Polynomial-Space and Complete Multiply Asynchronous Search with Abstractions

Marius-Călin Silaghi, Djamila Sam-Haroud, and Boi Faltings
EPFL, CH-1015, Switzerland
{Marius.Silaghi,Djamila.Haroud,Boi.Faltings}@epfl.ch

ABSTRACT

Problems may consist of constraints distributed among several agents. Some problems are naturally distributed. Others were originally centralized but the distribution is expected to help for their resolution. Among naturally distributed problems we focus on those where the distribution is reinforced by privacy requirements. Privacy, robustness against timing variations and robustness against hard failures are improved by asynchronism.

The tractability of centralized problems can often be ameliorated using abstractions. No sustained research work for integrating abstraction schemas in asynchronous search has been done. The existing techniques suffered from the reduced propagation In this work we show how a high degree of asynchronism in search can accommodate abstraction schemas for distributed problems. Three abstraction techniques, as well as possible extensions are discussed in the framework of asynchronous complete distributed techniques with polynomial space requirements and asynchronous maintenance of consistency with asynchronous reordering. The new framework can deal even with "continuous" domains for numerical constraints.

1. INTRODUCTION

A constraint satisfaction problem (CSP) is defined as a set of variables taking their values in particular domains and subject to constraints that specify consistent value combinations. Solving a CSP amounts to assigning to its variables values from their domains so that all the constraints are satisfied. Distributed constraint satisfaction problems (DisCSPs) arise when the constraints or variables come from a set of independent but communicating agents.

Maintaining local consistency with dynamic reordering during backtrack search (e.g. [15]) is one of the most powerful techniques for solving centralized CSPs. While in that setting, instantiation and consistency enforcement steps alternate sequentially, more elaborate combination schemes are required within distributed CSPs for enabling the agents to act asynchronously. Asynchronism is desirable since it gives the agents more freedom in the way they can contribute to search and enforce their privacy policies. Namely, before the deadline of an agent for giving some information, other agents' announcements may preempt the need of undesired disclosures. It also increases both parallelism and robustness. In particular, robustness is improved by the fact that the search may still detect unsatisfiability even in the presence of crashed agents.

In this paper we target problems with finite domains. However, digital-computer-based approaches to problems with reals (elements of $\mathbb{R}$), actually use very large but finite domains (subsets of $\mathcal{F}$), due to the finite representations available for floating-point numbers [25, 9]. The elements of $\mathcal{F}$ have the form $[f, f]$ or (f_1, f_2), where f,f_1,$f_2$$\in$$\mathbb{F}$, $\mathbb{F}$ is the set of representable floating-point numbers on the used system and f_1, f_2 are consecutive elements of $\mathbb{F}$. The successful approaches to problems with domains from $\mathbb{R}$ *aggregate* consecutive elements of $\mathcal{F}$ into intervals (elements of $\mathbb{I}$). In this paper we extend an approach that can use such aggregations for dealing with large domains. Moreover, this approach is not necessarily constrained to aggregate only consecutive elements of a given domain [17].

We consider that each agent A_i wants to satisfy a local CSP, CSP(A_i). The agents may keep their constraints private but publish their interest on variables. We distinguish four facets of asynchronism.

a) *deciding instantiations* of variables by distinct agents. The agents can propose different instantiations asynchronously (e.g. Asynchronous Backtracking (ABT) [31]).

b) *enforcing consistency*. The distributed process of achieving "local" consistency on the global problem is asynchronous (e.g. Distributed Arc Consistency [33]).

c) *maintaining consistencies asynchronously*. Actually this corresponds to mixing the previous two asynchronisms. Instantiations and asynchronous local consistency enforcements no longer alternate sequentially. Consistency can be enforced concurrently at all levels in the distributed search trees. Domain reductions are used as soon as available (e.g. Maintaining Hierarchical Distributed Consistency [20]).

d) *performing reordering*. Dynamic reordering can be decided asynchronously by several agents. Asynchronous Weak Commitment [31] is a powerful technique allowing for asynchronous reordering, but requires exponential space in the number of variables for completeness.

Multiply Asynchronous Search (MAS) [21], is a generic complete protocol which integrates asynchronisms of type a, b, c, and d, and requires polynomial space. By $MAS_{(\pm,\pm,\pm)}$ are denoted versions of MAS that behave with asynchronisms of different types. "+" stands for presence and "−" for absence of some type of asynchronism, respectively a, b or d, according to the corresponding position. The asynchronism of type c is therefore represented by $MAS_{(+,+,\pm)}$.

Paraphrasing [8] one can say that a problem only becomes tractable when the appropriate abstractions are found. Sometimes it happens that an abstraction exists that brings important improvements for a whole class of problems without necessarily making any of its subclasses tractable. This is the case of the dichotomous search for numerical problems. We show in this work how asynchronism of type c allows for such abstractions to be exploited in new ways.

The section 3 presents the basics of instantiation asynchronism in Distributed CSPs while section 4 introduces the way in which different types of asynchronism coexist in MAS. Integration in MAS of a famous abstraction for numerical constraints, known mostly under the name of dichotomous search is presented in section 5. Two other types of abstractions, one of them being new for centralized CSPs as well, are described in section 6. They are based on the actual structure of the constraints.

2. RELATED WORK

The first complete asynchronous search algorithm for DisCSPs is the Asynchronous Backtracking (ABT) [30]. For simplicity the approach in [30] considers that each agent maintains only one distinct variable. More complex definitions were given later [32, 27]. The first version of ABT has requested the agents to store all the nogoods, but this constraint was easily removed in [31, 11, 27, 17], where versions of ABT with polynomial space-complexity are mentioned. Other definitions of DisCSPs have considered what happens in the case where the knowledge and interest on constraints is distributed among agents [33, 23, 17]. [23] proposes a static ordering and distribution of the variables in ABT that fits the natural structure of a real problem (the nurse transportation problem). The Asynchronous Aggregation Search (AAS) [17] algorithm actually extends ABT to the case where the same variable can be instantiated by several agents and an agent may not know all constraint predicates relevant to its variables. AAS offers the possibility to aggregate several branches of the search. An aggregation technique for DisCSPs was then presented in [14] and allows for simple understanding of the privacy/efficiency mechanisms. The order on variables in distributed search was so far addressed in [7, 1, 29, 23, 10], showing the strong impact it has on the solving algorithms. In what concerns distributed consistency algorithms, one of the first is presented in [33].

After the definition of asynchronous backtracking, work has mainly concentrated on the intelligence of the backtracking by tuning the quality of the nogoods selected for storage [11, 27, 24, 17]. Asynchronous maintenance of consistency was difficult due to the dynamism of the instantiations in asynchronous search. The first such algorithm is presented in [20, 21]. Performing asynchronous reordering was first proposed in [28] according to a min-conflict heuristic. The first polynomial-space complete asynchronous reordering algorithm is presented in [21] where its integration within asynchronous maintenance of consistency is also described. Asynchronous use of abstractions in complete search is first described in [17].

3. BACKGROUND & DEFINITIONS

In asynchronous backtracking, the agents run concurrently and asynchronously. Each agent, A_i, instantiates its unique and distinct variable, x_i, and communicates the variable value to the relevant agents. Since we do not assume FIFO channels, a **local counter** is incremented each time a new instantiation is chosen, and its current value **tags** each assignment.

Definition 1 (Assignment) *An* assignment *for a variable x_i is a tuple $\langle x_i, v, c\rangle$ where v is a value from the domain of x_i and c is the* tag *value.*

Between two different assignments for the same variable, the one with the highest *tag* (attached value of the counter) is the **newest**. A static order is imposed on agents and we assume that A_i has the i-th position in this order. If $i>j$ then A_i has a *lower priority* than A_j and A_j has a *higher priority* then A_i.[1]

Rule 1 (Constraint-Evaluating-Agent) *Each constraint C is evaluated by the lowest priority agent whose variable is involved in C.*

Each agent holds a list of **outgoing links** represented by a set of agents. Links are associated with constraints. ABT assumes that every link is directed from the value sending agent to the constraint-evaluating-agent.

Definition 2 (Agent_View) *The* agent_view *of an agent, A_i, is a set containing the* newest *assignments received by A_i for distinct variables.*

Based on their constraints, the agents perform inferences concerning the assignments in their *agent_view*. By inference the agents generate new constraints called *nogoods*.

Definition 3 (Nogood) *A* nogood *has the form $\neg N$ where N is a set of assignments for distinct variables.*

The following types of messages are exchanged in ABT:

- **ok?** message transporting an assignment is sent to a constraint-evaluating-agent to ask whether a chosen value is acceptable.
- **nogood** message transporting a *nogood*. It is sent from the agent that infers a *nogood* $\neg N$, to the constraint-evaluating-agent for $\neg N$.
- **add-link** message announcing A_i that the sender A_j owns constraints involving x_i. A_i inserts A_j in its *outgoing links* and answers with an **ok?**.

The agents start by instantiating their variables concurently and send **ok?** messages to announce their assignment to all agents with lower priority in their *outgoing links*. The agents answer to received messages according to the Algorithm 1[27] (except for pseudo-code delimited by '*').

Definition 4 (Valid assignment) *An assignment $\langle x, v_1, c_1\rangle$ known by an agent A_l is valid for A_l as long as no assignment $\langle x, v_2, c_2\rangle$, $c_2>c_1$, is received.*

[1] They can impose first eventual preferences they have on their values

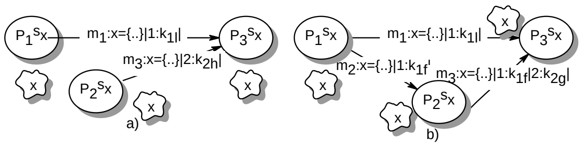


Figure 1: Simple scenarios with messages for proposals on a resource, x.

A **nogood is invalid** if it contains invalid assignments. The next property is mentioned in [31] and it is also implied by the Proposition 1, presented later.

Property 1 *If only one nogood is stored for a value then ABT has polynomial space complexity in each agent, $O(dn)$, while maintaining its completeness and termination properties. d is the domain size and n is the number of agents.*

3.1. HISTORIES

Now we introduce a *marking technique* [18] that allows for the definition of a total order among the proposals made concurently and asynchronously by a set of ordered agents on a shared resource (e.g. assignment of a variable, an order on a set of agents).

Definition 5 *A* **proposal source** *for a resource $\mathcal{R}$ is an entity (e.g. a delegated agent) that can make specific proposals concerning the allocation (or valuation) of $\mathcal{R}$.*

We consider that an order $\prec$ is defined on *proposal sources*. The *proposal sources* with lower position according to $\prec$ have a higher priority. The *proposal source* with position k is noted $P_k^{s_\mathcal{R}}$, $k \geq k_0^\mathcal{R}$. $k_0^\mathcal{R}$ is the first position.

