Noname manuscript No.
(will be inserted by the editor)

A New Distributed Algorithm for Efficient Generalized Arc-Consistency Propagation

Shufeng Kong · Jae Hee Lee · Sanjiang Li

Received: date / Accepted: date

Abstract Generalized arc-consistency propagation is predominantly used in constraint solvers to efficiently prune the search space when solving constraint satisfaction problems. Although many practical applications can be modelled as distributed constraint satisfaction problems, no distributed arc-consistency algorithms so far have considered the privacy of individual agents.

In this paper, we propose a new distributed arc-consistency algorithm, called DisAC3.1, which leaks less private information of agents than existing distributed arc-consistency algorithms. In particular, DisAC3.1 uses a novel termination determination mechanism, which allows the agents to share domains, constraints and communication addresses only to relevant agents. We further extend DisAC3.1 to DisGAC3.1, which is the first distributed algorithm that enforces generalized arc-consistency on k-ary ($k \geq 2$) constraint satisfaction problems. Theoretical analyses show that our algorithms are efficient in both time and space. Experiments also demonstrate that DisAC3.1 outperforms the state-of-the-art distributed arc-consistency algorithm and that DisGAC3.1's performance scales linearly in the number of agents.

Keywords Distributed constraint satisfaction problems · Generalized arc-consistency · Privacy · Termination detection

1 Introduction

A *constraint satisfaction problem* (CSP) consists of a set of variables ranging over some domains of possible values, and a set of constraints that specify allowed value combinations for these variables [42]. Solving a CSP amounts to assigning values to its variables such that its constraints are satisfied. Due to this general nature of CSPs, many real world problems such as scene labelling [26], natural language

Shufeng Kong, Jae Hee Lee and Sanjiang Li
Centre for Quantum Software and Information, FEIT, University of Technology Sydney, Sydney, Australia
E-mail: Shufeng.Kong@student.uts.edu.au, {JaeHee.Lee, Sanjiang.Li}@uts.edu.au

parsing [35], picture processing [42], and spatial and temporal reasoning [13,31] can be modelled and solved by using CSPs.

In contrast to the centralized setting, where a single CSP is solved by one agent, in multi-agent setting different CSPs belong to different agents, where some of those problems are linked together through extra constraints. Solving such a *distributed constraint satisfaction problem* (DisCSP) cannot be tackled by a centralized method, as we require a reliable protocol that can coordinate concurrent processes efficiently and preserve the privacy of individual agents to the greatest extent possible.

Formally, a DisCSP consists of a set of local CSPs and a set of external constraints, where each local CSP is owned by an autonomous agent and each external constraint is shared by at least two different agents [53]. These agents aim to assign values to their own local variables cooperatively such that all constraints of the network are satisfied. DisCSPs can be used to model many combinatorial problems that are distributed by nature, e.g., distributed resource allocation problems [11], distributed scheduling problems [49], distributed interpretation tasks [36], multi-agent truth maintenance tasks [27] and multi-agent temporal reasoning [6].

Since solving a CSP is NP-hard in general, local consistency techniques are often used to prune the search space before or during the search for a solution. Among those local consistency techniques, *arc-consistency* (AC) is the most studied and used pruning method for solving binary CSPs [3,5,33,41]. AC has been generalized to *generalized arc-consistency* (GAC) for k-ary ($k \geq 2$) CSPs [41].

There are several distributed AC algorithms proposed in the literature, including DisAC3 [2], DisAC4 [43] and DisAC6 [2], which are, respectively, the distributed versions of AC3 [33], AC4 [43] and AC6 [3]. Another distributed algorithm DisAC9 [21], which is also a distributed version of AC6, is currently the state-of-the-art.

Although privacy is one main motivation and a major concern of solving distributed constraint satisfaction problems (DisCSPs) [17,19,52,55], no distributed AC algorithms so far have considered the privacy of individual agents. Indeed, the distributed AC algorithms mentioned above either assume a complete agent communication graph, which reveals the *communication address*[1], thus the *identity*, of every agent, or broadcast deleted values of variable domains, revealing the existence of variables and their domains.

More precisely, the termination procedure of DisAC3, DisAC4, DisAC6 and DisAC9 assumes that the agent communication graph is complete[2], i.e., any two agents know the communication address of each other, which implies that they know the existence of each other and can directly send messages to each other. Also, whenever an agent deletes a value from one of its local domains, the agent broadcasts this information to all other agents immediately. This setting has the following drawbacks: (i) the algorithm may need to send unnecessarily many number of messages; (ii) the identities of agents and deleted domain values are revealed to irrelevant agents.

In this paper we propose a new distributed algorithm for enforcing AC and the *first* distributed algorithm for enforcing GAC on DisCSPs, which allows the

[1] The address of an agent, as defined in [53], represents the unique identity of the agent in the agent communication graph.

[2] Although not explicitly stated in [21], DisAC9 implicitly assumes a complete agent communication graph, as it uses the distributed snapshot algorithm [8,44].

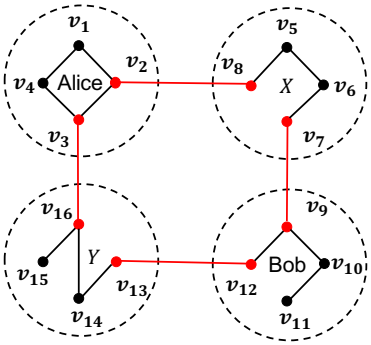


Fig. 1: An illustration of Example 1. Alice, Bob, company X, company Y are four agents, each owning a local network. The dots represent variables and edges constraints. Red edges represent constraints that are shared by two different agents.

agents to share domains, constraints and communication addresses only to relevant agents. It is worthwhile to note that the aspect of parallel processing (cf. [12,20, 45,56]) is not considered in this paper, as it requires global knowledge about the input problem, which is often not available when the knowledge about the problem (i.e., domains and constraints) is distributed among autonomous agents. Because of privacy reasons, collecting all such knowledge from the individual agents is undesirable or impossible [53,54]. Also we assume that each agent owns a local CSP including variables and domains, and not just a single constraint as discussed in [22,23].

The following example, taken from [28], illustrates why we need to consider the privacy of agents when enforcing AC.

Example 1 Let us consider a temporal reasoning problem with discrete time domains illustrated in Fig. 1. Suppose that Alice is looking for a position at company X, she might need to arrange an interview appointment with X. Suppose that her colleague Bob is also applying for the position and Alice and Bob are both applying for another position at another company Y.

To represent and solve such an interview scheduling problem, we need to formulate the problem as a DisCSP. When solving the DisCSP it is desirable that

- Bob does not know that Alice is seeking for a new job (and vice versa);
- company X does not know Alice is also applying for a position at another company Y (and vice versa);
- only the time variables (including their domain values) that are relevant should be shared only with relevant agents.

When existing distributed AC algorithms are applied to the DisCSP, they would reveal the communication address of Alice to all agents, as they assume a complete agent communication graph. In addition, DisAC3, DisAC4 and DisAC6 reveal variable domain values to irrelevant agents.

In this paper we first propose a new distributed AC algorithm, called DisAC3.1, which is based on the optimal AC algorithm AC2001/3.1 [5] and avoids the aforementioned issues. DisAC3.1 uses a novel termination determination mechanism,

where a dedicated agent determines the termination of the algorithm by comparing the timestamps of each message that the sender and the recipient report separately. This termination mechanism does not require a complete agent communication graph, as agents send messages to relevant agents only when necessary. As a result, DisAC3.1 does not reveal more information of each agent than necessary.

Moreover, DisAC3.1 has a low time complexity $O(ed^2)$ and a low space complexity $O(ed)$, where e is the number of edges and d is the largest domain size in the constraint graph of the input DisCSP. Moreover, our experiments also show that DisAC3.1 outperforms the state-of-the-art distributed arc-consistency algorithm DisAC9.

Furthermore, we extend algorithm DisAC3.1 to the *first* distributed algorithm, called DisGAC3.1, that enforces generalized arc-consistency on k-ary ($k \geq 2$) CSPs. Experiments show that the performance of DisGAC3.1 scales linearly in the number of agents.

The remainder of the paper is organized as follows. After a short introduction of necessary background knowledge in Section 2, we describe in Section 3 our new distributed AC algorithm DisAC3.1 and, in Section 4, gives theoretical analyses of DisAC3.1. Then, we extend DisAC3.1 to the first distributed GAC algorithm DisGAC3.1 in Section 5, and evaluate our algorithms empirically in Section 6. The last section concludes the paper.

2 Preliminaries

In this section we recall some basic notions relevant to (distributed) CSPs and arc-consistency.

Definition 1 (constraint satisfaction problem) A *constraint satisfaction problem (CSP)* $\mathcal{N}$ is a triple $\langle V, \mathcal{D}, C\rangle$, where:

- V is a non-empty finite set of variables;
- $\mathcal{D} = \{D_v \mid v \in V\}$ is a collection of finite sets of values. We call $D_v \in \mathcal{D}$ the *domain* of variable $v \in V$;
- C is a finite set of pairs (s, R), called *constraints*, where
  - s, which is called the *scope* of (s, R), is a tuple of variables from V;
  - R is a relation defined over the variables in s, i.e., if $s = (v_1, \ldots, v_l)$ then $R \subseteq D_{v_1} \times \cdots \times D_{v_l}$.

The *arity* of a constraint (s, R) is defined as the cardinality of its scope s. A constraint with arity k is called a *k-ary* constraint and, in particular, a 2-ary constraint is also called a *binary* constraint. A CSP is called k-ary if it has a k-ary constraint but has no constraint of arity greater than k.

Let $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ and $\mathcal{N}' = \langle V', \mathcal{D}', C'\rangle$ be CSPs. We say that $\mathcal{N}'$ is a *subnetwork* of $\mathcal{N}$, if $V' = V$, $C' \subseteq C$ and $D'_v \subseteq D_v$ for each $v \in V$, where D'_v and D_v are the domains of v in $\mathcal{N}'$ and $\mathcal{N}$, respectively.

Let f be a function that assigns to each variable v a value from its domain D_v. We say that f *satisfies* constraint $((v_1, \ldots, v_k), R)$ if $(f(v_1), \ldots, f(v_k))$ is in R and call f a *solution* of $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ if it satisfies all constraints in C. We write $sol(\mathcal{N})$ for the set of solutions of $\mathcal{N}$ and say two CSPs $\mathcal{N}$ and $\mathcal{N}'$ are equivalent, if $sol(\mathcal{N}) = sol(\mathcal{N}')$. We say that $\mathcal{N}$ is *consistent* if it has a solution, i.e., $sol(\mathcal{N}) \neq \varnothing$,

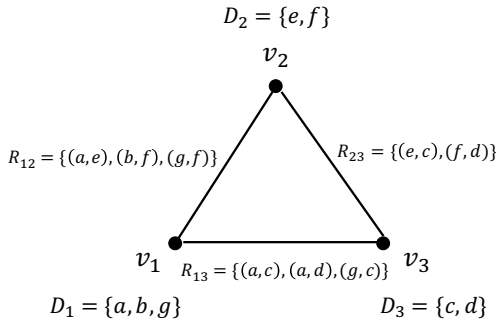


Fig. 2: A CSP example and its constraint graph.

and *inconsistent*, otherwise. We say that $\mathcal{N}$ is *empty* if at least one of its domains or relations is empty; if $\mathcal{N}$ is empty, then it is trivially inconsistent. Fig. 2 illustrates a CSP, where the assignment $(v_1 = a, v_2 = e, v_3 = c)$ is a solution.

In this paper we start with binary CSPs. In Section 5 we extend our results to k-ary CSPs.

Definition 2 (constraint graph) Given a binary CSP $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$, the *constraint graph* G of $\mathcal{N}$ is a pair (V, E), where E is a set of undirected edges over V, such that there is an edge e_{ij} between v_i and v_j in E if and only if $((v_i, v_j), R_{ij}) \in C$.

A binary CSP and its constraint graph are given in Fig. 2.

Let $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ be a binary CSP. We assume that for any pair of variables (v, w), there exists at most one constraint from v to w in C. We write this constraint as $((v, w), R_{vw})$ if it exists, and write

$$R_{vw}^{-1} = \{(b, a) \mid (a, b) \in R_{vw}\}.$$

for the *inverse* of R_{vw}. We also assume that $((w, v), R_{wv})$ with $R_{wv} = R_{vw}^{-1}$ is in C, if $((v, w), R_{vw})$ is in C. For brevity, we often write R_{vw} for the constraint $((v, w), R_{vw})$.

Given a binary CSP $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ and two variables $v, w \in V$, the *image* of $a \in D_v$ under R_{vw}, denoted as $R_{vw}(a)$, is the set $\{b \in D_w \mid (a, b) \in R_{vw}\}$. Each value b in $R_{vw}(a)$ is called a *support* of a on R_{vw}. In Fig. 2 the set of supports of $a \in D_1$ on R_{13} is $\{c, d\}$.

Definition 3 (distributed binary CSP) A *distributed binary CSP* is defined as a pair $\langle \mathcal{P}, C^X\rangle$, where

- $\mathcal{P} = \{\mathcal{N}_1, \ldots, \mathcal{N}_p\}$ is a set of binary CSPs with $\mathcal{N}_i = \langle V_i, \mathcal{D}_i, C_i\rangle$ $(1 \le i \le p)$;
- $C^X = \bigcup\{C_{ij} \mid 1 \le i, j \le p, i \ne j\}$ is a set of *external* constraints, where each C_{ij} is a set of constraints from some variable in V_i to some variable in V_j. We assume that C_{ji} consists of the inverses of the constraints in C_{ij} for all $1 \le i, j \le p, i \ne j$.

