Applied Intelligence manuscript No.
(will be inserted by the editor)

Multi-variable Distributed Backtracking with Sessions

René Mandiau · Julien Vion ·
Sylvain Piechowiak · Pierre Monier

Received: date / Accepted: date

Abstract The Constraint Satisfaction Problem (CSP) formalism is used to represent many combinatorial decision problems instances simply and efficiently. However, many such problems cannot be solved on a single, centralized computer for various reasons (*e.g.*, their excessive size or privacy). The Distributed CSP (DisCSP) extends the CSP model to allow such combinatorial decision problems to be modelled and handled. In this paper, we propose a complete DisCSP-solving algorithm, called *Distributed Backtracking with Sessions* (*DBS*), which can solve DisCSP so that each agent encapsulates a whole "complex" problem with many variables and constraints. We prove that the algorithm is sound and complete, and generates promising experimental results.

Keywords multi-agent system · Distributed CSP · sessions

1 Introduction

The Multi-Agent Systems community is usually interested in coordination issues, and especially with respect to interactions between distinct entities. Many definitions have been proposed in the literature, from the concept of meta-actions that define the interactions [41] to the complex notions of anticipation [36], including self-organization and emergence [12, 15, 57], and junction tree models [61, 62]. In this paper, we consider that coordination can also be defined as a search process in the "distributed problem-solving" context [26, 53, 66]. Here, the coordination can be seen as the decomposition of a problem into sub-problems, the resolution of these sub-problems and the mechanisms underlying the inter-agent exchange of partial solutions between agents until a global solution is found.

René Mandiau · Julien Vion · Sylvain Piechowiak · Pierre Monier
Univ Lille Nord de France, F-59000 Lille, France
UVHC, LAMIH, F-59313 Valenciennes, France
CNRS, UMR 8201, F-59313 Valenciennes, France
E-mail: {rene.mandiau, julien.vion, sylvain.piechowiak}@univ-valenciennes.fr

The Distributed Constraint Satisfaction Problem (DisCSP) [1, 34, 38, 66, 69] is a formalism for studying these mechanisms. Intra- and inter-agent reasoning is based on a set of relations between variables. The problem to be solved requires interactions between the agents in order to find a global solution among the local solutions. DisCSP are designed to handle problems that rely on physically distributed data and cannot be solved in a centralized way.

Most research on DisCSP considers that each agent represents exactly one variable of the problem. However, many problems are more naturally modeled with each agent representing a whole sub-problem comprised of many variables and constraints. Agents generally match the modeler's view of abstract or physical entities. For example:

– Timetabling/meeting problems [30, 47, 63] – Each participant wants to schedule several events at once. Each event is a variable, but constraints and other data for each participant must remain private and are thus encapsulated in an agent that represents a participant.
– Road traffic [17, 18] – Road traffic simulation can also be modeled using a DisCSP with several variables per agent. Each vehicle at an intersection perceives other vehicles as variables constrained by priority relationships. In this particular problem, there is no direct communication between agents and no interface variables. Each variable is specific to each agent and its representation of the environment. Constraints and variables are subsequently private. Solving the problem determines the next action to be performed; this alters the agent's representation of the world.
– Multi-robot exploration [51, 52, 56, 60] – A third application considers a fleet of autonomous robots exploring an unknown environment. These robots are agents that build a representation in which other agents are considered as variables. The constraint modeled are defined by communication ranges, obstacles and collision avoidance.
– Distributed configuration [22, 35] – The configuration of the product involves assembling different elements according to different constraints (*e.g.*, physical constraints between elements, user preferences) and delivering individualized products to the end user. In this context, the knowledge of these different elements is naturally distributed and the agents may be the different participants.

Such models can be easily transformed by splitting the sub-problems until only one variable per agent remains. However, this raises major issues. First, the user's model is implicitly altered, which can be confusing if the user is not an expert. Second, such an alteration of the model modifies not only the agents, but also their relationships since additional constraints between agents must be added to the model. Third, managing a dynamic environment (*i.e.*, an environment in which the agents, variables and constraints vary during the search) is even tougher if the sub-problems are altered. Fourth, agents allow the decision problems to be modeled with privacy or size restrictions. Splitting the problem implicitly adds restrictions where they are not needed, thus negatively altering the performance of algorithm solving.

Several algorithms dedicated to asynchronously solving DisCSP can be found in the literature; the most representative are *Multi-ABT* [32, 33], *Multi-AWC* [68] and *AFC* [21, 44]. In this paper, we propose a complete DisCSP-solving algorithm, named *Distributed Backtracking with Sessions* (*DBS*), which has multiple variables per agent. Instead of using *nogoods* to establish a context for the backtrack messages, *DBS* uses *sessions*. The nogood concept has been defined as an inconsistent combination of values. The principle is to record a nogood whenever a conflict occurs during a backtracking search. Nogoods are considered extremely costly to compute [31, 42]. We show that nogoods can be replaced with sessions without altering the completeness of the algorithm. This results in much more efficient message processing, but at a price: sessions are less informative than nogoods, and *DBS* tends to send more messages than *ABT*. The compromise we seek is to have the cost of processing these additional messages be less than that of computing the nogoods. Moreover, with these sessions, many useless messages can be filtered without affecting the completeness of the algorithm.

The paper is organized as follows. Section 2 presents the general DisCSP problem and its various solving algorithms, which will be used as a basis of comparison for *DBS*. Section 3 proposes our *DBS* algorithm. Section 4 presents the properties of the algorithm (*e.g.*, soundness, completeness, space and time complexities). Section 5 describes the various filtering techniques used to avoid the processing of useless messages. Finally, Section 6 reports our experimental results by comparing *DBS* with *Multi-ABT*, *Multi-AWC* and *AFC*, and Section 7 presents our conclusions and our prospects for future research.

2 Distributed Constraint Satisfaction Problem

We briefly present useful definitions for the DisCSP (2.1) and DisCSP-solving algorithms (2.2).

2.1 Definitions

Definition 1 (DisCSP) A Distributed CSP is a triple $(\mathscr{X}, \mathscr{A}, \mathscr{C})$.

- $\mathscr{X} = \{X_1, X_2, \dots, X_N\}$ is a finite set of N variables. A domain $\mathrm{dom}(X_i)$ is associated with each variable $X_i \in \mathscr{X}$ and denotes the finite set of values the variable can represent.
- $\mathscr{A} = \{A_1, A_2, \dots, A_m\}$ is a finite set of m agents. Each agent A encapsulates variables denoted $\mathrm{var}(A)$ (with $\mathrm{var}(A) \subseteq \mathscr{X}$), so that each variable of $\mathscr{X}$ is encapsulated in exactly one agent.
- $\mathscr{C} = \{C_1, C_2, \dots, C_e\}$ is a finite set of e constraints. Each constraint C_i involves some variables $\mathrm{var}(C_i) \subseteq \mathscr{X}$. $\mathscr{C}$ is divided into inter-agent and intra-agent constraints, denoted $\mathscr{C}_{inter}$ and $\mathscr{C}_{intra}$ respectively. An intra-agent constraint incorporates variables from a single agent.

In addition to N (the total number of variables), e (the total number of constraints) and m (the number of agents), we use the following additional metrics: d is the size of the largest variable domain and n is the maximal number of variables that can

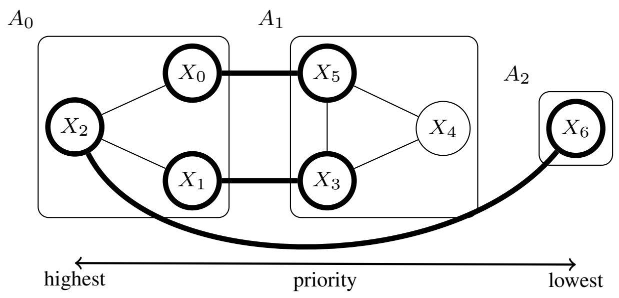


Fig. 1 A DisCSP example with three agents A_0, A_1, A_2. Fine (resp. thick) edges represent intra-agent constraints $\mathscr{C}_{intra}$ (resp. inter-agent constraints $\mathscr{C}_{inter}$)

be encapsulated in a single agent. Formally, $\forall X \in \mathscr{X}, |\mathrm{dom}(X)| \leq d$, and $\forall A \in \mathscr{A}, |\mathrm{var}(A)| \leq n$. Note that $N \leq mn$.

In the remainder of the paper, like for other existing works, we also considered binary DisCSPs. Efficiently converting non-binary constraints into binary ones is still considered an unresolved problem [13, 14]. In most cases, a polynomial decomposition is possible, even though the pruning capabilities of the decomposed model are inferior to those of the original [8].

Definition 2 (Interface Variable) A variable X, where $\exists C \in \mathscr{C}_{inter} \mid X \in \mathrm{var}(C)$ is called an *interface variable*.

Figure 1 represents a DisCSP with three agents $\mathscr{A} = \{A_0, A_1, A_2\}$, shown as a graph-like macro-structure: vertices denote variables and edges denote constraints. The DisCSP contains seven variables. Each variable is owned by one of the three agents (all variables, except X_4, are interface variables): $\mathrm{var}(A_0) = \{X_0, X_1, X_2\}$, $\mathrm{var}(A_1) = \{X_3, X_4, X_5\}$ and $\mathrm{var}(A_2) = \{X_6\}$.

Agents are given a priority order, denoted $\succ$. Without losing generality, we will assume a lexicographic order throughout this paper: $(A_i \succ A_j) \iff (i < j)$. This priority order is used to break cycles in the search for a global solution (*i.e.*, a change for agent A_1, which would imply a change for agent A_2, which would imply a change for A_1). "Better" priority orders can be determined using heuristics [2, 28].

Definition 3 (Acquaintances[1] $\Gamma^{\pm}(A)$) Agents that share a common constraint with a given agent A are called the acquaintances of A. They are divided into the sets $\Gamma^{-}(A)$ (higher acquaintances) and $\Gamma^{+}(A)$ (lower acquaintances), which denote different agents with higher and lower priority than A, respectively.

For the problem shown in Figure 1, with $A_0 \succ A_1 \succ A_2$, there are $\Gamma^{-}(A_1) = \{A_0\}$ and $\Gamma^{+}(A_1) = \emptyset$. There is no agent with priority lower than A_1 that shares a common constraint.

[1] The acquaintances are also called neighbors in constraint graphs.

2.2 Multiple variables DisCSP-solving algorithms

Several algorithms (for single-variable problems) have been proposed to solve DisCSP instances, such as Asynchronous Backtracking (*ABT*) [66], Distributed Backtracking (*DIBT*) [27, 28], Distributed Dynamic Backtracking (*DDB*) [6], Distributed Breakout (DB) [66, 67], Asynchronous Weak-Commitment (*AWC*) [65] and Asynchronous Forward-Checking (*AFC*) [21, 44]. Optimization algorithms, such as *Distributed Dynamic Branch and Bound* [5] and *DyBop* [20] also deserve to be mentioned. These algorithms can only solve DisCSP in which each agent encapsulates exactly one variable.

ABT is the reference algorithm for most work on DisCSP solving [6, 44, 54]. Thus, *ABT*, as well as *AWC* and *AFC*, are explained below. Of these algorithms, some were generalized for multi-variable problems: *Multi-ABT*, *Multi-AWC* (*AFC* is multi-variable). We are focusing on these multi-variable versions of the algorithms. We compare the results from these algorithms with the results of our *DBS* algorithm in multiple-/single- variable problem contexts.

2.2.1 ABT/ Multi-ABT

The *Asynchronous Backtracking* (*ABT*) algorithm asynchronously solves instances of DisCSP. As presented by Yokoo in 2001 [66], each agent encapsulates exactly one variable. There is an agent-variable bijection, thus making the words "agent" and "variable" interchangeable. Each agent simultaneously assigns a value to its variable, then sends a message to its lower acquaintances to inform them of the instantiation. If an agent cannot find any suitable value, it sends a backtrack request to one of the higher priority neighbors (the agent with the lowest priority in the nogood).

Let us consider agent A (and its associated variable X_A). A set $agentView(A)$ is associated with the agent, and contains the different assignments given by A's higher acquaintances. If no value is compatible with $agentView(A)$, agent A builds a backtrack request, which contains a *minimal nogood*. It takes the form $\{(A_i, v_i), \ldots, (A_k, v_k)\}$ and consists of the smallest subset of values from $agentView(A)$ for which agent A cannot instantiate its variable. This minimal nogood is sent to the lowest priority agent that appears in the nogood. *ABT* has been improved by considering only one nogood per removed value, and can be implemented with polynomial space complexity [7].

Each agent can work asynchronously. Thus, when a message is sent, it may already be obsolete, *e.g.*, when the recipient of the message has modified its current value before the message was received. The nogood is used to detect this. When agent A receives a nogood, it considers it only if its current value v as well as all values from the $agentView(A)$ are identical to the values that appear in the nogood (*i.e.*, $nogood \subseteq \{(A, v)\} \cup agentView(A)$). In some cases, agent A may receive a nogood containing some previously unknown variable, for example, from an agent A'. A link is then added between A and A' to enable them to communicate with each other.