Definition 6 *A* **conflict resource** *is a resource for which several agents can make proposals in a concurent and asynchronous manner.*

Each *proposal source* $P_i^{s_\mathcal{R}}$ maintains a counter $C_i^{p_\mathcal{R}}$ for each *conflict resource* $\mathcal{R}$ for which it can make proposals. The markers involved in our *marking technique for ordered proposal sources* are called **histories**.

Definition 7 *The* **history** *is a chain h of pairs, $|a{:}b|$, that can be associated to each proposal for $\mathcal{R}$. A pair $p{=}|a{:}b|$ in h signals that a proposal for $\mathcal{R}$ was made by $P_a^{s_\mathcal{R}}$ when its $C_a^{p_\mathcal{R}}$ had the value b, and it knew the prefix of p in h.*

An order $\propto$ (read "precedes") is defined on pairs such that $|i_1{:}l_1| \propto |i_2{:}l_2|$ if either $i_1 < i_2$, or $i_1 = i_2$ and $l_1 > l_2$.

Definition 8 *A history h_1* **is newer than** *a history h_2 if a string-like comparison on them, using the order $\propto$ on pairs, decides that h_1 precedes h_2.*

This is a generalization of the notion *newer* on assignments. $P_k^{s_\mathcal{R}}$ builds a history for a new proposal on $\mathcal{R}$ by prefixing to the pair $|k{:}l_{k^j}|$, the newest history that it knows for a proposal on $\mathcal{R}$ made by any $P_a^{s_\mathcal{R}}$, $a{<}k$. l_{k^j} is the current value of $C_k^{p_\mathcal{R}}$. The $C_a^{p_\mathcal{R}}$ in $P_a^{s_\mathcal{R}}$ is reset each time an incoming message announces a proposal with a newer history, made by higher priority *proposal sources* on $\mathcal{R}$. $C_a^{p_\mathcal{R}}$ is incremented each time $P_a^{s_\mathcal{R}}$ makes a proposal for $\mathcal{R}$.

Definition 9 *A history h_1 built by $P_i^{s_\mathcal{R}}$ for a proposal on $\mathcal{R}$ is* **valid** *for an agent A if no other history h_2 (eventually known only as prefix of a history h_2') is known by A such that h_2 is newer than h_1 and was generated by $P_j^{s_\mathcal{R}}$, $j \leq i$, for $\mathcal{R}$.*

For example, in Figure 1 the agent $P_3^{s_x}$ may get messages concerning the same resource x from $P_1^{s_x}$ and $P_2^{s_x}$ and has to decide which of them is the most up to date. In Figure 1 a), if the agent $P_3^{s_x}$ has already received m_1, it will always discard m_3 since the *proposal source* index has priority. However, in the case of Figure 1 b) the message m_1 will be maintained only if $k_{1f} < k_{1l}$. In each message, the length of the history for a resource is upper bounded by the number of *proposal sources* for that *conflict resource*.

3.2. ASYNCHRONOUS AGGREGATION SEARCH

Asynchronous Aggregation Search (AAS) [17] is an extension of ABT where several agents are allowed to simultaneously propose instantiations for the same shared variable. We assume that before search, each agent announces the shared variables that it wants to be able to assign. This is made possible by using the *histories* presented in the previous subsection where agents A_i can act as $P_i^{s_{x_k}}$ for some shared variable x_k. $k_0^{x_k} = 1$ for any shared variable x_k.

Constraint enforcement We allow any agent A_i to own private constraints C that are not known to all the agents that can make proposals on the variables in C. To enforce C, A_i:

- has to announce at beginning that it wants to modify all the shared variables in C (this is always possible), or
- has to be ordered such that some agents with lower positions want to modify all the shared variables in C that A_i does not want to modify (as for nogoods in ABT).

An agent does not need to enforce a constraint, C, that it has when it knows that another agent with higher position enforces C (this is the case of initial constraints in ABT).

Aggregations If all the agents owning constraints on some variable x_i announce at beginning that they want to make proposals with assignments for x_i (or at most one agent owning constraints on x_i makes exception but is ordered after the others), than the agents can *aggregate* several assignments for x_i into one proposal. Several branches of the search are therefore aggregated. For simplicity, in the rest of the description we consider that this is the case for all shared variables. In AAS, as presented further in this subsection, the agents exchange messages about sets of values for combinations of variables (aggregate-sets). We refer to an *aggregate* proposed for a variable x by an agent A_i as a *proposal of A_i on x*.

Definition 10 *An* **aggregate** *is a triplet (x_j, s_j, h_j) where x_j is a variable, s_j a set of values for x_j, $s_j{\neq}\emptyset$, and h_j a history of the pair (x_j, s_j). It is a generalization of an* assignment.

The history guarantees a correct message ordering. It determines if a given aggregate is more recent than another. Let $a_1 = (x_j, s_j, h_j)$ and $a_2 = (x_j, s_j', h_j')$ be two aggregates for the variable x_j. a_1 is *newer* than a_2 if h_j is more recent than h_j'. The newest aggregates received by an agent A_i for each variable define its **view**, view(A_i) (the extension of an *agent_view* in ABT). An **aggregate-set** is a set of aggregates and can be seen as a Cartesian-product of the sets of assignments defined by these aggregates (a set of tuples corresponding to partial valuations). Let V be an aggregate-set and vars(A_i) the variables of CSP(A_i). The set of tuples disabled from CSP(A_i) by V is formally $\mathbf{T_i(V)}=\{t \mid t=(x_1=v_t^1, ..., x_n=v_t^n), \forall j, x_j \in \text{vars}(A_i);\ \forall u \neq j, x_j \not\equiv x_u;\ \text{n}=|\text{vars}(A_i)|; \exists k \in [1..n], (x_k, s_k, h_k) \in V, v_t^k \notin s_k\}$.

Definition 11 *$V' \rightarrow \neg T_i(V)$ is a* **nogood entailed for $\mathbf{A_i}$ by its view V**, *denoted* $\mathbf{NV_i(V)}$, *iff $V' \subseteq V$ and $T_i(V')=T_i(V)$.*

Definition 12 *An* **explicit nogood** *has the form $\neg V$, or "$V \rightarrow$fail", where V is an aggregate-set.*

The information in the received nogoods that is essential for completeness can be stored compactly in a polynomial space structure called conflict list nogood.

Definition 13 *A* **conflict list nogood (CL) for $\mathbf{A_i}$** *has the form "$V \rightarrow \neg T$", where $V \subseteq$view(A_i) and T is a set of tuples: $T=\{t \mid t=(x_{t^1}=v_t^1, ..., x_{t^{n_t}}=v_t^{n_t}), \forall k, x_{t^k} \in vars(A_i)\}$, such that T can be represented by the structures (e.g. stack) of a centralized backtracking algorithm.*

In order to obtain instantiation asynchronism (type a), with no infinite loops, AAS uses a strict order $\prec$ on agents as proposed for ABT. In the sequel of the paper, A_i^j denotes the agent A_i with the position $j, j \geq 1$, when the agents are ordered by $\prec$. If $j > k$, we say that A^j has a lower priority than A^k. A^j is then a successor of A^k, and A^k a predecessor of A^j.

The AAS protocol is defined by the next messages:

- **ok?** messages having as parameter an aggregate-set, V.
- **nogood** messages announcing an explicit nogood.
- `addlink(vars)` messages transporting a set of variables. They are sent from agent A^j to agent A^i, $j>i$ and inform A^i that A^j is interested in the variables $vars$.

`ok(`V`)` messages announce proposals of domains for a set of variables and are sent from agents with higher priorities to agents with lower priorities. The proposal is sent to all successor agents interested in it. Let the set of valid aggregates known to the sender A_i be denoted known(A_i). $V \subseteq$known(A_i). Any tuple not in T_i(known(A_i)) must satisfy the local constraints of the sender A_i and its valid nogoods[2]. An agent maintains its view and a valid CL and always enforces its CL and its nogood entailed by the view. Generally, an aggregate has to be built and added to V by A_i only if the newest aggregate for the same variable known by A_i does not have the same set of values.[3] `nogood` messages are sent from agents with lower priorities to agents with higher priorities. If given its constraints and valid nogoods an agent can find no proposal, in finite time it sends an explanation under the form of an explicit nogood $\neg N$ via a `nogood` message to the lowest priority agent that has built an aggregate in N. An empty nogood signals failure of the search. On the receipt of a valid nogood that negates its last proposed aggregate-set, V, an agent knows that proposal V is refused. Any received valid explicit nogood is merged into the maintained CL using the next inference technique:

[2]Except for constraints about which A_i knows that a successor enforces them (as in ABT).

[3]Exceptions appear for the first proposal made by A_i after nogoods of certain types are discarded (two alternatives are presented in [26]).

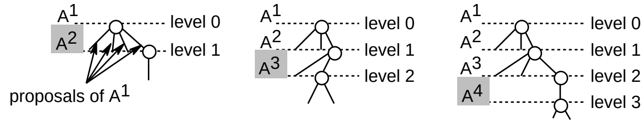


Figure 2: *Distributed search trees: simultaneous views of distributed search seen by A^2, A^3, and A^4, respectively. Each arc corresponds to a proposal of an agent.*

$$\begin{array}{rl} & V_1 \wedge V_2 \rightarrow \neg T^1 \\ & V_1 \wedge V_3 \rightarrow \neg T^2 \\ \hline \Rightarrow & V_1 \wedge V_2 \wedge V_3 \rightarrow \neg(T^1 \vee T^2), \end{array} \quad (1)$$

where V_1, V_2 and V_3 are aggregates, obtained by grouping the elements of the nogoods, such that they have no variable in common.

Property 2 *AAS is correct, complete, terminates and only requires polynomial space for its completeness.*

The proof is given in [17] and is also a corollary of Proposition 1 presented later.

Distributed Consistency (DC) A centralized local consistency algorithm prunes from the domain of variables locally inconsistent values. A computed restricted domain is called **label**. Let the **union** of CSPs and constraints be a CSP containing all the constraints and variables referred in arguments. Let P be a Distributed CSP with the agents $A_i, i \in \{1..n\}$ and let $C(P)$ be $\cup_{i \in \{1..n\}}$CSP(A_i). Let $\mathcal{A}$ be a centralized local consistency algorithm that computes unary labels (e.g. arc-, bound-consistency). We denote by DC($\mathcal{A}$) a distributed consistency algorithm that computes by exchanging value eliminations the same labels for P than $\mathcal{A}$ for $C(P)$. If such labels were computed for P, we say that P becomes DC consistent. Generic instances of DC($\mathcal{A}$) are denoted by DC. Typically with DC, the maximum number of generated messages is a^2vd and the maximum number of sequential messages is vd (v:number of variables, d:domain size, a:number of agents).

Each agent has its own perception of the distributed search. In this perception, the search spaces associated to arcs of the search tree are defined by proposals from its predecessors. In Figure 2 is shown a simultaneous view of three agents. Only A^2 knows the fourth proposal of A^1. A^3 has not yet received the third proposal of A^2 consistent with the third proposal of A^1. However, A^4 knows that proposal of A^2. Since it has not received anything valid from A^3, A^4 assumes that A^3 agrees with A^2. In practice it may happen that an agent sees only partly the valid proposal of another agent. The depth in distributed search trees is referred to as **level**. Histories help agents to eventually get coherent views at the same level.

