Chapter 9

Asynchronous Aggregation Search

It is not the answer that enlightens but the question.
Eugen Ionescu

PREVIOUS asynchronous search techniques do not allow a proper treatment of intervals and numeric domains. These techniques enable only one agent to make proposals for a given variable. The consequence is that any domain proposed by that agent has an universal quantifier.

By proper treatment of intervals and numeric domains we will mean techniques presented in Chapter 5. In this thesis I will enable these kind of techniques and start in this chapter by enabling the use of aggregations (intervals) (see Section 5.2.1)

Definition 9.1 *A value v of a variable x in a solution t is interchangeable for a set S of values, iff by exchanging the instantiation of x to any value in S, t remains a solution.*

Lemma 9.1 *A set s_i proposed instead a single value of x_i in ABT has the semantic: the projection of the solution on x_i* **must** *be interchangeable, $\forall x_i \in s_i$.*

Proof. Consider the example in Figure 9.1. Imagine that an agent A_1 proposes $x \in s_1$ where s_1 is a set of values. When no other agent can change this proposal, other agents can only propose values for their variables if the whole Cartesian-product defined by these proposals are completely feasible. Otherwise, some agent A_2 may reason about a subset s_2 of s_1 and another agent A_3 about a disjoint subset s_3, leading to incoherent decisions and deadlock. □

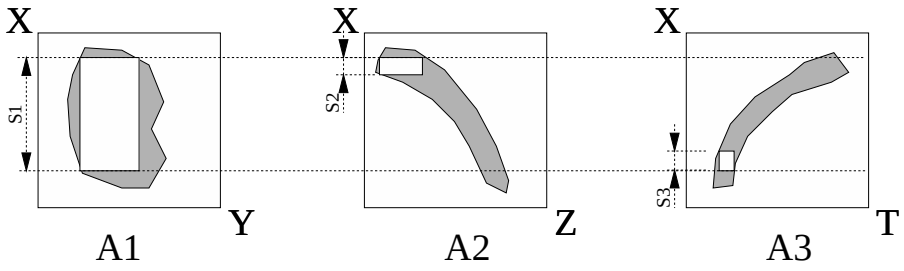

Figure 9.1: Use of intervals in ABT with existential semantic would lead to incoherences: A_1 proposes $x \in s_1$ while A_2 and A_3 are satisfied with $x \in s_2$ respectively $x \in s_3$, but $s_2 \cap s_3 = \emptyset$.

Remark 9.1 *For soundness, an (interval/set) proposed by an agent in ABT will either be rejected, or it will be completely accepted in a Cartesian product with the proposal of the other agents. Any agent that cannot propose a completely feasible Cartesian-product with the received proposals have to refuse them with a nogood.*

In Figure 9.1, ABT does not allow A_2 to constrain A_3 to choose its proposals in s_2. This is different from what is done in centralized settings and the treatment of intervals is not proper. An extension of ABT is proposed now where such an improved treatment of intervals is possible. AAS allows for proposals of sets with existential quantifier.

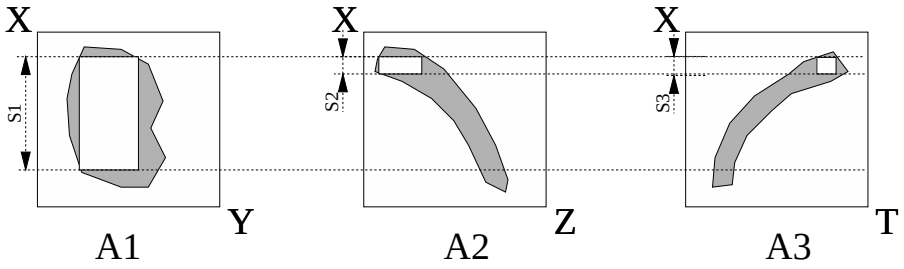

Figure 9.2: Use of intervals in AAS with existential semantic: $\exists x \in s_1$ and $\exists x \in s_2$ and $\exists x \in s_3$.

