Openness in Asynchronous Constraint Satisfaction Algorithms

Marius-Călin Silaghi*
Florida Institute of Technology (FIT)
Melbourne, Florida 32901-6988, USA
msilaghi@cs.fit.edu

Boi Faltings
Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne, Switzerland
Boi.Faltings@epfl.ch

Abstract

Constraint satisfaction occurs in many practical applications. Recently, there has been increasing interest in *distributed* constraint satisfaction (DisCSP), where variables and constraints are distributed among several agents, and algorithms for finding solutions through asynchronous exchange of messages among agents.

An important reason for distributed problem solving is *openness*: allowing the problem to define itself dynamically through the combination of agents that participate in it. We investigate openness in complete asynchronous search algorithms for DisCSP, in particular the problem of agents joining and leaving a search process in progress without destroying consistency for the other agents, and give complete search algorithms that satisfy this property.

1 Introduction

Constraint satisfaction has been applied with great success to many important practical applications. Increasingly, it also occurs in distributed settings with variables and constraints defined by different agents. There has thus been significant interest in the distributed constraint satisfaction (DisCSP [29]) problem with asynchronous, distributed algorithms for solving it.

A major characteristic of distributed systems is their *openness*: the combination of agents making up the system is not built into the algorithms, and may not even be known when the algorithm is running, and agents may even be joining and leaving the system while an algorithm is active. Algorithms for DisCSP have focussed on asynchronous execution, but have not considered openness to a great extent.

In particular, asynchronous search algorithms for DisCSP assume that the set of agents, variables and constraints is known at the beginning of the search and fixed throughout. In this paper, we show how to create the possibility for agents to join and leave an ongoing search process without halting it, and reusing the results of the ongoing search as much as possible.

In general, constraint satisfaction algorithms can be easily adapted to allow additions of constraints during search. Similarly, it turns out to be straightforward to allow new agents to introduce their variables and constraints in DisCSP search without need to restart an ongoing asynchronous search process.

It is however considerably more complex to allow an agent to leave and remove its variables and constraints, since this may make valid assignments which had earlier been discarded. For the centralized case, Bessiere ([3]) has shown how to dynamically put back values eliminated by consistency techniques, so that search can restart with the variable or constraint that has been removed rather than from the beginning. Verfaille and Schiex ([6]) has shown techniques for adapting complete solutions to changes in the constraint system. Both techniques only address centralized search algorithms, and it is not clear how they could be generalized to an asynchronous case.

A central innovation we present in this paper is a relaxation technique for asynchronous, distributed search algorithms that allows us to reactive the right parts of the search space when agents and their associated variables and constraints leave an ongoing asynchronous search.

*This work was performed while the first author was working at EPFL, supported by the Swiss National Science Foundation project number 21-52462.97.

We first introduce the asynchronous search algorithm and the extra structures required to allow reordering and removal of agents. We then show how we can allow agents to leave the search by reordering them into the last position of the search hierarchy, and then dropping them from the search. We develop the technique into an open protocol where agents can join and leave an asynchronous search. Finally, we also investigate the case where agents leave without announcing it, such as when an agent crashes.

2 Background

The first complete asynchronous search algorithm for DisCSPs is the Asynchronous Backtracking (ABT) [28]. For simplicity the approach in [28] considers that each agent maintains exactly one distinct variable. More complex cases were considered later [30]. The first version of ABT has requested the agents to store all the nogoods, but this requirement was removed in [29, 10, 21], where versions of ABT with polynomial space-complexity are mentioned. The version in [7] supports agents owning Dynamic CSPs. Some definitions of DisCSPs have considered what happens in the case where the knowledge and interest on constraints is distributed among agents [31, 23, 21]. [23] proposes a static ordering and distribution in ABT that fits the natural structure of a real problem (the nurse transportation problem). The Asynchronous Aggregation Search (AAS) [21] algorithm actually extends ABT to the case where the same variable can be instantiated by several agents. AAS offers the possibility to aggregate several branches of the search. An aggregation technique for DisCSPs is presented in [15] and allows for simple understanding of the privacy/efficiency mechanisms. The order on variables in distributed search was so far addressed in [4, 1, 27, 23, 9, 21]. In what concerns distributed consistency algorithms, one of the first is presented in [31].

