Generalized Dynamic Ordering for Asynchronous Backtracking on DisCSPs

Marius C. Silaghi
Florida Institute of Technology at Melbourne
msilaghi@fit.edu

ABSTRACT

Dynamic reordering of variables is known to be very important for solving constraint satisfaction problems (CSPs). Many attempts were made to apply this principle for improving distributed constraint solvers [1, 17, 20, 10, 11, 31, 19]. Armstrong et.al in [1] report for good heuristics an improvement of approximately 30% over random ordering. Asynchronous Backtracking with Reordering (ABTR aka ABT_DO) [20, 18, 17, 31] is the previous algorithm enabling the largest number of reordering heuristics in asynchronous search while maintaining completeness, where only the first agent had to be fix. Experiments on a simulator [31] show an order of magnitude improvements in constraint checks with heuristics intended to mimic Dynamic Backtracking (DB) [5], but actually resembling AWC [28][1]. Here

a) We fix the previous chain of independent erroneous claims confusing the heuristics of DB and AWC.[1]

b) We prove that a single modification to ABTR/ABT_DO also enables the exact reordering of DB and better approximations of AWC, where the first agent is reordered, being the only self-stabilizing asynchronous protocol with this property.

c) Experimental evaluation with a really distributed implementation shows that while AWC's reordering heuristic works better than other common heuristics based on domain size, the improvement it brings in terms of rounds (length of the longest causal chain of messages) is only of the order of 30% compared to no reordering.

Our results confirm the efficiency prediction in [1] and differ largely from the more recent prediction by simulations in [31] (whose heuristic is only slightly different from our heuristics). This raises questions on whether the difference in reported improvements comes from small details in heuristics, from chosen metrics, from assumptions in simulator, or from unreported fine tuned optimizations in the different implementations. The really distributed implementation reported here is available on-line [13].

[0]The presentation was much improved by integrating recent comments from Boi Faltings and by observing terminology introduced in [31] to describe ABTR's rediscovery under the name ABT_DO. The original description of this contribution is available in the CP2001 submission recorded in [19]. Main changes to [19] done here are: the *"reordering leader i"* is now renamed as *"owner of reordering counter i"*, *"history"* tag is now renamed *"signature vector clock"* tag.

[1]In [19] we erroneously claimed that DB and AWC's heuristics are similar, and a similar claim also appears in the rediscovery of ABTR in [31].

1. INTRODUCTION

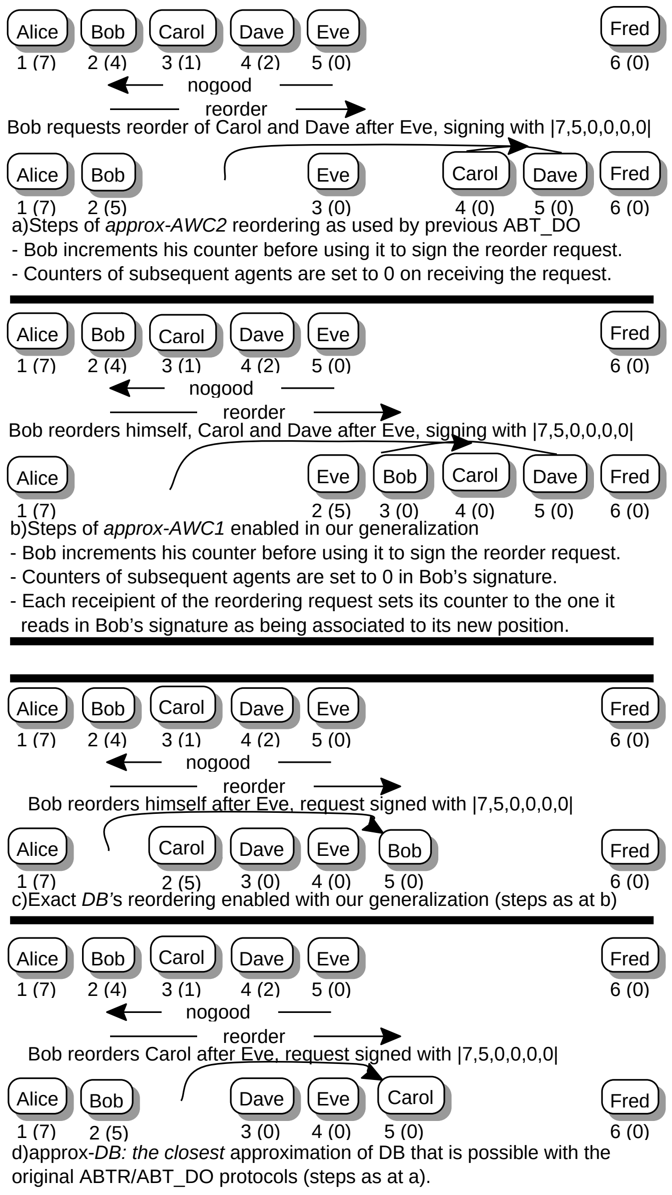
Distributed Constraint Satisfaction (DisCSP) is a powerful framework for modeling distributed combinatorial problems. A **DisCSP** is defined in [27] as: a set of agents $\mathcal{A} = \{A_1, ..., A_n\}$ where each agent A_i controls exactly one distinct variable x_i, and each agent knows all constraint 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. The case when an agent does not know all constraints relevant to its variables has impact on some details, as mentioned in the following on a case by case basis. Asynchronous Backtracking (ABT) [27, 28, 7] is the first *complete* and *asynchronous* search algorithm for DisCSPs. ABT can be run with polynomial space complexity. It uses a total priority order on variables. In ABT, agents propose assignments for their variables using **ok?** messages. No reply is provided for an accepted assignment. When rejecting an assignment, an explanation is provided using a **nogood** message that also lists the conflicting assignments having higher priority.

1.1 ABT and the Reordering Problem

The completeness of ABT is ensured with help of *a static order* imposed on agents. ABT is a slow algorithm and [24] introduces a faster algorithm called Asynchronous Weak Commitment (AWC). AWC allows for reordering agents asynchronously by decreasing their position when they are over-constrained. AWC has proven to be more efficient than ABT but its requirement for an exponential space could not be eliminated without losing completeness.

Before the work in [18, 20, 17], no fully asynchronous algorithm has offered the possibility to perform reordering without losing either the completeness, or the polynomial space property. ABT with Asynchronous Reordering (ABTR) [17], aka ABT with dynamic ordering (ABT_DO) [31], is an extension of ABT that allows the agents to asynchronously and concurrently propose changes to their order. Reordering in ABT is required in various applications (e.g. security [15]). Another typical use of reordering is for efficiency, but it is not obvious to find *time*-efficient reordering heuristics for DisCSPs. The reordering heuristic of AWC has proven to be cheap and efficient for distributed algorithms but it cannot be directly used in ABTR. The reordering of AWC is different from the reordering heuristic used in Dynamic Backtracking (DB) [5]. Heuristics in [19, 31] were intended to mimic DB but we find that both are more similar to AWC.

1.2 Sketch of the Contribution



Bob requests reorder of Carol and Dave after Eve, signing with |7,5,0,0,0,0|

a)Steps of *approx-AWC2* reordering as used by previous ABT_DO
- Bob increments his counter before using it to sign the reorder request.
- Counters of subsequent agents are set to 0 on receiving the request.

Bob reorders himself, Carol and Dave after Eve, signing with |7,5,0,0,0,0|

b)Steps of *approx-AWC1* enabled in our generalization
- Bob increments his counter before using it to sign the reorder request.
- Counters of subsequent agents are set to 0 in Bob's signature.
- Each receipient of the reordering request sets its counter to the one it reads in Bob's signature as being associated to its new position.

Bob reorders himself after Eve, request signed with |7,5,0,0,0,0|

c)Exact *DB's* reordering enabled with our generalization (steps as at b)

Bob reorders Carol after Eve, request signed with |7,5,0,0,0,0|

d)approx-*DB: the closest* approximation of DB that is possible with the original ABTR/ABT_DO protocols (steps as at a).

