Asynchronous Search with Aggregations

Marius Călin Silaghi and **Djamila Sam-Haroud** and **Boi Faltings**
Swiss Federal Institute of Technology, Artificial Intelligence Lab, DI-LIA, CH-1015 Lausanne, Switzerland
{silaghi,haroud,faltings}@lia.di.epfl.ch

Abstract

Many problem-solving tasks can be formalized as *constraint satisfaction problems* (CSPs). In a multi-agent setting, information about constraints and variables may belong to different agents and be kept confidential. Existing algorithms for distributed constraint satisfaction consider mainly the case where access to variables is restricted to certain agents, but constraints may have to be revealed. In this paper, we propose methods where constraints are private but variables can be manipulated by any agent.

We describe a new search technique for distributed CSPs, called *asynchronous aggregation search* (AAS). It differs from existing methods in that it treats sets of partial solutions, exchanges information about aggregated valuations for combinations of variables and uses customized messages to allow distributed solution detection. Three new distributed backtracking algorithms based on AAS are then presented and analyzed. While the approach we propose provides a more general framework for dealing with privacy requirements on constraints, our experiments show that its overall performance is comparable or better than that of existing methods.

Keywords: search, distributed AI, constraint satisfaction

Introduction

Multi-agent systems are often used for solving combinatorial problems such as resource allocation, scheduling, or planning. *Constraint satisfaction* has proven to be a highly successful paradigm for solving such problems in centralized settings. A constraint satisfaction problem (CSP) is given by:

- a set of n *variables* $x_1, ..., x_n$,
- a set of n *domains*, $D_1, ..., D_n$, for the variables,
- a set of k *relations*, $r_1 = (x_i, x_j, ...)$, ...,r_k, each of which is a subset of the set of variables, and
- a set of k *constraints*, $C_1, ..., C_k$. C_i gives the allowed value combinations for the corresponding relation r_i.

A *solution* to a CSP is an assignment of values from the corresponding domains to each variable such that for all relations, the combination of assigned values is allowed by the corresponding constraint. Many combinatorial problems, such as resource allocation, scheduling and planning can be modeled as CSPs. A *distributed* CSP (DCSP) arises when information is distributed among several agents. In the common definition of DCSP (Yokoo *et al.* 92), variables are distributed among agents so that each variable can only be assigned values by a single agent. Several *Asynchronous Search* (AS) algorithms have been developed that allow solving such problems by exchanging messages about variable assignments and conflicts with constraints (called nogoods) (Yokoo *et al.* 92; Hamadi & Bessière 98).

Asynchronous Search

In this section, we recall the basic background of AS using a small example (Figure 1). Without loosing the fundamental characteristics of AS, we restrict our description to the case with unbounded nogood recording (Yokoo *et al.* 92) and where each agent has exactly one variable. In this framework, each agent is responsible for maintaining the value of one variable. It has a link toward any agent that owns a constraint involving that variable. Agents are arranged in a priority order. A constraint is enforced by the agent which has the highest priority among those that are responsible for one of the variables in the corresponding relation.

In our example, there are four agents, A^1, A^2, A^3, A^4 who control the variables x_1, x_2, x_3, x_4, with identical domains D_1=D_2=D_3=D_4=$\{0, 1, 2, 3\}$. Agent A^1 wants to ensure that $3x_1 + 1 > x_3$, A^2 wants to have $x_1 > x_2 - 2$, A^3 requires that $x_1 > x_3 - 2$, and A^4 needs $x_2 + x_3 - x_4 > 4$ to hold. In order to solve this problem with conventional AS techniques, we first need to assign a priority to each agent, then move certain constraints to the agent with the higher priority. Let us assume that A^{i+1} has precedence over A^i. In this case, A^1's constraint has to be communicated to A^3 which will be responsible for its enforcement. Each agent will start by randomly assigning to its variable a value from its domain (0 in our example). Upon asynchronous backtracking, the local search space for each agent is determined by its local constraints along with the restrictions imposed by the other agents via **ok?** and **nogood** messages. When an agent assigns a value to its variable, it sends an **ok?**(var=value) message to all the higher-priority agents having a link with it. These agents then evaluate their constraints on that variable. If these constraints are satisfied by the new assignment, given all the known values for the other variables, they do nothing, otherwise they try a new value for

