ABSTRACT

Many algorithms for Distributed Constraints Satisfaction Problem (DisCSP) resolution use additional links between variables not connected by constraints. This causes a higher needed memory space. In this paper, we propose an algorithm for DisCSP resolution, called Distributed Backtracking with Sessions (DBS) which does not use such additional links so that the initial problem’s topology is respected. This algorithm is complete and requires a low space complexity.

The main feature of this algorithm is to use the concept of sessions to provide a context for the exchanged messages.

Keywords

MAS, Distributed CSP, session, Backtracking

1. INTRODUCTION

Early researches on Constraints Satisfaction Problems (CSP) began in Artificial Intelligence in the 1970s. The CSP formalism addresses many problems in a simple and efficient way such as road traffic management [2].

However, some problems which use physically distributed data cannot be solved in a classical and centralized way. The causes can be multiple:

- The time to gather stored data on a single site can be prohibitive.
- Too much memory space is needed to store the whole problem.
- The data exchanges on many sites can be limited for confidentiality or safety reasons.

Distributed CSPs (DisCSP) were proposed in order to solve naturally distributed problems. The problem to be solved is thus distributed on all agents. These agents interact in order to find a global solution based on each agent’s local solutions. DisCSP are usually applied to solve distributed problems such as timetabling [7].

More formally, a DisCSP is a 4-tuple \((X, D, C, A)\) where:

- \(X\) is a finite set of \(p\) variables: \(\{x_1, x_2, ..., x_p\}\),
- \(D\) is a set of domains associated with these variables \(D = \{\text{Dom}(x_1), \ldots, \text{Dom}(x_p)\}\),
- \(C\) is a finite set of \(m\) constraints: \(\{c_1, c_2, ..., c_m\}\) and
- \(A\) is a finite set of \(n\) agents: \(\{A_1, A_2, ..., A_n\}\) where each agent is given a subset of \(X\).

Many algorithms for DisCSP resolution have been described. The first and most known is \(AABT\) [8]. It is often used as a reference in many papers such as [1].

In this paper, we propose a complete algorithm to solve variable-based distributed problems (different from AAS [6] which is constraint-based). Our algorithm, called Distributed Backtracking with Sessions (DBS) does not add links between agents during the search.

Unlike most DisCSP algorithms such as \(AABT\), DBS priority does not consist in minimizing the number of exchanged messages but aims at minimizing the time required to process a message. The less an agent requires CPU time to process a message, the less the message will be exact and complete. This brings about an augmentation of the total number of messages (but less costly messages) to solve the DisCSP. The work exposed in this paper is based on this trade-off.

Another characteristic of \(DBS\) consists in using the concept of sessions to provide a context for each exchanged message. Moreover, these sessions allow the use of heuristics to remove obsolete messages contained in the inboxes (messages awaiting process) of each agent without eliminating possible solutions from the problem. Consequently, the number of exchanged messages is reduced.

The remainder of this paper is organized as follows. Section 2 presents our contribution for the \(DBS\) algorithm initially described in [3, 5]. The properties of this algorithm are exposed in section 3. Section 4 describes different heuristics used to delete obsolete messages. Performances of \(DBS\) with and without the proposed additional heuristics are compared using a DisCSP generator. Finally, section 5 discusses conclusions and possible improvements of the proposed algorithm.

2. DBS

The algorithm called Distributed Backtracking with Sessions (\(DBS\)) is detailed in this section. We start by giving assumptions and notations required by this algorithm and we explain in detail how it works (section 2.1). The behavior of \(DBS\) is illustrated in section 2.2 using a simple example.

2.1 Algorithm

A total order, called \(\succ\), is established between the different agents (a priority is assigned to each agent) to avoid infinite loop problems. For example, a change for \(A_1\) implies
a change for $A_2$ which involves a change for $A_1$. Given two agents $A_1$ and $A_2$, $A_1 \succ A_2$ means that $A_1$ has a higher priority compared to $A_2$. In this paper, agent’s priority is given according to the lexicography order ($A_1 \succ A_2 \succ \cdots \succ A_n$). Given a generic agent called $self$, $\Gamma^+(self)$ (respectively $\Gamma^-(self)$) represents $self$’s higher priority (respectively lower priority) acquaintances, these acquaintances being connected with $self$ by a constraint.