Figure 1: Comparing the behavior on receiving a nogood for a) approx-AWC2 reordering heuristics possible in previous ABTR work of [31, 20, 17], versus b) the approximation of AWC enabled here, c) the exact DB heuristic, enabled by this work, and d) a previously possible approximation of exact DB. Below each agent is written its current position and, in parentheses, the value of its current reordering counter. E.g., initially Bob has position 2 and counter value (4).

In this article we show how one can obtain a reordering heuristic for variables with behavior that is really identical as the one in Dynamic Backtracking. Namely, we prove that with a single modification to ABTR/ABT_DO one can also enable the exact reordering heuristic of DB while maintaining correctness and termination. We provide a proof for the generalization of ABTR, showing that ABTR's extension remains correct and terminates even with additional interesting dynamic variable reordering heuristics, in particular the exact heuristic of Dynamic Backtracking, and better approximations of AWC. The exact reordering heuristic of DB and the way it contrasts previously used approximations (that we denote *approx-AWC1* [19] and *approx-AWC2* [31]), respectively a new alternative denoted *approx-DB* is depicted in Figure 1. The scenario in Figure 1 compares the reorderings requested by an agent *Bob* upon receiving a **nogood** message from agent *Eve* when the previous ordering was: Alice, Bob, Carol, Dave, Eve, Fred. With the exact reordering heuristic of Dynamic Backtracking (Figure 1.c) it is Bob, the owner of the culprit variable, that is moved on the position after the owner of the variable discovering the nogood, here Eve. With approx-DB, the closest approximation of DB possible in the original ABTR/ABT_DO [17, 31] (Figure 1.d), it is the agent/variable after Bob, namely Carol, that is moved beyond Eve. Claiming in the original version of this paper [19] to be implementing both DB and AWC we erroneously used approx-AWC1 (Figure 1.b) which is close only to AWC. Similarly mentioning an intention to implement DB in ABTR/ABT_DO, [31] obtains approx-AWC2 (Figure 1.a) which is also closer to AWC and approx-AWC1.

ABTR's ordering coherence was insured by having a reordering counter in each agent and tagging messages with the vector-clock [9] induced by these counters. We show that ABTR remains correct with new reordering schemes if, at reordering, the reordering counter of Carol (that will replace Bob in DB) is set to the value of the counter of Bob in Bob's signature of the request for reordering (Figure 1.c). This new requirement is seamlessly unified with the previous ABTR/ABT_DO description in a way enabling additional heuristics. Namely, the provided proof identifies which optional exchange of reordering counters (i.e., dynamic mapping of reordering counters to agents) leave ABTR correct. One can optionally associate any dynamic variable reordering heuristic with a policy of dynamic changes to this mapping, as long as this is within the possibilities proven correct with the rules given here.

The performance of approx-AWC1 is evaluated on a really distributed implementation available from our web-site [13], and the obtained behavior is contrasted with the newer reported experimentation of approx-AWC2 on simulators in [31].

1.3 ABTR versus its Extension

The differences between ABTR/ABTR_DO and the extension proposed here are as follows (more details and proof are in the body of the paper). First note that in the first implemented/described versions of ABTR [20, 18, 17] each agent on a position k, has a reordering counter C^r_k. Each reordering request is tagged with a *signatures vector-clock* [9, 17, 11] listing all values of these counters known to the sender: $|c_1, ..., c_n|$ (c_k being the value of C^r_k known to the sender). An agent holding counter C^r_k can request a reordering of the agents on positions $k+1$ through n, and increments C^r_k whenever he proposes such a reordering. An example of a reordering for this case is shown in Figure 2, where A_1 (e.g. predicting that his proposal x_1=2 reduces the domain of x_3, with a min-domain heuristic) reorders A_3 before A_2. Counter values for $C^r_j, j>k$, are set to 0 in A_k's signature, since A_k does not know newer values. The first agent, A_1, cannot be reordered in the original ABTR/ABT_DO[2].

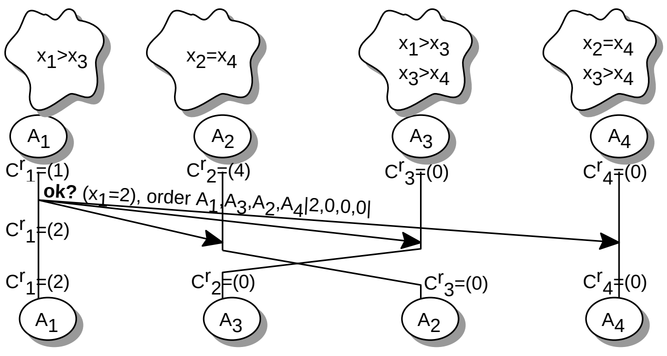


Figure 2: Each new assignment proposal by the agent A_i holding a counter C^r_k can be associated with proposing a new order on agents with position higher than k.

1.3.1 Some notations

An ordering on agents is defined by a permutation of the set of agents. The agent on position k in an ordering o is denoted $A^k(o)$. The notation $\mathbf{A^k_i(o)}$ is synonym for $A^k(o)$ and additionally tells that the agent on position k in ordering o is A_i. A similar notation is introduced for variables. Namely $\mathbf{x^k_i(o)}$ says that the variable of the agent on position k in ordering o is x_i and $x^k(o)$ is the variable on position k. The agent owning C^r_k in order o is denoted $\mathbf{R^k(o)}$. Sometimes the ordering o is missing from the previous notations in the context of a procedure of some agent A_j, and then the use of the current ordering known by A_j is implied.

1.3.2 Private constraints or arc consistency

For the case where an agent does not know all constraints on his variable (e.g., assume that in Figure 2, A_1 does not know the constraint between x_1 and x_3), that agent may not be able to predict how his assignments modify the domains of future variables, and min-domain heuristics cannot be implemented with the above protocol. Even in the basic ABT case where agents know all constraints on their variable, if arc consistency is maintained then agents cannot foresee the effect of their assignments on domains of lower priority agents.

To enable min-domain heuristics in such cases ABTR allows lower priority agents to inform others about features of interest to the reordering (such as new domain sizes) using **heuristic** messages. Based on this information, agents can renew the reordering proposed with a previous assignment (see Figure 3). This is shown later to lead to correct and terminating protocols for the heuristics where it can be proven separately that they comply with the following requirement:

[2]A trick around this limitation proposed in [18] is based on modeling, namely artificially introducing a new variable, e.g. x_0, with a single value and no constraint. Being the first, x_0 will never be reordered and generates no message but its counter C^r_0 allows to reorder A_1/x_1. x_0 and its counter can be hold by A_1. However, this trick by itself does not enable DB's reordering or approx-AWC1.

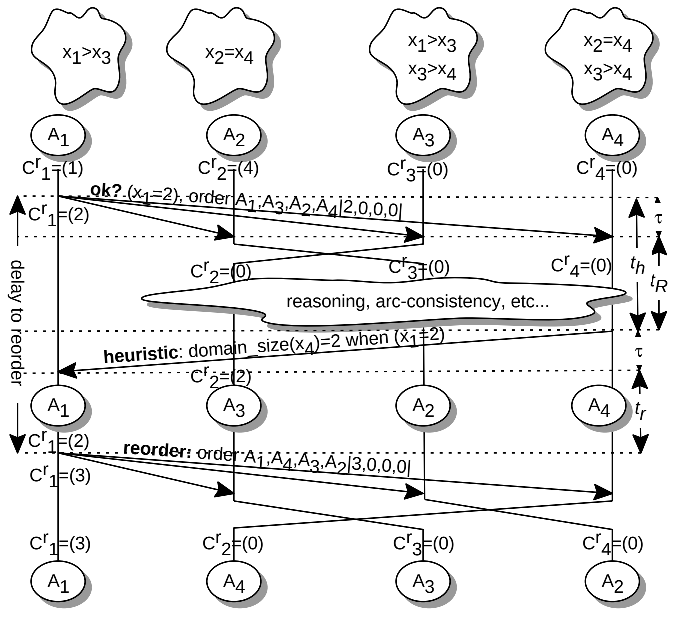


Figure 3: When the agent proposing reordering does not have the whole knowledge for efficient decisions, he can be informed with heuristic messages, if it can be proven that the delay t_h needed to generate such messages is finite (shown t_h is since A^1 sends the ok? message to when A^4 replies to the owner of C^r_1, which here is A^1).

