Agile Asynchronous Backtracking for Distributed Constraint Satisfaction Problems

Christian Bessiere, El Houssine Bouyakhf, Younes Mechqrane, Mohamed Wahbi

TECHNICAL REPORT

Agile Asynchronous Backtracking for Distributed Constraint Satisfaction Problems

Christian Bessiere[1] El Houssine Bouyakhf[2]
Younes Mechqrane[2] Mohamed Wahbi[1,2]

[1]LIRMM, University Montpellier 2, France
[2]LIMIARF/FSR, University Mohammed V Agdal, Morroco

May 2011 R.R.LIRMM RR–11017

Abstract

Asynchronous Backtracking is the standard search procedure for distributed constraint reasoning. It requires a total ordering on the agents. All polynomial space algorithms proposed so far to improve Asynchronous Backtracking by reordering agents during search only allow a limited amount of reordering. In this paper, we propose Agile-ABT, a search procedure that is able to change the ordering of agents more than previous approaches. This is done via the original notion of termination value, a vector of stamps labelling the new orders exchanged by agents during search. In Agile-ABT, agents can reorder themselves as much as they want as long as the termination value decreases as the search progresses. Our experiments show the good performance of Agile-ABT when compared to other dynamic reordering techniques.

1 Introduction

Various application problems in distributed artificial intelligence are concerned with finding a consistent combination of agent actions (e.g., distributed resource allocation [6], sensor networks [1]). Such problems can be formalized as Distributed Constraint Satisfaction Problems (DisCSPs). DisCSPs are composed of agents, each owning its local constraint network. Variables in different agents are connected by constraints. Agents must assign values to their variables so that all constraints between agents are satisfied. Several distributed algorithms for solving DisCSPs have been developed, among which Asynchronous Backtracking (ABT) is the central one [10, 2]. ABT is an asynchronous algorithm executed autonomously by each agent in the distributed problem. Agents do not have to wait for decisions of others but they are subject to a total (priority) order. Each agent tries to find an assignment satisfying the constraints with what is currently known from higher priority agents. When an agent assigns a value to its variable, the selected value is sent to lower priority agents. When no value is possible for a variable, the inconsistency is reported to higher agents in the form of a nogood. ABT computes a solution (or detects that no solution exists) in a finite time. The total order is static. Now, it is known from centralized CSPs that adapting the order of variables dynamically during search drastically fastens the search procedure.

Asynchronous Weak Commitment (AWC) dynamically reorders agents during search by moving the sender of a nogood higher in the order than the other agents in the nogood [9]. But AWC requires exponential space for storing nogoods. Silaghi et al. (2001) tried to hybridize ABT with AWC. Abstract agents fulfill the reordering operation to guarantee a finite number of asynchronous reordering operations. In [7], the heuristic of the centralized dynamic backtracking was applied to ABT. However, in both studies, the improvement obtained on ABT was minor.

Zivan and Meisels (2006) proposed Dynamic Ordering for Asynchronous Backtracking (ABTDO). When an agent assigns value to its variable, ABTDO can reorder lower priority agents. A new kind of ordering heuristics for ABTDO is presented in [13]. In the best of those heuristics, the agent that generates a nogood is placed between the last and the second last agents in the nogood if its domain size is smaller than that of the agents it passes on the way up.

In this paper, we propose Agile-ABT, an asynchronous dynamic ordering algorithm that does not follow the standard restrictions in asynchronous backtracking algorithms. The order of agents appearing *before* the agent receiving a backtrack message can be changed with a great freedom while ensuring polynomial space complexity. Furthermore, that agent receiving the backtrack message, called the backtracking *target*, is not necessarily the agent with the lowest priority within the conflicting agents in the current order. The principle of Agile-ABT is built on termination values exchanged by agents during search. A termination value is a tuple of positive integers attached to an order. Each positive integer in the tuple represents the expected current domain size of the agent in that position in the order. Orders are changed by agents without any global control so that the termination value decreases lexicographically as the search progresses. Since, a domain size can never be negative, termination values cannot decrease indefinitely. An agent informs the others of a new order by sending them its new order and its new termination value. When an agent compares two contradictory orders, it keeps the order associated with the smallest termination value.

