Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering

Marius-Călin Silaghi, Djamila Sam-Haroud, and Boi Faltings

Swiss Federal Institute of Technology Lausanne
1015 Ecublens,Switzerland
{Marius.Silaghi,Djamila.Haroud,Boi.Faltings}@epfl.ch

Abstract. Existing Distributed Constraint Satisfaction (DisCSP) frameworks can model problems where a)variables and/or b)constraints are distributed among agents. Asynchronous Backtracking (ABT) and Asynchronous Weak Commitment (AWC) are the first asynchronous complete algorithms for solving DisCSPs of type a. The order on variables is well-known as an important issue for constraint satisfaction and is the main strength of AWC. The previous polynomial space asynchronous search algorithms are extensions of ABT and require for completeness a static order on their variables. Here, we show first how agents can asynchronously and concurrently propose reordering in ABT while *maintaining the completeness* of the algorithm with polynomial space complexity. Reordering has various applications (e.g. security, efficiency). Finding *time*-efficient reordering heuristics for search in DisCSPs is not obvious. We then show how an efficient reordering heuristic inspired from the one used in AWC can be integrated in the new search protocol.

1 Introduction

Distributed Constraint Satisfaction (DisCSP) is a powerful framework for modeling distributed combinatorial problems. A **DisCSP** is defined in [14] as: a set of agents $A_1, ..., A_n$ where each agent A_i controls exactly one distinct variable x_i, and each agent knows all constraint predicates relevant to its variable. This is an acceptable simplification since the case with more variables in an agent can be easily obtained from it. Asynchronous Backtracking (ABT) [14] is the first *complete* and *asynchronous* search algorithm for DisCSPs. A simple modification was mentioned in [15, 5] to allow for versions with polynomial space complexity.

The completeness of ABT is ensured with help of *a static order* imposed on agents. ABT is a slow algorithm and [12] has introduced a new algorithm called Asynchronous Weak Commitment (AWC). AWC allows agents to reorder themselves asynchronously by decreasing their position when they are overconstrained. AWC has proven to be more efficient than ABT but its requirement for an exponential space could not be eliminated without losing completeness.

So far, no asynchronous algorithm has offered the possibility to perform reordering without losing either the completeness, or the polynomial space property. In this paper we first describe an extension of ABT that allows the agents to asynchronously and concurrently propose changes to their order. This extension is called ABT with Asynchronous Reordering (ABTR). We prove that the completeness of ABTR is ensured with polynomial space complexity. ABTR is required in various applications (e.g. security [9]). Another typical use of reordering is for efficiency. It is not obvious to find *time*-efficient reordering heuristics for DisCSPs. The reordering heuristic of AWC has proven to be enough cheap and efficient to fit distributed algorithms but it cannot be directly used in ABTR. We end this article by showing how an efficient heuristic with similar behavior as the one in AWC can be obtained and its performance is evaluated.

This is the first asynchronous search algorithm that allows for asynchronous dynamic reordering while being complete and having a polynomial space complexity. Here we have built on ABT since it is an algorithm easier to describe than its subsequent extensions. The technique can nevertheless be integrated in a straightforward manner in most extensions of ABT.

2 Related Work

The first complete asynchronous search algorithm for DisCSPs is the Asynchronous Backtracking (ABT) [14]. The approach in [14] considers that each agent maintains only one distinct variable. More complex definitions were given later [16, 11]. Other definitions of DisCSPs have considered the case where the interest on constraints is distributed among agents [17, 10, 7]. [10] proposes algorithms that fit the structure of a real problem (the nurse transportation problem). The Asynchronous Aggregation Search (AAS) [7] algorithm actually extends ABT to the case where the same variable can be instantiated by several agents and an agent may not know all constraint predicates relevant to its variables. AAS offers the possibility to aggregate several branches of the search. An

aggregation technique for DisCSPs was then presented in [6] and allows for simple understanding of the privacy/efficiency mechanisms. The order on variables in distributed search was so far addressed in [2, 13, 10, 4], showing the strong impact it has on the solving algorithms.

3 Asynchronous Backtracking (ABT)

In asynchronous backtracking, the agents run concurrently and asynchronously. Each agent A_i instantiates its variable x_i and communicates the variable value to the relevant agents. Since we do not assume FIFO channels, in our version a **local counter**, $C_i^{x_i}$, in each agent is incremented each time a new instantiation is chosen, and its current value **tags** each generated assignment.

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

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

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

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

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

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

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

The following types of messages are exchanged in ABT:

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

The agents start by instantiating their variables concurently and send **ok?** messages to announce their assignment to all agents with lower priority in their *outgoing links*. The agents answer to received messages according to the Algorithm 1 [11] (except for code delimited by '*').

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

A **nogood is valid** if it contains only valid assignments. The next property is mentioned in [15] and it is also implied by the Theorem 1, presented later.

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

3.1 Asynchronous Weak-Commitment (AWC)

In AWC, each agent A_i has a priority defined by a local value $k(A_i)$ in conjunction with the initial position i. $k(A_i)$ is increased when a new nogood $\neg N$ is computed, such that A_i will precede all the agents generating assignments in N. The new priority is broadcasted to the neighboring agents. Excepting this priority, AWC behaves like the ABT which stores all nogoods. AWC is guaranteed to terminate since the number of possible nogoods is bounded. However the number of possible nogoods is exponential in the size of the problem.