After the design of asynchronous backtracking, work has mainly concentrated on the intelligence of the backtracking by tuning the quality of the nogoods selected for storage [10, 11, 25, 21], and on algorithms for *achieving consistency* [2, 8]. The first algorithm for asynchronous *maintenance of consistencies* is presented in [21]. Performing asynchronous reordering was first proposed in [26] according to a min-conflict heuristic. The first polynomial-space complete asynchronous search algorithm with reordering is presented in [21], where its integration within asynchronous maintenance of consistency is also described. Asynchronous use of abstractions in complete search is first described in [21]. Issues on openness in incomplete search algorithms are discussed in [12, 16, 5].

2.1 Constraint Removal

Agents that leave a process are typically involved in many structures maintained by other agents: outgoing-links, assignments, explicit and consistency nogoods. To cleanly remove agents, the structures maintained by protocol have to be updated. An elegant way of isolating an agent A_i is to reorder it to the position with the lowest priority. Then, no other agent owns assignments issued by A_i, and no agent maintains structures storing consistency nogoods at corresponding search levels. Additional mechanisms are used for eliminating outgoing-links, nogoods inferred from internal constraints of A_i, and consistency nogoods generated by A_i.

Several researchers have studied constraint removal in the framework of centralized consistency achievement and maintenance. [18] proposes a natural and simple solution for AC3. [3] maintains more information with AC4, namely a separate justification for each eliminated value. This enables to recover some more information when a constraint predicate is removed. At the other extreme, [17] gives a technique which intelligently reuses much of the existing work without storing any additional data during computation of consistency. Open systems for problem solving are discussed in [19] and [24].

The Distributed Constraint Satisfaction Paradigm (DisCSPs) is defined in [29]. Some well known extensions are Partial Distributed Constraint Satisfaction [27] for approaching over-constrained problems, and Distributed Dynamic Constraint Satisfaction [7, 16, 12] for modeling dynamic local problems.

2.2 Asynchronous Search

We introduce now the main notions involved in asynchronous search algorithms. The agents run concurrently

and asynchronously, exchanging some types of proposals. A total order is defined on agents, and it is used to break ties in case of conflicts during search. By $A_j^i(o)$ we denote the agent A_j having position i in the order o on agents. If $i>j$ then A^i has a *lower priority* than A^j and A^j has a *higher priority* then A^i.[1]

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

The tags timestamp the assignments and allow for an order, called "stronger", on assignments generated by the same agent. The *strongest* assignments received by different agents form their *agent_view*.

In ABT, each constraint C has to be evaluated by a predefined agent:

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

In MAS, the assignments are called aggregates. The corresponding rule for MAS is generalized to:

Rule 2 *A constraint C can be enforced by the lowest priority agent, A_i, or one of its successors, if for each variable involved in C, either A_i or one of its predecessors can propose aggregates.*

This means that any agent that wants to enforce a constraint C has to propose assignments for any variable in C for which no higher priority agent proposes assignments. The set of predicates enforced by A_i is denoted CSP(A_i). Each agent holds a list of **outgoing links** represented by a set of agents. Links are associated with constraints. Every link is directed from the value sending agent to a corresponding constraint-evaluating-agent.

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

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

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

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

Rule 3 (Nogood-Evaluating-Agent) *An explicit nogood $\neg N$ is enforced by the lowest priority agent that has generated an assignment in N.*

The following types of messages are exchanged in ABT:

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

The agents start by instantiating variables concurrently and send **ok?** messages to announce their assignment to all agents with lower priority in their *outgoing links*.

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

Definition 5 *A* **nogood is invalid** *if it contains invalid assignments.*

In the described version of ABT, only one valid explicit nogood has to be stored for a value of a variable, but more nogoods can be stored for efficiency reasons.

MAS extends ABT with support for aggregations (abstractions), consistency, and reordering. In MAS, assignments become aggregates [21]. An aggregate $\langle x, v, h\rangle$ uses as **tag value** a *trace*, h, which generalizes the concept of the counter. The *trace* is a marker for ordered proposal sources, technique (used first time in [20]).

2.2.1 Traces

We describe 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, an assignment) [20]. Its philosophy is related to the one of logical clocks [13].

Definition 6 *A* **proposal source** *for a resource $\mathcal{R}$ is an entity (e.g. a delegated agent) that can make specific proposals concerning the allocation (or valuation) of $\mathcal{R}$.*

We consider that an order $\prec$ is defined on *proposal sources*. The *proposal sources* with lower position according to $\prec$ have a higher priority. The *proposal source* with position k is noted $P_k^{s\mathcal{R}}$, $k \geq 0$.

