DISTRIBUTED FORWARD CHECKING *

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

Miguel Angel Jiménez
Cap Gemini España
Diagonal 640
08017 Barcelona, Spain
migjimen@capgemini.es

ABSTRACT

One of the main reasons to distribute constraint satisfaction among agents is privacy: agents may not want to share their values with other agents, and they may wish to keep constraints as private as possible. However, current solving approaches do not satisfy completely these requirements. In this paper, we present the *Distributed Forward Checking* algorithm, a natural successor of Distributed Backtracking, where agent values are kept private. In addition, we introduce the *Partially Known Constraints* model, which allow a constraint between two agents to be not completely known by any of them. With these two elements, we obtain two solving algorithms that respect privacy at most, maintaining completeness. These algorithms have been implemented and experimental results are provided.

1. INTRODUCTION

In the last years, the IA 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 them in order to achieve a global solution. The World Wide Web offers many opportunities to actually solve real problems through agents.

Regarding constraint satisfaction, several approaches have considered it in a distributed form (see [11] for an introduction). These works are motivated by the existence of constraint problems naturally distributed, for which it is impossible or undesirable to gather all the problem knowledge into a single agent, where it would be solved using centralized algorithms.[1] There are several reasons for that. First, the cost of collecting all the information into a single agent could be very high. 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. And second, gathering all the 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 [9, 8, 10]. These algorithms require to have an total ordering among agents, static for ABT and dynamic for WCS (alternative dynamic orderings are investigated in [1]). When a deadend is detected, nogoods are exchanged among agents and stored, which may require a substantial amount of extra memory. A different approach is the distributed backtracking algorithm (DIBT) [4, 5], which performs graph-based backjumping without nogood exchange. Agents are totally ordered, according to an agent hierarchy and agent identifiers.

To enforce privacy, agents should not share their values, and constraints among agents (interagent constraints) should be kept as private as possible. However, these requirements are not totally satisfied by the above mentioned algorithms, which have in common the following points: (i) agents share their values with other agents, and (ii) an interagent constraint is totally known by all agents involved in that constraint[2].

In this paper, we propose a new solving approach trying to satisfy the aforementioned privacy requirements. First, we present a new algorithm, *Distributed Forward Checking* (DIFC), based on DIBT (in the same sense that Forward Checking (FC) [6] is based on Backtracking (BT) [2] in the single processor case). This algorithm is based in the following idea: instead of sending an agent its value to other agents, it sends the effects that its value has on other agents' domains. This idea reproduces FC pruning on future domains, in centralized algorithms. Second, we introduce the *Partially Known Constraints* (PKC) model, in which an actual constraint between two agents is substituted by two new constraints. One of these new constraints is known by one agent and unknown by the other, and the opposite occurs for the other constraint. In this way, the knowledge encoded in problem constraints is kept as private as possible for each agent.

*This research is supported by the Spanish CICYT project TIC99-1086-C03-02.

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

[2]Recently, a new algorithm called AAS has been proposed [7], keeping privacy on interagent constraints.

In this paper, we focus on privacy issues of distributed constraint solving methods, assuming that the distribution of problem elements (variables, domains and constraints) among agents has been done, and agents have agreed on an adequate ordering.

This paper is organized as follows. In Section 2, we provide some preliminaries for distributed constraint satisfaction, including a detailed description of the DIBT algorithm. In Section 3, we discuss privacy issues about values and constraints, proposing strategies for keeping agent values private and introducing the PKC model. In Section 4, we present the DIFC later modified to include the PKC model and we discuss their relative merits with respect to DIBT. In Section 5, we provide some preliminary experimental results of the proposed algorithms. Finally, Section 6 contains the conclusions and perspectives for future work.

2. PRELIMINARIES

2.1. DISTRIBUTED CONSTRAINT SATISFACTION

Classically, constraint satisfaction problems (CSPs) have been defined for a centralized 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;
- $\mathcal{D} = \{D_0(x_1), \ldots, D_0(x_n)\}$ 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 $(v_{i_1}, \ldots, v_{i_{r(i)}})$, $v_i \in D_0(x_i)$.