Example 9.12 *ABT, AWC, and DIBT supported only proposals with universal quantifiers (e.g.* **require***($\forall x \in \{1\}$)). I modify and generalize existing asynchronous search protocols to support proposals with existential quantifiers (e.g.* **require***($\exists x \in \{[1.5, 10^5] \cup \{0, 1\}\}$)).*

This is a first step towards the more complex goal of introducing binary splitting techniques with maintenance of local consistency, goal achieved in Chapter 13.

9.1 Initial Simplified Framework of AAS

When I designed AAS, the design was done in such a way to obtain a generalization of ABT. Nevertheless, I was advised that such an important result may be suspect for superficial skeptical reviewers. It was decided that it is better to describe in initial publications a weaker version which encompasses only a particular implementation. Under this consideration the simple formalism in (Silaghi *et al.* 2000a) has been introduced. This section recalls this historical formalism that was described as *an easy-read*.

The full description of the formalism of AAS, generalizing ABT, was later described informally in (Silaghi *et al.* 2001i) and more formally in the next section.

In (Silaghi *et al.* 2000a) DisCSPs 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.

Definition 9.2 (DisCSP) *A* **DisCSP** *is given by a set of agents $A_1, ..., A_n$ where each agent A_i wants to enforce some private constraints and negotiate the instantiation of all shared variables involved in its constraints. The agents want to agree on instantiations such that all the predicates are satisfied.*

Given this definition, each agent can propose fragmentations of the domains for its variables, in such a way that convergence to a fully feasible subspace can be achieved similarly as in classical techniques for numeric problems. However, in this chapter I will describe an asynchronous technique that can emulate only centralized techniques working with sets/intervals in the way of FCPR and FC-CPR (Chapter 5). Techniques exploiting this formalism and that can emulate FMBC1 and UCA6 are described in Chapters 12 and 13.

9.2 Asynchronous Aggregation Search (AAS)

We assume that before search, each agent announces the shared variables that it wants to be able to assign. Asynchronous Aggregation Search (AAS) (Silaghi *et al.* 2000a) is an extension[1]

[1]The exact implementation described in (Silaghi *et al.* 2000a) has details that cannot be considered as extensions of ABTp (namely, it included the features of GOB). Nevertheless, here we reformulate AAS as an extension of ABTp.

of ABTp where several agents are allowed to simultaneously propose instantiations for the same shared variable. This is made possible by using the *signatures* presented in the previous subsection where agents A_i can act as $P_i^{x_k}$ for some shared variable x_k. $k_0^{x_k} = 1$ for any shared variable x_k.

9.2.1 Constraint enforcement

A more complex definition using both variable- and constraint-agents, but still equivalent to the previous one has been proposed in (Hannebauer 2000). Nevertheless, the definition in (Hannebauer 2000) can be modified into the next extension of the Definition 9.2.

Yet another interesting definition of DisCSPs appeared in (Meseguer & Jiménez 2000), splitting the set of any binary constraints on x_i and x_j between the owners of the two variables. All these definitions can be seen in the framework of (Khedro & Genesereth 1994), discussed in Chapter 3.

The following definition imposes some restrictions on the operations that agents can perform, and has a theoretical relevance discussed in Section 9.2.6.

Definition 9.3 (DisCSP with modifiers) *A* **DisCSP** *is given by a set of agents $A_1, ..., A_n$ where each agent A_i wants to enforce some private constraint p_i.*

The set of shared variables involved in p_i is V_i. The agents negotiate the instantiation of the variables in V_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:

- the initial definition of AAS,
- the initial definition of ABT,
- the definitions in (Meseguer & Jiménez 2000).

Algorithms defined for *DisCSPs with modifiers* can then be correctly compared with algorithms developed for any sub-framework.

Definition 9.4 *The set of modifiers of x_i, M_i^s, is the set of agents having x_i in their M_i.*

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

Definition 9.5 (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.

Definition 9.6 (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 fully generalize the corresponding notions of ABT.

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 (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).