4 Histories

There is one major issue to be solved for allowing agents to asynchronously propose new orders on themselves. According to [2], in general, infinite loops cannot be avoided if always[1] more than one agent involved in a conflict can change their instantiation. The agents in a conflict must be able to eventually coherently decide which one of them should change its instantiation (an order).

A simple way to dynamically reorder agents in asynchronous search (e.g. for efficiency) is to intermittently synchronize among all agents a consistent view of what the order is. We call it *reordering with restarts*. The search is complete if the delay between two reorderings increases (e.g. monotonically) to ∞ [1].

On the other hand, if we need that agents propose reordering asynchronously (e.g. as in AWC or as required for security in [9]), the order typically changes before an agreement could be reached. What one can do in this case is to ensure that only *a finite number of orderings* can be considered. One then must enable agents to coherently decide *which of two received orderings is the newest*. However, we avoid putting an artificial bound on the number of reorderings proposed by each agent. This would not match the global complexity of a problem but corresponds to a two stage algorithm: an *incomplete min-conflict*-like search for

[1] There is not a problem if the event eventually occurs but it should not always occur.

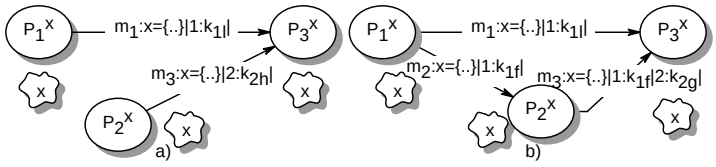


Fig. 1. Simple scenarios with messages for proposals on a shared variable, x.

a bounded time, followed if necessary by a version of ABT (e.g. upon AWC, this can be achieved by putting an upper-bound on $k(A_i)$ and running the ABT which stores only valid nogoods and taggs assignments with counters).

To exploit the analogy between maintaining agreement on a value assignment and agreement on an order, we propose to handle an ordering on agents as the value of a variable. Now we show how we decide which of two received orderings is the newest. We introduce a *marking technique* that allows for the easy definition of a total order among the proposals made asynchronously by a set of statically ordered agents on a shared variable (e.g. the ordering of other agents).

Definition 5. *A* **proposal source** *for a variable x is an entity (e.g. an abstract agent) that can make specific proposals concerning the valuation of x.*

We consider that a static order $\prec$ is defined on *proposal sources*. The *proposal source* with position k is noted P_k^x, $k \geq x_0^x$. x_0^x is the first position defined by $\prec$. Each *proposal source* P_i^x maintains **a counter** C_i^x for x. The markers involved in our *marking technique for ordered proposal sources* are called **histories**.

Definition 6. *A* **history** *is a chain h of pairs that can be associated to a proposal for a variable x. A pair $p{=}|a{:}b|$ in h signals that a proposal for x was made by P_a^x when its C_a^x had the value b, and it knew the prefix of p in h.*

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

Definition 7. *A history h_1* **is newer than** *a history h_2 if a lexicographic comparison on them, using the order $\propto$ on pairs, decides that h_1 precedes h_2.*

This is a generalization of the notion *newer* on assignments. P_a^x builds a history for a proposal on x by prefixing to the pair $|a{:}\text{value}(C_a^x)|$, the newest history that it knows for proposals on x made by some P_k^x, $k{<}a$. The C_a^x of P_a^x is reset each time incoming messages announce proposals with newer histories, made by some P_b^x, $b{<}a$. C_a^x is incremented before P_a^x makes a proposal for x.

Definition 8. *A history h_1 built by P_i^x for a proposal is* **valid** *for an agent A if no other history h_2 (eventually known only as prefix of a history h_2') is known by A such that h_2 is newer than h_1 and was generated by P_j^x, $j \leq i$.*

For example, in Figure 1 the agent P_3^x may get messages concerning the same variable x from P_1^x and P_2^x. In Figure 1a, if the agent P_3^x has already received m_1,

it will always discard m_3 since the *proposal source* index has priority. However, in the case of Figure 1b P_2^x knows $|1{:}k_{1f}|$ and the message m_1 is the newest only if $k_{1f}{<}k_{1l}$. The length of a history tagging proposals for a variable, x, is upper bounded by the number of *proposal sources* for x.

Alternatively to using histories, proposals could be tagged using a simple counter. In this case an agent needs to store the last proposals on x made by each predecessor proposal source and considers as current proposal a combination of them. Then P_a^x needs not resend its old proposal p when p remains consistent with the view of P_a^x that changes. Instead P_a^x would have to send a new proposal if its proposal changes to become identical with the newest received proposal. This is a tradeoff and the best alternative depends on the problem at hand. We conjecture that using histories is usually preferable for shared variables in AAS as well as for ordering with the reordering heuristics presented here.

5 Reordering

Now we show how the histories described in the previous section help agents during the search to asynchronously and concurrently propose new orders on themselves. In the next subsection we describe a didactic simple version that needs additional specialized agents.