4. MULTIPLY ASYNCHRONOUS SEARCH

In this paper, instantiation asynchronism (type a) is understood as in AAS. Consistency asynchronism (type b) is obtained using DC algorithms. In a distributed setting, asynchronous maintaining of consistency (type c) cannot be done in a straightforward manner if the instantiations are modified asynchronously. The classical techniques require instantiation and consistency maintenance stages to be interleaved sequentially and this induces synchronism of the instantiations.

A solution proposed in [20] consists in considering consistency maintenance as a hierarchical task. The consistency at any given level k in the 'distributed' search trees, denoted DC(k), corresponds to a level k in a hierarchy of concurrent and asynchronous DC processes. DC(k) computes labels based on the newest instantiations proposed by the agents $A^1, ..., A^k$.

For making a proposal (aggregate), there is no need to wait in a node until all the consistencies in the previous levels on the same branch converge. Equivalent labels can be computed asynchronously. In fact, at a given level value elimination results from one of the following processes: instantiation, DC at this level, or inheritance from DCs at previous levels along the same branch. A first version of $MAS_{(+,+,-)}$ is called Maintaining Hierarchical Distributed Consistency (MHDC) and runs all these processes concurrently [20].

We assume that a message does not necessarily need to head directly to its target agent. Its content can travel indirectly to the target via several agents. However, we require the content to arrive at destination in finite time. Parts of the content of a message may become *invalid* due to newer available information. The receiver can discard the invalid incoming information, or can reuse invalid nogoods with alternative semantics (e.g. as redundant constraints). However, we allow the channel (intermediary agents) to discard information that it knows invalid.

4.1. MAINTAINING CONSISTENCY ASYNCHRONOUSLY

The `ok`, `nogood` and `addlink` messages are as in AAS. In addition, the agents may exchange information about nogoods inferred by DCs. This is done using `propagate` messages.

Definition 14 *A* **consistency nogood** *for a level k and a variable x has the form $V \rightarrow (x \in l_x^k)$ or "$V \rightarrow \neg(x \in s \setminus l_x^k)$". V is an aggregate-set and may contain for x an aggregate (x,s,h), $l_x^k \subset s$. Any aggregate in V must have been proposed by agents $A^v, v \leq k$. l_x^k is a label, $l_x^k \neq \emptyset$.*

The `propagate` messages are sent to all interested agents $A^i, i>k$. They take as parameters the reference k of a level and a list of consistency nogoods. Each consistency nogood for a variable x and a level k is tagged with the value c of a **counter $\mathbf{C_x^k}$** maintained at sender. The consistency nogoods are meant to allow at any agent A^i computation of DC consistent labels on the problem obtained by integrating the most recent proposals of the agents $A^j, j \leq k$. A_i may receive valid consistency nogoods of level k with aggregates for the variables *vars*, *vars* not in vars(A_i). A_i must send `addlink` messages to all agents $A^{k'}, k' \leq k$ not yet linked to A_i for all *vars*. In order to achieve consistencies asynchronously, besides the structures of AAS, implementations can maintain at any agent A_i^u, for any level k, $k<u$ (some instances may also do for $k=u$):

- The aggregate-set, V_k^i, of the newest valid aggregates proposed by agents $A^j, j \leq k$, for each interesting variable.
- For each variable x, $x \in$vars(A_i), for each agent $A_j^t, t>k$, the last consistency nogood sent by A_j for level k, denoted $cn_x^k(i,j)$, if valid. It has the form $V_{j,x}^k \rightarrow (x \in s_{j,x}^k)$.

C_x^k is incremented each time a new $cn_x^k(i,i)$ is stored. Let $cn_x^k(i,.)$ be $(\cup_{t,j}^{t \leq k} V_{j,x}^t) \rightarrow (x \in \cap_{t,j}^{t \leq k} s_{j,x}^t)$. If:

$$P_i(k) := \mathrm{CSP}(A_i) \cup (\cup_x cn_x^k(i,.)) \cup \mathrm{NV}_i(V_k^i) \cup \mathrm{CL}_k^i$$

then on each modification of $P_i(k)$, $cn_x^k(i,i)$ is recomputed by inference (e.g. using local consistency techniques) for each variable x for the problem $P_i(k)$. $cn_x^k(i,i)$ is initialized as an empty constraint set.

CL_k^i is the set of all nogoods known by A_i^u and having the form $V \rightarrow \neg T$ where $V \subseteq V_k^i$ and T is a set of tuples in CSP(A_i). CL_k^i may contain the CL of A_i^u. An agent can manage to maintain one CL for each instantiation level and the space requirements do no change. $cn_x^k(i,i)$ is stored and sent to other agents by `propagate` messages iff any constraint of CSP(A_i) or CL_k^i was used for its logical inference from $P_i(k)$ and its label shrinks.

We now give the proof the correctness, completeness and termination properties of $MAS_{(+,+,-)}$. We only use DC techniques that terminate. Techniques to achieve such a behavior are outside the scope of this paper. (see [33, 3]). By quiescence of a group of agents we mean that none of them will receive or generate any valid nogoods, new valid assignments, or addlink messages. We start by proving the following property:

Property 3 *$\forall i$ in finite time t^i either a solution or failure is detected, or all the agents $A^j, 1 \leq j \leq i$ reach quiescence in a state where they are not refused a proposal satisfying* CSP(A^j)$\cup$NV$_j$(view(A^j)).

Proof. The proof is by induction on i. Let this be true for the agents $A^j, j<i$. Let τ be the maximum time taken by a message. After $t^{i-1}+\tau$, A^i no longer receives `ok` messages. A^i receives the last valid `ok` message at time $t_o^i \leq t^{i-1}+\tau$. $\exists t_v^i$, $t^{i-1}+\tau \geq t_v^i$ such that after t_v^i, view(A^i) and all $V_k^u, k \leq i$ of any agent A_u are no longer modified.
The set of disabled tuples in $\mathrm{CL}_k^u, k \leq i$ can contain only a bounded number of elements for each agent A_u and even if they can be discarded, they cannot be invalidated after t_o^i. $\mathrm{CL}_k^u, k \leq i$ cannot be invalidated after t_v^i. Since DCs were assumed to terminate, they terminate after each modification of a CL_k^u. Since the number of such modifications that can generate a new consistency nogood after t_v^i is bounded, after a finite time no consistency nogood is received any longer by A^i for levels $k \leq i$.
Since the domains are finite, A^i can make only a finite number of different proposals satisfying view(A^i). Once any of them is sent, the total number of consistency nogoods that can be received before the proposal is modified is finite (this results by inference to $i+1$ of the reasoning in the previous paragraph).
Only one valid explicit nogood can be received for a proposal since the proposal is immediately changed on such an event. Therefore, there is a finite number of valid nogoods that can be received by A^i for any of its proposals made after t_v^i (and after t_o^i).

1. If one of the proposals is not refused by incoming nogoods, and since the number of such nogoods is finite, the induction step is correct.

2. If all proposals that A^i can make after t_o^i are refused or if it cannot find any proposal, A^i has to send according to rules inherited from AAS a valid explicit nogood $\neg N$ to somebody. $\neg N$ is valid since all the assignments of $A^k, k<i$ were received at A^i before t_o^i.

2.a) If N is empty, failure is detected and the induction step is proved.

2.b) Otherwise $\neg N$ is sent to a predecessor $A^j, j<i$. Since $\neg N$ is valid, the proposal of A^j is refused, but due to the premise of the inference step, A^j either

2.b.i) finds a new proposal modifying at least one of the elements of N or

2.b.ii) discards $\neg N$ which does not completely fit CL, or A^j discards $\neg N$ after merging it to a CL invalidated later,or

2.b.iii) announces failure by computing an empty nogood (induction proven).

In the case (i), since $\neg N$ was generated by A^i, A^i is interested in all its variables, and it will be announced by A^j of the modification by an `ok` messages. In the case (ii), according to the rules of AAS for merging valid nogoods to CLs, all the variables of A^j not reassigned by its predecessors are reassigned by itself and either the reassignment from predecessors, or the one from A^j will be sent to A^i.

Both cases (i and ii) contradict the assumption that the last `ok` message was received by A^i at time t_o^i and the induction step is therefore proved for all alternative cases. The property can be attributed to an empty set of agents and it is therefore proved by induction for all agents. □

Proposition 1 $MAS_{(+,+,-)}$ *is correct, complete, terminates.*

Proof.

Completeness: All the nogoods are generated by logical inference from existing constraints. Therefore, if a solution exists, no empty nogood can be generated.

No infinite loop: The result follows from property 3.

Correctness: All valid proposals are sent to all interested agents and stored there. At quiescence all the agents know the valid interesting assignments of all predecessors. If quiescence is reached without detecting an empty nogood, then all the agents agree with their predecessors and their intersection is nonempty and correct as proved for AAS. □

If the agents using $MAS_{(+,+,-)}$ maintain all the valid consistency nogoods that they have received, then DCs in $MAS_{(+,+,-)}$ converge and compute a local consistent global problem at each level. If not all the valid consistency nogoods that they have received are stored, but some of them are discarded after inferring the corresponding $cn_x^k(i,i)$, some valid bounds or value eliminations can be lost when a $cn_x^k(i,i)$ is invalidated. Different labels are then obtained in different agents for the same variable. These differences have as result that the DC at the given level of $MAS_{(+,+,-)}$ can stop before the global problem is DC consistent at that level.

Among the consistency nogoods, $cn_x^k(i,i)$, that an agent computes itself from its constraints let it store only the last valid one for each variable. Let A_i also store only the last valid consistency nogood, $cn_x^k(i,j)$, sent to it for each variable $x \in$CSP(A_i) at each level k from any agent A_j. Then:

Proposition 2 *DC($\mathcal{A}$) labels computed at quiescence at any level with the help of* `propagate` *messages are equivalent to $\mathcal{A}$ labels when computed in a centralized manner on a processor. This is true whenever all the agents reveal consistency nogoods for all minimal labels, l_x^k, which they can compute.*

Proof. In each sent `propagate` message, the consistency nogood for each variable is the same as the one maintained by the sender. Any aggregate invalid in one agent will eventually become invalid for any agent. Therefore, any such nogood is discarded at any agent, iff it is also discarded at its sender. The labels known at different agents, being computed from the same nogoods, are therefore identical and the distributed consistency will not stop at any level before the global problem is local consistent in each agent. □

Proposition 3 *The minimum space an agent needs with $MAS_{(+,+,-)}$ for ensuring maintenance of the highest degree of consistency achievable with DC is O($a^2v(v+d)$). With bound consistency, the required space is O($(av)^2$).*

Proof. The space required for storing all valid aggregates is O(dav) for values and O(av) for histories (stored separately). The agents need to maintain at most a levels, each of them dealing with v variables, for each of them having at most a last consistency nogoods. Each consistency nogood refers at most v aggregates in premise and stores at most d values in label. The stack of labels requires therefore O($a^2v(v+d)$). The space required by CL and by the algorithm for solving the local problem depends on the corresponding technique (e.g. chronological backtracking requires O(v)). CL also refers at most v aggregates in its premise. An agents has to maintain at most av counters C_x^k for tagging consistency nogoods. □

The consistency maintaining computations performed by each agent in $MAS_{(+,+,-)}$ reuses available nogoods. In our knowledge, this is new even for centralized algorithms.

