Each agent maintains a single explicit nogood which integrates each new incoming explicit nogood using the relaxation rule.

  In the case of AAS0, the right part of the nogood description corresponds to the expanded tuples and the left one is referred to as the conflict list (CL).

The core backtrack procedure that we used in our experiments for each agent is the same for the three algorithms, and is as expected exponential in the arity of the constraints of the agent. The AAS algorithm obtained from ABT with the extensions proposed here is described by the pseudocode of Algorithm 2 and Algorithm 3. Each agent A_i stores the newest history received for each variable x_k, history(x_k), as well as a counter, $C^i_{x_k}$, of the number of assignments sent for x_i. The consequence of a nogood $\neg N$ from the point of view of A_i, consequence$_i(\neg N)$, is the

when received *(***ok?**,$\langle x_j, s_j, h_j\rangle$*)* **do**
  if(history(x_j) invalidates h_j) return;
  add($\langle x_j, s_j, h_j\rangle$) to *agent_view*;
  reconsider stored and invalidated nogoods according to AAS0/AAS1/AAS2;
  check_agent_view;

when received *(***nogood**,A_j,$\neg N$*)* **do**
  add new assignments for already connected variables from $\neg N$ to *agent_view*;
  if *(((*A_i *knows* $\neg$*M)* $\wedge$ *(consequence*$_i$*(*$\neg N$*) covered by consequence*$_i$*(*$\neg$*M))* $\wedge$
      $\neg$*(better* $\neg N$ *than* $\neg$*M))*
    $\vee$ *invalid(*$\neg N$*))* **then**
    if *(AAS2) //(i.e. I do not want to discard* $\neg N$ **then**
      when $\langle x_k, s_k, h_k\rangle$*, where* x_k *is not connected, is contained in* $\neg N$
        send **add-link**(x_k) to modifiers(x_k);
        add $\langle x_k, s_k, h_k\rangle$ to *agent_view*;
      store $\neg N$;
    else
      when $\langle x_k, s_k, h_k\rangle$*, where* x_k *is not connected, is contained in* $\neg N$
        send **add-link**(x_k) to modifiers(x_k); add $\langle x_k, s_k, h_k\rangle$ to *agent_view*;
      put $\neg N$ in *nogood-list*;
  reconsider stored and invalidated nogoods according to AAS0/AAS1/AAS2;
  old_aggregate $\leftarrow$ *current_inst_aggregate*;
  check_agent_view;
  for all oa = ca*; (*oa$\in$old_aggregate*)*$\wedge$*(*ca$\in$current_inst_aggregate*)* **do**
    send (**ok?**,$\langle var(ca), \text{set}(ca), history(ca)|\text{i:}C^i_{var(ca)}|\rangle$) to A_j;

Algorithm 2: Responses of agent A_i for receiving messages in AAS.

Cartesian product of the assignments of A_i found in $\neg N$. The set of predecessors that can make proposals about a variable x_k is denoted by modifiers(x_k), the variable of an assignment a by var(a), its set of alternative valuations by set(a), and its history by history(a). The initial constraints of A_i are denoted by CSP(A_i).

The procedure for reacting to a message of a given type is denoted by the underlined name of the type. At the beginning, each agent A_i is in the state `Searching` where it tries to generate a current instantiation from C_{A_i}. At any time in the state `Searching`, an agent can transit into the state `Accepting` where it accepts **ok?** or **nogood** messages. These cause the agent to execute the procedures Ok, respectively Nogood (see Algorithm 2) which update the local search space (i.e the views, the nogoods lists and eventually the other structures of the employed technique used for searching satisfying Cartesian-products for the agent's problem) according to the content of the messages. When, in the state `Searching`, its C_{A_i} is found empty, the agent A_i announces a nogood (and removes the assignments of the agent target of the nogood). When, on the contrary, a local solution is found (i.e. a set of tuples can be extracted from C_{A_i}), the agent announces the instanti-

```
procedure check_agent_view do
    when agent_view and current_inst_aggregate are not consistent
        if no aggregate, V, in C_{A_i} is consistent with agent_view then
            backtrack;
        else
            select V ⊆ C_{A_i} where agent_view, CSP(A_i) and V are consistent;
            clean(current_inst_aggregate);
            for all a ∈ V do
                if (need_multicast(a)) then
                    x_k ← var(a); C^i_{x_k}++;
                    history(a) ← append(history(x_k),|i:C^i_{x_k}|);
                    send (ok?,⟨x_k, set(a), history(x_k)|i:C^i_{x_k}|⟩) to lower priority
                          agents in outgoing links(x_k);
                    current_inst_aggregate ← current_inst_aggregate ∪ a;
                else
3.1                 if (needed(a)) then
                        current_inst_aggregate ← current_inst_aggregate ∪ a;
procedure backtrack do
    ⋮
    for every V ∈ nogoods do
        select A_k, the lowest priority agent proposing assignments in V;
        send (nogood,A_i,V) to A_k;
        remove from agent_view all assignments proposed by A_k ;
        reconsider stored and invalidated explicit nogoods;
    check_agent_view;
```

Algorithm 3: Procedures of agent A_i in AAS.

ation by sending **ok?** messages to the concerned agents and transits into the state `Solution`. The current instantiation of the agent is known as long as it remains in the state `Solution`. The three algorithms differ by the actions undertaken in the procedures Ok and Nogood, respectively described in Algorithm 2.

The procedure Ok treats incoming **ok?** messages. The parameter of such a message is an aggregate. We say that a received assignment (x_j, set_j, h_j) is *obsolete* if the view of the receiving agent contains a newer assignment for x_j. The procedure Ok starts by filtering the obsolete assignments and then proceeds to updating the set C_{A_i} according to the remaining valid assignments. Suppose that one of these assignments offers a new possibility of valuation for an external variable x_j with respect to the current view. In AAS2 or AAS1 all the nogoods which do not take the new possibility into account will be *disabled*. In AAS1 this means that they will be removed. In AAS2 they will be marked and kept for an eventual further usage. In AAS0, if the nogood obtained by the relaxed inference rule contains such

a variable but does not take the new value into account, the conflict list will be reset, i.e. removing the nogood obtained by the relaxation rule. Resetting C_{A_i} means that all the tuples allowed by the current nogoods and view are introduced in C_{A_i}. In the end, the previous instantiation can be updated and renewed.

A new assignment of A_i is not of interest for successor agents if it does not modify in any way the previous newest assignment proposed by either A_i or its predecessors. Therefore we denote by need_multicast(a) a predicate telling when assignment a is of interest for one's successors. At line 3.1, needed(a) succeeds when a has the same set as some assignment b for var(a), found in *old_aggregate*, and b is still valid. Then, a inherits the history of b, history(a)←history(b). clean() removes the invalidated assignments from current_inst_aggregate.