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 is a CSP where variables, domains and constraints are distributed among automated agents. Formally, a finite distributed CSP is defined by a 5-tupla $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$, 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 intraagent and interagent constraint sets, respectively. An intraagent 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 interagent constraint c_j is known by every agent that owns a variable of $var(c_j)$ [10, 4]. The distribution of interagent constraints among agents is a main topic of this paper, and it is considered in detail in Section 3.2.

As in the centralized 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 interagent 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 [10],

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 interagent 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.

2.2. DISTRIBUTED BACKTRACKING

The *Distributed Backtracking* algorithm (DIBT) [4, 5] performs a graph-based backjumping over the set of problem agents. To assure completeness, it is required a total order among agents. In the constraint graph, constraints are oriented forming a directed acyclic graph, from which an agent hierarchy is built using heuristic criteria (max-degree, for instance). This hierarchy divides agents into levels, satisfying two conditions,

1. There is one single agent at the top of the hierarchy.
2. There is no constraint among agents at the same level.

In some cases, some dummy constraints are added to achieve condition (1) [5]. Given an agent in the hierarchy, its *parents* acquaintances, Γ^-, are those agents constrained with it appearing at higher levels in the hierarchy. Conversely, its *children* acquaintances, Γ^+, are those agents constrained with it appearing at lower levels in the hierarchy. A constraint between agents i and j is checked by the agent with lower level of the two. When unrelated acquaintances are merged, they

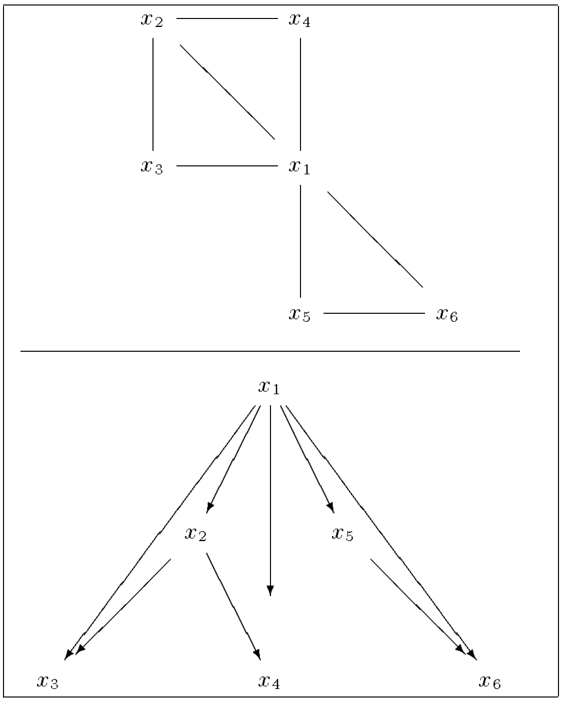

Figure 1: Constraint graph (left) transformed into an agent hierarchy (right) (from [4]).

are ordered using agent identifiers. An example of constraint graph transformed into a constraint hierarchy appears in Fig. 1.

The DIBT algorithm is executed on each agent. Considering a generic agent, $self$, its agent view is the set of values assigned to Γ^- agents. DIBT selects a current value for $self$, and it performs graph-based backtracking when a deadlock occurs. DIBD allows three types of messages,

1. *stop*. It informs that no solution exists, and causes the process of the receiver agent to stop. This message occurs between normal agents and an extra agent called *system*.
2. $infoVal$: $(agent, value)$. It informs that $agent$ has taken $value$ as its current value. This message is sent by $agent$ to its Γ^+ acquaintances.
3. *btSet* : $(agents, values)$. This is a backtracking message, asking the receiver agent to change its current value. The message contents, $agents$ and $values$, are required to perform graph-based backjumping.

DIBT exchanges messages using $getMsg$ and $sendMsg$ primitives. The agent $self$ selects its current value $myValue$ using the function $getValue$ as follows,

- $getValue(info)$. It initializes $D(x_{self}) \leftarrow D_0(x_{self})$. It selects from $D(x_{self})$ a value compatible with the agent view of Γ^-. If $myValue$ exists, it tries $myValue$ first.
- $getValue(bt)$. It updates $D(x_{self}) \leftarrow D(x_{self}) - \{myValue\}$. It selects from $D(x_{self})$ a value compatible with the agent view of Γ^-.