Definition 7 *A* **conflict resource** *is a resource for which several agents can make proposals in a concurrent and asynchronous manner.*

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

Definition 8 *A* **trace** *is a sequence h of pairs, $|a{:}b|$, that is associated to a proposal for $\mathcal{R}$. A pair $p{=}|a{:}b|$ in h signals that a proposal for $\mathcal{R}$ was made by $P_a^{s\mathcal{R}}$ when its $C_a^{\mathcal{R}}$ had the value b, and it knew the prefix of p in h. A proposal source only knows traces proposed by proposal sources with higher priority than itself.*

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

Definition 9 *A trace h_1* **is stronger than** *a trace h_2 if a lexicographical comparison on them, using the order $\propto$ on pairs, decides that h_2 precedes h_1, $h_2 \propto h_1$.*

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

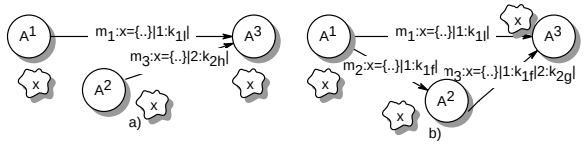

Figure 1: Simple scenarios with messages for proposals on a resource, x.

Definition 10 *A trace h_1 built by $P_i^{s\mathcal{R}}$ for a proposal is* **valid** *for an agent A if no other trace h_2 is known by A (eventually known as prefix of a trace h_2') such that h_2 is stronger than h_1 and was generated by $P_{\leq i}^{s\mathcal{R}}$.*

For example, in Figure 1 the agent A^3 may get messages concerning the same resource x from A^1 and A^2 and has to decide which of them is the most up to date. In Figure 1 a), if the agent A^3 has already received m_1, it will always discard m_3 since the *proposal source* index has priority. However, in the case of Figure 1 b) the message m_1 will be maintained only if $k_{1f}{<}k_{1l}$. By construction, no trace contains two pairs with the same *proposal source* index. Therefore, in each message, the length of the trace for a resource is upper bounded by the number of *proposal sources* for that *conflict resource* [20].

2.2.2 Multiply Asynchronous Search (MAS)

For domain splitting, the proposal sources are the agents. Each agent considers that the proposal sources for domain splitting are ordered according to the order it currently knows. For reordering, the proposal sources are a set of abstract agents, that can be dynamically delegated to existing physical agents. The order among the proposal sources on reordering (the abstract agents) is static. The proposal source with position k, $k{\geq}0$ is referred to as R^k, *the k-th reordering leader*. R^k can reorder agents found on positions higher than k by proposing a set of guidelines that have to be complied with by lower priority reordering sources (e.g. the agent on position $k{+}1$), and proposes an order. The notation $A_j^i(o)$ tells that the agent A_j has position i in the current order o. Each agent has a different perception of the distributed search, as a search tree.

Definition 11 *The depth in distributed search trees is referred to as* **level**.

In order to maintain consistencies asynchronously, MAS uses **consistency nogoods**.

Definition 12 *A* **consistency nogood** *for a level k and a variable x has the form $V \rightarrow (x \in l_x^k)$ or $V \rightarrow \neg(x \in s \backslash l_x^k)$. V is an aggregate and may contain for x an assignment $\langle x, s, h \rangle$, $l_x^k \subset s$. Any assignment in V must have been proposed by predecessors of A^k. l_x^k is a label, $l_x^k \neq \emptyset$.*

An aggregate $\langle x, s, h \rangle$ in V is redundant if $s = l_x^0$.

Definition 13 *An* order *in MAS is an ordering proposal on agents, and a delegation of proposal sources for reordering, all these being marked with a marker for ordered proposal sources.*

Besides the messages used in ABT, MAS uses the following types of messages:

- **propagate** message transporting a set of consistency nogoods.
- **heuristic** message transporting information useful for reordering.
- **reorder** message transporting an order.

A **propagate** message is sent by the agent that infers the consistency nogoods in parameter to the agents that see the corresponding level of the search. Some versions only send **propagate** messages to agents A_i for which CSP(A_i) involves the variables in the transported labels [21]. Each consistency nogood is tagged with its level and the value of a counter $C_{i,x,k}$ maintained by A_i, incremented on consistency nogoods generation by the inferring agent A_i, for variable x, at level k.