The procedure Nogood treats incoming **nogood** messages. The argument, $\neg N$, of such a message is an explicit nogood. Let V be the view of the receiving agent. Suppose that there exists in N, respectively in V, an assignment a_1, respectively a_2 for the variable x_j such that a_1 is newer than a_2. We will say that the nogood gives a *new view* for the variable x_j. In this case, the agent has to update its view (and perform all the operations for the receipt of an **ok?** message). An explicit nogood is valid if it contains only valid assignments and concerns (i.e. invalidates) the current instantiation of the agent. If the received nogood is stored and if it contains variables that are unknown in the current view of the agent A_i, the procedure *Add links* will establish new links with all the agents $A_j, j<i$, for which these variables are local. The relation *better* between two nogoods can be defined by the user according to any heuristic. Typical heuristics, at each agent A_i, that have been used in the past are: *the nogoods that after removing A_i's assignments should be sent to higher priority agents are better* (in the experiments on ABTR [33]), and *nogoods with less variables are better*.

3.2 Solution Detection

Recall that a solution is a valuation of each variable such that all the constraints of each agent are satisfied.

Remark 1 *If a set of several solutions are proven at once, the returned solution is picked randomly among the proven ones.*

In the existing asynchronous search algorithms, solutions are only detected upon quiescence [2] . This state is usually recognized using a general purpose distributed mechanism [5]. We have noticed that in the particular case of asynchronous search, solutions can be detected before quiescence. This means that termination can be inferred earlier and that the number of messages required for termination detection

[2] end of **ok?**, **nogood** and **add-link** messages

can be reduced. We have introduced a system message (not considered in the notion of quiescence and not interfering with the search) called **accepted** which informs the sender of an **ok?** message of the acceptance of its proposal:

- **accepted** messages are sent from an agent to the lowest priority predecessor initially linked. Such a link is called acceptance link. If the agent has been an end-agent (agent initially having no incoming link), it sends an **accepted** to the *system agent*,
- an **accepted** message has as parameter a set of assignments obtained by computing the union of the ones in the current proposal of the sender with the parameters of the last **accepted** messages received from all its outgoing acceptance links and replacing all assignments for the same variable with a single assignment obtained by intersecting the corresponding sets of alternative valuations,
- an **accepted** message is sent by an agent only when its parameter is non empty (i.e does not contain empty domains), all the outgoing acceptance links have presented an **accepted** message, and the agent is in the state `Solution`,
- the agents checks whether to send **accepted** messages when they reach the state `Solution` or when they receive **accepted** messages.

accepted messages are FIFO ordered (e.g. using counters for each acceptance link).

Let D_i be the subgraph induced by the agents A_j with $j > i$ such that A_j can be reached from A_i along the directed acceptance links initialized by the *system agent*.

Proposition 1 *If a given agent A_i receives an* **accepted**(S_k) *message from all its outgoing acceptance links and if $\forall k, \bigcap S_k \neq \emptyset$, then A_i can infer that $\bigcap S_k$ is a solution for the partial CSP defined by the agents of D_i.*

Proof. D_i is a directed tree. If a given node A_j of this tree receives an **accepted**(S_k) message from all its k direct successors such that $\bigcap S_k \neq \emptyset$, it is obvious that the k successors have found an agreement on all the elements of $\bigcap S_k$. Following the definition of **accepted** messages, the agent A_j can in turn send an **accepted** through its incoming acceptance link and the process be repeated recursively. The proposition is therefore simply proved by induction on D_i. □

Corollary 1 *A correct solution is detected when the system agent receives an* **accepted**(S_i) *message from each initial end agent A_i and when $\bigcap_i S_i \neq \emptyset$.*

The termination algorithms can be classified in two families [25]: a) techniques based on a system agent launching termination-detection-rounds at regular time intervals, launching termination-detection-rounds by any agent that suspects termination, and b) having continuous processes that monitor the global state by following the active subsets of agents.

Lemma 1 *The highest number of messages for termination detection by any prob-*

ing mechanism with termination-detection-rounds is exponential for AAS.

Proof. AAS has an exponential time complexity (considering a worst case with no aggregations, and a single constraint forbidding everything and enforced by the last agent), therefore probing at any regular time intervals leads to an exponential number of termination-detection-rounds.

In AAS (or ABT), each agent suspects termination after the handling of any received message if it requires no sending of new messages. This happens an exponential number of times e.g. when the order of the agents is the DFS order of [7]. Therefore launching termination-detection-rounds by any agent that suspects termination also leads to an exponential number of rounds (in worst case).

Lemma 2 *The highest number of messages for termination detection with methods that monitor the global state by following the active subsets of agents is exponential for AAS.*

Proof. In AAS, the active agents cannot be localized in topologic groups as links can be added between each pair of agents. Given any temporary grouping, this can change an exponential number of times due to backtrack with nogoods.

The previous two lemmas state that any of the classical termination detection algorithms requires (in worst case) an exponential number of messages. Our solution detection algorithm is an adapted version of one of the best termination-detection techniques known in distributed systems (see the *Channel counting method* [25]). While techniques based on probing at vary large intervals of time may need less messages, they also delay the solution detection to multiples of the time intervals.

3.3 Message ordering

In asynchronous backtrack search (ABT), the messages must respect a FIFO channel order of delivery to ensure correct termination [41]. Our algorithm requires a stronger condition to hold since the channel for each variable is no longer a star but a graph. This means that several messages can arrive to the same agent, for changing the value of the same variable, through different paths of the graph. For example, in Figure 2 agent A_3 can receive messages concerning variable x_1 from both A_1 and A_2. An order must therefore be established between these kind of messages. In ABT it is sufficient to maintain a counter, for the emitter, and include its value within each message sent in order to obtain a FIFO order of delivery. In our algorithm, we include an additional such counter for each agent that modified a given domain in the message. The history of changes is built by associating a chain of pairs $|a : b|$ to each variable of a message (see Figure 2). Such a pair means that a change of the variable's domain was performed by the agent with index a when its counter for the corresponding variable had the value b. The local counters are reset

to 0 each time an incoming **ok?** changes the known history of the corresponding variable. It is incremented each time the agent proposes a change to the domain of that variable. To ensure correct termination, we use the following convention: The history of changes where the agent with the smaller index or the counter with the larger value occurs first is the most recent. If a history is the prefix of the other, then the longer one is more recent.

3.4 Correctness, Completeness, Termination

Proposition 2 *AAS0 is correct, complete, and terminates.*

Summary of the Proof. [3] Correctness is an immediate consequence of Corollary 1.

The proof that quiescence is reached is close to the one given for ABT in [41], using the additional knowledge that only **ok?** messages could remove nogoods of the agent with the least priority among those involved in the hypothetical infinite loop.

Quiescence can correspond to failure or solution, but it can correspond as well to deadlock. In order to prove that AAS0 cannot lead to deadlock, we shown that if the system reaches quiescence without having detected a solution or failure, a correct solution will be detected in finite time afterwards.

After receiving the last **ok?** message and performing the subsequent search, either each agent A_i has a final instantiation that is consistent with its view, or failure is detected.

At quiescence, the view of each agent A_i consists of the intersection of the instantiations of all instantiated agents $A_j, j < i$, for the variables it is interested in. This intersection corresponds, for each variable, to the newest received assignment.

From the previous steps it follows that in a finite time after quiescence, the intersection of the instantiations of all agents $A_j, j \leq i$ is nonempty and consistent with all the constraints in the agents $A_j, j \leq i$, for all i. Consequently, the last **accepted** messages sent by an agent to its predecessors are such that at receiver, $\bigcap S_k \neq \emptyset$. This is true for all the agents, which means that the **accepted** messages needed for solution detection will reach the system agent.