We call $v \in V_i$ a *shared* variable of agent i if it is connected to a variable $w \in V_j$ $(j \ne i)$ through an external constraint, and call v a *private* variable of agent i

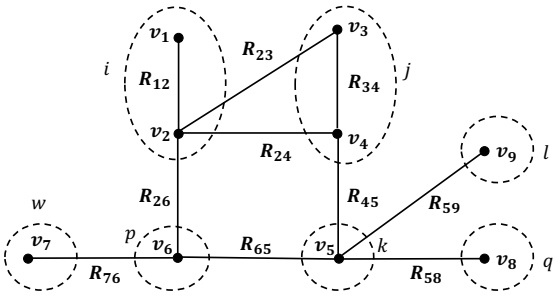


Fig. 3: A distributed binary CSP example: $V_i = \{v_1, v_2\}, C_i = \{R_{12}\}$, $V_j = \{v_3, v_4\}, C_j = \{R_{34}\}$, $V_w = \{v_7\}$, $V_p = \{v_6\}$, $V_k = \{v_5\}$, $V_q = \{v_8\}$, $V_l = \{v_9\}$, $C_{ij} = \{R_{23}, R_{24}\}$, $C_{ip} = \{R_{26}\}$, $C_{jk} = \{R_{45}\}$, $C_{wp} = \{R_{76}\}$, $C_{pk} = \{R_{65}\}$, $C_{kl} = \{R_{59}\}$ and $C_{kq} = \{R_{58}\}$.

otherwise. A constraint R_{vw} is called a private constraint of agent i if $R_{vw} \in C_i$. We call R_{vw} a shared constraint of agent i if either v or w but not both belong to V_i.

The constraint graph of a given distributed binary CSP is defined similarly to binary CSP. An example of a distributed binary CSP is given in Fig. 3. We note that in Definition 3 each binary CSP $\mathcal{N}_i$ corresponds to an agent i and that C_{ij} is the set of constraints shared between agents i and j. We call agent j a *neighbor* of agent i if there is an external constraint between them, i.e., $C_{ij} \neq \varnothing$.

2.1 Arc-Consistency

Definition 4 (arc-consistency) Given a binary CSP $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ and $v, w \in V$, we say that there is an *arc* (v, w) from v to w iff $R_{vw} \in C$. A constraint R_{vw} is *arc-consistent* (AC) iff for every $a \in D_v$, there is a support of a on R_{vw}, i.e., $R_{vw}(a) \neq \varnothing$. We say that $\mathcal{N}$ is AC iff every constraint in C is AC.

In Fig. 2 R_{13} is not AC because $b \in D_1$ has no support on R_{13}, i.e., $R_{13}(b) = \varnothing$. On the other hand, R_{12} is AC because $R_{12}(a)$, $R_{12}(b)$ and $R_{12}(g)$ are all nonempty.

Definition 5 (AC-closure) Given a binary CSP $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$, let $\mathcal{N}' = \langle V, \mathcal{D}', C'\rangle$ be a subnetwork of $\mathcal{N}$. We call $\mathcal{N}'$ an AC-subnetwork of $\mathcal{N}$, if $C' = C$ and $\mathcal{N}'$ is arc-consistent and not empty. We call $\mathcal{N}'$ the *AC-closure* of $\mathcal{N}$, if $\mathcal{N}'$ is the *largest* AC-subnetwork of $\mathcal{N}$ that is equivalent to $\mathcal{N}$, in the sense that every other AC-subnetwork $\mathcal{N}'' = \langle V, \mathcal{D}'', C\rangle$ of $\mathcal{N}$ that is equivalent to $\mathcal{N}$ is a subnetwork of $\mathcal{N}'$.

For every binary CSP $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$, since the AC-subnetworks we considered in the above definition are different from $\mathcal{N}$ only in variable domains, we will identify the AC-subnetwork with its variable domains when the context is clear (e.g., $\mathcal{D}'$ for the AC-subnetwork $\mathcal{N}' = \langle V, \mathcal{D}', C\rangle$ of $\mathcal{N}$). Consider for example the binary CSP in Fig. 2. We have that $\{D'_{v_1} = \{a\}, D'_{v_2} = \{e\}, D'_{v_3} = \{c\}\}$ is an AC-subnetwork of $\mathcal{N}$, while $\{D''_{v_1} = \{a, g\}, D''_{v_2} = \{e, f\}, D''_{v_3} = \{c, d\}\}$ is the AC-closure of $\mathcal{N}$.

Algorithm 1: AC2001/3.1

Input : A binary CSP $\mathcal{N} = \langle V, \mathcal{D}, C \rangle$.
Output: The AC-closure of $\mathcal{N}$, or "inconsistent".

```
1 Q ← {(v, w) | R_vw ∈ C, v ≠ w}
2 while Q ≠ ∅ do
3     (v, w) ← Q.POP()
4     if REVISE(v, w) then
5         if D_v = ∅ then
6             return "inconsistent"
7         Q ← Q ∪ {(u, v) | R_uv ∈ C, u ≠ v}
8 return D
```

Function REVISE(v, w)

```
1  revised ← false
2  foreach a ∈ D_v do
3      b ← SS_vw(a)
4      if b ∉ D_w then
5          b ← NEXTVALUE(b, D_w)
6          while b ≠ NIL and (b ∉ D_w or (a, b) ∉ R_vw) do
7              b ← NEXTVALUE(b, D_w)
8          if b ≠ NIL then
9              SS_vw(a) ← b
10         else
11             delete a from D_v
12             revised ← true
13 return revised
```

The definition of arc-consistency, AC-subnetwork and AC-closure of binary CSPs naturally carry over to distributed binary CSPs. The AC-closure of a (distributed) binary CSP $\mathcal{N}$ has the same solution set as $\mathcal{N}$, but its search space is smaller, which often significantly facilitates the search for a solution. Several AC algorithms for building the AC-closures of binary CSPs have been proposed in the past decades [3,5,33,41,42]. In this paper, we will extend AC2001/3.1 [5], which is known to be optimal, to a distributed algorithm.

The AC2001/3.1 algorithm, presented in Algorithm 1, is similar to the classical AC3 algorithm [33] with the following differences in the REVISE function: in AC2001/3.1 we assume that for any domain D_v, there is an arbitrary ordering imposed on values in D_v. For each constraint R_{vw} and for each value $a \in D_v$, the *smallest support* of a on R_{vw} is stored in $SS_{vw}(a)$. Therefore, if the smallest support of a, say b, is deleted, then the algorithm only needs to search for the next smallest support $b' > b$ of a and does not need to search the whole domain every time as is the case of AC3. For example, suppose that the binary CSP in Fig. 2 is an input to AC2001/3.1, then we have that $SS_{21}(f) = b$. If b is removed from D_1 and the ordering is $a < b < g$, AC2001/3.1 searches a new support for f by starting from g instead of from a. It turns out that introducing "smallest support" makes AC2001/3.1 an optimal AC algorithm.

Theorem 1 ([5]) AC2001/3.1 *has the optimal worst case time complexity* $O(ed^2)$ *with space complexity* $O(ed)$*, where* e *is the number of edges in the constraint graph of the input binary CSP and* d *is the largest domain size.*

3 A New Distributed Arc-Consistency Algorithm

In this section we extend AC2001/3.1 to a new distributed AC algorithm DisAC3.1. In the distributed setting, agents pass messages to communicate with each other. We follow the communication model of [53]:

- Agents communicate by sending messages. Agent i can send messages to agent j iff agent i knows the *communication address* of agent j. Agent i calls function SEND$(i, j, msgType, msgContent)$ to send a message with content *msgContent* and of type *msgType* to agent j.
- For each agent i, there is a message queue associated with it. Messages sent to agent i are stored in the queue. Agent i calls function RECEIVE() to receive all messages stored in the queue. Once RECEIVE() is called, agent i waits until at least one message is received.
- The delay of delivering a message is finite. For the transmission between any pair of agents, messages are received in the order in which they were sent.

Note that this model does not require a physical communication graph to be fully connected (i.e., a complete graph). This model only assumes the existence of a reliable underlying communication structure and is independent of the implementation of the physical communication graph.

Definition 6 (agent communication graph) Given a distributed binary CSP $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$, an *agent communication graph* of $\mathcal{M}$ is a pair (A, E_A), where $A = \{1, 2, ..., p\}$ is the set of agents in $\mathcal{M}$, and E_A is a set of undirected edges over A: $e_{ij} = \{i, j\} \in E_A$ iff agents i and j know the communication address of each other. An agent communication graph is called *standard* if it further satisfies the condition that agents i and j know the communication address of each other iff they share at least one external constraint.

In this paper, we assume that a distributed binary CSP cannot be split into two or more disjoint distributed binary CSPs; in other words, the standard agent communication graph of a distributed binary CSP is connected. The standard agent communication graph of the distributed binary CSP in Fig. 3 is given in Fig. 4.

In order to handle termination of our distributed AC algorithm, we select a *dedicated* agent who decides when to terminate, and allow other agents to communicate with the dedicated agent through a path in the standard agent communication graph. To this end, we build a spanning tree of the standard agent communication graph by using the *echo algorithm* [10]. The root of the spanning tree becomes then the dedicated agent, called the *root agent*. In the echo algorithm, no agent knows the configuration or extent of the communication graph or the addresses of non-neighboring agents, and thus, privacy is not violated. The algorithm runs in $O(\hat{D})$ time and $O(p)$ space, and sends $O(\hat{e})$ messages, where $\hat{D}$, p and $\hat{e}$ are the diameter, number of nodes and number of edges of the standard agent communication graph, respectively.

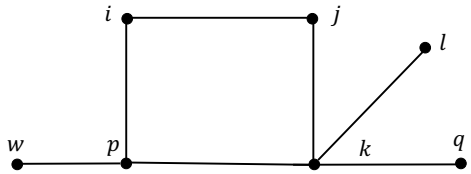


Fig. 4: The standard agent communication graph of the distributed binary CSP in Fig. 3.

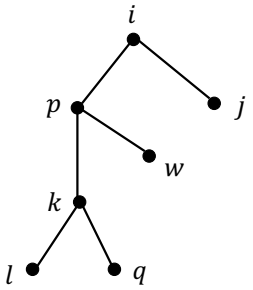


Fig. 5: A T-tree for the standard agent communication graph in Fig. 4.

Given the standard agent communication graph G of a distributed binary CSP $\mathcal{M}$, we call the spanning tree obtained by the echo algorithm the *termination tree* or the *T-tree* of G. We can build a T-tree (cf. Fig. 5) of the standard agent communication graph in Fig. 4. An agent can send messages to the root agent via its path to the root agent, where agents in the middle of the path need to help forwarding messages. For example, in Fig. 5 agent k can send messages to agent i via the path (k, p, i).

3.1 The Algorithm

The new distributed AC algorithm is presented as Algorithm 2. Given a distributed binary CSP $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$, each agent i takes its portion of $\mathcal{M}$ as an input and runs its own copy of Algorithm 2 separately and concurrently. Agent i's portion of $\mathcal{M}$ includes:

- $\mathcal{N}_i = \langle V_i, \mathcal{D}_i, C_i \rangle$.
- For each i's neighbor agent j we have as inputs:
  - a set C_{ij} of all external constraints between i and j,
  - a set V_j^i of all variables of agent j that are related to i through an external constraint in C_{ij}, and
  - a set $\mathcal{D}_j^i$ of domains D_w^i for each $w \in V_j^i$, where each $D_w^i \in \mathcal{D}_j^i$ is a copy of agent j's domain D_w.

Example 2 In Fig. 3 agent i's portion of $\mathcal{M}$ is as follows:

- $\mathcal{N}_i = \langle V_i = \{v_1, v_2\}, \mathcal{D}_i = \{D_{v_1}, D_{v_2}\}, C_i = \{R_{12}\} \rangle$,

Algorithm 2: DisAC3.1

Input : Agent i's portion of a distributed binary CSP $\mathcal{M}$.
A: a map that assigns to each variable $v \in V_i$ the set of all neighbor agents to which v is related through an external constraint.
Output: $\mathcal{D}_i$ or "inconsistent".

Run the echo algorithm to build a T-tree.
if parent $= i$ **then**
  `// The root agent.`
  foreach $k, j \in \{1, 2, \ldots, p\}$ **do**
    $s_{kj} \leftarrow -\infty$, $r_{kj} \leftarrow -\infty$
    $isIdle[k] \leftarrow$ `false`
foreach $j = 1, 2, \ldots, p$ **do** $r_j \leftarrow -\infty$
$Q \leftarrow \left\{(v, w) \,\middle|\, v \in V_i, R_{vw} \in C_i \cup \bigcup_j C_{ij}, v \neq w\right\}$
while `true` **do**
  while $Q \neq \varnothing$ **do**
    $(v, w) \leftarrow Q$.POP()
    if REVISE(v, w) **then**
      if $D_v = \varnothing$ **then**
        Send to all neighbors messages of type "inconsistent".
        return "inconsistent"
      if $A(v) \neq \varnothing$ **then**
        foreach $j \in A(v)$ **do**
          foreach $u \in V_j^i$ **s.t.** $R_{uv} \in C_{ji}$ **do**
            if REVISE(u, v) **then**
              $s \leftarrow$ CURRENTTIME()
              SEND$(i, j,$ "domain update", $(v, D_v, s))$
              SEND$(i,$ parent, "message sent", $(i, j, s))$
              break
      $Q \leftarrow Q \cup \{(u, v) \mid u \in V_i$ s.t. $R_{uv} \in C_i, u \neq v\}$
  SEND$(i,$ parent, "up to date", $(i, (r_j)_{j=1,\ldots,p}))$
  $messages \leftarrow$ RECEIVE()
  while $messages \neq \varnothing$ **do**
    $(msgType, msgContent) \leftarrow messages$.POP()
    if $msgType =$ "arc-consistent" **then**
      Forward the message to all of its children.
      return $\mathcal{D}_i$
    else if $msgType =$ "inconsistent" **then**
      Forward the message to the neighbors.
      return "inconsistent"
    else if $msgType =$ "domain update" **then**
      $(w, D_w^i, r_j) \leftarrow msgContent$
      $Q \leftarrow Q \cup \{(v, w) \mid v \in V_i$ s.t. $R_{vw} \in C_{ij}, v \neq w\}$
    else if parent $= i$ **then**
      `// The root agent.`
      if $msgType =$ "message sent" **then**
        $(k, j, s_{kj}) \leftarrow msgContent$
        if $s_{kj} > r_{jk}$ **then** $isIdle[j] \leftarrow$ `false`
        if $(s_{\ell j})_{\ell=1,\ldots,p} = (r_{j\ell})_{\ell=1,\ldots,p}$ **then** $isIdle[j] \leftarrow$ `true`
      if $msgType =$ "up to date" **then**
        $(j, (r_k)_{k=1,\ldots,p}) \leftarrow msgContent$
        if $(r_{jk})_{k=1,\ldots,p} = (s_{kj})_{k=1,\ldots,p}$ **then** $isIdle[j] \leftarrow$ `true`
      if $isIdle[k] =$ `true` *for all* $k = 1, \ldots, p$ **then**
        Send each of its children a message with type "arc-consistent".
        return $\mathcal{D}_k$
    else Forward the message to its parent.

