Distributed Dynamic Backtracking

Christian Bessière
LIRMM-CNRS
161 rue Ada
34392 Montpellier
France
bessiere@lirmm.fr

Arnold Maestre
LIRMM-CNRS
161 rue Ada
34392 Montpellier
France
maestre@lirmm.fr

Pedro Meseguer
IIIA-CSIC
Campus UAB
08193 Bellaterra
Spain
pedro@iiia.csic.es

ABSTRACT

In the scope of distributed constraint reasoning, the main algorithms presented so far have a feature in common: the addition of links between previously unrelated agents, before or during search. This paper presents a new search procedure for finding a solution in a distributed constraint satisfaction problem. This algorithm makes use of some of the good properties of centralised dynamic backtracking. It ensures the completeness of search, and allows a high level of asynchronism by sidestepping the unnecessary addition of links.

1. INTRODUCTION

In the last years, the AI community has shown an increasing interest for 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 [Yokoo and Ishida, 1999] 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, to solve it using centralised 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.

Considering complete algorithms for distributed constraint satisfaction, we have to mention the pioneer work of Yokoo and colleagues, who proposed the asynchronous backtracking (ABT) and the asynchronous weak-commitment search (WCS) algorithms [Yokoo *et al.*, 1992, Yokoo, 1995, Yokoo *et al.*, 1998]. These algorithms require a total ordering among agents, static for ABT and dynamic for WCS (alternative dynamic orderings are investigated in [Armstrong and Durfee, 1997]). When a dead-end is detected, nogoods are exchanged among agents and stored, which may require a substantial amount of extra message passing and extra memory. A different approach is the distributed backtracking algorithm (DIBT) [Hamadi *et al.*, 1998, Hamadi, 1999], which performs graph-based backjumping without nogood exchange. Agents are ordered according to an agent hierarchy and agent identifiers. Unfortunately, this algorithm has been shown incomplete [Yokoo, 2000]. We give a few details in appendix **??**. More recently, a new algorithm, called Asynchronous Aggregation Search (AAS) has been proposed in [Silaghi *et al.*, 2000]. It is based on exchange of sets of partial solutions in a constraint-based distributed model, while the other algorithms were proposed to deal with a variable-based distributed model.

In this paper, we propose a new algorithm for the variable-based distributed model. In designing this algorithm our goal was twofold: first, we tried to remain as general as possible in the communication protocol to allow some flexibility regarding the type of messages exchanged (if some level of privacy is required for example [Meseguer, 2000]). Second, we tried to keep the best of ABT and DIBT to obtain an algorithm as asynchronous as possible, minimising the message passing while remaining complete. From ABT, we take its use of nogoods. From DIBT, we keep the early will to avoid linking unrelated agents. Our algorithm has also some clear relations to Dynamic Backtracking [Ginsberg, 1993].

The paper is organised as follows. In Section 2, we provide some preliminaries for distributed constraint satisfaction. Section 3 presents our new algorithm with its proof of correctness and completeness. In Section **??**, we discuss briefly the main features of the existing distributed search algorithms, with their weaknesses. Finally, Section **??** contains the conclusion.

2. PRELIMINARIES

Classically, constraint satisfaction problems (CSPs) have been defined for a centralised architecture. A finite CSP 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;

[1] By *centralised* we mean single processor, as opposite to *distributed*.

- $\mathcal{D} = \{D_0(x_1), \ldots, D_0(x_n)\}$ is a collection of finite domains; $D_0(x_i)$ is the initial set of possible values for x_i, while $D(x_i)$ is the current set of possible values for x_i;
- $\mathcal{C}$ is a set of constraints among variables. A constraint c_i on the ordered set of variables $var(c_i) = (x_{i_1}, \ldots, x_{i_{r(i)}})$ specifies the relation $rel(c_i)$ of the *allowed* combinations of values for the variables in $var(c_i)$. An element of $rel(c_i)$ is a tuple $(v_{i_1}, \ldots, v_{i_{r(i)}})$, $v_{i_j} \in D_0(x_i)$.

A *solution* of the CSP is an assignment of values to variables which satisfies every constraint. Typically, CSPs are solved by a single, tree-search procedure with backtracking.

