How to get AAS' when you just have AAS

Marius-Călin Silaghi[⋆]

Florida Institute of Technology (FIT),
Melbourne, Florida
msilaghi@cs.fit.edu

Abstract. It is well known that the development of a new complex CSP algorithm from scratch is difficult and error prone. Asynchronous Aggregation Search (AAS) [2] is a family of algorithms whose efficient implementation involves complex features like: dynamic detection of interchangeabilities, interruptibility of the backtracking with switch of contexts, and management of more or less intelligent nogood stores. Moreover, all these may have different behavior in different states of local state machines.

AAS has different related versions (AAS0, AAS1, AAS2) that can be obtained with small effort from one another. A remarkable related family of algorithms is the Asynchronous Aggregation Search without signatures (AAS') [2]. AAS' shares several characteristics with AAS, allowing for similar aggregations, but the semantic of the information is complementary. Namely, the costly operations are triggered by changes of local instantiations rather than by changes of composed instantiations.

Here I show how, despite the quite deep differences between the basic operations of AAS and AAS', one can perfectly implement AAS' based on any AAS implementation, by a 'trick' in problem formulation and local rules.

1 Introduction

It is well known that the development of a new complex CSP algorithm from scratch is difficult and error prone. Asynchronous Aggregation Search (AAS) [2] is a family of algorithms whose efficient implementation involves complex features like: dynamic detection of interchangeabilities, interruptibility of the backtracking with switch of contexts, and management of more or less intelligent nogood stores. Moreover, all these may have different behavior in different states of local state machines.

AAS has different related versions (AAS0, AAS1, AAS2) that can be obtained with small effort from one another. A remarkable related family of algorithms is the Asynchronous Aggregation Search without signatures (AAS') [2]. AAS' shares several characteristics with AAS, allowing for similar aggregations, but the semantic of the information is complementary. Namely, the costly operations are

[⋆] This work started while the author was working at EPFL, supported by the Swiss National Science Foundation project number 21-52462.97.

triggered by changes of local instantiations rather than by changes of composed instantiations.

Here I show how, despite the quite deep differences between the basic operations of AAS and AAS', one can perfectly implement AAS' based on any AAS implementation, by a 'trick' in problem formulation and rules for local reasoning. It is remarkable to note that AAS' can be obtained even from simplified versions of AAS.

2 Problem

We consider naturally distributed constraint satisfaction problems involving privacy requirements (e.g. for solving auctions [2]).

Definition 1 (DisCSP). *A* **DisCSP** *(A,V,D,C) is given by a set of agents $A_1, ..., A_n$ where each agent A_i wants to enforce some private constraint C_i in C. The set of shared variables involved in C_i is V_i (from V). The agents want instantiation of the variables in V_i in the corresponding domains D_i.*

Such a problem can be solved by centralizing it into a trusted party, operation that translates it into a typical CSP. [1]

2.1 (First) Solution of a CSP

Here are presented SCSP, the simple and formal mathematical equations for defining the first solution to a CSP [2]. Imagine we want to solve a CSP $P = (X, C, D)$ where X is a set of variables $x_1, x_2, ...x_n$, C is a set of constraints and D a set of domains for X. The domain of x_i is D_i, whose values are $v^i_1, v^i_2, ..., v^i_{|D_i|}$.

Definition 2 (first solution). *The first solution of a CSP given a total order on its variables and a total order on its values is the first among solution tuples when these are ordered lexicographically.*

Let *gconsistent(P)* be a function such that:

$$gconsistent(P) = \begin{cases} 1 \text{ if } P \text{ has a solution} \\ 0 \text{ if } P \text{ is infeasible} \end{cases}$$

A simple example is shown later.

Consider now a set of functions: $f_1, f_2, ..., f_n$, $f_i : CSP \rightarrow \mathbb{N}$, such that each f_i will return the index of the value of x_i in the first solution, or 0 if no solution exists.

$$f_i(P) = \begin{cases} k \text{ if } P \text{ has the first solution for } x_i = v^i_k \\ 0 \text{ if } P \text{ has no solution} \end{cases}$$

[1] Once the constraints of different agents are securely shared, securely solving the DisCSP reduces to securely solving the CSP obtained by the union of all constraints enforced by the different agents.