- $V_j^i = \{v_3, v_4\}$, $\mathcal{D}_j^i = \{D_{v_3}^i, D_{v_4}^i\}$, $C_{ij} = \{R_{23}, R_{24}\}$,
- $V_p^i = \{v_6\}$, $\mathcal{D}_p^i = \{D_{v_6}^i\}$, $C_{ip} = \{R_{26}\}$.

In Algorithm 2, lines 2–5 and lines 37–47 are only run by the root agent, who is the dedicated agent to handle the termination. (Note that in our algorithm we distinguish the root agent from non-root agents by setting the parent of the root agent to itself.) These extra codes are used for handling termination of the algorithm, which are explained in details in Section 3.2.

In Algorithm 2, a queue Q is first initialized, which stores all arcs (v, w) with $R_{vw} \in C_i \cup \bigcup_j C_{ij}$ (line 7). The queue Q includes arcs (v, w) that correspond to constraints $R_{vw} \in C_i$ as is the case in AC2001/3.1. In addition, Q also includes arcs (v, w) that correspond to external constraints $R_{vw} \in C_{ij}$ with $v \in V_i$; excluded in the queue are arcs (w, v) that correspond to constraints $R_{wv} \in C_{ji}$ with $w \in V_j^i$ and $v \in V_i$.

Example 3 Let the distributed binary CSP in Fig. 3 be an input of DisAC3.1. Then Q in line 7 includes arcs $(v_1, v_2), (v_2, v_3), (v_2, v_4)$ and (v_2, v_6) but not arcs $(v_3, v_2), (v_4, v_2)$ and (v_6, v_2).

Then the agent iteratively takes an arc (v, w) from Q, and revises D_v to make R_{vw} arc-consistent (lines 9–23). We will consider two cases: (i) If a domain D_v becomes empty, then the agent broadcasts "inconsistent" to its neighbors, and then returns "inconsistent" as its output (lines 12–14). Its neighbors will further broadcast "inconsistent" to their neighbors (in the standard agent communication graph) who have not yet received an "inconsistent" message (lines 31–32) and so on, until every agent has received an "inconsistent" message (c.f. Fig. 7a). (ii) If a domain D_v is revised but is not empty, then there are arcs (u, v) that are potentially affected by the revision of D_v so that R_{uv} is no longer arc-consistent. Therefore, arcs (u, v) with $u \in V_i$ are added to Q (line 23), if they were not included already in Q. Arcs (u, v) with $u \in V_j^i$ for an agent j are dealt with slightly differently: since the domain D_u of u is maintained by agent j in this case, agent i reports to its neighbor agent j about the revision of D_v so that agent j can revise D_u only when necessary (lines 15–20).[3] Here agent i sends for each revised domain at most one message to agent j, because there is no need to report to agent j about the same revision more than once (cf. line 22). The following example illustrates the ideas mentioned in this paragraph.

Example 4 We consider the variable v_2 as well as other variables, domains and relations that are relevant to v_2, of the distributed binary CSP in Fig. 3, which is illustrated in Fig. 6. Let $D_{v_2} = \{a, b\}$, $D_{v_3} = \{c\}$, $D_{v_4} = \{d\}$, $D_{v_6} = \{e\}$ and $R_{23} = \{(a, c)\}$, $R_{24} = \{(a, d)\}$, $R_{26} = \{(a, e), (b, e)\}$. Then if a is removed from D_{v_2}, agent i will add arc (v_1, v_2) to Q if arc (v_1, v_2) was not already in Q, because constraint R_{12} may be no longer arc-consistent. Also, constraints R_{32}, R_{42} and R_{62} may be no longer arc-consistent. So agent i will notify the revision of D_{v_2} to agents j and p when necessary. Suppose i first checks whether it should inform p about the revision of D_{v_2} by calling Revise(v_6, v_2). Recall that i owns a copy $D_{v_6}^i$ of the original domain D_{v_6} of v_6. In this case, $D_{v_6}^i = D_{v_6} = \{e\}$. We know that Revise$(v_6, v_2)$ will return false, because although a is removed from D_{v_2}, we can

[3] This technique was first introduced in [21].

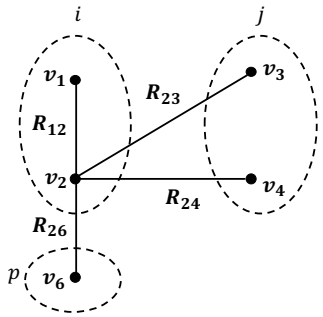


Fig. 6: The relevant parts of variable v_2 of the distributed binary CSP in Fig. 3.

still find another support b for e on R_{62}. In the process of running REVISE(v_6, v_2), the algorithm only changes e's smallest support on R_{62} from a to b. In other words, if REVISE(v_6, v_2) returns false, we know that every element in $D^i_{v_6}$ is still supported on R_{62}. Thus, R_{62} is arc-consistent w.r.t. $D^i_{v_6}$ and D_{v_2}. Since we always have $D^i_{v_6} \supseteq D_{v_6}$ (cf. Lemma 2), we also have that R_{62} is arc-consistent w.r.t. D_{v_6} and D_{v_2}. Therefore, i finds that there is no need to notify the revision of D_{v_2} to p. However, with a similar analysis, i will find that R_{32} is not arc-consistent w.r.t. $D^i_{v_3}$ and D_{v_2}, and R_{42} is not arc-consistent w.r.t. $D^i_{v_4}$ and D_{v_2}, so i needs to notify the revision of D_{v_2} to the owners of v_3 and v_4. Because both D_{v_3} and D_{v_4} are owned by j, i only needs to inform the revision of D_{v_2} to j once.

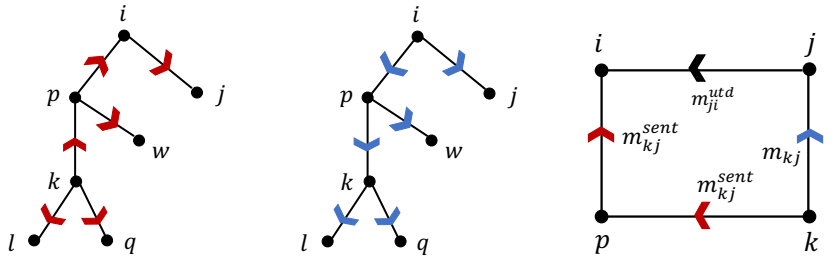
The root agent uses the T-tree to collect timestamps of each message that the sender and the recipient report. If an agent i sends a "domain update" message to one of its neighbors j, i will notify the root agent that a message was sent to agent j. To this end, agent i will first send a "message sent" message to its parent, and then the parent will forward the message to its parent (line 48) and so on, until the message is received by the root agent.[4] Note that in both of the messages to agent j and the root agent, we add the same timestamp s to the messages, which serves as a means for the root agent to confirm later that the message from agent i is received and processed by agent j.

Example 5 Suppose that we have the T-tree in Fig. 5 for the standard agent communication graph in Fig. 4. Suppose that k sends a "domain update" message m_{kj} to j at time s. Then k sends a message $(k, p,$ "message sent", $(k, j, s))$ (line 21) with timestamp s to p and p will forward the message to its parent i by sending another message $(p, i,$ "message sent", $(k, j, s))$ (line 48). See Fig. 7c for an illustration.

Once agent i is done with adding new arcs to Q and sending messages to other agents, it repeats the whole process with the updated Q until there are no more arcs in Q (lines 9–23), i.e., Q is empty.

When there are no more arcs in Q to be processed, agent i reports to the root agent that its state is up to date (line 24), which is forwarded by the ancestors

[4] Note that the sender's "id" remains in the forwarded message, but this "id" does not need to be the real id of the sender, it can be a *code name* (cf. [30]), e.g., a randomly generated character string. In this case, two neighbours need to exchange their code names with each other before running the algorithm.



(a) Agent k is broadcasting inconsistency.

(b) The root agent is broadcasting arc-consistency.

(c) Conformation of a "domain update" message m_{kj}: m_{kj}^{sent} is the "message sent" message and m_{ji}^{utd} is the "up to date" message.

Fig. 7: Illustrations of messages flow.

of agent i (cf. line 48), and shares with the root agent the latest timestamps $(r_{ij})_{j=1,\ldots,p}$ of all incoming messages of its neighbor agents j; these timestamps allow the root agent to determine whether the last message agent i received from agent j is the one that agent j reported to the root agent about.

Afterwards, agent i processes the messages that have been waiting to be processed in its message queue one by one in a FIFO manner (line 25). If the type of a message is "arc-consistent" or "inconsistent", then i will forward the message to relevant neighbors and then the algorithm terminates and returns $\mathcal{D}_i$ or the value "inconsistent", respectively (lines 28–33). If the type of a message is "domain update" sent from another agent j, which prompts agent i to update a domain D_w^i when $w \in V_j^i$, then agent i updates D_w^i and adds all arcs (v, w) that are potentially affected by the update to Q (lines 34–36).

When there are no more messages to be processed, then the algorithm repeats again with processing the arcs in the queue Q (line 9–23).

3.2 Handling Termination

A distributed algorithm should terminate when there are no messages in transit and all agents are waiting for new messages (i.e., idle). In fact, this is a well studied problem in the field of distributed systems, called *distributed termination detection* (DTD) [44]. DTD is a difficult problem as there is no simple way of gathering global knowledge of the distributed system. A number of algorithms to solve the DTD problem have been proposed in the literature (cf. [37]). These algorithms can be categorized based on eight features: algorithm type (wave, parental responsibility, credit-recovery, or ad-hoc), required topology of the agent communication graph, algorithm symmetry, required process knowledge (e.g., upper bound on the diameter of the agent communication graph or identity of the central agent), communication protocol (synchronous or asynchronous), communication channel behavior (first-in first-out (FIFO) or non-FIFO), message optimality, and fault tolerance [37].

Most DTD algorithms, like the one by Chandy and Misra [7], are wave algorithms and typically repetitive. These wave algorithms send wave after wave until termination is detected, which causes it to send up to $O(M \times p)$ control messages, where M is the number of basic messages and p is the number of agents. This repetitive property makes it unattractive for implementation. The algorithm proposed by Shavit and Francez [47] combines the algorithm of Dijkstra and Scholten [15] with a cycle-based repetitive wave algorithm.

One may integrate one of these DTD algorithms into DisAC3.1 to handle the its termination, but there are several aspects we need to bear in mind.

- Requirement on the topology of the agent communication graph and process knowledge should not be too strict. For example, we cannot require the agent communication graph to be a Hamiltonian cycle agent network (such as Dijkstra et al. [14], Shavit and Francez [47] and Mayo and Kearns [39]), or require agents to know the upper bound on agent network diameter (such as Skyum and Eriksen [16]), or require agents to know which one is the central agent (i.e., initiator and/or detector) (such as Huang [24]).
- Concerning performance, we would not consider algorithms that have very high message complexity (such as the repetitive wave algorithms [50, 7, 38] which require a large number of messages in the worst case) and algorithms that have high memory requirement (such as [51]).

In this paper, we choose to develop a customized termination handling approach for algorithm DisAC3.1, which is not a wave algorithm, not parental responsible or credit-recovery. In our approach, agents report timestamps of both sent and received "domain update" messages to the root agent (lines 21 and 24). See Fig. 8 for an illustration. The root agent runs additional codes (lines 2–5 and lines 37–44) to handle these timestamps.

There are two conditions to determine the termination of a distributed AC algorithm:

1. For each agent $i = 1, \ldots, p$ all "domain update" messages sent from other agents to agent i are received and processed by agent i and agent i is waiting for new messages;
2. An agent sent "inconsistent" messages to its neighbors, which indicates that some local domain has become empty and the distributed CSP is thus inconsistent.

It is straightforward to check the satisfiability of the second condition. Once an agent finds out that one of its variable domains is empty, it sends an "inconsistent" message to each of its neighbors (line 13) and then other agents will help to broadcast inconsistency (lines 31–32). See Fig. 7a for an example.

To check the satisfiability of the first condition, the root agent receives for each "domain update" message m_{ij} from sender i to recipient j two reports—one from the sender and one from the recipient, where the sender's report has type "message sent" and includes the same timestamp of m_{ij} and the recipient's report has type "up to date" and includes a timestamp not earlier than that of m_{ij}.[5] The root agent stores the timestamp reported by the sender i in variable s_{ij} (line 39) and the timestamp reported by the recipient j is stored in variable r_{ji} (line 43).