In addition, $first$ returns the closest agent to $self$ in Γ^-, and $merge$ takes two ordered sets of acquaintances and returns their ordered union, according to the agent hierarchy.

The DIBT algorithm appears in Fig. 2. Lines 1-5 contain the initialization where variables take simultaneously their first values. Lines 6-28 contain the main loop (line 6), formed by three possible options after reading a message (line 7),

1. *stop* (line 8): the loop ends, no solution exists.
2. $infoVal$ (lines 9-13): a new assignment arrives from agent j of Γ^-. The view of agent j is updated (line 10), and $self$ tries to get a value compatible with its current agent view (line 11). If it succeeds, it informs to Γ^+ (line 12).[3] Otherwise, it performs backtraking (line 13).
3. *btSet* (lines 14-28): a backtracking message arrives. First, this message is tested for obsolescence, checking if $self$ appears with its current value in the message body (line 15). If not, the message is discarded. If the message is accepted, $self$ tries to get a new value (different from $myValue$) compatible with its current agent view (line 16). If it succeeds, it informs of its new value to Γ^+ (line 17). Otherwise, it checks if the problem has no solution (line 19). If so, the $system$ agent is informed and the loop ends (lines 20-21). Otherwise, the backtracking message is passed to the previous variable in graph-based backjumping (lines 22-24). If that variable does not belong to Γ^-, a new value is selected for $self$, compatible with the agent view of Γ^-, and Γ^+ is informed.

When a solution is found, all agents remain idle waiting for a message. This situation is not detected in the algorithm. Distributed termination detection algorithms [3] could be used.

3. PRIVACY OF VALUES AND CONSTRAINTS

Solving the current model of distributed CSP with the DIBT algorithm, there are two main issues regarding privacy,

1. Values. Using DIBT, agents exchange their values either by $infoVal$ or *btSet* messages.
2. Constraints. An interagent constraint C_{ij} is *totally known* by the agents owning its related variables, that is, C_{ij} is totally known by agents i and j.

[3] If $myValue$ does not change, this message can be avoided.

```
DIBT
compute Γ+, Γ−; order Γ−;
nearest ← first(Γ−);
myValue ← getValue(info);
sendMsg(Γ+, infoVal : (self, myValue));
end ← false;
while (not end) do
    m ← getMsg();
    if (m = stop) end ← true;
    if (m = infoVal : (j, a))
      value[j] ← a;
      myValue ← getValue(info);
      if (myValue) sendMsg(Γ+, infoVal : (self, myValue));
      else         sendMsg(nearest, btSet : (Γ−, value[Γ−]));
    if (m = btSet : (set, values))
      if (values[self] = myValue)
        myValue ← getValue(bt);
        if (myValue) sendMsg(Γ+, infoVal : (self, myValue, ));
        else
          if (Γ− = ∅ ∧ set = ∅)
            sendMsg(system, stop);
            end ← true;
          else
            followSet ← merge(Γ−, set);
            follow ← first(followSet);
            sendMsg(follow, btSet : (followSet, value[Γ−] ∪ values));
            if (follow ∉ Γ−)
              myValue ← getValue(info);
              sendMsg(Γ+, infoVal : (self, myValue));
```

Figure 2: Distributed backtracking algorithm.

This approach may be inappropriate for those applications for which privacy is the main reason to be solved in a distributed form. In that case, agents may desire to hide the actual values of their variables from other agents, considered as potential competitors. For the same reasons, the information contained in the problem constraints may be considered reserved, and agents could not be willing to share it with other agents. In the following, we develop strategies to keep privacy in agent values and in interagent constraints, allowing agents to share enough information to achieve a global consistent solution.

3.1. PRIVACY OF VALUES

The agent $self$ exchanges assigned values with other agents in two cases,

1. When $self$ changes its value. This value is notified to Γ^+ agents in by $infoVal$ messages.

2. When $self$ initiates or propagates a backtracking message. The agent view of Γ^- agents (plus possibly the information included in a previous backtracking message) is sent back to the corresponding agent by a $btSet$ message.