A simple example is also shown later.

Let us first define the functions $g_{i,1}, g_{i,2}, ..., g_{i,|D_i|}$. $g_{i,j} : CSP \rightarrow \{0,1\}$.

$$g_{i,j}(P) = \begin{cases} 1 \text{ if } P \text{ has a first solution for } x_i = v^i_j \\ 0 \text{ if } P \text{ is infeasible for } x_i = v^i_j \end{cases}$$

A simple implementation is:

$$g_{i,j}(P) = gconsistent(P \cup \{x_i = v^i_j\} \cup_{k<i} (x_k = v^k_{f_k(P)})) \tag{1}$$

where the union of a problem and some constraints yields the problem tighten by enforcing the additional constraints.

Now we can define the next simple implementation for the functions f_i.

$$f_j(P) = \sum_{i=1}^{|D_j|} i * (g_{j,i}(P) * \prod_{k<i} (1 - g_{j,k}(P))) \tag{2}$$

In [2] we show that the functions g and f given by Equations 1 and 2 correspond to their definition.

2.2 Expensive implementation of the satisfiability function

Let $SS(P)$ be the ordered set of all tuples in the cross-product of the domains of P. Each constraint c in the set of constraints C is a function, $c : SS(P) \rightarrow \{0,1\}$. The secret parameters of the distributions are the various values $c(t)$ where t is a tuple. Let us define the function p, $p : SS(P) \rightarrow \{0,1\}$, defined as $p(t) = \prod_{c \in C} c(t)$.

$$gconsistent(P) = \sum_{t_i \in SS(P)} (p(t_i) \prod_{k<i} (1 - p(t_k)))$$

As shown in [2], given the previous definitions of the functions p, $gconsistent()$, $g_{i,j}$, and f_i, and a (Dis)CSP P, the vector $\langle v^i_{f_i(P)} \rangle$ defines a solution of P (the first one).

Remark 1. The computation of the vector $\langle f_i(P) \rangle$ requires only additions and multiplications and can be easily compiled unto a secure protocol.

Remark 2. Actually whenever an element of the vector $\langle f_i(P) \rangle$ is 0, the computation can be stopped since P is infeasible.

Remark 3. Submitted shares of an input value, v, of a tuple in a constraint can be verified by checking that $v(v-1)=0$.

Due to the drawbacks of this approach (cost, insecurity to colluders) we consider it important to study the alternatives based on constructive search, two important ones being compared next, AAS vs. AAS'.

3 Framework

Both AAS and AAS' are designed to solve naturally distributed constraint satisfaction problems. Their framework is not restricted to defining the search space and the solution space of the problems, but also agent competence [2]. The following definition imposes some restrictions on the operations that agents can perform.

Definition 3 (DisCSP with modifiers). *A* **DisCSP** *(A,V,D,M,C) is given by a set of agents $A_1, ..., A_n$ where each agent A_i wants to enforce some private constraint C_i.*

The set of shared variables involved in C_i is V_i. The agents negotiate the instantiation of the variables in V_i in the corresponding domains D_i by either revealing conflicts or by proposing instantiations for a subset M_i of V_i.

Any variable x_k that is involved with an existential quantifier in at least one constraint, has to be in the M_i set of at least one agent A_i. The agents want to agree on instantiations such that all the predicates are satisfied.

This definition generalizes most of the definitions known to the author. Algorithms defined for *DisCSPs with modifiers* can then be correctly compared with algorithms developed for any sub-framework.

Definition 4. *The set of modifiers of x_i, M_i^s, is the set of agents having x_i in their M_i.*

$$M_i^s = \{A_k | x_i \in M_k\}$$

Remark 4 (external constraint). A non-unary constraint in a *DisCSP with modifiers* is called external if at least two of the variables it involves, x_i and x_j, are such that $|M_i^s \cup M_j^s| > 1$.

One can similarly define internal constraints.

