Distributed Forward Checking May Lie for Privacy ⋆

Ismel Brito and Pedro Meseguer

IIIA, Institut d'Investigació en Intel.ligència Artificial
CSIC, Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Spain.
{ismel|pedro}@iiia.csic.es

Abstract. *DisFC* is an *ABT*-like algorithm that, instead of sending the value taken by the high priority agent, it sends the domain of the low priority agent that is compatible with that value. With this strategy, plus the use of sequence numbers, some privacy level is achieved. In particular, each agent knows its value in the solution, but ignores the values of the others. However, the idea of sending the whole compatible domain each time an agent changes its value may cause a privacy loss on shared constraints that was initially overlooked. To solve this issue, we propose *DisFC*$_{lies}$, an algorithm that works like *DisFC* but it may lie about the compatible domains of other agents. It requires a single extra condition: if an agent sends a lie, it has to tell the truth in finite time afterwards. We prove that the algorithm is sound, complete and terminates. We provide experimental results on the increment in privacy achieved, at the extra cost of more search.

1 Introduction

In the last years, there is an increasing interest for solving constraint satisfaction problems in a distributed form. This has generated a new model, called *DisCSP*, where the information of a *CSP* instance is distributed among several agents but it is never concentrated into a single agent. To solve this new model, new algorithms have appeared that communicate by message passing. Among them, we underline the pioneering *ABT* algorithm [11, 12], that has been shown correct and complete.

There are several motivations to solve a *CSP* instance in a distributed form. We can mention the difficulty to collect and move into a single server all the elements of an instance if it is very large, if different formats coexists and the cost of translating them is high. In addition, privacy is a motivation for distributed solving. Many problems appear to be naturally distributed, each part belonging to a different agent. In the solving process, agents desire to keep as private as possible the information they have, and specially they do not want to reveal the values of the solution to other agents.

Although the initial *ABT* was not concerned with privacy issues (agents exchanged their values freely), privacy has been a key aspect for new *DisCSP* solving algorithms. Generally speaking, most distributed algorithms leak some kind of information in the solving process, which can be exploited by some agents to deduce the reserved information of other agents. So far, there are two main approaches to enforce privacy. One considers the use of cryptographic techniques to conceal values and constraints [13, 10].

⋆ Supported by the Spanish project TIN2005-09312-C03-01.

Alternatively, other authors try to enforce privacy by using different search strategies. Our past work has followed this line, and this paper is a further step on this approach.

In previous work, we proposed *Distributed Forward Checking* (*DisFC*) [2]. It is an *ABT*-like algorithm that, instead of sending the value of x_i to agent j (assuming i with higher priority than j), it sends the subset of values that j can take which are compatible with the i value. This idea, combined with the formulation of *Partially Known Constraints*, and the use of sequence numbers to conceal the actual value taken by an agent, allow for some degree of privacy. In particular, when a solution is found, each agent knows its own value but ignores the values of other agents. However, we overlooked the effect that sending the whole subset of compatible values may have in constraint privacy. If i has d different values and in the solving process j receives d different compatible subsets, then j knows all the rows of the constraint matrix that i has, but without knowing their position. In the solving process, it is possible to deduce that some positions are discarded for some rows [8]. At the end, agent j may have a non-negligible amount of information about the constraint that i owns, which could be used to break privacy. Nevertheless, computing the set of constraints that are compatible with the information leaked in the solving process requires a significant amount of work (computing all solutions of a *CSP* instance, that is, solving an NP-hard problem).

To prevent this issue, we suggest a new algorithm called *DisFC*$_{lies}$. It works like standard *DisFC* with a single modification: it may lie in the subsets of compatible values that j may take. Obviously, to keep completeness it has to tell the truth in the values that i truly has. So if i has d values $\{v_1, v_2, \ldots, v_d\}$, *DisFC*$_{lies}$ works as if i would have $d + k$ values $\{v_1, v_2, \ldots, v_d, v_{d+1}, \ldots, v_{d+k}\}$. We call *true values* as the first d values, while the rest are *false values*. When i takes the true value $v_p, 1 \leq p \leq d$, it sends to agent j the subset of values that are compatible with v_p. When i takes the false value $v_q, d < q \leq d + k$, *DisFC*$_{lies}$ sends an invented subset of compatible values to j, with the purpose of making more difficult the hypothetical deduction of j on the actual constraint matrix of i. Again, to assure completeness, *DisFC*$_{lies}$ has to allow all its true values for assignment. As result, this strategy increases the level of privacy at the extra cost of losing performance. This expected result poses a trade-off between efficiency and privacy: enforcing privacy causes to decrease efficiency and vice versa.

In practical terms, what does this mean? First, we have to notice that, even in unsolvable instances (where the major privacy loss occurs), not every agent will have the same level of leaked information. Imagine an instance that contains a single unsolvable subproblem. If the empty nogood is derived exclusively from the interaction of agents in that subproblem, they will have a high level of information about their neighboring constraints, since all possible combinations have been tried. However, agents in other parts of the instance may have less information, if they have reached a consistent assignment with less search. Considering *DisFC* on binary random problems without solution of 16 variables, 10 values per variable and constraint connectivity of 0.4, it may exist one agent that would find 1 matrix compatible with the information leaked, that is, the constraint of the other agent. On the same problems, the new *DisFC*$_{lies}$ would approximately multiply this number by 2, 20 or 200 when allowing 1, 3 or 5 lies per agent, at the extra cost of incrementing computation and communication costs up to the level of solving problems with 11, 13 or 15 values per variable.