Rule 1 (delay to reorder). *The delay between the moment the last* **ok?** *message was sent by agents $A^k, k\leq i$, and any subsequent reordering request sent by the owner of C^r_i, (***delay to reorder** *in Figure 3)* **must be** *finite.*

We sometimes explain[3] this rule with the following equivalent formulation:

Rule 2 (finite number of orderings per **ok?**). *For a given* **ok?** *message sent by A^i, the owner of C^r_i can only propose a finite number of distinct ordering requests.*

For example, with min-ordering and when agents know all the constraints on their variable, an agent issues a single ordering with each **ok?** message that it sends. If it receives **heuristic** messages later (due to private constraints/arc-consistency) then it may issue additional ordering requests, but only a finite number of them for each given **ok?** message it has previously sent.

With reordering heuristics such as min-domain in basic ABT, where no **heuristic** message is needed and A^i owns C^r_i, the fact that the *delay to reorder* is finite is ensured by the fact that a reordering is sent only simultaneously with an **ok?** message. Therefore the delay to reorder is 0 in this case.

For certain delicate reordering heuristics, such as the ones of DB and approx-AWC1, we will prefer to prove directly that the delay to reorder is finite, since it follows directly from special properties intrinsic to that scheme. For other schemes, such as min-domain with maintenance of

[3]E.g., to Roie Zivan during IJCAI-05.

arc consistency, we prefer to prove it by using the alternative/equivalent rule introduced next.

As Figure 3 illustrates, assuming that the maximum travel time, τ, of a message is finite and that the time t_r to reply to a **heuristic** message is finite, the proven requirement for correctness and termination of a given heuristic is then shown to reduce to:

RULE 3 (DELAY FOR HEURISTIC t_h). *For any agent A_j, the time t_h since agents $A^k, k \leq i$, sent the last* **ok**? *message, to the moment when a* **heuristic** *message is sent by A_j to the owner of C_i^r,* **must be** *finite (see Figure 3).*

In the original versions of ABTR and in ABT_DO, the owner of C_i^r is always the same as A^i. A less general version of the last rule but that is very useful as a guideline to directly apply for constructing certain heuristics, such as min-domain ordering with consistency maintenance is:

RULE 4 (DELAY TO REPLY t_R). *For any agent A_j, the time t_R between receiving the last* **ok**? *message from agents $A^k, k \leq i$, to the moment when a* **heuristic** *message is sent by A_j to the owner of C_i^r,* **must be** *finite (see Figure 3).*

1.3.3 Enabling DB's and approx-AWC1 heuristic

The rule of ABTR (that will change here) is:

RULE 5 (ABTR / ABT_DO). *Upon receiving a reordering request o with signatures vector-clock $|c_1, ..., c_n|$ stronger[4] than any other request received before, an agent A^j requested by o to move to another position, k,* **resets** *its new reordering counter C_k^r to* **zero**.

Instead of it, the corresponding rule of the extension of ABTR that is introduced here is:

RULE 6 (EXTENDED ABTR). *Upon receiving a reordering request o with signatures vector-clock $|c_1, ..., c_n|$ stronger than any other request received before, an agent A^j requested by o to move to another position, k,* **initializes** *its new reordering counter C_k^r* **with $\mathbf{c_k}$**.

Note that nothing changes for previously supported heuristics of ABTR, since in them the corresponding values of c_k is zero (because the sender does not have any opportunity to learn about changes to counters in lower priority agents). This new rule, if coupled with a decision to have each agent A^k hold the counter C_{k-1}^r rather than C_k^r, is sufficient to support Dynamic Backtracking's reordering heuristic. Namely, holding the counter C_{k-1}^r gives A^k the right to reorder itself (according to the above description of ABTR), while the Rule 6 ensures continuity in the value of C_{k-1}^r when the agent on position k and holding C_{k-1}^r is moved because of its own decision (see Figure 4).

1.3.4 Further generalization

We further provide a rule for a scenario not defined in ABTR, namely where the reordering counters C_k^r have a dynamic mapping to agents, including the possibility that an agent is the owner of several of these counters, and that a reordering request can explicitly ask to move lower priority reordering counters to other agents. The corresponding rule in this case is:

[4]I.e. *following* in lexicographic order. The concept is formally introduced later.

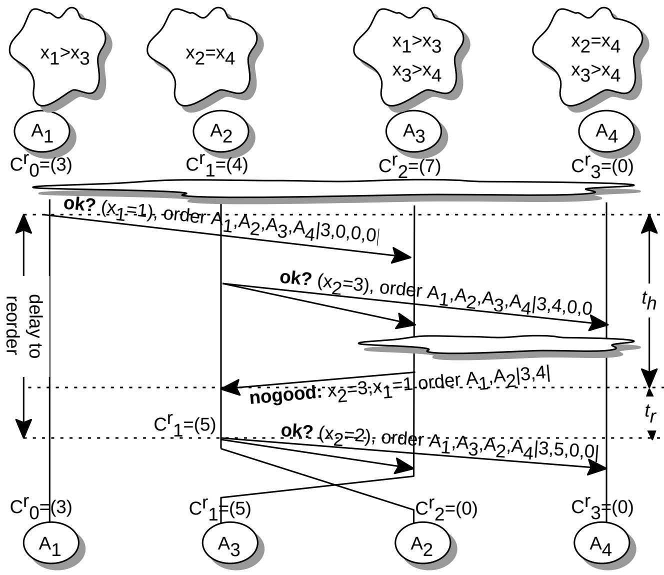


Figure 4: To obtain DB's heuristic, the reordering counter of an agent on a position k will count the reordering events of the sets of agents monitored in previous versions of ABTR by C_{k-1}^r, and initializes it from signatures of received messages. A counter C_0^r is introduced to tag reorderings of the first agent. On receiving a nogood, A_2 implicitly and instantaneously delivers to itself a heuristic message. Similar to DB's termination proof we show that the delay to reorder and t_h are finite, implying as for previous ABTR heuristics that this algorithm is correct and terminates.

RULE 7 (GENERALIZED ABTR). *Upon receiving a reordering request o with signatures vector-clock $|c_1, ..., c_n|$ stronger than any other request received before, an agent A^j requested by o* **to start maintaining a reordering counter** *C_k^r will initialize his new reordering counter C_k^r with c_k.*

We prove directly only this generalization as this applies to all aforementioned specializations. Note that this scenario allows all previous heuristics and new ones. No overhead in message sizes occurs if the new mapping of reordering counters to agents associated with a new ordering o does not need to be explicitly transmitted but can be inferred from o using some predefined convention (as happen in the original ABTR scenario where we know that C_k^r is mapped to the agent on position k).

ABTR [17, 31] is the first asynchronous search algorithm that allows for asynchronous dynamic reordering while being complete and having a polynomial space complexity. Here we use its version built on ABT since ABT is an algorithm easier to describe than its subsequent extensions. ABTR's reordering technique can nevertheless be integrated in a straightforward manner in most extensions of ABT such as AAS and DMAC, as we reported and experimented in [18, 20].

1.4 Organization of the document

First we introduce the basic asynchronous backtracking (ABT) algorithm. Then we introduce the reordering counters and how vector clocks (signatures) are induced by such counters. This technique is similar to one used for tagging assignments in AAS [16]. For simplicity here we adopt the representation of this signatures where 0-valued counters are listed [11, 31].

In a subsequent section we first illustrate our extension of ABTR for the case of dynamic reordering schemes it previously supported (min-domain, approx-AWC2 and approx-DB), showing that no behavior changes there, and that we have a pure generalization. Then we show how to specify the exact heuristic of DB such that it can be used with ABTR. Three implementations of approx-AWC1 are also introduced. These versions are compared describing the available experiments. A methaphoric way to describe ABTR is provided in the Annexes Section 9.

