The Asynchronous Backtracking Family

Christian Bessiere[1] Ismel Brito[2] Arnold Maestre[1] Pedro Meseguer[2]
[1] LIRMM-CNRS, 161 rue Ada, 34392 Montpellier, France
`bessiere@lirmm.fr` `maestre@lirmm.fr`
[2]IIIA-CSIC, Campus UAB, 08193 Bellaterra, Spain
`ismel@iiia.csic.es` `pedro@iiia.csic.es`

TR 03139
March 2003
LIRMM-CNRS

Abstract

In the last years, the AI community has shown an increasing interest in distributed problem solving. In the scope of distributed constraint reasoning, several asynchronous backtracking procedures have been proposed for finding solutions in a constraint network distributed among several computers. They differ in the way they store failing combinations of values (nogoods), and in the way they check the possible obsolescence of these nogoods. In this paper, we propose a unifying framework for asynchronous backtracking search. We discuss the choices that can be made to obtain a correct and complete algorithm. These choices can lead to already known procedures, or to new algorithms. Our framework permits to better understand the basic steps of these procedures, and to highlight their differences and similarities. We present original techniques that can be added to these algorithms to improve their behavior and to better fit the features of distributed networks. Finally, experiments permit to assess the relative performances of the different versions of the algorithms, and to show the benefit of using some of the proposed improvements.

1 Introduction

In the last years, the AI community has shown an increasing interest in distributed problem solving using the agents paradigm. Different parts of the problem are held by different agents, which behave autonomously and collaborate among themselves in order to achieve a global solution. The World Wide Web offers many opportunities to actually solve real problems through agents.

Several works consider constraint satisfaction in a distributed form (see [15] for an introduction). These works are motivated by the existence of naturally distributed constraint problems, for which it is impossible or undesirable to gather the whole problem knowledge into a single agent, and to solve it using centralized algorithms.[1] There are several reasons for that. The cost of collecting all information into a single agent could be taxing. This includes not only communication costs, but also the cost of translating the problem knowledge into a common format, which could be prohibitive for some applications. Furthermore, gathering all information into a single agent implies that this agent knows every detail about the problem, which could be undesirable for security or privacy reasons [17].

[1]By *centralized* we mean single processor, as opposed to *distributed*.

Considering complete algorithms for distributed constraint satisfaction, we have to mention the pioneer work of Yokoo and colleagues, who proposed the asynchronous backtracking (ABT) algorithm [13, 14]. This algorithm assumes a variable-based distributed model, where each problem variable belongs to one agent and constraints are shared between agents. ABT requires a total ordering among agents. When a dead-end is detected, it may require to add communication links between previously non connected agents; then, nogoods are exchanged among connected agents, and stored. This algorithm has been used as a basis from which several extensions were proposed, as for example its adaptation to dynamic agent re-ordering [10] or consistency maintenance [9]. A different approach is the distributed backtracking algorithm ($DIBT$) [4, 5], which performs graph-based backjumping without nogood storage. However, in its formulation given in [4, 5], $DIBT$ is not complete [12, 1]. More recently, a new algorithm, called Asynchronous Aggregation Search (AAS) has been proposed in [8]. It is based on exchange of sets of partial solutions assuming a *constraint-based* distributed model, where each constraint belongs to one agent and variables shared by two constraints not belonging to the same agent are duplicated.

Understanding precisely the behavior of an algorithm is much more difficult in a distributed environment than in a centralized one. In the search for solutions in distributed constraint networks, it is quite difficult to circumscribe what are the necessary/sufficient operations that ensure completeness and correctness of a procedure. In this paper, we propose a unifying framework for asynchronous backtracking search in a variable-based distributed constraint network. We first present a simple procedure that contains all the basic features of a backtracking algorithm for distributed constraint satisfaction. We characterize why such a simple procedure is not correct. Then, we analyze the changes that have to be made to obtain a correct and complete algorithm. Depending on the way we modify the basic procedure, we obtain already known algorithms (such as Yokoo's ABT or a correct version of $DIBT$), or new ones. We believe this will help to better understand the behavior of asynchronous search, and to emphasize on the differences and similarities between the possible versions. The pseudo-codes we present for the different algorithms are written to be as explicit as possible. We sometimes had to take decisions with respect to existing versions that let some implementation choices to the programmer. We discuss the consequences of these choices with regard to other alternatives. For instance, our algorithms are based on a *one nogood per value* principle. We analyze the effect of the selection of nogoods on the search process, and propose heuristics to select *better* nogoods. Afterwards, we give to the agents the possibility to be less reactive to single messages, which means that we give them the capability to process several messages as a whole instead of processing them one by one. This raises some questions on how to deal with this concurrent, and possibly conflicting information. We propose answers to these issues. Finally, an experimental evaluation is presented. The test-bed is composed of randomly generated problems and distributed meeting scheduling problems. We compare the different versions of asynchronous search, and evaluate the amount of improvement produced by the new heuristics we propose.

The rest of the paper is organized as follows. Section 2 contains some basic definitions. Section 3 recalls the asynchronous backtracking algorithm. In Section 4, we introduce our framework as the kernel for asynchronous backtracking, showing that it is sound but incomplete. In Section 5, we analyze several ways to reach completeness deriving several algorithms, some already known in the literature while others are new. This derivation allows one to understand better the relations between different algorithmic approaches, as well as their mutual dependencies. In Section 6, the nogood selection is discussed, and some heuristics are proposed to decrease the search effort. Section 7 considers the issue of processing packets of messages as a whole, instead of processing them one by one. Section 8 contains the experimental evaluation. Finally, Section 9 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 declaring those value combinations which are acceptable for variables. The CSP involves finding values for the problem variables that satisfy all the constraints, namely, the solutions of the constraint network. We restrict our attention to constraints involving two variables, namely *binary* constraints. A constraint among the variables x_i and x_j will be denoted by c_{ij}.

A distributed CSP (DisCSP) is a CSP where the variables, domains and constraints of the underlying network are distributed among automated agents. Formally, a finite variable-based distributed constraint network is defined by a 5-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ are as before. $\mathcal{A} = \{1, \ldots, p\}$ is a set of p agents, and $\phi : \mathcal{X} \to \mathcal{A}$ 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, $\mathcal{C}_{intra} = \{c_{ij} | \phi(x_i) = \phi(x_j)\}$, and $\mathcal{C}_{inter} = \{c_{ij} | \phi(x_i) \neq \phi(x_j)\}$, 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 the other agents. Usually, it is considered that an inter-agent constraint c_{ij} is known by the agents $\phi(x_i)$ and $\phi(x_j)$ [14, 4].

As in the centralized case, a solution of a DisCSP is an assignment of values to variables satisfying every constraint (although DisCSP literature focuses mainly on solving inter-agent constraints). DisCSPs are solved by the collective and coordinated action of agents $\mathcal{A}$, each holding a process of constraint satisfaction.