The structure of the paper is as follows. In Section 2 we present the basic concepts used in the paper. In Section 3 we discuss the privacy issues of *DisFC* algorithm. In Section 4 we propose the new algorithm that may lie about the values agents take. In Section 5 we provide experimental results. Finally, in Section 6 we extract some conclusions from this work.

2 Preliminaries

A *Constraint Satisfaction Problem* (*CSP*) involves a finite set of variables, each one taking a value in a finite domain. Variables are related by constraints that impose restrictions on the combinations of values that subsets of variables can take. A *solution* is an assignment of values to variables which satisfies every constraint. Formally, 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;
- $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a collection of finite domains; $D(x_i)$ is the initial 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 $prm(C_i)$ of the *permitted* combinations of values for the variables in $var(C_i)$. An element of $prm(C_i)$ is a tuple $(v_{i_1}, \ldots, v_{i_{r(i)}}),\ v_i \in D(x_i)$.

A *Distributed Constraint Satisfaction Problem* (*DisCSP*) is a *CSP* where variables, domains and constraints are distributed among automated agents. Formally, a finite *DisCSP* 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 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 $var(C_j)$ [12].

As in the centralized case, a *solution* of a *DisCSP* is an assignment of values to variables satisfying every constraint. *DisCSP*s are solved by the collective and coordinated action of agents $\mathcal{A}$. Agents communicate by exchanging messages. It is assumed that the delay in delivering a message is finite but random. For a given pair of agents, messages are delivered in the order they were sent.

For simplicity purposes, and to emphasize on the distribution aspects, in the rest of the work 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 $\mathcal{C}_{intra} = \emptyset$. Furthermore, we assume that all constraints are binary. A constraint is written C_{ij} to indicate that it binds variables x_i and x_j.

3 Privacy and DisFC

There are two main concerns about privacy when solving *DisCSP*:

- Privacy of constraints: if agent i is constrained with agent j, i may want to keep private on the part of the constraint known by itself, and the same may occur for j. This generates the *Partially Known Constraints* model (*PKC*) described below.
- Privacy of assignments: agents do not want to reveal the values assigned to their variables to other agents. This is especially relevant for the values of the solution.

3.1 The PKC Model for Constraint Privacy

To enforce constraint privacy, we proposed [2] the *Partially Known Constraints* (*PKC*) model of a *DisCSP* as follows. A constraint C_{ij} is partially known by its related agents. Agent i knows the constraint $C_{i(j)}$ where:

- $var(C_{i(j)}) = \{x_i, x_j\}$;
- $C_{i(j)}$ is specified by three disjoint sets of value tuples for x_i and x_j:
  - $prm(C_{i(j)})$, the set of tuples that i knows to be permitted;
  - $fbd(C_{i(j)})$, the set of tuples that i knows to be forbidden;
  - $unk(C_{i(j)})$, the set of tuples which consistency is not known by i;
- every possible tuple is included in one of the above sets, that is, $prm(C_{i(j)}) \cup fbd(C_{i(j)}) \cup unk(C_{i(j)}) = D_i \times D_j$.

Similarly, agent j knows $C_{(i)j}$, where $var(C_{(i)j}) = \{x_i, x_j\}$. $C_{(i)j}$ is specified by the disjoint sets $prm(C_{(i)j})$, $fbd(C_{(i)j})$ and $unk(C_{(i)j})$ relative to j. Between a constraint C_{ij} and its corresponding partially known constraints $C_{i(j)}$ and $C_{(i)j}$ it holds

$$C_{ij} = C_{i(j)} \otimes C_{(i)j}$$

where $\otimes$ depends on the constraint semantics (see [4] for an example of this). The above definitions satisfy:

- If the combination of values k and l, for x_i and x_j is forbidden in at least one partial constraint, then it is forbidden in the corresponding total constraint.
- If the combination of values k and l, for x_i and x_j is permitted in both partial constraints, then it is also permitted in the corresponding total constraint.

Here, we only consider constraints for which $unk(C_{(i)j}) = unk(C_{i(j)}) = \emptyset$. Then, a partially known constraint $C_{i(j)}$ is completely specified by its permitted tuples and $prm(C_{ij}) = prm(C_{i(j)}) \cap prm(C_{(i)j})$.

For example, let us consider 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 on the chessboard in such a way that no piece attacks any other. As *DisCSP*, the problem is formulated as,

- Variables: one variable per piece.
- Domains: all variables share the domain $\{1, \ldots, m^2\}$ of chessboard positions (cells are numbered from left to right, from top to bottom).

- 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 multiset of pieces {*queen, castle, bishop, bishop, knight*}, on a 4×4 chessboard, with the variables,

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

If agent 1 knows that agent 5 holds a knight, and agent 5 knows that agent 1 holds a queen, this is 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. It only knows 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 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,

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

$C_{1(5)}$ does not depend on agent 5. It codifies the way a queen attacks, independently of any other piece.