Remark 5 (internal constraint). A non-unary constraint in a *DisCSP with modifiers* is called internal if any two of the variables it involves, x_i and x_j, are such that $|M_i^s \cup M_j^s| = 1$.

It is worth to be noticed that the newly introduced notions of internal/external constraints generalize their most common usage.

To enforce C in AAS, A_i:

- has to announce at beginning that it wants to modify all the shared existentially quantified 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 existentially quantified variables in C that A_i does not want to modify itself.

An agent does not need to enforce a constraint, C, that it has when it knows that another agent with higher position enforces C.

4 Asynchronous Aggregation Search (AAS)

AAS imposes a total order on the agents participating in the computation. Lower positioned agents in this order have a higher priority than higher positioned ones.

The main idea behind AAS is to allow the agents to aggregate several assignments into one proposal. Agents refine proposals of higher priority agents. To allow for this flexibility, AAS asks agents to tag their proposals with a signature. The signature defines the validity of the proposal as function of proposals of higher priority agents. These elements are detailed in the rest of this section.

4.1 Proposals on shared variables

All the agents can *aggregate* several assignments for a variable x_i into one proposal 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.

Moreover:

Remark 6. More than one agent enforcing initial constraints on x_i may not announce at beginning that they want to make proposals with assignments for x_i. Let them be ordered in such a way that the first of them has position k_i. Let $S_i^{k_i}$ be the set of agents that enforce initial constraints on x_i and that are positioned before k_i. All the agents in $S_i^{k_i}$, excepting the last positioned of them, are allowed to aggregate several assignments in one proposal (aggregate).

Several branches of the search are therefore aggregated.

4.2 Signatures

Signatures enable several agents to make proposals on the same variable. The real problem comes from the fact that the different agents should be able to coherently agree on a common proposal.

A decision can be taken by giving different priority to distinct agents. However, this would simply lead to the loss of any proposal coming from lower priority agents and the problem is not solved. The notion of proposal was redefined in a way that allows several proposals to be composed.

Remark 7. We say that the relation "p1 complies to p2" between two proposals is *intended* if the agent generating p1 explicitly states this relation.

This is related to the compliance of decisions from lower ranked authorities to decisions of higher ranked ones, as taken in human societies. There, an ordnance is often specified as complying to a certain law, etc. (see Ciglean's Democracy [2]).

Now I recall a *marking technique* that allows for the definition of a total order among the proposals made concurrently and asynchronously by a set of ordered agents on a shared resource (e.g. an order, a variable, processor time). This technique also allows to specify implicit "complies to" relations.

A **conflict resource** is a resource for which several agents can make proposals in a concurrent and asynchronous manner. $\mathcal{R}$, denotes a generic shared resource.

Remark 8. A **proposal source** for a resource $\mathcal{R}$ is an entity (e.g. an agent, abstract agent) that can make specific proposals concerning the allocation (or valuation) of $\mathcal{R}$.

We will consider that a total 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 $P_k^{\mathcal{R}}$, $k \geq k_0^{\mathcal{R}}$. $k_0^{\mathcal{R}}$ is the first position.

Each *proposal source* $P_i^{\mathcal{R}}$ maintains a counter $\mathbf{C_i^{\mathcal{R}}}$ for the *conflict resource* $\mathcal{R}$. Markers for defining order on messages in distributed settings have been first introduced in [1]. The markers involved in our *marking technique for ordered proposal sources* are called **signatures**.

Definition 5. *A* **signature** *is a chain h of pairs, $|a{:}b|$, that can be associated to a proposal, Z, for $\mathcal{R}$. A pair $p{=}|a{:}b|$ in h signals that Z complies to a proposal for $\mathcal{R}$ that was made by $P_a^{\mathcal{R}}$ when its $C_a^{\mathcal{R}}$ had the value b, and it knew the prefix of p in h.*

Example 1. The next are examples of signatures:

- $||$ -an empty signature
- $|1{:}5|$ -a one pair signature
- $|1{:}5|2:4|$ -a two pairs signature
- $|2{:}4|3:2|5:2|7:23|$ -the pairs are ordered in a signature.

