Asynchronous Backtracking Without Adding Links: A New Member in the ABT Family

Christian Bessiere, Arnold Maestre, Ismel Brito, Pedro Meseguer

Asynchronous Backtracking without Adding Links: A New Member in the ABT Family *

Christian Bessière
Arnold Maestre
LIRMM-CNRS, 161 rue Ada
34392 Montpellier, France

Ismel Brito
Pedro Meseguer
IIIA-CSIC, Campus UAB
08193 Bellaterra, Spain

Abstract

Following the pioneer work of Yokoo and colleagues on the *ABT* (asynchronous backtracking) algorithm, several ABT-based procedures have been proposed for solving distributed constraint networks. They differ in the way they store nogoods, but they all use additional communication links between unconnected agents to detect obsolete information. In this paper, we propose a new asynchronous backtracking algorithm which does not need to add links between initially unconnected agents. To make the description simpler and to facilitate the comparisons between algorithms, we present a unifying framework from which the new algorithm we propose, as well as existing ones, are derived. We provide an experimental evaluation of these algorithms.

1 Introduction

In the last years, the AI community has shown an increasing interest in distributed problem solving using the agents paradigm. In particular, several works have considered constraint satisfaction in a distributed form. They are motivated by naturally distributed constraint problems, for which it is not convenient to gather the whole problem knowledge into a single agent, and to solve it using centralized algorithms. The cost of collecting all information into a single agent could be taxing. Besides, gathering all information into a single agent could be undesirable for security or privacy reasons.

Considering complete algorithms for distributed constraint satisfaction, we mention the pioneer work of Yokoo and colleagues, who proposed the asynchronous backtracking algorithm (*ABT*) [15, 16]. *ABT* assumes a *variable-based* model, where each variable belongs to one agent and constraints are shared between agents. *ABT* requires a total agent ordering. When a dead-end is detected, it may require to add communication links between previously unconnected agents. Nogoods are exchanged among connected agents, and stored. Several extensions were proposed, as its adaptation to dynamic agent re-ordering [12] or consistency maintenance [11]. The distributed backtracking algorithm (*DIBT*) [6] performs graph-based backjumping without nogood storage. However, *DIBT* is not complete in its original form [14, 1]. A different approach is the asynchronous aggregation search algorithm (*AAS*) [10], that assumes a *constraint-based* model where each constraint belongs to one agent.

In this paper, we propose ABT_{not}, a new ABT-based algorithm that does not require to add communication links between initially unconnected agents. We first present ABT_{kernel}, a

*The work of Ismel Brito and Pedro Meseguer is supported by the Spanish project REPLI: TIC-2002-04470-C03.

simple procedure that contains the basic features of an asynchronous backtracking algorithm for variable-based distributed constraint satisfaction. We show that ABT_{kernel} is sound but may not terminate, and we identify the condition that has to be accomplished to obtain a complete procedure. Depending on the way this condition is implemented, we obtain already known algorithms (such as Yokoo's ABT or a correct version of $DIBT$), or new ones.[1] Among them, we emphasize on ABT_{not}, a new algorithm that behaves like ABT without requiring to add new communication links. This algorithm could be useful for applications where agents are not permitted to establish new communication links (due to a security policy of the site, for instance). In addition, the algorithmic analysis of ABT_{kernel} allows us to understand better the behavior of asynchronous search, clarifying the similarities and differences among ABT-based algorithms. We provide an experimental evaluation of these algorithms, using random problems and distributed meeting scheduling problems.

The rest of the paper is organized as follows. Section 2 contains some basic definitions and a description the asynchronous backtracking algorithm. In Section 3, we introduce ABT_{kernel} and provide the condition to achieve completeness. In Section 4, we implement this condition in several ways, deriving some already known algorithms and new ones like ABT_{not}. Section 5 contains the experimental evaluation. Finally, Section 6 contains some conclusions of this work.

2 Preliminaries

Classically, constraint satisfaction problems (CSPs) have been defined for a centralized architecture. A constraint network is defined by a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is the set of their respective finite domains, and $\mathcal{C}$ is a set of constraints specifying the acceptable value combinations for variables. The CSP involves finding values for the variables that satisfy all the constraints. We consider constraints involving two variables only, namely binary constraints. A constraint between x_i and x_j is denoted by c_{ij}.

A distributed CSP (DisCSP) is a CSP where the variables, domains and constraints of the underlying network are distributed among agents. A finite variable-based distributed constraint network is defined by a 5-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \varphi)$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ are as before. $\mathcal{A} = \{1, \ldots, p\}$ is a set of p agents, and $\varphi : \mathcal{X} \rightarrow \mathcal{A}$ is a function that maps each variable to its agent. Each variable belongs to one agent. The distribution of variables divides $\mathcal{C}$ in two disjoint subsets, $\mathcal{C}_{intra} = \{c_{ij} | \varphi(x_i) = \varphi(x_j)\}$, and $\mathcal{C}_{inter} = \{c_{ij} | \varphi(x_i) \neq \varphi(x_j)\}$, called intra-agent and inter-agent constraint sets, respectively. An intra-agent constraint c_{ij} is known by the agent owner of x_i and x_j, and it is unknown by other agents. Usually, it is considered that an inter-agent constraint c_{ij} is known by the agents $\varphi(x_i)$ and $\varphi(x_j)$ [16, 6]. As in the centralized case, a solution of a DisCSP is an assignment of values to variables satisfying every constraint. DisCSPs are solved by the collective action of agents $\mathcal{A}$, each holding a process of distributed constraint satisfaction.