Agents communicate by sending messages, with the following assumptions [14],

1. An agent can send a message to other agents iff it knows the addresses of the receivers.

2. 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 purposes, and to emphasize on the distribution aspects, in the rest of the paper we assume that, each agent owns exactly one variable. We identify the agent number with its variable index ($\forall x_i \in \mathcal{X}, \phi(x_i) = i$). From this assumption, all constraints are inter-agent constraints, so $\mathcal{C} = \mathcal{C}_{inter}$ and $\mathcal{C}_{intra} = \emptyset$.

We have to point out here that this definition of DisCSPs fits the one used in *ABT* [14] and *DIBT* [4], but not the one used in *AAS* [8]. In this last case, there are no inter-agent constraints. The way consistency of values is ensured for a variable shared by two constraints not in the same agent is duplication of the variable on these agents. The communication protocol guarantees consistency of the values taken by this variable on each agent (simulating an equality constraint between the two copies of the variable).

3 The Asynchronous Backtracking algorithm

Asynchronous Backtracking (*ABT*) [13, 14, 15, 16] was a pioneer algorithm to solve DisCSP, dating its first version from 1992. *ABT* is an asynchronous algorithm executed autonomously by each agent in the distributed constraint network. Each agent takes its own decisions and informs other agents of 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. In the following, we provide a short description of *ABT*; for further details the interested reader is addressed to the original sources.

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. When the value-sending agent makes an assignment, it informs the constraint-evaluating agent, which tries to find a consistent value. If it cannot, it sends back a message to the value-sending agent to cause backtracking. To make the network cycle-free there is a total order among agents, which is followed by the directed links.

The *ABT* algorithm is executed on each agent, keeping its own agent view and nogood list. 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 list keeps the nogoods received by $self$ as justifications of inconsistent values. Agents exchange assignments and nogoods. *ABT* always accepts new assignments, updating the agent view accordingly. When receiving a nogood, it is accepted if it is consistent with the agent view of $self$, otherwise it is discarded as obsolete. An accepted nogood is used to update the nogood list. When an agent cannot find any value consistent with its agent view, because the original constraints or because the received nogoods, new nogoods are generated from its agent view and sent to the closest agent in the new nogood, causing backtracking. If $self$ receives a nogood including 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.

4 The Unifying Kernel

More than a single algorithm, *ABT* provides an algorithmic schema that can be instantiated in different ways. In the following we describe ABT_{kernel}, a generic search algorithm for variable-based DisCSPs. This algorithm captures the fundamentals of *ABT*, and its subsequent analysis will allow us to identify the main elements of asynchronous backtracking assessing their roles and costs. From it, we will consider the different alternatives to obtain sound and complete algorithms, all sharing this basic kernel.

4.1 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.

Regarding nogood management, generated). a *nogood* for a value c of variable x_k is an expression of the form,

$$x_i = a \wedge x_j = b \wedge \ldots \Rightarrow x_k \neq c$$

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 $x_i, x_j, \ldots$. This is a *directed nogood*. Its left-hand and right-hand sides (abbreviated as `lhs` and `rhs` respectively) are defined from the position of $\Rightarrow$. ABT_{kernel} takes the following options on nogoods,

1. *One nogood per removed value.* At each agent, only one nogood is maintained per removed value. This option, also taken in some versions of the original *ABT*, assures a polynomial complexity in space.

2. *Nogood resolution.* A new nogood is generated from nogood resolution. When all values of the variable x_k are ruled out by some nogood, they are resolved computing a new nogood 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. The left-hand side of the new nogood is the conjunction of the left-hand sides of all nogoods for values of x_k removing variable x_j. The right-hand side of the new nogood is $x_j \neq b$. The new nogood is sent to x_j, removing those nogoods with variable x_j in their left-hand side.

Each agent keeps some amount of local information about the global search, namely an agent view and a nogood store. The agent view is the set of values assigned (from $self's$ point of view) to agents above it in the ordering. The agent view and the nogoods kept in the store need to be consistent. Agents exchange assignments and nogoods, and perform the search by repeating a very simple loop until a solution is found or inconsistency is detected.

ABT_{kernel} allows the following types of messages (where $self$ is the receiver agent),

- $Info(agent, value)$. It informs $self$ that *agent* has taken *value* as its current value; this message is sent by *agent* to the agents in $\Gamma^+(agent)$.

- $Back(agent, nogood)$. It informs $self$ that *agent* has found *nogood* as the cause of inconsistency, and it requires $self$ to change its value. This message is sent when *agent* has no consistent value with its agent view, and $self$ is the closest agent included in *nogood.*

- *Stop.* It informs $self$ that no solution exists and causes it to stop. This message occurs between normal agents and an extra agent called *system.*

The ABT_{kernel} algorithm appears in Figure 1. The main procedure `ABT`$_{kernel}$ is executed on every agent. After initialization (line 1), each agent selects a value and informs other agents of its decision (`CheckAgentView` call, line 2). Then, a loop considers the reception of the three possible message types: $Info$, *Back* and *Stop.*

$Info$ messages are processed by the procedure `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). The procedure `CheckAgentView` checks if the current value of $self$ is still consistent, and in this case it does nothing (line 1). Otherwise, it tries to select a consistent value (`ChooseValue` call, line 2). In this process, some values of $self$ may appear as inconsistent. In this case, the nogoods justifying their removal are added to the nogood store (line 3 of procedure `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, every value of $self$ is forbidden by some nogood and $self$ has to backtrack (`Backtrack` call, line 4). The procedure `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). removed