In order to prevent agents from exchanging their values, we present the following strategies for the above mentioned cases,

1. When $self$ changes its value. Instead of sending the new value to Γ^+ agents, $self$ sends the subset of $D_0(x_\gamma)$ of compatible values with its new value to each $\gamma \in \Gamma^+$. This strategy is inspired in the FC algorithm [6], and it forms the basis for the *Distributed Forward Checking* algorithm presented in Section 4.

2. When $self$ initiates or propagates a backtracking message. Each agent keeps the number of times it has changed its value. This number is called the sequence number. A message informing $self$ that an agent has a new value is tagged with the agent sequence number, which is recorded by $self$ in the agent view. When $self$ sends a backtracking message, instead of sending the values of Γ^- agents, it sends the last sequence number recorded for each agent in Γ^-. When a backtracking message is received, if the sequence number contained in it does not match the agent sequence number, the message is discarded by obsolete. In this way, values are

substituted by sequence numbers in backtracking messages, keeping its original functionality (checking if a message is obsolete, line 15, Fig. 2).

The agent view, formerly composed by values of Γ^- agents, is now composed by two elements for each agent $\gamma \in \Gamma^-$: (i) the subset of compatible values of $D_0(x_{self})$ with γ, and (ii) the sequence number of the last message from γ informing that its value has changed.

3.2. PRIVACY OF CONSTRAINTS

In the current model of distributed CSP, it is assumed that an interagent constraint C_{ij} is *totally known* by the agents owning its related variables, that is, C_{ij} is totally known by agent i and agent j (see Section 2.2 of [10] and Section 2.2 of [4]). We say that they follow the *Totally Known Constraints* (TKC) model. In fact, DIBT requires $rel(C_{ij})$ to be known by the agent with lower level of i, j only, which is in charge of checking the constraint.

In this context, we introduce the *Partially Known Constraints* (PKC) model of a distributed CSP in the following terms. An interagent constraint C_{ij} is only partially known by its related agents. From C_{ij}, agent i knows the constraint $C_{i(j)}$ with agent j, defined as follows,

$$var(C_{i(j)}) = \{x_i, (x_j)\} \qquad rel(C_{ij}) \subseteq rel(C_{i(j)})$$

where (x_j) in $var(C_{i(j)})$ means that agent i knows little about the other variable of the constraint. From constraint C_{ij}, agent j knows the constraint $C_{(i)j}$,

$$var(C_{(i)j}) = \{(x_i), x_j\} \qquad rel(C_{ij}) \subseteq rel(C_{(i)j})$$

It is required that,

$$rel(C_{ij}) = rel(C_{i(j)}) \cap rel(C_{(i)j})$$

that is, the intersection of the constraints known by agent i and agent j is the actual constraint existing between i and j.

As example, let us consider a distributed scheduling problem, where task t_i should be scheduled before task t_j in the temporal interval $[0, \ldots, p]$. Tasks t_i and t_j are held by agents i and j respectively. The constraint C_{ij} is defined by,

$$st_i + d_i < st_j \qquad st_j + d_j \leq p$$

where st_i and d_i are the start time and duration of task t_i. If the task duration is a reserved information, only known by the agent owner of the task, the constraint C_{ij} cannot be completely developed by any agent. Instead, agent i can build the constraint $C_{i(j)}$ defined by,

$$st_i + d_i < st_j \qquad st_j < p$$

Analogously, agent j develops the constraint $C_{(i)j}$ as,

$$st_i < st_j \qquad st_j + d_j \leq p$$

The intersection of these two constraints is the initial C_{ij}.

Another example is the *n-pieces m-chessboard* problem. Given a set of n chess pieces and a $m \times m$ chessboard, the goal is to put all pieces in the chessboard in such a way that no piece attacks any other. As distributed CSP, the problem can be formulated as follows,

- Variables: one variable per piece.
- Domains: all variables share the same domain $\{1, \ldots, m^2\}$ of chessboard positions, numbered from left to right and from up to down.
- Constraints: one constraint between every pair of pieces, from chess rules.
- Agents: one agent per variable.