A **heuristic** message is sent by an agent A^k to $R^{>j}$, and can be sent within delay t_h after it receives a new order, aggregate or label for a level no higher than j. The exact schema depends on the used heuristic.

A **reorder** message is sent by a proposal source for reordering, R^k, to all reordered agents $A^{>k}$ and optionally to lower priority proposal sources for reordering, $R^{>k}$. It can be sent within delay t_r after a **heuristic** message is received, or from the start of the search.

2.2.3 Reordering

The ordering on agents is considered as a *conflict resource* and a *trace* is attached to each proposal on ordering. As previously mentioned, the *proposal sources* for the ordering on agents are the agents R^i, where $R^i \prec R^j$ if $i<j$. R^i is the *proposal source* that when knowing an ordering, o, can propose orderings that reorder only agents on positions p, $p>i$. To specify that two notations: A_i, R^j, refer to the same physical agent, we use the notation $A_i \equiv R^j$. For dealing with reordering, each agent stores two orders: the strongest known order and the proposed order.

Definition 14 (Known order) *The ordering known by R^i (respectively A^i) is the order o with the strongest trace among those proposed by the agents $R^k, 0 \leq k < i$ and received by R^i (respectively A^i). A^i has the position i in o. This order is referred to as the* known order of R^i *(respectively A^i).*

Definition 15 (Proposed order) *An ordering, o,* proposed by R^i *is such that the agents placed on the first i positions in the* known order of R^i *must have the same positions in o. o is referred to as the* proposed order of R^i.

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

Definition 16 (Reorder position) *Let u be the lowest position such that p_1^u and p_2^u are different and let $v = |h_1|$. The* **reorder position** *of h_1 and h_2 is either $min(a_1^u, a_2^u) + 1$ if $u > v$, or $a_2^{v+1} + 1$ otherwise. This is the position of the highest priority reordered agent between h_1 and h_2.*

An agent R^i announces its proposed order o by sending **reorder** messages to all agents A^k, $k>i$, and to all agents $R^k, k>i$. Each agent A_i and each agent R^i has to store an ordering denoted C^{ord}. C^{ord} is the ordering with the strongest trace that was received. For allowing asynchronous reordering, each **ok?** and **nogood** message receives as additional parameter an order and its trace. The **ok?** messages hold the strongest *known order*

of the sender. The **nogood** messages hold the order in the C^{ord} at the sender A^j that A^j believes to be the strongest *known order of* the receiver, A^i. This ordering consists of the first i agents in the strongest ordering known by A^j and is tagged with a trace obtained from the trace of its C^{ord} by removing all the pairs $|a{:}b|$ where $a{\geq}i$.

When a message is received which contains an order with a trace h that is stronger that the trace h^* of C^{ord}, let the *reordering position* of h and h^* be I^r. The assignments/aggregates for the variables x^k, $k \geq I^r$, are invalidated. This simple approach loses many nogoods by invalidation, but techniques to recover some of these nogoods are described in [21].

3 Open System

For simplification, here we consider that all sent messages arrive in finite time to their destination, when the destination does not leave or suffer a crash. E.g. using TCP connections – when on the failure of such a connection, one should make sure that the destination agent eventually leaves or starts crash recovery procedures. Something similar to this is implemented in RETSINA [24].

3.1 Joining Asynchronous Search

It is very easy to allow new agents to join a search process since all existing work remains valid. When new agents want/accept to get involved in the computation, all the existing agents should receive, via an **involved** message, information on

- the initial priority, and
- the external variables of the new agents.

The agent that receives an **involved** message have to send their valid proposals, order, as well as their last generated valid labels, to the new agents. The simplest solution is to place new agents on positions following existing agents. Protocols supporting agent reordering (such as AWC, ABTR, MAS) allow then for reordering agents toward any other wished configuration.

Proposition 1 *If the first message sent by newly joining agents is sent after all previous agents have received the corresponding* **involved** *message, any running protocol that is an instantiation of MAS remains correct, complete and terminates.*

Proof. The proposition is obvious since the protocol obtained by this combination behaves as an instance of MAS where t_h is higher than the time up to the involvement of the new agents, some messages are delayed for this period of time, and channels discard invalidated messages. □

When all agent answers **involved** messages with an acknowledgment toward the new agents, the last ones can straightforwardly detect the condition in the assumptions of Proposition 1. Alternatively, the treatment of messages could be adapted to ensure correct treatment of messages from unexpected agents. It has to be noted that termination and solution detection algorithms [14, 21] that continuously monitor the system have to be adapted for taking into account the insertion of the new agents.