2. RELATED WORK

The first complete asynchronous search algorithm for DisCSPs is the Asynchronous Backtracking (ABT) [27]. The approach in [27] considers that each agent maintains only one distinct variable. More complex definitions were given later [29, 26]. Other definitions of DisCSPs have considered the case where the interest on constraints is distributed among agents [30, 23, 16]. [23] proposes algorithms that fit the structure of a real problem (the nurse transportation problem). The Asynchronous Aggregation Search (AAS) [16] 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. The order on variables in distributed search was so far addressed in [3, 25, 23, 6, 1, 17, 20, 10, 11, 31, 4, 19], showing the interest and strong impact it has on the solving algorithms. The work in [1] is the first to evaluate the impact of reordering in asynchronous backtracking, by introducing synchronization points that separate short asynchronous epochs, and report 30% improvement for good ordering heuristics over random ordering. In [17] we show how to apply those heuristics without synchronizations, introducing the ABTR protocol. Experimentation of min-domain heuristics with ABTR, combined with consistency maintenance and aggregations of AAS (version called MAS) is reported in [20, 18].

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. If one does not assume generalized FIFO (aka causal order) channels, a **local counter**, $C_i^{x_i}$ (not a reordering counter), 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 (i.e., the value of $C_i^{x_i}$).*

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 8 (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.*

Each agent stores at most one nogood for each value in its domain, in the array *nogood-list*.

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?** message.

The agents start by instantiating their variables concurrently 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 [26].

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 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. We will also say that $\neg N_2$ is a **worse nogood** then $\neg N_1$.

The next property is mentioned in [28, 2] 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.*

```
when received (ok?,$\langle x_j, d_j, c_{x_j}\rangle$) do
    if $c_{x_j}$ larger than known counter for $x_j$ then
        add($x_j$,$d_j$,$c_{x_j}$) to agent_view;
        clean nogoods;
        check_agent_view;
    end
end do.
when received (nogood,$A_j$,$\neg N$) do
    if an assignment in $N$ is old then return;
    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
        send (ok?,$\langle x_i$, current_value, $c_{x_i}\rangle$ ) 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?,$\langle x_i, d, c_{x_i}\rangle$ ) 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 $V$;
1.1     send (nogood,$A_i$,$V$ ) to $A_j$;
        remove ($x_j$,$d_j$,$c_{x_j}$) from agent_view; clean nogoods;
    check_agent_view;
end do.
```

Algorithm 1: *Procedures of A_i for receiving messages in ABT. The counters c_{x_i} used in this version for tagging assignments allow to better treat non-causal order channels and to recognize in received nogoods the assignments that are newer than what the agent knew.*

4. Signatures VECTOR-CLOCKS

We now introduce reordering counters and how "vector-clocks" [9, 17, 11] called signatures are induced by such counters.

If we need that agents propose reordering asynchronously (e.g. as in AWC or as required for security in [15]), the order typically changes before an agreement could be reached. What one has to do in this case is to ensure that *the sequence of reorderings eventually terminates.* We later prove this property for our protocol, given any reordering heuristic that respects certain simple rules.[5] For this, one must enable agents to coherently decide *which of two received ordering requests is the strongest.*

When agents receive several asynchronous reordering requests, to help select which reordering request is the strongest, messages are tagged with signatures vector-clocks. These are obtained as follows. A set of reordering counters $\{C^r_0, ..., C^r_{n-2}\}$ are maintained, one for each suffix of agents of size n-k. The reordering counter C^r_k is incremented each time a reordering is requested between A^{k+1} and some other agents in $\{A^{k+2}, ..., A^n\}$. A vector $|C^r_0, ..., C^r_{n-2}|$ with the current values of these vector-clocks, called *signatures* vector-clock, tags each reordering request.[6] Also, the current ordering known to an agent, together with the signatures vector-clock of the ordering, tag each **ok?** and **nogood** message that it sends.

REMARK 1. *Note that by dropping 0-valued reordering counters, the signatures vector clock $|3,0,5,1,0,0|$ for an order could be written as $|0{:}3|2{:}5|3{:}1|$, namely a sequence of pairs $k{:}C^r_k$, called signature. This version was the one originally used in [17], and the simpler version introduced above and used in the following was proposed in [11].*

Between two signatures vector-clocks, the strongest is given by the (inverse) lexicographic order. E.g., $|3,1,2,0|$ is stronger than $|3,0,3,1|$, since it also integrates the higher priority reordering registered by counter C^r_1.

REMARK 2. *With the original representation for signatures [12], $|1{:}3|2{:}1|3{:}2|4{:}0|$ is stronger than $|1{:}3|3{:}3|4{:}1|$ since the pair $|2{:}1|$ has priority over pair $|3{:}3|$. The advantage of signatures over signatures vector-clocks consists only in*

[5]We show that dynamic min-domain, as well as the reordering of dynamic backtracking do respect those rules.

[6]Note that counters C^r_k for k=n and k=$n-1$ are redundant, since reordering of one agent or 0 agents is of no interest.

smaller space complexity for sparse vector-clocks. This is at the expense of a less simple implementation, which may explain why most subsequent works use signatures vector-clocks rather than signatures [11, 31].

An agent builds the signatures vector-clocks he generates for outgoing messages only with the reordering counter values that he knows. Each agent monitors the values of reordering counters for which he is not the current maintainer by recording them from the strongest signatures that he receives with incoming messages.

5. REORDERING DETAILS

Now we show how the signatures described in the previous section help agents during the search to asynchronously and concurrently propose new orderings dynamically.

We attach to each ordering a *signature* or signatures vector-clocks as defined in the previous section. The agent maintaining the counter C^r_i in a given order o is denoted $R^i(o)$. $R^i(o)$ can propose orderings that reorder only agents on positions p, $p>i$, in o. With the initial version of ABTR/ABT_DO, $R^i(o)$ is the same agent as $A^i(o)$, the notations being synonyms.

Definition 5 (Known order O^{crt}). *The ordering o known by an agent A_i is the order with the strongest signature among those received by A_i. This ordering is referred to as the* known order of A_i *and is denoted O^{crt}.*

Rule 9 (Requested order). *An ordering, o', proposed by $R^i(o)$ is such that the agents placed on the first i positions in the* known order of $R^i(o)$ *must have the same positions in o'. o' is referred to as the* requested order of $R^i(o)$.

Let us consider two different requested orderings, o_1 and o_2, with their corresponding signatures h_1 and h_2: $O_1 = \langle o_1, h_1\rangle$, $O_2 = \langle o_2, h_2\rangle$.

Definition 6 (Reorder position). *The* **reorder position** *of h_1 and h_2,* $\mathbf{R(h_1,h_2)}$, *is the position of the highest priority agent reordered by some request $O = \langle o, h\rangle$ where h's strength is between that of h_1 and of h_2. If u is the lowest index where signatures vector-clocks h_1 and h_2 differ (assuming the index in vector-clocks starts with 0), then $R(h_1,h_2)=u+1$.*

New (optional) messages for enabling reordering are:

- **heuristic** messages for heuristic dependent data (e.g. announcing changes of a variable's domain size to higher priority agents), and
- **reorder** messages requesting a new ordering o and tagged with signatures h, $\langle o, h\rangle$.

An agent R^i announces its requested order o by sending **reorder** messages to successor agents $\mathcal{S}_i = \{A^k(o) \mid k > i\}$, and to all agents $R^k(o), k > i$, not in $\mathcal{S}_i$. Each agent A_i stores its *known order* denoted $O^{crt}(A_i)$. For allowing asynchronous reordering, each **ok?** and **nogood** message receives as additional parameter an order and its signature (see Algorithm 1). The **ok?** messages hold the strongest *known order of* the sender. Each **nogood** message sent from A^j to A^i is tagged with an order consisting of the prefix of i agents in the $O^{crt}(A^j)$ and the prefix of i counters in the signatures vector-clock of A^j's O^{crt} (denoted O^{crt}_i). For example, in Figure 5 the 4th message contains the prefix (A_1, A_2) of the current order at A^3, (A_1, A_2, A_3).