[5] This is because when several messages from agent i are received, the "up to date" message will only bear the timestamp of the latest one.

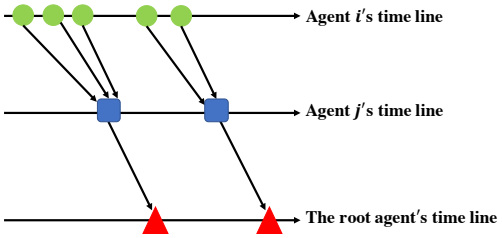


Fig. 8: Time lines of sending and receiving messages, where circles stand for the time points that agent i sends "domain update" messages to agent j, squares stand for the time points that agent j reports the latest received time stamps of "domain update" messages to the root agent via "up to date" messages, and triangles stand for the time points that the root agent receives "up to date" messages from agent j.

In what follows, we write m_{ij}^{sent} for the corresponding "message sent" message sent from agent i to the root agent which bears the same timestamp as m_{ij}; and write m_{ij}^{utd} for the first "up to date" message sent from agent j to the root agent after it has received and processed m_{ij}.

For a message m_{ij} of type "domain update" we say m_{ij} is *confirmed* by the root agent, if (cf. Fig. 7c)

- the root agent has received m_{ij}^{sent} (of type "message sent") from i;
- the root agent has received m_{ij}^{utd} (of type "up to date") sent from j.

When confirming a message we have to consider two cases. Since the communication speeds may be different among different pair of agents, m_{ij}^{sent} from agent i to the root agent may arrive earlier or later than m_{ij}^{utd} from agent j to the root agent. It is also worth reminding that every agent processes the messages that have been waiting to be processed in its message queue one by one in a FIFO manner.

Given a "domain update" message m_{ij} sent from agent i to agent j, if m_{ij}^{sent} arrives *earlier* at root agent than m_{ij}^{utd}, then the confirmation of message m_{ij} is accomplished when the root agent receives m_{ij}^{utd} (lines 42–44). On the other hand, if m_{ij}^{sent} arrives *later* at root agent than m_{ij}^{utd}, then the confirmation of message m_{ij} is accomplished when the root agent receives m_{ij}^{sent} (lines 38–41).

Whenever all messages sent to an agent i have been confirmed by the root agent (i.e., $s_{ji} = r_{ij}$ for all $1 \leq j \leq p$), then variable $isIdle[i]$ is set to `true` (lines 41 and 44). When $isIdle[i] =$ `true` for all agents i, then the root agent sends "arc-consistent" messages to all of its children (lines 45–46) and other agents further help broadcast the arc-consistent messages (lines 28–29). See Fig. 7b for an example.

4 Analysis of DisAC3.1

4.1 Privacy of Individual Agents

For a distributed algorithm it is desirable that the algorithm does not give away the privacy of the involved agents, i.e., information of an individual agent is shared

with other agents only when it is necessary to achieve a common goal. In the case of distributed binary CSPs such information could include information about the variables, domains, constraints and the communication address of each agent.

In this paper, we assume that agents are *honest*, in the sense that no agent reveals its and/or its neighbors' private information (e.g., its variables, domains, constraints, or communication addresses) to irrelevant agents. With this assumption, in this subsection we show that DisAC3.1 leaks less private information of agents than existing distributed arc-consistency algorithms.

The following notion of *semi-private information* is adapted from [30].

Definition 7 (semi-private information) A piece of information about a DisCSP is regarded as *semi-private* if one agent or several agents might consider it private, but it can inevitably be leaked to irrelevant agents by their local knowledge of the AC-closure of the DisCSP.

In this paper, the semi-private information of a DisCSP concerns mainly about privacy of one agent (e.g., its variables, domains, and constraints) that can be inferred by another agent (or several agents jointly) by comparing the change of domains they know before and after enforcing a distributed arc-consistency algorithm.

Example 6 Consider a DisCSP which has three variables x, y, z and two agents A, B such that A owns x, y and B owns z. Suppose there is a shared constraint R_{xz} that is AC in the beginning. Note that A knows the domain of z and B knows the domain of x, but it does not know the existence of y (let alone its domain) in the beginning. After enforcing AC, if D_z is changed, then B can infer from this change that A must own another variable which is connected to x. The information that "A has another variable which is connected to x" is thus a piece of semi-private information. This information will be leaked to B inevitably (even if we use encryption).

Theorem 2 *Let $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$ be an input to Algorithm 2 and $\mathcal{N}_i$ the local binary CSP of agent i. Algorithm 2 satisfies the following conditions modulo semi-private information:*[6]

1. *The existence of a variable v of agent i and its domain are only known to agents who share the variable.*
2. *An external constraint of agent i is only known to another agent who shares the constraint with agent i.*
3. *The communication address of agent i is only known to its neighbor agents.*
4. *Private variables and their domains, and private constraints are all hidden to other agents.*

Proof In Algorithm 2, information about shared variables and external constraints is only shared by agents that are involved in the external constraints, thus conditions 1–2 are satisfied modulo semi-private information. Furthermore, Algorithm 2 only requires the standard agent communication graph, i.e., two agents know the communication address of each other iff they share at least one external constraint, thus condition 3 is satisfied. Since no other information is revealed to other agents, condition 4 is satisfied modulo semi-private information. □

[6] Here by 'modulo semi-private information', we mean that we do not consider leak of semi-private information as privacy loss.

The four conditions in Theorem 2 are closely related to three types of privacy proposed by Léauté and Faltings [30], namely, agent privacy, topology privacy and constraint privacy and the type of domain privacy introduced by Grinshpoun [18]. Agent privacy relates to the identities of the participants. Topology privacy concerns the existence of topological constructs in the constraint graph, such as nodes (i.e., variables) and edges (i.e., constraints). Constraint privacy corresponds to the nature of a constraint and domain privacy concerns the content of the domain of a variable, which could also be regarded as constraint privacy related to a unary constraint. Clearly, condition 1 is related to both topology and domain privacy, condition 2 is related to constraint privacy, and condition 4 involves topology, domain, and constraint privacy. As mentioned in [30], a particular consequence of the agent privacy is that two agents should only be allowed to communicate directly if they share a constraint. Therefore, condition 3 is related to the agent privacy.

However, we note that DisAC3.1 as presented in Algorithm 2 does not strictly protect privacy of individual agents. First, constraint privacy is not strictly maintained if we care about semi-private information. Because even though agents cannot learn directly about constraint information, they can implicitly learn some information regarding constraints due to values that disappear from the domains of their neighbors. If such information is crucial in applications, one must apply some privacy protection measure (such as encryption [55]) to address the issues.

Second, DisAC3.1 does not protect all information about the agent constraint graph. Actually, before running DisAC3.1, we assume each agent only knows its neighbors in the standard agent communication graph and its parent and children in the T-tree, in particular, each agent knows whether it is the root agent. When an "up to date" or "message sent" message is sent from an agent, passing through internal agents, to the root agent, these internal agents and the root agent may learn about partial topology information of the agent communication graph. Consider the example in Fig. 3. Suppose at time s agent k sends to j a "domain update" message m and sends to its parent p (cf. Fig. 5) a "message sent" message m_{kj}^{sent}. Then, since k is not the root, p will forward m_{kj}^{sent} to root i after receiving it. Note that m_{kj}^{sent} carries the information (k, j, s_{kj}) (here we assume j, k are disguised in their code names without leaking their identities) and thus both p and i know that (i) k is a descendant and (ii) k and j share a constraint. In general, each agent will know, in the extreme case and when the algorithm terminates, all its descendants (disguised in their code names) in the T-tree and all neighbors of each of these descendants in the standard agent communication graph. But this does not mean it will know the whole agent communication graph: it is unlikely that, for example, k knows that i is the root of the T-tree in Fig. 5.

Note that leaking topology information of the agent communication graph to intermediate agents can be avoided by encrypting the whole message such that only the root agent can see the content, i.e., all the agents know the public key of the root agent and encrypt the messages using that key and the root agent, once receiving a message, decrypts the message by using its private key. No other intermediate agents can read the messages' content, as they do not have the private key of the root agent.

Alternatively, we can also alleviate the second limitation by introducing a *trustworthy system agent* instead of using the root agent of a T-tree to detect termination. The idea is to ask each agent to send their "message sent" and "up to date"

messages directly to the system agent; if it detects that every agent is idle, then it sends an "arc-consistent" message to each agent; and if there is an inconsistency detected by an agent, it sends an "inconsistent" message directly to the system agent and the latter forwards the message to all other agents. In this *system-agent version of* DisAC3.1,[7] only the system agent knows the standard agent communication graph but it does not know the other private information (i.e., variables, domains, and constraints information about the DisCSP). For all non-system agents, they know only their neighbors in the standard agent communication graph. Another advantage of the system-agent version is that it's termination detection procedure is also message optimal (see Corollary 1). The system-agent version of DisAC3.1 requires a trustworthy system agent connecting to all other agents, which is not always available.

Unlike DisAC3.1, none of DisAC3, DisAC4 and DisAC6 satisfies conditions 1, 3, and 4 in Theorem 2 (modulo semi-private information). This is because they broadcast deleted values of variable domains to all agents, which reveals the existence of agents' variables and domains, and they assume a complete agent communication graph, which reveals every agent's communication address. DisAC9 does not satisfy condition 3 either, because the termination protocol it employs also assumes a complete agent communication graph. Therefore, we can conclude that DisAC3.1 leaks less information about individual agents than all previous distributed AC algorithms. Nevertheless, we note that privacy loss of a distributed algorithm is difficult to evaluate outside the perspective of its integration with final applications such as search [48]. Since we do not integrate our algorithm with search in this paper, we will not evaluate privacy loss of distributed AC algorithms. Future research may use existing privacy measuring approaches such as [34, 46] to measure privacy loss of distributed AC algorithms integrated with search.

4.2 Termination and Correctness

In this subsection, we prove that DisAC3.1 terminates and is correct. The proof of the termination result uses the following lemma.

Lemma 1 *The following conditions are equivalent:*

1. *isIdle*$[i]$ = `true` *for all* $i = 1, \ldots, p$.
2. *The root agent has received from all agents their first "up to date" messages (cf. line 24) and* $s_{ij} = r_{ji}$ *for all* $i, j = 1, \ldots, p$.
3. *All "domain update" messages are confirmed and no agent is running the first inner while-loop (cf. lines 8-23).*

Proof See Appendix.

Theorem 3 *Algorithm 2 terminates.*

Proof Let $\mathcal{N}_i = \langle V_i, \mathcal{D}_i, C_i \rangle$ be agent i's local binary CSP of an input distributed binary CSP $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$. We note first that after a certain number of iterations of the outer while-loop (lines 8–48) of Algorithm 2, either the algorithm exits and returns a value ($\mathcal{D}_i$ or "inconsistent") or waits for new messages (line 25).

[7] In this paper, unless otherwise stated, DisAC3.1 always refers to the original Algorithm 2.

The outer while-loop cannot iterate forever, as this can only happen, if at each iteration: (i) there exists a neighbor agent j who sends "domain update" messages to agent i (line 34), i.e., the domain of some variable of j has been updated and its size decreased; however, because all domains of $\mathcal{M}$ are finite, the number of "domain update" messages is finite, or (ii) there exists some agent k sends "message sent" messages to the root agent; however, because one "domain update" message induces one "message sent" message, the number of "message sent" messages is also finite, or (iii) there exists some agent k sends "up to date" messages to the root agent; however, because one "domain update" message induces at most one "up to date" message, the number of "up to date" messages is also finite. Analogously, the two inner while-loops (lines 9–23 and lines 27–48) iterate only finitely many times.

Since i is chosen arbitrarily above, we can assume that all "domain update" messages that were sent have been confirmed by the root agent and no agent is running the first inner while-loop. Then, by Lemma 1, *isIdle*$[i]$ = `true` for all $i = 1, \ldots, p$ and the root agent sends "arc-consistent" messages to its children and exits (lines 45–47), and agent i, on receiving the message, forwards the message and returns $\mathcal{D}_i$, and then Algorithm 2 terminates. □

To prove the correctness of DisAC3.1, we need the following two lemmas.

Lemma 2 *Let $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$ be an input distributed binary CSP for Algorithm 2, and let $v \in V_i$ and $u \in V_j$ such that $R_{uv} \in C_{ji}$ for some agents i and j, $i \neq j$. Then, the algorithm terminates with $D_u^i \supseteq D_u$.*

Proof See Appendix.

Lemma 3 *Let $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$ be an input distributed binary CSP for Algorithm 2. Then, each application of* Revise *does not change the AC-closure of $\mathcal{M}$.*

Proof When popping an arc (v, w) out of Q (line 10), the application of function Revise(v, w) only removes values from D_v that have no supports w.r.t. R_{vw}. We know that those values cannot be parts of the AC-closure [33]. Thus, each application of Revise does not change the AC-closure of $\mathcal{M}$. □

Theorem 4 *Given an input distributed binary CSP $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$, if Algorithm 2 returns a new set of domains $\mathcal{D}'_i$ for $i = 1, \ldots, p$, then $\mathcal{M}' = \langle \mathcal{P}', C^X \rangle$, where $\mathcal{P}' = \{\mathcal{N}'_1, \ldots, \mathcal{N}'_p\}$ and $\mathcal{N}'_i = \langle V_i, \mathcal{D}'_i, C_i \rangle$, is the AC-closure of $\mathcal{M}$. If the algorithm returns "inconsistent" then $\mathcal{M}$ is inconsistent.*

Proof Suppose Algorithm 2 returns $\mathcal{D}'_i$ for $i = 1, \ldots, p$. It does so only if during the execution of the distributed algorithm there was a time (or a state) t_{AC} where *isIdle*$[i]$ = `true` for all $i = 1, \ldots, p$ (cf. Algorithm 2, lines 45–47). Thus to prove the claim we need to show that at time t_{AC} the outcome $\mathcal{M}' = \langle \mathcal{P}', C^X \rangle$ is AC. To this end, we show that at time t_{AC} for each agent $i = 1, \ldots, p$ and for each neighbor agent j of agent i, any constraint $R_{v_0 w_0} \in C_i \cup C_{ij}$ is AC.