3.2 Assignment Privacy on *DisFC*

To achieve assignment privacy, we proposed [2] the *Distributed Forward Checking* (*DisFC*) algorithm as follows. In the centralized case, *Forward Checking* (FC) [6] filters future domains when the current variable is assigned, removing inconsistent values. *DisFC* extends this idea to the distributed case. It performs an *ABT*-search, with the following differences. When a variable x_i is assigned, instead of sending its value to the connected agent j, it sends to j the part of D_j compatible with its value. Variable x_j will choose a new value consistently with x_i (by selecting its new value from the received filtered domain) but without knowing x_i actual value.

To perform backtracking, variable x_j should know some identifier of the value currently assigned to x_i (otherwise, obsolete backtracking cannot be detected). In *ABT* this identifier is the own value; instead, we propose to use the variable sequence number. Each variable keeps a sequence number that starts from 1 (or some random value), and increases monotonically each time the variable changes its value, acting as a unique identifier for each value. Messages including the sender value replace that value by the sequence number of the sender variable. The agent view of the receiver is composed

by the sequence numbers it believes are hold by variables in higher priority agents. Nogoods are formed by variables and their sequence numbers.

DisFC uses both strategies. Each *DisFC* agent sends filtered domains to other agent variables, and it replaces its own value by its sequence number. This allows one agent to exchange enough information with other agents to reach a global consistent solution (or proving that no solution exists) without revealing its own assignment. *DisFC* algorithm performs the same search as *ABT*, with the difference that a constraint is checked by the higher priority agent when sending the filtered domain to the lower priority agent. *DisFC* inherits the correctness and completeness properties of *ABT*. Similar to *ABT*, we can prove that *DisFC* finds a solution or detects inconsistency in finite time.

3.3 DisFC versions

In the PKC model, if agents i and j are constrained, i knows $C_{i(j)}$ and j knows $C_{(i)j}$, but none knows the total constraint C_{ij}. Assuming this model, there are two versions of *DisFC*. The first proposed was *DisFC*$_2$ [2]. It consists of a cycle of two phases,

- Phase I. Constraints are directed forming a DAG, and a compatible total order of agents is selected. Then, *DisFC* finds a solution with respect to constraints $C_{i(j)}$, where i has higher priority than j. If no solution is found, the process stops, indicating unsolvable instance.
- Phase II. Constraints and the order of agents are reversed. Now $C_{(i)j}$ are considered, where j has higher priority than i. x_j informs x_i of its filtered domain with respect to x_j value. If the value of x_i is in that filtered domain, i does nothing. Otherwise, i sends a **ngd** message to j, which receives that message and does nothing. Quiescence is detected.

If no **ngd** messages are generated in phase II, the solution provided in phase I also satisfies $C_{(i)j}$, so it is a true solution. Otherwise, phase I restarts. The nogoods generated in phase II are considered by the receiver agents, now with low priority, so they can change their values to find compatible ones. This cycle iterates until a solution is found or the no solution condition is detected. This strategy is correct and complete.

Instead of checking a part of the constraints in phase I and verifying the proposed solution in phase II, Zivan and Meisels proposed that all constraints could be tested simultaneously [14]. Combining this idea with *DisFC*, we obtain the *DisFC*$_1$ version, that works as follows. An agent has to check all its partially known constraints with both higher and lower priority agents. To do this, an agent has to inform to all its neighbors agents when it takes a new value, and **ngd** messages can go in both directions (from lower to higher as in *ABT* but also from higher to lower). *DisFC*$_1$ inherits the good properties of *ABT-ASC* [14]. *DisFC*$_1$ is correct, complete and terminates.

Similar to *DisFC*, *DisFC*$_1$ agents check consistency with respect to their partial constraints and detect obsolete nogood messages without revealing their assignments. Let $self$ be a generic agent. After an assignment, $self$ informs all constraining agents (with higher and lower priority) via **ok?** messages. Each **ok?** message contains the subset of values for the message recipient that are consistent with $self$'s assignment (filtered domain) and the sequence number corresponding to the $self$'s assignment.

In addition, if a conflict exists between $self$'s assignment and a previously received filtered domain from a lower priority agent, a **ngd** message is sent to that agent.

When $self$ receives an **ok?** message from a higher priority agent i, it checks $C_{(i),self}$ looking for a consistent value. $self$ discards those values which are inconsistent with higher priority agents. If no consistent value is found, $self$ backtracks solving conflicts in $myNogoodStore$, as in *ABT*, sending a **ngd** message. When $self$ receives an **ok?** message from a lower priority agent j, it checks $C_{self(j)}$. If the assignments of $self$ and j are not consistent, $self$ sends a **ngd** message informing to j that its assignment is not valid for $self$'s assignment. Otherwise, $self$ does nothing. **ngd** messages are processed in the same way, no matter if they come from higher or lower priority agents.

The code of *DisFC*$_1$ appears in Figure 1. This code is concurrently executed by each agent. In $myAgentView$, each agent stores the sequence number received from its neighboring agents. In $myNogoodStore$, each agent stores received nogoods from higher and lower agents. In $myFilteredDomains$, each agent saves the last filtered domains received from its (higher and lower) neighbors. Γ^- refers to agents related to $self$ with higher priority, while Γ^+ refers to agents related to $self$ with lower priority.