A distributed CSP (DisCSP) is a CSP where variables, domains and constraints are distributed among automated agents. Formally, a finite variable-based distributed CSP 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, and

- $\mathcal{A} = \{1, \ldots, p\}$ is a set of p agents,
- $\phi : \mathcal{X} \rightarrow \mathcal{A}$ 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, $\mathcal{C}_{intra} = \{c_i | \forall x_j, x_k \in var(c_i), \phi(x_j) = \phi(x_k)\}$, and $\mathcal{C}_{inter} = \{c_i | \exists x_j, x_k \in var(c_i), \phi(x_j) \neq \phi(x_k)\}$, called intra-agent and inter-agent constraint sets, respectively. An intra-agent constraint c_i is known by the agent owner of $var(c_i)$, and it is unknown by the other agents. Usually, it is considered that an inter-agent constraint c_j is known by every agent that owns a variable of $var(c_j)$ [Yokoo *et al.*, 1998, Hamadi *et al.*, 1998].

As in the centralised case, a solution of a distributed CSP is an assignment of values to variables satisfying every constraint (although distributed CSP literature focuses mainly on solving inter-agent constraints). Distributed CSPs 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 [Yokoo *et al.*, 1998],

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.

Without loss of generality and for simplicity purposes, in the rest of the paper we assume that,

1. 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 $\mathcal{C}_{intra} = \emptyset$.
2. All constraints are binary. A constraint is written C_{ij} to indicate that it binds variables x_i and x_j. From this assumption, we will speak about the *network* of agents when referring to the graph having agents as nodes and inter-agent constraints as edges.

We have to point out here that this definition of distributed CSPs fits the one used in ABT and DIBT, but not the one used in AAS. 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. DISTRIBUTED DYNAMIC BACKTRACKING (DISDB)

3.1. CENTRALISED DB

Dynamic backtracking (DB) [Ginsberg, 1993] is a tree search procedure that keeps for each removed value c of variable x_k a justification of the form, $x_i = a \wedge x_j = b \wedge \ldots \Rightarrow x_k \neq c$, as long as values $a, b, \ldots$ are assigned to variables $x_i, x_j, \ldots$. This justification is called a *directed nogood* where its left-hand and right-hand sides are defined from the position of $\Rightarrow$. Nogoods are maintained in a *nogood store*. DB selects an unassigned variable as current, and tries to assign its values. If a value is inconsistent with a previously assigned variable, that value is discarded and the corresponding nogood is added to the store. When all values of the current variable x_k are ruled out by some nogood, they are resolved computing a new nogood as follows. Let be x_j the chronologically most recent variable 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 added to the store, removing those nogoods with variable x_j in their left-hand side. Variable x_j is unassigned, and the procedure iterates selecting a new variable for assignment. It is proved that DB is complete, and terminates with a correct answer.

3.2. DISTRIBUTED DB

Distributed Dynamic Backtracking (DisDB) performs dynamic jumps over the set of conflicting agents. In the constraint graph, constraints are oriented forming a directed acyclic graph, from which an agent hierarchy may be built using heuristic criteria (max-degree, for instance) following the DisAO ordering scheme [Hamadi *et al.*, 1998]. If no heuristic is used, this hierarchy defaults to lexicographic ordering among agents. Considering a generic agent $self$ in the hierarchy, $\Gamma^-(self)$, it the set of agents constrained with $self$ and appearing at higher levels in the hierarchy. Conversely, $\Gamma^+(self)$ is the set of agents constrained with $self$ appearing at lower levels in the hierarchy. This hierarchy induces a partial order among agents, which should be completed to form the total order $<_o$ (using for instance agents identifiers) to ensure completeness.

The DisDB algorithm is executed on each agent, keeping its own agent view and nogood store. The agent view of $self$ is the set of values that it believes to be assigned to agents before $self$ in the total order. The agent view is always consistent with the nogoods in the store. Agents exchange assignments and nogoods. DisDB always accepts new assignments, updating the agent view accordingly. When receiving a nogood, it is accepted if it is consistent with the agent view in $\Gamma^-(self) \cup \{self\}$, otherwise it is discarded due to obsolescence. An accepted nogood is used to update the agent view of agents not in $\Gamma^-(self)$, and the set of stored nogoods. When all values of an agent are discarded by some nogood, the set of stored nogoods is resolved as in the centralised case, generating a new nogood which is sent to the variable in its right-hand side. This variable plus all variables in the left-hand side of the new nogood that are not in $\Gamma^-(self)$ are unassigned in the agent view, and nogoods are updated accordingly. 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. DisDB uses three types of messages,

1. Stop($system$). No solution exists and the receiver agent stops. It involves an extra agent called $system$, which in turn is in charge of stopping the whole network.