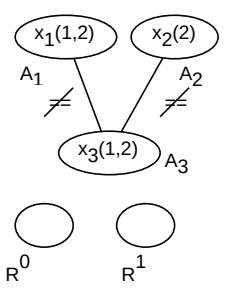
5.1 Reordering with dedicated agents

The histories in the previous section are built by proposal sources ordered statically. We only know to use histories built by statically ordered proposal sources. The idea behind asynchronous reordering in asynchronous search is to consider the proposal sources for ordering as additional abstract agents. Their activity can be delegated to different physical agents along the search. This seems complicated to apprehend at once and we start by first considering that the proposal sources for the ordering are a set of external distinct agents. Besides the agents $A_1, ..., A_n$ in the DisCSP we want to solve, we consider that there exist $n{-}1$ other agents, $R^0, ..., R^{n-2}$, solely devoted for reordering the agents A_i.

Definition 9. *An* **ordering** *is a variable that can take as value a sequence of distinct agents* $A_{k_0}, ..., A_{k_n}$.

We attach to each value of ordering a *history* as defined in the previous section. The *proposal sources* for the ordering on agents are the agents R^i, where $R^i \prec R^j$ if $i{<}j$ and $x_0^{\text{order}}{=}0$. R^i is the *proposal source* that when knowing an ordering, o, can propose orderings that reorder only agents on positions p, $p > i$. In terms of value agreement this means that the agents agree on a value o for ordering when each R^i agrees with the prefix of length $i{+}1$ in o.

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



$A_1/\mathbf{A^1}$ ——**ok?**$\langle x_1,1,1\rangle(A_1,A_2,A_3)$——→ A_3
$A_1/\mathbf{A^1}$ ——**heuristic**$\langle x_1,1,1\rangle(A_1,A_2,A_3)$——→ R^1
$A_2/\mathbf{A^2}$ ——**ok?**$\langle x_2,2,1\rangle(A_1,A_2,A_3)$——→ A_3
$A_3/\mathbf{A^3}$ —**nogood**$\neg(\langle x_1,1,1\rangle,\langle x_2,2,1\rangle)(A_1,A_2)$—→ A_2
$A_2/\mathbf{A^2}$ ——**add-link**——→ A_1
$\mathbf{R^1}$ —— **reorder** $(A_1,A_3,A_2)|1:1|$——→ A_3
$\mathbf{R^0}$ —— **reorder** $(A_3,A_1,A_2)|0:1|$——→ A_3
$\mathbf{R^1}$ —— **reorder** $(A_1,A_3,A_2)|1:1|$——→ A_2
$\mathbf{R^0}$ —— **reorder** $(A_3,A_1,A_2)|0:1|$——→ A_1
$A_3/\mathbf{A^1}$ ——**ok?**$\langle x_3,1,2\rangle(A_3,A_1,A_2)|0:1|$——→ A_1
$A_3/\mathbf{A^1}$ ——**ok?**$\langle x_3,1,2\rangle(A_3,A_1,A_2)|0:1|$——→ A_2

Fig. 2. Simplified example for ABT with random reordering based on dedicated agents. A_i/A^j in the left column shows that A_i had the position j when it has sent the message.

also referred to by A_l as either x^j, $x^j(o)$ or x_i^j. All the agents owning variables related by a constraint with the variable of the current agent are inserted in *outgoing links* at initialization. **ok?** messages are sent only to those agents in *outgoing links* that have lower priority than the current agent.

Definition 10 (Known order). *An ordering known by R^i (respectively A^i) is the order o with the newest history among those proposed by the agents $R^k, 0{\leq}k{<}i$ and received by R^i (respectively by A^i). A^i has the position i in o. This ordering is referred to as the* known order of R^i *(respectively of A^i).*

Rule 2 (Proposed order) *An ordering, o,* proposed by R^i *is such that the agents placed on the first i positions in the* known order of R^i *must have the same positions in o. o is referred to as the* proposed order of R^i.

Let us consider two different orderings, o_1 and o_2, with their corresponding histories: $O_1 = \langle o_1, h_1\rangle$, $O_2 = \langle o_2, h_2\rangle$; such that $|h_1| \leq |h_2|$. Let $p_1^k = |a_1^k{:}b_1^k|$ and $p_2^k = |a_2^k{:}b_2^k|$ be the pairs on the position $k, k > 0$ in h_1 respectively in h_2.

Definition 11 (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, $R(h_1{,}h_2)$, is either $min(a_1^u, a_2^u) + 1$ if $u > v$, or $a_2^{v+1} + 1$ otherwise. This is the position of the highest priority agent reordered between h_1 and h_2.*

New optional messages for reordering are:

- **heuristic** messages for heuristic dependent data, and
- **reorder** messages announcing a new ordering, $\langle o, h\rangle$.