For completeness, we have proven that failure cannot be announced by AAS0 when a solution exists. A nogood is a redundant constraint with respect to the CSP to solve. Since all the additional nogoods are generated by logical inference, an empty nogood cannot be inferred when a solution exists. □

[3] The detailed proof is available in [38].

Proposition 3 *AAS1 and AAS2 are correct, complete and terminate.*

Proof. Immediate consequence of the fact that AAS1 and AAS2 only add redundant constraints to AAS0 (under the form of nogoods) and of Proposition 2. □

The optimisations to local processing proposed in [22,12] (e.g. forward checking of labels) can also be applied. However, we did not make any effort at this level. We mention that an integration of approximation techniques into ABT has also been described in [16]. A synchronous approach close to AAS2 has been proposed for solving design problems (see [8]).

4 Asynchronous consistency maintenance

Maintaining consistency through constraint propagation is one of the most important techniques in centralized constraint programming. We now present a new distributed algorithm, called Maintaining Hierarchical Distributed Consistency (MHDC), that incorporates distributed consistency into asynchronous backtracking. One of its main characteristics is to consider consistency maintenance as a hierarchical task. Enforcing the hierarchies of consistency and performing search can then be done with a high degree of asynchronism. This gives the agents more flexibility and freedom in the way they can contribute to search, and increases parallelism. As expected, the experimental results show that substantial gains in computational power can result from combining distributed search and distributed local consistency algorithms.

4.1 Distributed Bound-Consistency

Bound-consistency is a simple form of arc-consistency that for each variable maintains only a pair of outer bounds that enclose the consistent values when the domain is ordered. Centralized algorithms for maintaining bound-consistency on discrete and continuous problems are presented in [32,21].

For integrating distributed bound consistency with Asynchronous Aggregation Search (AAS), we now present an algorithm called *Distributed Hierarchical Consistency* (DHC), that builds on AC3 [24] and is similar to DAC [42,27]. In DHC, we consider a DisCSP with m agents, and call the constraints and variables owned by an agent A_i its *local* CSP $CSP(A_i)$. $Var(A_i)$ denote the variables in $CSP(A_i)$.

Each agent A_i maintains a stack S^i of labelings for its local variables. The stack's entry $S^i(k)$ of level k, with k varying from 0 to i, contains the labeling based on the known assignments of all agents $A_t, t \leq k$. By construction, the domain allowed by

$S^i(k)$ for a variable v is contained in the domain for v in $S^i(k-1)$ (see Figure 3). The reason for maintaining a stack of labels is that this allows easily adjusting the labeling when a variable of agent A_j changes value: it will require discarding labels of level j and higher. We define:

Definition 6 (Label) *A label generated by the agent A_i at consistency level k for the variable x, and denoted by* $\texttt{label}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$*, is a triplet (x,* $\texttt{r}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$*,* $\texttt{context}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$*). *$\texttt{r}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$ *is a range of values giving the label of level k for the variable x.* $\texttt{context}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$ *is the context in which* $\texttt{r}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$ *is generated.*

We denote by $\texttt{DHC}^{\texttt{i}}_{\texttt{k}}$ a process executed by the agent A_i and enforcing distributed bound consistency on the labelings $S^i(k)$. The context is an aggregate which contains all the assignments involved in the computation of $\texttt{r}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$ by $\texttt{DHC}^{\texttt{i}}_{\texttt{k}}$. It is used for updating the local information of the receiving agent, checking the validity of the propagated nogood and inferring nogoods after search or a consistency maintenance process has detected domain wipe out. It will be described in more details later. The context, $\texttt{context}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$, of a label (x, $\texttt{r}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$, $\texttt{context}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$) is simply the subset of the view of A_i used for computing $\texttt{r}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$ using a $\texttt{DHC}^{\texttt{i}}_{\texttt{k}}$ process. It is used to explain the result. Explained nogoods have been used so far in dynamic arc-consistency [3], and in maintainance of consistency in dynamic backtracking [18].

To compute the context, the local consistency algorithm employed to revise the local constraints of an agent is extended by keeping a separate explanation for each variable. This explanation is initialized with the current context of the label for that variable at the corresponding level. Each time a revision of a constraint between x_i and x_j removes a value of the variable x_i, the explanation of x_j is merged into the explanation of x_i.

We note that with bound consistency this can be refined to keep a separate explanation for each bound (two explanations for a variable/label). In this case, on the reduction of a bound b of x_i with a set of values D_b one adds to the explanation of b only the explanations of the bound(s) of x_j removing any of the initials supports of D_b. Similarly, with arc consistency one can maintain a separate explanation for the removal of each value. In these cases, on the removal of a value of x_i, only the explanations of its initial supports need to be merged together to explain the removal.

Recall that an assignment (x_j, r_j, h_j) received by an agent A_i is obsolete if the view of A_i contains a newer assignment for x_j. Similarly, we define:

Definition 7 (Valid label) *A label sent in a* $\texttt{propagate}()$ *message is* valid *if and only if its context contains no obsolete assignment.*

Definition 8 (Labeling) *A labeling generated by the agent A_i at consistency level k, and denoted* $\texttt{labeling}^{\texttt{i}}_{\texttt{k}}$*, is a list of labels* $\texttt{label}^{\texttt{i}}_{\texttt{k}}(\texttt{x})$.

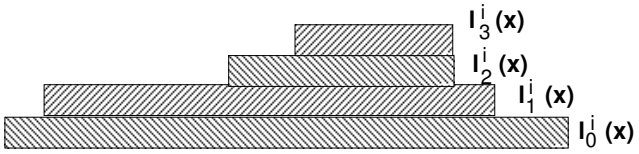


Fig. 3. The local labeling stack maintained by agent A_i for variable x.

We define a label of level k as a label computed by (bound) consistency using the proposals of agents $A_t, t \leq k$ (stored as shown in Figure 3).

4.2 Maintaining Distributed Bound-Consistency

In a centralized framework, consistency algorithms are run whenever a variable is assigned a value. This could be implemented in a distributed environment as a synchronous mechanism, which we call *synchronous MDC*. However, it lacks parallelism, requires complex termination tests for detecting the convergence of each local propagation, and has all the previously mentioned drawbacks of the synchronism. We therefore propose to maintain consistency in a parallel and asynchronous process. To achieve this goal, we consider bound-consistency maintenance as a hierarchical task where each agent runs up to m separate processes that each maintain a label of a different level.

$\texttt{DHC}_0^i$ is the particular DHC process that can be launched by agent A_i when it has no information about the instantiation of other agents. $\texttt{DHC}_0^i$ corresponds to the elementary level where the original global CSP is brought bound-consistent. $\texttt{DHC}_k^i$, with $i>k>0$, is the DHC process of level k that can be launched by A_i.