Nogoods detected by consistency The domain wipe-outs generated due to consistency nogoods may be detected by several agents simultaneously. This happens when inference from consistency nogoods generated concurrently by two agents leads to a domain wipe-out represented as an explicit nogood. We call the result of this inference **consistency explicit nogood**. Versions of MAS can be designed where all consistency nogoods are sent to all agents generating the corresponding consistency level. Moreover, the algorithm design can be such that any consistency nogoods of level i has in its premise an aggregate generated by the agent A^i. (This is the case for the version described here when A^u stores consistency nogoods for k=u and or all variables.) In such versions, a consistency explicit nogood for level i does not have to be sent by `nogood` messages since it is guaranteed that the target agent, A^i detects the nogood itself.

If only the condition of having premises generated by A^i in consistency nogoods at level i is not respected, the target may not be A^i, but an improvement is still possible by assigning only to A^i the task of sending corresponding consistency explicit nogoods by messages.

However, even if none of the conditions of algorithm design just mentioned are fulfilled, a convention can be established for sending by a nogood message from an agent A^k a consistency explicit nogood detected at level i for a domain wipe-out on the variable v and having as target an agent A^j. The convention requires A^k to send such a message only if none of the agents $A^{l,l\in[i,k)}$ is interested in v.

After computing a consistency explicit nogood from consistency nogoods at level i, an agent A^k does not have to wait for this nogood to be invalidated. A^k is allowed to use the otherwise idle time for a search where the consistency nogoods of level i and upper are not used.

Consistency nogoods enforcement In ABT, parallelism is increased by discarding the assignment of the agent to which a nogood is sent. In MAS the same technique can be applied. However, other sources of parallelism are available in MAS due to consistency maintenance. As we have shown for Generalized Partial Order Backtracking (GOB) [16], it is sometimes useful to keep valid even nogoods for which a nogood message is sent, while the space complexity does not change. In MAS, when a *consistency explicit nogood*, c_i^k, is computed by an agent A_i at a level k, A_i continues to keep all assignments in all consistency levels. However, as long as c_i^k is valid, A_i only enforces in its proposals consistency nogoods at levels lower than k. This is a simplification since some consistency nogoods at higher levels may not depend on A^k.

4.2. REORDERING WITH RESTARTS

The importance of reordering has been highlighted in both centralized and distributed environments. In centralized settings special efforts have been made to introduce reordering in dynamic backtracking [4]. That algorithm reuses an important quantity of previous work after reordering. In distributed environments, [27] proposes several algorithms based on reordering such as: the "Distributed Breakout" algorithm and the "Asynchronous Weak-Commitment Protocol" (AWC). Of these, only AWC has a version offering completeness. However, the complete version needs exponential space and maintains no consistency.

One of the simplest approaches to reordering consists of using timeouts for restarting the search with random agent, variable and value order. In this context, completeness is ensured by monotonously increasing the timeout after each restart. The random reordering refers to agents, to their variables or constraints and to the sequence of the values in domains. Even if this is a quite simple method, within centralized settings of satisfiability problems it was shown to be sometimes comparable in efficiency to more complex reordering strategies [2].

4.3. REORDERING AGENTS ASYNCHRONOUSLY

In the presence of private constraints, besides efficiency considerations, the order of the agents have an additional importance. The last agent has a better position from the point of view of the security of its privacy. It has to reveal at most one feasible tuple, after which a solution is found. The last agent can also refuse all the proposals it receives and this way it can find the whole set of solutions for the other agents. In turn, when the agents are honest then the first agents are advantaged. They can launch the search first on the alternatives they prefer. If solutions are found, they are not interested in declaring alternatives that are less good for them. From the point of view of privacy it is interesting to let disadvantaged higher priority agents reorder lower priority ones. Security considerations can also require special orders on the participants and such orders change dynamically and their need is detected asynchronously.

Reordering with dedicated agents Besides the agents $A_1, ..., A_n$ in the DisCSP we want to solve, we consider now that there exist $n-1$ other agents, $R^0, ..., R^{n-2}$, that are solely devoted for reordering the agents A_i.

Definition 15 *An* **ordering** *is a sequence of distinct agents* $A_{k_0}, ..., A_{k_n}$.

An agent A_i may receive the position j, $j \neq i$, $j>0$. Let us assume that the agent A_l, knowing an ordering o, believes that the agent A_i, owning the variable x_i, has the position j. A_l can refer A_i as either A^j, $A^j(o)$ or A_i^j. The variable x_i is also referred to by A_l as either x^j, $x^j(o)$ or x_i^j.

We propose to consider the ordering on agents as a *conflict resource*. We attach to each ordering a *history* as defined in section 3.1. The *proposal sources* for the ordering on agents are the agents R^i, $P_i^{x_{order}}=R^i$, where $R^i \prec R^j$ if $i<j$ and $x_0^{\text{order}}=0$. R^i is the *proposal source* that when knowing an ordering, o, can propose orderings that reorder only agents on positions p, $p > i$.

Definition 16 (Known order) *An ordering known by R^i (respectively by A^i) is the order o^{crt} with the newest history among those proposed by the agents $R^k, 0 \leq k<i$ and received by R^i (respectively A^i). A^i has the position i in o^{crt}. This order is referred to as the* known order of R^i *(respectively of A^i).*

Definition 17 (Proposed order) *An ordering, o,* proposed by R^i *is such that the agents placed on the first i positions in the* known order of R^i *must have the same positions in o. o is referred to as the* proposed order of R^i.

Let us consider two different orderings, o_1 and o_2, with their corresponding histories: $O_1 = \langle o_1, h_1 \rangle$, $O_2 = \langle o_2, h_2 \rangle$; such that $|h_1| \leq |h_2|$. Let $p_1^k = |a_1^k{:}b_1^k|$ and $p_2^k = |a_2^k{:}b_2^k|$ be the pairs on the position k in h_1 respectively in h_2. Let u be the lowest position such that p_1^u and p_2^u are different and let $v = |h_1|$.

Definition 18 (Reorder position) *The* **reorder position** *of h_1 and h_2, denoted R(h_1,h_2), is either $min(a_1^u, a_2^u) + 1$ if $u > v$, or $a_2^{v+1} + 1$ otherwise. This is the position of the highest priority reordered agent between h_1 and h_2.*

New optional messages for reordering are: `heuristic` messages for heuristic dependent data, and `reorder` messages announcing a new ordering, $\langle o, h \rangle$.

An agent R^i announces its proposed order o by sending `reorder` messages to all agents $A^k(o), k > i$, and to all agents $R^k, k > i$. Each agent A_i and each agent R^i has to store an ordering denoted o^{crt}. o^{crt} contains the ordering with the newest history that was received. For allowing asynchronous reordering, each **ok?** and **nogood** message receives as additional parameter an order and its history (see Algorithm 1). The **ok?** messages hold the newest *known order of* the sender. The **nogood** messages hold the order in the o^{crt} at the sender A^j that A^j believes to be the newest *known order of* the receiver, A^i. This ordering consists in the first i agents in the newest ordering, o^{crt}, known by A^j and is tagged with a history obtained from the history of its o^{crt} by removing all the pairs $|a{:}b|$ where $a \geq i$.[4]

When a message is received which contains an order with a history h that is newer that the history h^* of o^{crt}, let the *reordering position* of h and h^* be I^r (I^r=R(h,h^*)). The assignments for the variables x^k, $k \geq I^r$, are invalidated.[5]

[4]The agents absent from the ordering in a nogood are not needed by A^i. A^i receives them when it receives the corresponding `reorder` message.

[5]Alternative rule: A^i can keep valid the assignments of new variables $x^k(h)$, $i \geq k \geq I^r$ but broadcasts assignments of x^i again. In AAS, histories of assignments kept valid have to be updated by removing pairs for predecessors that become successors and updating the indexes for predeces-

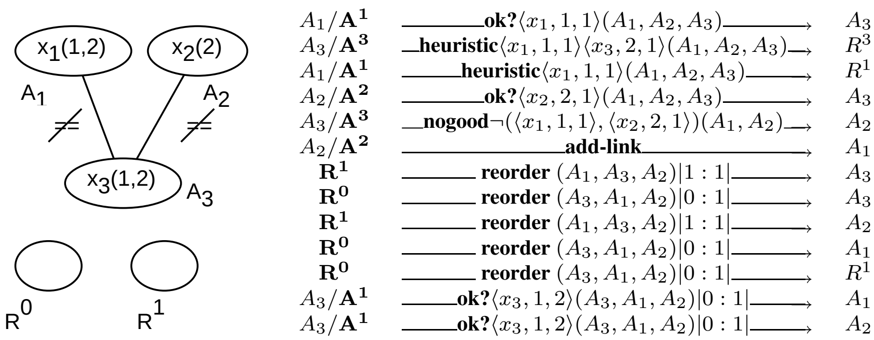


Figure 3: Simplified example for ABT with random reordering based on dedicated agents (R^0, R^1). A_i/A^j in the left column shows that A_i has believed to have the position j when it has sent the message. The time increases downwards.

The agents R^i can modify the ordering in a random manner or according to special strategies appropriate for a given problem.[6] Sometimes it is possible to assume that the agents want to collaborate in order to decide an ordering.[7] The `heuristic` messages are intended to offer data for reordering proposals. The parameters depend on the used reordering heuristic. The `heuristic` messages can be sent by any agent to the agents R^k. `heuristic` messages may only be sent by an agent to R^k within a bounded time, $\mathbf{t_h}$, after having received or decided a new assignment for $x^j, j \leq k$. Agents can only send `heuristic` messages to R^0 within time t_h after the start of the search. Any `reorder` message is sent within a bounded time $\mathbf{t_r}$ after a `heuristic` message is received.

Besides C_k^porder and o^{crt}, the other structures that have to be maintained by R^k, as well as the content of `heuristic` messages depend on the reordering heuristic. The space complexity for A^k remains the same as with ABT.