3.2 Leaving Asynchronous Search

3.2.1 Isolating an agent

Our first step towards removing an agent consists in isolating it from search. Given the reordering capabilities of MAS, it becomes easily possible to place any leaving agent, A_i, on the last position before removing outgoing-links and triggering the relaxation of the nogoods that A_i has generated.

We describe here the case where the agents agree on the convention: $A^j{\equiv}R^j$. Each reordering leader R^i stores the set L of agents that have left. Let us consider the case when an agent A^i leaves and N agents remain in the search process. Let $q{=}\min(i{-}1, N{-}2)$. When R^q knows that an agent A^i_j leaves, $i{\leq}N$, then R^q has to reorder A^i_j. If the *known order of* R^q specifies the sequence of agents: $A^1_{p_1}, ..., A^{i-1}_{p_{i-1}}, A^i_j, A^{i+1}_{p_{i+1}}, ..., A^{N+1}_{p_{N+1}}$, then the agent R^q has to broadcast the message **reorder**$(A^1_{p_1}, ..., A^{i-1}_{p_{i-1}}, A_{p_{i+1}}, ..., A_{p_{N+1}}, A_j)$ to all the agents $A_j, A_{p_{i+1}}, ..., A_{p_{N+1}}$. This order is tagged with a trace, as previously discussed. Any new proposed order should put the set L at the end of the sequence of agents.

3.2.2 Nogood management

We now describe a constraint relaxation schema for private constraints[2]. That technique consists of explaining inferences with references to constraints (CR). To enhance privacy support, the CRs do not necessarily stand for a given constraint, but provide a way to signal when due to relaxations, a nogood is invalidated. At any inference, the nogood resulting from the inference is tagged with the union between:

- the CRs of the constraints used for inference, and
- the union of CRs tagging the nogoods used for inferences.

Each agent is also associated with a predefined CR, CR(A_i). The algorithms remain polynomial in space only if the number of CRs in use is bounded. The reuse of CRs can be enabled by attaching to them counters. When CRs are obtained from A_i, the versions with lower value of the counters cannot be used any longer, being transformed into CR(A_i).

In Figure 2 is given an example of ABT with maintenance of CRs. CR(A_i) is denoted by C^{A_i}. The example uses the simple problem exploited in [28] to illustrate ABT. The agent A_3 enforces the constraints $x_1{\neq}x_3, x_2{\neq}x_3, x_3{\in}\{1,2\}$. Each of these constraints can be represented with distinct CRs: $\{C_1^{A_3}, C_2^{A_3}, C_3^{A_3}\}$. For privacy reasons, A_3 can use more than three CRs, having several CRs for the same constraint. The separate relaxation of some of these constraints can be announced by broadcasting the set of CRs representing them. Therefore, each agent discards the nogoods tagged by CRs of relaxed constraints.

For simplicity, each agent in the example of Figure 2 uses only one CR. The agent A_3 infers the nogood $\neg(\langle x_1,1,1\rangle, \langle x_2,2,1\rangle)$ and tags it with C^{A_3}, (CR(A_3)). When the agent A_2 infers $\neg(\langle x_1,1,1\rangle, \langle x_2,2,1\rangle)$, it tags this nogood with $\{C^{A_2}, C^{A_3}\}$ by adding C^{A_2} due to its constraint: $x_2{\in}\{2\}$.

3.2.3 Agents Announcing their Retreat

We see the departure of an agent A_i as a relaxation, namely the removal of the constraints of A_i. Obviously, the remaining agents need additionally to remove the links that they have towards A_i and also eliminate the corresponding data structures (assignments and set of labels generated by A_i for different variables). The removal of the occupied position, p, can also be realized easily in MAS if the reordering leader R^{p-1} generates a **reorder** message which places that agent at the end of the search. When the agent delegated to act for R^0 withdraws, the remaining agents have to reach a consensus on a new delegation of R^0. Such a consensus can be easily obtained using the convention that the agent on the first position given the last order among the remaining agents, (e.g. A^1 if it did not leave), will act for R^0.

A new message, **leaving**, has to be used for signaling departure. When A_i leaves the search, it broadcasts **leaving**(A_i,order,R^0) to all other agents. The message takes as parameter the strongest order and the identity of R^0 known by the sender.