The agents communicate by sending **propagate** messages. The argument of a **propagate** message is a list of labels. We denote by $\texttt{propagate}_k^i()$ the message sent by the agent A_i at level k. A $\texttt{propagate}_k^i()$ message can only be sent by an agent A_i with $i > k$ to agents A_j with $j \geq k$. It informs the concerned agents about nogoods (domain reductions) inferred by a $\texttt{DHC}_k^i$ process. The agent A_i can run a $\texttt{DHC}_k^i$ process, $k>0$, as soon as it has received an **ok?** message from the agent A_k or a $\texttt{propagate}_k^j()$ message from any agent A_j with $j > k$.

The labels, $\{(x_k, r_k, c_k) | k \in Var(A_i) \cap Var(A_j)\}$, sent as an argument by A_j to A_i only contain variables x_k whose r_k has just been modified by A_j and that are local variables common to A_i and A_j.

Each agent A_i starts by enforcing bound-consistency on its own local CSP, $CSP(A_i)$. This computation initializes the labels of $\texttt{DHC}_0^i$ which are then sent to all the agents interested in the same variables. When A_i receives a new label (`var`,`new-dom`) via a **propagate** message, it combines this label with the corresponding entry (`var`,`old-dom`) of S^i. Such a combination is done by do-

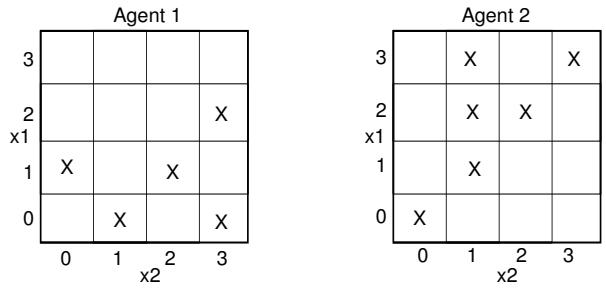
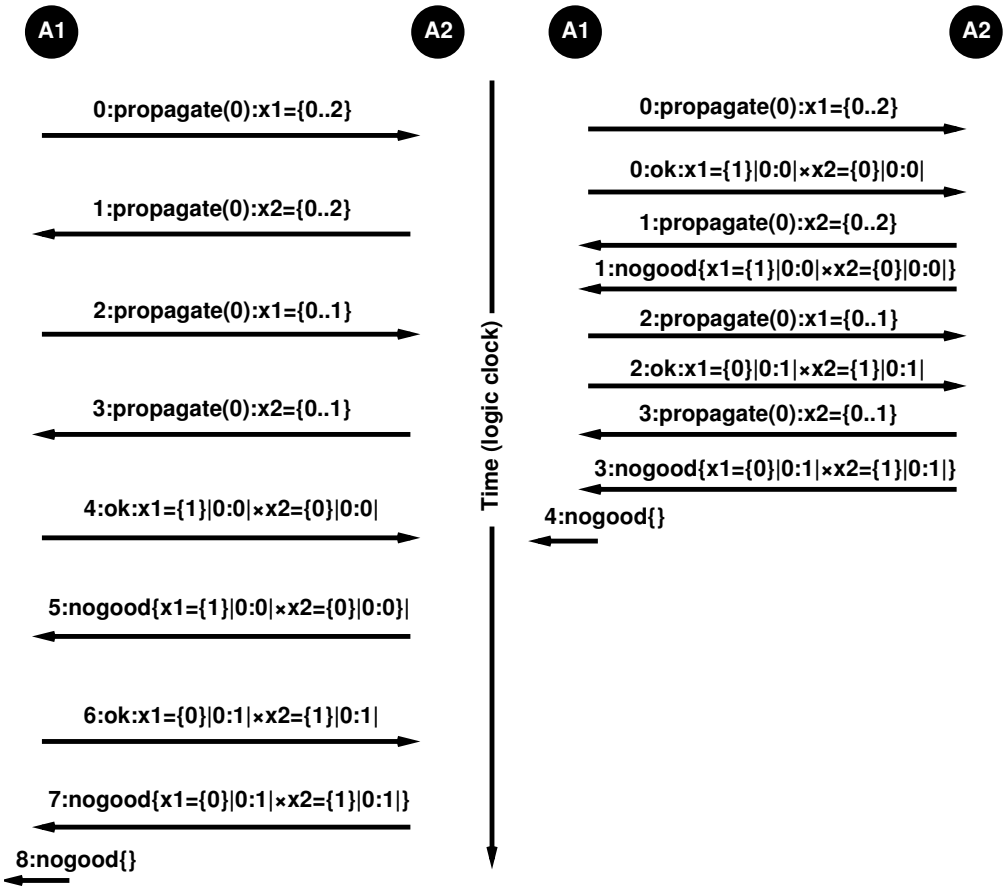


Fig. 4. A simple problem where running search and consistency maintenance asynchronously helps. Crossed combinations are feasible.

main intersection and if it results in a reduction of `old-dom`, all the constraints of $CSP(A_i)$ involving `var` are reinserted in a revision queue, used for re-enforcing bound-consistency on $CSP(A_i)$. The resulting modified labels, if any, are further broadcast to the concerned agents. The process has converged when no further messages are generated. Following the complexity of AC3, the maximum number of generated messages is m^2nd and the maximum length of a chain of sequential messages accountable by logical clocks [20] is nd (n:number of variables, d:maximum domain size, m:number of agents). There can be m^2 simultaneous messages.

In a synchronous algorithm *synchronous MDC*, each agent A_i would only launch the process $\texttt{DHC}^{\texttt{i}}_{\texttt{i}}$ when the $\texttt{DHC}^{\texttt{k}}_{\texttt{j}}$ have reached convergence for all k and for all $j < i$. However, we can enable the $\texttt{DHC}^{\texttt{i}}_{\texttt{j}}$'s, with $0 \leq j \leq i$, to run asynchronously in A_i for different j's, together with the asynchronous backtracking of AAS. This is simpler and can lead to much faster execution, as shown by the following example with two agents handling a single constraint each. The constraints involve the same variables x_1 and x_2 and are shown in Figure 4.

Figure 5 compares the traces of message passing for this example between the synchronous and asynchronous variants of maintaining distributed consistency. The basic algorithm behaves as AAS except that, in addition to the **ok?** and **nogood** messages, it also sends **propagate** messages which inform the agents about domain reductions computed by the DHC processes. In Figure 5, **propagate(k):x=**$\{$**a..b**$\}$ is a message informing the receiver that the domain reduction $[a, b]$ has been computed by a process DHC of level k for the variable x. The cost refers to the length of the longest chain of sequential messages encountered before this message could be generated (also known as the logic clock of the message [20]). In *synchronous MDC* four sequential messages are exchanged before the convergence of DHC. Still, the same amount of search remains to be done. Figure 5b shows that the whole search space can be exhausted with the same number of messages by AAS performed in parallel with DHC in asynchronous MHDC. In both cases, the messages for termination detection are not taken into account. The *synchronous MDC* creates longer causal chains of messages, since it requires to wait for the detection of the termination of distributed consistencies.



a) synchronous consistency maintenance b) asynchronous consistency maintenance

Fig. 5. Traces of messages for the example in Figure 4. Both solutions require the same number of messages but they are exchanged simultaneously in the asynchronous version.

4.3 The MHDC algorithm

We now give the MHDC algorithm and its interaction with AAS. The notions of *assignment*, *aggregate* and *explicit nogoods* are defined as in AAS. The *current solution space* of A_i, denoted as C_{A_i}, is described by the local constraints, the entry $S^i(i)$, a list of explicit nogoods, and a *view*.