2. Info($child, value$). It informs $child$ that $self$ has taken $value$ as value. It is sent to $\Gamma^+(self)$ agents.

3. Back($nogood$). Backtracking message, containing $nogood$ and addressed to the agent in its right-hand side.

The DisDB algorithm appears in Fig. 1. Messages are exchanged by `getMsg` and `sendMsg` primitives. The main procedure `DisDB` is a receiving loop that switches depending on the type of message received. After receiving the Stop message from the $system$ agent, the procedure terminates.

After an Info message, the procedure `GoAhead` is executed. It updates the agent view, referred as $myContext$ (line 1), and if the new information disables the current value $myValue$ (line 2), a new value is tried (line 3). If such value exists, $\Gamma^+(self)$ is informed of the new value of $self$ (lines 5, 6). Otherwise (i.e., all values of $self$ are discarded), the procedure `ResolveNogoods` is called (line 7). `GoAhead` is also called with a null argument to assign $self$ a value consistent with its agent view (line 2 of `DisDB` and line 10 of `ResolveNogoods`).

A Back message is accepted (line 1 of `Resolve-Conflict`) if it is consistent with the value of $self$ and the agent view for those agents for which $self$ has a direct link to know their values through Info messages. Otherwise, the Back message is discarded by obsolescence. After accepting a Back message, the agent view is updated on those

```
procedure DisDB()
1 compute Γ+(self), Γ−(self);
2 GoAhead(null);
3 end ← false;
4 while (¬end) do
5   msg ← getMsg();
6   switch(msg.type)
7     Stop   : end ← true;
8     Info   : GoAhead(msg);
9     Back   : ResolveConflict(msg);
procedure GoAhead(msg)
1 if (msg) then Update(myContext, msg.Context));
2 if (¬msg or ¬ consistent(myValue, myContext) then
3   myValue ← ChooseValue();
4   if (myValue) then
5     for each child ∈ Γ+(self) do
6       sendMsg:Info(child, myValue);
7   else ResolveNogoods();
procedure ResolveConflict(msg)
1 if consistent(myContext,
                 msg.Context in Γ−(self) ∪ {self}) then
2   Update(myContext, msg.Context);
3   add(msg.Context ⇒ ¬myValue, myNogoods);
4   myValue ← ChooseValue();
5   if (myValue) then
6     for each child ∈ Γ+(self) do
7       sendMsg:Info(child, myValue);
8   else ResolveNogoods();
procedure ResolveNogoods()
1 newNogood ← solve(myNogoods);
2 if (newNogood = empty)
3   end ← true;
4   sendMsg:Stop(system);
5 else
6   sendMsg:Back(newNogood);
7   Update(myContext,rhs(newNogood) ← unknown);
8   for each var ∈ lhs(newNogood) \Γ−(self)
9     Update(myContext, var ← unknown);
10  GoAhead(null);
function ChooseValue()
1 for each v ∈ D0(self) not eliminated by myNogoods do
2   if consistent(v, myContext) then
3     return (v);
4   else                /*X: var inconsistent with v*/
5     add(X = valX ⇒ ¬v, myNogoods);
6 return (empty);
procedure Update(myContext, newContext)
1 include(newContext, myContext);
2 for each ng ∈ myNogoods
3   if (¬ consistent(lhs(ng), newContext)) then
4     myNogoods ← myNogoods \ {ng};
```

Figure 1: The Distributed Dynamic Backtracking algorithm.

agents not in $\Gamma^-(self)$ (line 2), it is stored in the nogood store (line 3) and a new value for $self$ is tried (line 4). If such value exists, $\Gamma^+(self)$ is informed of the new value of $self$ (lines 6, 7). Otherwise (i.e., all values of $self$ are discarded), the procedure `ResolveNogoods` is called (line 8).

Procedure `ResolveNogoods` resolves all nogoods in the store $myNogoods$, generating a new nogood $newNogood$ (line 1). The variable appearing in the right-hand side of $newNogood$ is the variable in $myNogoods$ closer to $self$ in the total order $<_o$ among agents. If the empty nogood is generated, the problem has no solution and the Stop message is sent (lines 3, 4). Otherwise, $newNogood$ is sent (line 6). Then, the variable in the right-hand side of $newNogood$ is unassigned in the agent view (line 7). All variables in the left-hand side of $newNogood$ not in $\Gamma^-(self)$ are unassigned, and the nogoods supported for them are forgotten (lines 8, 9).