ABT was extended to *Multi-ABT* in order to handle problems in which each agent encapsulates multiple variables and constraints [32, 33]. In *Multi-ABT*, the variables of agent A are replaced by a single variable whose values are tuples corresponding to the solutions of A's local problem. In the example shown in Figure 1, if the local CSP of agent A has two solutions – $(X_0 = 1, X_1 = 1, X_2 = 0)$ and $(X_0 = 0, X_1 = 1, X_2 = 1)$ – then the domain of new variable X_A will be comprised of these two instantiations.

2.2.2 AWC/ Multi-AWC

The *Asynchronous Weak-Commitment* (*AWC*) [65] is an *ABT* variant. A problem for *ABT* is that the convergence towards a solution becomes slow when the decisions of higher priority agents are poor: the decisions cannot be revised without an exhaustive search.

The solution proposed by Yokoo [65] consists of introducing dynamic changes in priority order: if an agent encounters a dead-end situation, its priority is increased ($max{+}1$, where max is the priority of the current highest-priority agent). In addition, the *min-conflict* heuristic [46] is used to select the next value to instantiate. The worst-case space complexity is exponential; however, in practice, this can be limited by forgetting older nogoods. With ABT, all lower priority agents must finish their search before modifying the variable that caused the inconsistency. Note that *AWC* [68] is a sound and complete algorithm for solving DisCSPs.

AWC was extended to *Multi-AWC* in order to handle DisCSP with multiple variables per agent [68]. Each agent creates virtual agents, and each virtual agent manages one local variable, thus considerably increasing the number of agents. *Multi-AWC* keeps the original inter-agent organization: all initial intra-agent constraints must be satisfied before sending messages "outside" the initial agents.

2.2.3 AFC

The *Asynchronous Forward-Checking* (*AFC*) algorithm [44] is a multi-variable per agent DisCSP-solving algorithm. This algorithm coordinates agents synchronously, while asynchronously propagating assignations using Forward Checking [29]. To describe *AFC*'s behavior, we distinguish its synchronous parts from its asynchronous parts:

Synchronous: in *AFC*, assignments are performed by one agent at a time. An agent assigns its local variables when it receives a Current Partial Assignment (CPA) message. Agents try to extend this partial assignment to obtain a valid global solution. When an agent assigns its local variables, it adds them to the CPA.

Asynchronous: when an agent sends a CPA, it sends a copy, called FC_CPA, to all unassigned agents. Agents that receive the FC_CPA message update the domains of their variables so that all their values are compatible with currently assigned variables. If a domain is emptied, then an inconsistency is detected, and a backtrack request is built. The backtrack request (a *Not_OK* message) contains a minimal nogood, which may be costly to compute. Since the CPA is unique, only one backtrack request is considered, even though several Not_OK messages have been received.

AFC detects a global solution when a CPA containing all variables of the DisCSP is created. Thus, *AFC* does not require a termination detection procedure. When a DisCSP is unsatisfiable, an agent will detect it in finite time and will order all agents to stop the search.

We implemented and tested the original version of *AFC* in the experimental section of this paper, although the algorithm has been improved by using dynamic backtracking techniques [7].

To the best of our knowledge, no studies have tried to reduce the CPU time required to build backtrack requests and process messages. Our algorithm, called *Distributed Backtracking with Sessions* (*DBS*), was designed to reduce the CPU time. Instead of using the nogoods required by most algorithms [66], *DBS* uses *sessions*. We also define a set of *filtering rules* that exploit sessions to significantly reduce the number of processed messages without modifying the completeness of the algorithm.

3 Distributed Backtracking with Sessions (*DBS*) algorithm

Our algorithm, called *Distributed Backtracking with Sessions* (*DBS*), is a complete DisCSP-solving algorithm in which each agent encapsulates a "complex" problem [68] with several variables and constraints. The main feature of *DBS* is the use of *work sessions*, which define the context of messages. *DBS* was first introduced by Doniec et al [16] with a single variable per agent.

In this section, we present some notations (3.1) and *DBS* message types (3.2), and then describe the agent data structures (3.3). Next, we explain the features of our algorithm (3.4), present the pseudo-code (3.5) and finish with an example (3.6).

3.1 Hypotheses and notations

Each agent executing *DBS* encapsulates a local CSP, which is a subproblem of the whole DisCSP. The agents' local CSPs are connected using inter-agent constraints. *DBS* requires that the agents have a static priority order, denoted $\succ$. When agent A finds a solution to its local CSP, it communicates this solution to its lower acquaintances $\Gamma^+(A)$, and then waits for incoming messages. However, agent A anticipates that its current local solution cannot be extended to a global solution, and thus continues to search for other local solutions during this time. These solutions must be distinct in terms of the agent's interface variables. Like most DisCSP algorithms (*e.g.*, sequence numbers are used in *ABT*), messages are received in the order in which they were sent for *DBS*. Moreover, *DBS* also considers that the amount of time that passes between sending and receiving a message is fast and cheap.

Backtrack requests are transmitted from $agentView$ to the highest priority agent, denoted A. A will forward the request to the faulty agent (*i.e.*, the agent that performed the wrong instantiation). This differs from *ABT*, which directly sends the request to the right agent using the minimal nogood. More messages are required in our approach, but we guarantee that no information is lost.

DBS is a fully asynchronous algorithm. The agents explore the search tree until a global solution is found. During the search, agents may receive backtrack requests pertaining to obsolete local solutions. *Work sessions* are used to filter out such messages [16].

Definition 4 (Work session) A *work session* between agent A and its lower acquaintances $\Gamma^{+}(A)$ is defined by a natural number that indicates the state of the global search from agent A's perspective.

Each agent's work session is initialized to 0. When an agent receives backtrack request M, it can be filtered out if the session number attached to message M does not correspond to the agent's work session, indicating that the message is obsolete. When agent A receives an $\langle \text{OK?} \rangle$ message with the current solution and session number from a higher priority agent, agent A closes its session and increases its session number.

The *AFC* [43, 44] and *DDBJ* [55] algorithms also make use of a concept similar to work sessions: *time-stamps* or *step-counters*. The concept of histories [58], which was used in the *AAS* algorithm, is also a similar approach. For example, in *AFC*, the time-stamp of agent A is always equal to the highest time-stamp received by agent A and is updated synchronously when *CPA* messages are sent.

3.2 *DBS* message types

In *DBS*, agents exchange $\langle \text{STOP} \rangle$, $\langle \text{OK?} \rangle$ and $\langle \text{BT} \rangle$ messages.

If one solution is found, *DBS* will reach a stable state. A global solution combining all agents' local solutions can then be built. In contrast, if the DisCSP cannot be satisfied, one agent will detect it and will send a $\langle \text{STOP} \rangle$ message to stop the algorithm.

The $\langle \text{OK?} \rangle$ message (with arguments (A, sol, s)) is used by agent A to send its current local solution $sol \in \text{sol}(A)$ to its lower acquaintances $\Gamma^{+}(A)$. The agent's current session is attached to the message. A set called $agentView(A)$ is associated with each agent, and this set contains $(\text{agent}, \text{solution}, \text{session})$ triples, thus containing an agent's higher acquaintances' current solutions (see 3.3).

The $\langle \text{BT} \rangle$ message (with arguments (A, sol, s) and $btSet$) is used by agent A' to request a backtrack to $A \in \Gamma^{-}(A')$. This backtrack request concerns agent A's current solution sol during session s. The backtrack request can be safely ignored if the session number attached to the message is not equal to agent A's current session number or agent A is already working on some other solution because some other agent has already requested a backtrack. $btSet$ is a set of triples (agent, solution, session) containing triples from an agent with higher-priority than agent A's priority. If agent A cannot find a consistent local solution according to its $agentView$, triples from $btSet$ will eventually allow agent A to continue the backtrack to an agent in $btSet$.

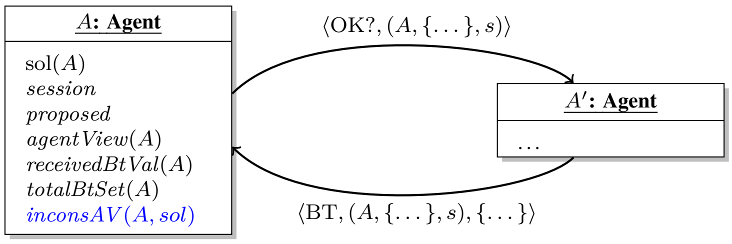


Fig. 2 Summary of DBS messages and data structures ($A \succ A'$).

3.3 *DBS* data structures

The context of agent A is defined by (Figure 2):

– Set $\mathrm{sol}(A)$ contains the set of solutions of agent A's local CSP. Each solution from $\mathrm{sol}(A)$ is like $\{(X_1 = v_1), \dots, (X_n = v_n)\}$ where $\{X_1, \dots, X_n\} = var(A)$ and satisfies the local CSP. In practice, the content of the set is computed "on the fly" (*i.e.*, local solutions are added while the other agents concurrently work on finding a global solution); obsolete local solutions are filtered out when backtrack requests are received
– Agent A's current session
– A set called *proposed*, which contains solutions already transmitted to $\Gamma^+(A)$ during the current session
– Set $agentView(A)$, which contains triples (A_i, sol_i, s_i) that are used to keep track of its higher acquaintances' current local solutions. When A receives an $\langle$OK?$, (A', sol, s)\rangle$ message, $agentView(A)$ is updated with the (A', sol, s) triple contained in the message
– Set $receivedBtVal(A)$ is used to avoid repeatedly processing the same assignment. Due to asynchronous messages, each agent can receive many BT messages for a given same assignment ($X_i = v$) of the current session
– Set $totalBtSet(A)$ contains triples (A_i, sol_i, s_i). When agent A requests a backtrack, this set is used to forward the backtrack request to other agents
– Set $inconsAV(A, sol)$ is mapped to each solution sol (with $sol \in \mathrm{sol}(A)$) and is always included in $agentView(A)$ (without consideration for the session number). This set is used to "explain" the inconsistency of sol. If $inconsAV(A, sol) = \emptyset$, then sol is consistent with the *agent view* and should be proposed as a local solution and added to the *proposed* set, as described in the following section.

3.4 Handling "complex" local problems

The well-known *ABT* and *AWC* algorithms were initially designed to handle problems in which each agent encapsulates only one variable. Later, they expanded to handle "real" local problems. They were extended using two distinct existing methods:

1. Each agent first solves its problem and finds all solutions. The problem is then reformulated so that each agent contains only one variable, whose the domain

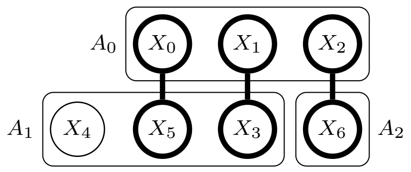


Fig. 3 DisCSP example

is composed of the set of all local solutions [32, 33]. For example, if agent A encapsulates three variables X, Y and Z, and the intra-agent constraints lead to one single solution, it can be reformulated as a single variable $X_A = \{(X = 1), (Y = 1), (Z = 0)\}$.

2. "Virtual agents" are created so that each agent encapsulates a single variable [68]. With this method, agents do not need to find all local solutions before starting the global search. However, this considerably increases the number of agents, and subsequently the number of messages exchanged.

We chose to use and improve the first method. Each agent solves its CSP and stores solutions. However, it is not necessary to find all local solutions. First, only solutions that are different from the interface variables can be considered (see *Property 1* below). Second, the global search is performed at the same time as the local search. Therefore, a global solution can be found before all agents finish solving their local problems. Those solutions that are compatible with the agent's current $agentView$ can be considered the search priority. All solutions compatible with the agent's current $agentView$ must be tested before a backtrack request is sent.

Property 1 Given two agents, A and A', a message from agent A, notifying agent A' of a change in instantiation $(\dots, X = a, \dots)$ to $(\dots, X = b, \dots)$, is worth sending if and only if $\exists X' \in \mathrm{var}(A')$ s.t. there is an inter-agent constraint involving both X and X'.

Proof A' must find solutions that are consistent with agent A, and thus must check the inter-agent constraints between A and A' (*i.e.*, formally: $\{C \in \mathscr{C}_{inter} \mid \mathrm{var}(C) \subseteq \mathrm{var}(A) \cup \mathrm{var}(A')\}$). Knowing the domains/values of $\mathrm{var}(C)$ is sufficient; the other variables are not relevant. □

When an agent changes its current local solution, it sends $\langle$OK?$\rangle$ messages to its acquaintances. According to *Property 1*, only the projection of the solution onto the appropriate interface variables is relevant. With the example shown in Figure 3, agent A_0 would inform agents A_1 and A_2 of its current solution using the following two messages:

- $\langle \text{OK?}, (A_0, \{(X_0 = 0), (X_1 = 0)\}, session = 0)\rangle$ to agent A_1
- $\langle \text{OK?}, (A_0, \{(X_2 = 0)\}, session = 0)\rangle$ to agent A_2.