ABT with Asynchronous Reordering (ABTR) In fact, we have introduced the physical agents R^i in the previous subsection only in order to simplify the description of the algorithm. Any of the agents A_i or other entity can be delegated to act for any R^j. When proposing a new order, R^i can also simultaneously delegate the identity of $R^i, ..., R^{n-2}$ to other entities, P_k, by attaching a sequence $R^0 \rightarrow P_{k_i}, ..., R^{n-2} \rightarrow P_{k_j}$ to the ordering. At a certain moment, due to message delays, there can be several entities believing that they are delegated to act for R^i based on the ordering they know. However, any other agent can coherently discriminate among messages from simultaneous R^is using the histories that R^is generate. The R^i themselves coherently agree when the corresponding orders are received. The delegation of $R^j, j > i$ from a physical entity to another poses no problem of information transfer since the counter C_j^porder of R^j is reset on this event. The counter of a new R^i delegated by a previous different R^i is set to the incremented value of the counter for the pair with index i in the history that tags the `reorder` message.

Other examples of heuristics In the example in Figure 4 we describe the case where the activity of R^i is always per-

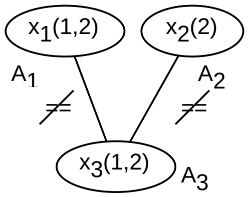
sors that change their position.

[6] e.g. first the agents forming a coalition with R^i, or first the agents that are suspected to be actually trying to spy the constraints of others.

[7] This can aim to improve the efficiency of the search. Since ABT performs forward checking, it may be possible to design useful heuristics.

when received *(***ok?**,*(x_j,d_j *,c_{x_j}, $\langle o,h\rangle$*))* **do**
  ***if**(¬**getOrder**($\langle o,h\rangle$) or old c_{x_j}) return*; //ABTR;
  add(x_j,d_j*,c_{x_j}*) to *agent_view*; **check_agent_view**;
end do.
when received *(***nogood**,A_j,nogood*,$\langle o,h\rangle$*) **do**
  ***if**(¬**getOrder**($\langle o,h\rangle$)) return*; //ABTR;
  *discard *nogood* if it contains invalid assignments **else***; //ABTR;
    when (x_k,d_k,c_k), where x_k is not connected,
        is contained in *nogood*
      send **add-link** to A_k; add (x_k,d_k,c_k) to *agent_view*;
    add *nogood* to *nogood-list*;
    add other new assignments to *agent_view*;
  old_value ← *current_value*; **check_agent_view**;
  when *old_value* = *current_value*
1.1   send (**ok?**,(x_j,*current_value*,c_{x_i})) to A_j;
end do.
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 ← d; c_{x_i}++;
      send (**ok?**,(x_i,d,c_{x_i})) to lower priority agents in *outgoing links*;
    end
end do.
procedure backtrack do
  nogoods ← $\{V \mid V =$ 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*;
    select (x_j,d_j*,c_{x_j}*) where x_j has the lowest priority in V;
    send (**nogood**,x_i,V) to A_j;
    remove (x_j,d_j*,c_{x_j}*) from *agent_view*;
  check_agent_view;
end do.
function getOrder($\langle o,h\rangle$) → *bool //ABTR*
  when h is invalidated by the history o^{crt} **then** return false;
  when not newer h than o^{crt} **then** return true;
  I ← *reorder position* for h and the history of o^{crt};
  invalidate assignments for $x^j, j \geq I$ (or alternative in footnote [5]);
  $\langle o,h\rangle \rightarrow o^{crt}$;
  make sure that `send` (**ok?**,(x_i,d,c_{x_i})) will be performed
    to all lower priority agents in *outgoing links*;
  return true;
end.

Algorithm 1: *Procedures for Receiving Messages in ABT and ABTR.*



$A_1/\mathbf{A^1}/R^1$	**ok?**$\langle x_1,1,1\rangle(A_1,A_2,A_3) \longrightarrow$	A_3
$A_2/\mathbf{A^2}/R^0/R^2$	**ok?**$\langle x_2,2,1\rangle(A_1,A_2,A_3) \longrightarrow$	A_3
$A_3/\mathbf{A^3}/R^3$	**nogood**$\neg(\langle x_1,1,1\rangle,\langle x_2,2,1\rangle)(A_1,A_2) \longrightarrow$	A_2
$A_1/\mathbf{R^1}/A^1$	**reorder** $(A_1,A_3,A_2)\vert 1:1\vert \longrightarrow$	A_3
$A_2/\mathbf{R^0}/A^3/R^2$	**reorder** $(A_3,A_1,A_2)\vert 0:1\vert \longrightarrow$	A_3
$A_1/\mathbf{R^1}/A^1$	**reorder** $(A_1,A_3,A_2)\vert 1:1\vert \longrightarrow$	A_2
$A_2/\mathbf{R^0}/A^3/R^2$	**reorder** $(A_3,A_1,A_2)\vert 0:1\vert \longrightarrow$	A_1
$A_3/\mathbf{A^1}/R^1$	**ok?**$\langle x_3,1,2\rangle(A_3,A_1,A_2)\vert 0:1\vert \longrightarrow$	A_1
$A_3/\mathbf{A^1}/R^1$	**ok?**$\langle x_3,1,2\rangle(A_3,A_1,A_2)\vert 0:1\vert \longrightarrow$	A_2

Figure 4: Simplified example for ABTR with random reordering. In this example, R^i delegations are done implicitly by adopting the convention "A^i is delegated to act for R^i". Left column: $A_i/A^j/R^{i_1}/R^{i_2}$... shows the roles played by A_i when the message is sent. In bold is shown the capacity in which the agent A_i sends the message. The addlink message in not shown.

formed by the agent believing itself to be A^i. R^i can send a `reorder` message within time t_r after an assignment is made by A^i since a `heuristic` message is implicitly transmitted from A^i to R^i. We also consider that A_2 is delegated to act as R^0. R^0 and R^1 propose one random ordering each, asynchronously. The receivers discriminate, based on histories, that the order from R^0 is the newest. The known assignments and nogood are discarded. In the end, the *known order* for A_3 is $(A_3, A_1, A_2)|0 : 1|$.

Another interesting instantiation of ABTR is when the activity of R^i is always performed by the agent believing itself to be A^{i+1}. In this case, R^i can delegate the sender of any valid received nogood to be the next R^i. The condition that $\exists t_h, t_r$ such that a reordering is not issued by R^k after $t_h + t_r$ from the quiescence of $A^1, ..., A^k$ is respected since as shown for GOB [16], a finite set of valid nogoods cover the space in finite time. Actually, any nogood received by R^k is stored in a format that only depends on assignments built by $A^1, ..., A^k$. This version resembles Asynchronous Weak Commitment (AWC) and we denote it ABT-AWC.

Property 4 *$\forall i$, in finite time t^i either a solution or failure is detected, or all the agents $A^j, 0<j\leq i$ reach quiescence in a state where they are not refused an assignment satisfying the constraints that they enforce and their* agent_view.

Proof. Let all agents $A^k, k<i$, reach quiescence before time t^{i-1}. Let τ be the maximum time needed to deliver a message.

$\exists t_p^i < t^{i-1}$ after which no **ok?** is sent from A^k, $k<i$. Therefore, no heuristic message towards any R^u, $u<i$, is sent after $t_h^i = t_p^i + \tau + t_h$. Then, each R^u becomes fixed and announces its last order before a time $t_r^i = t_h^i + \tau + t_r$. After $t_r^i + \tau$ the identity of A^i is fixed as A_l. A_l^i receives the last new assignment or order at time $t_o^i < t_r^i + \tau$.

Since the domains are finite, after t_o^i, A_l^i can propose only a finite number of different assignments satisfying its view. Once any assignment is sent at time $t_a^i > t_o^i$, it will be abandoned when the first valid nogood is received (if one is received in finite time). All the nogoods received after $t_a^i + 2\tau$ are valid since all the agents learn the last instantiations of the agents $A^k, k < i$ before $t_a^i + \tau$. Therefore the number of possible incoming invalid nogoods for an assignment of A^i is finite.

1.If one of the proposals is not refused by incoming nogoods, and since the number of such nogoods is finite, the induction step is correct.

2.If all proposals that A^i can make after t_o^i are refused or if it cannot find any proposal, A^i has to send a valid explicit nogood $\neg N$ to somebody. $\neg N$ is valid since all the assignments of $A^k, k<i$ were received at A^i before t_o^i.

2.a) If N is empty, failure is detected and the induction step is proved.

2.b) Otherwise $\neg N$ is sent to a predecessor $A^j, j<i$. Since $\neg N$ is valid, the proposal of A^j is refused, but due to the premise of the inference step, A^j either

2.b.i) finds an assignment and sends **ok?** messages, or

2.b.ii) announces failure by computing an empty nogood (induction proven).

In the case (i), since $\neg N$ was generated by A^i, A^i is interested in all its variables (has sent once an **add-link** to A^j), and it will be announced by A^j of the modification by an **ok?** messages. This contradicts the assumption that the last **ok?** message was received by A^i at time t_o^i (induction proved).

From here, the induction step is proven since it was proven for all alternatives.

In conclusion, after t_o^i, within finite time, the agent A^i either finds a solution and quiescence or an empty nogood signals failure.

After $t_h + t_r$, R^0 is fixed and the property is true for the empty set. The property is therefore proven by induction on i □

Proposition 4 *ABTR is correct, complete and terminates.*

Proof. Completeness: All the nogoods are generated by logical inference from existing constraints. Therefore, if a solution exists, no empty nogood can be generated.

No infinite loop: This is a consequence of the Property 4 for $i = n$.

Correctness: All assignments are sent to all interested agents and stored there. At quiescence all the agents know the valid interesting assignments of all predecessors. If quiescence is reached without detecting an empty nogood, then according to the Property 4, all the agents agree with their predecessors and the set of their assignments is a solution. □

Asynchronous reordering in MAS An **order** is given by a sequence o of distinct agents $A^1..A^k$. All the agents that do not appear in o are considered to be ordered by their name and follow A^k. All the decisions (aggregates and labels) taken within a certain order are tagged with that order. Figure 6 shows how for the problem in Figure 5 dynamic asynchronous reordering is achieved at each node. The problem has 3 variables and is defined by 3 agents with one constraint each. The heuristic in this example consists of choosing the next agent as the one with the least number of allowed tuples (least *volume*). To allow an agent A^k compute the volumes of its successors, `heuristic` messages are used. For example, if A^k acts for R^k, `heuristic` messages are sent to A^k by all agents that prove value eliminations for levels lower than or equal to $k+1$. The reordering decisions are communicated to agents $A^j, j > k$. In Figure 6 the ovals represent decisions taken by an agent. We assume that before the shown trace, some agent has decided the `order:`$\{A_1\}$`|0:0|` establishing the identity of A^1 as A_1. Agent A_1 decides the shown instantiation and order and informs appropriately its successors by sending `reorder` and `ok` messages. Upon receipt of an `ok` message from A_1, A_3 sends a propagate message to A_2 with a reduced domain for c. Once A_2 receives this reduction, the domain of a becomes $\{3\}$. This information is reported to A_1 by messages 6 and 7 and allows A_1 to reconsider the order. The reordering decision sent by agent A_1 via message