Back messages are processed by the procedure `ResolveConflict`. A *Back* message coming from *sender* is accepted if its nogood is coherent with (it has the same values as) $\Gamma^-(self) \cup$

```
procedure ABT$_{kernel}$()
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 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 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 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.

$\{self\}$, that is, the part of the agent view directly connected with $self$ (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 considered obsolete. Considering obsolete messages, 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 procedure `Backtrack`). Otherwise, it does nothing because there is an $Info$ message containing the value of $self$, travelling towards *sender* but it 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. If no solution exists, the empty nogood will be generated.

4.2 Formal Properties

ABT_{kernel} is a sound algorithm that cannot infer inconsistency if the problem is solvable. These properties are proved in the following propositions.

Proposition 1 ABT_{kernel} *is sound.*

Proof. This simple kernel is sound, since whenever a solution is claimed, the system is in a stable state. We mean by this that a termination detection algorithm is used at system level to detect quiescence in the network. Hence, all agents must satisfy all their constraints. Let us assume quiescence in the network. If the global state is not a solution, there exists at least one violated constraint, i.e., an agent still unsatisfied with its current state. In this case, at least one message has been sent from the unsatisfied agent to the nearest culprit. We can easily see that 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. We can see every nogood resulting from an $Info$ message as a redundant constraint with regard to the DisCSP to solve: its knowledge is implicitly enclosed in the initial constraint network. Since all additional nogoods are then generated by logical inference when a domain wipe-out occurs, all nogoods that can be generated during search are logical extensions of the initial constraint network. In particular, this means that the empty nogood cannot be inferred if the network is satisfiable. □

In spite of these good properties, we will show that ABT_{kernel} may fail to terminate under some circumstances. The main problem lies in the obsolescence of nogoods. On the one hand, the way the nogoods are generated guarantees that every variable appearing in the nogood is above $self$ in the ordering. On the other hand, nothing ensures that those variables are in $\Gamma^{-}(self)$. (If they were in $\Gamma^{-}(self)$, each obsolete nogood would be discarded as soon as $self$ would receive a new value for one of the variables involved.) This leads us to the following observation.

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

Proof. Since a nogood may contain an unrelated agent x_u above $self$ in the ordering, it cannot be locally checked for obsolescence as x_u will not send its new value to $self$. Thus, an agent can end up storing indefinitely an information which no longer accurately describes the so-called global state of the system. □

Worse, the agent will be using that information to prune a value in its domain. If this value is part of a solution, this solution will be missed. Since we know that the algorithm cannot infer inconsistency if a solution exists, it will then fail to terminate.

Lemma 2 *In the presence of obsolete information, some ABT_{kernel} agents may fall into an infinite loop*

Proof. To see how this may happen, let us consider an agent x_i keeping a nogood about an unrelated agent x_u above x_i in the ordering, for example $x_u = a \Rightarrow x_i \neq c$. Let us now imagine that this nogood has become obsolete since x_u changed its value. Finally, let us assume that c is the only value in x_i domain belonging to a solution. x_i will then try all other values in its domain, find them unfeasible (locally or from the agents beneath), and generate a backtrack message. When this message will reach x_u, it will be discarded for it is now obsolete, and x_i will continue looping on the same inconsistent subdomain, sending backtrack messages which are doomed to be dropped by x_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 however we have some way of eliminating obsolete information in finite time, the problem becomes completely different, because 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. We will first discuss the behaviour of the first agent in the ordering, then use our findings to qualify the behaviour of a larger set of agents.

Lemma 3 *Let x_1 be the variable of the first agent in the distributed agent ordering. x_1 can never fall into an infinite loop.*

Proof. We have seen that whenever agent $self$ receives newer information about an agent x_i inside a backtrack message (which will forcibly discard all those nogoods in $self$ inconsistent with the new value for x_i), x_i precedes $self$ in the ordering. In particular, whenever x_1 receives a nogood ng, it has an empty left side, so ng will not make any existing nogood obsolete, nor will it ever be removed by any later one. □

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, x_k cannot fall into an infinite loop.*

Proof. Let us suppose x_k is looping. Since we assume that no obsolete information can last forever, some of the backtracks sent by x_k will indeed be seen as relevant, and will lead to value deletions. Since no agent among $x_1, \dots x_{k-1}$ 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 x_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, x_k is not in an infinite loop. □

Proposition 4 *Any instance of ABT_{kernel} in which obsolete information is eliminated in finite time will eventually terminate.*

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

Therefore, complete algorithms based on ABT_{kernel} should be able to discard obsolete nogoods, no matter which variables they include. In fact, one has to make sure that a given information may survive in the network for only a limited period of time after it has become obsolete. We will now describe such strategies.

5 The ABT Family

In the following, we explore different techniques to remove obsolete information from the ABT_{kernel} in finite time, producing several sound and complete algorithms. We globally name these algorithms *the ABT family*, because its close relation with ABT. This process allow us to rediscover already existing algorithms, like ABT [14], $DIBT$ [4] or $DisDB$ [1], which are derived from ABT_{kernel} in a clean and elegant form. In this way, a number of algorithms proposed independently in the past can be seen inside an unifying framework, understanding that they share a large common root and clarifying the points in which they differ.

Since the bane of ABT_{kernel} is the lack of information from unrelated agents, a possible solution is to add communication links in the network of agents, in order to give a more accurate view of the global state of the system to the holder of a given nogood, thus allowing it to determine whether or not the nogood holds.

A communication link from agent i to agent j can be seen as the universal constraint between x_i and x_j, which permits all value tuples. In addition, i and j are related, so $x_i \in \Gamma^-(x_j)$ and $x_j \in \Gamma^+(x_i)$, which implies that x_j will be informed of the value changes of x_i. While communication links do not change the solution space for the problem at hand, they allow recipients to accurately discard outdated information. Communication links were proposed [14] as a part of the original ABT algorithm. Here, we consider the different alternatives,

1. *Permanent links.* When a new link is added it is maintained until the end of the execution. Two linked agents remain connected.
2. *Temporary links.* When a new link is added, it is maintained until some condition (typically, on the number of messages sent by this link) is achieved. After that, the link is removed disconnecting the two agents.
3. *No links.* No new links are added between agents.

These three options generate the following four algorithms,

- ABT_{all}. This algorithm adds all the potentially useful links during a preprocessing phase. These new links are permanent.
- ABT. This algorithm adds dynamically links between agents. A link is requested by $self$ when it receives a $Back$ message containing unrelated agents above $self$ in the ordering. New links are permanent. This approach is taken by the original ABT.
- ABT_{temp}. This algorithm add dynamically links between agents, as ABT. The difference with ABT is that new links are temporary. When a new link is set, it remains until a fixed number of messages have been exchanged through this link. After that, it is removed disconnecting both agents.

- 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 it backtracks it forgets all nogoods that hypothetically could become obsolete.

In the following we present each of these algorithms in some detail. Experimental results of these algorithms are presented in Section 8.

5.1 ABT_{all}: Adding links as preprocessing

In a preprocessing phase before search starts, 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 the computation of the induced constraint graph from the initial constraint graph [2]. 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 (graph node) 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 [4] as the *DiBT* algorithm.

ABT_{all} can be downgraded even more, by adding new directed links compatible with the agent ordering such that the constraint graph is now a clique. This is equivalent to include all agents below $self$ in $\Gamma^+(self)$, and all agents above $self$ in $\Gamma^-(self)$. Again, there is no need to store nogoods. A backtrack from $self$ is always directed to the previous agent in the ordering. The agent view is attached to allow the recipient to choose whether he should discard its current value or not. This algorithm acts like a distributed asynchronous version of the well known chronological backtracking, the agent ordering being used instead of the order of instantiation.

5.2 ABT: Adding links during search

Instead of linking all possible sources of conflict beforehand, cluttering the communication graph and incurring the corresponding cost in terms of message passing, we can wait until the conflict is actually detected during search, and dynamically add a link at that point. This is the approach taken by the original ABT algorithm, that we rename here as ABT for homogeneity purposes.

ABT has been proven correct and complete, and 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 the `SetLink` procedure. 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,