If agent A receives a backtrack request pertaining to a subset of its current solution (*e.g.*, $\{(X = 0)\}$), agent A should avoid suggesting any solution containing $\{(X = 0)\}$ during its current session. For this purpose, all solutions from $\mathrm{sol}(A)$

containing $\{(X = 0)\}$ are added to the $proposed$ set (this method comes from the *OGDiBT* algorithm [3]).

When the $agentView(A)$ is modified, agent A must find a solution consistent with all triples in the set. The simplest method is to scan $\mathrm{sol}(A)$ for a consistent solution. This method requires a very large number of constraint checks. Moreover, when agent A receives a new $\langle$OK?$\rangle$ message and closes its session, all computations are lost. The $inconsAV$ map was designed to solve this problem. $inconsAV(A, \mathrm{sol})$ contains the subset of $agentView(A)$ for which sol is inconsistent. Iff $inconsAV(A, \mathrm{sol}) = \emptyset$, then sol is consistent with $agentView(A)$. The *Partially Interchangeable Neighborhoods* (PINs) defined below are used in the `checkAgentView` procedure and the `submitAssign` function described in the following section to determine if a solution is consistent with some $agentView$ elements.

For example, for agent A, $sol \notin proposed$ and $inconsAV(A, sol) = \{(A_0, \{X = 0\}), (A_1, \{Y = 0\})\}$ mean that sol has not yet been proposed in the current session, and that sol is inconsistent with $agentView(A)$. $agentView(A)$ is updated upon reception of $\langle$OK?$\rangle$ messages: $proposed$ set is then emptied, and all $inconsAV(A, sol_i)$ sets are updated. This feature prevents many constraint checks, but requires some computations to update $inconsAV$ sets. This PIN feature is borrowed from the research of Ezzahir *et al.* [19] and has been introduced into our algorithm [50]: the objective of this feature is to surmount this computation time. PINs are based on the projection (denoted $\downarrow$) of local solutions onto a constraint. Projection and PIN are defined as follows:

Definition 5 (Projection $\downarrow$) The projection of a local solution sol, constituted of a set of $(X_i = v_i)$ pairs on C, is defined as $S \downarrow C = \{(X = v) \in sol \mid X \in \mathrm{var}(C)\}$.

Definition 6 (Partially Interchangeable Neighbourhood) Two local solutions of an agent, sol_1 and sol_2, are said to be a *Partially Interchangeable Neighborhood* (PIN) in terms of inter-agent constraint C iff $sol_1 \downarrow C = sol_2 \downarrow C$. All solutions sol_i that share the same projection in terms of C are said to be in the same *PIN set*.

Property 2 Iff sol satisfies C, then all solutions from the same *PIN set* as sol in terms of C will satisfy C.

Proof Any constraint C can be defined as a set of allowed instantiations, called $\mathrm{rel}(C)$. The constraint is satisfied by a given instantiation I iff $I \in \mathrm{rel}(C)$. By definition of a CSP, a solution sol satisfies C iff $sol \downarrow C \in \mathrm{rel}(C)$. Thus, iff sol_0 satisfies C and $sol_i \downarrow C = sol_0 \downarrow C$, sol_i satisfies C. $\square$

Given agent A, for each inter-agent constraint C involving a variable of $\mathrm{var}(A)$, we define $\mathrm{PINSets}(A, C) = \{\mathrm{PIN}_0, \ldots, \mathrm{PIN}_k\}$ as the partition of $\mathrm{sol}(A)$ such that $\mathrm{PIN}_0, \ldots, \mathrm{PIN}_k$ are distinct *PIN sets*. When an agent looks for a new satisfiable local solution in terms of C, it first checks the first solution of each *PIN set* from $\mathrm{PINSets}(A, C)$. If the first solution violates C, then it can skip directly to the next *PIN set*.

Figure 4 gives an example. Agent A_2 encapsulates three variables: X_3, X_4 and X_5. Inter-agent constraints C_0 to C_2 connect A_2 to agents A_0 and A_1. For each

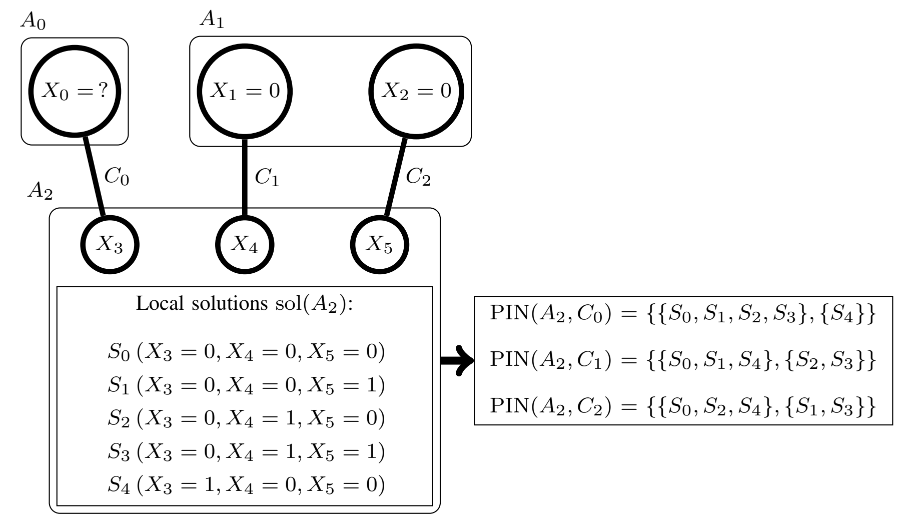

A_0
A_1
$X_0 = ?$
$X_1 = 0$
$X_2 = 0$
C_0
C_1
C_2
A_2
X_3
X_4
X_5
Local solutions sol(A_2):
S_0 ($X_3 = 0, X_4 = 0, X_5 = 0$)
S_1 ($X_3 = 0, X_4 = 0, X_5 = 1$)
S_2 ($X_3 = 0, X_4 = 1, X_5 = 0$)
S_3 ($X_3 = 0, X_4 = 1, X_5 = 1$)
S_4 ($X_3 = 1, X_4 = 0, X_5 = 0$)
$\mathrm{PIN}(A_2, C_0) = \{\{S_0, S_1, S_2, S_3\}, \{S_4\}\}$
$\mathrm{PIN}(A_2, C_1) = \{\{S_0, S_1, S_4\}, \{S_2, S_3\}\}$
$\mathrm{PIN}(A_2, C_2) = \{\{S_0, S_2, S_4\}, \{S_1, S_3\}\}$

Fig. 4 Example using of *PIN sets*

inter-agent constraint, a PIN set is created. $\mathrm{PIN}(A_2, C_1) = \{\{S_0, S_1, S_4\}, \{S_2, S_3\}\}$ means that iff S_0 does not satisfy C_1, then neither S_1 nor S_4 will satisfy it, so the algorithm may skip to S_2 in the next PIN set. Then, if S_2 does not satisfy C_0, we know from $\mathrm{PIN}(A_2, C_0)$ that S_3 will not satisfy C_0. The $agentView(A_2)$ is thus inconsistent, and a backtrack request must be built.

PINs help reduce the number of constraint checks needed to update the $inconsAV$ maps. Our experimental results (see Section 6) show that our algorithm benefits from this reduction. In practice, since using PIN sets requires computing and memorizing all local CSP solutions, they may not be practical if the local problems have too many solutions. In this case, solutions can be recalculated instead of being stored using a local solver [2] (*e.g.*, CSP4J [64]) or using an algorithm to detect conflicts (*e.g.*, QuickXplain [37]). PIN sets can then be emulated using local nogoods. In the preceding example, if the first computed solution S_0 does not satisfy C_1, then the local nogood $X_4 \neq 0$ is temporarily added to the local problem. Thus, the local solver will automatically filter out the other solutions from the same PIN set.

3.5 Algorithm

When *DBS* starts, all agents start solving their local problems The `submitAssign` function is the entry point for each agent. As soon as an agent finds a local solution, it notifies its lower acquaintances (according to *Property 1*) *via* an $\langle$OK?$\rangle$ message. As shown in Figure 5, our algorithm is composed of two listeners, three procedures and one function that communicate with each other (Σ are listeners and ⊐ are procedures or functions that can send messages). In the following algorithms, $self$ is used to identify the current agent executing the code.

[2] Local solver means here a mechanism (like constraint propagator) to find a solution for the local CSP.

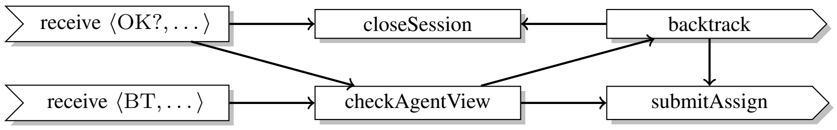


Fig. 5 Listener/procedure/function calls in *DBS*

Listener 1 (receives ⟨OK?⟩ message) is used to update the $agentView$ set (Line 1), to close the agent's session (Line 2) and to start the search for a solution consistent with the $agentView$ (Line 3).

```
Listener 1: when receives ⟨OK?, (A, sol, s)⟩
1 update $agentView$ with $(A, sol, s)$;
2 closeSession();
3 checkAgentView($A$, OK?);
```

Listener 2 (receives ⟨BT⟩ message) is used to handle backtrack requests. First, the algorithm checks whether or not the request is in the current session and whether or not it is not obsolete (Line 1). $receivedBtVal$, $totalBtSet$, $proposed$ and $currentSol$ are updated (Lines 2 to 5). Note that $totalBtSet$ is expanded with elements from $btSet$ that pertain to variables that are not yet assigned in $totalBtSet$ (this set is updated by the $\uplus$ operator). Finally, the agent starts searching for another consistent solution by calling `checkAgentView` (Line 6).

```
Listener 2: when receives ⟨BT, (A, (X = v), s), btSet⟩
1 if $A = self \wedge s = currentSession \wedge (X = v) \notin receivedBtVal$ then
2     $receivedBtVal \leftarrow receivedBtVal \cup \{(X = v)\}$ ;
3     $totalBtSet \leftarrow totalBtSet \uplus btSet$ ;
4     $proposed \leftarrow proposed \cup \{sol \in \mathrm{sol}(self) \mid (X = v) \in sol\}$ ;
5     if $(X = v) \in currentSol$ then $currentSol \leftarrow$ null;
6     checkAgentView(null, BT) ;
```

The $\uplus$ operator is used to add new assignment information to $totalBtSet$ without altering previous data. In other words, only new assignments are added to the data structure, but already instantiated variables are ignored. Formally, the $\uplus$ operator is defined as following:

Definition 7 **($\uplus$)** Let T_1 and T_2 be two sets of triples $(A, (X = v), s)$, where A is an agent, X is a variable, v is a value from $\mathrm{dom}(X)$, and s is a session number. With "_" denoting any value or session, we define:

$$T_1 \uplus T_2 = T_1 \cup \{(A, (X = v), s) \in T_2 \mid \nexists (A, (X = \_), \_) \in T_1\}$$

Procedure closeSession

```
1 $currentSol \leftarrow$ **null** ;
2 $currentSession \leftarrow currentSession + 1$ ;
3 $receivedBtVal \leftarrow \emptyset$;
4 $proposed \leftarrow \emptyset$ ;
5 foreach $sol \in \mathrm{sol}(self)$ do update $inconsAV(self, sol)$
```

The `closeSession` procedure mainly involves reinitializing the data structures in order to search for solutions consistent with the new $agentView$. The current solution is reset (Line 1) and the session number is incremented (Line 2). Next, the $receivedBtVal$ set is emptied (Line 3). This data structure is used to save information pertaining to local solutions that were inconsistent according to the agent's lower acquaintances. All previously proposed solutions are now obsolete, so the $proposed$ set is also emptied (Line 4). Finally, the $inconsAV$ map is updated according to the new $agentView$ (Line 5).

Function submitAssign: boolean

```
1 select $sol \in (sol(self) - proposed) \mid \mathrm{consistent}(sol, agentView)$ ;
2 if $sol$ is null then return false ;
3 $currentSol \leftarrow sol$ ;
4 $proposed \leftarrow proposed \cup \{sol\}$ ;
5 foreach $A \in \Gamma^{+}(self)$ do
6     $subset \leftarrow \{X \mid \exists (C, X, Y) \in \mathscr{C}_{inter} \times \mathrm{var}(self) \times \mathrm{var}(A) \mid \{X, Y\} \subseteq \mathrm{var}(C)\}$;
7     send $\langle \text{OK?}, (self, currentSol \downarrow subset, currentSession)\rangle$ to $A$ ;
8 return true;
```

The `submitAssign` function is used to select a solution consistent with the $agentView$ (Line 1). The solution is then transmitted to the agent's lower acquaintances according to *Property 1* (Lines 5 to 7).