The rest of the paper is organized as follows. Section 2 recalls basic definitions. Section 3 describes the concepts needed to select new orders that decrease the termination value. We give the details of our algorithm in Section 4 and we prove it in Section 5. An extensive experimental evaluation is given in Section 6. Section 7 concludes the paper.

2 Preliminaries

The Distributed Constraint Satisfaction Problem (DisCSP) has been formalized in [10] as a tuple $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{A}$ is a set of agents, $\mathcal{X}$ is a set of variables $\{x_1, \ldots, x_n\}$, where each variable x_i is controlled by one agent in $\mathcal{A}$. $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of domains, where D_i is a finite set of values to which variable x_i may be assigned. The initial domain size of a variable x_i is denoted by d_i^0. $\mathcal{C}$ is a set of binary constraints that specify the combinations of values allowed for the two variables they involve. A constraint $c_{ik} \in \mathcal{C}$ between two variables x_i and x_k is a subset of the Cartesian product $c_{ik} \subseteq D_i \times D_k$.

For simplicity purposes, we consider a restricted version of DisCSP where each agent controls exactly one variable. We use the terms agent and variable interchangeably and we identify the agent ID with its variable index. All agents maintain their own counter, and increment it whenever they change their value. The current value of the counter *tags* each generated assignment.

Definition 1 *An* **assignment** *for an agent $A_i \in \mathcal{A}$ is a tuple (x_i, v_i, t_i), where v_i is a value from the domain of x_i and t_i is the tag value. When comparing two assignments, the most up to date is the one with the highest tag t_i. Two sets of assignments $\{(x_{i_1}, v_{i_1}, t_{i_1}), \ldots, (x_{i_k}, v_{i_k}, t_{i_k})\}$ and $\{(x_{j_1}, v_{j_1}, t_{j_1}), \ldots, (x_{j_q}, v_{j_q}, t_{j_q})\}$ are* **coherent** *if every common variable is assigned the same value in both sets.*

A_i is allowed to store a unique order denoted by o_i. Agents appearing before A_i in o_i are the higher agents (predecessors) denoted by $Pred(A_i)$ and conversely the lower agents (successors) $Succ(A_i)$ are agents appearing after A_i.

Definition 2 *The* **AgentView** *of an agent A_i is an array containing the most up to date assignments of $Pred(A_i)$.*

Agents can infer inconsistent sets of assignments, called **nogoods**. A nogood can be represented as an implication. There are clearly many different ways of representing a given nogood as an implication. For example, $\neg[(x_1{=}v_1) \wedge \cdots \wedge (x_k{=}v_k)]$ is logically equivalent to $[(x_2{=}v_2) \wedge \cdots \wedge (x_k{=}v_k)] \rightarrow (x_1 \neq v_1)$. When a nogood is represented as an implication, the left hand side (lhs) and the right hand side (rhs) are defined from the position of $\rightarrow$. A nogood ng is **compatible** with an order o_i if all agents in $lhs(ng)$ appear before $rhs(ng)$ in o_i.

The current domain of x_i is the set of values $v \in D_i$ such that $x_i \neq v$ does not appear in any of the right hand sides of the nogoods stored by A_i. Each agent keeps only one nogood per removed value. The size of the current domain of A_i is denoted by d_i.

3 Introductory Material

Before presenting Agile-ABT, we need to introduce new notions and to present some key subfunctions.

3.1 Reordering details

There is one major issue to be solved for allowing agents to asynchronously propose new orders: The agents must be able to coherently decide which order to select. We propose that the priority between the different orders is based on *termination values*. Informally, if $o_i = [A_1, \ldots, A_n]$ is the current order known by an agent A_i, then the tuple of domain sizes $[d_1, \ldots, d_n]$ is the termination value of o_i on A_i. To build termination values, agents need to exchange *explanations*.

Definition 3 *An* **explanation** *e_j is an expression of the form $lhs(e_j) \rightarrow d_j$, where $lhs(e_j)$ is the conjunction of the left hand sides of all nogoods stored by A_j as justifications of value removals, and d_j is the number of values not pruned by nogoods in the domain of A_j. d_j is also denoted by $rhs(e_j)$.*

Each time an agent communicates its assignment to other agents (by sending them an ***ok?*** message), it inserts its explanation in the ***ok?*** message for allowing other agents to build their termination value.