An agent R^i announces its proposed order o by sending **reorder** messages to all agents $A^k(o), k > i$, and to all agents $R^k, k > i$. Each agent A_i and each agent R^i has to store its *known order* denoted $O^{crt}(A_i)$ respectively $O^{crt}(R^i)$. For allowing asynchronous reordering, each **ok?** and **nogood** message receives as additional parameter an order and its history (see Algorithm 1). The **ok?**

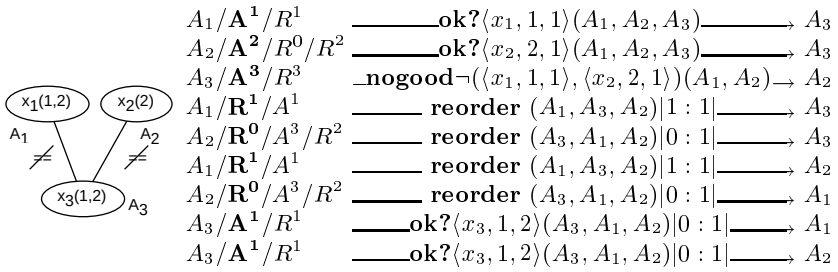


Fig. 3. Example for ABTR with random reordering. R^i delegations are done implicitely by adopting the convention "A^i is delegated to act for R^i". Left column: $A_i/A^j/R^{i_1}/R^{i_2}$... shows the roles played by A_i when the message is sent. In bold is shown the capacity in which the agent A_i sends the message. The addlink is not shown.

messages hold the newest *known order of* the sender. Each **nogood** message sent from A^j to A^i holds the order, in agreement with $O^{crt}(A^j)$, that A^j *believes* to be $O^{crt}(A^i)$. This ordering denoted $O_i^{crt}(A^j)$ consists of the first i agents in the $O^{crt}(A^j)$ and is tagged with a history obtained from the history of its O^{crt} by removing all the pairs $|a{:}b|$ where $a{\geq}i$. For example, in Figure 2 the 4th message contains $O_2^{crt}(A^3){=}(A_1, A_2)$.

When A_i receives a message which contains an order with a history h that is newer than the history h^* of $O^{crt}(A_i)$, let the *reordering position* of h and h^* be I^r. The assignments known by A_i for the variables x^k, $k \geq I^r$, are invalidated.

The agents R^i can modify the ordering in a random manner or according to special strategies appropriate for a given problem.[2] Sometimes it is possible to assume that the agents want to colaborate in order to decide an ordering. The **heuristic** messages are intended to offer data for reordering proposals. These parameters depend on the used reordering heuristic. The **heuristic** messages can be sent by any agent to the agents R^k. **heuristic** messages may only be sent by an agent to R^k within a bounded time, $\mathbf{t_h}$, after having received a new assignment for $x^j, j{\leq}k$. Agents can only send **heuristic** messages to R^0 within time t_h after the start of the search. Any **reorder** message is sent within a bounded time $\mathbf{t_r}$ after a **heuristic** message is received or from start.

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

5.2 ABT with Asynchronous Reordering (ABTR)

In fact, we have introduced the physical agents R^i in the previous subsection only in order to simplify the description of the algorithm. Any of the agents A_i or other entity can be delegated to act for any R^j. When proposing a new order, R^i can also simultaneously delegate the identity of $R^i, ..., R^{n-2}$ to other entities,

[2] e.g. first the agents forming a coalition with R^i.

when received *(***ok?**,$\langle x_j, d_j*, c_{x_j}*\rangle$ *,$\langle o, h\rangle$ *)* **do**
*__getOrder__($\langle o, h\rangle$); **if**(old c_{x_j}) return*; //ABTR;
add(x_j,d_j*,c_{x_j}*) to *agent_view*; clean *nogoods*; **check_agent_view**;
end do.
when received *(***nogood**,A_j,$\neg N$ *,$\langle o, h\rangle$ *[*,L*]**)* **do**
*__getOrder__($\langle o, h\rangle$); **if** I am not CEA($\neg N$) return*; //ABTR;
*[**if** L newer than L^o **then** $L^o \leftarrow L$]*; //ABTR-wc1;
***if** (($\langle x_i, d, c\rangle \in N$ and I have better nogood for x_i=d wanting to discard $\neg N$) or
(if $\neg N$ contains invalid assignments)) **then** discard $\neg N$ **else***; //ABTR;
when $\langle x_k, d_k*, c_k*\rangle$, where x_k is not connected, is contained in $\neg N$
send **add-link** to A_k; add $\langle x_k, d_k*, c_k*\rangle$ to *agent_view*; clean *nogoods*;
put $\neg N$ in *nogood-list* for x_i=d;
add other new assignments to *agent_view*; clean *nogoods*;
old_value $\leftarrow$ *current_value*; **check_agent_view**;
when *old_value* = *current_value* *and if A_j has lower priority than A_i*
1.1 send (**ok?**,$\langle x_i$, *current_value*$*, c_{x_i}*\rangle$*,O^{crt}*) to A_j;
end do.
procedure check_agent_view do
when agent_view *and* current_value *are not consistent*
if *no value in D_i is consistent with* agent_view **then**
backtrack;
else
select $d \in D_i$ where *agent_view* and d are consistent;
current_value $\leftarrow$ d; *c_{x_i}++ *;
send (**ok?**,$\langle x_i, d*, c_{x_i}*\rangle$*,$O^{crt}$*) to lower priority agents in *outgoing links*;
end
end do.
procedure backtrack do
nogoods $\leftarrow \{V \mid V =$ inconsistent subset of *agent_view*$\}$;
when an empty set is an element of *nogoods*
broadcast to other agents that there is no solution, terminate this algorithm;
for every $V \in$ *nogoods*;
select (x_j,d_j*,c_{x_j}*) where x_j has the lowest priority in V;
1.2 send (**nogood**,A_i,V*,O_j^{crt}[,O^{crt}]*) to A_j;
remove (x_j,d_j*,c_{x_j}*) from *agent_view*; clean *nogoods*;
check_agent_view;
end do.
function getOrder*($\langle o, h\rangle$) $\rightarrow$ *bool //ABTR*
when h is invalidated by the history O^{crt} **then** return false;
*[**when** newer h than L^o or same h as for L^o but longer o **then** $L^o \leftarrow \langle o, h\rangle$]*;
when not newer h than O^{crt} **then** return true;
$I \leftarrow$ *reorder position* for h and the history of O^{crt};
invalidate assignments for $x^j, j \geq I$; $\langle o, h\rangle \rightarrow O^{crt}$;
1.3 make sure that `send` (**ok?**,$\langle x_i$, some value, $c_{x_i}\rangle$,O^{crt}) will be performed
to all lower priority agents in *outgoing links*;
return true;
end.