Agents communicate by sending messages. It is assumed that the delay in delivering a message is finite but random. For a given pair of agents, messages are delivered in the order they were sent. For simplicity, in the rest of the paper we assume that each agent owns exactly one variable. We identify the agent number with its variable index ($\forall x_i \in \mathcal{X}, \varphi(x_i) = i$). From this assumption, all constraints are inter-agent constraints, so $\mathcal{C}_{inter} = \mathcal{C}$ and $\mathcal{C}_{intra} = \emptyset$.

Asynchronous backtracking (ABT) [15, 16] was a pioneer algorithm to solve DisCSP, its first version dating from 1992. ABT is executed autonomously and asynchronously by each agent in the network. Each agent takes its own decisions, informs other agents about them, and no agent has to wait for decisions of others. It computes a global consistent solution (or detects that no solution exists) in finite time; its correctness and completeness have been proven.

[1] A comprehensive description of the different algorithms derived from ABT_{kernel} is presented in [2].

ABT requires constraints to be directed. A constraint causes a directed link between the two constrained agents: the value-sending agent, from which the link departs, and the constraint-evaluating agent, to which the link arrives. To make the network cycle-free there is a total order among agents, which is followed by the directed links.

Each agent keeps its own agent view and nogood store. Considering a generic agent $self$, the agent view of $self$ is the set of values that it believes to be assigned to agents connected to $self$ by incoming links. The nogood store keeps nogoods as justifications of inconsistent values. Agents exchange assignments and nogoods. When $self$ makes an assignment, it informs those agents connected to it by outgoing links. $self$ always accepts new assignments, updating its agent view accordingly. When $self$ receives a nogood, it is accepted if it is consistent with $self$'s agent view, otherwise it is discarded as obsolete. An accepted nogood is added to $self$'s nogood store to justify the deletion of the value it targets. When $self$ cannot take any value consistent with its agent view, because of the original constraints or because of the received nogoods, new nogoods are generated as inconsistent subsets of the agent view, and are sent to the closest agent involved, causing backtracking. If $self$ receives a nogood mentioning another agent not connected with it, $self$ requires to add a link from that agent to $self$. From this point on, a link from the other agent to $self$ will exist. The process terminates when achieving quiescence, meaning that a solution has been found, or when the empty nogood is generated, meaning that the problem is unsolvable.

3 The Unifying Kernel

In the following we describe ABT_{kernel}, a generic algorithm for variable-based DisCSPs. This algorithm is sound but it may fail termination. We identify the condition to assure termination.

3.1 The ABT_{kernel} algorithm

The ABT_{kernel} algorithm requires, like *ABT*, that constraints are directed —from the value-sending agent to the constraint-evaluating agent— forming a directed acyclic graph. Agents are ordered statically in agreement with constraint orientation. Agent i has higher priority than agent j if i appears before j in the total ordering. Considering a generic agent $self$, $\Gamma^-(self)$ is the set of agents constrained with $self$ appearing above it in the ordering. Conversely, $\Gamma^+(self)$ is the set of agents constrained with $self$ appearing below it in the ordering.

A *directed nogood* for value c of variable x_k is $x_i = a \wedge x_j = b \wedge \ldots \Rightarrow x_k \neq c$, meaning that the assignment of c to x_k is inconsistent with the assignments of $a, b, \ldots$ to $x_i, x_j, \ldots$. This nogood is a justification of c removal, as long as values $a, b, \ldots$ are assigned to variables $x_i, x_j, \ldots$. Its left-hand (`lhs`) and right-hand sides (`rhs`) are defined from the position of $\Rightarrow$. Nogoods can have been received from lower priority agents, or derived from constraints with higher priority agents. ABT_{kernel} takes the following options on nogoods,

1. *One nogood per removed value.* Each agent keeps only one nogood per removed value. This option, also taken in some version of *ABT*, assures a polynomial space complexity.

2. *Nogood resolution.* When every value of a variable x_k is ruled out a nogood, these nogoods are resolved computing a new nogood $newNogood$ as follows. Let x_j be the closest variable (in the total order) to x_k in the left-hand side of the nogoods, with value b. `lhs`($newNogood$) is the conjunction of the left-hand sides of all nogoods for values of x_k removing x_j. `rhs`($newNogood$) is $x_j \neq b$. $newNogood$ is sent to x_j. Agent k removes from its nogood store nogoods with x_j in their left-hand side.

Each agent keeps its agent view and a nogood store, which must be consistent. The agent view of $self$ is the set of values it believes are assigned to $\Gamma^-(self)$ agents. Agents exchange assignments and nogoods until a solution is found or inconsistency is detected. A message msg can be of the following types ($sender$ is the sending agent and $self$ is the receiver),

- $Info$: it informs $self$ that $sender$ has done a new assignment $msg.Assig$.
- $Back$: it informs $self$ that $sender$ has found a nogood $msg.Nogood$ as a cause of inconsistency requiring $self$ not to take `rhs`($msg.Nogood$).
- $Stop$: it informs $self$ that no solution exists and causes it to stop.

ABT_{kernel} appears in Figure 1. In the main procedure `ABT`$_{kernel}$, each agent selects a value and informs other agents (`CheckAgentView` call, line 2). Then, a loop receives and processes messages (lines 3-8).

$Info$ messages are processed by `ProcessInfo` and they are always accepted. After receiving an $Info$ message, the agent view of $self$ is updated to include the new assignment, and any nogood inconsistent with the agent view is removed (`Update` call, line 1). Then, a consistent value for $self$ is searched after the change in the agent view (`CheckAgentView` call, line 2). `CheckAgentView` checks if the current value of $self$ is still consistent (line 1). If not, it tries to select a consistent value (`ChooseValue` call, line 2). In this process, some values of $self$ may appear as inconsistent. The nogoods justifying their removal are added to the nogood store (line 3 of `ChooseValue`). If a new consistent value is found, this new assignment is notified to all agents in $\Gamma^+(self)$ through $Info$ messages (line 3). Otherwise, $self$ has to backtrack (`Backtrack` call, line 4). `Backtrack` generates a new nogood by the resolution of existing nogoods for the values of $self$ (line 1). If the new nogood is empty, a $Stop$ message is sent to the agent $system$ and the process stops (lines 2-3). Otherwise, the new nogood is sent in a $Back$ message to the agent appearing in its `rhs` (line 5). The value of this agent is deleted from the agent view (`Update` call, line 6), and a new consistent value is selected (`CheckAgentView` call, line 7).