Suppose $R_{v_0 w_0}$ is from C_i. Due to Lemma 1 we can assume that at time t_{AC} all "domain update" messages that were sent have been confirmed by the root agent and no agent is running the first inner while-loop. We show inductively that $R_{v_0 w_0}$ is AC.

1. After the algorithm initializing the queue Q (line 7), running the first inner while-loop (lines 9–23) for the first time and waiting for new messages (line 25), then $R_{v_0w_0}$ is AC. This is because all operations involved are exactly the same as those in AC2001/3.1 except for the operations in lines 15–22, which however do not affect the domains of v_0 and w_0.
2. After receiving new "domain update" messages (line 34), new pairs of variables are added to Q. When the algorithm enters the first inner while-loop again then the queue Q is processed as in AC2001/3.1 (except for the operations in lines 15–22), establishing arc-consistency of $R_{v_0w_0}$.

Thus, $R_{v_0w_0}$ is AC at time t_{AC}.

Now suppose j is a neighbor agent of i and $R_{v_0w_0}$ is from C_{ij}, i.e., variable w_0 does not belong to agent i but to agent j. Looking at the algorithm from agent j's perspective, we have by Lemma 2 that $D^j_{v_0} \supseteq D_{v_0}$ at time t_{AC}. We also observe that due to line 18 at time t_{AC} constraint $R_{v_0w_0}$ is AC w.r.t. $D^j_{v_0}$ and D_{w_0}. Thus, at time t_{AC} for any $a \in D_{v_0} \subseteq D^j_{v_0}$ there exists $b \in D_{w_0}$ such that $(a, b) \in R_{v_0w_0}$, i.e., $R_{v_0w_0}$ is AC at time t_{AC}.

Thus $\mathcal{M}'$ is an AC-subnetwork of $\mathcal{M}$. By Lemma 3 it is also the AC-closure of $\mathcal{M}$.

If the algorithm returns "inconsistent", then it follows by Lemma 3 that the AC-closure of $\mathcal{M}$ has an empty domain. Thus, $\mathcal{M}$ is inconsistent. □

4.3 Time and Space Complexities

When analyzing the time complexity of a distributed algorithm, one usually assumes that the delay of sending a message is zero.

Theorem 5 *The time complexity of Algorithm 2 is $O(ed^2)$, where e is the number of edges in the constraint graph of the input distributed binary CSP and d is the largest domain size.*

Proof In the analysis of time complexity we are interested in the number of increment operations (lines 17–18 of Algorithm 2) on the elements of each domain as this dominates all other numbers of operations. These increment operations take place in lines 5 and 7 of Function Revise. To this end, we need to first determine the number of calls to Revise. We first note that Revise is applied to a pair (v, w) of variables only in one of the two cases: (i) the pair is from Q, in particular, the first variable of the pair is from V_i (lines 10–11 of Algorithm 2) (ii) the first variable of the pair is from V^i_j, the set of all variables from agent j that are related to i through an external constraint (lines 17–18 of Algorithm 2). In either of the mentioned two cases, Revise is applied to a pair of variables, which was added to Q not because of the initialization of Q in line 7, if and only if the domain of the second variable was revised. Except the first revision, since each revision of a domain reduces its size, each ordered pair of variables are revised at most $d + 1$ times. Let (v, w) be a pair of variables and t_ℓ $(1 \leq \ell \leq d + 1)$ be the number of increment operations that takes place at the ℓth call to Revise(v, w). Then $\sum_1^{d+1} t_\ell \leq d^2$, because for each $a \in D_v$ only d many increments of $SS_{vw}(a)$ is possible. Thus, for each agent i there are altogether $T_i = O((|C_i| + |C_{ij}|) \cdot \sum_1^{d+1} t_\ell) = O((|C_i| + |C_{ij}|) \cdot d^2)$ number

of increment operations. In the worst case, DisAC3.1 proceeds with a sequential behavior and the time complexity of DisAC3.1 is

$$\sum_{i=1}^{p} T_i = \sum_{i=1}^{p} O((|C_i| + |C_{ij}|) \cdot d^2) \leq d^2 \sum_{i=1}^{p} O(|C_i| + |C_{ij}|)$$
$$\leq d^2 \cdot O(|C|) = O(ed^2).$$

Since running the echo algorithm to build a T-tree only takes $O(\hat{D})$ time, where $\hat{D} \leq p \leq n$ is the diameter of the standard agent communicate graph, it does not affect the time complexity of DisAC3.1. □

Theorem 6 *The space complexity of Algorithm 2 is $O(ed)$.*

Proof For each variable $w \in V_j^i$ of some agent j, agent i keeps a copy of D_w, which requires $O(ed)$ space for all agents. Since the other used data structures of DisAC3.1 are the same as that of AC2001/3.1, the space complexity is the same as that of AC2001/3.1, i.e., $O(ed)$. Since the echo algorithm only takes $O(p)$ space with $p \leq n$, it does not affect the space complexity of DisAC3.1. □

4.3.1 Number of Message Passing Operations

We analyze the number of messages sent in DisAC3.1.

Theorem 7 *Let $\mathcal{M}$ be a distributed binary CSP. Then, on input $\mathcal{M}$,* DisAC3.1 *sends at most $O(nd\alpha h)$ messages, where α is the largest vertex degree of the standard agent communication graph G of $\mathcal{M}$ and h is the height of the T-tree of G.*

Proof DisAC3.1 send at most $nd\alpha$ "domain update" messages, because there are at most nd domain elements in total and each deletion of the elements in a domain leads to the dispatch of at most α "domain update" messages. Each dispatch of a "domain update" message is followed by the dispatch of a "message sent" message as well as of at most one "up to date" message. Because a "message sent" or an "up to date" message may need to be forwarded at most h times until it reaches the root agent, there are at most $O(nd\alpha h)$ messages that Algorithm 2 sends. Since the echo algorithm sends $O(\hat{e})$ messages, where $\hat{e} \leq e \leq n\alpha$ is the number of edges of the standard agent communication graph, it does not affect the message complexity of DisAC3.1. □

Consider the system-agent version of DisAC3.1. For each distributed binary CSP $\mathcal{M}$, after introducing an system agent A_0, the standard communication graph of G of $\mathcal{M}$ has a T-tree with height 1, in which the root is A_0 and all regular agents are children of A_0. By Theorem 7, the system-agent version of DisAC3.1 sends at most $O(nd\alpha)$ messages on input $\mathcal{M}$. Regarding "domain update" messages as basic messages and "message sent' and "up to date" messages as control messages, from the proof of Theorem 7, it is easy to see that the system-agent version of DisAC3.1 sends at most $O(M)$ contral messages in total, where M is the number of basic messages it sends. Because the number of control messages to detect the termination of a distributed system is at least M [9], the following corollary follows directly from Theorem 7.

Corollary 1 *The termination detection procedure of the system-agent version of* DisAC3.1 *is message optimal.*

v_1	v_2	v_3
a	c	e
a	c	f
b	d	f

Fig. 9: A ternary constraint (s, R), where $s = (v_1, v_2, v_3)$ and $R = \{(a, c, e), (a, c, f), (b, d, f)\}$.

5 Non-binary Constraints

In this section we extend our results to k-ary CSPs.

5.1 Generalized Arc-Consistency

Definition 8 (generalized arc-consistency [41]) Let $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ be a k-ary CSP. Let (s, R) be a constraint of C with $s = (v_1, \ldots, v_\ell)$. Given a value $a \in D_{v_i}$ with $1 \leq i \leq \ell$, a tuple $t = (t[v_1], \ldots, t[v_\ell]) \in R$ is called a *candidate support* of a on R, if $t[v_i] = a$. If t is a candidate support of a on R and additionally $t \in D_{v_1} \times \ldots \times D_{v_\ell}$ then we call t a *support* of a on R.

A constraint (s, R) is said to be *generalized arc-consistent (GAC)* iff for each variable v in s every value $a \in D_v$ has a support on R. CSP $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$ is called GAC iff every constraint of C is GAC.

We note that for binary CSPs GAC coincides with AC.

Definition 9 (GAC-closure) Given a network $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$, let $\mathcal{N}' = \langle V, \mathcal{D}', C'\rangle$ be a subnetwork of $\mathcal{N}$. We call $\mathcal{N}'$ an GAC-subnetwork of $\mathcal{N}$, if $C' = C$ and $\mathcal{N}'$ is generalized arc-consistent and not empty. We call $\mathcal{N}'$ the *GAC-closure* of $\mathcal{N}$, if $\mathcal{N}'$ is the *largest* GAC-subnetwork of $\mathcal{N}$ that is equivalent to $\mathcal{N}$, in the sense that every other GAC-subnetwork $\mathcal{N}'' = \langle V, \mathcal{D}'', C\rangle$ of $\mathcal{N}$ that is equivalent to $\mathcal{N}$ is a subnetwork of $\mathcal{N}'$.

Every k-ary constraint (s, R) can be represented as a table, where entries of the table are the tuples of R. We use $\prec_R$ to denote the ordering of entries of the table representation of R. For each constraint (s, R) and for each variable $v \in s$ and each value $a \in D_v$, the *smallest support* of a on R w.r.t. $\prec_R$ is stored in $SS_R(a)$.

Example 7 A ternary constraint (s, R) is represented as a table in Fig. 9. Suppose $D_{v_1} = \{a, b\}, D_{v_2} = \{c, d\}$ and $D_{v_3} = \{e, f, g\}$. Then the tuple (a, c, e) is the smallest support of $a \in D_{v_1}$ on R. However, $g \in D_{v_3}$ has no support on R, so this constraint is not GAC.

In [5], algorithm AC2001/3.1 is extended to a generalized arc-consistency (GAC) algorithm, called GAC2001/3.1. The main change is the extension of function Revise to n-ary constraints. In this extended Revise, finding a support for $a \in D_{v_i}$ on a given constraint (s, R) is realized by checking for each $t \in D_{v_1} \times \ldots \times D_{v_\ell}$ first whether $t[v_i] = a$ and then whether $t \in R$. This requires to check $O(d^{k-1})$ tuples, where d is the maximum number of elements in a domain and k is the arity of R.

In the following we present function GRevise, which is a modification of the extended Revise. Here, we check first whether $t \in R$ following the ordering $\prec_R$ and then check whether $t[v_i] = a$ and $t[v_j] \in D_{v_j}$ for all $v_j \in s \backslash \{v_i\}$. Since membership queries can be done in $O(1)$ using hash-tables, this modification allows for reduced number of checks when R is sparse.

Function GRevise(v, s, R)

```
1  revised ← false
2  foreach a ∈ D_v do
3      t ← SS_R(a)
4      if ∃v' ∈ s with t[v'] ∉ D_v' then
5          t ← NextTuple(t, R)
6          while t ≠ NIL and (t[v] ≠ a or ∃v' ∈ s with t[v'] ∉ D_v') do
7              t ← NextTuple(t, R)
8          if t ≠ NIL then
9              SS_R(a) ← t
10         else
11             delete a from D_v
12             revised ← true
13 return revised
```

Next we present a GAC algorithm as Algorithm 3, called GAC2001/3.1$*$. It is different from AC2001/3.1 in that it stores in Q triples (v, s, R). It is also different from GAC2001/3.1 in that it calls the modified function GRevise (line 4). GAC2001/3.1$*$ shares the nice property with AC2001/3.1, i.e., for each value $a \in D_v$ and each constraint (s, R) with $v \in s$, a tuple $t \in R$ is only visited once.

Theorem 8 *The time complexity of algorithm* GAC2001/3.1$*$ *is* $O(ek^2d\beta)$ *where* e *is the number of constraints of the input CSP,* k *is the arity of the network,* d *is the largest domain size and* β *is the largest number of candidate supports of a value on a relation.*

Proof We first analyze the cost of enforcing GAC on a single constraint (s, R). For any variable $v \in s$ and any value $a \in D_v$, there are at most β candidate supports needed to be confirmed (line 6 of Function GRevise). Confirming a candidate costs $O(k-1)$ time. Because there are rd values needed to be processed per constraint, enforcing GAC on a single constraint costs $O(rd \cdot \beta \cdot (k-1)) = O(k^2 d\beta)$. Since there are at most e constraints, the time complexity of GAC2001/3.1$*$ is $O(ek^2d\beta)$. □

Note that we must have $\beta \leq d^{k-1}$, so in the worst case the time complexity of GAC2001/3.1$*$ is $O(ek^2d^k)$, which is the same as that of GAC2001/3.1.

5.2 A Distributed Generalized Arc-Consistency Algorithm

We now extend GAC2001/3.1$*$ to a distributed generalized arc-consistency algorithm, called DisGAC3.1.