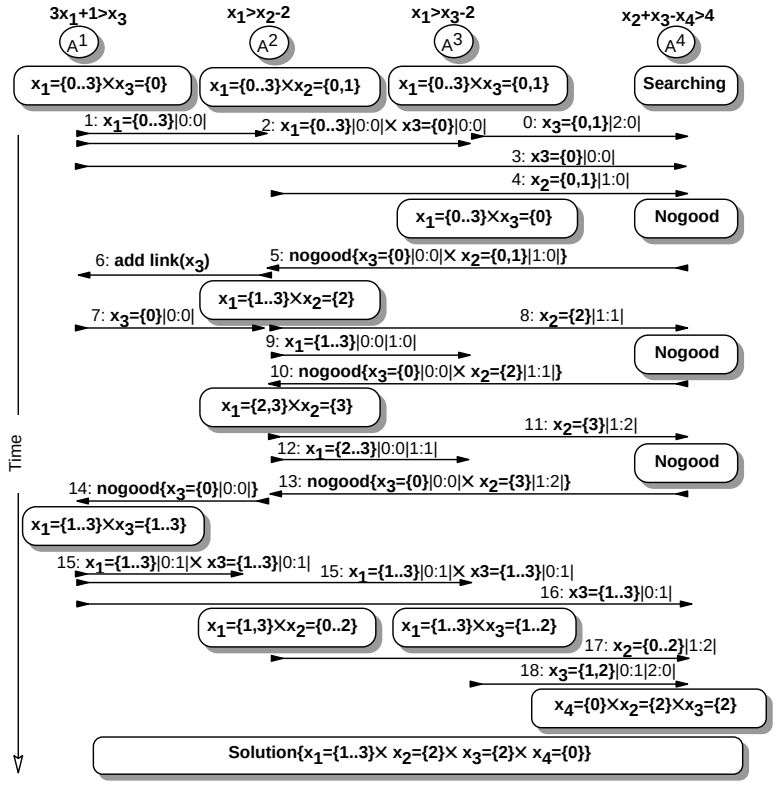

Figure 9.3: *Trace of a search with AAS. The states of the agents can be represented by the current solution to the local CSP defined by their constraints. The pairs |a:b| included in the messages are used for message ordering.*

9.2.2 Example

In AAS, each agent maintains values for the set of variables in which it is involved. Thus, A^1 maintains value combinations for x_1 and x_3, A^2 for x_1 and x_2, A^3 for x_1 and x_3, and A^4 for all of x_2, x_3 and x_4 (see Figure 9.3). More precisely, an assignment is replaced by a domain for the involved variable. Sets of such domains represent all the tuples of their Cartesian product. The proposal $x_1 = \{0..3\}, x_2 = \{0, 1\}$, for example, will represent all the tuples of the Cartesian product $\{0..3\}\times\{0, 1\}$. Similarly, a solution is no longer a list of individual assignments, but a Cartesian product of domains which represents a set of possible valuations. In scheduling and resource allocation problems with large domains, the savings allowed by the Cartesian product representation can be particularly significant.

Figure 9.3 illustrates the behavior of a version of AAS on our small example. Agent A^1 first selects the Cartesian product $\{x_1 = \{0..3\}\}\times\{x_3 = \{0\}\}$, and sends an **ok?** message with the needed parts of this information to A^2, A^3 and A^4 who manage constraints sharing variables with A^1. The algorithm now works in exactly the same manner as ABT, except that messages refer to domains and agents select different Cartesian products rather than value assignments. More specifically, A^4 finds that no combination in the Cartesian product $\{x_2 = \{0, 1\}\} \times \{x_3 = \{0\}\}$ is compatible with its constraint. It therefore generates a **nogood** for this combination which causes A^2 to select the next Cartesian product. Note that since this change selects a subrange of the values allowed by the knowledge of A^2 for x_1, it is not necessary to verify this change with A^1. If it were not possible to find such a subrange, a **nogood** would be generated and sent to A^1 in order to try another Cartesian-product there.

9.2.3 Proposals on shared variables

All the agents can *aggregate* several assignments for 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, (This exception is inspired from (Meseguer & Jiménez 2000))

Moreover:

Remark 9.2 *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. For simplicity, in the rest of the description we consider that this is the case for all shared variables.