$Back$ messages are processed by `ResolveConflict`. A $Back$ message coming from $sender$ is accepted if its nogood has the same assignments as $\Gamma^-(self) \cup \{self\}$ (line 1). In this case, the assignments in the nogood for variables not directly related with $self$ are taken to update the agent view (`Update` call, line 2). The nogood is stored, acting as justification for removing the current value of $self$ (line 3). A new consistent value for $self$ is searched (`CheckAgentView` call, line 4). If the message is not accepted, it is obsolete. Then, if the value of $self$ was correct in the received nogood, $self$ resends its value to $sender$ by an $Info$ message (line 5), because $sender$ has forgotten $self$ value when sending the $Back$ message (line 6 of `Backtrack`). If not, $self$ does nothing because there is an $Info$ message travelling from $self$ towards $sender$ that has not arrived yet.

A $Stop$ message means that the empty nogood has been derived, so the problem has no solution and the process has to stop.

Eventually, the system can stabilize in a state where each agent has a value and no constraint is violated. This state is a global solution and the network has reached *quiescence*, meaning that no message is travelling through it. Such a state can be detected using specialized snapshot algorithms [3]. If no solution exists, the empty nogood will be generated.

3.2 Formal Properties

ABT_{kernel} has the following formal properties.

Proposition 1 *ABT_{kernel} is sound.*

procedure $\texttt{ABT}_{kernel}\texttt{()}$
1 $myValue \leftarrow$ empty; $end \leftarrow$ false;
2 `CheckAgentView`();
3 **while** $(\neg end)$ **do**
4 $msg \leftarrow$ `getMsg`();
5 **switch**$(msg.type)$
6 $Info$: `ProcessInfo`(msg);
7 $Back$: `ResolveConflict`(msg);
8 $Stop$: $end \leftarrow$ true;

procedure `CheckAgentView`(msg)
1 **if** $\neg$`consistent`$(myValue, myAgentView)$ **then**
2 $myValue \leftarrow$ `ChooseValue()`;
3 **if** $(myValue)$ **then for each** $child \in \Gamma^+(self)$ **do** `sendMsg`:$Info(child, myValue)$;
4 **else** `Backtrack`();

procedure `ProcessInfo`(msg)
1 `Update`$(myAgentView, msg.Assig)$;
2 `CheckAgentView`();

procedure `ResolveConflict`(msg)
1 **if** `Coherent`$(msg.Nogood, \Gamma^-(self) \cup \{self\})$ **then**
2 **for each** $assig \in$ `lhs`$(msg.Nogood) \setminus \Gamma^-(self)$ **do** `Update`$(myAgentView, assig)$;
3 `add`$(msg.Nogood, myNogoodStore)$; $myValue \leftarrow$ empty;
4 `CheckAgentView`();
5 **else if** $msg.sender \in \Gamma^+(self) \wedge$ `Coherent`$(msg.Nogood, self)$ **then** `SendMsg`:$Info(msg.sender, myValue)$;

procedure `Backtrack`()
1 $newNogood \leftarrow$ `solve`$(myNogoodStore)$;
2 **if** $(newNogood =$ empty$)$ **then**
3 $end \leftarrow$ true; `sendMsg`:$Stop(system)$;
4 **else**
5 `sendMsg`:$Back(newNogood)$;
6 `Update`$(myAgentView,$`rhs`$(newNogood) \leftarrow$ unknown);
7 `CheckAgentView`();

function `ChooseValue()`
1 **for each** $v \in D(self)$ not eliminated by $myNogoodStore$ **do**
2 **if** `consistent`$(v, myAgentView)$ **then return** (v);
3 **else** `add`$(x_j = val_j \Rightarrow self \neq v, myNogoodStore)$; /*$v$ is inconsistent with x_j's value */
4 **return** (empty);

procedure `Update`$(myAgentView, newAssig)$
1 `add`$(newAssig, myAgentView)$;
2 **for each** $ng \in myNogoodStore$ **do**
3 **if** $\neg$`Coherent`$($`lhs`$(ng), myAgentView)$ **then** `remove`$(ng, myNogoodStore)$;

function `Coherent`$(nogood, agents)$
1 **for each** $var \in nogood \cup agents$ **do**
2 **if** $nogood[var] \neq myAgentView[var]$ **then return** false;
3 **return** true;

Figure 1: The ABT_{kernel} algorithm for asynchronous backtracking search.

Proof. If a solution is claimed, we have to prove that all agents satisfy their constraints. Let us assume quiescence in the network. If the current assignments are not a solution, there exists at least one violated constraint, i.e., an agent still unsatisfied with its current assignment. In this case, at least one message has been sent from the unsatisfied agent to the nearest culprit. This message is either not obsolete, in which case the recipient will change its value and break our quiescence assumption by sending a message, or obsolete, which means that some other message has not yet reached its destination and again breaks our assumption. □

Proposition 2 *ABT_{kernel} cannot infer inconsistency if a solution exists.*

Proof. Every nogood resulting from an $Info$ message is redundant with regard to the DisCSP to solve. Since all additional nogoods are generated by logical inference when a domain wipe-out occurs, the empty nogood cannot be inferred if the network is satisfiable. □

In spite of these good properties, ABT_{kernel} may fail to terminate. The problem lies in the obsolescence of nogoods. The way nogoods are generated guarantees that every variable appearing in the nogood is above $self$ in the ordering. But nothing ensures that those variables are in $\Gamma^-(self)$. This leads us to the following observation.