The variables in the left hand side of an explanation e_j must precede the variable x_j in the order because the assignments of these variables have been used to determine the current domain of x_j. An explanation e_j induces ordering constraints, called *safety conditions* in [4].

Definition 4 **A safety condition** *is an assertion $x_k \prec x_j$. Given an explanation e_j, $S(e_j)$ is the set of safety conditions induced by e_j, where $S(e_j)=\{(x_k \prec x_j) \mid x_k \in lhs(e_j)\}$.*

An explanation e_j is **compatible** with an order o if all variables in $lhs(e_j)$ appear before x_j in o. Each agent A_i stores a set E_i of explanations sent by other agents. During search, E_i is updated to remove explanations that are no longer valid.

Definition 5 *An explanation e_j in E_i is* **valid** *on agent A_i if it is* compatible *with the current order o_i and $lhs(e_j)$ is coherent with the AgentView of A_i .*

When E_i contains an explanation e_j associated with A_j, A_i uses this explanation to justify the size of the current domain of A_j. Otherwise, A_i assumes that the size of the current domain of A_j is equal to d_j^0. The termination value depends on the order and the set of explanations.

Definition 6 *Let E_i be the set of explanations stored by A_i, o be an order on the agents such that every explanation in E_i is compatible with o, and $o(k)$ be such that $A_{o(k)}$ is the kth agent in o. The* **termination value** *$TV(E_i, o)$ is the tuple $[tv^1, \dots, tv^n]$, where $tv^k = rhs(e_{o(k)})$ if $e_{o(k)} \in E_i$, otherwise, $tv^k = d^0_{o(k)}$.*

In Agile-ABT, an order is always associated with a termination value. When comparing two orders the *strongest* order is that associated with the lexicographically *smallest* termination value. In case of ties, we use the lexicographic order on agents IDs, the smallest being the strongest. Consider for instance the two orders o_1=$[A_1, A_2, A_5, A_4, A_3]$ and o_2=$[A_1, A_2, A_4, A_5, A_3]$ where agents are ordered according to their IDs from left to right. If the termination value associated with o_1 is equal to the termination value associated with o_2, o_2 is stronger than o_1 because the vector $[1, 2, 4, 5, 3]$ of IDs in o_2 is lexicographically smaller than the vector $[1, 2, 5, 4, 3]$ of IDs in o_1.

3.2 The backtracking target

When all values of an agent A_i are ruled out by nogoods, these nogoods are resolved, producing a new nogood ng. ng is the conjunction of lhs of all nogoods stored by A_i. If ng is empty, then the inconsistency is proved. Otherwise, one of the conflicting agents must change its value. In standard ABT, the agent that has the lowest priority must change its value. Agile-ABT overcomes this restriction by allowing A_i to select with great freedom the target agent A_k who must change its value (i.e., the variable to place in the right hand side of ng). The only restriction to place a variable x_k in the right hand side of ng is to find an order o' such that $TV(up\_E, o')$ is lexicographically smaller than the termination value associated with the current order of A_i. up_E is obtained by updating E_i after placing x_k in $rhs(ng)$.

Function `updateExplanations` takes as arguments the set E_i, the nogood ng and the variable x_k to place in the rhs of ng. `updateExplanations` removes all explanations that are no longer coherent after placing x_k in the right hand side of ng. It updates the explanation of agent A_k stored in A_i and it returns a set of explanations up_E.

This function does not create cycles in the set of safety conditions $S(up\_E)$ if $S(E_i)$ is acyclic. Indeed, all the explanations added or removed from $S(E_i)$ to obtain $S(up\_E)$ contain x_k. Hence, if $S(up\_E)$ contains cycles, all these cycles should

```
function updateExplanations($E_i, ng, x_k$)
1.  $up\_E \leftarrow E_i$;
2.  setRhs($ng, x_k$);
3.  remove each $e_j \in up\_E$ such that $x_k \in$ lhs($e_j$);
4.  if ($e_k \notin up\_E$) then
5.      setLhs($e_k$,$\emptyset$);
6.      setRhs($e_k$,$d_k^0$);
7.      add $e_k$ to $up\_E$;
8.  setLhs($e'_k$, lhs($e_k$) $\cup$ lhs($ng$));
9.  setRhs($e'_k$, rhs($e_k$)$-1$);
10. replace $e_k$ by $e'_k$ ;
11. return $up\_E$;
```