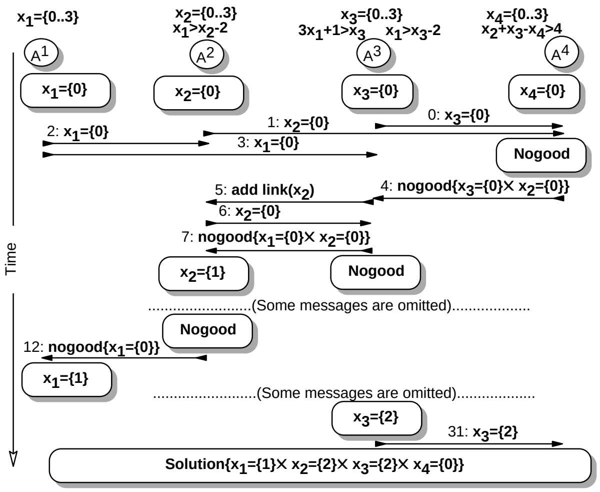

Figure 1: *Simplified trace of an asynchronous search process. Each agent A^i is assorted with a variable x_i, a set of constraints involving this variable and states represented by boxes. A state shows either the assignment chosen for the owned variable or a conflicting situation (nogood). The arrows represent messages. Each message is prefixed by a number.*

their variable. If any of them finds no available value, then it generates a **nogood** message. The agent receiving this **nogood** message will then have to incorporate the information in its local search space and change the faulty assignment or generate other nogoods, accordingly. Hence, constraints are always evaluated by higher-priority agents and values always changed by lower priority ones.

Figure 1 shows a simplified trace of message passing obtained for our example using the asynchronous backtracking algorithm described in (Yokoo *et al.* 92).

Each agent starts by assigning the value 0 to its variable. Agent A^1 then sends an **ok?** message to A^2 and A^3 and agents A^2 and A^3 both send **ok?** messages to A^4. Agents A^2 and A^3 both find the value received from A^1 to be compatible with their constraints. Hence, they do not react. However, A^4's constraint is violated and this agent returns a **nogood** message (4) to A^3.

Private constraints

In the AS formulation, constraints may need to be revealed to any other agent that controls a variable in the corresponding constraint. This corresponds well to certain applications, for example distributed control, but less well to negotiation where variables are public but constraints are private. In this paper, we address this latter case by proposing a technique called *Asynchronous Aggregation Search*. It differs from asynchronous search by the fact that agents exchange messages not about assignments to individual variables, but about tuples of variables. This allows eliminating restrictions on the order in which constraints are treated. Coupled with the fact that AAS allows aggregating ranges of tuples, we obtain efficiency gains over the existing asynchronous backtracking algorithms. The evaluation is done using three

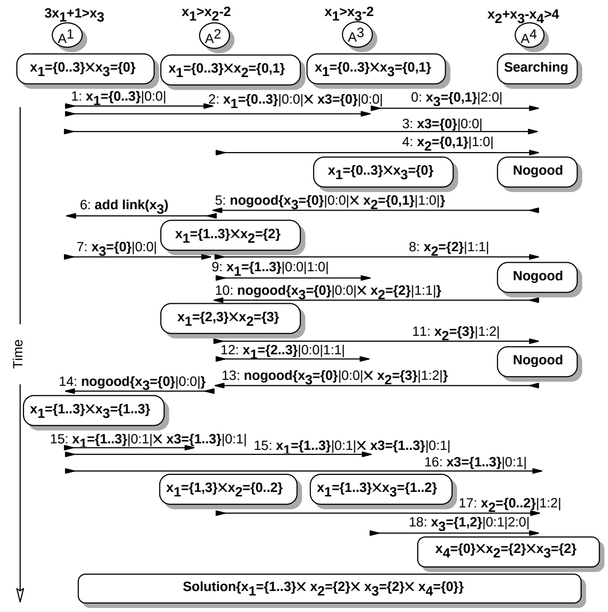

Figure 2: *Trace of a search with AAS. The states of the agents can be represented by the current solution to the local CSP defined by their constraints. The pairs $|a, b|$ included in the messages are used for message ordering.*

different implementations, based respectively on full, partial and no nogood recording.