In this paper, only one variable is assigned to each agent. Each agent assigns (in a concurrent way) a value for its variable and sends it to agents in $\Gamma^+(self)$ with an “ok?” message. If an agent $self$ cannot find an instantiation in agreement with the received values (stored in a set called agent_view($self$), $self$ informs an agent $A_i \in \Gamma^-(self)$ with a “backtrack” message.

To determine if a message is obsolete or not, the session concept is used: a session number is attached to each message.

**Definition 1.** Given an agent $self$ and $\Gamma^+(self)$ the set of acquaintances whose priority is lower than $self$’s priority. A session between $self$ and $\Gamma^+(self)$ is an integer indicating for each element of $\Gamma^+(self)$ the state of the global search from $self$’s point of view.

During the initialization step of the algorithm, the session of each agent is set to 0. When $self$ receives a backtrack message, called $m_b$, this one is processed only if the session number attached to $m_b$ is equal to $self$’s session number. Otherwise $m_b$ is considered obsolete. When $self$ receives an ok? message, it closes its session (its session number is incremented).

For a given pair of agent, we suppose that messages are received in the same order they are sent. $DBS$ uses three types of messages:

- (“ok?”, $(A_k, v_k, s_k)$): This message contains a triple which is composed of the sender of this message ($A_k$), the agent’s value ($v_k$) and the agent’s session ($s_k$).
- (“nogood”,$(A_k, v_k, s_k)$, listeBT). Backtrack request addressed to the agent $A_k$ for its value $v_k$ in the session $s_k$. A set of triple $(x, v, s)$ called listeBT is attached to this request. listeBT is used by $A_k$ to proceed, if needed, a backtrack message to an agent contained in listeBT.
- (“stop”): No solution exists and the receiver agent stops.

The context of an agent, called $self$, is defined by:

- $Self$’s current value.
- $Self$’s current session.
- A set called propose used to know, for each value $v_i \in D_{self}$ if $v_i$ has already been transmitted to $\Gamma^+(self)$ in the $self$’s current session.
- A set called agent_view containing triple $(x_i, v_i, s_i)$ used to determine the value $v_i$ and the session $s_i$ received from agents $x_i \in \Gamma^-(self)$.
- An integer set called receivedBtVal used to know, in the $self$’s current session, values in $D_{self}$ which have already received a backtrack request.

- A set called totalBtSet containing triple $(x, v, s)$. When $self$ has to send a backtrack request, this set allows $self$ to transmit, if needed, a backtrack message addressed to an agent in totalBtSet.

There are two different ways to terminate $DBS$. If a solution exists, there are no more exchanged messages between agents. This stable state allows to affirm that a solution has been found. If no solution exists, an agent will detect this lack of solution and will send a stop message.

Given $A$ and $B$ two sets of triples $(s, v, s)$, we note: $A \uplus B = A \cup \{(x, v, s) \in B | (x, v, s) \not\in A\}$. This operator is used by $DBS$ algorithm (for example, to update agent_view).

$DBS$ algorithm is composed of different procedures numbered from 1 to 6 and presented below. Procedures 1 and 2 are respectively used when $self$ receives an ok? or a nogood message.

Procedure 1 is used to update agent_view (line 1), to close $self$’s session (line 2) and to check if there exists a value which is consistent with agent_view (line 3).

Procedure 2 checks if a received nogood message is not obsolete (line 1). receivedBtVal and totalBtSet are updated (lines 2 and 3), then $self$ checks again if there exists a value which is consistent with agent_view. When $self$ receives a stop message, it terminates.

Procedures 3 and 4 are respectively used to close $self$’s session and to submit $self$’s current assignation.

Procedure 5 is used to check if a value in $D_{self}$ which is consistent with agent_view in order to transmit an ok? message (line 4) or a backtrack message (lines 2 and 6).