Procedure checkAgentView(A_k: Agent, $typeOfMessage$: MsgType)

```
1 if $typeOfMessage = \text{OK?} \wedge (\forall sol \in sol(self), sol \in$
  $proposed \vee \neg consistent(sol, \{(A, \_, \_) \in agentView \mid A \succeq A_k\}))$ then
2     backtrack($A_k$, $typeOfMessage$) ;
3 else if $\neg submitAssign()$ then
4     backtrack(null, $typeOfMessage$);
```

The `checkAgentView` procedure is used to verify whether or not a given solution is consistent with the $agentView$ (Line 1). If solutions exist, the `submitAssign` function is called (Line 3). If no solution exists, or if `submitAssign` does not find any further solution, a backtrack is requested (Lines 2 & 4).

The `backtrack` request is called in `checkAgentView` and is used to send backtrack messages if necessary.

Procedure backtrack(A_k: Agent, *typeOfMessage*: MsgType)

```
1  if $A_k \neq$ null then
2      select
       $T = (A_k, (X = v), s) \in agentView \mid \forall (A', (X' = \_), \_) \in agentView, A_k \succeq A'$ ;
3      if $T \neq$ null then
4          $btSet \leftarrow \{(A' \succ A_k, (X' = v'), s') \in agentView \uplus totalBtSet\}$ ;
5  else
6      if $typeOfMessage = $ OK? then
7          select
           $T = (A, (X = v), s) \in agentView \mid \forall (A', (X' = \_), \_) \in agentView, A \succeq A'$;
8          if $T \neq$ null then
9              $btSet \leftarrow \{(A' \succ A, (X' = v'), s') \in agentView \uplus totalBtSet\}$;
10     else
11         $btSet \leftarrow agentView \uplus totalBtSet$;
12         select $T = (A, (X = v), s) \in btSet \mid \forall (A', (X' = \_), \_) \in btSet, A' \succeq A$;
13         if $T \neq$ null then
14             $btSet \leftarrow btSet - \{(A, (X = v), s)\}$ ;
15 if $T =$ null then
16     broadcast $\langle STOP \rangle$ to acquaintances and terminate;
17 send $\langle \mathrm{BT}, T, btSet \rangle$ to $A$;
18 $totalBtSet \leftarrow totalBtSet - btSet - \{T\}$;
19 if $T \in agentView$ then
20     $agentView \leftarrow agentView - \{T\}$;
21 else if $typeOfMessage = $ BT then
22     closeSession() ;
23     submitAssign() ;
```

When it is possible to find triple $T = (A, (X = v), s)$ in $agentView$ or $agentView \uplus totalBtSet$ (Line 2, 7 or 12), the search for T depends on input agent A_k and input message type $typeOfMessage$. If $A_k \neq$ **null**, then triple $T = (A, (X = v), s)$ is found in $agentView$ with $A = A_k$. The $A_k \neq$ **null** case happens when the $typeOfMessage$ received by `checkAgentView` is OK? and all the local solutions already proposed are inconsistent with $agentView$ (Lines 2 to 4). If $A_k =$ **null** then there are two possible cases. In the first case, $typeOfMessage =$ OK?, the agent tries to find triple $T = (A, (X = v), s)$ in $agentView$ from higher acquaintances (Lines 6 to 9). In the second case, $typeOfMessage =$ BT, the agent tries to find a triple $T = (A, (X = v), s)$ in $agentView \uplus totalBtSet$ from lower acquaintances (Lines 10 to 14). In all the previous cases, the $btSet$ of the current agent is updated (Line 4, 9 or 14).

A backtracking message in accordance with selected triple T (Line 17) is then sent to agent A. Set $totalBtSet$ is then updated (Line 18). If T is already in $agentView$, then it is removed from this set (Lines 19 and 20). Otherwise, if $typeOfMessage =$ BT, then the agent closes its session and sends a new assignment to its lower acquaintances (Lines 21 to 23).

If the DisCSP is inconsistent, then it is not possible to find a triple T, the $\langle STOP \rangle$ message is broadcasted and finally global DisCSP problem solving will terminate (Lines 15 and 16).

3.6 Example

Figure 6 shows a distributed graph coloring problem, with three agents A_0, A_1 and A_2. Each agent encapsulates three variables, each of which can take one of two colors: ○ or ●. The goal is to color each variable so that two adjacent variables do not have the same color. Two variables are called adjacent if they are connected by the same constraint. The priority of agents is $A_0 \succ A_1 \succ A_2$. Messages sent by *DBS* are shown on Figure 7.

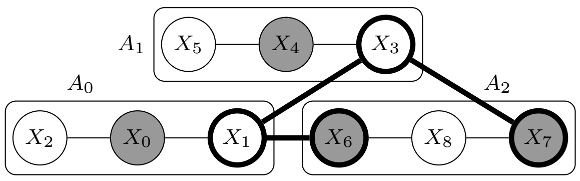
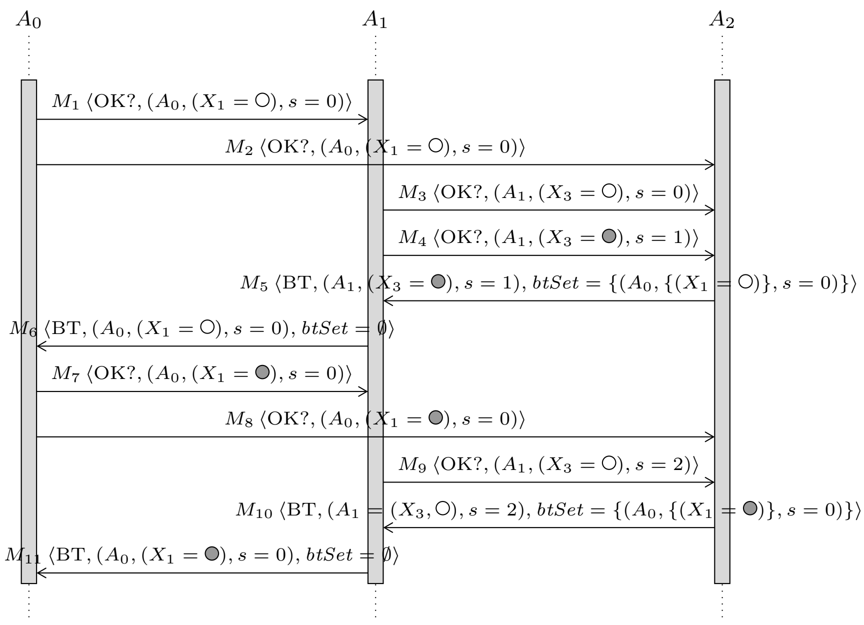


Fig. 6 A distributed graph coloring problem



M_1 ⟨OK?, $(A_0, (X_1 = ○), s = 0)$⟩

M_2 ⟨OK?, $(A_0, (X_1 = ○), s = 0)$⟩

M_3 ⟨OK?, $(A_1, (X_3 = ○), s = 0)$⟩

M_4 ⟨OK?, $(A_1, (X_3 = ●), s = 1)$⟩

M_5 ⟨BT, $(A_1, (X_3 = ●), s = 1), \mathit{btSet} = \{(A_0, \{(X_1 = ○)\}, s = 0)\}$⟩

M_6 ⟨BT, $(A_0, (X_1 = ○), s = 0), \mathit{btSet} = \emptyset$⟩

M_7 ⟨OK?, $(A_0, (X_1 = ●), s = 0)$⟩

M_8 ⟨OK?, $(A_0, (X_1 = ●), s = 0)$⟩

M_9 ⟨OK?, $(A_1, (X_3 = ○), s = 2)$⟩

M_{10} ⟨BT, $(A_1 = (X_3, ○), s = 2), \mathit{btSet} = \{(A_0, \{(X_1 = ●)\}, s = 0)\}$⟩

M_{11} ⟨BT, $(A_0, (X_1 = ●), s = 0), \mathit{btSet} = \emptyset$⟩

Fig. 7 Messages sent by the agents to solve the problem in Figure 6

Each agent solves its local CSP and finds a local solution: A_0 finds the solution $(X_0 = ●, X_1 = ○, X_2 = ○)$, A_1 finds $(X_3 = ○, X_4 = ●, X_5 = ○)$, and A_2 finds $(X_6 = ●, X_7 = ●, X_8 = ○)$.

A_0 notifies its acquaintances through messages M_1 and M_2. The $agentView$ of A_1 and A_2 are updated with $(A_0, (X_1 = \circ), s = 0)$. The current session of A_1 and A_2 are incremented to 1.

Simultaneously, A_1 notifies its acquaintances through message M_3. The $agentView$ of A_2 is updated with $(A_1, (X_3 = \circ), s = 0)$. Now, the $agentView$ of A_2 equals $\{(A_0, (X_1 = \circ), s = 0), (A_1, (X_3 = \circ), s = 0)\}$ and its current session is incremented to 2. The local solution of A_2 is consistent with its $agentView$.

However, A_1 detects an inconsistency ($X_1 = \circ$ and $X_3 = \circ$) between its local solution and its $agentView$. A_1 finds another local solution (($X_3 = \bullet, X_4 = \circ$, $X_5 = \bullet$). It sends the message M_4.

M_4 is received by A_2. Consequently its $agentView$ is updated with $(A_1, (X_3 = \bullet), s = 1)$ and $(A_1, (X_3 = \circ), s = 0)$ is removed, *i.e.*, the $agentView$ of A_2 equals $\{(A_0, (X_1 = \circ), s = 0), (A_1, (X_3 = \bullet), s = 1)\}$. Its current session is incremented to 3.

A_2 detects an inconsistency ($X_3 = \bullet$ and $X_7 = \bullet$) between its local solution and its $agentView$ and cannot find another local solution. It thus sends a backtrack request M_5 to the agent A_1. In the `backtrack` procedure (Lines 1 to 4), the selected triple T is $(A_1, (X_3 = \bullet), s = 1)$ and $btSet = \{(A_0, \{(X_1 = \circ)\}, s = 0)\}$. At the end of `backtrack` procedure, $(A_1, (X_3 = \bullet), s = 1)$ is removed from $agentView$ (*i.e.*, $\{(A_0, (X_1 = \circ), s = 0)\}$ remains).

A_1 receives M_5 (Listener 2). A_1 is still in session 1 and $(X_3 = \bullet)$ has not yet been received ($(X_3 = \bullet) \notin receivedBtVal$. To avoid repeatedly processing the same backtrack request from other agents, $(X_3, \bullet)$ is stored in $receivedBtVal$. If there are no more local solutions, the backtrack request will be forwarded to a higher acquaintance. $totalBtSet$ is updated (*i.e.*, $totalBtSet = \{(A_0, \{(X_1 = \circ)\}, s = 0)\}$) for this purpose. At this step, we know that all local solutions with $(X_3 = \bullet)$ become inconsistent, so the proposed set is updated accordingly. The current solution is discarded. The `checkAgentView` procedure is called with $A_k = null$ and $typeOfMessage = \text{BT}$.

With these parameters, in `checkAgentView` the `submitAssign` procedure is executed on line 3. Since the unique solution consistent with its $agentView$ has already been proposed in session 1, `submitAssign` immediately returns false.

Finally, the `checkAgentView` procedure ends with a call for the `backtrack` procedure with $A_k = null$ and $typeOfMessage = \text{BT}$. Lines 10 to 14 of the `backtrack` procedure are executed to forward the backtrack request. In the current state of A_1, we have $agentView = totalBtSet = \{(A_0, \{(X_1 = \circ)\}, s = 0)\}$, which is thus assigned to $btSet$. Line 12 selects the agent in $btSet$ with the highest priority; we have then $T = (A_0, \{(X_1 = \circ)\}, s = 0)$ and the tuple is removed from $btSet$, which is now empty. M_6 is now sent to A_0, and T is removed from $totalBtSet$ and $agentView$ for A_1, which are now empty.

A_0 receives this message. Like for message M_5 above, A_0 uses $proposed$ to store the information that all solutions with $(X_1 = \circ)$ should not be proposed again and calls `checkAgentView`.

However, there is another consistent local solution: $(X_0 = \circ, X_1 = \bullet, X_2 = \bullet)$. `submitAssign` is thus executed normally: the current solution is switched to the new local solution, the OK? messages M_7 and M_8 are sent to A_1 and A_2 respectively, and true is returned, which ends `checkAgentView`.

Since A_2 receives M_8, it updates its $agentView$ from $\{(A_0, (X_1 = \circ), s = 0)\}$ to $\{(A_0, (X_1 = \bullet), s = 0)\}$, increments its session number and calls `checkAgentView` (with $A_k = A_0$ and $typeOfMessage$ = OK?). There is a consistent local solution for A_2 with $(X_6 = \circ, X_7 = \circ, X_8 = \bullet)$.

Since A_1 receives M_7, it closes its session, which is incremented from 1 to 2, and selects a consistent solution: $(X_3 = \circ, X_4 = \bullet, X_5 = \circ)$. It notifies A_2 of this new solution with M_9.