contain x_k. However, there does not exist any safety condition of the form $x_k \prec x_j$ in $S(up\_E)$ because all of these explanations have been removed in line 3. Thus, $S(up\_E)$ cannot be cyclic. As we will show in Section 4, the updates performed by A_i ensure that $S(E_i)$ always remains acyclic. As a result, $S(up\_E)$ is acyclic as well, and it can be represented by a directed acyclic graph $G = (N, U)$ such that $N = \{x_1, \ldots, x_n\}$ is the set of nodes and U is the set of directed edges. An edge $(j, l) \in U$ if $(x_j \prec x_l) \in S(up\_E)$. Thus, any topological sort of G is an order that agrees with the safety conditions induced by up_E.

3.3 Decreasing termination values

Termination of Agile-ABT is based on the fact that the termination values associated with orders selected by agents decrease as search progresses. To speed up the search, Agile-ABT is written so that agents decrease termination values whenever they can. When an agent resolves its nogoods, it checks whether it can find a new order of agents such that the associated termination value is smaller than that of the current order. If so, the agent will replace its current order and termination value by those just computed, and will inform all other agents.

Assume that after resolving its nogoods, an agent A_i, decides to place x_k in the rhs of the nogood (ng) produced by the resolution and let up_E =`updateExplanations` (E_i, ng, x_k). The function `computeOrder` takes as parameter the set up_E and returns an order up_o compatible with the partial ordering induced by up_E. Let G be the acyclic directed graph associated with up_E. The function `computeOrder` works by determining, at each iteration p, the set $Roots$ of vertices that have no predecessor. As we aim at minimizing the termination value, function `computeOrder` selects the vertex x_j in $Roots$ that has the smallest domain size. This vertex is placed at the pth position and removed from G. Finally, p is incremented and all outgoing edges from x_j are removed from G.

Having proposed an algorithm that determines an order with small termination value for a given backtracking target x_k, one needs to know how to choose this variable to obtain an order decreasing more the termination value. The function `chooseVariableOrder` iterates through all variables x_k included in the nogood, computes a new order and termination value with x_k as the target (lines 22–24), and stores

```
    function computeOrder($up\_E$)
12. $G = (N, U)$ is the acyclic graph associated to $up\_E$;
13. $p \leftarrow 1$; $o$ is an array of length $n$;
14. while $G \neq \emptyset$ do
15.     $Roots \leftarrow \{x_j \in N \mid x_j$ has no incoming edges$\}$ ;
16.     $o[p] \leftarrow x_j$ such that $d_j = min\{d_k \mid x_k \in Roots\}$ ;
17.     remove $x_j$ from $G$;
18.     $p \leftarrow p + 1$;
19. return $o$;
```

the target and the associated order if it is the strongest order found so far (lines 25–29). Finally, the information corresponding to the strongest order is returned.

```
    function chooseVariableOrder($E_i, ng$)
20. $o' \leftarrow o_i$ ;  $TV' \leftarrow TV_i$ ; $E' \leftarrow nil$; $x' \leftarrow nil$;
21. for each $x_k \in ng$ do
22.     $up\_E \leftarrow$ updateExplanations($E_i$, ng, $x_k$) ;
23.     $up\_o \leftarrow$computeOrder($up\_E$);
24.     $up\_TV \leftarrow$ TV($up\_E, up\_o$);
25.     if ($up\_TV$ is smaller than $TV'$) then
26.         $x' \leftarrow x_k$;
27.         $o' \leftarrow up\_o$;
28.         $TV' \leftarrow up\_TV$;
29.         $E' \leftarrow up\_E$;
30. return $\langle x', o', TV', E' \rangle$;
```

4 The Algorithm

Each agent keeps some amount of local information about the global search, namely an AgentView, a NogoodStore, a set of explanations (E_i), a current order (o_i) and a termination value (TV_i). Agile-ABT allows the following types of messages (where A_i is the sender):

- ***ok?*** message is sent by A_i to lower agents to ask whether a chosen value is acceptable. Besides the chosen value, the ***ok?*** message contains an explanation e_i which communicates the current domain size of A_i. An ***ok?*** message also contains the current order o_i and the current termination value TV_i stored by A_i.
- ***ngd*** message is sent by A_i when all its values are ruled out by its NogoodStore. This message contains a nogood, as well as o_i and TV_i.
- ***order*** message is sent to propose a new order. This message includes the order o_i proposed by A_i accompanied by the termination value TV_i.