Lemma 1 *ABT_{kernel} may store obsolete information.*

Proof. Since a nogood may contain an unrelated agent u above $self$ in the ordering, it cannot be locally checked for obsolescence as u will not send its new value to $self$. Thus, an agent can end up storing indefinitely an information which is no longer updated. □

Worse, the agent may use that information to prune a value. If there is a solution including this value, it will be missed. Since ABT_{kernel} cannot infer inconsistency if a solution exists, it will not terminate.

Lemma 2 *Storing obsolete information, ABT_{kernel} agents may fall into an infinite loop*

Proof. Let i be an agent keeping a nogood about an unrelated agent u above i in the ordering, i.e. $x_u = a \Rightarrow x_i \neq c$. Suppose this nogood is now obsolete since x_u changed its value, and c is the only value of x_i in a solution. x_i will try all other values in its domain, find them unfeasible and generate a backtrack message. When this message will reach u, it will be discarded as obsolete, and i will continue looping on the same subdomain, sending backtrack messages which are doomed to be dropped by u. The solution will never be detected. □

Proposition 3 *ABT_{kernel} may fail to terminate.*

Proof. The proof flows naturally from lemma 1 and 2. □

If we eliminate obsolete information in finite time, it means that crucial values will not stay deleted forever. At least some of the backtrack messages will be processed, and will thus delete a value on some agent above $self$ in the ordering.

Lemma 3 *The first agent in the ordering can never fall into an infinite loop.*

Proof. Every variable in a nogood received by $self$ is above $self$ in the ordering. If agent 1 receives a nogood, it has an empty left-hand side. So it will never become obsolete. □

Lemma 4 *If the first $k-1$ agents in the ordering are not trapped in an infinite loop and obsolete information disappears in finite time, agent k cannot fall into an infinite loop.*

Proof. Suppose agent k is looping. Since we assume that no obsolete information can last forever, some of the backtracks sent by k will be seen as relevant, and will lead to value deletions. Since no agent among $1, \ldots, k-1$ is supposed to be in an infinite loop, they can accept only a finite number of relevant backtrack messages. Thus, they will either stabilize, in which case k will exit its so-called infinite loop as soon as the obsolete data are deleted, or generate an empty nogood, which will also stop the entire system. So, k is not in an infinite loop. □

Proposition 4 *Removing obsolete information in finite time, ABT_{kernel} will terminate.*

Proof. By recurrence, lemma 3 and lemma 4 show that none of our agents can fall into an infinite loop. So ABT_{kernel} terminates if obsolete information is erased in finite time. □

Therefore, complete algorithms based on ABT_{kernel} should be able to discard obsolete nogoods. If a nogood becomes obsolete, it may survive in the network for a limited period of time.

4 The ABT Family

In the following, we explore ways to remove obsolete information from ABT_{kernel} in finite time, producing several sound and complete algorithms. This allows us to rediscover already existing algorithms, like ABT [16] or $DIBT$ [6], derived from ABT_{kernel} in a clean and elegant form.

A first way to remove obsolete information is to add new communication links to allow a nogood owner to determine whether this nogood is obsolete or not. An added link from agent i to agent j can be seen as the universal constraint between x_i and x_j, permitting all value tuples. x_i should be included in $\Gamma^-(x_j)$ and x_j in $\Gamma^+(x_i)$, which implies that x_j will be informed of the value changes of x_i. These added links were proposed in the original ABT algorithm [16].

A second way to remove obsolete information is to detect when a nogood could become obsolete. In that case, the hypothetically obsolete nogood and the values of unrelated agents are forgotten. These two alternative ways lead to the following four algorithms,

- *Adding links as preprocessing: ABT_{all}.* This algorithm adds all the potentially useful new links during a preprocessing phase. New links are permanent.
- *Adding links during search: ABT.* This algorithm adds new links between agents during search. A link is requested by $self$ when it receives a $Back$ message containing unrelated agents above $self$ in the ordering. New links are permanent.
- *Adding temporary links: ABT_{temp}.* This algorithm adds new links between agents during search, as ABT. The difference is that new links are temporary. This idea has been informally proposed in [13]. A new link remains until a fixed number of messages have been exchanged through it. After that, it is removed.
- *No links: ABT_{not}.* No new links are added between agents. To achieve completeness, this algorithm has to remove obsolete information in finite time. To do so, when an agent backtracks it forgets all nogoods that hypothetically could become obsolete.

In the following we present each of these algorithms in some detail.

4.1 ABT_{all}: Adding links as preprocessing

In a preprocessing phase, ABT_{all} adds a permanent link between every pair of unrelated agents i and j such that x_j may receive a nogood mentioning x_i during the execution of ABT_{kernel}. This

is done adding exactly the same links as in the computation of the induced constraint graph from the initial ordered constraint graph [4]. These new links are computed as follows. Agents (graph nodes) are processed from last to first, along the total ordering of agents. When an agent is processed, all its parents (related agents before it in the ordering) are connected by new links if they were not connected before. These new links are directed, following the total ordering of agents. The structure of the induced graph is recorded in the sets Γ^- and Γ^+ of each agent.

During the search phase, ABT_{all} behaves exactly like ABT_{kernel}, which is now a complete algorithm because each agent is directly connected with every other agent that could appear in a nogood contained in a *Back* message. Obsolete nogoods will be removed in finite time, so ABT_{all} is a sound and complete algorithm that terminates with a correct answer.