```
procedure ABT()
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;
9.1   $AddL$   : SetLink($msg$);
procedure ResolveConflict($msg$)
1   if Coherent($msg.Nogood, \Gamma^-(self) \cup \{self\}$) then
2.1   CheckAddLink($msg$);
3     add($msg.Nogood, myNogoodStore$); $myValue \leftarrow$ empty;
4     CheckAgentView();
5.1 else if Coherent($msg.Nogood, self$) then sendMsg:$Info(msg.sender, myValue)$;
procedure SetLink($msg$)
1 add($msg.sender, \Gamma^+(self)$);
2 sendMsg:$Info(msg.sender, myValue)$;
procedure CheckAddLink($msg$)
1 for each ($var \in$ lhs($msg.Nogood$))
2   if ($var \notin \Gamma^-(self)$) then
3     sendMsg:$AddL(var, self)$;
4     add($var, \Gamma^-(self)$);
5     Update($myAgentView, var \leftarrow varValue$);
```

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

procedure `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). Procedure `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 $self$.

5.3 ABT_{temp}: Adding temporary links

Given that links used in ABT serve the sole purpose of informing $self$ when some of its nogoods become obsolete, we might want to add them dynamically, but 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 then 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 happen very often, it will be cost-effective to keep the link active for a number of $Info$ messages, carrying the value changes of the connected agent to $self$.

This is the approach taken by the ABT_{temp} algorithm, which requests links dynamically, exactly like ABT. 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

```
procedure ABT$_{temp}$()
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;
9.1   $AddL$  : SetLink($msg$);
procedure ProcessInfo($msg$)
1   Update($myAgentView, msg.Assig$);
1.1 if istemporarylink($msg.sender$)
1.2   $counter[msg.sender] \leftarrow counter[msg.sender] - 1$;
1.3   if $counter[msg.sender] = 0$ then
1.4     remove($msg.sender, \Gamma^-(self)$);
1.5     Update($myAgentView, msg.sender \leftarrow$ unknown);
2   CheckAgentView();
procedure ResolveConflict($msg$)
1   if Coherent($msg.Nogood, \Gamma^-(self) \cup \{self\}$) then
2.1   CheckAddLink($msg$);
3     add($msg.Nogood, myNogoodStore$); $myValue \leftarrow$ empty;
4     CheckAgentView();
5.1 else if Coherent($msg.Nogood, self$) then SendInfo($msg.sender, myValue$);
procedure CheckAgentView(msg)
1 if $\neg$consistent($myValue, myAgentView$) then
2   $myValue \leftarrow$ ChooseValue();
3.1 if ($myValue$) then for each $child \in \Gamma^+(self)$ do SendInfo($child, myValue$);
4 else Backtrack();
procedure SetLink($msg$)
1   add($msg.sender, \Gamma^+(self)$);
1.1 $counter[msg.sender] \leftarrow$ maxInfo;
2.1 SendInfo($msg.sender, myValue$);
procedure CheckAddLink($msg$)
1 for each ($var \in$ lhs($msg.Nogood$))
2   if ($var \notin \Gamma^-(self)$) then
3     sendMsg:$AddL(var, self)$;
4     add($var, \Gamma^-(self)$);
4.1   $counter[var] \leftarrow$ maxInfo;
5     Update($myAgentView, var \leftarrow varValue$);
procedure SendInfo($agent, myValue$)
1 sendMsg:$Info(agent, myValue)$;
2 if istemporarylink($agent$) then
3   $counter[agent] \leftarrow counter[agent] - 1$;
4   if ($counter[agent] \leq 0$) then remove($agent, \Gamma^+(self)$);
```

Figure 3: The ABT_{temp} algorithm with temporary links. Only the new or modified parts with respect to ABT_{kernel} in Figure 1 are shown.

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 $ABT_{temp}(k)$.

The ABT_{temp} algorithm appears in Figure 3, only for those parts that differ from ABT_{kernel}. It is very close to ABT, their differences come from the management of temporary links. The main procedure `ABT`$_{temp}$ is the same as `ABT` (Figure 2). Procedure `SetLink` processes the reception of an *AddL* message, initializing the counter of $Info$ messages that can be sent through this link (line 1.1) and sending the first one (line 2.1). Procedure `CheckAddLink` detects, as in ABT, if a new link is needed, and in this case initializes the counter of $Info$ messages that are expected through this new link (line 4.1). Procedure `ProcessInfo` processes the reception of an $Info$ message. If it comes through a temporary link, the corresponding counter is decremented (line 1.2), and if it has reached zero the link is removed, forgetting the value of the connected agent (lines 1.3-1.5). Procedure `SendInfo` encapsulates the sending of $Info$ messages, decrementing the counter if the used link is temporary, and if the counter has reached zero, the link is disconnected (lines 2-4). Finally, procedures `ResolveConflict` and `CheckAgentView` behave like in ABT but the `sendMsg` call has been substituted by the `SendInfo` call.

5.4 ABT_{not}: No links any more

The problem of obsolete nogoods can be tackled the other way around: instead of trying hard to be informed when the status of one of its conflicting ancestors evolves, the agent responsible for the storage of a particular nogood can study its own course of action over the value of the knowledge it holds, and update this knowledge accordingly. More precisely, whenever a nogood store is solved to generate a new nogood to be sent to the nearest culprit, the sender $self$ knows that this nogood will possibly reach each and every node it contains, forcing them all, in the worst case, to change their value. For those nodes appearing in $\Gamma^{-}(self)$, there is no need to worry, because they are bound to inform it. For all the other ones, the very action of initiating the backtrack can lead to the obsolescence of any nogood inside which they appear. Hence, $self$ will forget those insecure nogoods upon backtracking, eliminating the risk of keeping an obsolete nogood in memory.

Since the agent forgets these insecure nogoods only upon backtracking, what does it happen if a nogood becomes obsolete because an unrelated, higher priority agent has changed its value for any reason and $self$ has not been notified? In this case there are two possibilities: either the value suppressed by the obsolete nogood is not mandatory to find a solution, or it is. In the first situation, and in the worst case, $self$ will keep the obsolete nogood as if it were up-to-date, until the end of the search, which will end in a finite amount of time since we assumed that this mistake does not compromise the network capacity to find a solution. On the contrary, if that value is mandatory, $self$ will be forced to try every other value in its domain before backtracking. At this point, 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.

This is the approach taken by the ABT_{not} algorithm, which does not add any link. This algorithm was described in [1], under the name *DisDB*. We call it here ABT_{not} to follow our scheme. The ABT_{not} algorithm 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 4. 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, the *Back* message is built and sent. In this case, $self$ forgets the values of agents not in $\Gamma^{-}(self)$, and consequently the nogoods including those