A tuple is *contained* in the current local search space of agent A_i if it satisfies the local constraints and is not contained in the explicit or entailed nogoods of C_{A_i}. The *current instantiation* of an agent A_i is a set of assignments such that all tuples it allows are contained in C_{A_i}. Figure 3 schematizes the set of labels maintained by agents for each variable.

Definition 9 (Nogood entailed by consistency) *Let T be the set of tuples disabled from the original solution space by the labeling L of $S^i(k)$. We say that the nogood $L \rightarrow \neg T$ is entailed by consistency.*

We define a set $\{\texttt{labeling}^{\texttt{i}}_{\texttt{k}} \rightarrow \neg T^i_k\}$ with $\texttt{labeling}^{\texttt{i}}_{\texttt{k}} \in S^i(k)$ and $k \in \{0, \ldots, i\}$ that contains all the nogoods entailed by consistency for a given agent A_i. As with the merging of explicit nogoods in AAS0, several labels of the same variable and level in MHDC can be merged into a single label by applying the relaxation rule on the nogoods they entail.

4.3.1 The consistency maintenance procedure

A new type of messages is introduced, namely **propagate** messages. They are used by an agent A_i to send labels of level k to all interested agents $A_i, i{\geq}k$. The agents A_i use the most recent proposals of the agents $A_j, j{\leq}k$ when they compute consistent labels. A_i may receive valid consistency nogoods of level k with aggregate-sets for the variables *vars*, *vars* not in vars(A_i). As shown in Algorithm 4, A_i must then send **add-link** messages to all agents $A_{k'}, k'{\leq}k$ not yet linked to A_i for all their variables in *vars*.

To keep track of the order in which labels are generated by each agent, each agent A_i stores for each variable x_u that it owns a counter $c^t_{x_u}(i)$ incremented as shown in Algorithm 4 at line 4.1, whose value tags each sent label of x_u. The labels of each x_v in $S^i(k)$ are denoted by $cn^k_{x_v}(i)$.

The problem of level k on which A_i runs a local consistency algorithm with explanations is: $P_i(k) := \mathrm{CSP}(A_i) \cup (\cup_x cn^k_x(i)) \cup \mathrm{NV}_i(V^i_k) \cup \mathrm{CL}^i_k$. Here $\mathrm{NV}_i(V)$ is the nogood entailed to A_i by the view V. CL^i_k is the set of all nogoods known by A_i and having the form $V \rightarrow \neg T$ where V is a set of aggregates proposed by agents with positions lower or equal to k and T is a set of tuples in $\mathrm{CSP}(A_i)$. CL^i_k may contain the CL of A_i (introduced with AAS0). An agent can manage to maintain one CL for each instantiation level and the space requirements do no change.

A consistency nogood (e.g. obtained by composing several consistency nogoods, and where the label becomes an empty set), is an explicit nogood, and in this case, the conflict set is composed of the variables in the context.

In this version we choose to not enforce all the levels of consistency all the time at an agent A_i, but only those up to a value cL_i it stores. cL_i corresponds to the first level where an explicit nogood is found. Other small changes to the other procedures of AAS consist in calling **maintain_consistence** whenever the views change.

A theoretically remarkable version called DMAC and analysed in the following maintains all the last labels for each level from all agents. Therefore, each agent has to store a structure of the type $cn^k_{x_v}(i)$ for each other participant. However, DMAC is slightly less efficient in our experiments when compared to MHDC.

when received(propagate,A_j,k,$c^k_{x_v}(j)$,$V\rightarrow(x_v\notin l)$) **do**
  when have higher tag $c^k_{x_v}(j,i)\geq c^k_{x_v}(j)$ **then** return;
  $c^k_{x_v}(j,i) \leftarrow c^k_{x_v}(j)$; **when** any $\langle x, s, h\rangle$ in V is invalid (old h) **then** return;
  when $\langle x_u, s_u, h_{x_u}\rangle$, where x_u is not connected, is contained in V
    send **add-link** to agents proposing assignments for x_u;
    add $\langle x_u, s_u, h_{x_u}\rangle$ to **agent_view**;
  add other new assignments in V to **agent_view**; eliminate invalidated nogoods;
  merge $cn^k_{x_v}(i)$ with {V$\rightarrow$($x_v\notin$l)} using the inference rule proposed in AAS0;
  maintain_consistency(minimal level that is modified);
  check_agent_view; //only satisfies consistency nogoods of levels t, t<cL$_i$;

procedure maintain_consistency(*minT*) **do**
  if (minT $>$ cL$_i$) **then** return;
  for (t$\leftarrow$minT; t$\leq$i; t++)
    new-cns $\leftarrow$ consistency nogoods for all vars(A_i) after local consistency on $P_i(t)$;
    when (domain wipe out by computing explicit nogoods *nogoods*)
      for every $V \in$ *nogoods*;
        if V is an empty nogood, **then** anounce failure and terminate;
        select $\langle x_j, s_j, h_{x_j}\rangle$ where h_{x_j} is generated by the agent A_u with the lowest priority among those generating assignments in V;
        send (**nogood**,A_i,V) to A_u; eliminate invalidated explicit nogoods;
      cL$_i$ $\leftarrow$t;
      break;
    forall *new-cn* $\leftarrow$ consistency nogood for any variable x_u in *new-cns*
      when *new-cn* shrinks label of x_u (obtained from $\cup_{k\leq t} cn^k_{x_u}(i)$)
4.1 $cn^t_{x_u}(i) \leftarrow$ *new-cn*; $c^t_{x_u}(i)$++;
        send (**propagate**,A_i,t,$c^t_{x_u}$,*new-cn*) to agents $A_j, j\geq t, x_u\in$**vars**($\mathbf{A_j}$);

Algorithm 4: Procedures of agent A_i with position i in MHDC.

4.4 DMAC

MHDC builds on AAS which is proven to be correct and complete and terminates. MHDC is AAS with the inference and transmission of the additional nogoods generated by bound-consistency maintenance. As argued for AAS1 and AAS2, the use of additional nogoods in the local decisions maintains the correctness, termination and completeness properties.

To insure that the strength of the consistency maintained in MHDC and *synchronous MDC* are strictly equivalent, agents using MHDC need to maintain all the last valid nogoods entailed by consistency that they have received for each level and variable from each agent. That version is called DMAC.

Proposition 4 *The minimum space an agent needs with DMAC for ensuring maintenance of the highest degree of consistency achievable with arc consistency is*

when init do
  $cost[k] \leftarrow 0$ for all $k \in [0..K]$;

when received*(message,new_cost)* **do**
  $cost[0] \leftarrow max(cost[0], new\_cost[0] + 1)$;
  $cost[k] \leftarrow max(cost[k], new\_cost[k] + 5(k-1))$ for all $k \in [1..K]$;
  call procedure **when received**(message);

when local_constraint_check do
  $cost[k] \leftarrow cost[k] + 1$ for all $k \in [1..K]$;

procedure send*(message)* **do**
  send(message, $cost$);

Algorithm 5: Procedures of an agent for measuring performance on the MELY platform.