To know which CR belongs to which agent, CRs are tagged with the name of the owner agent. To enable the detection of the nogoods that depend on any leaving agent, any generated nogood has to be marked with the corresponding CRs. Whenever an agent, A_i, leaves the process, this information can be broadcasted to all the agents and the nogoods marked with CRs of A_i must be removed by everybody.

Figure 3 shows an example where A_2 leaves during the example in Figure 2. Normally, R^1 should undertake the task of reordering the agents, but since R^1 disappears when the number of agents reduces to 2 [21], the task is undertaken by R^0. The Figures 4 and 5 show the case

9: $\mathbf{A_2}/A^2$	**leaving**$(A_2, \|\|, A_1)\longrightarrow$	$\mathbf{A_1}/A^1/R^0/R^1$
10: $\mathbf{A_2}/A^2$	**leaving**$(A_2, \|\|, A_1)\longrightarrow$	A_3/A^3
11: $\mathbf{R^0}/A_1/A^1$	**reorder**$(A_1, A_3)\|0:1\|\rightarrow$	A_3/A^2

Figure 3: Example of ABTR when A_2 leaves the search. The default order is (A_1, A_2, A_3), $R^k \equiv A^k$, and initially $R^0 \equiv A_1$.

9: $\mathbf{A_2}/A^2/R^0$	**leaving**$(A_2, \|\|, A_1)\longrightarrow$	$\mathbf{A_1}/A^1/R^1$
10: $\mathbf{A_2}/A^2/R^0$	**leaving**$(A_2, \|\|, A_1)\longrightarrow$	A_3/A^3
11: $\mathbf{R^0}/A_1/A^1$	**reorder**$(A_1, A_3)\|0:1\|\rightarrow$	A_3/A^2

Figure 4: Example of ABTR when A_2 leaves the search. The default order is (A_1, A_2, A_3), $R^k \equiv A^k$, and initially $R^0 \equiv A_2$.

[2]Introduced in [22].

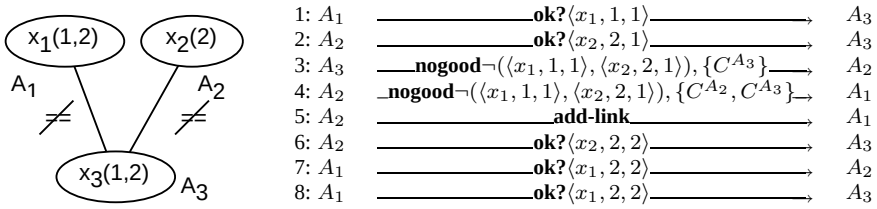

Figure 2: Example of ABT with maintenance of CRs.

9: $\mathbf{A_1}/A^1/R^0/R^1$ **leaving**$(A_1, ||, A_2)$ $\mathbf{A_2}/A^2$
10: $\mathbf{A_1}/A^1/R^0/R^1$ **leaving**$(A_1, ||, A_2)$ A_3/A^3
11: $\mathbf{R^0}/A_2/A^1$ **reorder**$(A_2, A_3)|0:1|$ A_3/A^2

Figure 5: Example of ABTR when A_1 leaves the search. The default order is (A_1, A_2, A_3), $R^k \equiv A^k$, and initially $R^0 \equiv A_1$.

where the agent delegated to act for R^0 leaves. In both cases, the first remaining agent is delegated to act for R^0 and generates the new order. In case $R^0 \equiv A_3$, no new order needs to be generated, but A_1 is delegated to act for R^0.

While all the links, nogoods and assignments for leaving agents are removed when the corresponding **leaving** message is received, those agents are still stored in the sets L of each agent.

3.2.4 Leaving without Announcing

Agents may leave without announcing. As long as nobody detects this departure, the search continues to use the nogoods inferred from the internal constraints of the disappeared agents. The dependencies continue to propagate and may lead to the replacement of valid nogoods with nogoods that depend on withdrawn agents. It is therefore important to detect such withdrawal as soon as possible.

When no time-out is established, one cannot ensure the achievement of any solution. If a time-out t_t is agreed-on, any agent that recovers after this time-out elapses will have to join as a new agent, and much information can be lost.