When A_i receives a message which contains an order with a signature h that is newer than the signature h^* of $O^{crt}(A_i)$, the assignments known by A_i for the variables x^k, $k \geq R(h, h^*)$, are invalidated.

Sometimes it is possible to assume that the agents want to collaborate in order to decide an ordering. The **heuristic** messages are intended to offer data for reordering requests. Its parameters depend on the used reordering heuristic. For example, we do not need to explicitly exchange **heuristic** messages to achieve Dynamic Backtracking's reordering scheme, but we need them for the min-domain heuristic with problems having private constraints or when maintaining arc-consistency [20]. The **heuristic** messages can be sent by any agent to the agents R^k (i.e., the agents currently holding the different reordering counters C^r_k).

If used, **heuristic** messages may be sent to R^k only at start or in response to a new assignment for $x^j, j\leq k$, (the computation time needed between receiving such an assignment and sending the **heuristic** message is denoted $\mathbf{t_R}$, see Figure 3). Sometimes an agent does not get the **ok?** messages sent by higher priority agents because he is not in their outgoing links (see A_2 in Figure 4), but can send heuristic messages if some other mechanism guarantees the finiteness of t_h (as we will show for DB's heuristic). We consider that an agent R^k receiving an assignment for some variable $x^j, j\leq k$, implicitly delivers to itself a **heuristic** message. This convention unifies the framework with the cases where no explicit **heuristic** messages are needed. We can say that the sending/reception of an assignment (and the start), are "enabling events" for sending a corresponding **heuristic** message.

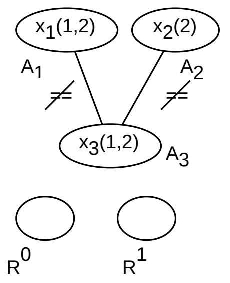
The local computation time needed by an agent R^i to analyze a given received **heuristic** message for deciding to request an ordering is denoted by $\mathbf{t_r}$.

5.1 Proof and Examples

In the previous section we did not specify how reordering counters are distributed to agents. The distribution is defined by a convention for dynamic mapping of reordering counters to agents, separately defined for each reordering heuristic.

Example 1. *With the approx-AWC2 reordering (used in [31]) as well as with the other reordering heuristics studied such as min-domain [20, 31], we have no C^r_0 (the first agent is not reordered by anybody). R^1, the agent owning C^r_1, is the agent on the first position since he can reorder agents on positions following the first position. For any other k, the agent owning C^r_k (R^k), is the agent on the k^{th} position (as it can reorder agents on positions following the k^{th} position).*

In general we prove that, when proposing a new order, an agent R^i can be allowed to explicitly request a remapping of the counters $\{C^r_i,\ldots,\ C^r_{n-2}\}$ to different agents $\{A_{k_i}, \ldots, A_{k_{n-2}}\}$, by attaching the description of the new mapping to the **reorder** message: "$C^r_i \rightarrow A_{k_i}, \ldots, C^r_{n-2} \rightarrow A_{k_{n-2}}$". An agent A_{k_j} receiving such a mapping request (by which it is delegated as holder of C^r_j), will start to maintain the counter C^r_j initially set to

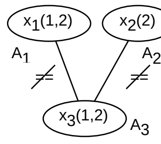


$A_1/\mathbf{A^1}$	$\xrightarrow{\mathbf{ok?}\langle x_1,1,1\rangle(A_1,A_2,A_3)\vert 0,0\vert}$	A_3
$A_1/\mathbf{A^1}$	$\xrightarrow{\mathbf{heuristic}\langle x_1,1,1\rangle(A_1,A_2,A_3)}$	R^1
$A_2/\mathbf{A^2}$	$\xrightarrow{\mathbf{ok?}\langle x_2,2,1\rangle(A_1,A_2,A_3)\vert 0,0\vert}$	A_3
$A_3/\mathbf{A^3}$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

Figure 5: 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. The example focuses on the usage of reordering counters, while the mapping of the reordering counters to agents is left unspecified.

```
when received (ok?,$\langle x_j, d_j, c_{x_j}\rangle$,$\langle o,h\rangle$) do
    validOrder$\leftarrow$ getOrder($\langle o,h\rangle$);
    if ($\neg$validOrder $\wedge$ ($c_{x_j}$ smaller than known counter for $x_j$)) then return;
    add($x_j$,$d_j$,$c_{x_j}$) to agent_view;
    clean nogoods;
    check_agent_view;
    send optional heuristic messages; // e.g. announcing domains for min-domain with private constraints;
end do.
when received (nogood,$A_j$,$\neg N$,$\langle o,h\rangle$) do
    validOrder$\leftarrow$ getOrder($\langle o,h\rangle$);
    if ($\neg$validOrder $\wedge$ ($A_i \neq$CEA($\neg N$))) then return;
    if (($\langle x_i,d,c\rangle \in N$ and nogood-list already has nogood not worse than $\neg N$ for $x_i$=d) 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; clean nogoods;
        put $\neg N$ in nogood-list for $x_i$=$d$;
        add other new assignments to agent_view; clean nogoods;
    end
    old_value $\leftarrow$ current_value; check_agent_view;
    when old_value = current_value and if $A_j$ has lower priority than $A_i$
        send (ok?,$\langle x_i$, current_value, $c_{x_i}\rangle$,$O^{crt}$) to $A_j$;
    send heuristic messages according to used heuristic;// e.g. for DB, approx-AWC1, approx-AWC2, etc.;
end do.
when received heuristic(index, data) do
    if (not currently owning $C^r_{index}$) then return;
    if (no new ordering is inferred due to data) then return;
    $C^r_{index}$++;
    use data to generate new order $O' = \langle o',h'\rangle$; // according to used reordering heuristic;
    send (reorder,$O'$) to the new owners of each $C^r_j$ (i.e. $R^j(o')$) for all $j \geq index$, as defined by the used reordering heuristic;
    getOrder($O'$);
end do.
function getOrder($\langle o,h\rangle$) $\rightarrow$ bool // returns true for a valid order and false for an order less strong than $O^{crt}$
    when $h$ is invalidated by the signature $O^{crt}$ then return false;
    when not newer $h$ than $O^{crt}$ then return true;
    $j \leftarrow$ R($h$, signature of $O^{crt}$); // i.e., reorder position for $h$ and the signature of $O^{crt}$;
    invalidate assignments for $x^k, k \geq j$;
    $O^{crt} \leftarrow \langle o,h\rangle$;
    for every reordering counter $C^r_w$ that I own in ordering $o$ do
        $C^r_w \leftarrow$ the value for $C^r_w$ in $h$; // here is the difference between ABTR/ABT_DO and its generalization;
    end do
    return true;
end.
```

Algorithm 2: *Procedures of A_i for receiving messages in ABTR. The procedures* **check_agent_view** *and* **check_agent_view** *are as in ABT, except that* **ok?** *and* **nogood** *messages sent to any agent A_j also take O^{crt} (respectively O^{crt}_j) as parameter.*



$A_1/\mathbf{A^1}/R^1$	$\longrightarrow$ **ok?**$\langle x_1,1,1\rangle(A_1,A_2,A_3)\|0,0\|\longrightarrow$	A_3
$A_2/\mathbf{A^2}/R^0/R^2$	$\longrightarrow$ **ok?**$\langle x_2,2,1\rangle(A_1,A_2,A_3)\|0,0\|\longrightarrow$	A_3
$A_3/\mathbf{A^3}/R^3$	**nogood**$\neg(\langle x_1,1,1\rangle,\langle x_2,2,1\rangle)(A_1,A_2)\|0,0\|\longrightarrow$	A_2
$A_1/\mathbf{R^1}/A^1$	$\longrightarrow$ **reorder** $(A_1,A_3,A_2)\|0,1\|\longrightarrow$	A_3
$A_2/\mathbf{R^0}/A^3/R^2$	$\longrightarrow$ **reorder** $(A_3,A_1,A_2)\|1,0\|\longrightarrow$	A_3
$A_1/\mathbf{R^1}/A^1$	$\longrightarrow$ **reorder** $(A_1,A_3,A_2)\|0,1\|\longrightarrow$	A_2
$A_2/\mathbf{R^0}/A^3/R^2$	$\longrightarrow$ **reorder** $(A_3,A_1,A_2)\|1,0\|\longrightarrow$	A_1
$A_3/\mathbf{A^1}/R^1$	$\longrightarrow$ **ok?**$\langle x_3,1,2\rangle(A_3,A_1,A_2)\|1,0\|\longrightarrow$	A_1
$A_3/\mathbf{A^1}/R^1$	$\longrightarrow$ **ok?**$\langle x_3,1,2\rangle(A_3,A_1,A_2)\|1,0\|\longrightarrow$	A_2

Figure 6: Example for ABTR with random reordering. R^i delegations are done implicitly 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 add-link is not shown.

the value that C^r_j has in the signature of the **reorder** request. When an agent R^i requests a remapping of a counter C^r_j, $j{>}i$, not maintained by itself, R^i will specify the value 0 for C^r_j in the signatures vector-clock of his request, thus resetting C^r_j (while incrementing C^r_i).