While the form of these pairs resembles the ones in [1], their semantic is different. An order $\propto$ (read "follows") is defined on pairs such that $|i_1{:}l_1|{\propto}|i_2{:}l_2|$ if either $i_1{>}i_2$, or $(i_1{=}i_2){\wedge}(l_1{<}l_2)$ — this is due to the fact that l_1 and l_2 are counters.

Example 2. The next are examples of order on pairs:

- $|1{:}5| \propto |1{:}7|$
- $|2{:}5| \propto |1{:}3|$
- $|4{:}5| \propto |2{:}5|$
- $|4{:}7| \propto |2{:}5|$

Definition 6. *A signature h_1* **is stronger than** *a signature h_2 if a lexicographic comparison on them, using the order $\propto$ on pairs, decides that $h_2 \propto h_1$.*

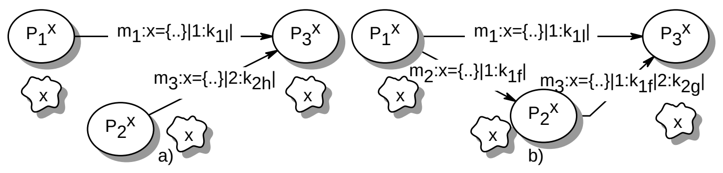

Fig. 1. Simple scenarios with messages for proposals on a resource, x.

Example 3. The next are examples of order on signatures:

$|1{:}7|$ stronger than $|2{:}8|$
$|2{:}9|$ stronger than $|2{:}8|$
$|1{:}7|3:5|$ stronger than $|1{:}7|$
$|1{:}7|3:5|$ stronger than $|1{:}7|4:7|$
$|1{:}7|3:6|$ stronger than $|1{:}7|3:5|$
$|1{:}7|3:5|4:2|$ stronger than $|1{:}7|3:5|$
$|1{:}7|3:6|$ stronger than $|1{:}7|3:5|4:3|$
$|1{:}7|3:6|4:2|$ stronger than $|1{:}7|3:5|4:3|$

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

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

For example, in Figure 1 the agent P_3^x may get messages concerning the same resource x from P_1^x and P_2^x. In Figure 1a, if the agent P_3^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 1b P_2^x knows $|1{:}k_{1f}|$ and the message m_1 is the strongest only if $k_{1f}{<}k_{1l}$. The length of a signature tagging proposals for a *conflict resource*, $\mathcal{R}$, is upper bounded by the number of *proposal sources* for $\mathcal{R}$.

I also define some notations:

– The strongest signature received by $P_k^{\mathcal{R}}$ up to and including a given event is denoted by signature(k), and
– the signature marking a message m is denoted signature(m).

4.3 Communication schema

All pairs of agents can communicate directly. The communication channels are considered reliable. Messages are delivered with random but finite delay.

4.4 Protocol

In AAS, as presented further in this subsection, the agents exchange messages about sets of values for variables (aggregates). Sets of aggregates for combinations of variables are called **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 7. *An* **aggregate** *is a triplet $\langle x_j, s_j, h_j \rangle$ where x_j is a variable, s_j a set of values for x_j, $s_j \neq \emptyset$, and h_j a signature of the pair (x_j, s_j). It is also called* assignment.

The signature guarantees a correct message ordering. It determines if a given aggregate is more recent than another.

Remark 10. The strongest aggregate received by an agent A_i for each variable define its **view**, view(A_i).

The local search space of an agent A_i, is denoted by SS(A_i) and is defined by the Cartesian product of the domains in CSP(A_i).

A computation in AAS reasons on some types of nogoods (conflicts). $V' \rightarrow \neg T_i$ is a **nogood entailed for $\mathbf{A_i}$ by its view V**, denoted $\mathbf{NV_i(V)}$, iff $V' \subseteq V$ and V and V' disable the same tuples, T_i, from SS(A_i). An **explicit nogood** has the form $\neg V$, or "$V \rightarrow fail$", where V is an aggregate-set. 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, such that T can be represented by the structures (e.g. stack) of a systematic centralized backtracking algorithm. An incoming explicit nogood whose conclusion is a superset of the current set of solutions for the local CSP of the agent may not be completely representable in CL (by the structures of the used backtracking algorithm). Such a nogood is called **overflowing nogood**.