Procedure 6 is used when it is necessary to send a backtrack request. The code given from lines 1 to 10 is used to determine the information to be attached to the backtrack request. If DisCSP is inconsistent, the triple $(x, v, s)$ (line 12) will be equal to null for a given agent and a stop message will be transmit. After the emission of the backtrack message, totalBtSet is updated (line 17). If the backtrack message has been sent to an agent in agent_view, this set is updated (line 19), else $self$ closes its session and then submits an assignment to $\Gamma^+(self)$ (lines 21 and 22).

Algorithm 1 when received (“ok?”, $(x_j, d_j, s_j)$) from $A_j$

1: agent_view $\leftarrow \{(x_j, d_j, s_j)\} \cup agent_view$
2: close_session()
3: check_agent_view($A_i$, “ok?”)

Algorithm 2 when received (nogood, $(x_j, d_j, s_j)$, BtSet) do

1: if $s_j = current_value \land d_j \not\in receivedBtVal$ then
2: receivedBtVal $\leftarrow receivedBtVal \cup \{d_j\}$
3: totalBtSet $\leftarrow BtSet \uplus totalBtSet$
4: if $d_j = current_value$ then
5: current_value $\leftarrow null$
6: end if
7: check_agent_view(null, “backtrack”)
8: end if
Algorithm 3 close_session
1: current_value ← null
2: current_session ← current_session + 1
3: receivedBTCV[d] ← 0
4: for all d ∈ D do
5: propose[d] ← false
6: end for

Algorithm 4 submit_assign
1: select d ∈ D | ¬propose[d] ∧ consistent(d, agent_view)
2: current_value ← d
3: propose[d] ← true
4: send(“ok?”,(x, current_value, current_session)) to outgoing links

Algorithm 5 check_agent_view(Agent: A̅k, String: type)
1: if A̅k ≠ null ∧ (∀d ∈ D, propose[d] ∨ ¬consistent(d, {A̅k, − A̅k} ∈ agent_view))
2: backtrack(A̅k, type)
3: else if ∃d ∈ D | ¬propose[d] ∧ consistent(d, agent_view)
4: submit_assign()
5: else
6: backtrack(null, type)
7: end if

Algorithm 6 backtrack(Agent: A̅k, String: type)
1: if A̅k ≠ null then
2: (x, v, s) ← {(x′, v′, s′) ∈ agent_view|x′ = A̅k}
3: BtSet ← {(x′, v′, s′) ∈ agent_view ∪ totalBtSet|x′ > x}
4: else if type = “ok?” then
5: (x, v, s) ← {(x′, v′, s′) ∈ agent_view|∀A ∈ agent_view, A ≥ x′}
6: BtSet ← {(x′, v′, s′) ∈ agent_view ∪ totalBtSet|x′ > x}
7: else
8: BtSet ← agent_view ∪ totalBtSet
9: (x, v, s) − {(x′, v′, s′) ∈ BtSet|∀A ∈ BtSet, A ≥ x′}
10: BtSet ← BtSet − {(x, v, s)}
11: end if
12: if (x, v, s) = null then
13: broadcast to other agents “stop” message
14: terminate this algorithm
15: end if
16: send (backtrack, (x, v, s), BtSet) to x
17: TotalBtSet ← TotalBtSet − {BtSet ∪ (x, v, s)}
18: if {(x, − A̅k, − A̅k)} ∈ agent_view then
19: agent_view ← ¬ agent view − {(x, v, s)}
20: else if type = “backtrack” then
21: close_session()
22: submit_assign()
23: end if

2.2 Example
A simple illustration of DBS algorithm is given in figure 1. This problem is composed of four variables \{x_1, x_2, x_3, x_4\} connected by three constraints \(x_1 \neq x_3\), \(x_1 \neq x_4\) and \(x_2 \neq x_4\). One possible execution of DBS appears in figure 2.

![Figure 1: DisCSP example.](image)

![Figure 2: Exchanged messages between A_1 and A_j.](image)

Initially, each agent chooses the first value of its domain \(D\) and then sends it to agents in \(\Gamma^+(self)\). Indeed, \(M_1, M_2\) and \(M_3\) messages are sent. Upon receiving \(M_1, A_0\) adds \((x_1, a, 0)\) to \(agent_view(A_3)\). Since \(A_3\) cannot satisfy the constraint \((x_1 \neq x_3)\), it transmits a backtrack message \(M_4\) to \(A_1\) then removes the triple containing \(A_1\) from \(agent_view(A_3)\).