Agents exchange five kind of messages: **ok?**, **ngd**, **adl**, **stp** and **qcc**. The meaning of **ok?** and **ngd** messages has been described above. **adl** has the same uses as in *ABT* and *DisFC*$_2$: to connect unrelated agents. An extra agent called *system* controls the termination of the algorithm by using **stp** and **qcc** messages. When an agent finds inconsistency it sends an **stp** message to *system*. When *system* receives an **stp** message from one agent or detects quiescence in the network (i.e. no message has traveled through the network in the last t_{quies} units of time), *system* sends messages to all agents informing them to finish the search. In former case, *system* sends **stp** messages to all agents, which is to say that the problem is unsolvable. In latter case, *system* sends **qcc** messages, which is to say that the problem has at least one solution which is given by the current variables' assignments. Quiescence state can be detected by specialized algorithms [5].

3.4 Breaking Privacy

Comparing *DisFC* with *ABT*, the basic difference is as follows. If agent i is constrained with j and i has higher priority, instead of sending the actual value of i to j, it sends the subset of D_j that is compatible with the actual value of i. After reception, j does not know the actual value of i, but it knows a complete row of $C_{i(j)}$ without knowing its position in the matrix. As search progresses, j may store new rows of $C_{i(j)}$. At the end, j has a subset of rows without knowing their position. In addition, some search episodes (changing from phase I to phase II in *DisFC*$_2$, nogood messages from high to low priority agents in *DisFC*$_1$) may reduce the number of acceptable positions for a particular row [8]. With all this, j may construct a *CSP* instance where the variables are the rows, their domains are the acceptable positions, under the constraints that two different rows cannot go to the same position and every row must get a position. Computing all solutions of this instance we obtain all matrixes which are compatible with the information obtained from the search. Of them, one is $C_{i(j)}$. So to break privacy, all solutions of this CSP instance have to be computed (an NP-hard task). In practice, solving this instance requires significant effort and in some cases subsumption testing is required.

```
procedure DisFC-1()
  myValue ← empty; end ← false; compute Γ+, Γ−;
  CheckAgentView();
  while (¬end) do
    msg ← getMsg();
    switch(msg.type)
      ok?     : ProcessInfo(msg);
      ngd     : ResolveConflict(msg);
      adl     : SetLink(msg);
      stp, qcc : end ← true;
procedure CheckAgentView()
  if (myValue = empty or myValue eliminated by myNogoodStore) then
    myValue ← ChooseValue();
    if (myValue) then
      mySeq ← mySeq + 1;
      for each child ∈ Γ+(self) ∪ Γ−(self) do
          sendMsg:ok?(child, mySeq, compatible(D(child), myValue));
      for each child ∈ Γ+(self) such that ¬ (myValue ∈ MyFilteredDomain[child]) do
          sendMsg:ngd(child, self = mySeq ⇒ ¬child.Assig);
    else Backtrack();
procedure ResolveConflict(msg)
  if coherent(msg.Nogood, Γ−(self) ∪ {self}) then
    CheckAddLink(msg);
    add(msg.Nogood, myNogoodStore); myValue ← empty;
    CheckAgentView();
  else if coherent(msg.Nogood, self) then
    SendMsg:ok?(msg.Sender, mySeq, compatible(D(msg.Sender), myValue));
procedure Backtrack()
  newNogood ← solve(myNogoodStore);
  if (newNogood = empty) then
    end ← true; sendMsg:stp(system);
  else
    sendMsg:ngd(newNogood);
    UpdateAgentView(rhs(newNogood) ← unknown);
    CheckAgentView();
function ChooseValue()
  for each v ∈ D(self) not eliminated by myNogoodStore do
    if consistent(v, myAgentView[Γ−]) then return (v);
    else add(x_j = val_j ⇒ self ≠ v, myNogoodStore); /*v is inconsistent with x_j's value */
  return (empty);
procedure UpdateAgentView(newAssig)
  add(newAssig, myAgentView);
  for each ng ∈ myNogoodStore do
    if ¬Coherent(lhs(ng), myAgentView) then remove(ng, myNogoodStore);
procedure SetLink(msg)
  add(msg.sender, Γ+(self));
  sendMsg:ok?(msg.sender, myValue);
procedure CheckAddLink(msg)
  for each (var ∈ lhs(msg.Nogood))
    if (var ∉ Γ−(self)) then
      sendMsg:adl(var, self);
      add(var, Γ−(self)); UpdateAgentView(var ← varValue);
```

Fig. 1. The *DisFC*$_1$ algorithm for asynchronous backtracking search.

4 *DisFC* May Lie

To enhance privacy in *DisFC* we propose that agents could lie. Instead of sending true rows of $C_{i(j)}$, the algorithm may send true and *false* rows. Each false row represents a lie. False rows will make much more difficult the hypothetical reconstruction of $C_{i(j)}$ by agent j, but it has to be done keeping the soundness and completeness of the algorithm.

This idea can be formalized as follows. If i has d values $D_i = \{v_1, v_2, \ldots, v_d\}$, it is assumed that i has an extended domain $D'_i = \{v_1, v_2, \ldots, v_d, v_{d+1}, \ldots, v_{d+k}\}$ of $d + k$ values. We call *true values* the first d values, while the rest are *false values*. When i assigns the true value $v_p, 1 \leq p \leq d$, it sends to agent j the subset of values that are compatible with v_p (that is, a true row of $C_{i(j)}$). When i assigns the false value $v_q, d < q \leq d + k$, it sends an invented subset of compatible values to j (that is a row which does not exist in $C_{i(j)}$). The only concern that an agent must have after assigning a false value is that it must tell the truth (assign a true value or perform backtracking if no more true values are available) in finite time. The point is that no solution could be based on a false value, so assignments including false values have to be removed in finite time (in fact, in a shorter time than required to detect quiescence).