Interestingly, it is possible to modify ABT_{all} in such way that agents do not store nogoods anymore, by fixing the agent to backtrack to the closest agent in $\Gamma^-(self)$. A somewhat erroneous form of this algorithm was published in [6] as the $DIBT$ algorithm.

4.2 ABT: Adding links during search

Instead of linking all possible sources of conflict beforehand, we can wait until the conflict actually happens, and add a link at that point. The original ABT takes this approach.

ABT uses a fourth type of message, *AddL*, to request the addition of a new communication link. Each time an agent j receives information about a higher priority agent i previously unheard of, an *AddL* message is sent. As a result, x_i extends its Γ^+ to include x_j, and sends its current value on the newly created link. This way, each agent storing a nogood is guaranteed to be informed whenever one of the variables in the nogood changes its value.

The ABT algorithm appears in Figure 2, only for those parts that differ from ABT_{kernel}. The main procedure `ABT` includes the reception of the *AddL* message (line 9.1), which is processed by `SetLink`. When a link request arrives, the sender is included in $\Gamma^+(self)$ (line 1) and $self$ sends its value through an $Info$ message (line 2). When a *Back* message is received, `ResolveConflict` considers if a request for a new link must be sent (`CheckAddLink` call, line 2.1). Also, the condition for resending $self$ value to senders of obsolete *Back* messages is simplified (line 5.1). `CheckAddLink` checks if unrelated agents appear in the received nogood (lines 1-2). In such case, it sends a request of new link for each unrelated agent, adding it to $\Gamma^-(self)$ (lines 3-4). Finally, it updates its agent view taking as the value of the unrelated agent the value coming in the nogood (line 5). This value will be confirmed or discarded later, when the link request will cause the just related agent to send its value to $self$.

4.3 ABT_{temp}: Adding temporary links

Given that links added in ABT serve the sole purpose of informing $self$ when some nogood becomes obsolete, we may add them during search on a temporary basis. In fact, as soon as $self$ knows the new value for the linked agent, obsolete nogoods are discarded and no further information from that agent is needed at this time, so this additional link could be dropped. It may happen that future *Back* messages will also mention this agent, so the link will have to be established again. If this happens often, it may be more efficient to keep the link active for a number of $Info$ messages, carrying the value changes of the linked agent to $self$.

This is the approach taken by ABT_{temp}. When a new link is set from agent i to j, it is maintained for a fixed number k of $Info$ messages going from x_i to x_j. After this number of messages has been sent, the link is removed and agents i and j become disconnected. The number k of messages for a link is known *a priori* by both agents, so two simple counters —one in each agent— allow for an effective implementation of this technique. When reporting results the number k is essential, and then this algorithm is mentioned as $ABT_{temp}(k)$.

```
procedure ABT()
1 myValue ← empty; end ← false;
2 CheckAgentView();
3 while (¬end) do
4   msg ← getMsg();
5   switch(msg.type)
6     Info   : ProcessInfo(msg);
7     Back   : ResolveConflict(msg);
8     Stop   : end ← true;
9.1   AddL   : SetLink(msg);

procedure ResolveConflict(msg)
1   if Coherent(msg.Nogood, Γ⁻(self) ∪ {self}) then
2.1   CheckAddLink(msg);
3     add(msg.Nogood, myNogoodStore); myValue ← empty;
4     CheckAgentView();
5.1 else if Coherent(msg.Nogood, self) then sendMsg:Info(msg.sender, myValue);

procedure SetLink(msg)
1 add(msg.sender, Γ⁺(self));
2 sendMsg:Info(msg.sender, myValue);

procedure CheckAddLink(msg)
1 for each (var ∈ lhs(msg.Nogood))
2   if (var ∉ Γ⁻(self)) then
3     sendMsg:AddL(var, self);
4     add(var, Γ⁻(self));
5     Update(myAgentView, var ← varValue);
```

Figure 2: The ABT algorithm with permanent links. Only the new or modified parts with respect to ABT_{kernel} in Figure 1 are shown.

4.4 ABT_{not}: No links any more

Instead of trying hard to be informed when an unconnected agent changes its value, $self$ can study its own course of action and update its knowledge accordingly. More precisely, when all values of $self$ have been removed, a new nogood is generated and sent to the nearest culprit. $self$ knows that this nogood will possibly reach every variable it contains, forcing them all, in the worst case, to change their value. For those variables in $\Gamma^-(self)$, there is no need to worry, because they are bound to inform $self$. For the others, the very action of backtracking can lead to the obsolescence of any nogood inside which they appear. Hence, $self$ will forget those insecure variables and nogoods upon backtracking.

There are two cases which deserve some attention. First, it may happen that a forgotten nogood does not become obsolete after all. If $self$ takes the value that this nogood was removing, then $self$ will necessarily receive again this nogood, rediscovered by a lower priority agent.

Second, it may happen that a nogood becomes obsolete because an unrelated, higher priority agent has changed its value and $self$ has not been notified. If the value suppressed by the obsolete nogood is not mandatory to find a solution, this mistake does not compromise finding a solution. On the contrary, if that value is mandatory, $self$ will be forced to try every other value in its domain before backtracking. A new nogood resolving all nogoods removing $self$ values will be produced. This nogood will include the agent that had changed its value, so when sending the $Back$ message, its value will be forgotten and search will be resumed.