\(A_4\) receives \(M_2\) and \(M_3\). Since it cannot find a possible instantiation in agreement with \(agent_view(A_4) = \{(x_1, a, 0), (x_2, b, 0)\}\), it sends a backtrack request to the lowest priority agent contained in \(agent_view(A_4)\), i.e. \(A_2\). It attaches to this message \((M_5)\) \(BtSet = \{(x_1, a, 0)\}\) in order to allow \(A_2\) to proceed, if needed, the backtrack request. \(A_4\) removes \((x_2, b, 0)\) from \(agent_view(A_4)\).

\(A_1\) receives \(M_4\) and modifies its current value then sends its new value to \(A_3\) and \(A_4\) using \(M_6\) and \(M_7\) messages. \(A_2\) receives \(M_5\) and updates \(totalBtSet\) with \(BtSet(M_5)\). Since it cannot find a value, it transmits a backtrack request \((M_8)\) to \(A_1\) which is the lowest priority agent contained in...
agent_view(A₂) ∪ totalBtSet. A₂ forgets all previous references to A₁ because A₂ removes them from totalBtSet. Then, since A₂ has just sent a backtrack request to an agent not included in agent_view(A₂), it closes its session (the session become 1) and sends again its current value to agents in F⁺ (here A₁) using M₀ message. A₁ receives M₀ but does not process it because A₁ has already received a backtrack request about the same value in its current session (0). A₄ receives M₄. There are no more exchanged messages. The solution is found: \{(x₁ = b), (x₂ = b), (x₃ = a), (x₄ = a)\}.

3. DBS PROPERTIES

In this section, we will demonstrate that DBS is sound (section 3.1), complete and that it terminates (section 3.2). Then, we show that its spatial complexity is polynomial bounded (section 3.3). In order to prove the DBS’s completeness, we use the completeness proof of both ABT [8] and DDB [1].

3.1 Soundness

Theorem 1. DBS is sound, in that it only claims a solution if one exists.

Proof 1. Whenever DBS detects a solution, all agents are in a stable state, waiting for messages. Such a state is incompatible with constraint violations, which would entail at least one message.

3.2 Completeness

To demonstrate that DBS is complete and terminates, we consider DBS_all, an alternative implementation of DBS (similar to ABT) with full Forbidden Instantiation Combination (FIC)¹ recording. The use of FIC is needed to prove the completeness. We will show that DBS_all is equivalent to ABT whose completeness has been proved [8]. Indeed, the DBS_all completeness proof will be shown. Then, in a second step, we will show that it is not needed to store all FIC during the search. Next, we will prove that DBS preserves all properties related to DBS_all. DBS completeness proof will be achieved.

3.2.1 DBS_all completeness proof

DBS algorithm uses temporarily stored Forbidden Instantiations (FI). A FI (for DBS) corresponds to a forbidden value by self’s variable. FI are stored in a set called receivedBtVal². When self’s session changes, this set is re-initialized.

To prove DBS completeness, we will modify this algorithm so that it can record FIC during all the solution search. The objective aims to describe the same completeness proof than for ABT. DBS algorithm with FIC recording is called DBS_all. Now, we will explain why DBS_all is similar to ABT (in four steps).

Nogood (FIC) recording.

In ABT, when self receives a nogood, it adds this nogood message to its set of nogoods. In DBS_all, when self receives a backtrack request (“backtrack”, (self, default_value, session), BtSet), it checks agent_view.

Let us suppose agent_view(self) = \{(x₁, v₁, s₁), ..., (xₖ, vₖ, sₖ)\}. Unlike DBS which adds default_value to the set called receivedBtVal, DBS_all completes default_value in order to create a FIC. This FIC, which is stored in a set called FICset, is \{\{(x₁, v₁, s₁)\}, ..., \{(xₖ, vₖ, sₖ)\}, (self, default_value, session)\} ∪ BtSet.