4.1 The *DisFC*$_{lies}$ Algorithm

DisFC$_1$ offers a better platform for privacy than *DisFC*$_2$, because it has no synchronization points between phases. For this reason, we implement the lies idea on top of *DisFC*$_1$ (although it can also be implemented on top of *DisFC*$_2$).

We call *DisFC*$_{lies}$ the new version of *DisFC*$_1$ where agents may exchange false pruned domains. *DisFC*$_{lies}$ appears in Figure 2. It includes most of the procedures, functions and data structures of *DisFC*$_1$, and uses the same types of messages. Each agent has a local clock to control when it has to tell the truth after a lie. In the structure $FalseDomains$, each agent puts away the false domains that it will send to its neighbors for each false value the agent's variable can take. $D_{true}(self)$ is the set of true values for $self$, while $D_{false}(self)$ is the set of its false values. $D(self)$ is the union of these two sets.

In the main procedure, $self$ first initializes its data structures and generates the false domain that it will sent for each false value. Secondly, $self$ assigns a value to its variable by invoking the function $CheckAgentView$. This value may be false or not. Then, $self$ enters in a loop, where incoming messages are received and processed. This loop ends, and therefore the algorithm, when $self$ receives either an **stop** or a **qcc** message from *system*. This is a special agent that handles these messages in the same way it did in *DisFC*$_1$. If $self$ ends the search because a **qcc** message, it means that a problem has at least one solution, otherwise, the problem is unsolvable. Quiescence state can be detected by specialized algorithms [5]. However, in order to assure the completeness and soundness of the algorithm, the time t_{quies} required by *system* to assure quiescence in the network (i.e no message has traveled through the network within the last t_{quies} units of time) must be larger than t_{lies}, the maximum time agents may wait until rectifying their lies, thus $t_{lies} < t_{quies}$.

In the following, we prove that *DisFC*$_{lies}$ is sound, complete and terminates.

```
procedure DisFC_lies()
  myValue ← empty; end ← false; compute Γ+, Γ−; tsaytrue ← 0;
  for each value ∈ D_false(self) do
    for each neig ∈ Γ+(self) ∪ Γ−(self) do generate FalseDomain[value][neig];
  CheckAgentView();
  while (¬end) do
    msg ← getMsg();
    switch(msg.type)
      ok?     : ProcessInfo(msg);
      ngd     : ResolveConflict(msg);
      adl     : SetLink(msg);
      stp, qcc : end ← true;
    if (value ∈ D_false(self)) and (gettime() ≥ tsaytrue) then TakeATrueValue();
procedure CheckAgentView()
  if (myValue = empty or myValue eliminated by myNogoodStore) then
    myValue ← ChooseValue();
    if (myValue) then
      mySeq ← mySeq + 1;
      if (myValue ∈ D_false(self)) then
        for each neig ∈ Γ+(self) ∪ Γ−(self) do
          sendMsg:ok?(neig, mySeq, FalseDomain[myValue][neig]);
        tsaytrue ← gettime() + t_lies;                    /* t_lies < t_quies */
      else
        for each neig ∈ Γ+(self) ∪ Γ−(self) do
          sendMsg:ok?(neig, mySeq, compatible(D(neig), myValue));
        for each child ∈ Γ+(self) such that ¬ (myValue ∈ MyFilteredDomain[child]) do
          sendMsg:ngd(child, self = mySeq ⇒ ¬child.Assig);
        tsaytrue ← 0;
    else Backtrack();
procedure ResolveConflict(msg)
  if coherent(msg.Nogood, Γ−(self) ∪ {self}) then
    CheckAddLink(msg);
    add(msg.Nogood, myNogoodStore); myValue ← empty;
    CheckAgentView();
  else if coherent(msg.Nogood, self) then
    if (myValue ∈ D_false(self)) then
      sendMsg:ok?(neig, mySeq, FalseDomain[myValue][neig]);
    else
      SendMsg:ok?(msg.Sender, mySeq, compatible(D(msg.Sender), myValue));
procedure TakeATrueValue()
  tsaytrue ← 0; myValue ← ChooseATrueValue();
  if (myValue) then
    mySeq ← mySeq + 1;
    for each neig ∈ Γ+(self) ∪ Γ−(self) do
        sendMsg:ok?(neig, mySeq, compatible(D(neig), myValue));
    for each child ∈ Γ+(self) such that ¬ (myValue ∈ MyFilteredDomain[child]) do
        sendMsg:ngd(child, self = mySeq ⇒ ¬child.Assig);
  else Backtrack();
function ChooseATrueValue()
  for each v ∈ D_true(self) not eliminated by myNogoodStore do
    if consistent(v, myAgentView[Γ−]) then return (v);
    else add(x_j = val_j ⇒ self ≠ v, myNogoodStore); /*v is inconsistent with x_j's value */
  return (empty);
```

Fig. 2. The *DisFC*$_{lies}$ algorithm for asynchronous backtracking search. Missing procedures/functions appear in Figure 1.