Asynchronous Search with Aggregations

We now introduce *asynchronous aggregation search* (AAS), a new technique that propagates aggregated tuples of values rather than individual values themselves. 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 2). AAS differs from AS in the fact that message arguments are not just individual assignments, but Cartesian products of assignments (Hubbe & Freuder 92) to different variables. More precisely, in the current implementation of AAS, an assignment is a list of domains, one for each involved variable, which represent all the tuples of their Cartesian product. The assignment $x_1 = \{0..3\}, x_2 = \{0,1\}$, 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 2 illustrates the behavior 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 AS, except that messages refer to Cartesian products 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.

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 92; Haselböck 93; Silaghi, Sam-Haroud, & Faltings 2000).

AAS Algorithms

In this section we will present three distributed backtrack search algorithms based on aggregation. We start by giving the necessary background and definitions. Similarly to the AS algorithm of (Yokoo *et al.* 92), the agents are assigned priorities. We assume that the agent A^i has priority over another agent A^j if $i > j$. A *link* exists between two agents if they share a variable. The link is directed from the agent with lower priority to the agent with higher priority. Let A^i and A^j be two agents related by a link such that $i > j$. A^i is called the *predecessor* of A^j and conversely, A^j is called the *successor* of A^i. The *end agents* are those without incoming links. The *system agent* is a special agent that receives the subscriptions of the agents for the search. It decides the order of the agents, initializes the links and announces the termination of the search.

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

The history provides the information necessary for a correct message ordering. It determines if a given assignment is more recent than another and will be described in more details later. Let $a_1 = (x_j, set_j, h_j)$ and $a_2 = (x_j, set'_j, h'_j)$ be two assignments for the variable x_j. a_1 is *newer* than a_2 if h_j is more recent than h'_j.

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

An aggregate will be denoted compactly by (V, S, H) where V is the set of variables, and S and H their respective sets of values and histories.

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

The agents communicate using channels without message loss via:

- **ok?** messages which have as parameter an aggregate. They represent proposals of domains for a given set of variables and are sent from agents with lower priorities to agents with higher priorities. An agent sends **ok?** messages containing only domains in which the target agent is interested. He does not send domains for assignments he was proposed and he has never changed. If he has not just discarded a recent applicable nogood[1], then he sends only the domains for which he proposes a new modification now. **ok?** messages are also sent as answers to **add-link** messages.
- **nogood** messages which have as parameter an explicit nogood. A **nogood** message is sent from an agent with higher priority to an agent with lower priority, namely to the agent with the highest priority among those that have modified an assignment in the parameter. An empty parameter signals failure.
- **add-link**(vars) messages: sent from agent A^j to agent A^i (with $j > i$). They inform A^i that A^j is interested in the variables $vars$.

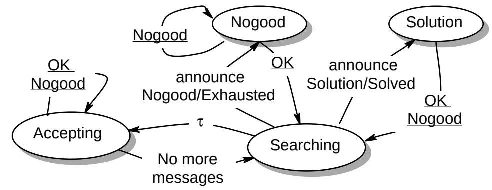

Figure 3: *Backtrack search procedure for each agent*

Each agent A^i owns a set of local constraints. The variables A^i is *interested in*, are those implied in its local constraints, called the *local variables* and those establishing links with other agents. The *current solution space* of A^i, denoted as C_{A^i}, is described by the local constraints, a list of explicit nogoods and a *view*.

Definition 4 (View) *The view of an agent* A^i *is an aggregate* (V, S, H) *such that* V *contains variables* A^i *is interested in.*

A view imposes restrictions on the original search space defined by the local constraints of an agent. It contains for each variable, the newest received assignment via **ok?** messages.