FICset is never re-initialized. Now, DBS_all records FIC as ABT records nogood. Note that FICset contains triple (variable, value, session) but that would be exactly the same if it contained (variable, value) pairs.

Backtrack message context.

Unlike ABT which attached a nogood set for each backtrack message, DBS_all uses the concept of session. The set called BtSet attached to these messages is only used to proceed, if needed, the backtrack request.

Concerning with the behavior of ABT, suppose that self receives the nogood: \{(x₁, v₁), ... (xₖ, vₖ), (self, default_value)\}. There will be a backtrack, for ABT, if the two following conditions are respected:

1. default_value is equal to self’s current value.
2. for each (xᵢ, vᵢ) pair in nogood, if xᵢ belongs to agent_view then vᵢ (of the nogood) must be equal to the value contained in self’s agent_view.

Concerning with the behavior of DBS_all, suppose that self receives (backtrack, (self, default_value, session), BtSet). There will be a backtrack, for DBS_all, if the two following conditions are respected:

1. default_value is equal to self’s current value.
2. session attached to this message is equal to self’s session.

For DBS_all, the first condition is the same as the first condition for ABT. The second one (for DBS_all) is true if self does not receives an ok? message between the time it sends its current value and the time it receives a backtrack message on this same value (so called nogood)

Choice of the backtrack message receiver (after receiving an ok? message).

ABT and DBS_all differ on the choice of the backtrack message receiver. For these two algorithms, when self receives an ok? message from an agent Aₖ, self update self’s agent_view. Suppose that no value in self’s domain is compatible with agent_view.

1. In ABT, self build a nogood set with self’s agent_view subsets where no solution exists. Then, for each subset, a backtrack message is sent to lowest priority agents in the subset.
2. In DBS, if no partial solution is found (a value for self which is consistent with Aᵢ in agent_view[Aᵢ ≥ A₁], the backtrack message is sent to A₁ (line 2 of backtrack procedure). If a partial solution exists, the backtrack message is addressed to the lowest priority agent in self’s agent_view (line 5 of backtrack procedure).
where DBS messages and a backtrack message.

Next, we need to define the message’s content.

Finally, we need to determine the content of the backtrack message. Suppose that the message must be transmitted to agent $x$. In DBS, self sends a backtrack message to the lowest priority agent in $G_{v}$. This is included to backtrack $m_{2}$, the value $v_{2}$ from $x_{2}$ (which is included in $G_{v}$’s agent_view) is included to the nogood ready to be sent. With $DBS_{alt}$, the triple $(x_{j}, v_{j}, s_{j})^{3}$ contained in self’s agent_view is included to the backtrack message. Moreover, in $DBS_{alt}$, self adds a set of triple $(x, e, s)$ called listeBT to the message so that $x_{j}$ can proceed the backtrack message.

To conclude on the first step of the completeness proof, we have described an alternative implementation for DBS, called $DBS_{alt}$, with full FIC recording during the search. We have shown that $DBS_{alt}$ is similar to DBS: no nogoods recording (FIC for $DBS_{alt}$), context of backtrack message, choice of the backtrack message receiver following an ok? message and a backtrack message. $ABT$ is complete and terminates.

3.2.2 DBS completeness proof

DBS differs from $DBS_{alt}$ on FIC recording. In this section, we will prove that eliminating Forbidden Instantiations (F1) cannot cause an infinite loop between agents. This equivalent to proof that DBS terminates.

We add the session $s_{j}$ because DBS uses the concept of session to determine message context.

LEMMA 1. Let $A_{1}$ be the agent with the highest priority. $A_{1}$ can never fall into an infinite loop because of the way obsolete Forbidden Instantiations (F1) are discarded.