Agile-ABT (Figures 1 and 2) is executed on every agent A_i. After initialization, each agent assigns a value and informs lower priority agents of its decision (`CheckAgentView` call, line 32) by sending ***ok?*** messages. Then, a loop considers the reception of the possible message types. If no message is traveling through the

```
procedure Agile-ABT( )
31. $t_i \leftarrow 0$; $TV_i \leftarrow [\infty, \infty, \ldots, \infty]$; $end \leftarrow$ false; $v_i \leftarrow$ empty;
32. CheckAgentView( ) ;
33. while $(\neg end)$ do
34.     $msg \leftarrow$ getMsg( );
35.     switch $(msg.type)$ do
36.         ok?    : ProcessInfo(msg);
37.         order  : ProcessOrder(msg);
38.         ngd    : ResolveConflict(msg);
39.         stp    : $end \leftarrow$ true;

procedure ProcessInfo($msg$)
40. CheckOrder(msg.Order, msg.TV) ;
41. UpdateAgentView(msg.Assig $\cup$ lhs(msg.Exp)) ;
42. if (msg.Exp is valid) then add(msg.Exp, E);
43. CheckAgentView( ) ;

procedure ProcessOrder($msg$)
44. CheckOrder(msg.Order,msg.TV) ;
45. CheckAgentView( ) ;

procedure ResolveConflict($msg$)
46. CheckOrder(msg.Order,msg.TV) ;
47. UpdateAgentView(msg.Assig $\cup$ lhs(msg.Nogood)) ;
48. if (Coherent(msg.Nogood, AgentView $\cup$ $x_i = v_i$) $\wedge$ Compatible(msg.Nogood, $o_i$)) then
49.     add(msg.Nogood,NogoodStore); $v_i \leftarrow$ empty;
50.     CheckAgentView();
51. else if (rhs(msg.Nogood)$=v_i$) then
52.     sendMsg:ok?$(v_i, e_i, o_i, TV_i)$ to $msg.Sender$ ;

procedure CheckOrder($o, TV$)
53. if ($o$ is stronger than $o_i$ ) then $o_i \leftarrow o$; $TV_i \leftarrow TV$;
54. remove nogoods and explanations incompatible with $o_i$;

procedure CheckAgentView()
55. if ($\neg$Consistent($v_i$,AgentView)) then
56.     $v_i \leftarrow$ ChooseValue( ) ;
57.     if $(v_i)$ then sendMsg:ok?$(v_i, e_i, o_i, TV_i)$ to $Succ(A_i)$;
58.     else Backtrack();
59. else if ($o_i$ was modified) then
60.     sendMsg:ok?$(v_i, e_i, o_i, TV_i)$ to $Succ(A_i)$;

procedure UpdateAgentView($Assignments$)
61. for each var $\in$ Assignments do
62.     if (Assignments[var].t $>$ AgentView [var].t ) then
63.         AgentView [var] $\leftarrow$ Assignments[var];
64. remove nogoods and explanations incoherent with AgentView;
```

Figure 1: The Agile-ABT algorithm (Part 1).

```
    procedure Backtrack( )
65. $ng \leftarrow$ solve(NogoodStore) ;
66. if $(ng = empty)$ then $end \leftarrow$ true; sendMsg:stp(system) ;
67. $\langle x_k, o', TV', E' \rangle \leftarrow$chooseVariableOrder($E_i$, ng) ;
68. if ($TV'$ is smaller than $TV_i$ ) then
69.     $TV_i \leftarrow TV'$; $o_i \leftarrow o'$; $E_i \leftarrow E'$ ;
70.     setRhs($ng, x_k$);
71.     sendMsg:ngd($ng, o_i, TV_i$) to $A_k$ ;
72.     remove $e_k$ from $E_i$ ;
73.     broadcastMsg:order($o_i, TV_i$) ;
74. else
75.     setRhs($ng, x_k$);
76.     sendMsg:ngd($ng, o_i, TV_i$) to $A_k$ ;
77. UpdateAgentView ($x_k \leftarrow unknown$) ;
78. CheckAgentView() ;

    function ChooseValue()
79. for each ($v \in D_i$ not eliminated by NogoodStore ) do
80.     if ($\exists\, x_j \in AgentView$ such that $\neg$Consistent($v$, $x_j$)) then
81.         add($x_j$=$v_j \Rightarrow x_i \neq v$, NogoodStore) ;
82. if $(D_i = \emptyset)$ then return $(empty)$;
83. else $t_i \leftarrow t_i$+1 return $(v)$;                              /* $v \in D_i$ */
```