Algorithm 1: *Procedures of A_i for receiving messages in ABT, ABTR and ABTR-wc1. Code between '*' is added to ABT. Code between '[' ']' is only for ABTR-wc1*

P_k, by attaching a sequence $R^0 \rightarrow P_{k_0}, ..., R^{n-2} \rightarrow P_{k_{n-2}}$ to the ordering. At a certain moment, due to message delays, there can be several entities believing that they are delegated to act for R^i based on the ordering they know. However, any other agent can coherently discriminate among messages from simultaneous R^is using the histories that R^is generate. The R^i themselves coherently agree when the corresponding orders are received. The delegation of $R^j, j>i$ from a physical entity to another poses no problem of information transfer since the counter C_j^{order} of R^j is reset on this event. The counter C_i^{order} of a new R^i delegated by a previous different R^i is set to k where $|i{:}k|$ is a pair in the history that tags the new received ordering.

For example, in Figure 3 we describe the case where the activity of R^i is always performed by the agent believing itself to be A^i. R^i can send a **reorder** message within time t_r after an assignment is made by A^i since a **heuristic** message is implicitely transmitted from A^i to R^i. We also consider that A_2 is delegated to act as R^0. R^0 and R^1 propose one random ordering each, asynchronously. The receivers discriminate based on histories that the order from R^0 is the newest. The known assignments and nogood are discarded. In the end, the *known order* for A_3 is $(A_3, A_1, A_2)|0:1|$.

By **quiescence** of a group of agents we mean that none of them will receive or generate any valid nogoods, new valid assignments, **reorder** messages or **add-link** messages. The proof of the next property of ABTR is given in Annexes.

Property 2 *$\forall i$, in finite time t^i either a solution or failure is detected, or all the agents $A^j, 0<j\leq i$ reach quiescence in a state where they are not refused an assignment satisfying the constraints that they enforce and their* agent_view.

Theorem 1. *ABTR is correct, complete and terminates.*

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

No infinite loop: This is a consequence of the Property 2 for $i = n$.

Correctness: All assignments are sent to all interested agents and stored there. At quiescence all the agents know the valid interesting assignments of all predecessors. If quiescence is reached without detecting an empty nogood, then according to the Property 2, all the agents agree with their predecessors and the set of their assignments is a solution. □

As we show in [9], ABTR is required in secure algorithms for automated negotiations. In the sequel of this paper we deal with efficiency issues.

5.3 Saving effort across reordering

We let agents maintain their current assignment when a new order is received. If the old order known by A_i was $\langle o', h' \rangle$, after receiving a new order, $\langle o, h \rangle$, $A_i^j(o)$ removes from its *agent_view* only those assignments $\langle x^k(o), v, c \rangle$ where $k > j$. Therefore, $A_i^j(o)$ discards only nogoods containing some assignment $\langle x^k(o), v, c \rangle, k > j$, since the validity of such nogoods cannot be checked in o.

function getOrder$(\langle o, h\rangle) \rightarrow bool$
  when h is invalidated by the history of O^{crt} **then** return false;
  when not newer h than O^{crt} **then** return true;
  $O^{crt} \leftarrow \langle o, h\rangle$; invalidate assignments for $x^j(o), j > v$ (remove from *agent_view*);
2.1 make sure that `send` (**ok?**,$\langle x_i$, some value, $c_{x_i}\rangle$,O^{crt}) will be performed
    to all lower priority agents in *outgoing links*;
  when I am R^w, $C_w^{\text{order}} \leftarrow (\exists |w{:}k| \in h)?k : 0$;
  return true;
end.