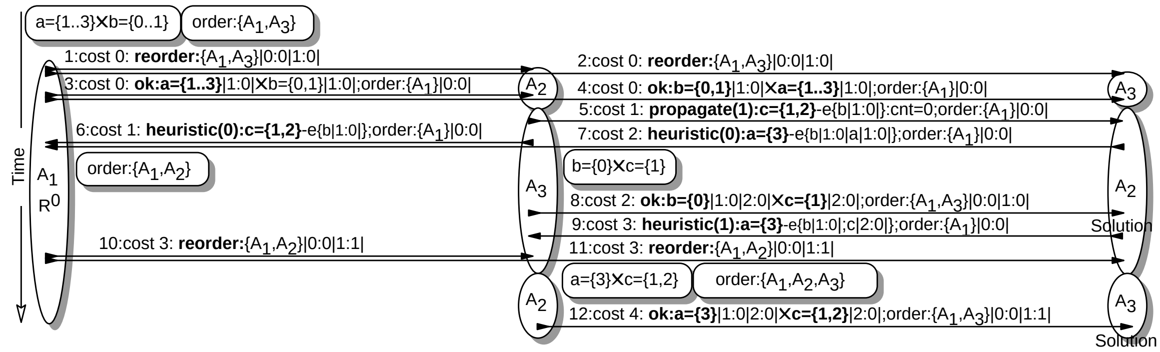


Figure 6: *A simple run of consistency with asynchronous reordering.* `addlink` *messages are redundant for this heuristic.*

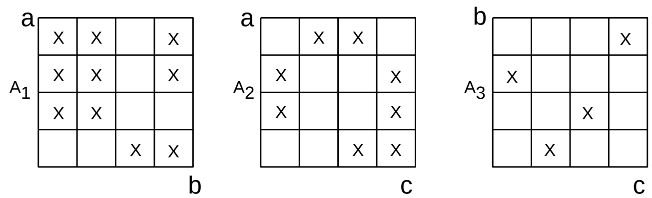


Figure 5: *A problem with 3 agents.*

2 is discarded at A_3 when the decision in message 10 arrives with a newer history.

`heuristic` messages may only be sent by an agent to R^k in a bounded time after having received an `ok` from $A^j, j{\leq}k$ or a `propagate` message of level $k'', k''{\leq}k{+}1$.

`addlink` messages are no longer sent only to predecessors, but to successors as well. When the heuristic in the example of Figure 6 is used, `addlink` messages are redundant.

In our example (Figure 5), we consider the case where agents announce all the interested reordering leaders of all the labels they send. In Figure 6, the reordering leader of level k, R^k corresponds to the agent A^k. R^0 corresponds to the agent A_1. The volume of A_i is estimated as the product of the size of domains for all variables in vars(A_i). R^i maintains for all variables and for the levels k, $k{\leq}i{+}1$ similar structures to those of $MAS_{(+,+,-)}$ at agent A^u_j for vars(A^u_j) and levels t, $t{\leq}u$. The space complexity remains the same as with $MAS_{(+,+,-)}$.

Proposition 5 *MAS is correct, complete and terminates.*

Proof summary. **Completeness** and **correctness** are proved using the same reasoning as for Proposition 1.

No infinite loop: We prove by induction on increasing i the Property 3 where quiescence now also refers to `reorder` messages and to any `propagate` message of level $k{\leq}i{+}1$. Let all agents $A^j, j{<}i$ reach quiescence before time t^{i-1}. Reasoning as for property 3, $\exists t^i_h$ after which no propagate of level k, $k{\leq}i$ and therefore no heuristic message toward any R^u, $u{<}i$ is exchanged. Then, R^u becomes fixed, receives no message, and announces its last order before a time t^i_r. After t^i_r the identity of $A^k, k{\leq}i$ is fixed. A^i receives the last new aggregate or order at time t^i_o. Reasoning as in property 3 for A^i and R^i, it follows that after t^i_o, within finite time, the agent A^i either finds a solution and quiescence or exhausts its possibilities and sends a valid explicit nogood to somebody. From here, the induction step is proven immediately. R^0 is fixed in finite time $t_h + t_r$ and the property is true for the empty set. The termination results by induction on i. □

We see in Figure 6 that the solution detection method proposed for AAS can detect solutions before quiescence. However, if all the solutions are required, that solution detection algorithm may detect several times the same solutions. A quiescence detection algorithm can avoid this inconvenient.

Property 5 *If all the aggregates proposed by reordered agents are invalidated on the receipt of a new ordering, the strength of the consistency maintained by DC(A) in $MAS_{(+,+,+)}$ equals the strength of the one maintained by A in centralized algorithms.*

Proof. This holds since a label is invalidated by one agent iff it is invalidated by all agents □

4.4. HARD FAILURES

Hard failures are failures where the agents have acted correctly until a certain moment when they disappear. The failure of agents can be usually detected with failure detectors. The failure detectors are not always safe. However, in this section we consider only the case when safe failure detectors are available. We present now an algorithm for recovering of agents in MAS from hard failures. The new messages used by this algorithm are **recovery** messages telling the receiver that the sender needs **recovery-data**. **recovery-data** messages are sent in answer to **recovery** messages and transport information that the receiver should know.

In the case of hard failures, the agents that do not crash can continue the inference work on all consistency levels. The search for exhausting the space allowed by the proposals of the crashed agents also continues whenever no answer to nogood messages is waited from them. On recovery, a crashed agent A_i can easily reintegrate itself in the distributed computation. It has to request (by **recovery** messages) from any other agent A_k: the last aggregates, order and labels that A_i and A_k have built respectively sent at all levels $l \leq u$. u is the position known by A_k for A_i in the moment when it sends the recovery data. All this data is sent to A_i via **recovery-data** messages. A_k can clear all the add-links towards A_i and has to request again all add-links from A_i. During recovery, the recovering agent A_i must process only the incoming **recovery** and **recovery-data** messages. All other messages have to be stored and processed after the recovery stage.

After recovery data is obtained from all correct agents, the labels, order, aggregates and add-link messages generated by

A_i are sent back to all interested agents that did not reported them. Counters that are not restored can be safely reinitialized to 0. Any message m received by A_k from A_i after the n-th recovery request of A_i is answered, is discarded if m was sent before the n-th recovery request of A_i. If we don't have FIFO channels, this condition has to be ensured (e.g. by attaching to all messages a counter of the crashes of the sender).

Proposition 6 *After the recovery data is received, the recovered agents are fully integrated in search and their instantiation, view and counters are coherent with the other agents. The consistency labels they get are also coherent with the other correct agents, ensuring that the maximum degree of consistency allowed by DC is reached at quiescence if all crashed agents recover.*

Proof sketch. It is only the last labels, the valid aggregates and the last add-links known by other agents that need to be known. All these are received in the recovery data. If an agent knows something newer than the others, the only case when it will not send that information is if it crashes before. But in this case the crashed agent forgets that information. All the received information is then broadcasted back so that everybody that is interested knows it.

Another agent A_j may be simultaneously in the recovery stage. A_j does not yet know all its own generated data but will broadcast it at the end of its recovery. If the last aggregates and labels sent by A_i before the crash were sent successfully to all other alive agents and were not meanwhile invalidated by other agents, then that data is received back for recovery and the situation is coherently reestablished.

However, data may have been successfully sent to only a subset of the interested agents before the crash. If any agent has received it correctly, that information will be received back on recovery if it was not invalidated. The last sent data is then resent, if valid, to all the other agents so that the views of the correct agents become coherent. □

Even if the search continues in a clean manner after crashed agents recover, the explicit and conflict-list nogoods that they lose due to the crash cannot be automatically recovered from others. The lost is important especially if the agent had a high priority. Therefore the agents need to make backups of their explicit and conflict-list nogoods so that they are enabled to recover as much as possible of the already spanned search space. If nogoods in backups contain newer or older aggregates than the received recovery data, the corresponding nogoods have to be invalidated. The validity of an aggregate in the nogoods in backups can be checked by testing whether their set of values is the same with the set in some valid aggregate received at recovery for the same variable. If this holds, than the history of the valid aggregate substitutes the one of the recovered one. If no valid aggregate can be found for some aggregate a, then a and the nogood that contains a are invalidated. The set of messages required in order to reconnect to broker, to get the addresses of the other agents and to request recovery data are obvious and are not presented here.

4.5. ONE SHOT LINKS

In MAS, the links added dynamically during the search become redundant when nogoods are discarded. It is known that the nogoods are invalidated on the first change of the history of an aggregate. A convention can be established such that an agent eliminates another agent from its dynamically added outgoing-links immediately after an aggregate with a new history is broadcasted for the variable defining that link. However, this strategy is acceptable only for problems where the link does not reappear frequently.

5. DOMAIN SPLITTING

One of the ways of approaching large problems is by successively solving increasingly detailed abstractions of it, until the original problem is solved. In the following sections we show how distributed search can be performed asynchronously and in parallel over all abstractions. The approach is inspired from the techniques used in numerical CSPs.

The approaches that enumerate values in domains, feasible tuples in constraints or aggregates of such feasible tuples are called here *enumeration techniques*. The inherent disadvantage of enumeration techniques is the dependence of the efficiency of the search for finding a solution on the order of the elements. Their efficiency for proving unsatisfiability is also dependent on the size of the problems. The heuristics for value reordering theoretically can help, but are seldom sufficiently informed. For hard problems with large domains and weak value reordering heuristics, the enumeration techniques are therefore hopeless. However, when the domains are "naturally" ordered, which means that the constraints have a high *density of the current value order* [19], dichotomous techniques that recursively split domains into halves often improve efficiency on difficult problems. The best known examples consist of the numerical constraint satisfaction problems that can be solved when dichotomous techniques are combined with some kind of bound consistency [22].