Within the aforementioned framework for conflict resources, when the shared resource is a variable, x, a proposal is defined by a set of values, S:

- the "sample requirement" is: $\forall x \in S$.
- the "set of alternative requirements" is: $\{\forall x \in S' \mid S' \subset S, S' \neq \emptyset\}$

9.2.4 Aggregations

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 9.7 *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 a generalization of an* assignment.

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

Definition 9.8 *Given an aggregate $a = \langle x, s, h \rangle$, we use the notations:*

- *var(a) for a function returning x.*
- *set(a) for a function returning s.*
- *signature(a) for a function returning h.*

Let $a_1 = \langle x_j, s_j, h_j \rangle$ and $a_2 = \langle x_j, s'_j, h'_j \rangle$ be two aggregates for the variable x_j. a_1 is *stronger* than a_2 if h_j is stronger than h'_j.

Definition 9.9 *The strongest aggregate 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).

Definition 9.10 ($\mathbf{T_i(V)}$) *The set of tuples disabled from CSP(A_i) by V is formally* $\mathbf{T_i(V)}=\{t \mid t=(x_1=v_t^1,...,x_n=v_t^n), \forall j, x_j \in vars(A_i);\ \forall u \neq j,\ x_j \not\equiv x_u;\ n=|vars(A_i)|; \exists k \in [1..n], \langle x_k, s_k, h_k\rangle \in V,\ v_t^k \notin s_k\}$.

This can be read: the set of tuples t of assignments to all variables in vars(A_i) such that at least one aggregate $\langle x_k, s_k, h_k\rangle$ is contained in V, such that the value assigned to x_k in t does not belong to s_k.

Part of the view of A_i may have no effect on the set of tuples disabled from CSP(A_i). However, it is up to the agent to optimize the detection of conflicts.

Definition 9.11 (Nogood entailed by the view) *$V' \rightarrow \neg T_i(V)$ is a* **nogood entailed for $\mathbf{A_i}$ by its view $\mathbf{V}$**, *denoted* $\mathbf{NV_i(V)}$, *iff $V' \subseteq V$ and $T_i(V')=T_i(V)$.*

Definition 9.12 *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 9.13 *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 systematic centralized backtracking algorithm.

The correctness is typically possible since the current set of local solutions is already represented by any systematic backtracking.

Definition 9.14 (overflowing nogood) *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.*

In order to obtain instantiation asynchronism, with no infinite loops, AAS uses a strict order $\prec$ on agents as proposed for ABT. 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.

Definition 9.15 (search space (SS)) *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).*

9.2.5 The AAS protocol

An agent maintains its view and a valid CL and always enforces its CL and its nogood entailed by the view.

In AAS an agent typically makes proposals for several variables and, in consequence, can receive **add-link** messages for several variables. 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. We maintain the name of the original ABT messages:

- **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.

Definition 9.16 *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. While no parameter is needed in ABT where the receiver can make proposals on only one variable, when an agent can modify several variables, it becomes useful to specify which variables are required.

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 5 *Any tuple not in T_i(known(A_i)) must satisfy the local constraints of the sender A_i and its valid nogoods*[2].

Remark 9.3 (see ABTr (Chapter 6)) *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.*

Here I give 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:

- **Rule 6** *The resend condition is activated when:*
  - *An overflowing nogood is discarded while being valid.*
  - *When a CL obtained by inference from at least 2 nogoods is invalidated.*
- **Rule 7** *The resend condition appears after any valid nogood is lost. This is when, after applying an inference:*
  $$(CL_{initial} \wedge \neg N) \rightarrow CL_{final}$$
  either $CL_{initial}$ or $\neg N$ is discarded while being valid.

Definition 9.17 (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:

$$\begin{array}{l} V_1 \wedge V_2 \rightarrow \neg T^1 \\ V_1 \wedge V_3 \rightarrow \neg T^2 \\ \hline \Rightarrow \quad V_1 \wedge V_2 \wedge V_3 \rightarrow \neg(T^1 \vee T^2), \end{array} \tag{9.1}$$

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. Optimizations can split the agent as proposed in Replica-based DisCSPs (see Chapter 13).

  AAS2 is the extreme case of AAS1 where all the distinct received nogoods are stored.