Algorithm 2: *Procedure of A_i^v for receiving new orderings in ABTR1.*

when received (**nogood**,A_j^v,$\neg N$,$\langle o, h\rangle$) **do**
  validOrder$\leftarrow$ **getOrder**($\langle o, h\rangle$); **if** ($\neg$*validOrder* $\wedge$ ($A_i \neq$CEA($\neg N$))) return;
  if (($\langle x_i, d, c\rangle \in N$ and I have not-worse nogood for $x_i{=}d$ wanting to discard $\neg N$)
      or (if $\neg N$ contains invalid assignments)) **then** discard $\neg N$ **else**;
    when $\langle x_k, d_k, c_k\rangle$, where x_k is not connected, is contained in $\neg N$
      send **add-link** to A_k; add $\langle x_k, d_k, c_k\rangle$ to *agent_view*;
    put $\neg N$ in *nogood-list* for $x_i{=}d$;
    add other new assignments in N to *agent_view*;
  when (validOrder *and valid* $\neg N$ *and* d=current_value)
3.1 C_{v-1}^{order}++; *new_order* $\leftarrow A^1, ..., A^{v-1}, A_j, ...$;
    getOrder($\langle$*new_order*,new-history$\rangle$);
3.2 make sure to send O^{crt} or (**reorder**,O^{crt}) to all $A^j, j \geq v$;
  check_agent_view;
end do.

Algorithm 3: *Procedure of A_i^v for receiving* **nogood** *messages in ABTR-wc.*

A_i (re)sends its current assignment via **ok?** messages to any agent $A^u(o)$ in its *outgoing list* where $u > j$. This version of ABTR is referred to as ABTR1.

Theorem 2. *ABTR with assignments reuse across reordering (ABTR1) has polynomial space, is correct, complete and terminates.*

Proof. Since the assignments are resent, the only difference with ABTR is that some additional nogoods are stored. These additional nogoods are consistent with the Theorem 1. The space remains polynomial since only one assignment is valid and only one nogood continues to be stored for each value of each variable. The space complexity is not modified. □

6 AWC-like heuristic for ABTR1 (ABTR-wc)

In the previous section we have described a reordering technique that allows for using **heuristic** messages for guiding reordering but we did not propose any specific heuristic. ABT performs Forward Checking since the domains of the future variables are pruned after each assignment. However, using **heuristic**

messages can be expensive since 3/2 message roundtrips $A^i \rightarrow A^j \rightarrow R^i \rightarrow A^{i+1}$ are required for getting the result of prunning.

In this section we show how the idea behind the heuristic used in AWC can be reused with ABTR1. In AWC each agent A_i that discovers a new nogood $\neg N$ increases its own priority. Let us assume that the lowest priority assignment in N concerns x^v. In ABTR the agent A_i has to send $\neg N$ to A^v. We can spare messages by letting A^v decide the new order that gives more priority to A_i.

The priority of a variable is the priority of the agent owning it. $\neg N_1$ is a **better nogood** then $\neg N_2$ if N_1 contains only higher priority variables than the lowest priority variable in N_2. To make sure that it increases the priority of A_i in the general case, A^v can offer to A_i its position v or a lower one. To do this, A^v has to act for some $R^{v-t}, t > 0$. Let each A^k act for R^{k-1} and let it cede its position to the sender of each valid nogood received (Algorithm 3). This protocol is called ABTR-wc.

Theorem 3. *Better or valid nogoods can be received by agent A^i in ABTR-wc only in finite time after the agents $A^j, j < i$ have reached quiescence.*

Proof. Each valid nogood received by A^i eliminates a value. Each value eliminated by such a nogood will never be available since the nogoods eliminating them (or better nogoods received later) use fixed assignments of quiescent agents and cannot be invalidated. There are maximum dn such values and better nogoods can be received for a value only n times. Therefore, they are received in a finite time. □

It is required that R^k does not send reorders beyond delay $t_r + t_h + 2\tau$ after an instantiation is done by some $A^j, j \leq k$.

Corolary 1 *$\exists t_h$ such that, when proposing reordering, the agent A^k acts legally for R^{k-1} and ABTR-wc is an instance of ABTR1.*

There exists a finite t_h such that any valid or better nogood is received by R^{k-1}/A^k in a time bounded by t_h after the quiescence of the agents $A^l, l < k$.

6.1 Saving effort across reordering for ABTR-wc

In the Algorithm 3, the line 3.1 actually allows for several versions since agent A_i is free to put whatever order among the agents following A_j. By ABTR-wc we denote a version where the agents following A_j are ordered lexicographically. This convention requires no additional information. However, due to reordering involved on future agents many of the nogoods own by successors are invalidated.

In a version of ABTR-wc, that we denote ABTR-wc1, each agent maintains one more ordering called *last_known_successor_order* and denoted L^o. The line 1.2 in Algorithm 1 is modified in ABTR-wc1 such that the sent nogood is also accompanied with the ordering O^{crt}. L^o holds the most recent ordering among O^{crt} and the orderings received with **nogood** messages. In ABTR-wc1 the *new_order* at line 3.1 becomes $A^1, ..., A^{v-1}, A_j, A_i, ...$, where the agents following A_i are ordered according to L^o.

A further modification we make to ABTR-wc1 is that each new proposed ordering is sent at line 3.2 to all agents so that all of them can update their *last_known_successor_order*. This last version is denoted ABTR-wc2.

7 Fair reordering heuristics

When the distributed search is used for negotiations, the position of an agent A in the ordering on agents corresponds to certain advantages and drawbacks for A. With some given problems agents prefer to be the first positioned ones such that they could propose their preferred alternatives first. For other problems, agents want to be positioned later such that they need to reveal less about their constraints/domains. In ABTR one can reorder randomly or 'round robin' the agents in order to enhance fairness. Then, one still has to check or trust that the agents respect the chosen reordering strategy.