For instance, we can take $n = 5$ with the set of pieces $\{$*queen, castle, bishop, bishop, knight*$\}$, to be located in a 4×4 chessboard, with the variables,

$$x_1 = queen,\ x_2 = castle,\ x_3 = bishop,\ x_4 = bishop,\ x_5 = k$$

With the TKC model, agent 1 knows that agent 5 holds a knight, and agent 5 knows that agent 1 holds a queen, enough information to develop completely constraint C_{15} by any of them,

$$C_{15} = \{(1,8), (1,12), (1,14), (1,15), \ldots\}$$

With the PKC model, agent 1 does not know which piece agent 5 holds. Its only information is about how a queen attacks, from which it can develop the constraint,

$$C_{1(5)} = \{(1,7), (1,8), (1,10), (1,12), \ldots\}$$

Analogously, agent 5 does not know which piece agent 1 holds. Its only information is about how a knight attacks, from which it can develop the constraint,

$$C_{(1)5} = \{(1,2), (1,3), (1,4), (1,5), (1,6), (1,8), \ldots\}$$

The whole constraint C_{15} appears as the intersection of these two constraints,

$$C_{15} = C_{1(5)} \cap C_{(1)5} = \{(1,8), \ldots\}$$

In fact, $C_{1(5)}$ does not depend on 5. It codifies the way a queen attacks, which is independent of any other piece. In this problem, the PCK model allows to represent each agent by itself, independently of other agents. We believe that this model could be useful for spatial or temporal configuration tasks which should be performed in a distributed form.

```
DIFC
1  compute Γ+, Γ−; order Γ−;
2  nearest ← first(Γ−);
3  myValue ← getValueLookahead(info); seq ← 1;
   if (myValue)
4    for γ ∈ Γ+ sendMsg(γ, infoDom : (self, seq, comp(D0(xγ), myValue)));
5    end ← false;
   else
     sendMsg(system, stop);
     end ← true;
6  while (not end) do
7    m ← getMsg();
8    if (m = stop) end ← true;
9    if (m = infoDom : (j, s, Dom))
10     D[j] ← Dom; seq[j] = s;
11     myValue ← getValueLookahead(info); if (myValue) seq ← seq + 1;
12     if (myValue) for γ ∈ Γ+ sendMsg(γ, infoDom : (self, seq, comp(D0(xγ), myValue)));
13     else          sendMsg(nearest, btSet : (Γ−, seq[Γ−]));
14   if (m = btSet : (set, seqValues))
15     if (seqValues[self] = seq)
16       myValue ← getValueLookahead(bt); if (myValue) seq ← seq + 1;
17       if (myValue) for γ ∈ Γ+ sendMsg(γ, infoDom : (self, seq, comp(D0(xγ), myValue)));
18       else
19         if (Γ− = ∅ ∧ set = ∅)
20           sendMsg(system, stop);
21           end ← true;
22         else
23           followSet ← merge(Γ−, set);
24           follow ← first(followSet);
25           sendMsg(follow, btSet : (followSet, seq[Γ−] ∪ seqValues]));
26           if (follow ∉ Γ−)
27             myValue ← getValueLookahead(info); seq ← seq + 1;
28             for γ ∈ Γ+ sendMsg(γ, infoDom : (self, seq, comp(D0(xγ), myValue)));
```

Figure 3: Distributed forward checking algorithm.

4. DISTRIBUTED FORWARD CHECKING

Combining the DIBT algorithm with the strategies for private values of Section 3.1, we present the *Distributed Forward Checking* algorithm (DIFC) under the TKC model. This new algorithm is a direct extension of DIBT. Their main differences are the following,

1. Instead of sending its current value, each agent sends the set of compatible values with its own value to Γ^+ agents. Because of that, DIFC requires $rel(C_{ij})$ to be known by the agent with higher level of i, j only, which is in charge of finding the set of compatible values.

2. Sequence numbers allow implementing graph-based backjumping without sending back current values.

DIFC appears in Fig. 3. Line numbers refer to equivalent lines of DIBT (Fig. 2). DIFC uses three types of messages,

1. *stop*. As in the DIBT algorithm.