Since no solution is consistent with its $agentView$ (*i.e.*, $\{(A_0, (X_1 = \bullet), 0), (A_1, (X_3 = \circ), 2)\}$), A_2 sends a backtrack request M_{10} to A_1. This is the same situation as when A_1 receives M_5. A_1 propagates the backtrack request to A_0 with message M_{11}. A_0 receives this message. Since its two local solutions were already proposed, A_0 affirms that there is no solution for the DisCSP and sends a ⟨STOP⟩ message to all agents to terminate the search.

4 Properties of the *DBS* Algorithm

In this section, we prove that *DBS* is sound (4.1), complete (4.2) and always terminates in finite time (4.3). Next, we discuss space complexity (4.4) and worst-case time complexity (4.5).

4.1 Soundness

Theorem 1 DBS *is sound: whenever DBS halts and reports a solution, all variables of all agents are assigned a value and all inter- and intra-agent constraints are satisfied.*

Proof When *DBS* detects a solution, we assume that all agents are in a stable state, which means that no more messages are sent. If an agent detects a violated constraint, then it will send a backtrack message. This would refute our assumption. This proof is similar to proofs of soundness for most *ABT* variants (*e.g.*, [7]). □

4.2 Completeness

Theorem 2 DBS *is complete.*

The following proof is based on completeness proofs *DDB* [6], *ABT* [66] and *DBS* with single-variable problem [49] algorithms. We prove the completeness of *DBS* in three steps:

1. We prove that a simplified version of *DBS*, called *DBS*^{ng}, in which all agents save all forbidden instantiations (which are equivalent to *ABT*'s nogoods) and encapsulate only one variable, is equivalent to *ABT*, and thus is complete.
2. We show that saving all forbidden instantiations is not necessary, and we show that *DBS*, restricted to one variable per agent, keeps all the properties of *DBS*^{ng} in terms of completeness.
3. We prove that encapsulating multiple variables per agent does not affect completeness.

4.2.1 DBS^{ng} *is complete*

DBS^{ng} is a variant of *DBS* in which all forbidden instantiations are stored in a set called $FISet$. We remind the reader that the full *DBS* algorithm temporarily saves selected forbidden values in $receivedBtVal$. We prove that *DBS*^{ng} is equivalent to *ABT*.

1. Saving forbidden instantiations: in *ABT*, when an agent receives a nogood, it adds it to its nogoods list. In *DBS*^{ng}, when agent A receives a backtrack request (*i.e.*, $\langle \mathrm{BT}, (A, value, session), btSet\rangle$), it checks its $agentView(A)$ and deduces a full forbidden instantiation in the following manner: if $agentView(A) = \{(X_1, v_1, s_1), \ldots, (X_k, v_k, s_k)\}$, the forbidden instantiation $\{(X_1, v_1, s_1), \ldots, (X_k, v_k, s_k), (X_A, value, session)\}$ is added to $FISet$, which is never emptied. *DBS*^{ng} saves the forbidden instantiations in the same way as *ABT* handles nogoods.

2. Context of a backtrack message: unlike *ABT*, which attaches nogoods to backtrack messages, *DBS*^{ng} uses sessions. *ABT* assumes that agent A receives the following nogood: $\{(X_1, v_1), \ldots, (X_k, v_k), (X_A, value)\}$. *ABT* would detect an inconsistency and request a backtrack iff both of the following two conditions are met:

(a). $value$ is agent A's current value, and
(b). $\forall (X, v) \in nogood$, either $(X, v) \in agentView(A)$ or $\nexists (X, \_) \in agentView(A)$.

With *DBS*^{ng}, if agent A receives the message $\langle \mathrm{BT}, (A, value, session), btSet\rangle$, it would consider the backtrack request if both of the following conditions are met:

(a). $value$ is agent A's current value, and
(b). the session number attached to the message is equal to agent A's session number.

The items (a) for both *ABT* and *DBS*^{ng} are identical. For *DBS*^{ng}, item (b) will be met iff agent A has not received any $\langle \mathrm{OK?}\rangle$ message since it last sent its current $value$. In this case, $agentView(A)$ is not modified and is equivalent to item (b) in *ABT*. This proves that *DBS*^{ng} and *ABT* use a similar context.

3. Selecting the recipient of a backtrack message: *ABT* and *DBS*^{ng} do not always select the same recipient when they send a backtrack request. In both algorithms, when agent A receives an $\langle \mathrm{OK?}\rangle$ message from agent A', it updates $agentView(A)$ accordingly. If no solution can be found:

In *ABT*, agent A builds a nogood for each subset of $agentView(A)$ for which no solution exists. A backtrack message is then sent to the lowest priority agents that appear in the nogood list.

In *DBS*^{ng}, if agent A does not find any solution according to the higher priority agents other than A' appearing in $agentView(A)$ (see Line 1 of the `checkAgentView` procedure), the backtrack message is sent to A'. If a solution exists, the backtrack message is sent to the lowest priority agent appearing in $agentView(A)$.

ABT sends the backtrack message directly to the conflicting agent, whereas *DBS*^{ng} sends the message either to conflicting agent A' or to a lower priority agent than A'.

4. Adding relationships: During the search, the *ABT* algorithm adds links dynamically in order to forward backtrack requests to agents that do not appear in the agent's $agentView$. In the worst case,this is actually equivalent to building a complete communication network between all agents. *DBS*^{ng} always considers that all agents can be reached.

Upon reception of a $\langle \mathrm{BT} \rangle$ message by agent A, *ABT* will forward it in the following way. If the nogood attached to the backtrack message contains an agent A' that agent A does not know, it adds agent A' to its $agentView(A)$ and starts searching for a compatible solution. If no solution exists, agent A builds a nogood for each inconsistent subset of its $agentView(A)$. For each nogood, agent A sends the backtrack request to the lowest priority agent of its higher acquaintances and then removes it from $agentView(A)$.

In *DBS*^{ng}, when agent A receives a backtrack request with an attached $btSet$ containing $(A', X = v, s)$ with $A' \notin agentView(A)$, it does not add this information to its $agentView$. This would not affect the solutions found by the agents, as no constraint exists between X and any variable of agent A. However, if agent A does not find any solution after the backtrack request, it forwards the request to the lowest priority agent from $agentView(A) \uplus totalBtSet$, which contains information about A'. The behavior is thus the same as that of the *ABT* algorithm's.

To finish the comparison between *ABT* and *DBS*^{ng}, we compare the contents of a backtrack message. To transmit a message from agent A to agent A', *ABT* attaches a nogood that includes the last known value v' from the variable in agent A' obtained from $agentView(A)$. *DBS*^{ng} sends the tuple $(A', X_{A'} = v_{A'}, s_{A'})$ from $agentView(A) \uplus (btSet \uplus totalBtSet)$ ($btSet$ comes from the backtrack message that caused agent A to backtrack), along with other information that will permit agent A' to forward the backtrack message if necessary.

This shows that *DBS*^{ng}, a variant of *DBS* that saves all forbidden instantiations, strictly has the same behavior as *ABT* (*i.e.*, nogoods, messages content and recipient, and dynamic relationships). Since *ABT* is complete [66], so is *DBS*^{ng}. $\square$

4.2.2 Single-variable DBS *is complete*

DBS has the same properties as *DBS*^{ng}. The only difference is that *DBS* removes obsolete forbidden instantiations. It is quite clear that removing forbidden instantiations cannot prune the search space, so no solution can be lost.

4.2.3 Multi-variable optimizations

We prove that the optimizations that added to efficiently handle multi-variable local problems in *DBS* do not affect the algorithm's completeness. *DBS* only considers local solutions that differ in terms of interface variables: an agent that wants to transmit its local solutions to its acquaintances does not send the whole solution (see *Property 1*). When an agent receives a backtrack request, the instantiation of the related interface variable is altered. This method does not change the completeness of the algorithm [3]. Agents use PINs to search for consistent local solutions, which does not affect completeness [19]. *DBS* saves the origin of the inconsistency of each local solution. With the PINs, all solutions are still tested; only the checking method is modified. This does not affect the completeness of the algorithm.

The three previous steps show the completeness for *DBS*. □

4.3 Termination

In this section, we prove that removing obsolete forbidden instantiations cannot lead to infinite loops, which proves that *DBS* always terminates.

Lemma 1 *Let agent A_0 be the highest priority agent. Agent A_0 can never be trapped in an infinite loop, even though* DBS *removes obsolete forbidden instantiations.*

Proof DBS reinitializes its $\mathit{receivedBtVal}$ set, which contains forbidden instantiations, only when an $\langle \text{OK?} \rangle$ message is received from a higher priority agent, or when an agent sends a backtrack request to a higher priority agent. Since agent A_0 is the highest priority agent, these events never occur and items in $\mathit{receivedBtVal}$ are never removed. Moreover, as the number of solutions of A_0 is finite, the size of $\mathit{receivedBtVal}$ is bound in $O(d^n)$. Since A_0 cannot reinitialize $\mathit{receivedBtVal}$, and since it can receive only a finite number of instantiations, it cannot be in an infinite loop. □

Lemma 2 *If given in the priority order considered by* DBS*, the first $(k-1)$ agents are not in an infinite loop, reinitializing the $\mathit{receivedBtVal}$ set cannot lead agent A_k into an infinite loop.*

Proof Let us assume that agent A_k sends and receives messages infinitely. This means that agent A_k loses forbidden instantiations that it should have kept, since its higher acquaintances have modified their local solution. Agent A_k removes forbidden instantiations when:

- Agent A_k receives an $\langle \text{OK?} \rangle$ message from a higher acquaintance; this increments agent A_k's session.
- Agent A_k sends a backtrack message to higher acquaintance A that does not appear in $\mathit{agentView}(A_k)$, since A has changed its value; this increments agent A_k's session.

Since we assumed that agents A_0 to A_{k-1} are not in an infinite loop, this means that they will stabilize, and ultimately stop sending $\langle OK \rangle$ messages. Thus, agent A_k will never reach an infinite loop. □

Theorem 3 DBS *terminates.*

Proof Lemmas 1 and 2 recursively prove that agents cannot reach an infinite loop, even though *DBS* removes obsolete instantiations. Thus, *DBS* terminates. □

4.4 Space complexity

Theorem 4 *The space complexity of each agent running* DBS *is in* $O(Nd^n)$ *(where* N*: total number of variables,* n*: maximum number of variables for each agent and* d*: maximum domain size).*

Proof The space requirements of a given agent depend on its stored context. The complexity is capped by the $inconsAV$ data structure. It may contain up to N triples for each one of the $O(d^n)$ local solutions. Thus, the space complexity of each agent is in $O(Nd^n)$. □

The $inconsAV$ data structure can consume a lot of memory when the number of variables is high. We plan to improve this in the future.

4.5 Time complexity

Theorem 5 *For a given agent* A*,* e_A *is the number of intra-agent constraints,* n_A *is the number of variables of domain size less than* d_A *encapsulated in* A*, and* e_{inter} *is the number of inter-agent constraints. The worst-case time complexity of* DBS *is:*

$$O(\sum_{A \in \mathscr{A}} e_A d_A^{n_A} + e_{inter} d^N)$$

Proof In the worst case, time complexity is the sum of the time required to solve local CSP and the time of the global search. We consider that messages are sent and received in constant time. The first step involves finding all solutions of all local CSPs. For a given agent A, the complexity is $O(e_A d_A^{n_A})$, *i.e.*, the first term. The second step involves performing the global search of all agents. Each agent A can consider $O(d^{n_A})$ solutions, and in the worst case, *DBS* must check the e_{inter} inter-agent constraints for all the $O(\prod_{A \in \mathscr{A}} d_A^{n_A}) \in O(d^N)$ combinations of local solutions. □

Since *DBS* is an asynchronous distributed algorithm, most of the work can be done at the same time, especially the search for local solutions. Assuming that we have one computing unit per agent with no additional cost, we can divide up the search for local solutions among all CPUs. The *wallclock time* of the first term $O(\sum_{A \in \mathscr{A}} e_A d_A^{n_A})$ can then be replaced by $O(\max_{A \in \mathscr{A}} e_A d_A^{n_A})$.

We can observe that $\forall A, d_A \leq d$ and $n_A \leq n$, and $\sum_{A \in \mathscr{A}} e_A = e_{intra}$, which leads to the simplified formulation for the worst-case time complexity:

$$O(e_{intra} d^n + e_{inter} d^N)$$

5 Message filtering

This section describes several techniques that can be used to filter out obsolete messages (*i.e.*, deleting them before Listener 1 or 2), with no impact on the completeness of the algorithm. The idea of global message processing to reduce the number of useless computations was also proposed by Brito & Meseguer [9]. These authors propose partitioning the set of messages into sublists. Messages are then chosen according to the sublist they appears in. The *MHDC* algorithm [59] also proposed a concept similar to filters: "compactors". When the inbox size exceeds a given threshold, the filtering procedure takes priority and removes useless messages. These approaches may be used to reduce the number of messages transmitted by existing algorithms and to propose enhanced evaluations.

Our approach differs from these in the way we apply the filters. The filtering process is applied as soon as an agent receives a message. The message is kept in the inbox only if no filter could remove it. Filtered messages are messages that remain in the agents' inboxes. We call the inbox of a given agent A, $IB(A)$.