Algorithm 3: GAC2001/3.1∗

Input : A CSP $\mathcal{N} = \langle V, \mathcal{D}, C\rangle$.
Output: The GAC-closure of $\mathcal{N}$, or "inconsistent".

```
1 Q ← {(v, s, R) | (s, R) ∈ C, v ∈ s}
2 while Q ≠ ∅ do
3     (v, s, R) ← Q.POP()
4     if GREVISE(v, s, R) then
5         if D_v = ∅ then
6             return "inconsistent"
7         Q ← Q ∪ {(w, s, R) | (s, R) ∈ C, v, w ∈ s, w ≠ v}
8 return N
```

Definition 10 (distributed k-ary CSP) A *distributed k-ary CSP (DisCSP)* is defined as a pair $\langle \mathcal{P}, C^X\rangle$, where

- $\mathcal{P} = \{\mathcal{N}_1, \ldots, \mathcal{N}_p\}$ is a set of k-ary CSPs with $\mathcal{N}_i = \langle V_i, \mathcal{D}_i, C_i\rangle$ $(1 \le i \le p)$;
- C^X is a set of *external* constraints, where each $(s, R) \in C^X$ is shared by at least two different agents, i.e., there exist two different variables $v, w \in s$ s.t. $v \in V_i, w \in V_j, i \neq j$.

The definitions of (standard) agent communication graphs of distributed binary CSPs (cf. Definition 6), GAC (cf. Definition 8) and GAC-closure (cf. Definition 9) naturally carry over to distributed k-ary CSPs.

The distributed GAC algorithm is presented as Algorithm 4. Given a DisCSP $\mathcal{M} = \langle \mathcal{P}, C^X\rangle$ $(1 \le i \le p)$ each agent i takes its portion of $\mathcal{M}$ as an input and runs its own copy of Algorithm 4 separately and concurrently. Agent i's portion of $\mathcal{M}$ includes:

- $\mathcal{N}_i = \langle V_i, \mathcal{D}_i, C_i\rangle$,
- $C^X(V_i) = \big\{(s, R) \in C^X \;\big|\; s \cap V_i \neq \varnothing\big\}$,
- a set $\mathcal{D}^i$ of D^i_w, where D^i_w is a copy D_w and w belongs to a neighbor agent $j \neq i$, i.e., there is an external constraint $(s, R) \in C(V_i)$ such that w is in scope s.

Example 8 Suppose we have a DisCSP $\mathcal{M} = \langle \mathcal{P} = \{\mathcal{N}_1, \mathcal{N}_2\}, C^X\rangle$, where $\mathcal{N}_1 = \langle V_1 = \{v_1\}, \mathcal{D}_1 = \{D_{v_1}\}, C_1 = \varnothing\rangle$, $\mathcal{N}_2 = \langle V_2 = \{v_2, v_3\}, \mathcal{D}_2 = \{D_{v_2}, D_{v_3}\}$, $C_2 = \varnothing\rangle$ and $C^X = \{(s_1 = (v_1, v_2, v_3), R_1), (s_2 = (v_1, v_2), R_2)\}$ as an input to Algorithm 4. Agent 1's portion of $\mathcal{M}$ includes:

- $\mathcal{N}_1$,
- $C^X(V_1) = \{(s_1, R_1), (s_2, R_2)\}$,
- $\mathcal{D}^1 = \{D^1_{v_2} = D_{v_2}, D^1_{v_3} = D_{v_3}\}$.

Algorithm 4 largely overlaps with Algorithm 2. Differences are highlighted in light gray background. In Algorithm 4, a queue Q is first initialized to store all the tuples (v, s, R) with $(s, R) \in C_i \cup C^X$ and $v \in s$ (line 7), and line 24 adds all new tuples (w, s', R') to Q with $w \in V_i, v, w \in s'$ and $w \neq v$ because of the revision of D_v. Also, if a new domain D_w is received from some other agent, agent i will add new tuples (v, s, R) to Q, where (s, R) is an external constraint, $v \in V_i$ and both

Algorithm 4: DisGAC3.1

Input : i's portion of a DisCSP $\mathcal{M}$.
Output: $\mathcal{D}_i$ or "inconsistent".

Run the echo algorithm to build a T-tree.
if parent $= i$ **then**
  // The root agent.
  foreach $k, j \in \{1, 2, \ldots, p\}$ **do**
    $s_{kj} \leftarrow -\infty$, $r_{kj} \leftarrow -\infty$
    $isIdle[k] \leftarrow$ `false`
foreach $j = 1, 2, \ldots, p$ **do** $r_j \leftarrow -\infty$
$Q \leftarrow \{(v, s, R) \mid v \in V_i, (s, R) \in C_i \cup C^X, v \in s\}$
while `true` **do**
  while $Q \neq \varnothing$ **do**
    $(v, s, R) \leftarrow Q$.POP()
    if GREVISE(v, s, R) **then**
      if $D_v = \varnothing$ **then**
        Send each of its neighbors a message with type "inconsistent".
        return "inconsistent"
      if $C^X(v) \neq \varnothing$ **then**
        $L \leftarrow \varnothing$
        foreach $(s, R) \in C^X(v)$ **do**
          foreach $u \in s$ **s.t.** $u \notin V_i$, OWNER$(u) \notin L$ **do**
            if GREVISE(u, s, R) **then**
              L.APPEND(OWNER(u))
              $s \leftarrow$ CURRENTTIME()
              SEND$(i, j,$ "domain update", $(v, D_v, s))$
              SEND$(i,$ parent, "message sent", $(i, j, s))$
      $Q \leftarrow Q \cup \{(w, s, R) \mid w \in V_i, (s, R) \in C_i \cup C^X, v, w \in s, w \neq v\}$
  SEND$(i,$ parent, "up to date", $(i, (r_j)_{j=1,\ldots,p}))$
  $messages \leftarrow$ RECEIVE()
  while $messages \neq \varnothing$ **do**
    $(msgType, msgContent) \leftarrow messages$.POP()
    if $msgType =$ "arc-consistent" **then**
      Forward the message to all of its children.
      return $\mathcal{D}_i$
    else if $msgType =$ "inconsistent" **then**
      Forward the message to the neighbors.
      return "inconsistent"
    else if $msgType =$ "domain update" **then**
      $(w, D^i_w, r_j) \leftarrow msgContent$
      $Q \leftarrow Q \cup \{(v, s, R) \mid v \in V_i, (s, R) \in C^X, v, w \in s\}$
    else if parent $= i$ **then**
      // The root agent.
      if $msgType =$ "message sent" **then**
        $(j, s_{kj}) \leftarrow msgContent$
        if $s_{kj} > r_{jk}$ **then** $isIdle[j] \leftarrow$ `false`
        if $(s_{\ell j})_{\ell=1,\ldots,p} = (r_{j\ell})_{\ell=1,\ldots,p}$ **then** $isIdle[j] \leftarrow$ `true`
      if $msgType =$ "up to date" **then**
        $(j, (r_k)_{k=1,\ldots,p}) \leftarrow msgContent$
        if $(r_{jk})_{k=1,\ldots,p} = (s_{kj})_{k=1,\ldots,p}$ **then** $isIdle[j] \leftarrow$ `true`
      if $disIdle[k] =$ `true` *for all* $k = 1, \ldots, p$ **then**
        Send each of its children a message with type "arc-consistent".
        return $\mathcal{D}_k$
    else Forward the message to its parent.

$v, w \in s$ (line 37). Lines 16, 18 and 20 are to guarantee that if a domain D_v is revised, D_v is sent to relevant agents at most once, just like line 22 of Algorithm 2.

The correctness of DisGAC3.1 follows from the correctness of DisAC3.1 and GAC2001/3.1∗, and we have the following:

Theorem 9 *Given an input DisCSP $\mathcal{M} = \langle \mathcal{P}, C^X \rangle$, Algorithm 4 terminates. If Algorithm 4 returns a new set of domains $\mathcal{D}'_i$ for $i = 1, \ldots, p$, then $\mathcal{M}' = \langle \mathcal{P}', C^X \rangle$, where $\mathcal{P}' = \{\mathcal{N}_1, \ldots, \mathcal{N}'_p\}$ and $\mathcal{N}'_i = \langle V_i, \mathcal{D}'_i, C_i \rangle$, is the GAC-closure of $\mathcal{M}$. If the algorithm returns "inconsistent" then $\mathcal{M}$ is inconsistent.*

Theorem 10 *The time and space complexities of algorithm* DisGAC3.1 *are $O(ek^2\beta)$ and $O(ndke)$, respectively, where e is the number of constraints of the input network, k is the arity of the network, d is the largest domain size and β is the largest number of candidate supports of a value on a relation.*

Proof In the worse case, algorithm DisGAC3.1 proceeds with a sequential behavior, so constraints will be enforced GAC one by one and DisGAC3.1 has the same worse case time complexity of algorithm GAC2001/3.1∗, which is $O(ek^2\beta)$.

Because every constraint can have a scope containing up to k variables, Q takes $O(ke)$ space. Since, in the worst case, each agent can share all the variables of other agents by means of external constraints, copied domains $\mathcal{D}^i$, $(i = 1, \ldots, p)$ takes in total $O(ndp)$ space. The data structure SS_R of Function GRevise associated with a constraint (s, R) takes $O(ndk)$ space, because there are at most $O(nd)$ domain values for a constraint (s, R) and we store for each value its smallest support, which takes $O(k)$ space. Since there are e constraints SS takes $O(ndre)$ space. By adding all this together we obtain $O(ne + ndp + ndke) = O(nd(p + ke))$. Since we assume that the standard agent communication graph is connected, for each agent there is an external constraint (s_i, R_i) shared by at most $k_i - 1$ other agents where k_i is the arity of (s_i, R_i) and we have $p \leq \sum_{i=1}^{e} k_i \leq ke$. Thus $O(ndke)$ is the space complexity of DisGAC3.1.

Similar to the cases of DisAC3.1, running the echo algorithm does not affect the time and space complexities of DisGAC3.1. □

Since the termination handling mechanism of DisGAC3.1 is same as that of DisAC3.1, we have the following result.

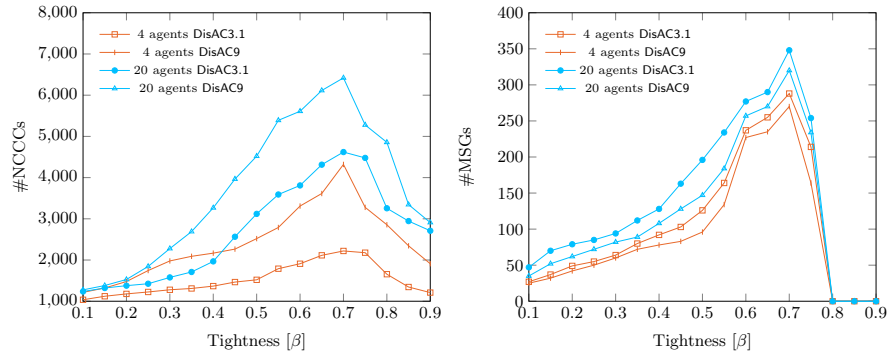
Theorem 11 *Let $\mathcal{M}$ be an input DisCSP for* DisGAC3.1. *Then* DisGAC3.1 *send at most $O(nd\alpha h)$ messages, where α is the largest vertex degree of the standard agent communication graph G of $\mathcal{M}$ and h is the height of the T-tree of G.*

Similar to the system-agent version of DisAC3.1 discussed before, we can also introduce a system-agent version of DisGAC3.1[8], which sends at most $O(nd\alpha)$ messages.

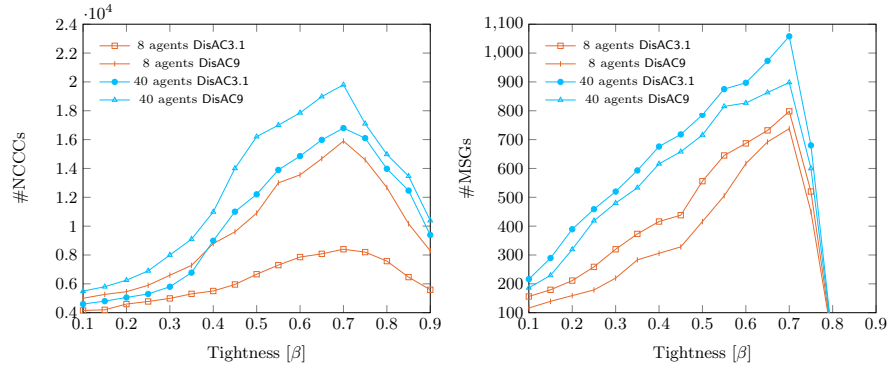
6 Evaluation

In this section we experimentally compare DisAC3.1 against the state-of-the-art distributed AC algorithm DisAC9. We also evaluate our GAC algorithm DisGAC3.1

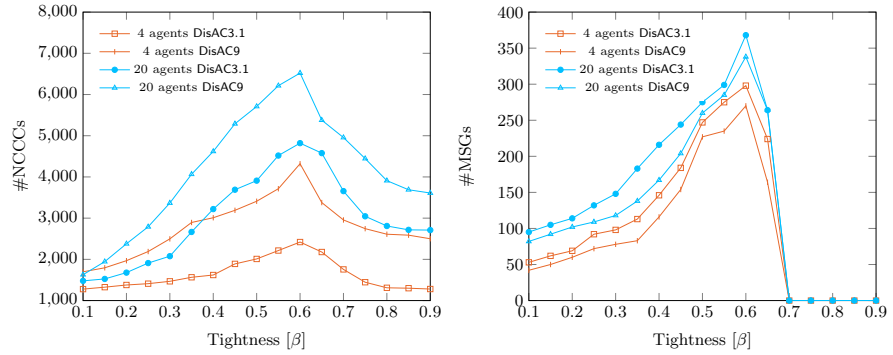
[8] In this paper, unless otherwise stated, DisGAC3.1 always refers to the original Algorithm 4.