Remark 3. *Note that an implementation does not necessarily need to explicitly append such a mapping to messages. For example, with min-domain [20] or approx-AWC2 [31], knowing the current ordering is sufficient for each agent to infer what is the mapping that he is expected to assume, according to the example mentioned before: $C^r_0{\to}A_1, C^r_1{\to}A^1, ..., C^r_{n-2}{\to}A^{n-2}$. In the above notation we mapped C^r_0 to A_1, but actually C^r_0 is not used in the min-domain and in the approx-AWC2.*

At a certain moment, due to message delays, there can be several agents believing that they are delegated to hold C^r_i (i.e. to be R^i) based on the ordering they know. However, any other agent can coherently discriminate the strongest among messages from such simultaneous R^is using the signatures that R^is generate.

Example 2. *For example, in Figure 6 we describe the case where R^i is always mapped to A^i similar to the case of min-domain, but assume we have (and use) a counter C^r_0. R^i can send a* **reorder** *message within time t_r after an assignment is made by A^i since a* **heuristic** *message can be considered to be implicitly transmitted from A^i to R^i (being the same agent). We also consider that A_2 holds C^r_0 (i.e., is R^0). R^0 and R^1 propose one random ordering each, asynchronously. The receivers discriminate based on signatures that the order from R^0 is the strongest. The known assignments and nogood are discarded. In the end, the* known order *for A_3 is $(A_3, A_1, A_2)|1, 0, 0|$.*

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.

Lemma 1. *$\forall i$, in finite time t^i either a solution or failure is detected, or all the agents A^j, $0{<}j{\le}i$, reach quiescence in a state where they are not refused an assignment satisfying their constraints (that they enforce) with their* agent_view.

Proof. The correct and coherent function of reordering counters is guaranteed with dynamic remapping by construction given the new rule of setting the counter's value from the signatures vector-clock tagging the reordering/remapping request. Namely, a counter C^r_i cannot be reset except if higher priority counters are incremented.

The property is now proved by induction:
Basis of the induction. After $t_h + t_r + \tau$, R^0 is fixed and the property is true for the empty set.

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

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^i_o and the induction step is proved.
From here, the induction step is proven since it was proven for all alternatives.
In conclusion, after t^i_o, within finite time, the agent A^i either finds a solution and quiescence or an empty nogood signals failure.
The property is therefore proven by induction on i □

Theorem 1. *Extended ABTR is correct, complete and terminates for any heuristics where the local computation times t_h of A^i reacting with* **heuristic** *messages to an* **ok?** *message can separately be proven finite after $A_j, j < i$, reach quiescence. (Messages travel times τ is assumed finite, i.e. no message is lost, and the time t_r of answering with* **reorder** *messages to a* **heuristic** *message is also assumed fi-*

nite[7].

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 Lemma 1 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 Lemma 1, all the agents agree with their predecessors and the set of their assignments is a solution. □

REMARK 4. *Note that the Theorem 1 does not prove that Dynamic Backtracking's heuristic (or some other named heuristic) would lead to a correct and terminating extended ABTR. It only proves that any heuristic and dynamic mapping policy respecting the rules mentioned so far leads to a correct and terminating protocol with extended ABTR. In subsequent sections one has to prove separately for each proposed heuristic the fact that it respects the conditions set so far (in particular the finiteness of the delay for heuristic, t_h.*

5.2 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, $\langle o, h\rangle$, placing A_i on position j, $A_i^j(o)$ 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 he cannot check the validity of such nogoods in o.

To ensure that assignments that might have been lost at reordering but may still be present in saved nogoods are correctly re-established, each 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 (see Algorithm 3).

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 valid 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. DYNAMIC BACKTRACKING'S HEURISTIC FOR ABTR

In this section we detail how the exact reordering heuristic used in Dynamic Backtracking can be implemented with ABTR1. One obtains DB's reordering heuristic by implementing a counter-to-agent mapping where the reordering counter C_j^r is hold by the agent A^{j+1}. This way C_0^r is maintained by the agent(s) on the first position in their known order, C_1^r by the agent(s) on the second position, etc. Knowing the current ordering is sufficient for each agent to infer what is the mapping that he is expected to assume, i.e.: $C_0^r \rightarrow A^1, C_1 \rightarrow A^2, ..., C_{n-2}^r \rightarrow A^{n-1}$. Therefore, the requested mapping does not need to be explicitly appended to messages, but can be inferred from the current order.

Assume an agent A^i with current ordering o receives from A^k a valid nogood that rejects his current assignment. For each such event we will let $A^i(o)$ request a reordering o' placing itself on the position k, and requesting that all agents $A^{i+1}(o)$ to $A^k(o)$ increase their priority with one position, being moved to positions i to $k-1$, respectively. This protocol is called ABTR-db. Its behavior is compared in Figures 1, 4 with behaviors possible in the previous ABTR algorithms [20, 31].

To prove correctness, it is required that $\exists t_r, t_h$ finite, such that A^{k+1} (which holds the counter C_k^r) does not send reordering requests with delay more than $t_r + t_h + \tau$ after an instantiation is done by some $A^j, j \leq k$, (see Theorem 1). $t_r + t_h + \tau$ accounts for the time taken by the **ok?** messages with the assignment to reach A^{k+1}, and the times allowed for the deliberation after the enabling event, i.e., t_r and t_h. We prove this in the following.

REMARK 5. *The fact that the time between start and when a last nogood is received by a variable (generating an implicit* **heuristic** *message) is finite, is guaranteed by the the proof of termination of Dynamic Backtracking [5].*

However in Algorithm 2 we slightly departed from the Dynamic Backtracking's nogood storage behavior by replacing existing nogoods with "better" nogoods when these are available. Below we explain this feature and we prove that it does not modify the termination of dynamic backtracking, therefore maintaining the needed guarantee that t_h is finite, as requested by the extended ABTR.

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 (i.e., terminate).*

Proof. Each valid nogood received by A_k^i eliminates a value for x_k. Each value eliminated by such a nogood after all predecessors $A^j, j < i$, reach quiescence, will never be available again 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 at most n times. All agents learn a new assignment within time $n\tau$ from its proposal. Therefore, all valid nogoods are received in a finite time. □

COROLLARY 1. *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 $A^l, l < k$.*

7. APPROX-AWC1 HEURISTIC FOR ABTR (ABTR-wc)[8]

To obtain approx-AWC1 one needs the same counter-to-agent mapping as for DB's reordering heuristic, namely $C_0^r \rightarrow A^1, C_1 \rightarrow A^2, ..., C_{n-2}^r \rightarrow A^{n-1}$. The requested mapping does not need to be explicitly appended to messages, but can be inferred from the current order.

[7]We describe versions not needing this assumption in [22].