5.1. REPLICAS-BASED DISTRIBUTED CSP

Implementing asynchronous dichotomous behavior in the distributed context of $MAS_{(+,+,+)}$ requires special treatment. The main impediment comes from the fact that the number of consistency levels in $MAS_{(+,+,+)}$ equals the number of agents. In $MAS_{(+,+,+)}$ the constraints are private to agents and simple dichotomous behavior would hinder the agents with high priority from requiring the satisfaction of their constraints. To overcome this problem, each agent A_i owning private constraints can be represented in the search by a set of k_i+1 replicas denoted $A_{i_0}, ..., A_{i_{k_i}}$. Assume that at a given moment t, $A_{i_{o_t}}$ is ordered after all other replicas of A_i. Then the replica $A_{i_{o_t}}$ is at time t called *checking replica of* A_i and has the goal usually taken by A_i in $MAS_{(+,+,+)}$, namely to make sure that the subproblem proposed to its followers is consistent with all the private constraints of A_i. All the *artificial replicas* (namely agents that are not *checking replica* of any agent) have to behave according to conservative splitting and reduction operators, and send messages consistent with the $MAS_{(+,+,+)}$ protocol. Each artificial replica of A_i tries to enforce a set of constraints that is a relaxation of the constraints of A_i.

$$\cup_{0\leq u\leq k_i} CSP(A_{i_u}) = CSP(A_i)$$

Typically (e.g. this is not the case of dichotomous splitting, but it is the case for the other abstraction techniques presented later), a replica $A_{i_j}^{t_{i_j}}$ of A_i is useful

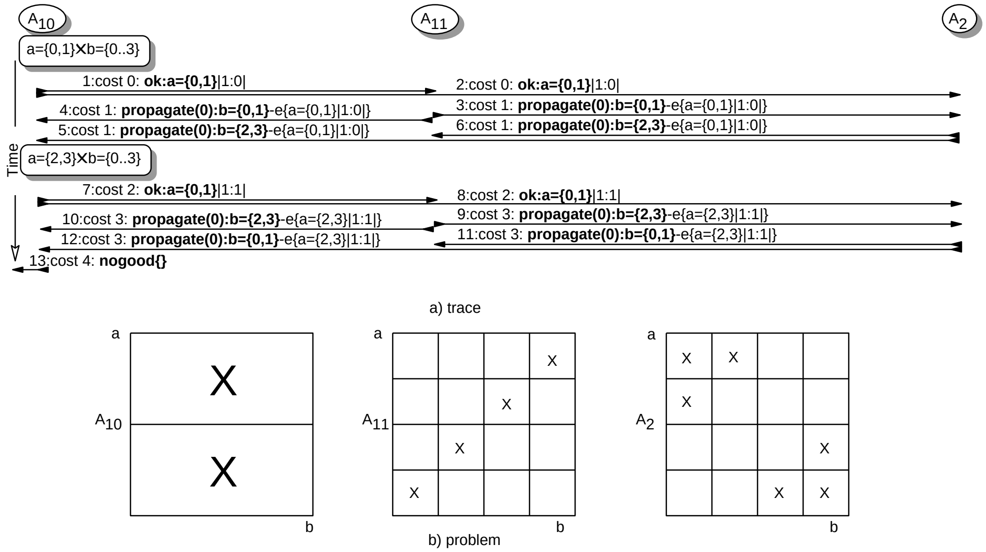


Figure 7: *Domain Splitting.*

at moment t if, $\cup_{u,t_{i_u}<t_{i_j}} CSP(A_{i_u}^{t_{i_u}})$ is a relaxation of $\cup_{u,t_{i_u}\leq t_{i_j}} CSP(A_{i_u}^{t_{i_u}})$.

We call the framework defined by this architecture, Replicas-based Distributed CSP (RDisCSP) and is shown further to allow for many interesting instances and algorithms. The fact that artificial replicas belong to some participant rather than being neutral was chosen in order to give the agents a leverage in the coordination of the search.

5.2. ASYNCHRONOUS DOMAIN SPLITTING

In Figure 7 is given a small example where domain splitting is used to solve a problem with two agents. The agent A_1 has two replicas A_{1_0} and A_{1_1}. A_{1_0} proposes the halves of the domain of variable a. The search space is exhausted with 4 sequential messages. Without domain splitting the minimal number of sequential messages is 6. In the given trace, we consider that consistency nogoods are sent to all agents, according to the heuristic used in the example of Figure 6. Therefore, the consistency-based domain wipe-out can be detected by the agent generating the faulty level. Otherwise, nogood messages would be sent by the agents A_{1_1} and A_2 to the agent A_{1_0}.

A second aspect is that the number of consistency levels depends on the pruning computed by bound consistency during search and cannot be safely estimated in advance. A straightforward way to solve this problem is by using a number of *artificial agents* equal with the maximal number of consistency levels ($\sum_{i\in X}\log_2 D_i$). However, this is a rather expensive solution for large problems. Two alternative solutions are studied here. The first alternative, **static agent configuration**, consists of allowing existing *artificial agents* to abandon their current proposal and generate new splits. Some nogoods are lost due to this procedure. The second one, **dynamic replica spawning**, consists of dynamically spawning new replicas whenever the current search space of the *checking replicas* is not contained in the solution space.

The role of the artificial agents is to make sure that the volume of the search space that they propose is (with a certain tolerance) half of the volume of the search space that they were proposed. Namely, checking replicas must generate aggregates such that:

$$|\text{volume}(A_i)/2\text{-volume}(\text{known}(A_i))| < \text{threshold}.$$

Whenever this condition is not respected due to modifications brought to the domains by distributed consistency, the artificial agents will generate a new proposal. When a proposal of an artificial agents is refused by some *nogood* message, half of the other half (or the other half, when the maximum resolution –an element of $\mathcal{F}$ for numerical constraints– is reached) of the search space is proposed. When its whole search space is similarly refused, an artificial agent sends an explicit nogood generated by inference from the received nogoods, the nogood entailed by the view and those entailed by consistency. Whenever the whole search space of an *artificial agent* A_{i_k} is feasible for the corresponding agent A_i, A_{i_k} does not need to split its search space (by generating new proposal). The feasibility test can be done for continuous domains in Numerical CSPs using the technique of Benhamou&Goualard employed in [22]. The variable that will be split first can be chosen to be the one with biggest impact on the future search (see [19, 22]). The heuristic in [19] is available since the agents declare their interest on variables. Alternatively to the dichotomous one, splitting techniques can be based on constraint analysis. Namely, each artificial agent A_{i_k} can choose a constraint of A_i and devise a split according to heuristics based on the constraint structure as described in [22]. The artificial agents need not generate consistency nogoods since their work would be redundant to the one of the checking replicas which enforce all the corresponding constraints.

A *replica spawning method* is described in Annexes since we conjecture that for large problems, they generate a large number of agents and increase the number of exchanged messages. Therefore, we only describe in this section *static agent configuration methods*.

5.3. STATIC AGENT CONFIGURATION

In order to avoid generating new agents, a solution is to let an agent A_{i_k} act for several replicas of A_i. Messages are then sent in only one transmission to all the replicas for which the receiver agent acts. In terms of internal structures, letting an existing agent act for a new replica amounts to storing new consistency levels, an additional CL and an additional order.

We have used the described $MAS_{(+,+,+)}$ protocol in order to obtain asynchronous domain splitting. The search starts with a predefined number of replicas for each agent (e.g. estimated by previous negotiation on the declared size of the internal problems of the agents). A timeout is monitored by artificial replicas (e.g. by the last artificial replica of each agent). On such a timeout, that agent abandons its current proposal and generates a new proposal by splitting again a variable. Therefore on each timeout, $volume(known(A_i))$ is divided by two. Whenever a message is received, the counter that generates the timeout is reset. If after receiving and processing a message, $volume(A_i)$ is modified and

$$|volume(known(A_i)) - volume(A_i)/2| > threshold,$$

then the current proposal is abandoned and a new proposal is generated.

6. ABSTRACTIONS

The RDisCSP framework can be used in a different context than dichotomous splitting. The domain splitting techniques are proven as being efficient for numeric constraints. In a more general view, the domain splitting can be viewed as a special type of global abstraction for ordered domains, abstraction suggesting that one half of a domain behaves differently from the other half. For other types of constraints, different abstractions can be used within the same RDisCSP framework. Two such algorithms are presented in the rest of this section.

6.1. CONSTRAINTS SPLITTING

The private problem of an agent can often be represented as a set of constraints. Let us consider that CSP(A_i) can be split in k_i+1 problems CSP(A_{i_0}),...,CSP($A_{i_{k_i}}$) (e.g. corresponding to variables or constraints of CSP(A_i)). A_i can therefore have k_i+1 replicas, each of them dealing with a subproblem of CSP(A_i) in such a way that $\forall j$, CSP(A_{i_j})$\subseteq$CSP(A_i) and $\cup_{j\in\overline{0,k_i}}$CSP($A_{i_j}$)=CSP($A_i$). We mention that such a distribution makes sure that any solution space accepted by all the agents A_{i_j} is acceptable to A_i. MAS solves the problem modeled this way, and the additional consistency levels introduced by replicas are a means for reusing nogoods at backtracking.

6.2. TUPLES CLUSTERING METHOD

Let us assume that each agent owns exactly one constraint (either obtained by the composition of several constraints, or by problem splitting with replicas as in section 6.1). Let us now assume that for each agent A_i owning exactly one constraint we use k_i+1 replicas, ordered A_{i_0},...,$A_{i_{k_i}}$, such that CSP($A_{i_{k_i}}$)=CSP(A_i) while any CSP($A_{i_{j,j<k_i}}$) is a relaxation of CSP(A_i). For a constraint with infinite domains, such relaxations can be built by merging elements of some rough outer covering. For any constraint with finite domains, such a relaxation can be obtained by means of *clustering methods* with *seeds* usually employed in automated classification techniques (e.g. speech recognition) [5]. The technique consists of distributing a predefined number of points (seeds) uniformly in the search space defined by the variables in the constraint. Iteratively, each feasible tuples is associated with the closest seed (Euclidean distance) and then each seed is moved to the center of mass for the partition that the seed i defines on feasible tuples. When this iteration converges, for each seed i we compute the minimal box B_i that encloses the whole partition it defines. B_i is given by the minimal and maximal values of each variable over the feasible tuples associated to the seed i. Therefore we get a relaxation composed of a number of boxes at most equal with the number of initial seeds. These boxes are not always disjoint.

Proposition 7 *A constraint obtained by clustering can be decomposed in maximum ks^2 disjoint boxes (covers) where k is the arity of the constraint and s is the number of seeds.*

Proof. Each box needs at least one *cover*. For each already covered box, a not yet covered box may need $2k-1$ additional disjoint boxes to be covered. The total number is therefore $s + (2k-1)(s-1)s/2 = ks^2 + s(1-k) + s(1-s)/2 \leq ks^2, \forall s, k \geq 1$. □

Each CSP($A_{i_{j,j<k_i}}$) can be built with a number of seeds that is a strictly monotonically ascending function on j. This heuristically yields an increasing detail of the abstraction. One of the problems that appear with **tuples clustering** is that with a small number of seeds, random constraints with high tightness may become total constraints (completely feasible). The covering boxes can be computed either statically or dynamically. Statically computed covering boxes can either be used without modification or they can be reaggregated if that is possible when given proposals are made. In this last case, care has to be taken to avoid reducing the number of boxes. A second problem is that when many seeds fall out during the clustering, it is possible to obtain the same abstraction in different replicas. Figure 8 shows an example where **tuples clustering** is used with one replica for A_1.

Both the **constraint splitting** and the **tuples clustering method** are complete and correct since there exist equivalent problems for which they are identical to $MAS_{(+,+,+)}$. The overhead consists of the communication with the new agents and can be reduced when the replicas are explicitly colocated. The **tuples clustering** is a special case of domain splitting.