2. $infoDom : (agent, seq, compValues)$. This message substitutes $infoVal : (agent, value)$ of DIBT. It informs that $agent$ has a new value which allow $compValues$ as the set of compatible values for the receiver agent. It contains seq, the sequence number of the sender.

3. $btSet : (agents, seqValues)$. This message is equivalent to $btSet : (agents, values)$ of DIBT, where the set $seqValues$ of sequence numbers substitutes the set $values$ of actual values.

The agent $self$ selects its current value $myValue$ using the function $getValueLookahead$, which substitutes $getValue$ of DIBT, as follows,

- $getValueLookahead(info)$. It initializes $D(x_{self}) \leftarrow D_0(x_{self})$. It selects from $D(x_{self})$ a value compatible with the agent view of Γ^- and compatible with at least one value in every domain of Γ^+. If $myValue$ exists, it tries $myValue$ first.

$DIFC - PKC$

```
4   for γ ∈ Γ+ sendMsg(γ, infoDom : (self, seq, myValue, comp(D0(xγ), myValue)));
12     if (myValue)
          for γ ∈ Γ+ sendMsg(γ, infoDom : (self, seq, myValue, comp(D0(xγ), myValue)));
17        if (myValue)
             for γ ∈ Γ+ sendMsg(γ, infoDom : (self, seq, myValue, comp(D0(xγ), myValue)));
28           for γ ∈ Γ+ sendMsg(γ, infoDom : (self, seq, myValue, comp(D0(xγ), myValue)));
```

Figure 4: New lines for the DIFC-PKC algorithm.

- $getValueLookahead(bt)$. It updates $D(x_{self}) \leftarrow D(x_{self}) - \{myValue\}$. It selects from $D(x_{self})$ a value compatible with the agent view of Γ^- and compatible with at least one value in every domain of Γ^+.

In addition, the function $comp(D_0(x_\gamma), myValue)$ computes the subset of values of $D_0(x_\gamma)$ compatible with $myValue$, the current value of x_{self}.

This approach has a main difference with FC in the centralized case: each constraint directed toward $self$ is considered independently by a different agent, so pruning on $D(x_{self})$ does not accumulate through agents in Γ^-. Pruning from different constraints arrives to $self$ by different messages coming from agents in Γ^-. The ability to accumulate pruning on a future domain, an essential feature of FC algorithm to speed up the detection of empty domains, disappears in the distributed case.

4.1. DIFC WITH PARTIALLY KNOWN CONSTRAINTS

Under the PKC model, the DIFC algorithm of Fig. 3 remains essentially valid, but the following change should be made: the $infoDom$ message has to include the current value $myValue$ of the agent sender. Otherwise, the receiver could select an incompatible value. This can be seen with a simple example of the *5-pieces 4-chessboard* problem explained in Section 3.2. If $x_1 \leftarrow 1$, it sends to x_5,

$$comp(D_0(x_5), 1) = \{7, 8, 10, 12, 14, 15\}$$

However, values 7 and 10 are forbidden for the knight (they are not forbidden for other pieces). Therefore, the queen value should be included. The new algorithm, DIFC-PKC, is equal to DIFC where lines of Fig. 3 with an $infoDom$ message should be changed to include $myValue$, as shown in Fig. 4.[4]

In addition, the function $getValueLookahead$ changes its behaviour, to take into account the constraints partially known by the receiver. The new $getValueLookahead$ is defined as follows,

- $getValueLookahead(info)$. It initializes $D(x_{self}) \leftarrow D_0(x_{self})$. It selects from $D(x_{self})$ a value compatible with the agent view of Γ^-, compatible with $\{C_{(\gamma)self} | \gamma \in \Gamma^-\}$, and compatible with at least one value in every domain of Γ^+ according to $\{C_{self(\gamma)} | \gamma \in \Gamma^+\}$. If $myValue$ exists, it tries $myValue$ first.
- $getValueLookahead(bt)$. It updates $D(x_{self}) \leftarrow D(x_{self}) - \{myValue\}$. It selects a value compatible with the agent view of Γ^-, compatible with $\{C_{(\gamma)self} | \gamma \in \Gamma^-\}$, and compatible with at least one value in every domain of Γ^+ according to $\{C_{self(\gamma)} | \gamma \in \Gamma^+\}$.