[8]In the report [19] we denoted it ABTR-wc.

```
function getOrder(⟨o, h⟩) → bool // returns true for a valid order and false for an order less strong than $O^{crt}$
    when $h$ is invalidated by the signature $O^{crt}$ then return false;
    when not newer $h$ than $O^{crt}$ then return true;
3.1 $j \leftarrow$ new position of $A_i$ in $o$;
    invalidate assignments for $x^k(o), k \geq j$;
    $O^{crt} \leftarrow \langle o, h \rangle$;
3.2 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;
    for every reordering counter $C^r_w$ that I own in ordering $o$ do
        $C^r_w \leftarrow$ the value for $C^r_w$ in $h$; // here is the difference between ABTR/ABT_DO and its generalization;
    end do
    return true;
end.
```

Algorithm 3: *Procedure of A^v_i for receiving new orderings in ABTR1 (reordering saving additional nogoods). The Line 3.2 is added to correctly handle savings in nogoods due to change at line Line 3.1.*

```
when received (nogood,$A_j$,$\neg N$,$\langle o, h\rangle$) do
    validOrder← getOrder($\langle o, h\rangle$);
    if ($\neg$validOrder $\wedge$ ($A_i \neq$CEA($\neg N$))) then return;
    if (($\langle x_i, d, c\rangle \in N$ and nogood-list already has nogood not worse than $\neg N$ for $x_i$=$d$)
                    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;
    end
    when (validOrder and valid $\neg N$ and $d$=current_value)
        $v \leftarrow$ my position in $O^{crt}$;
4.1     $C^r_{v-1}$++;
        new_order $\leftarrow \{A^1, A^2, ..., A^{v-1}, A_j, ...\}$; // remaining agents after $A_j$ are in lexicographic order;
        getOrder($\langle$new_order,build new signature$\rangle$);
4.2     make sure to send $O^{crt}$ either via ok? or via reorder messages to all $A^j, j \geq v$;
    check_agent_view;
end do.
```

Algorithm 4: *Procedure of A^v_i for receiving* **nogood** *messages in ABTR-wc. This is the optimized implementation used in experimentation and no heuristic messages are exchanged, but rather the processing for receiving implicitly sent* **heuristic** *is also done in this function.*

Each agent A^k maintains counter C^r_{k-1} and cedes its position to the sender of each valid nogood received (Algorithm 4). Let us assume again that an agent A^i with current ordering o receives from A^k a valid nogood that rejects his current assignment. For each such event $A^i(o)$ requests a reordering o' placing A^j on the position i, and decreasing with one position its priority and the priority of all agents $A^{i+1}(o)$ to $A^{k-1}(o)$, being moved to positions i+1 to k, respectively (see Figure 1.b). This protocol is called ABTR-wc. The correctness proof of ABTR-db also applies here without any modification.

7.1 Saving effort across reordering for ABTR-wc

In the Algorithm 4, the line 4.1 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 payload for messages. However, due to reordering involved on future agents many of the nogoods owned 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 1.1 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 between O^{crt} and the orderings received with **nogood** messages. In ABTR-wc1 the *new_order* at line 4.1 becomes $\{A^1, ..., A^{v-1}, A_j, A_i, ...\}$, where the agents following A_i are ordered according to L^o. A pseudo-code showing these changes is presented in [19].

A further modification we make to ABTR-wc1 is that each new proposed ordering is sent at line 4.2 to all agents so that all of them can update their *last_known_successor_order*. This last version is denoted ABTR-wc2.

8. EXPERIMENTATION

8.1 Termination and Solution detection

This section is not important for understanding ABTR, but describes the way in which we recover the solution (possibly even prior to quiescence) in our distributed implementation.

In ABT a solution is found when the agents reach quiescence. As proposed in [27] 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 [16] 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. [21] 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 [21] 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 [21] 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 specially when the constraints are public. Otherwise, the technique in [21] 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. □

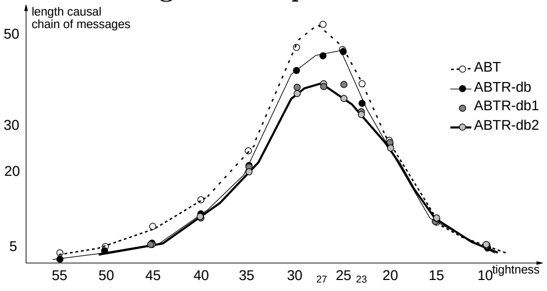
PROPOSITION 2. *The solution detection algorithm based on SRCCSs detects a solution for ABTR in finite time if the search does not fail.*

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. □

8.2 Experiments

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

Figure 7: Experiments



We have generated problems of variable tightness for a density of 27% where each variable has 3 values. Each point in Figure 7 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. For under-constrained problems (tightness under 15%) 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 to ABT.

Other experiments, based on the the dynamic min-domain heuristic were also performed and described [20] with a version of ABTR based on DMAC rather than based on ABT. However, DMAC with dynamic min-domain performed worse that DMAC without min-domain reordering. Experiments with the approx-AWC2 heuristic depicted in Figure 1 are reported in [31]. That report mentions improvements of one order of magnitude when compared to ABT without reordering. However, those experiments are based on a simulator. Also the metric used there measures the sequential constraint checks rather than the length of the longest causal chain of messages [8] reported here and which correlates better with the running time [14]. Unreported implementation details could also be part of the explanation of the discrepancy between the results in the two experiments. Our results correlate with the prediction in [1].

It is of a certain interest to perform experimental evaluations with the real implementation for additional heuristics, in particular for directly comparing approx-AWC1 with approx-AWC2. Unfortunately we no longer have access to a sufficiently large network of computers, but consider that the currently available experimental and theoretical results are already of sufficient interest. The implementation used for the experiments reported here is available for free download on our web-site and can be used for further research by whoever owns a sufficiently large network.

9. ANNEXES: REORDERING AND MODERN DEMOCRACIES

This section contains a metaphoric way of explaining ABTR that we first used in [12], similar to Lamport's Byzantine generals and Paxos metaphors. The observation is that one can define an understandable modern democracy that illustrates characteristics of the reordering technique introduced here.

9.1 Ciglean's Democracy

Following the failure of the Eastern Block and the evaporation of the last traces of the socialist administrative structures, a small unreachable isolated village in a crater on top of a hill develops a new and original democracy that we will call *Ciglean's democracy*.

9.1.1 Administrative body in Ciglean

After long delays when nobody wanted to take on administrative tasks, the people decided that everybody should have a responsibility. In Ciglean (100 inhabitants), each citizen must take an administrative office. The citizens of the village are old persons that live alone. To clearly define the responsibility of each administrative office, a strict hierarchy was defined on offices, where the *presidency* is the highest office and also cumulates the function of a prime-minister. The next office is for a second-minister, next for the third-minister, and so on. The lowest priority position is for the hundredth-minister. Each minister knowing the skills of the other citizens can distribute the tasks to lower priority ministers and accomplishes himself any task that cannot be done by any inferior.

The decisions and commands go from the president to all ministers and each minister can give commands to lower priority ministers. There exists only one semicircular road in Ciglean and all the houses are numbered distinctly. Initially the administrative positions were allocated according to the number of the house of the citizen. The president was the happy owner of the house numbered 1 (the influential peasant that had the idea of the new administration).

The phenomena observed in the first years of the Ciglean's Democracy was that the last minister had to repair the roads, to paint the trees, to gather the taxes, etc. while all the other ministers used to simply pass the commands that they received to their inferiors and were doing only accidentally some useful administrative work. Since there is too much work to do for a single person, the road of the village kept degrading each year.

9.1.2 Legislative body of Ciglean

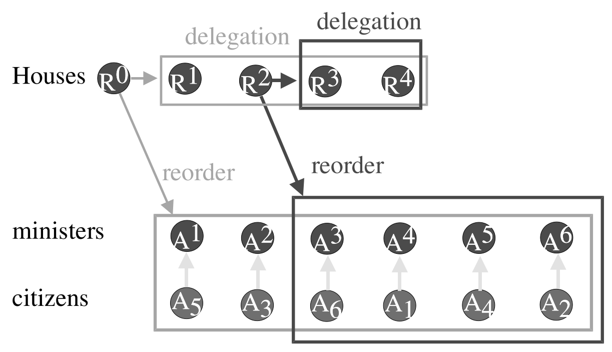


Figure 8: Democracy: People choose president and delegate deputies. Deputies approve cabinet and delegate parliamentary committees.