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

A tuple is *contained* in the current solution space of agent A^i if it satisfies the local constraints and is not contained in the explicit or entailed nogoods of C_{A^i}. The *current instantiation* of an agent A^i is a Cartesian product such that all its tuples are contained in C_{A^i}. The list of nogoods, respectively the view, of an agent A^i is updated by the **nogood**, respectively **ok?** messages it receives.

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

- AAS-2: is based on full nogood recording similarly to the AS algorithm of (Yokoo *et al.* 92).
- AAS-1: proceeds similarly to dynamic backtracking (Ginsberg & McAllester 94). It removes the nogoods depending on the instantiation of the modified variables, guaranteeing polynomial space complexity.

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

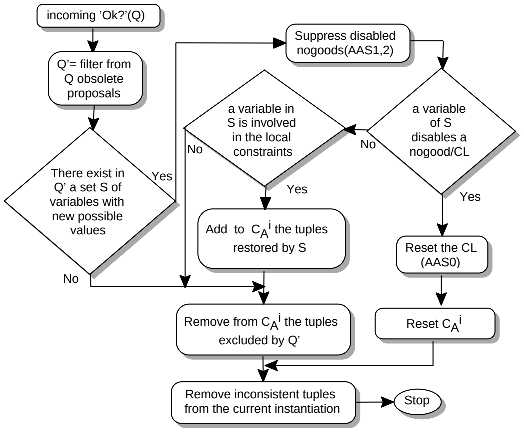

Figure 4: OK *procedure.*

- AAS-0: is a modification of AAS1 with less nogood recording. AAS0 is a novel algorithm which merges all the nogoods maintained by each agent of AAS1 into a single nogood using the relaxation rule:

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

  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. Each agent maintains a single explicit nogood which integrates each new incoming explicit nogood using the relaxation rule.

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

The core backtrack procedure for each agent is the same for the three algorithms. It is given by the finite state machine of Figure 3. At the beginning, each agent A^i is in the state `Searching` where it tries to generate a current instantiation from C_{A^i}. At any time in the state `Searching`, an agent can transit into the state `Accepting` where it accepts **ok?** or **nogood** messages. These cause the agent to execute the procedures Ok, respectively Nogood which update the local search space (i.e the views, the nogoods lists and the position in the search tree) according to the content of the messages. When, in the state `Searching`, its C_{A^i} is empty, the agent A^i announces a nogood and transits into the state `Nogood`. When, on the contrary, a local solution is found (i.e. a set of tuples can be extracted from C_{A^i}), the agent announces the instantiation by sending **ok?** messages to the concerned agents and transits into the state `Solution`. The current instantiation of the agent is known as long as it remains in the state `Solution`.

The three algorithms differ by the actions undertaken in the procedures Ok and Nogood, respectively described in Figures 4 and 5.

The procedure Ok treats incoming **ok?** messages. The parameter Q, of such a message is an aggregate. We say that a given assignment (x_j, set_j, h_j) of Q is *obsolete* if the view of the receiving agent contains a newer assignment for

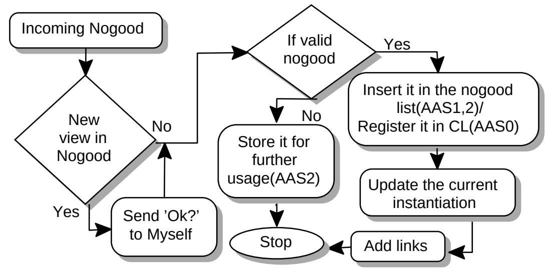

Figure 5: *Nogood procedure.*

x_j. The procedure Ok starts by filtering the obsolete assignments and then proceeds to updating the set C_{A^i} according to the remaining valid assignments. Suppose that one of these assignments offers a new possibility of valuation for an external variable x_j with respect to the current view. In AAS2 or AAS1 all the nogoods which do not take the new possibility into account will be *disabled*. In AAS1 this means that they will be removed. In AAS2 they will be marked and kept for an eventual further usage. In AAS0, if the nogood obtained by the relaxed inference rule contains such a variable but does not take the new value into account, the conflict list will be reset. Resetting C_{A^i} means that all the tuples allowed by the current nogoods and view are introduced in C_{A^i}. In the end, the previous instantiation can be updated and renewed.