4.2. DISCUSSION

Regarding privacy, the three algorithms considered in the paper, namely DIBT, DIFC and DIFC-PKC, present a situation depicted in Fig. 5. If agent i is constrained with agent j by C_{ij}, and agent i is at higher level in the hierarchy than agent j, algorithm requirements about privacy of values and constraints are as follows,

- DIBT. Agent i knows its value, $v[i]$, and it may know values of agents above i in the hierarchy, because of backtracking messages. Agent j knows its value, $v[j]$, and the value of i, $v[i]$. Agent j also knows totally the constraint C_{ij} in its relational form, $rel(C_{ij})$.
- DIFC. Each agent knows its value, and no agent knows other agent's value. In addition, agent i knows totally the constraint C_{ij} in its relational form, $rel(C_{ij})$.
- DIFC-PKC. Agent i knows its value, $v[i]$. Agent j knows its value, $v[j]$, and the value of i, $v[i]$. Regarding constraints, each agent knows a different part of C_{ij}, agent i knows $rel(C_{i(j)})$ and agent j knows $rel(C_{(i)j})$.

DIFC achieves a complete value privacy: *no agent knows the value of any other agent.* This represents an significant improvement with respect to DIBT, where values were easily disseminated among agents. In addition, DIFC requires that each constraint should be totally known by the agent of higher level, a requirement equivalent to the one of DIBT (substituting lower level by higher level agent).

[4] There are problems for which the value of the agent sender is not required in the PKC model. For instance, in the task scheduling example of Section 3.2, it is not needed that agent i send its st_i value to agent j; it is enough with the interval $[st_i + d_i, p]$ of compatible values for agent j.

Agents		DIBT	DIFC	DIFC-PKC
$i:$	x_i	$v[i], (v[i-1], \ldots)$	$v[i], rel(C_{ij})$	$v[i], rel(C_{i(j)})$
	$\downarrow C_{ij}$			
$j:$	x_j	$v[i], v[j], rel(C_{ij})$	$v[j]$	$v[i], v[j], rel(C_{(i)j})$

Figure 5: Values and constraints known by agents, for the three considered algorithms.

DIFC-PKC does not achieve a complete value privacy, because an agent will know the values of its Γ^- agents. Nevertheless, DIFC-PKC also improves over DIBT, because it will not send values in backtracking messages. Regarding constraints, DIFC-PKC achieves some level of constraint privacy: each constraint is decomposed in two, one for each agent. Given that the higher level agent i sends its value and the set of compatible values of $D_0(x_j)$ with it, agent j has enough information to try to reconstruct the constraint $C_{i(j)}$ (and therefore, to try to reconstruct C_{ij}). However, this cannot be achieved completely unless agent i tries all its values, which could be considered as an extreme case in distributed constraint satisfaction.

In summary, DIFC provides complete value privacy, while DIFC-PKC achieves some degree in value and constraint privacy. Selecting DIFC or DIFC-PKC as problem solving method will depend on the privacy requirements of the specific problem to solve.

5. EXPERIMENTAL RESULTS

We have tested DIBT, DIFC and DIFC-PKC algorithms on the n-pieces m-chessboard problem. This problem is totally connected; agents (variables) have been ordered in the hierarchy according to the following criterion: a queen precedes a castle, a castle precedes a bishop, a bishop precedes a knight. Values are selected randomly.

We have chosen $n = 9$, with three queens, two castles, two bishops and two knights, to be located on an 8×8 chessboard. We have recorded the sum of constraint checks performed by every agent, and the sum of exchanged messages among agents, averaged over 50 executions.