The ABT_{not} algorithm takes this approach. This algorithm was described in [1], under the

```
procedure Backtrack()
1 newNogood ← solve(myNogoodStore);
2 if (newNogood = empty) then
3   end ← true; sendMsg:Stop(system);
4 else
5   sendMsg:Back(newNogood);
6   Update(myAgentView,rhs(newNogood) ← unknown);
6.1 for each var ∈ lhs(newNogood) \ Γ⁻(self) do Update(myAgentView, var ← unknown);
7   CheckAgentView();
```

Figure 3: The ABT_{not} algorithm with no links. Only the new or modified parts with respect to ABT_{kernel} in Figure 1 are shown.

name $DisDB$. ABT_{not} only differs from ABT_{kernel} in the forgetting policy of nogoods that could become obsolete, and this concerns the procedure `Backtrack` that appears in Figure 3. This procedure computes the new nogood as the resolvent of the nogoods justifying the wipe out of $self$. If the new nogood is not empty, it is sent in a $Back$ message. Then, $self$ forgets the values of agents not in $\Gamma^-(self)$, and the nogoods including those agents (line 6.1). Finally, a new value consistent with the agent view is searched.

4.5 Discussion

Consider two agents i and j (i preceding j in the ordering) not originally constrained but connected in the induced constraint graph. The algorithms above differ in the way information flows between these two agents. If i takes a new assignment, we say that j is informed about this new assignment when j knows it. The cost of informing j is the minimum number of messages required since i takes the new assignment until j is aware of it. We will say that algorithm A is *better informed* than algorithm B if, for the same problem and the same agent ordering, the cost of informing j of i changes using A is less than or equal to the cost of informing j using B.

Using this definition, we can order the ABT family algorithms following the quality of the information they handle, from ABT_{all} to ABT_{not}. ABT_{all} is better informed than ABT because both behave the same except when ABT detects a conflict between i and j for the first time. In this case, ABT requires more messages to set up the link. ABT is better informed than ABT_{temp} since the latter can require some extra messages to set up again a suspended temporary link. And ABT_{temp} is better informed than ABT_{not} because the former could inform j in one or two messages, while the latter always requires at least two messages since j can be aware of i's value only through a $Back$ message.

5 Experimental Results

We have tested the ABT family algorithms in an asynchronous, single-processor environment. All agents are represented by Linux processes running on the same machine with the same scheduling priority.

We provide results on the search effort, counting the number of "concurrent constraint checks" (*#c-ccks*), as defined[2] in [9], following Lamport's logic clocks [8]. Informally, the number of concurrent constraint checks approximates the longest sequence of constraint checks not performed concurrently. We prefer this parameter to the total number of constraint checks,

[2]Except that in our implementation we do not take into account the cost of messages.

which does not take into account concurrency among agents. Also, we evaluate the global communication effort as the total number of messages exchanged among agents (*#msgs*). We do not report the number of concurrent messages (that were computed following the same technique as #c-ccks) because it was completely proportional to #c-ccks in all our experiments.

We implemented these algorithms considering the following improvements,

1. *Value in AddL.* When a new link with agent k is requested by $self$, instead of sending the $AddL$ message and wait for answer, ABT and ABT_{temp} include in the $AddL$ message the value of x_k recorded in the received nogood. After reception of the $AddL$ message, agent k informs $self$ of its current value only if it is different from the value contained in the $AddL$ message. In this way, some $Info$ messages can be saved.

2. *Avoid resending same values.* ABT family algorithms keep track of the last value taken by $self$. When selecting a new value, if it happens that the new value is the same as the last value, $self$ does not resend it to $\Gamma^+(self)$, because this information is already known. (See line 3 of `CheckAgentView` in Fig. 1.) Again, this may save some $Info$ messages.

3. *Sequence numbers ($ABT_{not}(seq)$).* It is possible to enhance slightly the quality of the information stored by ABT_{not} in the agent view, as follows. Each agent keeps a sequence number, which is incremented each time its value changes. Each time it sends its value, the sequence number is attached. The agent view stores the values and sequence numbers of previous agents in the ordering. When $self$ receives a message, it keeps the newest value for each variable in its agent view. In particular, a $Back$ message is discarded as obsolete if it contains older values than those recorded in $self$'s agent view. When $self$ sends a $Back$ message, the computed nogood contains the values and sequence numbers of involved variables, forgetting the values of unconnected variables but keeping their sequence numbers.

We have decided to keep one nogood per removed value. However, if two nogoods are available for a value, it is possible to select the most appropriate one. With this idea, we propose the following heuristic: when comparing two nogoods, select the one with *the highest possible lowest variable involved*. The rationale for this heuristic is to ensure that each time a wipe-out occurs, the $Back$ message is sent as high as possible in the agent ordering. A similar idea was proposed in [7]. This heuristic is computed as follows. When $self$'s domain is wiped out, each value is checked against each agent in $\Gamma^-(self)$, looking for a nogood better than the current one for this value. In addition, when $self$ receives a $Back$ message with a nogood obsolete on $self$ only, and removing value c, this nogood is still a valid justification for removing c. Then, this nogood is taken if it is better than the current stored nogood justifying c removal.

ABT algorithms, with or without the heuristic of selecting the best nogood, have been tested on two kind of problems, random DisCSPs and distributed meeting scheduling. Their results are reported and discussed in the following.

5.1 Random DisCSPs

Uniform binary random CSPs are characterized by $\langle n, d, p_1, p_2\rangle$ where n is the number of variables, d the number of values per variable, p_1 the network *connectivity* defined as the ratio of existing constraints, and p_2 the constraint *tightness* defined as the ratio of forbidden value pairs.