An agent maintains its view and a valid CL and always enforces its CL and its nogood entailed by the view. To compactly denote sets of messages of the same type exchanged at once among the same agents, in the AAS protocol we often directly write all their parameters as parameter of one message. The following types of messages are exchanged in AAS:

– **ok?** messages having as parameter an aggregate.
– **nogood** messages announcing an explicit nogood.
– **add-link** messages announcing the interest of the sender in a variable.

Remark 11. An agent is interested in a variable, x, if it enforces constraints involving x.

add-link(var) is sent from an agent A^j to an agent A^i, $j>i$ and informs A^i that A^j is interested in the variable(s) var.

ok?(a) 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). known(A_i) includes the view of A_i as well as aggregates built by A_i. $a \in$known(A_i).

Rule 1 *Any tuple not removed by known(A_i) must satisfy the local constraints of the sender A_i and its valid nogoods*[2]*.*

Remark 12. Generally, an aggregate has to be built and sent by A_i **only** if the strongest aggregate for the same variable known by A_i does not have the same set of values. Exceptions to this 'only' appear for the first proposal made by A_i after nogoods of certain types are discarded.

[2] gives two alternative rules for deciding the **resend condition** exception when an aggregate in known(A_i)\view(A_i) will be multicasted on outgoing links with a new signature.

Remark 13 (cover). We say that an aggregate-set V is covered by an aggregate-set V' if any tuple whose projection on the variables of V is in V, also projects on the variables of V' in V'.

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[3] 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:

$$\frac{\begin{array}{c} V_1 \wedge V_2 \rightarrow \neg T^1 \\ V_1 \wedge V_3 \rightarrow \neg T^2 \end{array}}{\Rightarrow V_1 \wedge V_2 \wedge V_3 \rightarrow \neg(T^1 \vee T^2),} \tag{3}$$

where V_1, V_2 and V_3 are aggregate-sets proposed by predecessors. They are obtained by grouping the elements of the nogoods, such that V_1, V_2 and V_3 have no aggregate in common.

There exist several versions of AAS:

– AAS1 is the version of AAS where the agents store all distinct valid nogoods. In AAS1, the CL becomes redundant, but the space complexity is exponential in the size of the local problem, even if it is polynomial in the size of the external problem.
  AAS2 is the extreme case of AAS1 where all the distinct received nogoods are stored.
– AAS0 is the version AAS where agents maintain a CL as described so far.

Procedures that can be followed by an agent A_i in AAS1, simplified for the case where all enforcers are modifiers, are described in Algorithm 1 and Algorithm 2. Modifications for the general case are described later.

[2] Except for constraints about which A_i knows that a successor enforces them.
[3] Covers the search space.

```
    when received (ok?,$\langle x_j, s_j, h_{x_j}\rangle$) do
        if(signature($x_j$) invalidates $h_{x_j}$) return;
        add($\langle x_j, s_j, h_{x_j}\rangle$) to agent_view;
        reconsider stored and invalidated nogoods; check_agent_view;
    end do.
    when received (nogood,$A_j$,$\neg N$) do
1.1     add the new valid aggregates for already connected variables in $\neg N$ to agent_view;
        if ((($\exists\neg M$) $\wedge$ ($A_i$ knows $\neg M$) $\wedge$ (consequence$_i$($\neg N$) covered by consequence$_i$($\neg M$))
                $\wedge$ $\neg$(better $\neg N$ than $\neg M$)) $\vee$ invalid($\neg N$)) then
            if (I do not want to discard $\neg N$) then
                when $\langle x_k, s_k, t_k\rangle$, where $x_k$ is not connected, is contained in $\neg N$
                    send add-link($x_k$) to modifiers($x_k$); add $\langle x_k, s_k, t_k\rangle$ to agent_view;

                store $\neg N$;
            end
        else
            when $\langle x_k, s_k, t_k\rangle$, where $x_k$ is not connected, is contained in $\neg N$
                send add-link($x_k$) to modifiers($x_k$); add $\langle x_k, s_k, t_k\rangle$ to agent_view;

            put $\neg N$ in nogood-list;
        end
1.2     reconsider stored and invalidated nogoods;
        old_aggregate_set $\leftarrow$ current_aggregate_set; check_agent_view;
        for all oa = ca; (oa$\in$old_aggregate_set)$\wedge$(ca$\in$current_aggregate_set) do
1.3         send (ok?,$\langle var(ca), \text{set}(ca), append(signature(ca), |i{:}C^{i}_{var(ca)}|)\rangle$) to $A_j$;
        end do
    end do.
```