O($nm^2(n+d)$) where n is the number of variables, m the number of agents and d the domain size. With bound consistency, the required space is O($(nm)^2$).

Proof. The agents need to maintain at most m levels, each of them dealing with maximum n variables, for each of them having at most m last consistency nogoods. Each consistency nogood refers at most n assignments in premise and stores at most d values in label. The stack of labels requires therefore O($nm^2(n+d)$).

5 Experiments

Our experiments were run with an implementation using a language based on state machines that we developed, where each of the local consistency processes at different levels in each agent are run concurrently with different priorities. We ran two sets of experiments with random binary CSPs: the first on problems of more or less constant difficulty with the purpose of verifying the gains achieved by aggregation in an asynchronous setting. The second set of experiments varied the difficulty of CSP in order to test the influence of consistency on the number of messages.

5.1 The measurements taken by our experimentation platform

For each solving instance, our platform measures a vector $cost[k], k \in [0..K]$ of costs (aiming to obtain similar graphs with the ones based on rounds, in [40]). However, we have a real implementation and not a simulator. We therefore computed equivalent constraint checks for messages using the framework of logic clocks [20], which with null costs for local events is also known as *the longest chain of causal (sequential) messages* [28]. The obtained measurement procedure implemented by the used platform (MELY) in an agent is given in Algorithm 5. As an exception, **ac-**

cepted messages are considered to belong to the infrastructure, and do not trigger the procedures in Algorithm 5. However, at the end of the computation the platform reports the cost vector built by the system agent from received **accepted** messages (see Section 3.2), according to Algorithm 5.

The first four results graphs are built by using the values of $cost[k], k \in [1..K], K = 9$. For each abscisae $x \in \{5x | x \in [0..K-1]\}$ we plot the value $cost[1 + (x/5)]$, representing the equivalent constraint checks when a message is considered to cost as much as x constraint checks. This results in a curve that has a slope proportional to $cost[0]$, reason for which we did not draw $cost[0]$ separately.

Note that the value for $cost[0]$ is the length of the longest chain of causal messages. Lamport's logic clock measurement as well as the usage of $cost[1]$ (the intersection with the $0y$ axes) were recently 'independently reinvented' and recommended under the name of concurrent constraint check (CCC) and are used as such in several works. However, even if we employed these techniques in our work on AAS [31] in order to be comparable with the work in [40], we do not particularly recommend any of these measurement values as sufficient. Our experience has shown that in real settings, only the size of the longest causal chain of messages (measured by $cost[0]$) was somewhat proportional with the execution time in seconds (for different algorithms, running on the Internet). Therefore this is the only measure that we used in evaluating our most recent work, like MHDC.

5.2 Experiments to quantify aggregation behavior in AAS

AAS0, 1 and 2 have been evaluated on randomly generated problems with 15 and 20 agents, situated on distinct computers on a LAN. The constraints have been distributed to the agents in the same way that they would have been enforced in ABT so that they can be compared. As a consequence, the number of variables equals the number of agents. The size of domains is of 5 values and the problems are generated near the peak of difficulty [6] (for 15 agents) with a density of 30% and a tightness of constraints of 45%. Each test is averaged over 50 instances.

An evaluation of distributed algorithms has to take into account both the cost of messages and the cost of local computation. The relative cost of a message vs. a constraint check may be very low if agents are running in different threads on the same computer, or very high if they are communicating through the internet. To show how the algorithms will behave in different environments, it is customary in distributed algorithms to show the cost for different ratios between the cost of a message and the cost of a local operation, in this case a constraint check. As in [40], the logic cost of a local event (constraint check) is fixed to 1 and the logic cost of a message varies between 0 (agents are threads on a computer) and 40. Constraint checks are measured by the highest logic clock of the computation [20],

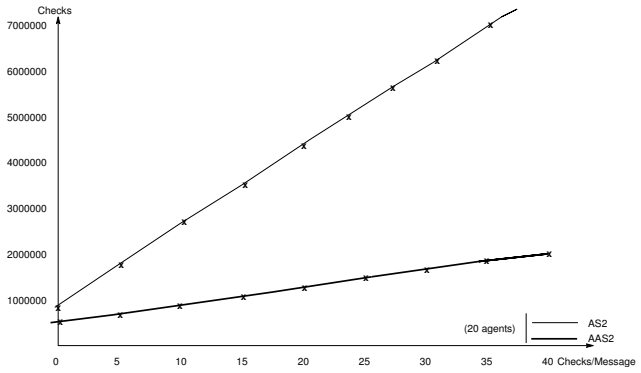


Fig. 6. The average number of checks on randomly generated problems. Abscissae select the relative time needed for sending a message divided by the time for a constraint check.

i.e. the longest sequence of interdependent steps in the computation. To evaluate aggregation, we implemented three versions of ABT that have similar behavior to AAS0/AAS1/AAS2 in treating nogoods, by simply disabling aggregation in the implementation of AAS. The three obtained versions are denoted AS0, AS1, respectively AS2.

We show the results as graphs that plot this measure against the cost ratio of messages/constraint checks:

- the slope of the curves approximatively corresponds to the number of messages. For example, in Figure 6, at a relative cost of 20 checks/message, AS2 requires about $(430'000 - 80'000)/20 = 17'500$ messages, while AAS2 requires about $(110'000 - 60'000)/20 = 2'500$ messages.
- the intercept with the y-axis gives the number of constraint checks without considering messages. For example, in Figure 6, AS2 requires about 80'000 checks and AAS2 about 60'000 checks.

The fact that the curves are almost straight lines shows that the number of messages does not vary much with the speed at which messages are delivered.

AAS2 versus AS2. AAS2 performs slightly better than its version without aggregation, AS2 (see Figure 6). There are specific cases where AS2 performs better for finding the first solution. However, for discovering that no solution exists AAS2 always performs better than AS2 since the whole search space needs to be expanded. AAS2 also reduces the longest sequence of messages, as well as the number of nogoods stored.

AAS0 versus AAS1. AAS1 needs more messages than AAS2, and AAS0 even more (see Figure 7). However, they do not present memory problems.

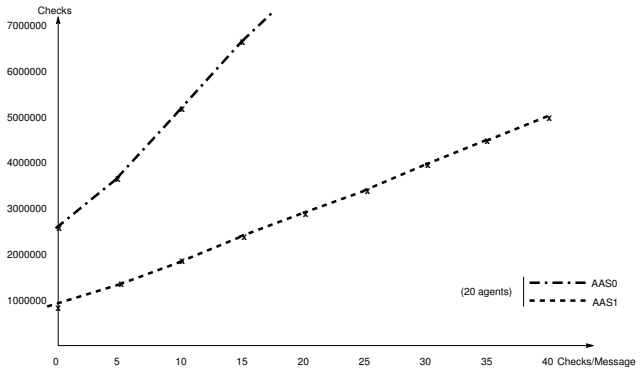


Fig. 7. The average number of checks on randomly generated problems. Abscissae select the relative time needed for sending a message divided by the time for a constraint check.