```
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);
6.1 for each $var \in$ lhs($newNogood$) $\setminus \Gamma^{-}(self)$ do Update($myAgentView, var \leftarrow$ unknown);
7   CheckAgentView();
```

Figure 4: The ABT_{not} algorithm with no links. Only the new or modified parts with respect to ABT_{kernel} in Figure 1 are shown.

agents (line 6.1). Finally, a new value consistent with the agent view is searched.

5.5 Complexity Analysis

5.5.1 Number of links

We presented instantiations of the ABT_{kernel} that differ in the technique used to check nogoods for obsolescence. Some of them ABT_{all}, ABT and ABT_{temp}, put additional links between non connected agents, while ABT_{not} uses a *forgetting* policy on the nogoods that permits termination without adding links. In the former case, these links are added before starting search by ABT_{all}, or on the fly when necessary (ABT, ABT_{temp}). Finally, a link is added permanently in ABT_{all}, ABT, while it has a temporary life in ABT_{temp}.

The question that arises is: how many such links can be added by ABT or ABT_{temp} during the search for a solution? Or by ABT_{all} before the search? The actual number will obviously depend on the instance to be solved, and on the way wipes out occur. However, we can give an upper-bound to the worst-case behavior.

When a wipe out occurs on an agent i, the agent i builds a nogood by resolution of its nogood store, and sends the obtained nogood to the agent j with the lowest priority in this set. When agent j receives the nogood, it checks the compatibility of the nogood with its own agent view. But, since this nogood can contain variables, say x_k, unknown for agent j (because there is a constraint between x_i and x_k but not between x_j and x_k), agent j will ask the agents k containing such a variable, to add a link from k to j. In the worst-case, a wipe out occurring at agent i will generate a nogood involving the whole set $\Gamma^{-}(i)$ of the agents linked to i, and preceding i in the agent ordering. Thus, the receiver of the nogood, say agent j, will have to add a link from each agent in $\Gamma^{-}(i) \setminus \Gamma^{-}(j)$ to itself. More generally, when traveling back to all the ascendent agents, this nogood (or its consequences) can lead to the addition of links between each pair of agents in $\Gamma^{-}(i)$, leading to a total number of links equal to $|\Gamma^{-}(i)| \cdot (|\Gamma^{-}(i)| + 1)/2$ in the set $\Gamma^{-}(i) \cup \{i\}$. Doing that recursively from the last agent to the top agent ($\Gamma^{-}(j)$ has increased after the reception of a nogood from i), we build exactly the induced graph (the same graph as Adaptive-consistency does in its first phase [2]). This means that given the total ordering o on the agents used by the algorithm, if $W(o)$ is the width of the ordering o, the number of links that can finally be added is $n \cdot W(o) \cdot (W(o) + 1)/2 - |\mathcal{C}|$, where n is the number of variables, and $W(o) \cdot (W(o) + 1)/2$ is the number of possible links inside a set formed by an agent and its parents at the end of the search. This depends on the induced width of the network and on the quality of the initial ordering o.

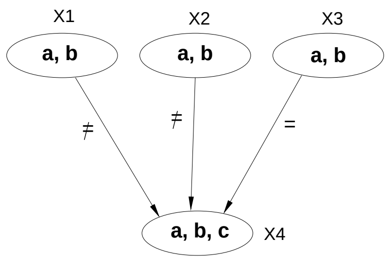

Figure 5: A sample network

5.5.2 Resolving several nogoods

When presenting the ABT_{kernel} algorithm, we decided to store only one nogood per removed value. Hence, when a wipe out occurs on a variable x_i, only one nogood can be built by resolution of the selected nogoods of the values of x_i. However, we could take into account several/all the nogoods explaining the deletion of a value.[2] In this case, when a wipe out occurs on a variable, these nogoods could be resolved to produce several new nogoods, which are all valid. The following example illustrates this fact.

Example 1 Suppose we have a network with four agents A_1, A_2, A_3, A_4, owning variables x_1, x_2, x_3, x_4 respectively. $D(x_1) = D(x_2) = D(x_3) = \{a, b\}$, and $D(x_4) = \{a, b, c\}$. x_4 is linked to x_1 and x_2 by the $=$ constraint, while x_4 is linked to x_3 by the $\neq$ constraint. This network appears in Figure 5. Agents are ordered lexicographically. If the variables x_1, x_2, x_3 have all taken value a and sent $Info$ messages to x_4, a wipe out occurs in $D(x_4)$ after receiving the $Info$ from x_3. The agent view of x_4 is $\{x_1 = a, x_2 = a, x_3 = a\}$. From it we can build[3] the following nogoods for values in $D(x_4)$:

$$x_1 = a \Rightarrow x_4 \neq a; x_2 = a \Rightarrow x_4 \neq a$$

$$x_3 = a \Rightarrow x_4 \neq b$$

$$x_3 = a \Rightarrow x_4 \neq c$$

A valid resolvent nogood can be built by taking one nogood per value and resolving them. Since we have two nogoods for the deletion of $x_4 = a$, we have several possible resolutions,

$$x_1 = a \wedge x_2 = a \wedge x_3 = a \Rightarrow x_4 \neq a$$

$$x_1 = a \wedge x_3 = a \Rightarrow x_4 \neq a$$

$$x_2 = a \wedge x_3 = a \Rightarrow x_4 \neq a$$

From these resolutions, only the two last are minimal nogoods. We can note that the nogoods explaining the deletion of a value can be nogoods obtained from *Back* messages, even if this case does not appear in our example. $\square$

Let us now characterize the number of resolvents that can be built from a single wipe out when several nogoods are stored for each value. To simplify the analysis, let us suppose that a wipe out occurred on an agent that received only $Info$ messages. The nogoods of its values all

[2]Among the different versions of ABT in the literature, some are written to send several nogoods produced by resolution of the nogoods explaining deletions of values [14, 16], while in others, only one is sent [15].

[3]Depending on the implementation, nogoods can be built each time an $Info$ message arrives, or only when a wipe out occurs. Those coming from *Back* messages are stored anyway.

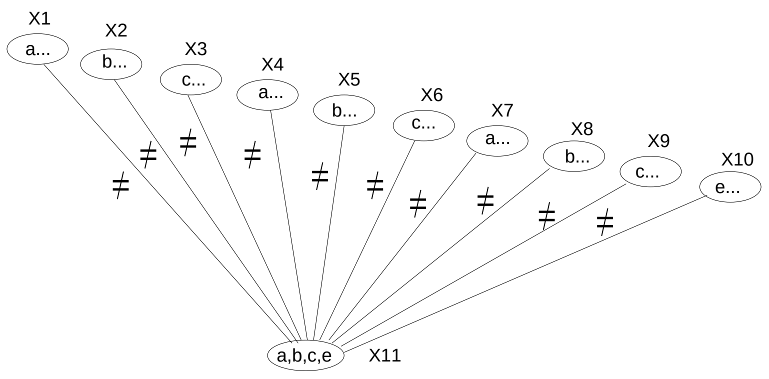

Figure 6: A pathological case for nogood resolution.

have a left hand side of size one. Because an agent has at most $n-1$ parents, the total number of nogoods is at most $(n-1)\cdot d$, which is polynomial. We want to know how many resolvents can be built from these nogoods explaining the value deletions. Let us consider a variable x with $p+1$ parents $x_1,\ldots,x_{p+1}$, and $d+1$ values $\{v_1,\ldots,v_{d+1}\}$ in its domain. Suppose that each of the p first parents of x sent an $Info$ message with a value removing k values of x among its d first values. Each $v_i \in \{v_1,\ldots,v_d\}$ has in average $k\cdot p/d$ nogoods explaining its removal. Now, suppose x receives an $Info$ message from its last parent, x_{p+1}, that removes v_{d+1}. Then, we have a wipe out, and resolution will take place. By assumption, all the nogoods will have x_{p+1} as a left hand side. If the $k\cdot p$ nogoods coming from $x_1,\ldots,x_p$ are equally distributed among the d first values of x, each of these d first values of x has at most $\lceil k\cdot p/d\rceil$ nogoods. Thus, by making all the possible combinations, we obtain a total number of resolvents bounded above by:

$$\lceil k\cdot p/d\rceil^d \tag{1}$$

This is an upper bound on the number of resolvents. Now, we are interested in the number of *minimal* nogoods, and not in the total number of nogoods. Indeed, if $x_1=a\wedge x_2=a\wedge x_3=a\Rightarrow x_4\neq a$ and $x_1=a\wedge x_3=a\Rightarrow x_4\neq a$ are two possible nogoods (see Example 1), only the latter needs to be sent since it subsumes the second one. We show in Example 2 that the upper bound (1) can be reached by the number of minimal nogoods. Hence, if x is x_n (the last variable in the ordering), if it shares constraints with all the other variables, and if its domain has a size d, the total number of minimal nogoods obtained after a wipe out, and having x_{n-1} as a right hand side, is $\Omega((k\cdot n/d)^d)$. This means that if the policy is to store several nogoods per value, and to send all the minimal resolvents, a single agent can send to the same parent an exponential number of *Back* messages, all related to the same wipe out.

Example 2 Suppose we have a network with 11 agents, where $x_1,\ldots,x_{10}$ are all linked to x_{11} with an inequality constraint. (See Fig. 6.) x_1,x_2,x_3 are assigned value a and send $Info$ messages, x_4,x_5,x_6 are assigned value b and send $Info$ messages, x_7,x_8,x_9 are assigned value c and send $Info$ messages, and finally, x_{10} is assigned value d and sends $Info$ message. The domain of x_{11} was $\{a,b,c,d\}$. Once x_{11} has received the $Info$ message sent by x_{10}, we have a wipe out on x_{11} with the following stored nogoods:

$$x_1=a\Rightarrow x_{11}\neq a; x_4=a\Rightarrow x_{11}\neq a; x_7=a\Rightarrow x_{11}\neq a$$

$$x_2=b\Rightarrow x_{11}\neq b; x_5=b\Rightarrow x_{11}\neq b; x_8=b\Rightarrow x_{11}\neq b$$

$$x_3=c\Rightarrow x_{11}\neq c; x_6=c\Rightarrow x_{11}\neq c; x_9=c\Rightarrow x_{11}\neq c$$