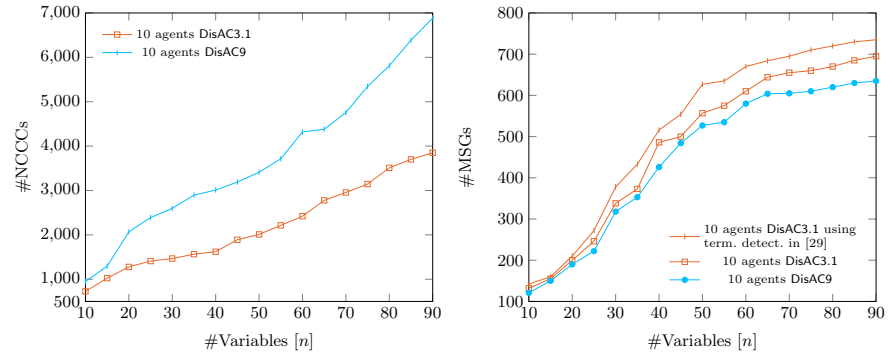
(a) $\langle n = 20, d = 10, \rho = 0.30 \rangle$



(b) $\langle n = 40, d = 20, \rho = 0.30 \rangle$



(c) $\langle n = 20, d = 10, \rho = 0.60 \rangle$



(d) $\langle d = 10, \rho = 0.30, \beta = 0.50 \rangle$

Fig. 10: Performance comparisons between DisAC3.1 and DisAC9 on random instances.

	DisAC3.1		DisAC9	
Instances $\langle n, d, m, p\rangle$	#NCCCs	#Msgs	#NCCCs	#Msgs
$\langle 1000, 10, 1000, 20\rangle$	1.031e+4	964	2.137e+4	923
$\langle 500, 100, 500, 25\rangle$	3.013e+5	3027	5.854e+5	3078
$\langle 300, 300, 300, 30\rangle$	1.364e+6	7318	4.565e+6	7271

Table 1: Performance comparisons between DisAC3.1 and DisAC9 on the DOMINO problem. Note that m is the number of constraints and p is the number of agents used by both the algorithms.

by comparing it against its centralized counterpart GAC2001/3.1$*$, owing to lack of other existing distributed GAC algorithm. All experiments used an asynchronous simulator in which agents are simulated by processes which communicate only through message passing. Two independent measures of performance are commonly used for evaluating the performance of distributed algorithms: (i) the number of *non-concurrent constraint checks* (#NCCCs) (cf. [6, 40]) and (ii) the total number of messages sent (#MSGs) (cf. [32]).[9] All experiments for distributed algorithms used an asynchronous simulator in which (i) agents are simulated by processes which communicate only through message passing and (ii) default communication latency is assumed to be zero. Our experiments were implemented in Python 3.6 and carried out on a computer with an Intel Core i5 processor with a 2.9 GHz frequency per CPU, 8 GB memory.

6.1 DisAC3.1 vs. DisAC9

To evaluate the two distributed AC algorithms, we consider both benchmark problems and randomly generated binary CSPs whose AC-closures are not empty. The random instances were generated by using the random model proposed in [25]. The generator involves four parameters: (1) the number of variables n, (2) the largest domain size d, (3) the density $\rho = 2|C|/n(n+1)$ of the input binary CSP, and (4) the tightness of constraints $\beta = M/d^2$, where M is the number of allowed tuples of a binary constraint.

Following the convention of [3, 5, 21, 40], experimental results with random instances are presented as figures where the x-axis represents the constraint tightness. The experiments presented here were performed on four groups of instances, where group A has instances with $n = 20, d = 10, \rho = 0.3$, group B has instances with $n = 40, d = 20, \rho = 0.3$, group C has instances with $n = 20, d = 10, \rho = 0.6$ and group D has instances with $d = 10, \rho = 0.3, \beta = 0.5$. Note that we have conducted the experiments on a large number of different classes of CSPs and the results were similar. The results reported here present a summary of all the results we have observed.

[9] Note that these two measures do not reflect the cost of the echo algorithm used in our distributed algorithms. However, since the number of agents are relatively small comparing to the number of variables of the input and the time complexity of the echo algorithm is only $O(\hat{D})$, where $\hat{D} \leq p \leq n$ is the diameter of the standard agent communication graph, the cost of the echo algorithm is negligible.

Instances $\langle n, d, m, p \rangle$	DisAC3.1		DisAC9	
	#NCCCs	#Msgs	#NCCCs	#Msgs
SCEN#01 $\langle 916, 44, 5548, 35 \rangle$	4.411e+5	5690	1.123e+6	5223
SCEN#11 $\langle 680, 44, 4103, 30 \rangle$	2.762e+6	8703	2.292e+7	9132
GRAPH#09 $\langle 916, 44, 5246, 35 \rangle$	4.735e+5	5350	1.391e+6	5378
GRAPH#10 $\langle 680, 44, 3907, 30 \rangle$	7.156e+5	6467	2.007e+6	6718
GRAPH#14 $\langle 916, 44, 4638, 35 \rangle$	4.094e+5	4779	1.227e+6	5137

Table 2: Performance comparisons between DisAC3.1 and DisAC9 on the Radio Link Frequency Assignment Problem. Note that m is the number of constraints and p is the number of agents used by both the algorithms.

Experimental results with groups A, B, C and D are summarized in Fig. 10a, Fig. 10b, Fig. 10c and Fig. 10d, respectively, where every data point in each graph is obtained by averaging the results of 20 instances. We can observe from the figures that, if the number of agents is moderate (e.g. $n/5$), then DisAC3.1 only requires approximately half #NCCCs of DisAC9, but DisAC3.1 sends slightly more messages than DisAC9. We also notice that if too many agents are used (e.g. n), performances of the distributed algorithms become worse. This phenomenon owes to the fact that too many agents would cause overloaded inter-agent communications and thus decrease the concurrency of the distributed algorithms.

In the results of groups A and B, we observe that #NCCCs and #Msgs phase transitions happen at tightness = 0.7. When constraint tightness is larger than 0.8, #Msgs drops down to zero, because with increasing constraint tightness the need for deleting domain values when enforcing AC dramatically decreases, which results in decrease of inter-agent communications.

We notice that the constraint tightness at which #NCCCs and #Msgs phase transitions happen is lower for group C than groups A and B: In group C the phase transitions are at tightness=0.6, whereas in groups A and B the phase transitions are at tightness=0.7. This phenomenon owes to the fact that higher network density implies higher number of constraints, which in turn increases the number of constraint checks and inter-agent communications. Therefore, when the network density is higher as is the case of group C, then the distributed algorithms require lower constraint tightness to reach the phase transitions.

Furthermore, we observe that the distributed algorithms require higher #NCCCs and #Msgs for group B, which is not a surprise considering the networks in group B have larger number of variables and domain size.

We now compare the scalability of DisAC3.1 with DisAC9 (cf. Fig. 10d). From the left subfigure, we observe that both algorithms show a linear behaviour w.r.t. #variables, but #NCCCs of DisAC3.1 grows slower than that of DisAC9. We also evaluated Lai and Wu's termination detection algorithm [29] for DisAC3.1, which is based on Dijkstra-Scholten termination detector and has an *optimal* message complexity.[10] In order to compare the performance of our termination detection method with Lai and Wu's termination detection algorithm [29], we modified DisAC3.1 by replacing our termination detection procedure with Lai and Wu's

[10] Note that Lai and Wu's algorithm assumes fully-connected agent communication graphs, and every agent knows the identifications of all agents in the system.

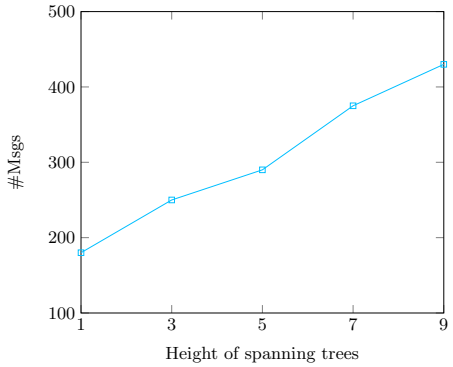


Fig. 11: Number of messages sent by agents w.r.t. height of spanning trees. We set $n = 10, d = 10, \rho = 1.0, \beta = 0.5$ and #agent $= 10$.

algorithm.[11] Result shows that all three algorithms show a sub-linear behavior w.r.t. the number of variables. The modified DisAC3.1 requires more agent messages than the other two, and DisAC9 is the best.

The main factor that affects the performance of our termination detection method is the height of spanning trees of agent communication graphs. We evaluated the #Msgs of DisAC3.1 w.r.t. the height of spanning trees. In order to generate spanning trees with height that we need, we set $\rho = 1$ for all random generated CSP instances, so that all agent communication graphs are complete. The result is shown in Fig. 11, where each data point is averaged over 20 random instances. As expected, the larger the height of spanning trees, the larger the #Msgs required by the algorithm. If the root is fixed, distributed breadth-first search [1] can be used to find the spanning trees with minimum height.

We also consider benchmark problems that are commonly used for the evaluating AC algorithms, i.e., the DOMINO problem [5,57], which is designed to study the worst case performance of AC3 and the Radio Link Frequency Assignment Problem [4,5].[12] Results are summarized in Tables 1 and 2. In Table 1, we can observe that DisAC3.1 only needs 43.2% of #NCCCs of DisAC9 on average, while #Msgs of both algorithms are comparable. Similarly, in Table 2, we can observe that DisAC3.1 only needs 30.9% of #NCCCs of DisAC9 on average, while #Msgs of both algorithms are comparable.

6.2 DisGAC3.1 vs. GAC2001/3.1

To evaluate our DisGAC3.1 algorithm, we compare it with GAC2001/3.1 by using benchmark problems from the fourth international constraint solver competition.[13] We have extracted instances of the Renault Megane configuration problem

[11] Note that we do not compare #NCCCs between the modified DisAC3.1 and DisAC3.1, as termination detector is *superimposed* on the underling computation, i.e., it cannot intervene in the underling computation [15] and thus does not affect #NCCCs.

[12] Website `https://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

[13] Website `http://www.cril.univ-artois.fr/CSC09/`

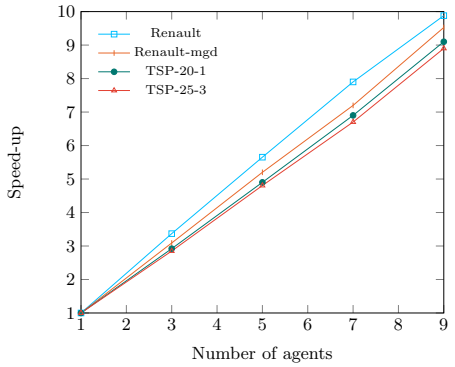


Fig. 12: Speed-up vs. number of agents on benchmark problems.

(Renault) and the traveling salesman problem (TSP) for experiments, where the Renault instances have arity of 10 and the TSP instances have arity of 3. Results are presented in Fig. 12. Here, the performance of GAC2001/3.1 is chosen as the baseline (number of agents = 1) and the speed-up (logical speed, i.e., NCCCs) is measured relative to the baseline. As one can expect DisGAC3.1 is more efficient than GAC2001/3.1 and we observe that it has a linear speed-up in the number of agents. We can also conclude that the performance of DisAC3.1 increases linearly in the number of agents, since DisAC3.1 is a special case of DisGAC3.1.

7 Conclusion

In this paper we have presented new distributed algorithms DisAC3.1 and DisGAC3.1 for efficient AC and GAC propagations. These algorithms do not assume a complete agent communication graph and release less private information of individual agents when enforcing AC and GAC. More precisely, an agent i only shares information about its communication address, its domain D_u, and its external constraint R_{uv} with another agent j, if the variable v is owned by j.

Our theoretical analysis shows that our algorithms are efficient in both time and space. Our experiments on both randomly generated instances and benchmark problems show that (i) DisAC3.1 requires significantly less number of non-concurrent constraint checks than the state-of-the-art distributed AC algorithm DisAC9, and (ii) the performance of DisGAC3.1 scales linearly in the number of agents, maximizing the merit of the distributed setting.

As technological advancements allow for solving larger problems that are highly interwoven and dependent on each other, efficiency and privacy have become critical requirements. For problems that can be modelled as distributed CSPs, we have shown that DisAC3.1 and DisGAC3.1 are efficient algorithms that can serve as good candidates for search space pruning.

Very recently, it has been shown in [28] that enforcing AC is sufficient to solve the simple temporal problem (STP) [13]. Methods developed in this paper also can be adapted to solve the multi-agent STP (MaSTP) [6,28].

Acknowledgments

The authors sincerely thank the anonymous reviewers of Autonomous Agents and Multi-Agent Systems for their very helpful comments.

Appendix

Proof of Lemma 1

Proof (1) $\Rightarrow$ (2): Given an agent i, the property *isIdle*[i] is set to `true` only in line 41 and line 44. If *isIdle*[i] was set to `true` in line 41, then $s_{ji} = r_{ij}$ for all $j = 1, \ldots, p$ and there is an agent k who sent a "domain update" message m_{ki} with timestamp s_{ki} to agent i, and the corresponding "message sent" message m_{ki}^{sent} from agent k to the root agent has been received. Since $s_{ki} = r_{ik}$, the root agent has also received the message m_{ki}^{utd}, and thus, the root agent has received agent i's first "up to date" message.

The other case, i.e., *isIdle*[i] was set to `true` in line 44, immediately implies that the root agent have received an "up to date" message from agent i and $s_{ji} = r_{ij}$ for all $j = 1, \ldots, p$.

(2) $\Rightarrow$ (3): We prove this by contradiction.

First, suppose there is at least one "domain update" message that

has not been confirmed by the root agent and let m_{ij} be such a message (sent from agent i to agent j) which bears the *earliest* timestamp s and let m_{ij}^{sent} and m_{ij}^{utd} be the corresponding "message sent" message and "up to date" message, respectively (cf. Section 3.2).