The procedure Nogood treats incoming **nogood** messages. The argument, Q, of such a message is an explicit nogood. Let V be the view of the receiving agent. Suppose that there exists in Q, respectively in V, an assignment a_1, respectively a_2 for the variable x_j such that a_1 is newer than a_2. We will say that the nogood gives a *new view* for the variable x_j. In this case, the agent has to update its view by sending an **ok?** message to itself. An explicit nogood is valid if it concerns (i.e. invalidates) the current instantiation of the agent. If the received nogood is valid and if it contains variables that are unknown in the current view of the agent A^i, the procedure *Add links* will establish new links with all the agents $A^j, j < i$, for which these variables are local.

Solution Detection

In the existing asynchronous search algorithms, solutions are only detected upon quiescence[2]. This state is usually recognized using general purpose distributed mechanism (Chandy & Lamport 85). We have noticed that in the particular case of asynchronous search, solutions can be detected before quiescence. This means that termination can be inferred earlier and that the number of messages required for termination detection can be reduced. We have introduced a system message (not considered in the notion of quiescence) called **accepted** which informs the sender of an **ok?** message of the acceptance of its proposal:

- **accepted** messages are sent from an agent to all its predecessors (along all incoming links). If the agent has been an end agent, it also sends an **accepted** to the *system agent*,

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

- an **accepted** message has as parameter a Cartesian product obtained by intersecting the current instantiation of the sender with the parameters of the last **accepted** messages received from all its outgoing links[3],
- an **accepted** message is sent by an agent only when its parameter is non empty (i.e does not contain empty domains), all the outgoing links have presented an **accepted** message and the agent is in the state `Solution`,
- the agents checks whether to send **accepted** messages when they reach the state `Solution` or when they receive **accepted** messages.

accepted messages are FIFO ordered.

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

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

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

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

Message ordering

In asynchronous search (AS), the messages must respect a FIFO channel order of delivery to ensure correct termination (Yokoo *et al.* 92). Our algorithm requires a stronger condition to hold since the channel for each variable is no longer a tree but a graph. This means that several messages can arrive to the same agent, for changing the value of the same variable, through different paths of the graph. For example, in Figure 2 agent A^3 can receive messages concerning variable x_1 from both A^1 and A^2. An order must therefore be established between these kind of messages. In AS it is sufficient to maintain a counter, for the emitter, and include its value within each message sent in order to obtain a FIFO order of delivery. In our algorithm, we include such counters for all the agents that modify a given domain in the message. The history of changes is built by associating a chain of pairs $|a : b|$ to each variable of a message (see Figure 2). Such a pair means that a change of the variable's domain was performed by the agent with index a when its counter for the corresponding variable had the value b. The local counters are reset each time an incoming **ok?** changes the known history of the corresponding variable. It is incremented each time the agent proposes a change to the domain of that variable. To ensure correct termination, we use the next conventions: The history of changes where the agent with the smaller index or the counter with the larger value occurs first is the most recent. If a history is the prefix of the other, then the longer one is more recent.

[3]We define the intersection $S_i \cap S_j$ of two Cartesian products S_i and S_j as the Cartesian product of the union of all variables implied in S_i and S_j. The domain of each variable of $S_i \cap S_j$ is given by the intersection of its domains in S_i and S_j.

Correctness, Completeness, Termination

The detailed proofs are available at (WebProof 2000).

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

Summary of Proof. Correctness is an immediate consequence of Corollary 1.

The proof that quiescence is reached is close to the one given for AS in (Yokoo *et al.* 92), using the additional knowledge that only **ok?** messages could remove nogoods of the agent with the least priority among those implied in the hypothetical infinite loop.