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

- AAS0 is the version AAS where agents maintain a CL as described so far. Two variants that we have mentioned so far are:
  - AAS0_1 is the variant of AAS0 where agents act in agreement with the Rule 6.
  - AAS0_2 is the variant of AAS0 where agents act in agreement with the Rule 7.

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 24. Modifications for the general case are described later.

- *consequence*$_i(\neg\text{N})$ 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_k^s, 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.

The function *need_multicast(a)* fails only when both: the resend condition fails, and a does not modify $T_i(\text{known}(A_i))$. *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 24.7, 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 24 stores all the valid assignments. However, it could be modified to store only the strongest one (see (Silaghi *et al.* 2000a)).

9.2.6 When not all enforcers are modifiers

In the Definition 9.3, agents may not be able/want to propose splitting of domains of some of their variables. As explained in Section 9.2.3, the order on agents defines restrictions on their ability to aggregate several proposals in one assignment.

Definition 9.18 *An agent A_i is* modifier *for a variable x_k only if it can propose aggregates for x_k ($A_i \in M_k^s$).*

Definition 9.19 *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 (see Section 9.2.1).*

Some specifications are needed to Algorithm 24, in order to accommodate the Definition 9.3. First of all:

- a vector of boolean values, $m_i[k]$, tells whether A_i is modifier for the variable x_k.
- a vector of boolean values, $a_i[k]$, tells whether A_i is aggregator for the variable x_k.
- a vector of signatures, *signatures*$_i[k]$, tells to A_i the strongest signature it received so far for x_k.
- in procedure *check_agent_view*, an agent that is not aggregator for x_k selects at line 24.5 only aggregate-sets that do not aggregate more than one value for x_k.
- in procedure *check_agent_view*, an agent that is not aggregator for x_k is not satisfied at line 24.4 by current_aggregate_sets that aggregate more than one value for x_k.
- an agent that is not modifier for x_k does never send **ok?** messages with aggregates for x_k.

Proposition 9.1 *The modifications described in this section do not change the correctness, completeness and termination of AAS.*

Proof. From the conditions described at Section 9.2.3 it follows that any agent that is not modifier either:

- has no proposal to make since it has no outgoing link to lower priority agents
- only receives non-aggregated assignments and does never need to modify them, while it can refuse them.

Moreover, the fact that no proposal is made at initialization on non-modified variables can be interpreted as due to the fact that a proposal has already been received from higher priority agents and the invalid messages are discarded by the channels.

It follows that the restriction on the messages allowed to non-modifier agents are transparent to the rest of the reasoning on the properties of AAS. □

9.2.7 Properties of AAS

In general AAS2 is not an extension of ABT, even when the problem is defined in the same way (the same *modifiers*, etc.), since it uses assignments from nogoods. AAS2 has to renounce to the use of new assignments arriving with nogoods in order to make sure it emulates ABT.

Remark 9.4 *The line 24.1 can be optional in AAS2. In that way, AAS2 emulates ABT, with the overhead of the tags for assignments.*

Definition 9.20 (Quiescence in AAS) *By quiescence of a group of agents in AAS we mean that none of them will receive or generate any valid nogoods, new valid assignments, or* **add-link** *messages.*

We prove by induction on increasing i that:

Property 9.2 *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* $\mathrm{CSP}(A^j){\cup}\mathrm{NV_j}(\mathrm{view}(A^j))$.

Proof. 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 $t^i_o < t^{i-1}+\tau$. A^i receives the last valid **ok?** message at time t^i_o. Only one valid explicit nogood can be received for a proposal since the proposal is immediately changed on such an event. Similarly, there is a finite number of valid nogoods that can be received by A^i for any of its proposals made after t^i_o.

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^i_o are refused or if it cannot find any proposal, A^i has to send a valid explicit nogood $\neg N$ to somebody.

   I. If N is empty, failure is detected and the induction step is proved.

   II. 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:

   a) detects failure.

   b) or finds a new proposal modifying at least one of the elements of N. In this case, since 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.

   c) or discards $\neg N$ after merging it to a CL invalidated later. In the last case, 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.