Finally, procedure `ChooseValue` selects a value consistent with the agent view, updating the nogood store if needed, and procedure `Update` updates the agent view and keeps the coherence between the agent view and the set of stored nogoods. Procedures `lhs` and `rhs` compute the left-hand and right-hand sides of a nogood, respectively.

3.3. AN EXAMPLE

A simple problem to illustrate DisDB appears in Fig. 2. It contains four variables x_1, x_2, x_3, x_4 ordered lexicographically, their corresponding domains and three constraints $\neq$ connecting the first three variables with x_4. Each variable belongs to a different agent. One possible execution of DisDB appears in Fig. 3 (time increasing downwards), showing for each agent (or equivalently, for its owned variable) and time, its context (as a vertical vector), the set of nogoods stored and the exchanged messages.

Initially, the four variables take the first value in their domains, and the first three inform x_4 of their values. Variable x_4 receives messages from x_1 and x_2, computes two nogoods $x_1 = b \Rightarrow x_4 \neq b$, $x_2 = a \Rightarrow x_4 \neq a$, it resolves them and sends a backtracking message to x_2. Variable x_4 forgets nogoods including x_2 and takes value a.

Next, it receives a message from x_3 which causes a new nogood $x_3 = a \Rightarrow x_4 \neq a$, which is resolved with $x_1 = b \Rightarrow x_4 \neq b$, producing a backtracking message to x_3. Variable x_4 forgets nogoods including x_3 and takes value a.

Variable x_2 receives its nogood but it has no other value available. It computes the new nogood $\Rightarrow x_1 \neq b$, which is sent to x_1. Variable x_2 forgets nogoods including x_1 and takes value a, informing x_4. Variable x_3 receives its nogood and changes its value to b, informing x_4. Variable x_1 receives its nogood and changes its value to a, informing x_4.

Variable x_4 receives messages from x_1, x_2, x_3 (in that order) with their new values. After receiving the new value a of x_1, x_4 forgets its previous nogood $x_1 = b \Rightarrow x_4 \neq b$, and the new nogood $x_1 = a \Rightarrow x_4 \neq a$ is generated. x_4 takes value b. No nogood is generated from the message of x_2. The message of x_3 generates the nogood $x_3 = b \Rightarrow x_4 \neq b$, which is solved with the previous one, causing a backtracking message to x_3. Variable x_4 forgets nogoods including x_3 and takes again value b.

Variable x_3 receives its message and discards previous nogood $x_1 = b \Rightarrow x_3 \neq a$ because it is obsolete: x_1 is not in $\Gamma^-(x_3)$ and in the received nogood x_1 has a different value. Variable x_3 takes value a and informs x_4, which does nothing because the new value is consistent with its context and stored nogoods. The execution ends and a solution has been found.

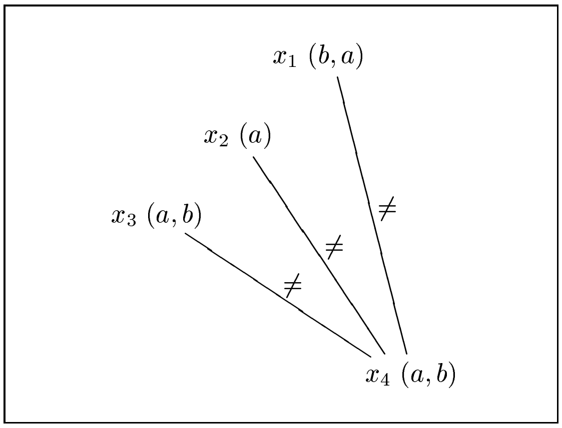

Figure 2: A simple problem.

3.4. CORRECTNESS AND COMPLETENESS

We need to demonstrate that DisDB is sound, complete and that it terminates. Additionally, we will show that its spatial complexity is polynomially bounded.

Theorem 1 (Spatial complexity) *Each agent performing distributed dynamic backtracking requires a polynomially bounded storage space.*

Proof. The space needed in each agent is dominated by the storage requirements of the nogoods. There can be no more than d of these, each one referring in the worst case to n variables. So, the storage requirement in each agent is bounded by $O(nd)$. □

Theorem 2 (Soundness) *DisDB is sound, in that it only claims a solution if one exists.*

Proof. Whenever DisDB detects a solution, all agents are in a stable state, waiting for a message. Such a state is incompatible with constraint violation, which would entail at least one message. □

The following lemma will be used below, because we need some data about which variables can be part of a nogood.