Figure 2: The Agile-ABT algorithm (Part 2).

network, the state of quiescence is detected by a specialized algorithm [5], and a global solution is announced. The solution is given by the current variables' assignments.

When an agent A_i receives a message (of any type), it checks if the order included in the received message is stronger than its current order o_i (`CheckOrder` call, lines 40, 44 and 46). If it is the case, A_i replaces o_i and TV_i by those newly received (line 53). The nogoods and explanations that are no longer compatible with o_i are removed to ensure that $S(E_i)$ remains acyclic (line 54).

If the message was an ***ok?*** message, the AgentView of A_i is updated to include the new assignments (`UpdateAgentView` call, line 41). Beside the assignment of the sender, A_i also takes newer assignments contained in the left hand side of the explanation included in the received ***ok?*** message to update its AgentView. Afterwards, the nogoods and the explanations that are no longer coherent with AgentView are removed (`UpdateAgentView` line 64). Then, if the explanation in the received message is valid, A_i updates the set of explanations by storing the newly received explanation. Next, A_i calls the procedure `CheckAgentView` (line 43).

When receiving an ***order*** message, A_i processes the new order (`CheckOrder`) and calls `CheckAgentView` (line 45).

When A_i receives a ***ngd*** message, it calls `CheckOrder` and `UpdateAgentView` (lines 46 and 47). The nogood contained in the message is accepted if it is coherent with the AgentView and the assignment of x_i and compatible with the current order of A_i. Otherwise, the nogood is discarded and an ***ok?*** message is sent to the sender as in ABT (lines 51 and 52). When the nogood is accepted, it is stored, acting as justification for removing the current value of A_i (line 49). A new value consistent with the AgentView is searched (`CheckAgentView` call, line 50).

The procedure `CheckAgentView` checks if the current value v_i is consistent with the AgentView. If v_i is consistent, A_i checks if o_i was modified (line 59). If so, A_i must send its assignment to lower priority agents through ***ok?*** messages. If v_i is not consistent with its AgentView, A_i tries to find a consistent value (`ChooseValue` call, line 56). In this process, some values of A_i may appear as inconsistent. In this case, the nogoods justifying their removal are added to the NogoodStore (line 81 of function `ChooseValue`). If a new consistent value is found, an explanation e_i is built and the new assignment is notified to the lower priority agents of A_i through ***ok?*** messages (line 57). Otherwise, every value of A_i is forbidden by the NogoodStore and A_i has to backtrack (`Backtrack` call, line 58).

In procedure `Backtrack`, A_i resolves its nogoods, deriving a new nogood (ng). If ng is empty, the problem has no solution. A_i terminates execution after sending a **stp** message (line 66). Otherwise, one of the agents included in ng must change its value. The function `chooseVariableOrder` selects the variable to be changed (x_k) and a new order (o') such that the new termination value TV' is as small as possible. If TV' is smaller than that stored by A_i, the current order and the current termination value are replaced by o' and TV' and A_i updates its explanations by that returned by `chooseVariableOrder` (line 69). Then, a ***ngd*** message is sent to the agent A_k owner of x_k (line 71). Then, e_k is removed from E_i since A_k will probably change its explanation after receiving the nogood (line 72). Afterwards, A_i sends an ***order*** message to all other agents (line 73). When TV' is not smaller than the current termination value, A_i cannot propose a new order and the variable to be changed (x_k) is the variable that has the lowest priority according to the current order of A_i (lines 75 and 76). Next, the assignment of x_k (the target of the backtrack) is removed from the AgentView of A_i (line 77). Finally, the search is continued by calling the procedure `CheckAgentView` (line 78).