We have tested random instances of 16 agents and 8 values per agent, considering two connectivity classes, sparse ($p_1 = 0.2$) and medium ($p_1 = 0.5$). Experiments are at the complexity peak considering 50 instances. Specifically, we tested the random classes $\langle 16, 8, 0.2, 0.7\rangle$ (20 solvable instances out of 50) and $\langle 16, 8, 0.5, 0.42\rangle$ (27 solvable instances out of 50). Results appear

$p_1 = 0.20$	#c-ccks	#msgs	$p_1 = 0.50$	#c-ccks	#msgs
ABT_{all}	5,196	8,095	ABT_{all}	39,148	56,206
ABT	5,496	7,675	ABT	40,564	54,694
$ABT_{temp}(10)$	5,530	7,485	$ABT_{temp}(5)$	40,599	50,455
ABT_{not}	35,443	40,223	ABT_{not}	61,658	66,331
$ABT_{not}(seq)$	28,599	33,345	$ABT_{not}(seq)$	59,402	63,451

Table 1: Plain ABTs

$p_1 = 0.20$	#c-ccks	#msgs	$p_1 = 0.50$	#c-ccks	#msgs
ABT_{all}	13,144	7,486	ABT_{all}	101,898	51,891
ABT	13,939	7,470	ABT	102,367	50,558
$ABT_{temp}(10)$	13,889	7,174	$ABT_{temp}(5)$	103,401	48,120
ABT_{not}	75,020	36,454	ABT_{not}	153,320	60,393
$ABT_{not}(seq)$	47,437	27,558	$ABT_{not}(seq)$	128,164	58,907

Table 2: ABTs with nogood selection heuristic

in Tables 1 and 2, where we report the number of concurrent constraint checks and the total number of messages exchanged, averaged over 50 executions.

Table 1 contains the results for the plain ABT algorithms. The parameter k for ABT_{temp} was adjusted manually after some trials. Only the results for the best value of k are given. Considering the three algorithms adding links, ABT_{all}, ABT, and ABT_{temp}, we observe that the better informed the algorithm is, the less concurrent constraint checks it requires to solve the problem. This is at the cost of exchanging more messages. ABT_{temp} is the algorithm exchanging less messages, followed by ABT and ABT_{all}. ABT_{not} requires the highest number of concurrent constraint checks. Because it is the worst informed algorithm, it is more likely to make wrong decisions, requiring more effort than previous algorithms to solve the same problem. This also implies a higher number of messages exchanged. $ABT_{not}(seq)$ dominates ABT_{not} because sequence numbers avoid some of the wrong decisions taken by ABT_{not}.

The effect of using the nogood selection heuristic appears in Table 2. We observe that the number of concurrent constraint checks increases because agents have to do more checks in order to compare nogoods from the store with potentially better nogoods from the constraints. However, the number of messages decreases consistently for all the algorithms, showing the benefits of the heuristic. The relative performance of the algorithms in #msgs remains unchanged with respect to the plain versions.

5.2 Distributed Meeting Scheduling

To compare our algorithms on structured problems, we solved distributed meeting scheduling problems: a number of people with an already partially filled planning, are looking for a place where they can meet at the same time [5]. In our experiment, attendees are divided into three thematic groups. A group, formed by four attendees, has its own meeting to schedule in one of three cities. Two meetings cannot be held at the same time in the same city. Cities are separated by a given travel time. One of the members of the group is in charge of communicating with the other groups.

Each attendee is represented by an agent, with its starting domain matching the attendee's current planning: the predefined appointments (time/place pairs), as well as the time/places which are unreachable because of said appointments, are removed from the domain before the

	$p = 8$		$p = 10$		$p = 12$	
	#c-ccks	#msgs	#c-ccks	#msgs	#c-ccks	#msgs
ABT_{all}	2,482	352	35,937	3,511	2,312	328
ABT	2,513	336	38,439	3,503	2,517	310
$ABT_{temp}(1)$	2,606	220	38,888	3,018	2,873	308
ABT_{not}	3,607	319	47,257	3,793	3,403	375
$ABT_{not}(seq)$	3,417	307	44,017	3,319	3,268	359

Table 3: Plain ABTs

	$p = 8$		$p = 10$		$p = 12$	
	#c-ccks	#msgs	#c-ccks	#msgs	#c-ccks	#msgs
ABT_{all}	3,046	346	55,944	3,451	2,812	322
ABT	3,053	335	59,723	3,438	3,024	303
$ABT_{temp}(1)$	3,119	218	59,892	2,962	3,198	301
ABT_{not}	3,993	317	72,350	3,744	4,013	369
$ABT_{not}(seq)$	3,717	303	65,742	3,492	3,807	354

Table 4: ABTs with nogood selection heuristic

search starts. Our experiment is composed of 5 days, with 6 time slots per day and 3 meeting places. This gives $5 \cdot 6 \cdot 3 = 90$ possible values in the domain of each agent. Meetings and time slots are both one hour long. The 'travel times' between the three cities are 1 hour, 1 hour, and 2 hours. The actual instances are generated by randomly posting p predefined appointments in each agent's planning. We have tested three different classes of problems, with $p = 8$, $p = 10$, and $p = 12$ that correspond respectively to under-constrained, critically constrained, and over-constrained problems. Results appear in Tables 3 and 4, where we report #c-ccks and #msgs averaged over 100 instances.

Table 3 contains the results for plain ABTs. With $p = 8$ (left part of the table), we are at the beginning of the phase transition, where 92% of the instances were satisfiable. On these instances, the best informed algorithms, ABT_{all} and ABT, show the worst performance in number of messages exchanged. The temporary link policy of ABT_{temp} significantly pays off. This can be explained by the fact that these problems are structured as cliques with few constraints outside them. A single nogood between two agents belonging to different cliques leads ABT to the addition of a link that will remain active during the whole search, even if they no longer share nogoods. ABT_{temp} takes advantage of this by activating the link just for solving the current conflict. A confirmation of this is that we observed that $ABT_{temp}(k)$ decays performance as soon as $k > 2$. Even ABT_{not}, with its poorly informed agents, requires less messages than the two best informed algorithms, ABT_{all} and ABT, while on random problems it was always the greatest consumer of messages.

The number of concurrent constraint checks presents the same steady increase from better informed to worse informed algorithms as on random problems, even if the differences are smaller.