Lemma 1 *Given $<_o$ the total order imposed by DisAO, in every nogood ng of the form ($x_1 = v_1 \wedge x_2 = v_2 \wedge \ldots x_n = v_n \Rightarrow x_j \neq v_j$), we have the following property:*
$\forall x_k$ such that x_k appears in the right side of ng, $x_k <_o x_j$.

Proof. This is true when a nogood is recorded after an Info message, because our distributed agent ordering ensures that for all x_i in $\Gamma^-(j)$, $x_i <_o x_j$. Then, if a nogood is generated due to a domain wipe-out, it will be sent to the conflicting agent x_i which number in the ordering is closest to $self$, thus ensuring that every variable in left side precedes x_i in the sense of $<_o$. □

To demonstrate that DisDB is complete and terminates, we will consider $DisDB_{all}$, an alternate implementation of DisDB with full nogood recording, making sure to enforce only those nogoods relevant in the current context by marking those DisDB would drop as obsolete. We will prove

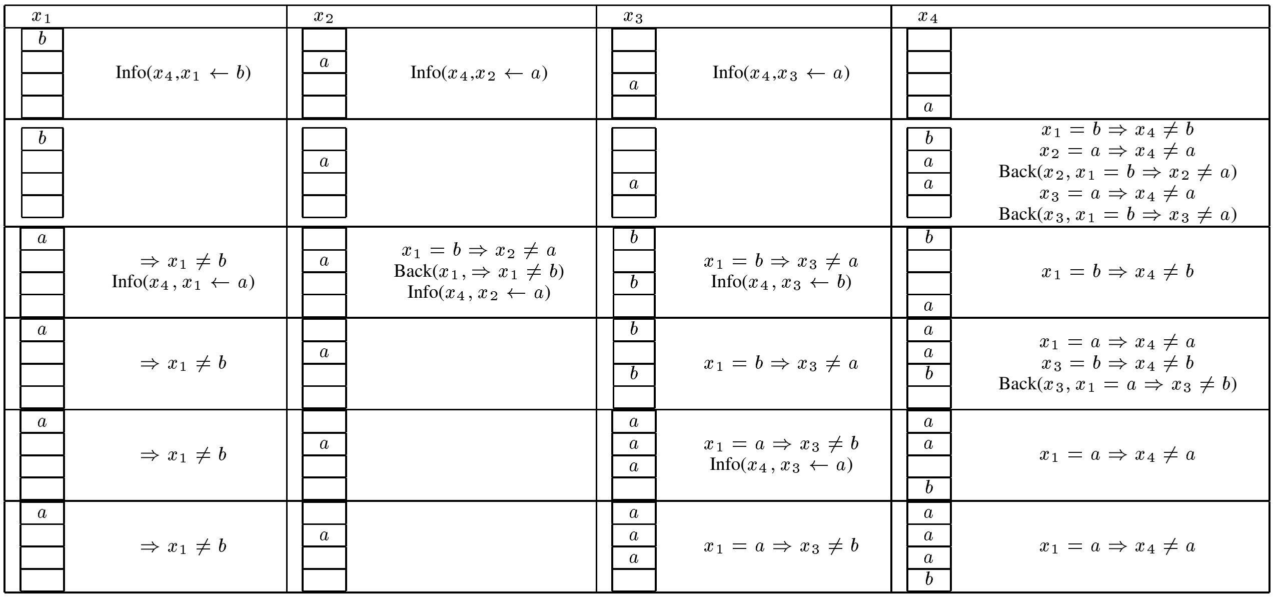

Figure 3: Execution trace of DisDB on the problem of Fig. 2. Time increases downwards.

that it is correct and complete, then we will show that the way DisDB discards obsolete nogoods is safe with respect to those properties.

Theorem 3 *$DisDB_{all}$ is complete and terminates.*

Proof. Soundness is immediately inherited from DisDB: the mechanism ensuring termination detection is the same for both algorithms.

Our argument for completeness is close to the one given in [Silaghi *et al.*, 2000]. Every nogood resulting from an Info message is a redundant constraint with regard to the DisCSP to solve. Since all additional nogoods are generated by logical inference when a domain wipe-out occurs, an empty nogood cannot be inferred when a solution exists.