PROOF 1. DBS re-initializes the set called receivedBtVal containing F1 when an agent $A_{1} \in \Gamma^{-}(self)$ sends an ok? message to self or when self sends a backtrack message to an agent $A_{j} \in \Gamma^{-}(self)$, opposite of $A_{1}$’s agent_view. $A_{1}$ is the highest priority agent so $\Gamma_{A_{1}} = \emptyset$. Values stored in receivedBtVal for $A_{1}$ can never be removed. Moreover $A_{1}$ has a finite domain, so the size of the set called receivedBtVal will be, in the worst case, equal to $d$ where $d$ is the size of the $A_{1}$’s domain. $A_{1}$ can never receive backtrackVal and $A_{1}$ can only receive a finite number of values $d$, $A_{1}$ cannot fall into an infinite loop.

LEMMA 2. If the first $(k - 1)$ agents, in the order defined by DBS, are not trapped in an infinite loop, $A_{k}$ cannot fall into an infinite loop because of the way receivedBtVal (containing obsolete Forbidden Instantiations F1) is reinitialized.

PROOF 2. Let us suppose $A_{k}$ is actually looping. That means that it forgets F1 because $A_{k}$’s predecessors which continuously change their values. F1 are removed from $A_{k}$ (procedure close_session) when:

- $A_{k}$ receives an ok? message from an agent in $\Gamma^{-}(A_{k})$.
- $A_{k}$’s session is so incremented.
- $A_{k}$ sends a backtrack message to an agent $A_{j}$ in $\Gamma^{-}(A_{k})$ which is not included in $A_{j}$’s agent_view because $A_{k}$ has modified its value. $A_{k}$’s session is so incremented.

But since we assume that no agent among $A_{1}, \ldots , A_{k-1}$ is in an infinite loop, they will stabilize in a finite time. Indeed, $A_{k}$ exits its so-called infinite loop or find there are no solution to the DistCSP. $A_{k}$ is not in an infinite loop.

THEOREM 2. DBS is sound, complete, and terminates.

PROOF 2. By recurrence about lemmas 1 and 2, we affirm that none of DBS agents can fall into an infinite loop, despite the fact that DBS discards obsolete Forbidden Instantiations (F1). DBS has the same properties than $DBS_{alt}$ and terminates.

We have shown an alternative DBS implementation called $DBS_{alt}$. This one is equivalent to $ABT$ whose the completeness proof is known. We have proved that $DBS_{alt}$ is complete too. Then, we have shown that none of $DBS_{alt}$ agents can fall into an infinite loop. We have demonstrated that DBS is sound, complete, and terminates.

3.3 Spatial complexity

THEOREM 3. Each agent performing DBS requires a polynomially bounded storage space.

PROOF 3. Section 2.1 underlines that each agent has a domain (of size $d$), a set called agent_view (of size $n$ in the worst case), a set called propose (of size $d$ in the worst case), a set called receivedBtVal (of size $d$ in the worst case) and a set called totalBtSet (of size $n$ in the worst case). So, memory space needed for each agent is $O(d + n)$.

Memory space needed for each agent performing DBS is $O(d + n)$. It is better than both $DDB$ ($O(d + n)$) and $ABT$ ($O(d^{2n})$).
4. EXPERIMENTAL RESULTS

We describe four heuristics for DBS algorithm. These heuristics (section 4.1) allow a reduction of the number of exchanged messages during the solution search without eliminating solutions. Results obtained by DBS, described in section 4.2, are compared to ABT, a classical algorithm used as a reference.

4.1 Heuristics

Given a generic agent self, self’s inbox containing unprocessed messages is called IB_{self}.

**Heuristic 1.** If self has, in IB_{self}, several ok? messages from an agent A_k, only the last ok? message sent by A_k is processed, others are removed. This heuristic, called h1, does not remove any solutions.

**Proof 1.** A session, for a given agent, cannot be decremented. If an agent A_k sends several ok? messages to self, then the last ok? message, called m_{last}, have an attached session superior or equal to others ok? messages sent by A_k.