4.2 Theoretical Results

Lemma 1. *When DisFC$_{lies}$ finds a solution, the last filtered domain received by agent i from agent j corresponds to a (true) row in the partial constraint matrix $C_{i(j)}$.*

Proof. For *DisFC$_{lies}$* the current variables' assignments are a solution if no constraint is violated and network has reached quiescence. Let us assume that *DisFC$_{lies}$* reports a solution in which variable x_i takes a false value. So the last filtered domains sent by agent i are false too. However, *DisFC$_{lies}$* requires that, after lying, an agent must rectify in finite time. That is, assigning a true value and sending to its neighbors the true filtered domains, or performing backtrack. So, at least one **ok?** message or a **ngd** message has traveled through the network after i lied, in contradiction with the initial assumption that the network had reached quiescence. Therefore, the solution condition cannot be reached unless true filtered domains are sent in the last messages from any agent. □

Proposition 1. *DisFC$_{lies}$ is sound.*

Proof. If a solution is claimed, we have to prove that current agents' assignments satisfy their partial constraints. Lemma 1 shows that if *DisFC$_{lies}$* reports a solution the last variables's assignments correspond to true values. Therefore, one can prove that *DisFC$_{lies}$* is sound by using the same arguments to prove that *DisFC$_1$* is sound.

Let us assume quiescence in the network. If the current assignment is not a solution, there exists at least one partial constraint that is violated by agent j. In that case, agent j has sent a **ngd** message to agent i, the closest agent involved in the conflict. This **ngd** is either discarded as obsolete or accepted as valid by agent i. If the message is discarded, it means that some message has not yet reached its recipient, which breaks our assumption of quiescence in the network. If the message is valid, i has to find a new consistent values, which will produce several **ok?** messages or one new **ngd** message, which again breaks our assumption of quiescence in the network. □

Proposition 2. *DisFC$_{lies}$ is complete.*

Proof. Considering only nogoods based on true values, we can prove that *DisFC$_{lies}$* is complete by using the same arguments to prove that *DisFC$_1$* is complete. Since nogoods resulting from an **ok?** message are redundant with respect to the partial constraint matrixes, and the additional nogoods are generated by logical inference, the empty nogood cannot be inferred if the problem is solvable.

Let us prove that *DisFC$_{lies}$* cannot infer inconsistency based on false values if the problem is solvable. Suppose that agent j detects inconsistency because a lie introduced by agent i. We know that j detects inconsistency when it infers an empty nogood. Besides, we know that the left-hand side of the nogoods (justifications of forbidden values) stored by j is either empty or includes agents with higher priority than j. Since we assume that inconsistency discovered by j is based on the false value of i, i is before j in the agents' ordering and there is at least one nogood stored by j including i in its left-hand side. Therefore, when j finds no consistent value, it has to send a backtracking messages to i, which breaks our assumption that j derives an empty nogood. □

Lemma 2. *DisFC$_{lies}$ agents will not store indefinitely nogoods based on false values.*

Proof. Let us assume that a false nogood (i.e. a nogood including an agent with a false value) will be stored indefinitely by an agent. In that case, the lying agent cannot change its variable's assignment, otherwise the nogood will become obsolete and, therefore, deleted by the holder agent. But a lying agent *must* tell the truth in finite time. So, in finite time, the agent storing the false nogood will be informed of a new true value, the false nogood will become obsolete and, therefore, it will be deleted by the holder agent. This breaks our assumption that the false nogood lasts forever. □

Proposition 3. *DisFC$_{lies}$ terminates.*

Proof. By Lemma 2, nogoods based on false values are discarded in finite time. About nogoods based on true values, *DisFC$_{lies}$* performs the same treatment as *DisFC*$_1$. Since *DisFC*$_1$ terminates in finite time, *DisFC$_{lies}$* also terminates in finite time. □

Proposition 4. *If a DisFC$_{lies}$ agent detects inconsistency, every agent directly connected with it has received d true rows.*

Proof. Let i be that agent. If i finds the empty nogood, it means that there is a nogood for every true value of i. These nogoods have an empty left-hand side (otherwise, i could not deduce the empty nogood). So they have been produced as result of **ngd** messages coming from the lower priority agents. Therefore, every possible true value of i has been taken, so i has sent to its neighbors d true rows. □

4.3 Privacy Improvements of *DisFC$_{lies}$*

The inclusion of false values has two direct consequences. First, agent j may receive false rows of $C_{i(j)}$. Then j has more difficulties to reconstruct $C_{i(j)}$, since it is uncertain whether some received rows truly belong to $C_{i(j)}$ or not. Second, this strategy decreases performance, because any computation that includes a false assignment will not produce any solution, so it is a wasted effort, only useful for privacy purposes.

For a solvable instance, Lemma 1 shows that the last assignments correspond to true values. So, agent j knows that the last message from i correspond to a true assignment, and it contains a true row of $C_{i(j)}$. Agent j cannot discriminate whether previous assignments are true or false, so it cannot include the rows of these messages when trying to compute $C_{i(j)}$. So j knows a single row of $C_{i(j)}$ but it does not know its location. The number of different constraint matrixes compatible with this information is approximately $d \cdot 2^{(d^2-d)}$ (d, the number of possible locations for the true row, times $2^{(d^2-d)}$, the number of compatible matrixes when d elements are known). This is a big difference with the approach without lies, where all received rows truly belong to $C_{i(j)}$.