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

Step 1 *After receiving the last* **ok?** *message and performing the subsequent search, either each agent A^i has a final instantiation that is consistent with its view, or failure is detected.*

Step 2 *At quiescence, the view of each agent A^i consists of the intersection of the instantiations of all instantiated agents $A^j, j < i$, for the variables it is interested in. This intersection corresponds, for each variable, to the newest received assignment.*

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

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

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

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

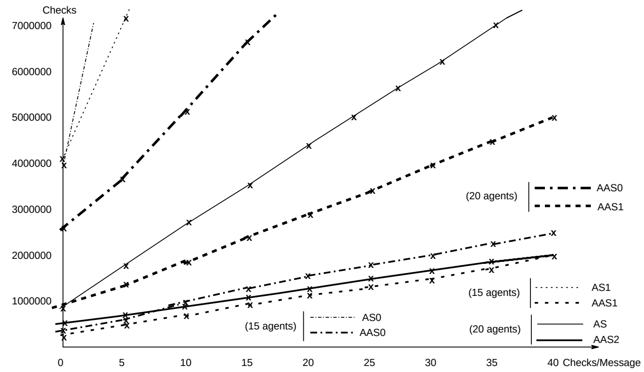

Figure 6: *Comparison of the number of checks on four sets of randomly generated problems near the peak. Abscissae select the relative time needed for sending a message divided by the time for a constraint check.*

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 in AS so that they can be compared with their variable-oriented counterparts. The size of domains is of 5 values and the problems are generated near the peak of difficulty (Cheesman, Kanefsky, & Taylor 1991) with a density of 30% and a tightness of constraints of 55%. The cost of search is evaluated using the longest sequence of messages and constraint checks. Each test is averaged over 50 instances. The measured parameters used for evaluation are the same as those given in (Yokoo *et al.* 92). In Figure 6, 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. AAS2 performs slightly better than AS. There are specific cases where AS performs better for finding the first solution. However, for discovering that no solution exists AAS2 performs steadily better than AS since the whole search space needs to be expanded. AAS2 also reduces the longest sequence of messages as well as the number of nogoods stored by a factor of 50% on average. AAS1 needs more messages than AAS2, and AAS0 even more. However, they do not present memory problems. 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.

Conclusion

We have presented AAS, a new asynchronous backtrack search technique which requires no artificial redistribution of constraints, allows for aggregating the information transmitted using a Cartesian product representation and includes an enhanced termination detection mechanism. AAS provides a natural support for enforcing privacy requirements on constraints. Its evaluation has been done using three different algorithms called AAS2, AAS1 and AAS0. AAS2 is based on full nogood recording while AAS1 and AAS0 are distributed variants of the centralized dynamic backtracking based on partial nogood recording. In particular, AAS0 is a novel algorithm which only stores a single nogood. The experiments have shown that the overall performance of AAS2 is comparable to that of AS (Yokoo *et al.* 92). AAS0 and AAS1 have more potential in practice since the space they require is bounded. Their evaluation have shown that aggregation is of interest for reducing the number of messages exchanged in distributed asynchronous search.

In the current implementation, the agents with the lower priority may have to reveal more information about their constraints. If undesirable, such a behavior can be avoided using random or cyclic agent reordering. Moreover, situations where some agents are forced to reveal their whole constraint are not precluded. This can occur, for example, in problems where all the agents but the last accept everything and the last one nothing. Malicious agents can form coalitions and create intentionally such problems in order to determine certain external constraints. In the future we plan to analyze the importance of these issues. We will also investigate how the dynamic change of constraints, which often occurs in human negotiation, can be integrated.

Acknowledgements

This work was performed at the Artificial Intelligence Laboratory of the Swiss Federal Institute of Technology in Lausanne and was sponsored by the Swiss National Science Foundation under project number 21-52462.97.

References

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

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

Ginsberg, M., and McAllester, D. 94. Gsat and dynamic backtracking. In J.Doyle., ed., *Proceedings of the 4th IC on PKRR*, 226–237. KR.

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

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

Hubbe, P. D., and Freuder, E. C. 92. An efficient cross product representation of the constraint satisfaction problem search space. In *Proc. of AAAI*, 421–427.

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

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

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