We first note that according to our assumption (2) the root agent already received agent i's first "up to date" message m^0 and $s_{ik} = r_{ki}$ for all $k = 1, \ldots, p$. Furthermore, because m_{ij} is the earliest message that has not been confirmed yet and the root agent receives messages in FIFO manner, no other messages from agent i to j sent later than m_{ij} have been confirmed by the root agent. Then it follows that both m_{ij}^{sent} and m_{ij}^{utd} have not been received by the root agent; otherwise, we would have either $s_{ij} > r_{ji}$ or $s_{ij} < r_{ji}$, contradicting our assumption.

Furthermore, agent i must have sent m_{ij} after sending m^0 to the root agent; otherwise, as the root agent receives messages from agent i in a FIFO manner, it would have received m_{ij}^{sent} before receiving m^0. Now that agent i sent m_{ij} after sending m^0, it must have sent m_{ij} when it re-entered the first inner while-loop, i.e., it received a "domain update" message m_{ki} from an agent k after sending m^0. Note that this message m_{ki} is not confirmed either, because the corresponding "up to date" message m_{ki}^{utd} from agent i to the root agent must be sent after m_{ij}^{sent}, but m_{ij}^{sent} has not been received by the root agent. As m_{ki} bears an earlier timestamp than m_{ij}, we have the contradiction that m_{ij} is an unconfirmed "domain update" message with the earliest timestamp. Thus, we can assume that all "domain update" messages have been confirmed.

Now suppose there is at least one agent running the first inner while-loop and let i be such an agent. Then, since agent i sent already its first "up to date" message to the root agent, agent i must have received a new message of type "domain update" so that it could re-enter the first inner while-loop. Let this "domain

update" message m_{ki} be from an agent k. Then agent i has not yet sent the corresponding "up to date" message m_{ki}^{utd} to the root agent, as it has not finished the first inner while-loop. Thus m_{ki} has not been confirmed. This is a contradiction to our assumption that all "domain update" messages have been confirmed.

(3) $\Rightarrow$ (1): We prove this by contradiction.

We note first that the state of *isIdle*$[i]$ $(i = 1, \ldots, p)$ will remain fixed in the future, because all "domain update" messages have been confirmed and no agent is running the first inner while-loop, and thus, no new message of type "message sent" or "up to date" will be received by the root agent in the future. Suppose there is an agent i such that *isIdle*$[i]$ = `false`. Then the agent must have received at least one "domain update" message from an agent k, otherwise $s_{ji} = r_{ij} = -\infty$ for all $j = 1, \ldots, p$ and as a consequence *isIdle*$[i]$ had to be set to `true` in line 44 after the root agent received the last "up to date" message of agent i.

Now suppose that m_{ki} is the last message that agent i received, which was sent from agent k, and let m_{ki}^{sent} and m_{ki}^{utd} be the corresponding "message sent" message and "up to date" message, respectively. Then, because m_{ki} has been confirmed, both m_{ki}^{sent} and m_{ki}^{utd} must have been received by the root agent. If m_{ki}^{sent} was received later, then the root agent must set *isIdle*$[i]$ to `true` in line 41, and if m_{ki}^{utd} was received later, the root agent must set *isIdle*$[i]$ to `true` in line 44. Both cases contradict our assumption that *isIdle*$[i]$ = `false`. Hence, *isIdle*$[i]$ = `true` for all $i = 1, \ldots, p$. □

Proof of Lemma 2

Proof We note first that at the beginning of the algorithm we have vacuously $D_u^i \supseteq D_u$, as D_u^i is a copy of D_u.

Now suppose $D_u^i \supseteq D_u$ during the process of the algorithm. Then we note that $D_u^i \not\supseteq D_u$ can only happen in line 18 of the algorithm, where there exists a variable $v \in V_i$ with $R_{uv} \in C_{ji}$ such that Revise(u, v) is true, i.e., there exists $b \in D_u^i$ for which no $a \in D_v$ exists with $(b, a) \in R_{uv}$ in which case we remove b from D_u^i. Agent i then prompts agent j to replace D_v^j with the content of D_v (line 20). Agent j—*we now switch the perspective and look at the algorithm from agent j's point of view*—then replaces D_v^j with the content of D_v (line 35) and adds among other arcs (u, v) to its queue (line 36), and applies function Revise to (u, v) (line 11). This deletes b from D_u, as there is no $a \in D_v^j$ with $(b, a) \in R_{uv}$. Thus, we have $D_u^i \supseteq D_u$. □

References

1. Awerbuch, B., Gallager, R.: A new distributed algorithm to find breadth first search trees. IEEE Transactions on Information Theory 33(3), 315–322 (1987)
2. Baudot, B., Deville, Y.: Analysis of distributed arc-consistency algorithms. Tech. rep., Université catholique de Louvain (1997)
3. Bessiere, C.: Arc-consistency and arc-consistency again. Artificial Intelligence 65(1), 179 – 190 (1994)
4. Bessiere, C., Freuder, E.C., Régin, J.C.: Using constraint metaknowledge to reduce arc consistency computation. Artificial Intelligence 107(1), 125 – 148 (1999)
5. Bessiere, C., Régin, J.C., Yap, R.H., Zhang, Y.: An optimal coarse-grained arc consistency algorithm. Artificial Intelligence 165(2), 165 – 185 (2005)
6. Boerkoel Jr, J.C., Durfee, E.H.: Distributed reasoning for multiagent simple temporal problems. Journal of Artificial Intelligence Research 47, 95–156 (2013)

7. Chandy, K.M., Misra, J.: Logics and models of concurrent systems. chap. A Paradigm for Detecting Quiescent Properties in Distributed Computations, pp. 325–341. Springer-Verlag New York, Inc., New York, NY, USA (1985)
8. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems (TOCS) 3(1), 63–75 (1985)
9. Chandy, K.M., Misra, J.: How processes learn. Distributed Computing 1(1), 40–52 (1986)
10. Chang, E.J.H.: Echo algorithms: Depth parallel operations on general graphs. IEEE Transactions on Software Engineering SE-8(4), 391–401 (Jul 1982)
11. Conry, S.E., Kuwabara, K., Lesser, V.R., Meyer, R.A.: Multistage negotiation for distributed constraint satisfaction. IEEE Transactions on Systems, Man, and Cybernetics 21(6), 1462–1477 (1991)
12. Cooper, P.R., Swain, M.J.: Arc consistency: parallelism and domain dependence. Artificial Intelligence 58(1-3), 207–235 (1992)
13. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. Artificial Intelligence 49(1-3), 61–95 (1991)
14. Dijkstra, E.W., Feijen, W.H., Van Gasteren, A.M.: Derivation of a termination detection algorithm for distributed computations. In: Control Flow and Data Flow: concepts of distributed programming, pp. 507–512. Springer (1986)
15. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Information Processing Letters 11(1), 1–4 (1980)
16. Eriksen, O., Skyum, S.: Symmetric distributed termination. DAIMI Report Series 14(189) (1985)
17. Faltings, B., Léauté, T., Petcu, A.: Privacy guarantees through distributed constraint satisfaction. In: 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology. vol. 2, pp. 350–358 (2008)
18. Grinshpoun, T.: When you say (DCOP) privacy, what do you mean? - categorization of DCOP privacy and insights on internal constraint privacy. In: Filipe, J., Fred, A.L.N. (eds.) ICAART 2012 - Proceedings of the 4th International Conference on Agents and Artificial Intelligence, Volume 1 - Artificial Intelligence, Vilamoura, Algarve, Portugal, 6-8 February, 2012. pp. 380–386. SciTePress (2012)
19. Grinshpoun, T., Tassa, T.: P-syncbb: A privacy preserving branch and bound dcop algorithm. J. Artif. Int. Res. 57(1), 621–660 (2016)
20. Gu, J., Sosic, R.: A parallel architecture for constraint satisfaction. In: International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems. pp. 229–237 (1991)
21. Hamadi, Y.: Optimal distributed arc-consistency. Constraints 7(3-4), 367–385 (2002)
22. Hassine, A.B., Ghedira, K.: How to establish arc-consistency by reactive agents. In: Proceedings of the15th European Conference on Artificial Intelligence. pp. 156–160 (2002)
23. Hassine, A.B., Ghedira, K., Ho, T.B.: New distributed filtering-consistency approach to general networks. In: Proceedings of the 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems. pp. 708–717 (2004)
24. Huang, S.T.: Detecting termination of distributed computations by external agents. In: [1989] Proceedings. The 9th International Conference on Distributed Computing Systems. pp. 79–84 (1989)
25. Hubbe, P.D., Freuder, E.C.: An efficient cross product representation of the constraint satisfaction problem search space. In: Proceedings of the Tenth National Conference on Artificial Intelligence. pp. 421–427 (1992)
26. Huffman, D.A.: Impossible objects as nonsense sentences. Machine Intelligence 6(1), 295–323 (1971)
27. Huhns, M.N., Bridgeland, D.M.: Multiagent truth maintenance. IEEE Transactions on Systems, Man, and Cybernetics 21(6), 1437–1445 (1991)
28. Kong, S., Lee, J.H., Li, S.: Multiagent Simple Temporal Problem: The Arc-Consistency Approach. In: Thirty-Second AAAI Conference on Artificial Intelligence, AAAI'18. AAAI Press (2018)
29. Lai, T.H., Wu, L.F.: An (n -1)-resilient algorithm for distributed termination detection. IEEE Trans. Parallel Distrib. Syst. 6(1), 63–78 (1995)
30. Léauté, T., Faltings, B.: Protecting privacy through distributed computation in multi-agent decision making. Journal of Artificial Intelligence Research 47, 649–695 (2013)
31. Li, S., Liu, W., Wang, S.: Qualitative constraint satisfaction problems: An extended framework with landmarks. Artificial Intelligence 201, 32–58 (2013)
32. Lynch, N.A.: Distributed algorithms. Morgan Kaufmann (1996)

33. Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence 8(1), 99 – 118 (1977)
34. Maheswaran, R.T., Pearce, J.P., Bowring, E., Varakantham, P., Tambe, M.: Privacy loss in distributed constraint reasoning: A quantitative framework for analysis and its applications. Autonomous Agents and Multi-Agent Systems 13(1), 27–60 (2006)
35. Maruyama, H.: Structural disambiguation with constraint propagation. In: ACL. pp. 31–38 (1990)
36. Mason, C.L., Johnson, R.R.: Datms: A framework for distributed assumption based reasoning. Tech. rep., Lawrence Livermore National Lab., CA (USA) (1988)
37. Matocha, J., Camp, T.: A taxonomy of distributed termination detection algorithms. Journal of Systems and Software 43(3), 207–221 (1998)
38. Mattern, F.: Algorithms for distributed termination detection. Distributed Computing 2(3), 161–175 (1987)
39. Mayo, J., Kearns, P.: Distributed termination detection with roughly synchronized clocks. Information Processing Letters 52(2), 105–108 (1994)
40. Meisels, A., Zivan, R.: Asynchronous forward-checking for discsps. Constraints 12(1), 131–150 (2007)
41. Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence 28(2), 225 – 233 (1986)
42. Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. Information Sciences 7, 95–132 (1974)
43. Nguyen, T., Deville, Y.: A distributed arc-consistency algorithm. Science of Computer Programming 30(1-2), 227–250 (1998)
44. Raynal, M.: Distributed termination detection. In: Distributed Algorithms for Message-Passing Systems, pp. 367–399. Springer Berlin Heidelberg (2013)
45. Samal, A., Henderson, T.: Parallel consistent labeling algorithms. International Journal of Parallel Programming 16(5), 341–364 (1987)
46. Savaux, J., Vion, J., Piechowiak, S., Mandiau, R., Matsui, T., Hirayama, K., Yokoo, M., Elmane, S., Silaghi, M.: DisCSPs with privacy recast as planning problems for self-interested agents. In: 2016 IEEE/WIC/ACM International Conference on Web Intelligence (WI). pp. 359–366 (2016)
47. Shavit, N., Francez, N.: A new approach to detection of locally indicative stability. In: International Colloquium on Automata, Languages, and Programming. pp. 344–358. Springer (1986)
48. Silaghi, M.c.: A comparison of distributed constraint satisfaction approaches with respect to privacy. In: In DCR. Citeseer (2002)
49. Sycara, K., Roth, S.F., Sadeh, N., Fox, M.S.: Distributed constrained heuristic search. IEEE Transactions on Systems, Man, and Cybernetics 21(6), 1446–1461 (1991)
50. Topor, R.W.: Termination detection for distributed computations. Information Processing Letters 18(1), 33 – 36 (1984)
51. Venkatesan, S.: Reliable protocols for distributed termination detection. IEEE Transactions on Reliability 38(1), 103–110 (1989)
52. Wallace, R.J., Freuder, E.C.: Constraint-based reasoning and privacy/efficiency tradeoffs in multi-agent problem solving. Artificial Intelligence 161(1), 209 – 227 (2005)
53. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction problem: Formalization and algorithms. IEEE Transactions on Knowledge and Data Engineering 10(5), 673–685 (1998)
54. Yokoo, M., Hirayama, K.: Algorithms for distributed constraint satisfaction: A review. Autonomous Agents and Multi-Agent Systems 3(2), 185–207 (2000)
55. Yokoo, M., Suzuki, K., Hirayama, K.: Secure distributed constraint satisfaction: reaching agreement without revealing private information. Artificial Intelligence 161(1), 229 – 245 (2005)
56. Zhang, Y., Mackworth, A.K.: Parallel and distributed algorithms for finite constraint satisfaction problems. In: Proceedings of the 3th IEEE Symposium on Parallel and Distributed Processing. pp. 394–397 (1991)
57. Zhang, Y., Yap, R.H.C.: Making ac-3 an optimal algorithm. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence. pp. 316–321 (2001)