5 Correctness and complexity

In this section we demonstrate that Agile-ABT is sound, complete and terminates, and that its space complexity is polynomially bounded.

Theorem 1 *The spatial complexity of Agile-ABT is polynomial.*

Proof. 1 The size of nogoods, explanations, termination values, and orderings, is bounded by n, the total number of variables. Now, on each agent, Agile-ABT only stores one nogood per value, one explanation per agent, one termination value and one ordering. Thus, the space complexity of Agile-ABT is in $O(nd + n^2 + n + n) = O(nd + n^2)$ on each agent. □

Theorem 2 *The algorithm Agile-ABT is sound.*

Proof. 2 Let us assume that the state of quiescence is reached. The order (say o) known by all agents is the same because when an agent proposes a new order, it sends it to all other agents. Obviously, o is the strongest order that has ever been calculated by agents. Also, the state of quiescence implies that every pair of constrained agents satisfies the constraint between them. To prove this, assume that there exist some constraints that are not satisfied. This implies that there are at least two agents A_i and A_k that do not satisfy the constraint between them. Let A_i be the agent which has the highest priority between the two agents according to o. Let v_i be the current value of A_i when the state of quiescence is reached (i.e., v_i is the most up to date assignment of A_i) and let *M* be the last ***ok?*** message sent by A_i before the state of quiescence is reached. Clearly, M contains v_i, otherwise, A_i would have sent another ***ok?*** message when it chose v_i. Moreover, when M was sent, A_i already knew the order o, otherwise A_i would have sent another ***ok?*** message when it received (or generated) o. A_i sent M to all its successors according to o (including A_k). The only case where A_k can forget v_i after receiving it is the case where A_k derives a nogood proving that v_i is not feasible. In this case, A_k should send a nogood message to A_i. If the nogood message is accepted by A_i, A_i must send an ***ok?*** message to its successors (and therefore M is not the last one). Similarly, if the nogood message is discarded, A_i have to re$-$send an ***ok?*** message to A_k (and therefore M is not the last one). So the state of quiescence implies that A_k knows both o and v_i. Thus, the state of quiescence implies that the current value of A_k is consistent with v_i, otherwise A_k would send at least a message and our quiescence assumption would be broken. □

Theorem 3 *The algorithm Agile-ABT is complete.*

Proof. 3 All nogoods are generated by logical inferences from existing constraints. Therefore, an empty nogood cannot be inferred if a solution exists. □

In order to prove that Agile-ABT terminates, we first establish two facts by proving lemmas 1 and 2.

Lemma 1 *For any agent A_i, while a solution is not found and the inconsistency of the problem is not proved, the termination value stored by A_i decreases after a finite amount of time.*

Proof. 4 Let $TV_i = [tv^1, \ldots, tv^n]$ be the current termination value of A_i. Assume that A_i reaches a state where it cannot improve its termination value. If another agent succeeds in generating a termination value smaller than TV_i, lemma 1 holds since A_i will receive the new termination value. Now assume that Agile-ABT reaches a state σ where no agent can generate a termination value smaller than TV_i. We show that Agile-ABT will exit σ after a finite amount of time. Let t be the time when Agile-ABT reaches the state σ. After a finite time δt, the termination value of each agent $A_{j\in\{1,\ldots,n\}}$ will be equal to TV_i, either because A_j has generated itself a termination value equal to TV_i or because A_j has received TV_i in an order message. Let o be the lexicographically smallest order among the current orders of all agents at time $t + \delta t$.

The termination value associated with o is equal to TV_i. While Agile-ABT is getting stuck in σ, no agent will be able to propose an order stronger than o because no agent is allowed to generate a new order with the same termination value as the one stored (Figure 2, line 68). Thus, after a finite time $\delta' t$, all agents will receive o. They will take it as their current order and Agile-ABT will behave as ABT, which is known to be complete and to terminate.