Filter 1 **(F_1)** If more than one $\langle$OK?$\rangle$ message from a given agent A' sits in $IB(A)$, only the latest $\langle$OK?$\rangle$ message should be considered, and the other messages can be filtered out.

Since the work sessions of an agent can never decrease, if agent A' sends several $\langle$OK?$\rangle$ messages to agent A, then the latest $\langle$OK?$\rangle$ message's session number will be greater than or equal to the session number of all other $\langle$OK?$\rangle$ messages sent by A'.

Suppose that several $\langle$OK?$\rangle$ messages from A' remain in in $IB(A)$. Let the most recent message be $m_{last} = \langle \text{OK?}, (A', sol, s)\rangle$. The other messages from agent A' will be $\langle \text{OK?}, (A', sol', s')\rangle$ with either $sol \neq sol'$ or $s \neq s'$.

Consider the three different cases:

$s' < s$: the context of a message is defined by a session number. The messages with an obsolete session number are ignored. If agent A processes this message of session number s', all other messages with $s < s'$ will become obsolete and can be filtered out.

$sol = sol'$ and $s = s'$: this case cannot occur since an agent cannot propose the same instantiation during the same session.

$sol \neq sol'$ and $s = s'$: we assume that the local search will not give the same solution twice. If agent A' changed its solution sol' to sol, this was due to a backtrack request for sol', and thus sol' becomes obsolete and can be ignored. Since we assumed that each agent performs its local search using a tree-like search with static variable ordering, we know that solutions will be given in order. Given *Property 1*, this means that a single instantiation cannot be proposed more than once.

Filter 2 **(F_2)** If agent A has several backtrack messages and at least one $\langle$OK?$\rangle$ message in $IB(A)$, all backtrack messages can be removed.

Let us suppose that agent A processes the $\langle$OK?$\rangle$ message first. Agent A's work session will be incremented. All backtrack messages in $IB(A)$ are obsolete since they pertain to a local solution from an past agent A session.

Filter 3 **(F_3)** If agent A has sent a backtrack message to agent A', and this backtrack message appeared in $agentView(A)$, then until A has received an $\langle \text{OK?} \rangle$ message from agent A', it can remove all backtrack messages contained within $IB(A)$.

Since agent A sent a backtrack message to agent A', it will necessarily receive, after a finite time, an $\langle \text{OK?} \rangle$ message from agent A'. This message will make all backtrack messages from agent A's lower acquaintances obsolete.

Filter 4 **(F_4)** If agent A receives several $\langle \text{OK?} \rangle$ messages, it processes the messages from the highest priority agents first.

This "filter" actually does not remove the messages from the agents' inboxes. F_4 is a heuristic rather than a filter but we call it a "filter" for consistency reasons. Since *DBS* is complete and asynchronous, it works regardless of the order in which the messages are received. However, we still assume the following hypothesis: For a given pair of agents, all messages are received in the order in which they were sent.

6 Experiments

We describe the various hypotheses (6.1) for the different experiments in the following three sections (6.2 to 6.4).

6.1 Hypotheses for the experiments

Our algorithm was tested with graph coloring problems, which were also used to evaluate the performances of *Multi-AWC* [68]. Each problem was randomly generated according to parameters $\langle m, n, d, e \rangle$. Following the standard notation, m is the number of agents, n is the number of variables per agent, d is the number of values/colors, and e is the total number of constraints.

We used the JADE multi-agent platform [4]. This well-known multi-agent platform automatically manages agent distribution, scheduling and message exchange. We re-implemented each algorithm on our platform (in this paper, we refer to the initial *ABT* version, which keeps all nogoods). We also compared *DBS* with different stripped down versions:

- *DBS*$^{\overline{PF}}$: a version of *DBS* from which the PINs and filters have been removed
- *DBS*$^{\overline{P}}$: a version of *DBS* without the PINs
- *DBS*$^{\overline{F}}$: a version of *DBS* without the filters.

Each result presented in the tables is an average of over 100 experiments with the same parameters, but each with a different random seed. For the different experiments, we give "CPU time" (*i.e.*, the amount of CPU time used by all agents), "User time" (*i.e.*, the total execution time observed by the JADE multi-agent platform. It is wall-clock time includes agent and message management), "Max CPU time" (*i.e.*, the amount of CPU time used by the slowest agent), Non-Concurrent Constraints Checks ("NCCC") [45] and the number of messages. Checked constraints are a common measure for centralized CSP and have been extended for DisCSP. $NCCC_i$ is

Table 1 Filtering *DBS* obsolete messages: $\langle m, n, d, e\rangle = \langle 15, 5, 5, 250\rangle$

Filter(s)	CPU time (s)	Max CPU time (s)	User time (s)	NCCC (millions)	Messages	Max IB size
DBS$^{\overline{PF}}$	28.5	12.9	24.1	63.6	37,208	648
DBS$^{\overline{P}}$	13.1	3.6	13.8	18.7	15,437	4
none (*i.e.*,*DBS*$^{\overline{F}}$)	28.1	12.2	23.4	59.9	36,960	668
F_1	12.9	3.5	13.6	18.5	15,852	5
F_2	25.8	10.9	21.4	55.5	34,053	657
F_3	26.4	11.5	22.8	56.9	34,588	592
F_4	24.6	11.0	22.1	55.2	31,022	566
$F_1 + F_2$	12.9	3.4	13.4	18.2	15,770	4
$F_1 + F_2 + F_3$	12.4	3.4	13.5	17.9	14,732	4
DBS	9.3	2.3	11.9	13.6	10,146	4
Multi-ABT	0.7	0.2	4.5	0.9	1,127	16
Multi-AWC	1.8	1.5	25.2	2.8	320	12
AFC	OoM					

the maximum of number of checked constraints for all agents and NCCC is the sum of these different $NCCC_i$. For the first experiment, we also added "Max IB size" (*i.e.*, the maximum Inbox size for an agent).

We report on three series of experiments we conducted in order to compare our algorithm with the others. We first evaluated our different techniques for obsolete message filtering (6.2). Then, we observed the behavior of *DBS* in the case described in (6.3), where each agent encapsulates a single variable. Finally, we compare *DBS* to other DisCSP algorithms on problems with multiple variables per agent: *Multi-ABT* [32, 33], *Multi-AWC* [68] and *AFC* [44] in (6.4).

Results were obtained on a computer equipped with a quad-core 2.8 GHz Intel Core CPU and 3 GB of RAM. We noted also that the studied case is "OoM" when the number of instances producing *Out of Memory* errors is greater than 30% of total instances.

6.2 Experiment 1: Evaluating filtering for removing obsolete messages

We first evaluated the impact of our *DBS* obsolete message filtering properties, as defined in Section 5. We adjusted the constraint generator to make two out of three constraints be inter-agent constraints, in order to emphasize the impact of the filtering properties. Table 1 shows the *DBS* results (with various filters combinations) for solving distributed graph coloring problems generated with parameters $\langle 15, 5, 5, 250\rangle$ (15 agents, each with 5 variables, a domain size of 5 values and 250 constraints). For reference, we also provide the results for *DBS*$^{\overline{PF}}$ and *DBS*$^{\overline{P}}$.

The tests for *AFC* led to 52% out of memory errors for the different instances. We note also that *DBS* is less efficient than the two existing algorithms. There are numerous messages exchanged for *DBS*$^{\overline{PF}}$; however, Table 1 shows that the different filters significantly reduce this number. The F_1 filter avoids most obsolete messages;

Table 2 Impact of the number of agents: $\langle m, n, d, e\rangle = \langle m, 1, 3, 2.3m\rangle$

m		*ABT*	*AWC*	*AFC*	*DBS*$^{\overline{PF}}$	*DBS*$^{\overline{P}}$	*DBS*$^{\overline{F}}$	*DBS*
	User time (s)	4.2	6.1	5.6	4.2	4.2	4.2	4.2
	CPU time (s)	3.87	0.52	0.11	3.96	3.79	3.85	3.76
15	Max CPU time (s)	0.73	0.08	0.03	0.62	0.6	0.6	0.58
	Messages ($\cdot 10^3$)	20.3	3.8	0.6	20.5	19.4	19.8	19.1
	NCCC ($\cdot 10^3$)	275	158	4	34	33	34	31
	User time (s)	4.5	8.8	6.6	7.5	7.3	7.3	7.4
	CPU time (s)	4.0	8.8	1.9	12.1	12.5	12	12.1
30	Max CPU time (s)	0.4	0.6	0.2	1.1	1.2	1.1	1.1
	Messages ($\cdot 10^3$)	20.1	35.3	9.8	20.5	19.4	19.8	19.1
	NCCC ($\cdot 10^3$)	306	19,062	122	67	70	68	67
	User time (s)	6.2	40.1	19.5	45.3	45.3	49	41.9
	CPU time (s)	8.1	100.3	51.6	123.7	124.6	138.5	116
45	Max CPU time (s)	0.5	7.8	3	10.2	10.2	11.9	8.6
	Messages ($\cdot 10^3$)	27.5	13.3	2.2	566.1	571.9	637.2	529.1
	NCCC ($\cdot 10^3$)	656	195,398	2,915	573	572	638	530
	User time (s)	16.9	OoM	690.2	478	443.8	442.3	436.9
	CPU time (s)	38.7		386.1	1,421.9	1,329.5	1,340.5	1,315.0
60	Max CPU time (s)	1.9		103.4	90.6	80.2	83.9	81.0
	Messages ($\cdot 10^3$)	147.5		211.9	6,581.7	6,137.2	6,160.5	6,005.1
	NCCC ($\cdot 10^3$)	4,321		55,175	5,826	5,558	5,604	5,191
	User time (s)	67.1		OoM	1,179	1,154.7	1,220.7	1,149.3
	CPU time (s)	208.4			3,530.7	3,461.6	3,674.7	3,434.8
75	Max CPU time (s)	10.6			203.5	197.1	230.8	173.3
	Messages ($\cdot 10^3$)	445.5			16,096.9	15,824.4	16,704.4	15,929.5
	NCCC ($\cdot 10^3$)	27,219			12,805	12,390	13,868	12,046

additional filters always result in increased performance for nearly all metrics. In particular, the Max Inbox size metric demonstrates that the efficiency of the filters reduces the number of pending messages.

6.3 Experiment 2: Evaluating *DBS* with single-variable problems

This second experiment addresses the evaluation of the special case of one variable per agent, with one agent encapsulating each variable. Table 2 shows results for a DisCSP generated with the parameters $\langle m, 1, 3, 2.3m\rangle$. We note that *AFC* is unable to solve the 100 instances due to a memory problem when $m = 75$ (represents 50% of the out of memory issues) whereas *AWC* gives 50% out of memory for $m = 60$.

The more agents there are, the more efficient the *ABT* algorithm is (fewer messages and less CPU time than *DBS*). We observe that when the number of agents is high ($m = 75$), only *ABT* and *DBS* give results. Unlike *ABT*, *DBS* does not require the computation of all inconsistent subsets of the $agentView$ to build backtrack messages, which is very costly. Once again, the message filtering properties help reduce the number of messages. When *DBS* is used in the single-variable-per-agent case, PINs are not useful since the only variable is an inter-agent variable. The addition of filters is insufficient to decrease computational time for *DBS*. This is not surprising, because the compromise between time and the number of inter-agent messages is inadequate (accepting additional messages led to additional time). In the single-variable context, we believe that additional filters should be investigated (without maintaining the completeness of our algorithm).

6.4 Experiment 3: Evaluating DBS with multi-variable problems

In this section, we experiment *DBS* on handling multi-variable problems. We adjusted the constraint generator to more or less obtain the same number of inter- and intra-agent constraints. As reported by Yokoo and Hirayama [68], a homogeneous distribution would result in a relatively low number of intra-agent constraints. More specifically, we tested the impact of two parameters on the performance of the algorithms: the number of agents m and the number of variables per agent n (we set d to 3 colors).

Table 3 focuses on smaller DisCSP ($m = 2$ to 7 agents, $n = 5$ variables). The number of constraints is set to $2N = 10m$, since there are not enough variables encapsulated in the agents to contain half of the DisCSP's constraints. This table shows that *DBS* is more efficient (or equivalent to) its counterparts, even in the stripped down *DBS*$^{\overline{P}}$ and *DBS*$^{\overline{F}}$ versions. The relatively poor performance of *Multi-AWC* and *AFC* seem to come from the one-virtual-agent-per-variable scheme, which generates many messages, and the minimal nogoods computing, which is very costly. The results show that saving all nogoods within *Multi-ABT* implies numerous NCCCs.

In Table 4, we set m to 15 agents. Each agent encapsulates the same number of variables n ($N = nm$). We varied n from 1 to 13. To obtain hard problems, we chose a number of constraints e close to the phase transition: $e \approx 2.3N = 34.5n$ [11]. This table does not include the results due to memory problems: *Multi-AWC* produces more than 30% "OoM" for $n > 7$ and *AFC* generates more than 45% "OoM" for $n > 5$.