For an unsolvable instance, Proposition 4 shows that every agent j directly connected with the agent i that detects inconsistency would have received d true rows. In addition, since all possibilities have been tried, they have received $d + k$ rows (observe that j cannot receive more than $d + k$ rows). Assuming that j has received $d + k$ *different* rows, if j wants to compute $C_{i(j)}$, it has to select d rows, take them as true rows and solve the corresponding *CSP*. j has to repeat this process $\binom{d+k}{d}$ times, that is, once for each different subset of d rows. This increases the number of *CSP*s to solve, in

order to compute the matrixes compatible with the leaked information. However, j may have received less than $d + k$ *different* rows. In that case, j considers that some rows are repeated. If there is no way to identify repeated rows, in addition to the previously described combinations, we have to consider each possible row as possible repeated, increasing greatly the number of *CSP* instances to solve. As consequence, the privacy level of the solving process is improved.

5 Experimental Results

In this Section, we compare the performance of *DisFC*$_1$ and *DisFC*$_{lies}$ solving instances of binary random classes. A binary random class is defined 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* (the ratio of existing constraints) and p_2 the constraint *tightness* (the ratio of forbidden value pairs). We solved instances of the class $\langle 15, 10, 0.4, p_2 \rangle$ with varying tightness (p_2) between 0.1 to 0.9 in increments of 0.1. For creating these instances in PKC, first we generate random instances and then we split the forbidden tuples of each constraint between its two partial constraints.

We consider three versions of *DisFC*$_{lies}$ that differ from each other in the number of false values that their agents add to initial domains: 1, 3 and 5 false values. Results for all algorithms were produced using a simulator, in which agents are individual processes. Agents are activated randomly. When an agent takes a value, it chooses between true and false values with probability 0.5. t_{lies} is randomly chosen between 1 and 99 internal units of time. Messages are processed by packets, as described in [3].

Algorithmic performance is evaluated by communication effort, computation cost and privacy of constraints. Communication effort is measured by the total number of exchanged messages (msg). Computation cost is measured by the number of non-concurrent constraint checks ($nccc$) [7]. Privacy of constraints is measured by the number of constraint matrixes consistent with the information exchanged among agents. Generally, lower priority agents work more than higher priority ones, therefore they reveals more information than higher priority ones. Thus, we report the minimum (min), median (med) and average (avg) of the numbers of constraint matrixes that are consistent with information exchanged among agents.

Figure 3 shows the computation and communication costs. In both plots, results are averaged on 100 instances. In terms of computation cost, we observe that *DisFC*$_{lies}$ is more costly than *DisFC*, and the cost increases with the number of allowable lies. The differences between algorithms are greater at the difficulty peak ($p_2 = 0.6$). Except for $p_2 = 0.5$, *DisFC*$_{lies}(5)$ always requires more $nccc$ than the others, while *DisFC* performs the lowest number of $nccc$. Similar results appear for communication costs.

Table 1 contains the values of parameters min, med and avg to measure the privacy of constraints. Larger values mean higher privacy. The critical privacy occurs when the number of constraint matrixes is 1 (at least one agent knows exactly the partial constraint matrix of one of its constraining agents). Regarding privacy of constraints in *DisFC*$_1$, the values of min and med decrease when p_2 increases. Actually, in problems with constraint tightness greater than 0.4, at least one agent can infer exactly the partial constraint of one of its constraining agents (see column min). From med values in

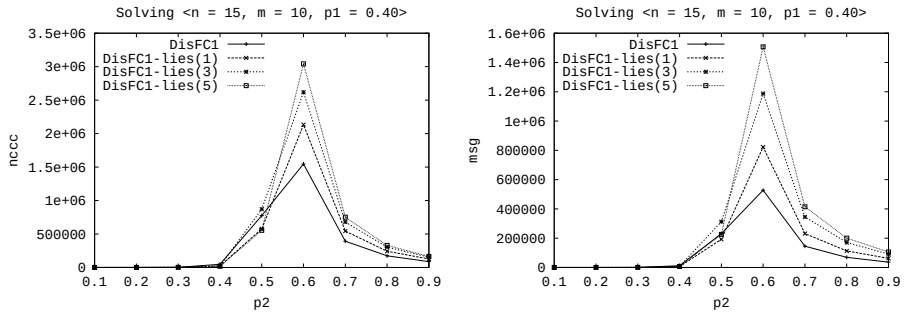

Fig. 3. Computation and communication cost of *DisFC* and versions of *DisFC*$_{lies}$.

unsolvable instances ($p_2 \geq 0.6$), we conclude that approximately in half of partial constraint matrixes all rows are revealed during search since 10^6 is close to $10! = 3.6 \times 10^6$ (the number of permutations of 10 rows). In terms of avg, higher privacy loss occurs at the complexity peak ($p_2 = 0.6$).