From here, it results that either empty nogood is detected, or A^i will receive a new proposal. This contradicts the assumption that the last **ok?** message was received by A^i at time t^i_o and the induction step is proved.

The property can be attributed to an empty set of agents and it is therefore proved by induction for all agents. □

Property 9.3 *AAS is correct, complete, terminates and only requires polynomial space for its completeness.*

Proof. A very detailed proof is given in (WebProof 2000).

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 lack of an infinite loop is a direct consequence of the Property 9.2.

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 (see (WebProof 2000)). If quiescence is reached without detecting an empty nogood, then all the agents agree with their predecessors and their intersection is nonempty and correct. □

There are several ways in which the agents can build the aggregations. Aggregation algorithms guaranteeing a complete and non-redundant covering of the solution space determined by local constraints are given in (Hubbe & Freuder 1992; Haselböck 1993a; Silaghi *et al.* 2000c; 1999c). The ones described in the last two mentioned papers have been presented in chapter 5.

In (Silaghi *et al.* 1999c), is explained how the technique of aggregations is related to the technique of interchangeabilities (Freuder 1991). However, aggregations are less strong than neighbourhood interchangeability. As proven in (Silaghi *et al.* 1999c), consistency and additional information can allow agents to compute stronger interchangeabilities that can have important impact on efficiency.

Remark 9.5 *The agents tend to rediscover successively the same nogood several times. The agents can store the last sent nogood, LSN. Each newly computed nogood can be checked against LSN and can be discarded if they are identical. With AAS2, this can be done even when the two nogoods differ only in signatures.*

The optimisations to local processing proposed in (Luo *et al.* 1992; Hamadi 1999b) (e.g. forward checking of labels) can also be applied. However, I did not make any effort at this level.

9.2.8 Experiments

AAS0, 1 and 2 have been evaluated on randomly generated problems with 15 and 20 agents, situated on distinct computers on a LAN. The constraints have been distributed to the agents in the same way that they would have been enforced in ABT so that they can be compared. As a consequence, the number of variables equals the number of agents. The size of domains is of 5 values and the problems are generated near the peak of difficulty (Cheesman *et al.* 1991) with a density of 30% and a tightness of constraints of 55%.

9.2.8.1 AAS2 versus AS2

An evaluation of distributed algorithms tries to take into account both the cost of messages and the cost of local computations. Function of the given problem, the cost of a message with respect to the cost of local computations has different importance. When the algorithms are run between agents that are threads on the same computer, the cost of a message may be low in comparison to other computations. Instead, when the agents are on different computer on an Ethernet or even very remote computers on the Internet, the cost of the local computations can become less and less important.

I evaluated algorithms with agents placed on distinct computers on an Ethernet LAN. The measured cost is the highest logic clock of the computation (Lamport 1978), where we have to choose the logic cost of local events and of messages. To analyze how the algorithms will behave on other kinds of networks/systems, I compute the cost for different samples of possible ratios between the cost of a message and the cost of a local operation (constraint check). As in (Yokoo *et al.* 1992), the logic cost of a local event (constraint check) is fixed to 1 and the logic cost of a message varies between 0 (agents are threads on a computer) and 40.

The cost of search is evaluated using the longest sequence of messages and constraint checks, LES (see Section 3.4.4). Each test is averaged over 50 instances. The measured parameters used

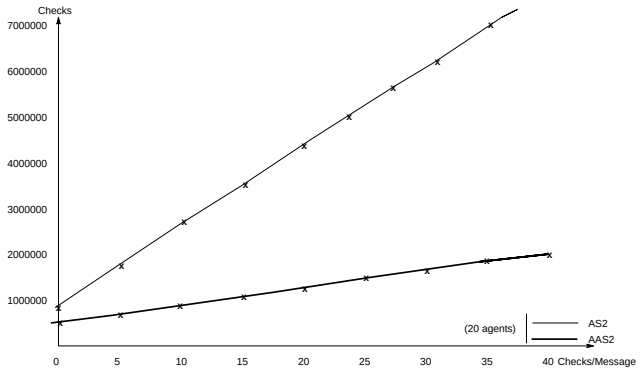

Figure 9.4: Comparison of the number of checks on four sets of randomly generated problems near the peak. Abscissae select the relative time needed for sending a message divided by the time for a constraint check.