Algorithm 1: Procedures of A_i for receiving messages in AAS1.

- *consequence*$_i$($\neg$N) is a function returning the maximal aggregate-set included in N, which was generated by A_i.
- *modifiers*(x_k) is a function returning M^s_k, the set of agents that have announced that they want to modify x_k.
- *append*(h,p) appends the pair p to the signature h.

Remark 14. Here the **add-link** are sent to all the agents that can modify involved variables. But actually, as long as no agent reordering is performed, it is redundant to send them to higher priority agents [2].

The function *need_multicast(a)* fails only when both: the resend condition fails, and a does not modify the tuples removed from SS(A_i) by known(A_i). *clean()* removes the invalidated aggregates from *current_aggregate_set*. signature(x) returns the signature of the strongest aggregate for x in view(A_i).

At line 2.4, needed(a) succeeds when a has the same set as some assignment b for var(a), found in *old_aggregate_set*, and b is still valid. Then, a inherits the signature of b, signature(a)$\leftarrow$signature(b).

Algorithm 1 stores all the valid assignments. However, it could be modified to store only the strongest one [2].

```
    procedure check_agent_view do
2.1     when agent_view and current_aggregate_set are not consistent
            if no aggregate_set, V, in SS($A_i$) is consistent with agent_view then
                backtrack;
            else
2.2             select V $\subseteq SS(A_i)$ where agent_view, CSP($A_i$) and V are consistent;
                clean(current_aggregate_set);
                for all $a \in V$ do
                    if (need_multicast(a)) then
                        $x_k \leftarrow$ var(a); $C^i_{x_k}$++;
                        signature(a) $\leftarrow$ append(signature($x_k$),$|i{:}C^i_{x_k}|$);
                        send (ok?,$\langle x_k, set(a), signature(x_k)|i{:}C^i_{x_k}|\rangle$)
                                to lower priority agents in outgoing links($x_k$);
2.3                     current_aggregate_set $\leftarrow$ current_aggregate_set $\cup$ a;
                    else
2.4                     if (needed(a)) then
                            current_aggregate_set $\leftarrow$ current_aggregate_set $\cup$ a;
                            $signatures_i$[var(current_aggregate_set)] $\leftarrow$ signature of
                                    var(current_aggregate_set) in old_aggregate_set
                        end
                    end
                end do
            end
    end do.
    procedure backtrack do
        nogoods $\leftarrow \{V \mid V =$ inconsistent subset of agent view$\}$;
        when an empty set is an element of nogoods
            broadcast to other agents that there is no solution; terminate this algorithm;

        for every $V \in$ nogoods do
            select $A_k$, the lowest priority agent among those proposing aggregates in $V$;
2.5         send (nogood,$A_i$,$V$) to $A_k$;
            remove from agent_view all aggregates proposed by $A_k$ ;
            reconsider stored and invalidated explicit nogoods;
        end do
        check_agent_view;
    end do.
```

Algorithm 2: Other procedures of A_i in AAS1.

In the Definition 3, agents may not be able/want to propose splitting of domains of some of their variables. As explained in Section 4.1, the order on agents defines restrictions on their ability to aggregate several proposals in one assignment.