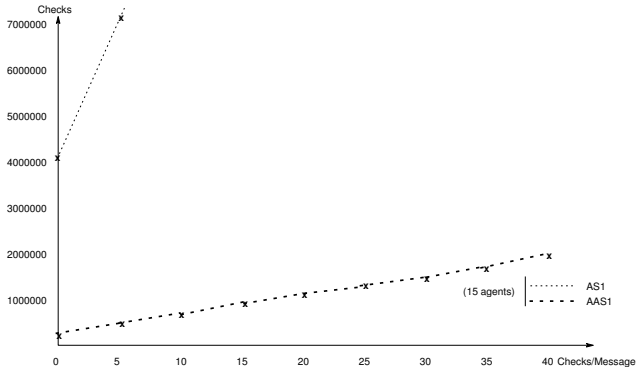


Fig. 8. The average number of checks on randomly generated problems. Abscissae select the relative time needed for sending a message divided by the time for a constraint check.

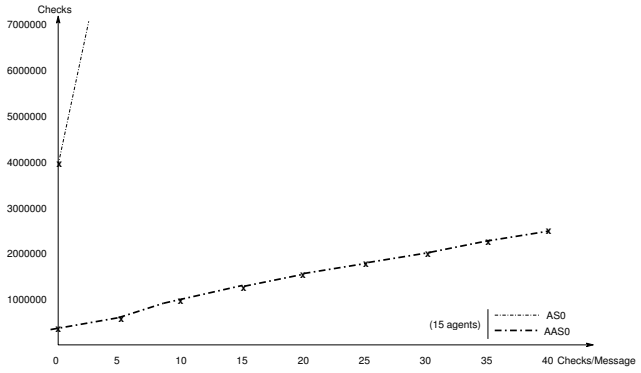


Fig. 9. The average number of checks on randomly generated problems. Abscissae select the relative time needed for sending a message divided by the time for a constraint check.

AAS0 vs. AS0 and AAS1 vs. AS1. We have tested the usefulness of the aggregation by comparing AAS0 and AAS1 against our versions of AS where the equivalent nogood policies are used (AS0 respectively AS1). It saves 95% of the messages. If space is available, it seems useful to store some additional nogoods.

These experiments reveal a close connection between the storage of nogoods and the usefulness of aggregation. Beside its intrinsic gain, aggregates can provide an

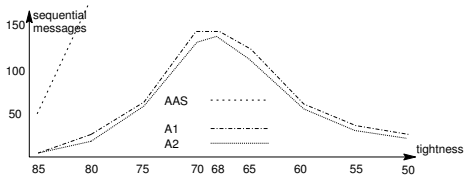


Fig. 10. Results averaged over 500 problems per point.

opportunity to replace the efficiency of some nogoods with a bounded space alternative. During traces, we noticed that in versions with full nogood storage, nogoods for single proposals can disable a large part of an aggregate, reducing the usefulness of the last one. The need to remove nogoods increases the importance of adding aggregation to this type of discrete problems. As shown later, due to the treatment of intervals, aggregation is necessary for problems with large or continuous domains.

5.3 Experiments to quantify behavior of consistency in MHDC

In this second experiment, we vary the tightness of the problems with the goal of comparing the gain in complexity achieved through consistency techniques as implemented in MHDC.

Since the complexity of random centralized problems is well studied, we have chosen to generate our distributed problems starting from centralized problems with known complexities. More precisely, we generate random centralized problems (CP) with n variables. We then circularly distribute the constraints to each agent, one constraint at a time.

In order to choose the constraint to be attributed to an agent, we pick randomly a constraint from the remaining ones of CP. We allow a number $\tau_1 n$ of trials to pick one constraint that had both its variables in the current agent. On failure we allow a number $\tau_2 n$ trials for picking one constraint that had at least one of its variables in the current agent. If this second chance has failed, then we simply pick a constraint at random. Once a constraint has been chosen for the current agent, we add it to the local CSP of that agent and remove it from CP. Thus we have two new parameters for the complexity of the obtained distributed problems, namely τ_1 and τ_2. These parameters quantify the effort for clustering the variables within agents. High values for τ_1 and τ_2 lead to groups of constraints among a limited number of variables, (choosing cliques in each agent). Instead, low values for τ_1 and τ_2 lead to agents being interested in most of the variables. In our experiments we used $2\tau_1 = \tau_2 = 1$.

The experiments show that the overall performance of asynchronous search with consistency maintenance is significantly improved compared to that of asynchronous search that does not maintain consistency.

The techniques used in our experimental evaluation maintains bound-consistency. In each agent, computation at lower levels is given priority over computations at higher levels. We generated randomly problems with 15 variables of 8 values and graph density of 20%. Their constraints were randomly distributed in 20 subproblems for 20 agents. Figure 10 shows their behavior for variable tightness (percentage of forbidden tuples in constraints), averaged over 500 problems per point. We tested two versions of MHDC, A1 and A2. A1 asynchronously maintains bound consistency at all levels. A2 is a relaxation where agents only compute consistency at levels where they receive new labels or assignments, not after reduction inheritance between levels. In both cases, the performance of MHDC is significantly improved compared to that of AAS, whose curve leaves the plot area (Figure 10) already at a tightness of 80. Even for the easy points where AAS requires less than 2000 sequential messages, MHDC proved to be more than 10 times better in average. A2 was slightly better than A1 on average (excepting at tightness 85%), perhaps due to the fact that inheriting levels often will be soon modified by updated assignments. In these experiments we have stored only the minimal number of nogoods. The nogoods are the main gain of parallelism in asynchronous distributed search. Storing additional nogoods was shown for AAS to strongly improve performance of asynchronous search. As future research topic, we foresee the study of new nogood storing heuristics [15,36].

6 Related Work

The first complete asynchronous search algorithm for DisCSPs is Asynchronous Backtracking (ABT) [40]. The approach in [40] considers that agents maintain distinct variables. Nogood removal was discussed in [15]. Other definitions of DisCSPs have considered the case where the interest on constraints is distributed among agents [42,35,13,9]. [35] proposes algorithms that fit the structure of a real problem (the nurse transportation problem). The Asynchronous Aggregation Search (AAS) family of protocols actually extends ABT to the case where the same variable can be instantiated by several agents (e.g. at different levels of abstraction [30]). An agent may also not know all constraint predicates relevant to its variables. The privacy achieved for constraints is further analyzed in [37,29]. An extension solving problems where some constraints may not be known to anybody is introduced in [34]. AAS offers the possibility to aggregate several branches of the search. An aggregation technique for DisCSPs was then presented in [26] and allows for simple understanding of privacy/efficiency mechanisms, also discussed in [10]. The use of abstractions not only improves on efficiency but especially on privacy since the agents need to reveal less details. A general polynomial space reordering protocol and several heuristics (e.g. weak commitment-like [39]) are discussed in [33]. In [4] it is explained how **add-link** messages can be avoided. Several algorithms for achieving distributed arc consistency are presented in [19,42,2].

7 Conclusions

Asynchronous search algorithms for distributed CSP so far have taken little advantage of the techniques that have led to highly efficient centralized algorithms for CSP. We have shown how two well-known techniques, value aggregation and arc consistency, can be used to very significantly improve the performance of distributed asynchronous search algorithms.