```
procedure Update($myAgentView, newAssig$)
1   add($newAssig, myAgentView$);
2   for each $ng \in myNogoodStore$ do
3     if $\neg$Coherent(lhs($ng$), $myAgentView$) then remove($ng, myNogoodStore$);
4.1 for each $v \in D(self)$
5.1   if $\neg$consistent($v, newAssig$) then selectNogood($v, newAssig \Rightarrow self \neq v$);

procedure SelectNogood($deletedValue, newNogood$)
1 if heuristic($newNogood$) > heuristic($myNogoodStore[deletedValue]$) then
2   delete($myNogoodStore[deletedValue], myNogoodStore$);
3   add($newNogood, myNogoodStore$);

function ChooseValue()
1.1 if $\exists v \in D(self)$ not eliminated by $myNogoodStore$ then return ($v$);
4.1 else return (empty);
```

Figure 7: $Info$ Nogood selection, eager way

$$x_{10} = d \Rightarrow x_{11} \neq e$$

In this example, each $Info$ message sent by the $p = 9$ first parents removes exactly $k = 1$ value among the $d = 3$ first values in $D(x_{11})$. By formula 1, we compute a bound of $(1 \cdot (10 - 1)/3)^3 = 27$ nogoods. In fact, if we resolve all the justifications of deletion, we obtain exactly 27 minimal left hand sides for 27 minimal nogoods having $x_{10} \neq e$ as a right hand side. □

6 Selecting Nogoods

In ABT_{kernel} we decided to keep one nogood for removed value. However, if several nogoods are available for each value, it may be advisable to choose the most appropriate resolvant in order to speed up search. Unfortunately, in the most general case, selecting the most suitable nogood with respect to one particular criterion (or set thereof) means generating all possible candidates in order to extract the best one, which could be prohibitively expensive (as seen above).

Heuristics are usually used to tackle such issues: a polynomial-time process, although unable to find the best candidate, should help select a worthy candidate in order to make search more accurate. In this case, when comparing two nogoods we have devised the following heuristic: select the nogood 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, thus saving unnecessary search effort. This can be done in linear time in the number of stored nogoods. Additionally, this nogood is generated by resolving the best candidate nogood for each value in the domain, so selection can be performed locally on each value.

In the following, we will discuss two basic ways of implementing this heuristic: the *eager* way and the *lazy* way. In the former, the nogood selection is processed aggressively at two different points in the algorithm, when updating the agent view and when receiving a nogood from a *Back* message. In the latter, nogood selection due to changes in the agent view is postponed until a wipe out occurs on the agent, but is processed as above when a *Back* message is received.

6.1 Eager Selection of Nogoods from the Agent View

This protocol, detailed in Figure 7, tries to make sure that each time the agent view is modified, the nogood store is in its best possible state given the knowledge available.