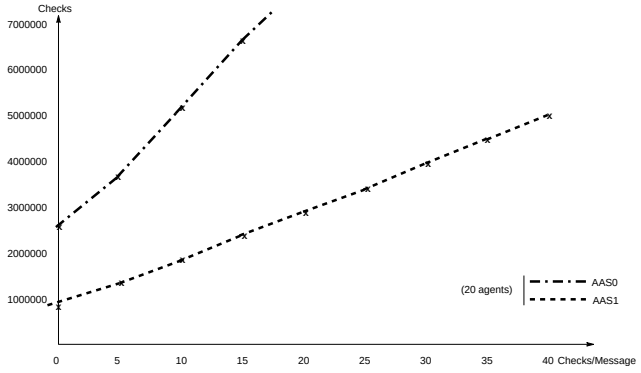

Figure 9.5: Comparison of the number of checks on four sets of randomly generated problems near the peak. Abscissae select the relative time needed for sending a message divided by the time for a constraint check.

for evaluation are the same as those given in (Yokoo *et al.* 1992). In graphs, the slopes of the curves give the number of messages. The intersections with the y-axis give the number of checks when the messages are considered instantaneous, LSC. AAS2 performs slightly better than its version without aggregation, AS2 (see Figure 9.4). There are specific cases where AS2 performs better for finding the first solution. However, for discovering that no solution exists AAS2 performs steadily better than AS2 since the whole search space needs to be expanded. AAS2 also reduces the longest sequence of messages, LSM, as well as the number of nogoods stored, by a factor of 50% on average.

9.2.8.2 AAS0 versus AAS1

AAS1 needs more messages than AAS2, and AAS0 even more (see Figure 9.5). However, they do not present memory problems.

9.2.8.3 AAS0 vs. AS0 and AAS1 vs. AS1

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

These experiments reveal a close connection between the storage of nogoods and the usefulness of aggregation. Beside its intrinsic gain, aggregates can provide an opportunity to replace the efficiency of some nogoods with a bounded space alternative. What I have noticed during traces,

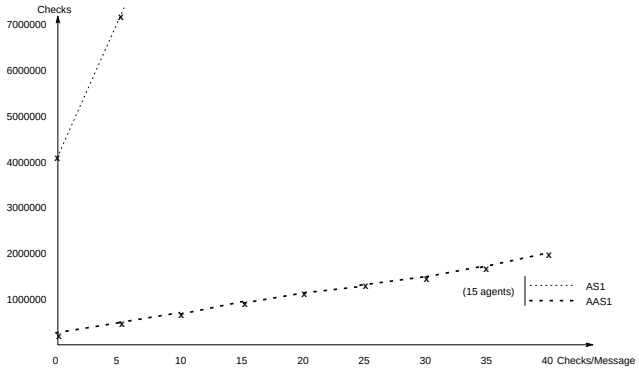

Figure 9.6: Comparison of the number of checks on four sets of randomly generated problems near the peak. Abscissae select the relative time needed for sending a message divided by the time for a constraint check.

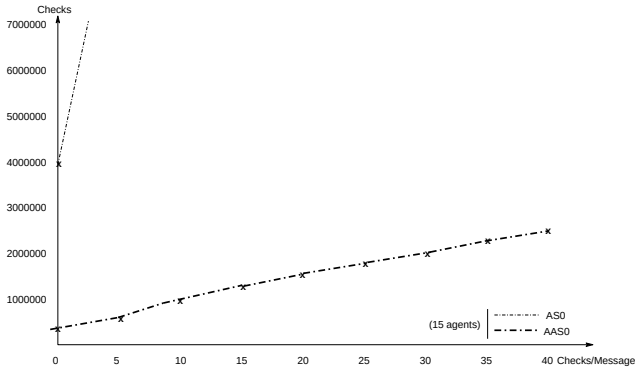

Figure 9.7: Comparison of the number of checks on four sets of randomly generated problems near the peak. Abscissae select the relative time needed for sending a message divided by the time for a constraint check.

is that in versions with full nogood storage, nogoods for single proposals can disable a large part of an aggregate, reducing the usefulness of the last one. The need to remove nogoods increases the importance of adding aggregation to this type of discrete problems.