The results show that successful PINs reduce the number of NCCCs and that filters greatly reduce the number of exchanged messages. However, PINs tend to increase the number of messages. *DBS* performs better than *Multi-ABT* in terms of CPU time and NCCCs (even for *DBS*$^{\overline{PF}}$), and the ratios seem to grow with the number of variables per agent. *DBS*$^{\overline{PF}}$ and *DBS*$^{\overline{F}}$ send more messages than *Multi-ABT*, which is an expected result. Filters significantly reduce the number of messages, so much so that *DBS*$^{\overline{P}}$ obtains a lower number than *Multi-ABT*.

However, when the PINs and filters are combined, the decrease in NCCC does not always succeed in compensating for the cost of the calculation of the PIN and additional messages. A way to solve this problem may be a solution introduced by the *AAS* algorithm [58]: exchanged messages are based on a set of partial solutions (Cartesian product of assignments to different variables). In this case, *AAS* involves an aggregation of this list of assignments.

Table 2 and Table 4 show that the algorithms are more efficient when a translation from a multi- to single-variable problems is avoided. Appropriate handling of multi-variable problems reduces the number of exchanged messages, as expected.

Table 3 Impact of the number of agents: $\langle m, n, d, e\rangle = \langle m, 5, 3, 10m\rangle$

m		*Multi-ABT*	*Multi-AWC*	*AFC*	*DBS*$^{\overline{PF}}$	*DBS*$^{\overline{P}}$	*DBS*$^{\overline{F}}$	*DBS*
	User time (s)	4.2	12.6	9.9	4.2	4.2	4.2	4.2
	CPU time (ms)	6	18	12	21	21	25	22
2	Max CPU time (ms)	5	11	11	17	17	19	18
	Messages	32	112	11	28	28	28	28
	NCCC ($\cdot 10^3$)	16.9	27.2	1.4	0.9	0.9	0.9	0.9
	User time (s)	4.2	14.3	10.5	4.2	4.2	4.2	4.2
	CPU time (ms)	548	69	22	36	38	33	30
3	Max CPU time (ms)	290	28	15	23	23	22	19
	Messages	2,989	340	29	78	77	77	77
	NCCC ($\cdot 10^3$)	1,598	131	5	2.7	2.7	2.7	2.7
	User time (s)	4.2	16.8	12.0	4.2	4.3	4.2	4.2
	CPU time (ms)	698	136	40	76	89	74	78
4	Max CPU time (ms)	321	47	23	37	41	36	38
	Messages	3,501	566	52	407	392	334	392
	NCCC ($\cdot 10^3$)	1,836.7	448	13.3	12	11.6	10.3	11.6
	User time (s)	4.3	28.7	28.9	4.2	4.2	4.2	4.2
	CPU time (ms)	599	144	61	126	112	111	112
5	Max CPU time (ms)	249	46	32	56	46	47	47
	Messages	2,923	691	64	664	500	517	506
	NCCC ($\cdot 10^3$)	1,287	564.1	28.2	18.6	14.4	15.3	14.4
	User time (s)	4.3	19.1	14.1	4.3	4.3	4.3	4.3
	CPU time (ms)	831	326	139	345	392	321	321
6	Max CPU time (ms)	297	86	72	110	124	101	102
	Messages	3,876	1,173	122	1,887	2,128	1,764	1,780
	NCCC ($\cdot 10^3$)	2,112.1	1,614.6	104.2	43.3	48.8	40.9	41.4
	User time (s)	4.3	29.3	14.8	4.5	4.4	4.4	4.4
	CPU time (ms)	906	1,272	486	343	380	343	393
7	Max CPU time (ms)	291	311	226	111	118	107	117
	Messages	4,120	1,893	208	1,864	2,061	1,847	2,078
	NCCC ($\cdot 10^3$)	1,906.3	7,940.5	302.6	40.1	42.9	36.5	40.5

7 Conclusion

We proposed an algorithm, called *Distributed Backtracking with Sessions* (*DBS*), that is applied to solve DisCSP containing several variables per agent. Each agent encapsulates a "complex problem", with different variables (*i.e.*, local variables and interface variables) and different constraints (*i.e.*, intra-agent and inter-agent constraints). Each agent solves its local problem and then finds all solutions. The global search is performed at the same time as the local search, and instead of using nogoods to establish a context for the backtrack messages, *DBS* uses *sessions*. The sessions allow us to avoid computing nogoods: the message processing is thus much faster even though more messages are processed.

The completeness of the *DBS* proof was described in three steps. First, we proved the completeness of a single-variable version per agent, called *DBS*^{ng}, in which all agents save all forbidden instantiations. Second, we showed that the elimination of

Table 4 Impact of the number of variables per agent: $\langle m, n, d, e \rangle = \langle 15, n, 3, 34.5n \rangle$

n		*Multi-ABT*	*Multi-AWC*	*AFC*	*DBS*$^{\overline{PF}}$	*DBS*$^{\overline{P}}$	*DBS*$^{\overline{F}}$	*DBS*
	User time (s)	4.2	6.1	5.5	4.2	4.2	4.2	4.2
	CPU time (s)	3.87	0.52	0.11	3.96	3.79	3.85	3.76
1	Max CPU time (s)	0.73	0.08	0.03	0.62	0.6	0.6	0.58
	Messages ($\cdot 10^3$)	20.3	3.8	0.6	20.5	19.4	19.8	19.1
	NCCC ($\cdot 10^3$)	275	158	4	34	33	34	31
	User time (s)	4.3	27.3	14.7	5.1	5.3	5.3	5.4
	CPU time (s)	2.85	40.03	4.75	3.17	3.05	3.41	3.72
3	Max CPU time (s)	0.50	7.9	1.9	0.49	0.49	0.54	0.69
	Messages ($\cdot 10^3$)	12.8	18.8	2.5	14.9	14.2	16.2	18.2
	NCCC ($\cdot 10^3$)	2,160	207,552	1,156	56	54	61	71
	User time (s)	12.2	91.9	252.0	41.3	41.7	90.4	12.1
	CPU time (s)	29.9	85.6	230.3	71.7	103.3	217.6	21.4
5	Max CPU time (s)	5.13	23.32	118.43	27.23	16.68	54.81	2.76
	Messages ($\cdot 10^3$)	27.5	13.3	2.2	235.9	444.4	715.9	85.5
	NCCC ($\cdot 10^3$)	73,429	596,799	86,542	3,629	6,248	11,333	1,034
	User time (s)	219.5	28.4	OoM	37.9	64.5	42.7	29.5
	CPU time (s)	821.4	135.1		90.5	173.1	111.1	69.8
7	Max CPU time (s)	127.84	28.45		14.94	25.97	20.65	10.5
	Messages ($\cdot 10^3$)	206.8	13.3		219.2	443.2	388.3	178.8
	NCCC ($\cdot 10^3$)	1,894,556	859		6,898	16,626	17,364	11,149
	User time (s)	343.4	OoM		88.8	66.7	17.3	126.9
	CPU time (s)	1,278.6			253.7	169.0	39.6	176.5
9	Max CPU time (s)	198.81			45.63	34.26	5.91	80.7
	Messages ($\cdot 10^3$)	274.9			522.5	528.2	98.2	750.6
	NCCC ($\cdot 10^3$)	3,473,200			70,815	76,784	10,218	111,461
	User time (s)	366.8			109.5	58.3	24.8	76.8
	CPU time (s)	1,326.8			313.6	146.6	38.5	211.1
11	Max CPU time (s)	224.6			67.21	25.14	7.04	37.14
	Messages ($\cdot 10^3$)	256.2			404.2	216.8	47.3	276.1
	NCCC ($\cdot 10^3$)	3,869,578			196,206	74,529	19,890	104,356
	User time (s)	463.7			169.3	144.4	195.1	136.7
	CPU time (s)	1,615.5			310.7	216.6	370.9	142.9
13	Max CPU time (s)	268.26			63.3	43.34	84.24	31.3
	Messages ($\cdot 10^3$)	275.3			148.1	87.3	184.3	52.3
	NCCC ($\cdot 10^3$)	4,870,692			204,926	135,128	270,188	87,562

obsolete forbidden instantiations does not change this property. Third, we proved that the context of multiple variables per agent does not modify the completeness.

Various filters make it possible to significantly reduce the total number of exchanged messages while maintaining the completeness of our algorithm. The exchanged solution for every agent was optimized so that a given agent did not receive information that did not pertain to it. The search for consistent local solutions received from the highest priority agent was improved. Agents do not iteratively compute every solution of their local CSP when they close a session, but they use a method that makes it possible to avoid testing the solutions with the same interface variable characteristics. In addition, the origin of every inconsistent local solution is saved to avoid useless processing.

The results of our experiments were reported in this paper. *DBS* was compared not only with other multi-variable algorithms per agent (*i.e.*, *Multi-ABT*, *Multi-AWC* and *AFC*), but also with the special case of one variable per agent. Our results have shown that *DBS* has behavior that is good for solving distributed multi-variable problems. In particular, it is not as memory-consuming as alternative DisCSP algorithms, and has a reasonably efficient w.r.t. computation time.

Experiments on a variety of problems will provide better insight on the limitations of *DBS*. As with most multi-variable algorithms, the need to store numerous local solutions for large, under-constrained problems will undoubtedly be problematic. We think that PINs are an interesting first step to solving this problem. Other steps, such as optimizing data structures and lazy computations of local solutions, are still required.

The comparative study we conducted in this paper should be applied to other algorithms, such as *APO* [24, 40], which may be well-suited to handling multi-variable problems. Moreover, another dynamic priority-related perspective for our approach may improve global performance. Research on DisCSP shows that considerable performance enhancements can be obtained by using dynamic priority for agents and appropriate heuristics [10, 44, 70]. Finally, an interesting improvement for *DBS* would involve investigating DCOP problems (*i.e.*, a DisCSP with a function to optimize) and seeking the optimal solution (instead of just one solution) [23, 25, 39, 48]. Existing studies have often examined a single-variable case, and generalizing the results to multi-variable contexts may also be a new challenge.

Acknowledgements This research was partially financed by the French Ministry of National Education, Research and Technology, The Nord/Pas-de-Calais Region, the French National Center of Scientific Research (CNRS) and the International Campus on Safety and Intermodality in Transportation.

Bibliography

1. Abril M, Salido M, Barber F (2010) No-good FC for solving partitionable constraint satisfaction problems. Journal of Intelligent Manufacturing 21(1):101–110
2. Armstrong A, Durfee E (1997) Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In: Pollack M (ed) Proceedings of 15th International joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufman, Nagoya, Japan, pp 620–625
3. Belaissaoui M, Bouyakhf E (2003) Optimal distributed intelligent backtracking. Techniques et Sciences Informatiques (TSI) 22:303–306
4. Bellifemine F, Giovani C, Tiziana T, Rimassa G (2000) Jade programmer's guide. Tech. rep., Telecom Italia
5. Benelallam I, Belaissaoui M, Ezzahir R, Bouyakhf E (2008) Dynamic branch-and-bound distribué. In: quatrièmes Journées Francophones de Programmation par Contraintes (JFPC), Nantes, France

6. Bessière C, Maestre A, Meseguer P (2001) Distributed dynamic backtracking. In: Silaghi M (ed) Workshop on Distributed Constraint Reasoning, held at International Joint Conference on Artificial Intelligence (IJCAI), Seattle, Washington (USA), pp 9–16
7. Bessière C, Maestre A, Brito I, Meseguer P (2005) Asynchronous backtracking without adding links: A new member in the abt family. Artificial Intelligence 161(1-2):7–24
8. Bessière C, Katsirelos G, Narodytska N, Walsh T (2009) Circuit complexity and decompositions of global constraints. In: Boutilier C (ed) Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI), Pasadena, California (USA), pp 419–424
9. Brito I, Herrero F, Meseguer P (2004) On the evaluation of DisCSP algorithms. In: Modi PJ (ed) Fifth International Workshop on Distributed Constraint Reasoning (DCR), held at 10th International Conference on Principles and Practice of Constraint Programming (CP), Toronto, Canada, pp 438–445
10. Brito I, Meisels A, Meseguer P, Zivan R (2009) Distributed constraint satisfaction with partially known constraints. Constraints 14(2):199–234
11. Cheeseman P, Kanefsky B, Taylor W (1991) Where the really hard problems are. In: Mylopoulos J, Reiter R (eds) Proceedings of 12th International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufman, Sydney, Australia, pp 331–337
12. Clair G, Kaddoum E, Gleizes MP, Picard G (2008) Self-regulation in self-organising multi-agent systems for adaptive and intelligent manufacturing control. In: Bruechner SA, Robertson P, Bellur U (eds) 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), IEEE Computer Society, Venice, Italy, pp 107–116
13. Dechter R (1990) On the expresiveness of networks with hidden variables. In: Shrobe HE, Dietterich TG, Swartout WR (eds) Proceedings of the 8th National Conference on Artificial Intelligence (AAAI), Boston, Massachusetts, pp 556–562
14. Dechter R, Pearl J (1989) Tree clustering for constraint networks. Artficial Intelligence 38(3):353–366
15. Di Marzo Serugendo G, Gleizes MP, Karageorgos A (2006) Self-organisation and emergence in MAS: An overview. Informatica (Slovenia) 30(1):45–54
16. Doniec A, Piechowiak S, Mandiau R (2005) A DisCSP solving algorithm based on sessions. In: Russell I, Markov Z (eds) Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference (FLAIRS), AAAI Press, Clearwater Beach, Florida (USA), pp 666–670
17. Doniec A, Espié S, Mandiau R, Piechowiak S (2006) Non-normative behaviour in multi-agent system: Some experiments in traffic simulation. In: Proceedings of IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT), IEEE Computer Society, Hong Kong, China, pp 30–36
18. Doniec A, Mandiau R, Piechowiak S, Espie S (2008) Anticipation based on constraint processing in a multi-agent context. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) 17:339–361