However, when the agents form equal sized coalitions, fairness can be ensured in ABTR by majority voting. Let us assume that the agents want to solve a problem where privacy is essential. Agents want to place other agents from their coalition on last positions. A fair alternation of agents from different coalitions, when we have equal sized coalitions, is obtained when the function of $R^k, k>0$ is undertaken by *an entity* defined by the set of agents $A^1, ..., A^k$. The agents in R^k choose the agent A^{k+1} by majority voting. Fairness results from the fact that for these problems, low position agents will propose on next lower positions agents from other coalitions.

The majority voting can be implemented by letting any A^i in R^k broadcast to any other agent in R^k the set of agents that A^i would like to see on position A^{k+1}. The set is tagged with the history of $O^{crt}(R^k)$. E.g., the agent chosen for the position A^{k+1} can be the currently lowest position agent among those with the highest number of occurrences in the proposals. The time t_r required by R^k for deciding a new order is then equal to the maximum message delivery delay τ. The correctness, completeness and termination properties of ABTR are therefore maintained.

Similarly, when the quality of the found solution is essential and privacy is less important, a fair reordering is obtained when the function of R^k is undertaken by the set of agents $A^{k+1}, ..., A^n$. With equal sized coalitions, a fair alternation is obtained when the agent A^{k+1} is decided with the majority of the votes of the agents forming the entity R^k.

8 Experimentation

Solution detection In ABT a solution is found when the agents reach quiescence. As proposed in [14] quiescence can be detected using general purpose algorithms. We rather use in our experiments a solution detection algorithm that can detect solutions before quiescence by composing partial valuations of the agents. This technique is first proposed in [7] where composed partial valuations are sent via **accepted** messages from lower priority agents to linked higher priority agents.

They are sent only when the partial valuation obtained by composing all incoming partial valuations from all *outgoing-links* with the local one is not empty. [8] uses a version of this technique where a *static spanning tree* having as root a system agent is defined over agents. Each agent sends partial valuations only along this spanning tree towards the root.

The *composition of two partial valuations*, v_1, v_2, consists in a valuation, v, $v{=}v_1{\cup}v_2$. If x_i is assigned in both v_1 and v_2 then v is non-empty only if x_i is assigned with the same value in v_1 and in v_2.

The solution detection technique in [8] can be used with no modification for versions of ABTR where each agent always enforces all the constraints it knows (e.g. ABTR reordering technique for AAS). Alternatively, for use with the general version of ABTR, the solution detection technique presented in [8] is modified by attaching as additional parameter of **accepted**(v) messages the *Set of References to the Constraints Completely Satisfied* by the partial valuation v, SRCCS(v). Each agent composing incoming partial valuations makes and attaches the union of the corresponding SRCCS.

The references in SRCCS refer only to initial constraints and not to constraints (nogoods) inferred during search. This algorithm assumes that agents know or may get references to (all) the constraints and that agents knowing the same constraint agree on a common reference for it. This is acceptable when the constraints are public. Otherwise, the technique in [8] is available.

Proposition 1. *When the partial valuation V computed by the system agent by composing the last incoming valuations v_k from each branch k of the root of the solution detection spanning tree is not empty and SRCCS(V)=$\cup_k$(SRCCS(v_k)) contains references for all the constraints of the agents, then V is a solution.*

Proof. By construction, any extension of a partial valuation v satisfies all the constraints referred in the associated SRCCS(v). Therefore V satisfies SRCCS(V), satisfying all agents when SRCCS(V) contains references to all the constraints. □

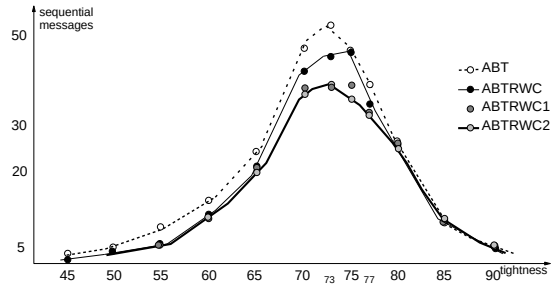
Proposition 2. *The solution detection algorithm based on SRCCSs detects a solution for ABTR in finite time if the search does not fail.*

Proof. Let us assume that a solution was not detected before quiescence. Then according to Theorem 1 quiescence is reached by ABTR in finite time. Since at quiescence any initial constraint C is enforced by some agent, its reference is sent with the last valuation in the SRCCS along the spanning tree. Since at quiescence for ABTR the instantiations define a solution and are coherent, the constraint references propagate up to the root and all references are present in the last SRCCS(V) computed at root. □

8.1 Tests

We have run tests on random problems with 20 agents. The agents were placed on distinct computers of our lab. We have generated problems of variable tightness

Fig. 4. Experiments



for a density of 27% where each variable has 3 values. Each point in Figure 4 is averaged over 100 random problems and shows the average number of sequential messages (half network round-trips) required to solve the problem. The number of round-trips is the only important cost when agents are placed remotely on internet and the local problems are not hard. The experiments show that ABTR-wc2 performed clearly better in average than ABT and performed better than other versions of ABTR-wc. This result shows that additional messages for heuristic data (function taken by the added **reorder** messages in ABTR-wc2) have actually improved efficiency. Therefore other efficient heuristics may be discovered for ABTR in the future. For under-constrained problems (tightness over 85%) where solutions are found without resorting to many **nogood** messages, few reorderings are proposed by ABTR-wc and therefore the new algorithms perform quite similarly with ABT.