As shown later, due to the treatment of intervals, aggregation is indispensable for problems with large or continuous domains.

9.2.9 Private constraints without signatures (DFC)

An alternative way of allowing agents to modify the same variable, x_i, in a CSP, P, was suggested in (Meseguer & Jiménez 2000) under the name Distributed Forward Checking (DFC). Here we present it from a different point of view than the one in (Meseguer & Jiménez 2000), namely as an instance of AAS. DFC consists in modeling x_i with a new variable x_i^j in each agent A^j. A constraint is then checked by the lowest priority agent modifying x_i in order to ensure that the intersection over all k of the proposals for x_i^k is nonempty. A new CSP, P', is obtained. AAS is run on P'. Since in P', only one agent sends proposals for each variable, the signature tagging an assignment for x_i^j will always have the form $|j$:value$(C_j^{x_i^j})|$. j is therefore redundant and value$(C_j^{x_i^j})$ can be sent alone as it was in ABT with polynomial space requiremnts. Whether the model P or the model P' are more efficient with AAS, is a question of future research.

9.2.10 Aggregations in AWC

As mentioned in (Silaghi *et al.* 2000g), since AWC resembles much to ABT, it is quite obvious to extend it to allow for aggregations in proposals. However, we can suspect that the use of signatures for proposals in protocols with agent reordering may prove less efficient than the use of alternative techniques, (e.g. the ones mentioned in Section 8.2.1, or the one used in DFC).

A related work discussing the distributed use of some aggregations is (Parkes 2001).

9.3 Summary

We have seen the way in which the marking technique introduced in the previous chapter can be used to allow agents to reason in a more abstract way about variable assignments, allowing for processing several assignments simultaneously. This is the case on which signatures have been initially developed.

The described technique is an essential step towards a proper treatment of numeric constraints. It introduces proposals of intervals and sets. AAS also offers efficiency improvements of up to an order of magnitude, inverse proportional to the size of nogood storage.

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**;

when received *(***nogood**,A_j,$\neg N$*)* **do**

24.1 add the new valid aggregates for already connected variables in $\neg N$ to *agent_view*;
  if *(((*$\exists \neg M$*) ∧ (*A_i *knows* $\neg M$*) ∧ (consequence*$_i$*(*$\neg N$*) covered by consequence*$_i$*(*$\neg M$*)) ∧*
    ¬(better $\neg N$ *than* $\neg M$*))*
    ∨ 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 $\neg N$;
    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*;

24.2 reconsider stored and invalidated nogoods;
  old_aggregate_set ← *current_aggregate_set*; **check_agent_view**;
  for all oa = ca*; (*oa∈old_aggregate_set*)∧(*ca∈current_aggregate_set*)* **do**

24.3 send (**ok?**,$\langle var(ca), \text{set}(ca), signature(ca)|\text{u:}C^u_{var(ca)}|\rangle$) to A_j;

procedure check_agent_view do

24.4 **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

24.5 select V ⊆ $SS(A_i)$ 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 ← var(a); $C^u_{x_k}$++; signature(a) ← append(signature(x_k),|u:$C^u_{x_k}$|);
          send (**ok?**,$\langle x_k, set(a), signature(x_k)|\text{u:}C^u_{x_k}|\rangle$) to lower priority agents
            in *outgoing links*(x_k);

24.6 *current_aggregate_set* ← *current_aggregate_set* ∪ a;
        else

24.7 **if**(needed(a)) *current_aggregate_set* ← *current_aggregate_set* ∪ a;

procedure backtrack do
  nogoods ← {V | 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;

24.8 send (**nogood**,A_i,V) to A_k;
    remove from *agent_view* all aggregates proposed by A_k ;
    reconsider stored and invalidated explicit nogoods;
  check_agent_view;

Algorithm 24: Procedures of A^u_i for receiving messages in AAS1.