7. CONCLUSIONS

Many of the contributions of this paper are enabled by the definition of a new simple framework called Replicas-based Distributed CSP (RDisCSP). It consists of distributing the initial problem of an agent to a set of logical agents. These logical agents can either be represented during search by a physical agent each, or the role of several logical agents can be played by the same physical agent. We have presented three abstraction techniques for RDisCSPs. They are integrated in MAS, a new distributed search protocol, which allows

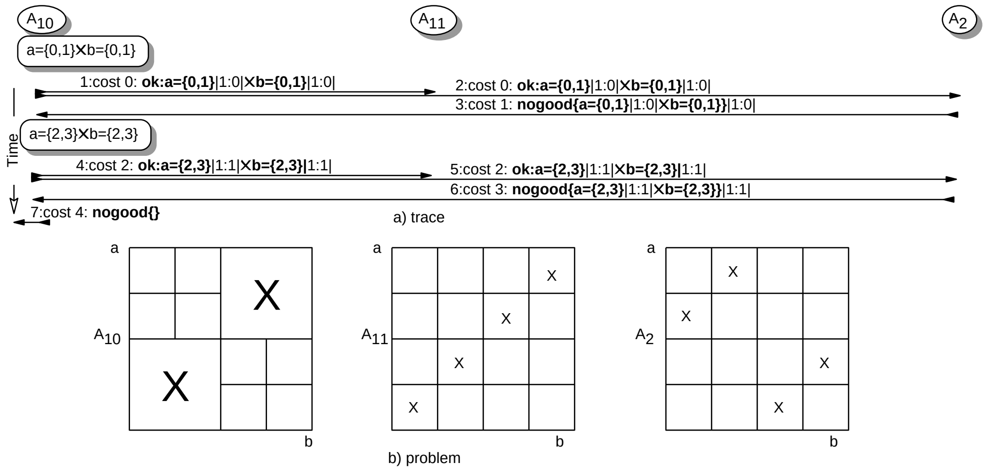


Figure 8: *Tuples Clustering Method.*

for maintaining distributed consistency with a high degree of parallelism and allows for asynchronous dynamic agent reordering. MAS is a generalization of the best-known distributed complete search algorithms with polynomial space requirements. In fact MAS is an extension of MHDC [20] with dynamic reordering without losing completeness and space complexity. This paper also describes MAS in details, describing its techniques for crash recovery and consistency nogood announcement. Additional results are given concerning the implications of reusing instantiations and corresponding nogoods across reordering and a couple of new enabled reordering heuristics (random, least volume, ABT-AWC) are described. RDisCSP helps to integrate new abstraction schema in asynchronous search. Several alternatives for implementing MAS for RDisCSP have been detailed. The *Domain Splitting* is a successful abstraction famous for centralized numerical constraints. *Tuples Clustering* is a new abstraction technique inspired from automatic classification methods. It seems appropriate for certain discrete constraints. *Constraints Splitting* is an abstraction that fits naturally for MAS. The proposed abstraction techniques have been implemented and partially tested using versions of MAS for RDisCSP with a static agent configuration.

REFERENCES

[1] A. Armstrong and E. F. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence IJCAI97*. IJCAI, 97.

[2] L. Baptista and J. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Proceedings of CP'2000*, number 1894 in LNCS, pages 489–494. Springer Verlag, 2000.

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

[4] C. Bliek. Generalizing partial order and dynamic backtracking. In *AAAI-98 Proceedings*, pages 319–325, Madison,Wisconsin, July 98. AAAI.

[5] R. Boite, H. Bourlard, T. Dutoit, J. Hancq, and H. Leich. *Traitement de la Parole*. Presses Polytechniques Universitaires Romandes, 2000.

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

[7] Z. Collin, R. Dechter, and S. Katz. Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*, 2000. UCI Technical Report, TR 92-67.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W.H.Freeman&Co, 1979.

[9] GlobSol. http://www.mscs.mu.edu/˜globso.

[10] Y. Hamadi and C. Bessière. Backtracking in distributed constraint networks. In *ECAI'98*, pages 219–223, 98.

[11] W. Havens. Nogood caching for multiagent backtrack search. In *Proc. AAAI'97 Constraints and Agents Workshop*, '97.

[12] D. Kumar. A class of termination detection algorithms for distributed computations. In N. Maheshwari, editor, *5th Conf. on Foundations of Software Technology and Theoretical Computer Science*, number 206 in LNCS, pages 73–100, New Delhi, 1985. Springer.

[13] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, (2):161–175, 87.

[14] P. Meseguer and M. A. Jiménez. Distributed forward checking. In *Proceedings of the International Workshop on Distributed Constraint Satisfaction*. CP'00, 2000.

[15] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings ECAI-94*, pages 125–129, 94.

[16] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Search techniques for CSPs with ordered domains. Technical Report #99/313, EPFL, May 1999.

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

[18] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with private constraints. In *Proc. of AA2000*, pages 177–178, Barcelona, June 2000.

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

[20] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Maintaining hierarchical distributed consistency. In *Proceedings of the International Workshop on Distributed Constraint Satisfaction*. CP'00, 2000.

[21] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous consistency maintenance with reordering. Technical Report #01/360, EPFL, March 2001.

[22] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Search techniques for non-liniar constraint satisfaction problems with inequalities. In *Proc. of AI2001*, Otawa, June 2001.

[23] G. Solotorevsky, E. Gudes, and A. Meisels. Distributed Constraint Satisfaction Problems - a model and application. Preprint: http://www.cs.bgu.ac.il/~am/papers.html, Oct 97.

[24] 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.

[25] P. Van Hentenryck. A gentle introduction to Numerica. *AI*, 103:209–235, 98.

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

[27] M. Yokoo. *Distributed Constraint Satisfaction*. Springer, 01.

[28] M. Yokoo. Constraint relaxation in distributed constraint satisfaction problem. In *ICDCS'93*, pages 56–63, June 93.

[29] M. Yokoo. Asynchronous weak-commitment search for solving large-scale distributed constraint satisfaction problems. In *Proc. of 1st ICMAS*, pages 467–318, 95.

[30] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *ICDCS'92*, pages 614–621, June 92.

[31] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. on Knowledge and Data Engineering*, 10(5):673–685, 98.

[32] M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of 3rd ICMAS'98*, pages 372–379, 1998.

[33] 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, 91.

8. ACKNOWLEDGEMENTS

This research is funded by the Swiss NSF, project number 21-52462.97. An initial draft of this paper was proofread by Dr. Marc Horisberger and by Dr. Melody Silaghi.

9. ANNEXES

9.1. DYNAMIC REPLICA SPAWNING

Here we present a method for dynamically spawning replicas. The next new messages are introduced:

- **spawn?** messages: are send from broker at quiescence to ask checking replicas if they are satisfied with the current solutions.
- **replicate** messages: are send from checking replicas to broker as answer to spawn? messages. They signal the existence of infeasible tuples in the newest proposals known by the sender and have as parameter the address of a new replica.
- **satisfied** messages: are send from checking replicas to broker in answer to spawn? messages. They signal that the space defined by the newest proposals known by the sender is feasible.
- **spawn** messages are sent by the broker to all agents and announce the insertion of the new replicas in search.

Dynamic replica spawning (DRS) is achieved by running $MAS_{(+,+,+)}$ in parallel with a solution detection and a termination/quiescence detection process. For this approach, the *checking replicas* need not create corresponding consistency levels by enumerating proposals, but only generate `accepted` messages when the whole space that was proposed to them is feasible for their constraints. For, the solution detection algorithm, the same technique can be used as for AAS. The termination/quiescence detection uses counters similarly with synchronous MDC [20, 21] as presented later. When the termination detection algorithm signals silence on the network, the broker sends **spawn?** messages to the *checking replicas*. The *checking replicas* that are not satisfied by the current domains send a **replicate** message to the broker, while the others send a **satisfied** message. Each **replicate** message contains the address of a new replica of the sender. The broker will then send a **spawn** message to all the agents, establishing a priority for the new replicas between the old *artificial replicas* and the *checking replicas*. The *artificial replicas* treat **spawn** messages as `addlink()` for the common variables. The *checking replicas* initialize the new corresponding consistency levels. To maintain coherence of the views and consistency levels of the agents, they must not restart the search before the broker acknowledges them that everybody has received the **spawn** message.[8]

The termination is ensured by the fact that whenever all agents answer with satisfied, it is guaranteed that a solution will be detected in finite time.

An algorithm with **static agent configuration** can also be obtained from DRS when a neutral agent A_r acts as a replica and is positioned between artificial and checking replicas. Whenever quiescence is detected, no agent is added, rather A_r is requested to split a domain for a variable of some agent that has sent a **replicate** message. The splitting is conservative, so that the eliminated space is tried on backtracking. It is no longer the broker that detects quiescence but A_r. The disadvantage consists in the confidence in the fairness of A_r.

9.2. TERMINATION DETECTION

In the previous algorithms (as in synchronous MDC) we need to detect the quiescence of the agents. In previous work, distributed CSP algorithms have detected quiescence using the techniques for distributed snapshot presented in [6]. That algorithms have the characteristic that termination tests need to be initialized by agents suspecting quiescence. If the interest is to detect quiescence as quickly as possible, other methods are more appropriate.

We present here a termination detection method which only requests one sequential message after the quiescence of the monitored protocol. This technique is well adapted to consistency maintenance procedures for distributed CSPs. Related termination detection protocols are already known to the distributed systems community [12, 13] and their proof also applies here.

Each agent A_i maintains a counter $c_{i,j}$ for outgoing messages towards each other agent A_j and a counter $c^{j,i}$ for incoming messages from each agent A_j. When A_i becomes idle, this counters are sent at any modification to the agent T checking the termination. When T detects that $c_{i,j} = c^{i,j}$ for all i and j, then T can announce termination. Since the counters can only increase, there is no need of time stamps or FIFO channels since the highest counter value is always the newest. Several termination detection stages can succeed synchronously one after another, as happens with the successive consistency rounds in synchronous MDC. We may want to distinguish them. A simple flag with two values that switches at the start of a new round has to be attached to each message. If each agent receives a message in each round, this is sufficient to announce the agents that a new stage begins and that the local counters must be reset to 0. Otherwise the flag has to be replaced by an increasing counter of the stages.

The termination detection messages need not be sent directly to T. Each agent keeps a list Lc of counters that it has received. Lc contains only the last received pair of vectors of counters for each agent. If A_i has to send a messages to a set of agents A, it chooses an agent A_j in A. A_i attaches to the message sent to A_j the modified counters in Lc, received from other agents, as well as and its own updated counters. Then A_i clears Lc. If A was empty after A_i has received some messages, it sends its counters and its Lc to T and also clears Lc. When Lc is large enough to fill the payload of a message, the network charge is reduced if an agent can sent the modified counters to T rather than sending them further to other agents. The size of Lc when the behavior has to change is a function of the size $|m|$ of the messages sent to agents in A and also depends on the maximum unfragmented payload (MTU) that can be sent to T. The threshold should be $|Lc| = (MTU - |m| mod MTU) mod MTU$.

[8] Alternatively, an ID of the last spawn message can be attached to any message. No message is then processed before the corresponding spawn is received.