9 Conclusions

Reordering is a major issue in constraint satisfaction and is the main strength of Asynchronous Weak-Commitment (AWC). All previous complete polynomial space asynchronous search algorithms for DisCSPs require a static order of the variables. We have presented an algorithm that allows for asynchronous reordering in Asynchronous Backtracking (ABTR). This is the first asynchronous complete algorithm with polynomial space requirements that has the ability to concurrently and asynchronously reorder variables during systematic asynchronous search. Asynchronous reordering is required for security reasons in managing coalitions in automated negotiation (see [9]). We then present ABTR-wc, an efficient reordering heuristic for ABTR inspired from AWC, that requires no heuristic message. As with most distributed asynchronous algorithms, certain resemblance [5, 7] can be found between the behavior of ABTR-wc and Dynamic Backtracking [3]. Here the resemblance extends also to the reordering heuristic.

Annexes (Proof)

Property 2 *$\forall i$, in finite time t^i either a solution or failure is detected, or all the agents $A^j, 0<j\leq i$ reach quiescence in a state where they are not refused an assignment satisfying the constraints that they enforce and their* agent_view.

Proof. Let all agents $A^k, k<i$, reach quiescence before time t^{i-1}. Let τ be the maximum time needed to deliver a message.
$\exists t_p^i < t^{i-1}$ after which no **ok?** is sent from A^k, $k<i$. Therefore, no heuristic message towards any R^u, $u<i$, is sent after $t_h^i = t_p^i + \tau + t_h$. Then, each R^u becomes fixed, receives no message, and announces its last order before a time $t_r^i = t_h^i + \tau + t_r$. After $t_r^i + \tau$ the identity of A^i is fixed as A_l. A_l^i receives the last new assignment or order at time $t_o^i < t_r^i + \tau$.
Since the domains are finite, after t_o^i, A_l^i can propose only a finite number of different assignments satisfying its view. Once any assignment is sent at time $t_a^i > t_o^i$, it will be abandoned when the first valid nogood is received (if one is received in finite time). All the nogoods received after $t_a^i + n\tau$ are valid since all the agents learn the last instantiations of the agents $A^k, k < i$ before $t_a^i + n\tau$. Therefore the number of possible incoming invalid nogoods for an assignment of A^i is finite.

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

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

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

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

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

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

In the case (i), since $\neg N$ was generated by A^i, A^i is interested in all its variables (has sent once an **add-link** to A^j), and it will be announced by A^j of the modification by an **ok?** messages. This contradicts the assumption that the last **ok?** message was received by A^i at time t_o^i and the induction step is proved.
From here, the induction step is proven since it was proven for all alternatives.
In conclusion, after t_o^i, within finite time, the agent A^i either finds a solution and quiescence or an empty nogood signals failure.
After $t_h + t_r + \tau$, R^0 is fixed and the property is true for the empty set. The property is therefore proven by induction on i □

References

1. L. Baptista and J. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *CP'2000*, pages 489–494, 2000.
2. Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of IJCAI 1991*, pages 318–324, 91.
3. M. L. Ginsberg. Dynamic backtracking. *Journal of AI Research*, 1, 93.
4. Y. Hamadi and C. Bessière. Backtracking in distributed constraint networks. In *ECAI'98*, pages 219–223, 98.

5. W. Havens. Nogood caching for multiagent backtrack search. In *Proc. AAAI'97 Constraints and Agents Workshop*, '97.
6. P. Meseguer and M. A. Jiménez. Distributed forward checking. In *Proceedings of the International Workshop on Distributed Constraint Satisfaction*. CP'00, 2000.
7. M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proc. of AAAI2000*, pages 917–922, 2000.
8. M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Maintaining hierarchical distributed consistency. In *Proceedings of the International Workshop on Distributed Constraint Satisfaction*. CP'00, 2000.
9. M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Generalized English Auctions by relaxation in dynamic distributed CSPs with private constraints. In *Proc. of the IJCAI-01 DCR Workshop*, pages 45–54, Seattle, August 2001.
10. G. Solotorevsky, E. Gudes, and A. Meisels. Distributed Constraint Satisfaction Problems - a model and application. Preprint: http://www.cs.bgu.ac.il/~am, 97.
11. M. Yokoo. *Distributed Constraint Satisfaction*. Springer, 01.
12. M. Yokoo. Constraint relaxation in distributed constraint satisfaction problem. In *ICDCS'93*, pages 56–63, June 93.
13. M. Yokoo. Asynchronous weak-commitment search for solving large-scale distributed constraint satisfaction problems. In *Proc. ICMAS*, pages 467–318, 95.
14. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *ICDCS'92*, pages 614–621, June 92.
15. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The Distributed CSP: Formalization and algorithms. *IEEE Trans. on KDE*, 10(5):673–685, 98.
16. M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of 3rd ICMAS'98*, pages 372–379, 1998.
17. Y. Zhang and A. K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Proc. of Third IEEE Symposium on Parallel and Distributed Processing*, pages 394–397, 91.