Suppose that several ok? messages sent by A_k (received by self) are contained in IB_{self}. The last received ok? message is: m_{last} = (“ok?”, (A_k, s_i, v_i)). Four cases are possible for others (“ok?”, (A_k, v_i, s_i)) messages, received from A_k:

1. (v_i = v_k and s_i = s_k): Impossible because an agent cannot submit multiple times the same value during a given session.
2. (v_i = v_k and s_i < s_k): The context of a message is defined by a session number. Messages containing an obsolete session number are not processed. In this case, if self processes the ok? message containing s_k (m_{last}), then ok? messages containing s_i become obsolete and can be removed.
3. (v_i ≠ v_k and s_i = s_k): Each agent uses values in its domain in an order defined in the initialisation step. This order does not change during the solution search. Indeed, value v_i from session s_i is an obsolete value compared to v_k (in the same session), h1 does not remove any solution even if DBS uses (after the session closing) a value ordonnancing heuristic like those cited in [4]. In fact, if A_k sends a new value v_k, it is because it has received a backtrack message for v_i. Ok? messages containing s_i are obsolete compared to m_{last} and are removed.
4. (v_i ≠ v_k and s_i < s_k): Messages containing s_i are obsolete compared to m_{last} and are deleted (see second case).

**Heuristic 2.** If self has in IB_{self}, many backtrack messages and, at least, an ok? message then backtrack messages are removed from IB_{self}. This heuristic, called h2, does not remove any solutions.

**Proof 2.** Suppose that self first processes an ok? message, self’s session is incremented. Backtrack messages contained in IB_{self} become obsolete because they contain a value from an obsolete self’s session. All backtrack messages are removed because they are not processed by DBS (line 1 from procedure 2).

**Heuristic 3.** If self has sent a backtrack message to A_k in self’s agent_view, then as long as self has not received an ok? message from A_k, self removes all backtrack messages from IB_{self}. This heuristic, called h3, does not remove any solutions.

**Proof 3.** Suppose that self has sent a backtrack message to A_k. Self will receive, in a finite time, an ok? message, called m_k from A_k. If self waits for this message (then processes it) before processing backtrack messages already contained in IB_{self} then, using h2, self removes all received backtrack messages.

**Heuristic 4.** If self have many ok? messages in IB_{self}, then self processes, in priority, those coming for higher priority agents. This heuristic, called h4, does not remove any solutions.

**Proof 4.** Using this heuristic, no message is removed from IB_{self}. DBS is complete (section 3.2), so this algorithm works for all message reception orders (even with the order obtained with h4). However, hypothesis from section 3.2 must be respected: “For a given pair of agent, we suppose that messages are received in the same order they are sent”.

4.2 Results

We evaluate the efficiency of DBS algorithm using JADE multi-agent platform. Our purpose here is to obtain a nondeterministic scheduling agent. In many papers, a discrete event simulator is used: each agent is activated one after another using cycles. One cycle consists in reading all incoming messages (during a cycle t) and sending messages (available to others agents’ inboxes in cycle t + 1). In real conditions, agents are not activated one after another. Moreover, this method would favour DBS when it uses message suppression heuristics. Indeed, the more messages there are in inboxes (which have not already been processed), the more our heuristics are efficient.

Unlike ABT, DBS’s priority does not consist in minimizing the number of exchanged messages but aims at minimizing the time required to process a message. Table 1 shows that DBS, for a DisCSP example, uses more messages than ABT but requires less CPU time. Moreover, the maximal number of messages contained simultaneously in the different inboxes and waiting for process is lower for DBS using heuristics than for ABT. For ABT, when an agent sends a backtrack message to an agent A_k ∈ Γ−, messages will be processed with a longer time than for DBS using heuristics.

We performed experimental evaluations on DisCSP created with a randomly DisCSP generator. This one requires four parameters: the number of agents/variables (N), the domain’s size for each variable (D), the percentage of constraints between agents (P_1) and the percentage of forbidden tuples per constraint (P_2). The number of constraints is 2^{N \times D} \times P_1. The number of forbidden tuples per constraint is d × d × P_2. A disCSP is represented as a tuple < N, D, P_1, P_2 >.

For example, the following problems have been generated:

- Problems with a constraint density equal to 40% < 15, 10, 0.40, P_2 > and equal to 80% < 15, 10, 0.80, P_2 >. For each problem, percentage of forbidden tuples for each constraint ranges from 10% to 90%. Each dot in the plots presented in figures 3 and 4 corresponds to the average CPU time (over
Table 1: $<20,10,0.20,0.60>$ DisCSP: average on 200 experiments.