Finally, it is worth noting that if we limit the analysis to the 8 inconsistent instances, ABT_{all} and ABT obtain results much closer to ABT_{temp} (232 messages in average for ABT_{all} and 225 for ABT versus 217 for ABT_{temp}). On these inconsistent problems, ABT_{not} is the worst (250 messages for both versions).

Increasing the number of predefined appointments per agent changes the proportion of solvable instances. With $p = 10$ (middle of Table 3), the problems are at the complexity peak (49% of satisfiable instances), and when $p = 12$ (right of Table 3), they are slightly over-constrained (only 12% of satisfiable instances). We observe that as inconsistent instances become more frequent, the average behavior changes. Regarding the number of messages, at the complexity

peak, the benefit of ABT_{temp} with respect to ABT_{all} and ABT decreases, and ABT_{not} becomes worse than ABT. At the right of the complexity peak, differences between ABT and ABT_{temp} are quite small, while differences between ABT_{not} and ABT increase. In this case, the relative results of the different algorithms are very similar to those observed on the 8 inconsistent instances with $p = 8$. The number of concurrent constraint checks reflects again the same trend: the more the problems are constrained, the better informed algorithms behave.

Table 4 shows the effect of the nogood selection heuristic. Regarding the number of messages, it appears that the heuristic is almost useless. One of the reasons is probably that in our implementation, the agents are ordered according to the 4-cliques. Hence, the causes of a conflict are most of the time circumscribed to a clique, which does not give the opportunity to select a nogood jumping much higher than another one chosen arbitrarily. As a consequence, the number of concurrent constraint checks can only increase since the heuristic has a tiny benefit on the search performance while it requires extra constraint checks.

5.3 Discussion

We have tested the ABT family algorithms on unstructured (random) problems as well as structured ones (meeting scheduling). From the results, we observe the following facts.

Regarding the search effort, consistently for all problems, the more informed an algorithm is, the smaller the number of concurrent constraint checks it requires. Regarding the number of messages exchanged, the dynamic links of ABT improve over the static approach of ABT_{all}. Temporary links of ABT_{temp} dominate the permanent link approach of ABT, and this dominance depends on the kind of problems. On unstructured problems, ABT_{temp} improves over ABT by a narrow margin, which becomes larger on structured problems, especially consistent ones. When considering unsatisfiable instances only, both algorithms exhibit a similar performance. ABT_{not}, the algorithm not adding links, is competitive only for structured problems at the left of the complexity peak, showing a slight improvement when sequence numbers are used. This leads us to conclude that when problems do not show some structure, or are not under-constrained, ABT_{not} has to be selected only if some privacy policy justifies its use.

Regarding the nogood selection heuristic, we observe clear benefits on unstructured problems but only minor advantages on structured ones. Apparently, the structure of the meeting scheduling problems prevents the heuristic to find large jumps over the network.

While ABT_{temp} appears as a good algorithm for asynchronous backtracking, the following question remains: how many $Info$ messages to allow through a temporary link? In the reported experiments this parameter was adjusted manually after some trials. We believe that it has not to be a fixed parameter of the algorithm, but it could be adjusted automatic and dynamically, customized for each agent. The automatic selection of this parameter is a direction for further research.

6 Conclusion

We have proposed ABT_{not}, a new asynchronous backtracking algorithm for distributed CSPs. This procedure is the first one that does not add links between agents not sharing constraints. This property can be important to avoid messages sent to agents which may not need to be informed. We presented ABT_{not} *via* a basic kernel that is sound but does not guarantee termination. By implementing the condition for termination in this kernel, we obtained existing ABT family algorithms, or new ones, such as ABT_{not}. We compared experimentally several algorithms from the ABT family.

References

[1] C. Bessière, A. Maestre, and P. Meseguer. Distributed dynamic backtracking. *Notes of the IJCAI'01 workshop on Distributed Constraint Reasoning*, pages 9–16, Seattle WA, 2001.

[2] C. Bessière, I. Brito, A. Maestre, and P. Meseguer. The Asynchronous Backtracking Family. Technical Report, LIRMM-CNRS, Montpellier, France, March 2003.

[3] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[4] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[5] E.C. Freuder, M. Minca, and R.J. Wallace. Privacy/efficiency trade-offs in distributed meeting scheduling by constraint-based agents. *Notes of the IJCAI'01 workshop on Distributed Constraint Reasoning*, pages 63–71, Seattle WA, 2001.

[6] Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. *Proc. 13th European Conference on Artificial Intelligence, ECAI'98*, pages 219–223, Brighton, UK, 1998.

[7] K. Hirayama and M. Yokoo. The effect of nogood learning in distributed constraint satisfaction. *Proc. 20th International Conference on Distributed Computing Systems, ICDCS'00*, pages 169–177, 2000.

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

[9] A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. Comparing performance of distributed constraints processing algorithms. *Notes of the AAMAS'02 workshop on Distributed Constraint Reasoning*, pages 86–93, Bologna, Italy, 2002.

[10] M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. *Proc. 17th National Conference on Artificial Intelligence, AAAI'00*, pages 917–922, Austin TX, 2000.

[11] M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Consistency maintenance for ABT. *Proc. 7th International Conference on Principles and Practice of Constraint Programming, CP'01*, pages 271–285, Paphos, Cyprus, 2001.

[12] M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Technical report, EPFL, Lausanne, 2001.

[13] M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Polynomial space and complete multiply asynchronous search with abstractions. *Notes of the IJCAI'01 workshop on Distributed Constraint Reasoning*, Seattle WA, 2001.

[14] M. Yokoo. Personal communication, 2000.

[15] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. *Proc. 12th International Conference on Distributed Computing Systems, ICDCS'92*, pages 614–621, 1992.

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