ABTR allows the agents to re-delegate R^0 during a predefined delay $t_r + t_h$ from the beginning of the search. Moreover, the withdrawal of the agent acting for R^0 also leads to the re-delegation of R^0. When the timeout is detected for an agent A_i, the system detecting it cannot be sure in asynchronous search whether A_i is or is not acting for R^0. This problem can be solved cleanly with a two rounds protocol, assuming that no other agent crashes during them. The first round consists in sending **leaving**$(A_i, \emptyset, \emptyset)$ messages to all remaining agents. When an agent A_j receives **leaving**$(A_i, \emptyset, \emptyset)$ without an order from an agent $A_k, k \neq i$, A_j will trigger the elimination of any message coming from A_i, but will not start acting for R^0, even if $A_j \equiv A^1$. Instead A_j sends a message **leaving-data**(A_i) to A_k attaching to it the strongest order known at A_j, and the estimated identity of R^0.

Proposition 2 *If A_k receives the answer* **leaving-data***(A_i) from all remaining agents, and if A_i is R^0 in the strongest received ordering, then the leaving agent is R^0.*

Proof. After any agent A_j receives a **leaving**(A_i), A_j will discard any new information generated by A_i. If A_i is R^0, it can no longer change it at A_j since any such change is discarded by A_j. □

After A_k receives **leaving-data**(A_i) from all remaining agents, if A_i is R^0 in the strongest received order, then A_k broadcasts **leaving**(A_i) with the strongest received order and identity for R^0.

If some other agent, A_u, does not answer to **leaving** messages, the removal procedure is interrupted after the corresponding time-out, A_k launches the protocol for announcing that both A_u and A_i have left (Algorithm 1). If A_k abandons himself without notice, any agent that has received from A_k an **leaving**(A_i) without an order and did not receive a **leaving**(A_i) with order, timeouts A_k after a delay $2t_t$ and starts the protocol for announcing the departure of both A_i and A_k.

procedure *elimination(*$A : \{A_i, ...\}$*)* **do**
broadcast **leaving**($A, \emptyset, \emptyset$);
wait until receives all **leaving-data**(A,order,R^0);
or timeout(t_t);
if *no timeout* **then**
broadcast **leaving**(A,strongest-order,R^0)
else
restart elimination procedure for agents that did not answer and for A
end
end do.
when A_k *detects time-out for* A_i **do**
elimination($\{A_i\}$);
end do.
when A_j *receives* **leaving***(*$A, \emptyset, \emptyset$*) from* A_k **do**
block A;
answer with **leaving-data**(A,strongest-order,R^0);
discard data from A_i or tagged C^{A_i}, $A_i \in \{A_i\}$;
launch timer $2t_t$ for $\{A_k\} \cup A$;
end do.
when A_j *receives* **leaving***(*A*,order,*R^0*) from* A_k **do**
block A;
discard data from $\{A_i\}$ or tagged C^{A_i}, $A_i \in \{A_i\}$;
if $A_j \equiv A^u$*, all* $A^{<u}$ *have left, and* $A_i \equiv R^0$ **then**
$A_j \leftarrow R^0$;
end
stop $R^u, u \geq N-1$, N-the nr. of remaining agents;
while $A_i \in A$, $A_i \equiv A^v$ *and (*$(v=N \wedge A_j \equiv R^{v-2})$ *or* $(v<N \wedge A_j \equiv R^{v-1})$*)* **do**
send to owned delegated reordering proposal sources: **heuristic**(reorder A_i as A^{N+1});
end
stop timer $2t_t$ for $\{A_k\} \cup A$;
end do.
when *timer* $2t_t$ *for* A **do**
elimination(A);
end do.

Algorithm 1: Procedures for eliminating a set of agents A after time-out is detected by A_k.

Unfortunately this technique leads to important losses when an agent cannot be reached for long time only due to network congestion. To reduce these problems, a longer timeout, T, can be established. Systems that are unreachable in acceptable time t, $t<T$, and that may have lost some messages, can be updated with a recovery mechanism similar to the one given in [21].

4 Main Problems and Research Directions

For existing complete search protocols with polynomial space requirements, the losses that it can incur when agents withdraw can vary from very little to almost everything. The worst cases are expected to occur either when a high priority agent withdraws, or when an agent involved in most stored nogoods withdraws. The last case is very likely towards the end of a search process for difficult problems.

The main advance that can be foreseen towards an improved response to openness in complete search protocols is the definition of some data structures to reduce the loss of information when certain patterns of events take place. Such a pattern is the withdrawal of only one agent. A possible strategy of A_j can consist of storing for each agent A_i, all the last valid labels where it was not involved in the inference process (labels and nogoods not tagged with CR(A_i)). The space complexity would increase with a factor n (the number of agents). Also the local worst case computation cost of the agents can increase by the same factor. Namely, each time a new valid label arrives, it has to be combined with all existing labels for agents not involved (as shown by tags) in its inference.