Definition 8. *An agent A_i is* aggregator *for a variable x_k only if it can propose aggregates containing more than one value for x_k.*

Some specifications are needed to Algorithm 1, in order to accommodate the Definition 3:

1. a vector of boolean values, $m_i[k]$, tells whether A_i is modifier for the variable x_k.
2. a vector of boolean values, $a_i[k]$, tells whether A_i is aggregator for the variable x_k.
3. a vector of signatures, $signatures_i[k]$, tells to A_i the strongest signature it received so far for x_k.
4. in procedure *check_agent_view*, an agent that is not aggregator for x_k selects at line 2.2 only aggregate-sets that do not aggregate more than one value for x_k.
5. in procedure *check_agent_view*, an agent that is not aggregator for x_k is not satisfied at line 2.1 by current_aggregate_sets that aggregate more than one value for x_k.
6. an agent that is not modifier for x_k does never send **ok?** messages with aggregates for x_k.

AAS is a complete search algorithm [2].

5 Asynchronous Aggregation Search without signatures (AAS')

An alternative implementation of the idea of AAS can be achieved without signatures, by tagging aggregates with simple counters. That alternative is called AAS'. AAS' shares several characteristics with AAS, allowing for similar aggregations, but the semantic of the information is complementary. Namely, the costly operations are triggered by changes of local instantiations rather than by changes of composed instantiations.

5.1 Alternatives to signatures

Alternatively to using signatures, proposals could be tagged using a simple counter. In this case an agent needs to store the last proposals on $\mathcal{R}$ made by each predecessor proposal source and considers as current proposal a combination of them. Then $P_a^{\mathcal{R}}$ needs not resend its old proposal p when p remains consistent with the view of $P_a^{\mathcal{R}}$ that changes. Instead $P_a^{\mathcal{R}}$ would have to send a new proposal if its proposal changes to become identical with the strongest received proposal.

The protocol where this technique is used instead of signatures [2], was called Asynchronous Aggregation Search without signatures (AAS').

5.2 Obtaining AAS' with AAS

Rather than implementing a new algorithm for AAS', we will see now how any implementation of AAS can be used to achieve AAS' by a simple reformulation of the treated DisCSP.

Consider that we have to solve a DisCSP (A,V,D,M,C), where A is a set of agents $A_1, ..., A_n$. V is a set of sets of variables $V_1, ...$, one set V_k for each agent

A_k. D is a set of sets of domains $D_1, ...$, one set D_k for each agent A_k. A domain $D_{k,j}$ in D_k specifies possible values for $x_{k,j}$. M is a set of sets of variables $M_1, ...$, one set M_k for each agent A_k, $M_k \in V_k$. M_k is the set of variables that can be modified by A_k. C is a set of sets of constraints $C_1, ...$, one set C_k for each agent A_k. Constraints in C_k involve only variables in V_k.

Consider that the variables in V_i are $x_{i,1}, ..., x_{i,v_i}$, and variables in V_i can be considered identical with other variables in the sets of variables of the other agents. We denote by V_i^i a set of newly generated distinct variables $x_{i,1}^i, ..., x_{i,v_i}^i$ (typically obtained by duplicating V_i).

By $\mathcal{P}$(S) we denote the set of subsets of the set S, including S itself. As difference to the standard convention, in this article we will consider that $\emptyset \notin \mathcal{P}$(S).

Definition 9. *The simplified version of the DisCSP P=(A,V,D,M,C), is a DisCSP P'=(A,V',D',M',C') such that:*

- $V_i' = \cup_{0<k\leq|A|} V_i^k$, *where* $V_i^k = \{x_{i,j}^k | \forall x_{i,j} \in V_i\}$
- $M_i' = \{x_{i,j}^i | \forall x_{i,j} \in M_i\}$
- $D_{i,j}' = \mathcal{P}(D_{i,j})$.
  The constraints are translated such that a tuple is feasible when the whole Cartesian product defined by a valuation is feasible. $C_i^{\frac{V_i}{V_i^i}}$ *denotes the translation of the constraints in* C_i*, such that they involve the variables* V_i^i *instead of the corresponding initial variables in* V_i*.*
- $C_i' = C_i^{\frac{V_i}{V_i^i}} \cup_{k\in\mathcal{P}(\{1..|A|\})} \{x_{i,j}^i \subseteq \cap_k x_{i,j}^k\}$

where $\{x_{i,j}^i \subseteq \cap_k x_{i,j}^k\}$ denotes a constraint requiring that the value of $x_{i,j}^i$ is a subset of the valuation of $x_{i,j}^k$ (a valuation of a variable is a set).