Results appear in Table 1. Regarding search effort, DIFC performs more constraint checks than DIBT as it could be expected, because DIFC follows an eager strategy testing all values of agents in Γ^+, while DIBT stops testing when it finds a compatible value with agents in Γ^-. DIFC-PKC performs less constraint checks than DIFC, because $self$ lookahead is done for one generic agent, and the result is sent to all agents in Γ^+. DIFC-PKC performs slightly more checks than DIBT. Regarding communication costs, the number of messages exchanged by DIBT, DIFC and DIFC-PKC is similar, although DIFC $infoDom$ messages are larger than DIBT $infoVal$ messages. This result confirms the fact that the three algorithms follow the same search strategy. It is also observed a smooth decrement in the number of messages from DIBT to DIFC-PKC; further research is needed to substantiate this fact.

	DIBT	DIFC	DIFC-PKC
Checks	2,242	14,030	2,547
Messages	124	120	115

Table 1: Results for the 9-pieces 8-chessboard problem.

In summary, DIFC is clearly less efficient than DIBT, while DIFC-PKC efficiency is close to DIBT. They exchange a similar number of messages, although DIFC uses larger messages. Adding lookahead means an extra overhead over DIBT, that is the price that distributed problem solving has to pay for assuring privacy. These preliminary results have to be completed testing no totally connected problems of different sizes, to really assess the magnitude of the extra overhead.

6. CONCLUSIONS

In this paper, we have presented the DIFC algorithm applying the lookahead idea of FC to the distributed case. In contrast with the centralized case, DIFC does not improve backtracking efficiency, since the ability to accumulate pruning on future domains is lost. However, DIFC achieves total value privacy: every agent knows its value and no agent knows other agent's value. This property is very valuable in the distributed case, where privacy concerns play an important role in problem solving. In addition, we have introduced the PKC model, which combined with DIFC allows for a reasonable degree of constraint privacy with less extra search effort.

As conclusion, we confirm the utility of the lookahead idea. Borrowed from the centralized case, where it has been widely and successfully used to improve efficiency, it has also been shown fruitful for the distributed case although providing a different kind of benefits. This result encourages to explore successful ideas of the centralized case, looking for new versions or original combinations of them, which could bring benefits to distributed constraint satisfaction.

As future work, we plan to make an extense experimental evaluation of the presented algorithms, testing several problems of different dimensions. We want to improve the efficiency of the proposed algorithms, considering search effort and communication costs. We plan investigate the applicability of the proposed strategies enforcing privacy to ABT and WCS algorithms. Finally, we would like to test this approach on real life problems for which privacy is a main concern.

REFERENCES

[1] Armstrong A., Durfee E.. Dynamic Priorization of Complex Agents in Distributed Constraint Satisfaction Problems. In *Proc. of the 15th IJCAI*, Nagoya, Japan, 620–625. 1997.

[2] Bitner J. M., Reingold E. M. Backtrack programming techniques *Comm. ACM* **18** (1975) 651–656.

[3] Chandy K., Lamport L. Distributed Snapshots: Determining Global States of Distributed Systems *ACM Trans. Computer Systems*, vol. 3, n. 2, 63–75, 1985.

[4] Hamadi Y., Bessière C., Quinqueton J. Backtracking in Distributed Constraint Networks. In *Proc. of the 13th. ECAI*, 219–223, Brighton, UK, 1998.

[5] Hamadi Y. *Traitement des problemes de satisfaction de constraintes distribues.* PhD thesis, Un. Montpellier II, 1999.

[6] Haralick R., Elliot G. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* **14** (1980) 263–313.

[7] Silaghi M.C., Sam-Haroud D., Faltings B. Asynchronous Search with Aggregations. In *Proc. of the 17th. AAAI*, 917–922, 2000.

[8] Yokoo M. Asynchronous Weak-Commitment Search for Solving distributed Constraint Satisfaction Problems. In *Proc. of the 1st. CP*, 88–102, 1995.

[9] Yokoo M., Durfee E.H., Ishida T., Kuwabara K. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. In *Proc. of the 12th. DCS*, 614–621, 1992.

[10] Yokoo M., Durfee E.H., Ishida T., Kuwabara K. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. Knowledge and Data Engineering* **10** (1998) 673–685.

[11] Yokoo M., Ishida T. Search Algorithms for Agents. *Multiagent Systems*, MIT Press, G. Weiss editor, 165–199, 1999.