Furthermore, since the extensive storage of nogoods prunes a monotonically increasing amount of the search space, our algorithm will eventually reach, in a finite amount of time, a state σ where every inconsistent assignment is forbidden. If no solution exists, the empty nogood will eventually be inferred, and the algorithm will terminate.

If a solution exists, however, an obsolete nogood ng_o may prevent one agent in our system to actually take a consistent value. In such case, we can prove that ng_o may only be deemed *viable*[2] for a finite amount of time after σ is reached.

Let x_i be the variable storing ng_o, and x_j the variable involved in ng_o that made it obsolete by changing its value. We know by lemma 1 that x_j precedes x_i in the $<_o$ ordering. If x_j belongs to $\Gamma^-(x_i)$, an inform message is on its way to make ng_o officially obsolete. If not, then provided the value v_i that ng_o forbids is the only assignment for x_i belonging to a solution, x_i will eventually reach a dead-end, send a Back message to the nearest preceding culprit, and mark as obsolete the nogoods involving a conflicting variable not belonging to $\Gamma^-(x_i)$, which include ng_o. Hence the theorem. □

[2]as opposed to *obsolete*.

So far we have shown that our algorithm is sound, complete, and terminates if all nogoods are stored ($DisDB_{all}$). We shall now prove that forgetting some nogoods is safe with respect to those desirable properties. Namely, the way nogoods are discarded shall not cause our algorithm to fall into an infinite loop, nor shall it subtract completeness.

In DisDB, a generic agent $self$ discards an obsolete nogood ng_o when receiving Info or Back messages with incompatible values for some of the variables appearing in the left side of ng_o, and also drops potentially obsolete nogoods when sending a Back message whose left side contains agents not belonging to $\Gamma^-(self)$, rather than merely marking those nogoods as obsolete.

Hence, from lemma 1, the safety of these operations with respect to termination can be induced in a way not unlike that of Yokoo in [Yokoo *et al.*, 1998].

Lemma 2 *Let x_1 be the variable of the agent with lowest index in the distributed agent ordering. x_1 can never fall into an infinite loop, despite the way DisDB discards obsolete nogoods.*

Proof. From lemma 1, we derive 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 DisAO. In particular, whenever x_1 receives a nogood ng, it has an empty left side, so ng won't make any existing nogood obsolete, nor will it ever be removed by any later one. □

Lemma 3 *If the first $k-1$ agents with regard to DisAO are not trapped in an infinite loop, x_k cannot fall into an infinite loop because of the way nogoods are discarded.*

Proof. Let us suppose x_k is actually looping. That means that it keeps forgetting nogoods about its predecessors because they keep changing value. But since we assume that no agent among $x_1, \ldots x_{k-1}$ is in an infinite loop, they will either stabilise, in which case x_k will exit its so-called infinite loop, or generate an empty nogood, which will also stop the entire system. So, x_k is not in an infinite loop. □

Theorem 4 *DisDB is sound, complete, and terminates.*

Proof. By recurrence, lemma 2 and lemma 3 show that none of our agents can fall into an infinite loop, despite the way DisDB discards obsolete nogoods.

Furthermore, discarding potentially obsolete nogoods (or marking them as obsolete) when generating a Back message is not only safe, but it has been shown to be a necessary condition to ensure termination. (See proof of theorem 3.)

Lastly, discarding nogoods is safe with respect to completeness, since if inconsistency cannot be induced from a set of nogoods Ω, it cannot be induced from any subset of Ω. This, along with theorem 2 and theorem 3 concludes our demonstration. □

4. RELATED WORK

When trying to find a solution in a variable-based DisCSP, until now we had the choice between three main algorithms, namely, ABT, AWC, and DIBT.

The main feature of ABT is the way it processes dead-ends to ensure completeness of search. When a dead-end occurs on an agent i (i.e., i cannot take any consistent value for its variable x_i), the agent i builds the set of instantiations that lead to the wipe out of its domain, and sends the "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 view of the other variables. The reason is that the nogood can be based on obsolete information, and then should be discarded. But, since this nogood can contain variables, say x_k, unknown for agent j (because there was a link between x_i and x_k but not between x_j and x_k), agent j will ask to the agents k containing such a variable, to add a link from k to j. This link was not part of the initial network since there were no constraint between x_k and x_j.