```
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 */
4.1 for each $v \in D(self)$ do
4.2   for each $x_j \in \Gamma^-(self)$ such that $(x_j, val_j)$ conflicts with $(self, v)$ do
4.3     selectNogood($v, x_j = val_j \Rightarrow self \neq v$);
4.4 return (empty);
```

Figure 8: $Info$ Nogood selection, lazy way

Procedure `Update` processes $Info$ messages. After updating the agent view (line 1) and removing nogoods inconsistent with it (lines 2-3), the eager protocol goes through all values of $self$ domain, checking their consistency with the new assignment (lines 4.1-5.1). If a value is found inconsistent, the corresponding nogood is built and considered for storage (`selectNogood` call, line 5.1). Procedure `selectNogood` compares two nogoods —a stored nogood and a new nogood— that justify the removal of the same value, keeping in the store the nogood with the highest heuristic value (lines 1-3).

Thus, no matter in which order the $Info$ messages were received, all active nogoods refer to the highest possible culprit. This property, after resolution, ensures that the nogood generated is, among all possible nogoods, the one containing the highest possible lowest assignment. As a side effect, the function `ChooseValue` can now select the first value in the domain not eliminated by a nogood (line 1.1), since the whole domain is filtered each time the view is updated.

6.2 Lazy Selection of Nogoods from the Agent View

The zeal of the eager protocol may be time consuming in some occasions, since the whole domain shall be checked again each time a new assignment for a parent is received. Our second implementation tries to delay the burden of selecting the best upstream nogood for each value as long as possible: it only kicks in when a wipe-out is detected, that is, when a `ChooseValue` fails, instead of rushing into action with each `Update`. Hence, the regular procedure `Update` shall be used. Details are shown in Figure 8. Notice that line 4.1 is only reached if no consistent value is found in the domain. At this point, the modified function will select the best candidate nogood by comparing those already stored to all those induced by conflicting parents (line 4.2) for each value in the domain (line 4.1) before a later call to Backtrack resolves those active nogoods into a new nogood.

6.3 Selecting nogoods from *Back* messages

ABT_{kernel} accepts a *Back* message if the incoming nogood is coherent with its whole view, including its own assignment. Once this nogood is stored, the local value it refers to is eliminated, which makes the nogood coherent with the whole agent view except for the local assignment. It is *all-but-self* relevant. If $self$ receives a *Back* message with a nogood coherent with its agent view but not with its own assignment, this nogood is *all-but-self* relevant. In this case, this nogood deserves to be considered because it brings valuable information: it gives a valid reason to discard a value, even though that value may already has been discarded. Thus, the incoming nogood has to be compared against the current nogood for its target value, and replace it if it is better from the heuristic point of view.

Figure 9 shows procedure `ResolveConflict`, the one affected by the selection of nogoods from *Back* messages. The scope of the consistency test is reduced to $\Gamma^-(self)$, for we are willing to process messages that are obsolete with respect to the current value (line 1.1). The agent view is updated with the assignments in the nogood for variables not directly related with $self$ (`Update` call, line 2). The incoming nogood is then compared against the stored one —or eventually none— (`selectNogood` call line 3.1) and replaces it if necessary. If the message was also consistent with the current value, this value is reset (line 4.1) and a new consistent value is searched for $self$ (line 4.2). The remaining part works as in ABT_{kernel} (line 5).