5 Conclusion

In this paper, we have shown how asynchronous search algorithms, in particular AAS, can be provided with extra structures that develop them into open protocols where agents can join and leave ongoing DisCSP search processes without need for restarting them. This allows asynchronous search to be used in an open environment with a dynamic agent population, which was not possible with known algorithms.

The main challenge we solve is to allow agents to leave a search process without restarting it. Our technique is based on the realization that agents can leave a search easily if they are at the last position in the hierarchy. Thus, leaving a search happens in two stages: first, the agent is reordered to the last position, then it can leave the search.

We have presented the techniques in the context of MAS (multiply asynchronous search), which can be seen as a generalization of many known asynchronous search algorithms such as ABT, AAS, ABTR or DMAC. It should thus be clear how the technique could be applied to these algorithms as well.

References

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

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

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

[4] Z. Collin, R. Dechter, and S. Katz. Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*, 2000.

[5] J. Denzinger. Tutorial on distributed knowledge based search. IJCAI-01, August 2001.

[6] G. Verfaillie et T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proc. of AAAI'94*, pages 307–312, Seattle (WA), 1994.

[7] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. Distributed configuration as distributed dynamic constraint satisfaction. In *Proc. of 14th Int. Conf. on Industrial and Engineering Applications of AI and Expert Systems*, volume 2070 of *LNAI*, pages 434–444, Budapest, June 2001. Springer.

[8] Y. Hamadi. Optimal distributed arc-consistency. In *Proceedings of CP'99*, Oct 1999.

[9] Y. Hamadi and C. Bessière. Backtracking in distributed constraint networks. In *ECAI'98*, pages 219–223, 1998.

[10] W. Havens. Nogood caching for multiagent backtrack search. In *AAAI CA Workshop*, 1997.

[11] K. Hirayama and M. Yokoo. The effect of nogood learning in distributed constraint satisfaction. In *Proceedings of ICDCS-2000*, pages 169–177, 2000.

[12] Hyuckchul Jung, Milind Tambe, Weixiong Zhang, and Wei-Min Shen. On modeling argumentation as distributed constraint satisfaction: Initial results. In *Proceedings of the International Workshop on Distributed Constraint Satisfaction*, pages 47–56. CP'00, 2000.

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

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

[15] P. Meseguer and M. A. Jiménez. Distributed forward checking. In *CP DCS Workshop*, 2000.

[16] Pragnesh Jay Modi, Hyuckchul Jung, Milind Tambe, Wei-Min Shen, and Shriniwas Kulkarni. Dynamic distributed resource allocation: A distributed constraint satisfaction approach. In *Distributed Constraint Reasoning, Proc. of the IJCAI'01 Workshop*, pages 73–79, Seattle, August 2001. IJCAI.

[17] Bertrand Neveu and Pierre Berlandier. Arc-consistency for dynamic constraint problems: An RMS-free approach. In Thomas Schiex and Christian Bessiére, editors, *Proceedings ECAI'94 Workshop on Constraint Satisfaction Issues raised by Practical Applications*, Amsterdam, 1994.

[18] P. Prosser, C. Conway, and C. Muller. A distributed constraint maintenance system. In *Les Systèmes Experts et leurs Applications*, Avignon, France, 1992.

[19] Anita Raja, Victor Lesser, and Thomas Wagner. Towards robust agent control in open environments. In *Proc. of AA2000*, pages 84–91, Barcelona, June 2000.

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

[21] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Polynomial space and complete multiply asynchronous search with abstractions. In *IJCAI-01 DCR Workshop*, pages 17–32, Seattle, August 2001.

[22] M.-C. Silaghi, Djamila Sam-Haroud, and Boi Faltings. Maintaining hierarchical distributed consistency. In *Workshop on Distributed CSPs*, Singapore, September 2000. 6th International Conference on CP 2000.

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

[24] Katia Sycara. Multi-agent infrastructure for agent interoperation in open computational environments. In *Proc. of IAT*, 2001. Invited Talk.

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

[26] M. Yokoo. Constraint relaxation in distributed constraint satisfaction problem. In *ICDCS'93*, pages 56–63, June 1993.

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

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

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

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

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