Regarding privacy of constraints in *DisFC*$_{lies}$, we notice the following. In solvable instances ($0.1 \leq p_2 \leq 0.5$), *DisFC*$_{lies}$ versions achieve the same level of privacy for min, med and avg, no matter the number of allowable lies. This occurs since each agent can only assure that the last filtered domain received from another agent truly corresponds to a row in the partial constraint matrix of that agent (see Lemma 1), which is independent to the number of false values that agents may have. In terms of min and med, *DisFC*$_{lies}$ versions are more private than *DisFC*$_1$. In unsolvable instances, *DisFC*$_{lies}$ versions have different level of privacy when considering min. *DisFC*$_{lies}(5)$ is one and two orders of magnitude more private than *DisFC*$_{lies}(3)$ and *DisFC*$_{lies}(1)$, respectively. *DisFC*$_{lies}(1)$ is the least private of these three algorithms although it is more private than *DisFC*. *DisFC*$_{lies}$ versions are equally private with respect to med and avg. For these parameters, *DisFC*$_{lies}$ versions are more private than *DisFC*$_1$.

	$DisFC_1$			$DisFC_{lies}(1)$			$DisFC_{lies}(3)$			$DisFC_{lies}(5)$		
p_2	min	med	avg	min	med	avg	min	med	avg	min	med	avg
0.1	10^{23}	10^{28}	10^{29}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}
0.2	10^{23}	10^{27}	10^{29}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}
0.3	10^{16}	10^{24}	10^{29}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}
0.4	10^{7}	10^{14}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}
0.5	**1**	10^{9}	10^{25}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}	10^{27}	10^{28}	10^{28}
0.6	**1**	10^{6}	10^{9}	**3.3**	10^{30}	10^{29}	**20**	10^{30}	10^{29}	**221**	10^{30}	10^{29}
0.7	**1**	10^{6}	10^{12}	**2**	10^{30}	10^{29}	**10.7**	10^{30}	10^{29}	**163**	10^{30}	10^{29}
0.8	**1**	10^{6}	10^{10}	**2.3**	10^{30}	10^{29}	**50.3**	10^{30}	10^{29}	**270**	10^{30}	10^{29}
0.9	**1**	10^{6}	10^{10}	**3.3**	10^{30}	10^{29}	**25.3**	10^{30}	10^{29}	**426**	10^{30}	10^{29}

Table 1. Privacy of constraints measured by the minimum (min), median (med) and average (avg) of the numbers of consistent constraint matrixes. Averaged on 10 instances.

6 Conclusions

From this work we can extract the following conclusions. First, lying is a suitable strategy to enhance privacy in *DisCSP* solving. We have presented *DisFC*$_{lies}$, a new version of the *DisFC* algorithm that may tell lies, sending false compatible domains to neighbor agents. The unique extra condition is that, after a lie, the lying agent has to tell the truth in finite time, lower than t_{quies}. We have proved that this algorithm is correct, complete and terminates. Second, we have shown analytical and experimentally that this idea effectively enhances constraint privacy in the *PKC* model, because it increases the number of partially known constraint matrixes that are compatible with the leaked information of the solving process. And third, although solving *DisCSP* lying is more costly than solving it without lies, experiments show that the extra cost required is not unreachable. It is clear that any strategy used to conceal information will have an extra cost, and this approach is not an exception. We believe that this approach could be useful for those applications with high privacy requirements.

References

1. Bessiere C., Brito I., Maestre A., Meseguer P. The Asynchronous Backtracking without adding links: a new member in the ABT family. *Artifical Intelligence*, 161, 1-2, 7-24, 2005.
2. Brito I., Meseguer P. Distributed Forward Checking. In *Proc. of the CP-2003, LNCS 2833*, 801–806, 2003.
3. Brito I., Meseguer P. Synchronous, Asynchronous and Hybrid algorithms for DisCSP. *CP-2004, Workshop on Distributed Constraint Reasoning*, 2004.
4. Brito I., Meseguer P. Distributed Stable Matching Problems with Ties and Incomplete Lists. *In Proc. of CP-2006, LNCS 4204*, 675–680, 2006.
5. Chandy K., Lamport L. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Computer Systems*, Vol. 3, n. 2, 63–75, 1985.
6. Haralick R., Elliot G. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* **14** (1980) 263–313.
7. Meisels A., Kaplansky E., Razgon I., Zivan R. Comparing Performance of Distributed Constraint Processing Algorithms. *AAMAS-02 Workshop on Distributed Constraint Reasoning*, 86–93, 2002.
8. Meisels A., Zivan R. Personal communication, 2006.
9. Silaghi M.C., Sam-Haroud D., Faltings B. Asynchronous Search with Aggregations. In *Proc. of the AAAI-2000*, 917–922, 2000.
10. Silaghi M.C. Solving a distributed CSP with cryptographic multi-party computations, without revealing constraints and without involving trusted servers, *IJCAI 2003, Workshop on Distributed Constraint Reasoning*, 2003.
11. Yokoo M., Durfee E., Ishida T., Kuwabara K. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. In *Proc. of the 12th. DCS*, 614–621, 1992.
12. Yokoo M., Durfee E., Ishida T., Kuwabara K. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. Knowledge and Data Engineering* **10** (1998) 673–685.
13. Yokoo M., Suzuki K., Hirayama K. Secure Distributed Constraint Satisfaction: Reaching Agreement without Revealing Private Information. In *Proc. of the CP-2002, LNCS 2470*, 387–401, 2002.
14. Zivan R., Meisels A. Asynchronous Backtracking for Asymmetric DisCSPs *IJCAI 2005 Workshop on Distributed Constraint Reasoning*, 2005.