The question that arises is: how many such links will be added during the search of a solution? If the actual number will obviously depend on the instance and the way dead-ends occur, we can give an upper-bound to the worst-case behaviour. In fact, the number of links ABT can add during the search for solution can be characterised the same way as the complexity is computed in Adaptative Consistency [Dechter and Pearl, 1998]. Given the total ordering o used by ABT on the agents, if $W(o)$ is the width of the network associated with o, the number of links that can finally be added is $n \cdot W(o) \cdot (W(o) + 1)/2 - |\mathcal{C}|$, where $W(o) \cdot (W(o) + 1)/2$ 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.

Regarding AWC, the ordering of the agents is dynamic, having the possibility to change during search. However, to ensure completeness, nogoods cannot be discarded as it is the case in ABT, and thus AWC has an exponential space complexity.

Finally, DIBT tries to preserve as much as possible the distributed structure of the network. It builds a hierarchy of the agents thanks to the Distributed Agents Ordering (DisAO) procedure, without adding any new link (neither before nor during search). Any heuristics (max-degree for instance) can be used to guide DisAO in building the hierarchy. DIBT does not store nogoods. Unfortunately, it appears that it misses completeness [Yokoo, 2000].

5. CONCLUSION

We have proposed a new search procedure, DisDB, for finding solutions in a distributed constraint satisfaction problem. It is complete (does not lose solutions). It uses the network topology as much as possible, avoiding the definitive addition of communication links between agents not sharing inter-agent constraints. This property is important to ensure as much asynchronism as possible, and to avoid messages sent to agents which may not need to be informed (no common knowledge).

REFERENCES

[Armstrong and Durfee, 1997] A. Armstrong and E. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proceedings IJCAI'97*, Nagoya, Japan, 1997.

[Dechter and Pearl, 1998] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1998.

[Ginsberg, 1993] M.L. Ginsberg. Dynamic bactracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

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

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

[Meseguer, 2000] P. Meseguer. Distributed forward checking. In M.C. Silaghi, C. Bessière, B. Faltings, D. Sam-Haroud, and M. Yokoo, editors, *Proceedings of the CP'00 Workshop on Distributed Constraint Satisfaction Problems*, Singapore, Thailand, 2000.

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

[Yokoo and Ishida, 1999] M. Yokoo and T. Ishida. Search algorithms for agents. In G. Weiss, editor, *Multi-Agent Systems*, pages 165–199. MIT Press, 1999.

[Yokoo *et al.*, 1992] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kubawara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings DCS*, pages 614–621, 1992.

[Yokoo *et al.*, 1998] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kubawara. The distributed constraint satisfaction problem: formalization and algoritms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998.

[Yokoo, 1995] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceedings CP'95*, pages 88–102, Cassis, France, 1995.

[Yokoo, 2000] M. Yokoo. Private communication, 2000.

A. THE FALSE PROMISES OF DIBT

1.1. COMPLETENESS IN QUESTION

The claim in [Hamadi *et al.*, 1998], apart from describing a distributed agent ordering method, was to propose a complete, fully distributed asynchronous backtrack scheme, in which communications always occur between connected variables (no links added) and no learning scheme is involved.

Unfortunately, while this is memory-wise very cost effective, this can lead the algorithm to overlook some possibly valid instantiations. Figure **??** shows a network of five agents, with one variable per agent, running `DiBT`. Links represent inequality constraints, and all variables have the two values $\{a, b\}$ in their initial domain, except x_3, whose initial domain is $\{a\}$.

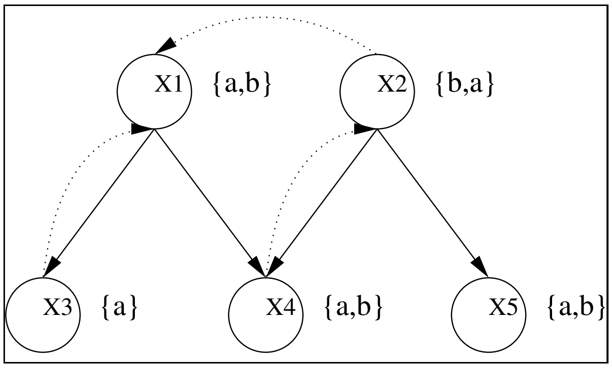

Figure 4: A sample distributed constraints network

`DiBT` could possibly run that way:

1. x_1 and x_2 respectively select a and b, and inform their children.