Lemma 1. *The simplified version of a DisCSP is equivalent to the initial DisCSP.*

Proof. Let us consider a DisCSP, P, and its simplified version P'.

One can easily check that any solution, s, of P satisfies P'. Actually, an agent can propose a set containing only the assignment in s of the corresponding variable and these proposals satisfy all added constraints.

Similarly, let s be a solution of P'. The intersection of all $x_{i,j}^k$ variables for all k satisfy P. This is a result of the fact that all the Cartesian products defined by this intersection are feasible.

Inference rules has to be added to the solver of each local agent:

Rule 2 $(N{\rightarrow}(x{\neq}a)) \rightarrow (N{\rightarrow}(x{\neq}c)), \forall c, c \subseteq a.$

where N denotes an aggregate-set.

Rule 3 $(N_1{\rightarrow}(x{\neq}a)){\wedge}(N_2{\rightarrow}(x{\neq}b))... \rightarrow ((N_1{\cup}N_2...){\rightarrow}(x{\neq}a{\cup}b...))$, *where* $N_1{\cup}N_2$ *denotes the union of the sets of aggregates in the compatible aggregate-sets* N_1*, and* N_2*.*

The dots show that the rule can be applied for combining simultaneously more than two nogoods. This rule can be generalized for Cartesian products T_1, and T_2 as follows:

Rule 4 *$(N_1{\rightarrow}(x{\notin}T_1)){\wedge}(N_2{\rightarrow}(x{\neq}T_2))... \rightarrow ((N_1{\cup}N_2...){\rightarrow}(x{\neq}T_1{\cup}T_2...))$, where $N_1{\cup}N_2$ denotes the union of the aggregates in N_1, and N_2.*

The dots show that the rule can be applied for combining simultaneously more than two Cartesian products.

Additionally, value reordering strategy has to be enforced in each agent:

Rule 5 *The agents will propose first, tuples corresponding to Cartesian products that are disjoint to the conclusion of any valid nogood and initial constraint.*

Now we can show that AAS over P' emulates AAS' over P.

Theorem 1. *AAS applied to the simplified version of a DisCSP emulates AAS' on its initial formulation.*

Proof. Each variable has only one modifier and no agent proposes aggregates. However, the proposal of an agent is now a set of (sub)domains defining a Cartesian product.

The proposals of an agent will be constrained to be coherent with the received aggregates and a new assignment is announced on each modification, as in AAS'.

Due to the fact that values as sets, the nogoods are also Cartesian products, as in AAS'.

Due to the dynamic value reordering strategy enforced, the agents propose first all possible Cartesian products of P that are completely feasible given their nogoods. After all these alternatives are proposed, as AAS' does, all remaining values will be removed by the added inference rules mentioned in this subsection, which together rule out the whole SS of the corresponding agent and lead to backtracking.

Remark 15. It is remarkable to note that AAS' can be obtained even from simplified versions of AAS, where local computation of Cartesian products is disabled (replaced with finding simple tuples).

6 Conclusions

Two important related families of algorithms for solving distributed problems have important differences in the implementation details. These are the Asynchronous Aggregation Search (AAS) techniques, which rely on signatures for their allowing aggregated proposals on the same variables, and AAS without signatures (AAS') that rely on re-forwarding proposals and intersection of agent views for reaching similar results.

In this paper it is shown how, despite the quite deep differences between the basic operations of AAS and AAS', one can perfectly implement AAS' based on an AAS implementation, by a 'trick' in problem formulation and local inference rules.

References

1. Paul R. Johnson and Robert H. Thomas. The maintenance of duplicated databases. RFC#677 NIC#31507 Network Working Group, January 27 1975.
2. Marius-Călin Silaghi. *Asynchronously Solving Distributed Problems with Privacy Requirements*. 2601, Swiss Federal Institute of Technology (EPFL), CH-1015 Ecublens, June 27, 2002.