<table>
<thead>
<tr>
<th>Used algorithm</th>
<th>CPU times (in seconds)</th>
<th>Number of checked constraints (in millions)</th>
<th>Number of exchanged messages</th>
<th>Maximal number of present messages in inboxes awaiting processing</th>
<th>Standard deviation (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABT</td>
<td>29.3</td>
<td>2.7</td>
<td>19 650</td>
<td>1 340</td>
<td>104</td>
</tr>
<tr>
<td>DBS</td>
<td>26.7</td>
<td>2.8</td>
<td>236 076</td>
<td>1 554</td>
<td>74</td>
</tr>
<tr>
<td>DBS using heuristics</td>
<td>14.8</td>
<td>1.3</td>
<td>101 5105</td>
<td>9</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 2: $<15,10,0.50,0.50>$ DisCSP: average on 200 experiments.

<table>
<thead>
<tr>
<th>Used algorithm</th>
<th>CPU times (in seconds)</th>
<th>Number of checked constraints (in millions)</th>
<th>Number of exchanged messages</th>
<th>Maximal number of present messages in inboxes awaiting processing</th>
<th>Standard deviation (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABT</td>
<td>428.2</td>
<td>24.1</td>
<td>109 700</td>
<td>15 471</td>
<td>751</td>
</tr>
<tr>
<td>DBS</td>
<td>20.5</td>
<td>2.6</td>
<td>170 137</td>
<td>18 823</td>
<td>13</td>
</tr>
<tr>
<td>DBS using heuristics</td>
<td>9.3</td>
<td>0.8</td>
<td>44 915</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 3: DisCSP constrained with 40% of the maximal constraint number.

Figure 4: DisCSP constrained with 80% of the maximal constraint number.

100 instances) required by different DisCSP algorithms. Results are obtained with an Intel Core 2 duo 2.4 Ghz (4 Go of Ram).

On figures 3 and 4, we can observe that DBS is generally slightly faster than ABT for over-constrained and under-constrained DisCSP. For problems located in the transition phase, DBS outperforms ABT in terms of CPU time. Moreover, heuristics (see section 4.1) allow a reduction of the number of exchanged messages and CPU time. For DisCSP constrained with 80% of the maximal constraint number, during the transition phase, we stopped ABT after four hours of computation and we do not report it in the average.

Table 1 shows the results obtained with $<20,10,0.20,0.60>$ DisCSP. We observe that the number of exchanged messages for ABT is lower compared to DBS. DBS requires less operations than ABT to process received backtrack messages: ABT computes all inconsistent agent_view subsets and sends, for each subset, a backtrack message (It requires high computation costs). Unlike ABT, DBS transmits a backtrack message to the faulty agent or an agent with a lower priority than the faulty agent (It requires less computation costs but more messages).

Following a similar protocol, another experiment using the following parameters $<15,10,0.50,0.50>$ was run (Table 2). ABT solves DisCSP using 109 700 messages in 428 seconds. DBS without heuristics solves DisCSP using 170 137 messages in 20 seconds and DBS using heuristics 44 915 messages in 9 seconds. We can observe that the standard deviation is very important. This is caused by the topology of the problems. Indeed there are simple problems which can be solved within 100 ms and much harder problems (generated with the same parameters) which require more than 500 seconds to be solved.

5. CONCLUSION

In this paper, we have proposed a DisCSP resolution algorithm. Its main feature is the use of sessions to determine message context. Moreover, DBS does not add communication links during the solution search between agents which are not sharing constraints. Indeed, when an agent transmits a backtrack message to an agent $A_i \notin \Gamma^-$, no trace of $A_i$ is kept after the message has been sent. The DBS completeness proof has been shown. A comparison between DBS and ABT, a classical algorithm often used as a reference, has been given.

Currently, DBS only solves DisCSP where one variable is assigned to each agent. We plan to improve our algorithm in order to solve DisCSP containing multiple variables per agent. A more detailed comparison of DBS with recent DisCSP algorithms proposed in the literature is also considered as a perspective of our work.
6. REFERENCES