2. Since x_3 has no other choice than a, it backtracks to x_1, while x_4 backtracks to x_2 due to its parents conflicting values.

3. x_1 and x_2 now respectively select b and a, and inform their children.

4. x_4 backtracks to x_2, whose domain is empty, the backtrack follows up to x_1, which infers inconsistency.

In this case, `DiBT` fails to find a consistent assignment to the network (namely, {b,b,a,a,a}). One reason for this is that x_2, being unaware of x_1's behaviour, does not reset its domain when x_1 changes from a to b, and enforces the suppression of its own value a.

In order to alleviate these difficulties and ensure completeness, [Hamadi, 1999] extends $Parents$ and $Children$ sets to make sure that $Infoval()$ messages will reset all relevant domains and that $btSet()$ messages will be able to backtrack all the way up to the potential culprits, adding beforehand the same links `ABT` ([Yokoo *et al.*, 1992]) would add during search.

1.2. A FLAW IN DIBT'99

But [Yokoo, 2000] pointed out that `DiBT` still isn't complete after those extensions. This is because when checking a nogood for obsolescence, `DiBT` only takes into account its own value, disregarding its $Parents$' values. This can lead to erroneous deductions, as the algorithm merges nogoods from different contexts. An example of such behaviour is illustrated by figure **??**, where links represent inequality constraints.

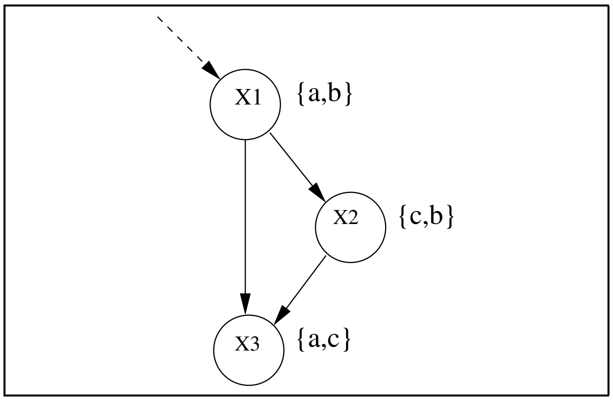

Figure 5: Another sample network

Given a sufficiently ill-favoured timing, `DiBT` could run that way:

1. x_1 selects a, and informs its $children$. x_2 selects c, and informs its $child$. x_3 has an empty domain, and backtracks to x_2 with the nogood $(x_1 = a \wedge x_2 = c)$.

2. Meanwhile, x_1 is forced to change value, due to a message from a $parent$ pruning value a. x_1 selects b and informs its $children$. This message prunes value b in x_2's domain.

3. The backtrack message sent by x_3 at step **??** now reaches x_2. This nogood is obsolete in this agent's context (x_1 is b, not a), but since it is only checked against the local value, it is processed. x_2 is forced to backtrack with the nogood $(x_1 = b)$.

4. x_1's domain is empty, x_1 backtracks. Solutions have been overlooked. The algorithm is not complete.

1.3. PATCHING THE HOLE

This bug could obviously be fixed by checking incoming nogoods against the whole context: each agent should make sure that incoming and outgoing information is consistent with its local view of the system's global state. That way, the nogood sent by x_3 at step **??** would be deemed obsolete and discarded by x_2 at step **??**. Upon receiving pending messages, x_3 would select value a, and the subproblem would stabilise on this partial solution.

While this is by no way a proof for completeness, the faulty behaviour illustrated above is thus fixed, and the rest of the arguments in [Hamadi, 1999] guaranteeing `DiBT`'s completeness should hold.

1.4. CONCLUSION

When presenting DiBT, the authors wanted to offer a complete asynchronous search scheme while avoiding two pitfalls common to the main distributed search algorithms: the addition of links during search, which destroys the initial structure of the problem, and the extensive storage of nogoods generated upon failure, which demands a potentially exponential storage space.

But, as already mentioned, DiBT in its later version actually adds before search the same links ABT or AAS ([Silaghi *et al.*, 2000]) would add during search. This, along with a more careful control of obsolete nogoods, is necessary to ensure completeness.

On the other hand, some of the learning schemes involved in AAS and later versions of ABT yield a polynomial-space nogood store.

The idea of building a complete asynchronous search algorithm with polynomial space learning but without additional links was, from that point on, a natural one. This is the aim of Distributed Dynamic Backtracking.