We have presented the AAS algorithm which uses value aggregation to significantly reduce the number of messages that have to be sent during search. Empirical evaluation shows that execution time is significantly reduced, particularly when the time required for sending messages is high relative to the time required per constraint check.

We have then presented MHDC, a new distributed search technique which allows maintaining distributed consistency with a high degree of parallelism and without resorting to intermediate termination detection. The preliminary evaluation has been done with a version based on AAS0 which, consequently, maintains a minimal number of nogoods. The experiments have shown that the overall performance of MHDC is significantly improved compared to that of AAS. MHDC has much potential in practice. It accommodates a higher number of agents than AAS, requires a bounded local space, reduces the number of messages needed for termination detection, and improves parallelization compared to *synchronous MDC*. MHDC fully exploits the aggregation capability of AAS. If built on AAS0, MHDC guarantees polynomial space complexity.

In other work, we have also experimented with variable reordering, but have not been able to achieve any significant efficiency gains. This is in strong contrast to centralized settings where dynamic variable ordering is crucial to achieving efficient search performance. We speculate that with known heuristics this cannot yet be achieved in the distributed case because of the large amount of asynchronous search effort that has to be discarded every time variables are reordered and of the size of the problems that can be addressed in experiments with the state of the art techniques.

8 Acknowledgments

Most of the MELY search agent platform (used for experiments) was implemented with the help of Michel Galley. We have to thank many people for comments, help, and encouragement, among which Djamila Sam-Haroud and Markus Zanker.

References

[1] A. Armstrong and E. F. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proceedings of 15th IJCAI*, 1997.

[2] B. Baudot and Y. Deville. Analysis of distributed arc-consistency algorithms. Technical Report RR-97-07, U. Catholique Louvain, 1997.

[3] C. Bessière. Arc-consistency in dynamic constraint satisfaction problems. In *AAAI'91*, 1991.

[4] C. Bessière, A. Maestre, and P. Meseguer. Distributed dynamic backtracking. In *Proc. IJCAI DCR Workshop*, pages 9–16, 2001.

[5] K.-M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *TOCS'85*, 1(3):63–75, 1985.

[6] P. Cheesman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on AI*, 1991.

[7] Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of IJCAI 1991*, pages 318–324, 1991.

[8] J. G. D'Ambrosio, T. Darr, and W. P. Birmingham. Hierarchical concurrent engineering in a multiagent framework. *Concurrent Engineering: Research and Applications (CERA) - An International Journal*, 4(1):47–57, March 1996.

[9] J. Denzinger. Distributed knowledge based search. IJCAI tutorial notes (MA2), 2001.

[10] E.C. Freuder, M. Minca, and R.J. Wallace. Privacy/efficiency tradeoffs in distributed meeting scheduling by constraint-based agents. In *Proc. IJCAI DCR*, pages 63–72, 2001.

[11] M. Ginsberg and D. McAllester. Gsat and dynamic backtracking. In J.Doyle, editor, *Proceedings of the 4th IC on PKRR*, pages 226–237. KR, 1994.

[12] Youssef Hamadi. *Traitement des problèmes de satisfaction de contraintes distribués*. PhD thesis, Université Montpellier II, Juillet 1999.

[13] M. Hannebauer. On proving properties of concurrent algorithms for distributed CSPs. In *Proceedings of the CP Workshop on Distributed Constraint Satisfaction*, 2000.

[14] A. Haselböck. Exploiting interchangeabilities in constraint satisfaction problems. In *Proceedings of IJCAI'93*, pages 282–287, 1993.

[15] W. Havens. Nogood caching for multiagent backtrack search. In *AAAI Constraints and Agents Workshop (W5)*, 1997.

[16] K. Hirayama and M. Yokoo. An approach to over-constrained distributed constraint satisfaction problems: Distributed hierarchical constraint satisfaction. In *Proceedings of ICMAS-2000*, 2000.

[17] P. D. Hubbe and E. C. Freuder. An efficient cross product representation of the constraint satisfaction problem search space. In *Proc. of AAAI*, pages 421–427, July 1992.

[18] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *CP'2000*, Singapore, 2000. Springer.

[19] S. Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Intelligence*, 45(3):275–286, October 1990.

[20] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[21] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *IJCAI*, pages 245–250, Acapulco, Mexico, August 2003.

[22] Q.Y. Luo, P.G. Hendry, and J.T. Buchanan. A hybrid algorithm for distributed constraint satisfaction problems. Technical Report KEG-3-92, University of Strathclyde, 1992. 1 Sept. 1991.

[23] Q.Y. Luo, P.G. Hendry, and J.T. Buchanan. Comparison of different approaches for solving distributed constraint satisfaction problems. Technical Report No. RR-93-74, University of Strathclyde, 1993.

[24] A. K. Mackworth. Consistency in networks of relations. *Artificial Intellegence*, 8(1):99–118, 1977.

[25] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[26] P. Meseguer and M. Jiménez. Distributed forward checking. In *CP'2000 Distributed Constraint Satisfaction Workshop*, 2000.

[27] Eric Monfroy and Jean-Hugues Rety. Chaotic iteration for distributed constraint propagation. In *SAC*, pages 19–24, 1999.

[28] A. Schiper. Distributed systems - lectures notes. EPFL Graduate Course, 1997.

[29] M.-C. Silaghi. Meeting scheduling guaranteeing n/2-privacy and resistant to statistical analysis (applicable to any DisCSP). In *3rd IC on Web Intelligence*, 2004.

[30] M.-C. Silaghi, Ş. Sabău, D. Sam-Haroud, and B.V. Faltings. Asynchronous search for numeric DisCSPs. In *Proc. of CP'2001*, page 786, Paphos,Cyprus, 2001.

[31] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proc. of AAAI2000*, pages 917–922, Austin, August 2000.

[32] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Fractionnement intelligent de domaine pour CSPs avec domaines ordonnés. In *RFIA2000*, Paris, February 2000.

[33] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Technical Report #01/364, EPFL, May 2001.

[34] M.-C. Silaghi, M. Zanker, and R. Bartak. Desk-mates (stable matching) application with privacy of preferences, and a new distributed csp framework. In *Proc. of CP'2004 Immediate Applications of Constraint Programming Workshop*, 2004.

[35] G. Solotorevsky, E. Gudes, and A. Meisels. Algorithms for solving distributed constraint satisfaction problems (DCSPs). In *AIPS96*, 1996.

[36] E. H. Turner and J. Phelps. Determining the usefulness of information from its use during problem solving. In *Proceedings of AA2000*, pages 207–208, 2000.

[37] R. Wallace and M.C. Silaghi. Using privacy loss to guide decisions in distributed CSP search. In *FLAIRS'04*, 2004.

[38] WebProof. Detailed Proof for AAS. http://liawww.epfl.ch/~silaghi/annexes/AAAI2000, 2000.

[39] M. Yokoo. Asynchronous weak-commitment search for solving large-scale distributed constraint satisfaction problems. In *1st ICMAS*, pages 467–318, 1995.

[40] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *ICDCS*, pages 614–621, June 1992.

[41] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE TKDE*, 10(5):673–685, 1998.

[42] Y. Zhang and A. K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Proc. of Third IEEE Symposium on Parallel and Distributed Processing*, pages 394–397, 1991.