```
procedure ResolveConflict($msg$)
1.1 if Coherent($msg.Nogood, \Gamma^{-}(self)$) then
2       for each $assig \in$ lhs($msg.Nogood$) $\setminus \Gamma^{-}(self)$ do Update($myAgentView, assig$);
3.1     selectNogood(rhs($msg.Nogood$).$value, msg.Nogood$);
3.2     if rhs($msg.Nogood$).$value = myValue$ then
4.1         $myValue \leftarrow$ empty;
4.2         CheckAgentView();
5   else if $msg.sender \in \Gamma^{+}(self) \wedge$ Coherent($msg.Nogood, self$) then SendMsg:$Info$($msg.sender, myValue$);
```

Figure 9: *Back* nogood selection

7 Processing Messages by Packets

In previous Sections, we have assumed that ABT family algorithms process messages one by one, reacting as soon as a message is received (updating the agent view or storing a nogood, finding another value if needed, sending further messages, etc.). This strategy of *single-message process* may cause to perform some useless work. For instance, consider the reception of an $Info$ message reporting a change of an agent value, inmediately followed by another $Info$ from the same agent. Processing the first message causes some work that becomes useless as soon as the second message arrives. More complex examples (involving $Info$ and *Back* messages) can be devised, causing to waste substantial effort.

To prevent this kind of useless work, we consider an alternative strategy. Instead of reacting after every single message that is received, the algorithm reads all messages that are in the input buffer and stores them in internal data structures. Then, the algorithm processes all read messages as a whole, ignoring those messages that become obsolete by the presence of another message. We call this strategy *processing messages by packets*, where a packet is the set of messages that are read from the input buffer until it becomes empty. This idea was somehow mentioned in [14], although it was not completely developed.

Fortunately, this idea is easy to implemented. Instead of the single-message reading/processing cycle of ABT_{kernel}, the main loop consists of reading a packet and processing it. Reading a packet requires three lists to store the incoming messages, the *$Info$-List*, *Back-List* and the *AddL-List*, where each list stores the messages of the corresponding type, following the reception order (obviously, ABT_{all} and ABT_{not} do not need $AddL - List$). Processing a packet involves the following steps.

1. *$Info$-List*. First, the *$Info$-List* is processed. For each sender agent, all $Info$ messages but the last are ignored. The remaining $Info$ messages update $self$ agent view, removing nogoods if needed (following procedure `Update`).

2. *Back-List*. Second, the *Back-List* is processed. For those messages containing the correct current value of $self$, the sender is recorded in *RemainderSet*. Obsolete *Back* messages are ignored. $self$ stores nogoods of no obsolete messages, and it sends *AddL* messages to unrelated agents appearing in those nogoods.

3. *AddL-List*. Third, the *AddL-List* is processed as in the single-message case, updating $\Gamma^+(self)$ without sending the $Info$ message.

4. Consistent value. Fourth, $self$ current value is checked for consistency with the agent view, looking for another value if the current is inconsistent. If a wipe-out happens in this process, the corresponding *Back* message is sent, and a consistent value is searched.

5. $Info$ sent. Fifth, if $self$ value has changed, $Info$ messages containing $self$ current value are sent to all agents in $\Gamma^+(self)$. if $self$ value has not changed, $Info$ messages containing $self$ current value are sent to the following agents,

   (a) agents in *RemainderSet*,

   (b) senders of *AddL* messages.

   The three lists become empty.

As in the single-message case, the loop ends when receiving a *Stop* message or an empty nogood is derived.

8 Experimental Results

We have tested the four *ABT* algorithms in a simulated environment under the GNU/Linux operating system. Each agent is a separate process, and the *system* agent is another process. All these processes run on the same machine and have the same priority. The simulation environment is really asynchronous: the standard Linux dispatcher is in charge of the asynchronous activation of each process, independently of each agent activity. The goal was to simulate the conditions of a real network under a controlled environment. Agents communicate exchanging messages via named pipes. The *system* agent is responsible for detecting quiescence. In order to make results reproducible, we executed the algorithms one by one on a CPU with no other load.

We provide results on the search effort, counting the number of "concurrent constraint checks" (*#c-ccks*), as defined[4] in [7], following Lamport's logic clocks [6]. 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, 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 the *ABT* family 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, it does not resend it to $\Gamma^+(self)$, because this information is already known. Again, this may save some $Info$ messages.

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

$p_1 = 0.20$	#c-ccks	#msgs.	$p_1 = 0.50$	#c-ccks	#msgs.
ABT_{all}	5,365	8,318	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

Table 1: Plain *ABT*s

$p_1 = 0.20$	#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
$ABT_{temp}(10)$	13,659	7,134	$ABT_{temp}(7)$	106,880	49,347
ABT_{not}	75,020	36,454	ABT_{not}	153,320	60,393

Table 2: *ABT*s with nogood selection heuristic

Regarding the implementation of the nogood selection heuristic, after some preliminary experiments, we decided to take the lazy selection from the agent view.

The *ABT* family algorithms, including the heuristic of selecting the best nogood and the processing by packets feature, have been tested on two kind of problems, random DisCSPs and distributed meeting scheduling. Their results are discussed in the following.

8.1 Random DisCSPs

We have evaluated the performance of the algorithms on uniform binary random CSP. A binary random CSP class is 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. The constrained variables and the forbidden value pairs are randomly selected [11]. A problem will be referred to as a $\langle n, d, p_1, p_2 \rangle$ network. Each agent is assigned one variable, and the constraints binding is to the neighboring agents.

Using this model, 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 have been performed at the complexity peak, where the differences among algorithms are more explicit, 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 in Tables 1, 2, 3 and 4, where we report the total number of consistency checks, the total number of messages exchanged and the average CPU time per agent. These parameters are averaged over 50 executions.

Table 1 contains the results for the plain ABT algorithms, where messages are processed one by one. 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.

$p_1 = 0.20$	#c-ccks	#msgs.	$p_1 = 0.50$	#c-ccks	#msgs.
ABT_{all}	6,002	7,387	ABT_{all}	47,290	43,513
ABT	5,946	7,110	ABT	52,554	42,925
$ABT_{temp}(10)$	5,598	7,001	$ABT_{temp}(5)$	51,305	41,667
ABT_{not}	26,321	24,673	ABT_{not}	73,701	50,112

Table 3: ABTs processing by packets

$p_1 = 0.20$	#c-ccks	#msgs.	$p_1 = 0.50$	#c-ccks	#msgs.
ABT_{all}	13,475	6,098	ABT_{all}	101,608	37,329
ABT	12,959	5,744	ABT	123,574	37,851
$ABT_{temp}(10)$	12,889	5,607	$ABT_{temp}(7)$	120,621	36,135
ABT_{not}	61,622	23,509	ABT_{not}	174,476	42,784

Table 4: ABTs with nogood selection heuristic and processing by packets

The effect of using the nogood selection heuristic (lazy selection for $Info$ messages) 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.

Processing messages by packets causes significant benefits with respect to the plain versions. Results appear in Table 3. In our implementation, each agent sleeps for 1 second before reading all the messages that are in the buffer. Then, they are treated as explained in Section 7. Every algorithm decreases in the number of exchanged messages. Relative algorithmic performance is maintained with respect to messages exchanged, ABT_{temp} remains the algorithm requiring less messages.

The combination of the nogood selection heuristic and processing messages by packets is beneficial, causing further savings in search effort and network usage. Results appear in Table 4. Relative performance is maintained with respect to the number of exchanged messages.

These experiments confirm the benefits of the proposed nogood selection heuristic and the advantadges of processing messages by packets. Consistently for all algorithms and all problems tested, their inclusion caused to decrease the number of exchanged messages. When combined, their benefits are reinforced, causing savings up to 40% in exchanged messages. Regarding algorithms, consistently for all problems tested ABT_{temp} offers the best performance in number of exchanged messages, followed closely by ABT and ABT_{all}. Therefore, these results suggest that ABT_{temp} is the algorithm of choice for random problems at the complexity peak.

8.2 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 [3]. 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

	$p = 8$		$p = 10$		$p = 12$	
	#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

Table 5: 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

Table 6: ABTs with nogood selection heuristic

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 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 5 and 6, where we report #c-ccks and #msgs averaged over 100 instances.

Table 5 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 5), the problems are at the complexity peak (49% of satisfiable instances), and when $p = 12$ (right of Table 5), 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 6 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.

8.3 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. 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.

9 Conclusion

We have proposed a simple basic procedure for asynchronous backtracking search. This procedure is sound but does not guarantee termination. We have shown some extensions of this basic procedure that handle nogoods and links in a way such that termination is ensured. Some of these procedures were already known, such as Yokoo's well-known ABT. Others are original. They differ only in their extensions with respect to the basic kernel. We believe that this characterization of asynchronous backtracking will help better understand these non trivial mechanisms. On this well-specified basis, we discussed the choices that can be made, and their consequences (such as the addition of links or the nogood storage). We proposed some improvements that fit well the behavior of really distributed networks (such as the possibility of processing messages by packets). Finally, we compared experimentally some of the algorithms and heuristics presented in the paper, and drawn some conclusions on what are the good techniques depending on the type of problem to solve.

References

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

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

[3] E.C. Freuder, M. Minca, and R.J. Wallace. Privacy/efficiency trade-offs in distributed meeting scheduling by constraint-based agents. In M.C. Silaghi, editor, *Proceedings of the workshop on Distributed reasoning*, pages 63–71, Seattle WA, 2001.

[4] Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *Proceedings ECAI'98*, pages 219–223, Brighton, UK, 1998.

[5] Hamadi Y. *Traitement des problèmes de satisfaction de contraintes distribués*. PhD thesis, Universite Montpellier II, July 1999. In French.

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

[7] A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In M. Yokoo, editor, *Proceedings AAMAS'03 workshop on Distributed Constraint Reasoning*, pages 86–93, Bologna, Italy, 2002.

[8] M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proceedings AAAI'00*, pages 917–922, Austin TX, 2000.

[9] M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Consistency maintenance for ABT. In *Proceedings CP'01*, pages 271–285, Paphos, Cyprus, 2001.

[10] 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.

[11] B. Smith. Phase transition and the mushy region in constraint satisfaction problems. In *Proceedings ECAI'94*, pages 100–104, Amsterdam, The Netherlands, 1994.

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

[13] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings ICDCS'92*, pages 614–621, 1992.

[14] 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.

[15] Yokoo M., Ishida T. Search Algorithms for Agents. In *Multiagent Systems*, G. Weiss editor, Springer, 1999.

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

[17] Yokoo M. , Suzuki, Hirayama K. Secure Distributed Constraint Satisfaction: Reaching Agreement without Revealing Private Information. *In Proc. of the 8th CP*, 387–401, 2002.