After two years, the whole population of the village meeting each Sunday evening in the local church accepted that inevitably a rotation of the positions is unavoidable in order to make Ciglean's road practicable. A system of elections was put up. Since it was not known in advance how efficient each citizen is in a given office, a mechanism is defined to allow reordering of offices whenever needed.

As the citizens did not want to change the tasks of each administrative office, with which whey were used, a parallel system was designed, where a hierarchical legislative body has the right to decide whenever certain re-delegations were felt needed. The person being president at the moment, has proposed for the legislative body a similar structure to the one of the administrative body.

A total of 99 legislative offices are organized. The priorities of these legislative offices are totally and statically ordered. The highest priority legislative authority is the whole population of the village meeting each Sunday evening in the courtyard of the church. The other legislative offices are called the First-House, the Second-House, etc. The parliament of Ciglean has 98 Houses.

Each time somebody asks, the village votes a new delegation of the administrative body that can change the holder of any administrative office. Each k^{th}-House can shuffle the holders of any t^{th}-minister where $t>k$. For example, no House can re-delegate the president and the 2^{nd}-House cannot redelegate the second-minister and the president.

With the opportunity of an administrative reordering vote, each legislative office can also redelegate the holders of all lower priority legislative offices by redefining the citizens that have to sign in order to enable the decision of each given House. These notions are also exemplified in Figure 8.

9.1.3 Signatures

Each decision of a legislative office is signed with a set of stamps that the office receives on its delegation. Each time a certain legislative House is delegated (its members are listed) the new holders of the House receive a copy of all the stamps of the authority delegating them plus a stamp naming the authority taking the decision and the reference number of the decision. The new members of the House receive a new set of forms where they can fill up decisions starting with reference number 1 and incrementing the decision reference number after each new decision.

Each decision for redelegating holders of administrative offices is signed with all the stamps of the House taking the decision, as well as with the name of the House and the reference number of the decision.

Whenever the members of a House or some ministers see a signature that is stronger than the one delegating her to a certain office, the old delegation is forgotten and the corresponding stamps are recycled.

9.1.4 Self Redelegation

Citizens of the village sometime leave for a while and they want to redelegate another group to hold the House. When this problem first appeared, the chosen solution was to let all the stamps of the House be simply handled to the new holders and the empty decision forms of the old holders are transmitted to the new ones (with the old reference numbers).

There are continuous discussions in Ciglean whether the general assembly of the village should be enabled to similarly redelegate its authority to some single person or to some group during periods of intensive agricultural work, but the citizens are afraid that the power will be retained by some person that will never re-convoke and yield back the main legislative power to the whole population.

9.1.5 Responsiveness

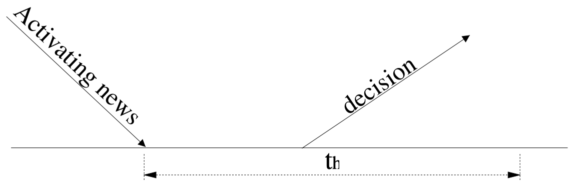


Figure 9: To ensure termination, the responsiveness of the Houses is limited to a timeout t_h.

To ensure that no malicious House would keep reshuffling the ministers for ever such that no work could ever be done, a certain limit is put on the validity of a House. A k^{th}-House can only issue new decisions within a time t_h after an information concerning (see Figure 9):

- the need of a new administrative work,
- a decision of a higher priority House
- a decision of a t^{th}-minister, with $t \leq k$

10. CONCLUSIONS

Reordering is a major issue in constraint satisfaction and is the main strength of Dynamic Backtracking and Asynchronous Weak-Commitment (AWC). Complete polynomial space asynchronous search algorithms for DisCSPs prior to ABTR required a static order of the variables. Asynchronous Backtracking with Reordering (ABTR aka ABT_DO) [17, 31] is an asynchronous complete algorithm with polynomial space requirements that has the ability to concurrently and asynchronously reorder variables (except for the first variable) during systematic asynchronous search. In previous versions of ABTR, the first variable cannot change its position. Asynchronous reordering is required for security reasons in managing coalitions in automated negotiation (see [15]).

Here we present an extension of ABTR/ABT_DO allowing for ABTR-db, a dynamic variable reordering heuristic for ABTR that exactly replicates the one employed in Dynamic Backtracking and that requires no exchange of **heuristic** messages. This reordering heuristic was not previously known to lead to a correct and terminating version of ABTR [20, 17, 31], as it was not covered by the initial proof of ABTR. Here we extend ABTR to allow for DB's reordering heuristic in a way that leads to a protocol that is correct and terminates. For correctness with the heuristic of Dynamic Backtracking we add the following rule (allowing to dynamically move reordering counters between agents): *At a reordering changing the priority of some agent, the counter of the agent receiving higher priority is set to the value of the reordering counter for that position, that is found in the signatures vector-clock tagging the request for reordering.* Therefore, an agent can move itself while keeping counters coherent, and the agent on the first position can also be reordered.

Additional reordering heuristic are enabled by the generalized proof, and the task of exploring them for useful properties remains open. An example consists of better approximations of the reordering heuristic of AWC, as compared to the previously possible approx-AWC2. An experimentation was led comparing ABTR-wc to ABT for a real distributed implementation. The discrepancy in efficiency of reordering found with respect to a recent report based on simulators is significant, confirming the earlier prediction in [1] and revealing the need for additional research on understanding the factors important in such distributed processes and improved simulators.

More experimental comparisons with additional heuristics using our real implementation define an interesting open research direction. We cannot follow this direction since we no longer have access to a network of computers of the corresponding size, but since 2004 our implementation is freely available for download to whoever owns such resources [13].

The interest of the reordering heuristic experimented here is suggested by the fact that it is the closest ABTR heuristic to the famously efficient AWC.

11. 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] C. Bessiere, I. Brito, A. Maestre, and P. Meseguer. Asynchronous backtracking without adding links: A new member in the abt family. *Artificial Intelligence*, 161:7–24, 2005.

[3] Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of IJCAI 1991*, pages 318–324, 1991.

[4] C. Fernàndez, R. Béjar, B. Krishnamachari, and C. Gomes. Communication and computation in distributed CSP algorithms. In *CP*, pages 664–679, 2002.

[5] M. L. Ginsberg. Dynamic backtracking. *Journal of AI Research*, 1, 1993.

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

[7] W. Havens. Nogood caching for multiagent backtrack search. In *AAAI Constraints and Agents Workshop*, 1997.

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

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

[10] A. Meisels and I. Razgon. Distributed forward checking with dynamic ordering. In *COSOLV*, pages 21–27, 2001.

[11] Viet Nguyen, Djamila Sam-Haroud, and Boi Faltings. Dynamic distributed backjumping. In *DCR Workshop*, pages 51–65, 2004.

[12] M.-C. Silaghi. *Asynchronously Solving Distributed Problems with Privacy Requirements.* PhD Thesis 2601, (EPFL), June 27, 2002. `http://www.cs.fit.edu/~msilaghi/teza`.

[13] M.-C. Silaghi. DisCSP algorithms on the Mely

platform.
`www.cs.fit.edu/~msilaghi/discsps/sJavap/`, 2004.

[14] M.-C. Silaghi and B. Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence Journal*, 161(1-2):25–53, October 2004.

[15] M.-C. Silaghi, D. Sam-Haroud, M. Calisti, 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. submitted to AA'2001.

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

[17] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. ABT with asynchronous reordering. In *IAT*, 2001.

[18] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous consistency maintenance with reordering. Technical Report #01/360, EPFL, March 2001.

[19] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Technical Report #01/364, EPFL, May 2001.

[20] 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.

[21] 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.

[22] Marius-Călin Silaghi and Boi Faltings. Openness in asynchronous constraint satisfaction algorithms. In *3rd DCR-02 Workshop*, pages 156–166, Bologna, July 2002.

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

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

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

[26] M. Yokoo. *Distributed Constraint Satisfaction.* Springer, 2001.

[27] 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.

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

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

[30] 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.

[31] R. Zivan and A. Meisels. Dynamic ordering for asynchronous backtracking on discsps. In *CP*, pages 161–172, 2005.