19. Ezzahir R (2008) Traitement des problèmes de satisfaction et d'optimisation de contraintes distribués. PhD thesis, Université Mohammed V - Agdal, Maroc
20. Ezzahir R, Bessière C, Benelallam I, Bouyakhf E, Belaissaoui M (2008) Dynamic backtracking for distributed constraint optimization. In: Ghallab M, Spyropoulos C, Fakotakis N, Avouris N (eds) Proceedings of the 18th European Conference on Artificial Intelligence, IOS Press, Patras, Greece, Frontiers in Artificial Intelligence and Applications, vol 178, pp 901–902
21. Ezzahir R, Bessière C, Wahbi M, Benelallam I, Bouyakhf EH (2009) Asynchronous inter-level forward-checking for DisCSPs. In: Gent IP (ed) Proceedings of the 15th International Conference on Principles and Practice of Constraints Programming (CP), Springer, Lisbon, Portugal, vol 5732, pp 304–318
22. Felfernig A, Friedrich G, Jannach D, Zanker M (2001) Towards distributed configuration. In: Baader F, Brewka G, Eiter T (eds) KI 2001: Advances in Artificial Intelligence, Lecture Notes in Computer Science, vol 2174, Springer Berlin Heidelberg, pp 198–212
23. Gershman A, Meisels A, Zivan R (2006) Asynchronous forward-bounding for distributed constraints optimization. In: Brewka G, Coradeschi S, Perini A, Traverso P (eds) Proceedings of the 17th European Conference on Artificial Intelligence (ECAI), IOS Press, Riva del Garda, Italy, Frontiers in Artificial Intelligence and Applications, vol 141, pp 103–107
24. Grinshpoun T, Meisels A (2008) Completeness and performance of the APO algorithm. Journal of Artificial Intelligence Research (JAIR) 33:223–258
25. Grinshpoun T, Grubshtein A, Zivan R, Netzer A, Meisels A (2013) Asymmetric distributed constraint optimization problems. Journal of Artificial Intelligence Research (JAIR) 47:613–647
26. Günay A, Yolum P (2013) Constraint satisfaction as a tool for modeling and checking feasibility of multiagent commitments. Applied Intelligence 39(3):489–509
27. Hamadi Y (1999) Traitement des problèmes de satisfaction de contraintes distribués. PhD thesis, Université de Montpellier II, France
28. Hamadi Y, Bessière C, Quinqueton J (1998) Backtracking in distributed constraint networks. In: Prade H (ed) 13th European Conference on Artificial Intelligence (ECAI), John Wiley & Sons, Ltd, Brighton, United Kingdom, pp 219–223
29. Haralick RM, Elliott GL (1980) Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence 14(3):263–313
30. Hassine AB, Ho TB, Ito T (2006) Meeting scheduling solver enhancement with local consistency reinforcement. Applied Intelligence 24(2):143–154
31. Hirayama K, Yokoo M (2000) The effect of nogood learning in distributed constraint satisfaction. In: Chen W (ed) Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS), IEEE Computer Society, Taipei, Taiwan, pp 169–177
32. Hirayama K, Yokoo M, Sycara K (2000) The phase transition in distributed constraint satisfaction problems: First results. In: Dechter R (ed) Proceedings of 6th International Conference on Principles and Practice of Constraint Programming (CP), Springer, Singapore, Lecture Notes in Computer Science, vol 1894, pp 515–519

33. Hirayama K, Yokoo M, Sycara K (2004) An easy-hard-easy cost profile in distributed constraint satisfaction. Information Processing Society of Japan (IPSJ) journal 45(9):2217–2225
34. Van der Hoek W, Witteveen C, Wooldridge M (2011) Decomposing constraint systems: Equivalences and computational properties. In: Sonenberg L, Stone P, Turner K, Yolum P (eds) Proceedings of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, Taipei, Taiwan, pp 149–156
35. Jannach D, Zanker M (2013) Modeling and solving distributed configuration problems: A CSP-based approach. IEEE Transactions on Knowledge and Data Engineering 25(3):603–618
36. Jennings N (1996) Coordination techniques for distributed artificial intelligence. In: O'Hare G, Jennings N (eds) Foundations of Distributed Artificial Intelligence, Wiley, pp 187–210
37. Junker U (2004) QuickXplain: Preferred explanations and relaxations for over-constrained problems. In: McGuinness DL, Ferguson G (eds) Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI) - Sixteenth Conference on Innovative Applications of Artificial Intelligence (IAAI), AAAI Press / The MIT Press, San Jose, California (USA), pp 167–172
38. Karagiannis P, Vouros G, Stergiou K, Samaras N (2012) Overlay networks for task allocation and coordination in large-scale networks of cooperative agents. Autonomous Agents and Multi-Agent Systems 24(1):26–68
39. Mailler R, Lesser V (2004) Solving Distributed Constraint Optimization Problems Using Cooperative Mediation. In: Proceedings of Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), IEEE Computer Society, New York, NY (USA), pp 438–445
40. Mailler R, Lesser V (2004) Using cooperative mediation to solve distributed constraint satisfaction problems. In: Proceedings of Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), IEEE Computer Society Press, New-York, NY (USA), vol 1, pp 446–453
41. Malone T (1988) What is coordination theory? working paper 2051-88, Cambridge, MA: MIT Sloan School of Management
42. Mammen DL, Lesser VR (1998) Problem structure and subproblem sharing in multi-agent systems. In: Demazeau Y (ed) Proceedings of the Third International Conference on Multiagent Systems (ICMAS), IEEE Computer Society, Paris, France, pp 174–181
43. Meisels A, Zivan R (2003) Asynchronous forward-checking on DisCSPs. In: Zhang W (ed) Workshop on the Fourth Distributed Constraint Reasoning (DCR), held at 18th International Joint Conference on Artificial Intelligence, IOS Press, Acapulco, Mexico, Frontiers in Artificial Intelligence
44. Meisels A, Zivan R (2007) Asynchronous forward-checking for DisCSPs. Constraints 12(1):131 – 150
45. Meisels A, Kaplansky E, Razgon I, Zivan R (2002) Comparing performance of distributed constraints processing algorithms. In: Yokoo M (ed) 3rd Workshop on Distributed Constraint Reasoning, Held at 1st International Conference on Autonomous and Multi-Agent Systems (AAMAS), Bologna, Italy

46. Minton S, Johnston MD, Philips AB, Laird P (1992) Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. Artificial Intelligence 58(1-3):161–205
47. MirHassani S, Habibi F (2013) Solution approaches to the course timetabling problem. Artificial Intelligence Review 39(2):133–149
48. Modi PJ, Shen WM, Tambe M, Yokoo M (2005) Adopt: asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence 161(1-2):149–180
49. Monier P, Piechowiak S, Mandiau R (2009) A complete algorithm for DisCSP: Distributed backtracking with sessions (dbs). In: Jennings N, Rogers A, Aguilar JR, Farinelli A, Ramchurn S (eds) Second International Workshop on Optimisation in Multi-Agent Systems (OptMas), Held at 8th Joint Conference on Autonomous and Multi-Agent Systems (AAMAS), Budapest, Hungary, pp 39–46
50. Monier P, Belaissaoui M, Piechowiak S, Mandiau R (2010) Résolution de CSP distribués avec problèmes locaux complexes. In: Cordier MO, Jolion JM (eds) Actes du 17ème congrès francophone AFRIF-AFIA (RFIA), Caen, France, pp 694–701
51. Monier P, Doniec A, Piechowiak S, Mandiau R (2010) Metrics for the evaluation of DisCSP: Some experiments of multi-robot exploration. In: Huang JX, Ghorbani AA, Hacid MS, Yamaguchi T (eds) Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT), IEEE Computer Society Press, Toronto, Canada, pp 370–373
52. Monier P, Doniec A, Piechowiak S, Mandiau R (2011) Comparison of DCSP algorithms: A case study for multi-agent exploration. In: Demazeau Y, Pechoucek M, Corchado JM, Pérez JB (eds) Proceedings of the 9th International Conference on Practical Applications of Agents and Multiagent Systems (PAAMS), Springer, Salamanca, Spain, Advances in Intelligent and Soft Computing, vol 88, pp 231–236
53. Mouhoub M, Sukpan A (2012) Managing dynamic CSPs with preferences. Applied Intelligence 37(3):446–462
54. Muscalagiu I (2005) The effect of flag introduction on the explosion of nogood values in the case of ABT family techniques. In: Pechoucek M, Petta P, Varga LZ (eds) Multi-Agent Systems and Applications IV, 4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS), Springer, Budapest, Hungary, Lecture Notes in Computer Science, vol 3690, pp 286–295
55. Nguyen V, Sam-Haroud D, Faltings B (2004) Dynamic distributed backjumping. In: Workshop on the Fifth Distributed Constraint Reasoning, in Tenth International Conference on Principles and Practice of Constraints Programming (CP), Toronto, CA, pp 51–65
56. Pal A, Ritu RT, Shukla A (2013) Communication constraints multi-agent territory exploration task. Applied Intelligence 38(3):357–383
57. Picard G, Glize P (2006) Model and analysis of local decision based on cooperative self-organization for problem solving. Multiagent and Grid Systems 2(3):253–265
58. Silaghi MC, Sam-Haroud D, Faltings B (2000) Asynchronous search with aggregations. In: Kautz HA, Porter BW (eds) Proceedings of the 17th National

Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence, AAAI Press/MIT Press, Austin, Texas (USA), pp 917–922
59. Silaghi MC, Sam-Haroud D, Faltings B (2000) Maintaining hierarchically distributed consistency. In: Silaghi MC (ed) International Workshop on Distributed Constraint Satisfaction, Held at 7th International Conference on Principles and Practice of Constraint Programming (CP), Singapore
60. Stansbury R, Agah A (2012) A robot decision making framework using constraint programming. Artificial Intelligence Review 38(1):67–83
61. Stefanovitch N, Farinelli A, Rogers A, Jennings N (2010) Efficient multi-agent coordination using resource-aware junction trees. In: van der Hoek W, Kaminka GA, Lespérance Y, Luck M, Sen S (eds) Proceedings of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, Toronto, Canada, pp 1413–1414
62. Stefanovitch N, Farinelli A, Rogers A, Jennings N (2011) Resource-aware junction trees for efficient multi-agent coordination. In: Sonenberg L, Stone P, Turner K, Yolum P (eds) Proceedings of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, Taipei, Taiwan, pp 363–370
63. Tsuruta T, Shintani T (2000) Scheduling meetings using distributed valued constraint satisfaction algorithm. In: Horn W (ed) Proceedings of the 14th European Conference on Artificial Intelligence (ECAI), IOS Press, Berlin, Germany, pp 383–387
64. Vion J (2007) CSP4J: a Black-Box CSP Solving API for Java. In: MRC van Dongen CL, Roussel O (eds) Proceedings of the Second International CSP Solver Competition, Held in conjunction with the 12th Int. Conf. on Principle and Practice of Constraint Programming (CP), Nantes, France, pp 75–88
65. Yokoo M (1995) Asynchronous weak-commitment search for solving distributed constraint satisfaction problem. In: Lecture Notes In Computer Science, vol 976, pp 88–102
66. Yokoo M (2001) Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-agent Systems. Springer Verlag
67. Yokoo M, Hirayama K (1996) Distributed breakout algorithm for solving distributed constraint satisfaction problems. In: Tokoro M (ed) Proceedings of the Second International Conference on Multiagent Systems (ICMAS), AAAI Press, Kyoto, Japan, pp 401–408
68. Yokoo M, Hirayama K (1998) Distributed constraint satisfaction algorithms for complex local problems. In: Demazeau Y (ed) Proceedings of the Third International Conference on Multiagent Systems (ICMAS), IEEE Computer Society, Paris, France, pp 372–379
69. Yokoo M, Hirayama K (2000) Algorithms for distributed constraint satisfaction: A review. Autonomous Agents and Multi-Agent Systems 3(2):185–207
70. Zivan R, Meisels A (2006) Dynamic ordering for asynchronous backtracking on discsps. Constraints 11:179 –197