We know that $d^0_{o(1)} - tv^1$ values have been removed once and for all from the domain of the variable $x_{o(1)}$ (i.e., $d^0_{o(1)} - tv^1$ nogoods with empty lhs have been sent to $A_{o(1)}$). Otherwise, the generator of o could not have put $A_{o(1)}$ in the first position. Thus, the domain size of $x_{o(1)}$ cannot be greater than tv^1 ($d_{o(1)} \leq tv^1$). After a finite amount of time, if a solution is not found and the inconsistency of the problem is not proved, a nogood –with an empty lhs– will be sent to $A_{o(1)}$ which will cause it to replace its assignment and to reduce its current domain size ($d'_{o(1)} = d_{o(1)} - 1$). The new assignment and the new current domain size of $A_{o(1)}$ will be sent to the $(n-1)$ lower priority agents. After receiving this message, we are sure that any generator of a new nogood (say A_k) will improve the termination value. Indeed, when A_k resolves its nogoods, it computes a new order such that its termination value is minimal. At worst, A_k can propose a new order where $A_{o(1)}$ keeps its position. Even in this case the new termination value $TV'_k = [d'_{o(1)}, \dots]$ is lexicographically smaller than $TV_i = [tv^1, \dots]$ because $d'_{o(1)} = d_{o(1)} - 1 \leq tv^1 - 1$. After a finite amount of time, all agents (A_i included) will receive TV'_k. This will cause A_i to update its termination value and to exit the state σ. This completes the proof. $\square$

Lemma 2 *Let* $TV = [tv^1, \dots, tv^n]$ *be the termination value associated with the current order of any agent. We have* $tv^j \geq 0, \forall j \in 1..n$

Proof. 5 Let A_i be the agent that generated TV. We first prove that A_i never stores an explanation with a rhs smaller than 1. An explanation e_k stored by A_i was either sent by A_k or generated when calling `chooseVariableOrder`. If e_k was sent by A_k, we have $rhs(e_k) \geq 1$ because the size of the current domain of any agent is always greater than or equal to 1. If e_k was computed by `chooseVariableOrder`, the only case where $rhs(e_k)$ is made smaller than the right hand side of the previous explanation stored for A_k by A_i is in line 9 of `updateExplanations`. This happens when x_k is selected to be the backtracking target (lines 22 and 29 of `chooseVariableOrder`) and in such a case, the explanation e_k is removed just after sending the nogood message to A_k (Figure 2, line 72 of `Backtrack`). Hence, A_i never stores an explanation with a rhs equal to zero.

We now prove that it is impossible that A_i generated TV with $tv^j < 0$ for some j. From the point of view of A_i, tv^j is the size of the current domain of $A_{o(j)}$. If A_i does not store any explanation for $A_{o(j)}$ at the time it computes TV, A_i assumes that tv^j is equal to $d^0_{o(j)} \geq 1$. Otherwise, tv^j is equal to $rhs(e_{o(j)})$, where $e_{o(j)}$ was either already stored by A_i or generated when calling `chooseVariableOrder`. Now, we know that every explanation e_k stored by A_i has $rhs(e_k) \geq 1$ and we know that `chooseVariableOrder` cannot generate an explanation e'_k with $rhs(e'_k) < rhs(e_k) - 1$, where e_k was the explanation stored by

A_i (line 9 of `updateExplanations`). Therefore, we are guaranteed that TV is such that $tv^j \geq 0, \forall j \in 1..n$. □

Theorem 4 *The algorithm Agile-ABT terminates.*

Proof. 6 The termination value of any agent decreases lexicographically and does not stay infinitely unchanged (lemma 1). A termination value $[tv^1, \ldots, tv^n]$ cannot decrease infinitely because $\forall i \in \{1, \ldots, n\}$, we have $tv^i \geq 0$ (lemma 2). Hence the theorem. □

6 Experimental Results

We compared Agile-ABT to ABT, ABTDO, and ABTDO with retroactive heuristics. All experiments were performed on the DisChoco 2.0 [3] platform, [1] in which agents are simulated by Java threads that communicate only through message passing. We evaluate the performance of the algorithms by communication load and computation effort. Communication load is measured by the total number of messages exchanged among agents during algorithm execution ($\#msg$), including termination detection (system messages). Computation effort is measured by an adaptation of the number of non-concurrent constraint checks ($\#nccc$) [11] where we also count nogood checks to be closer to the actual computational effort.

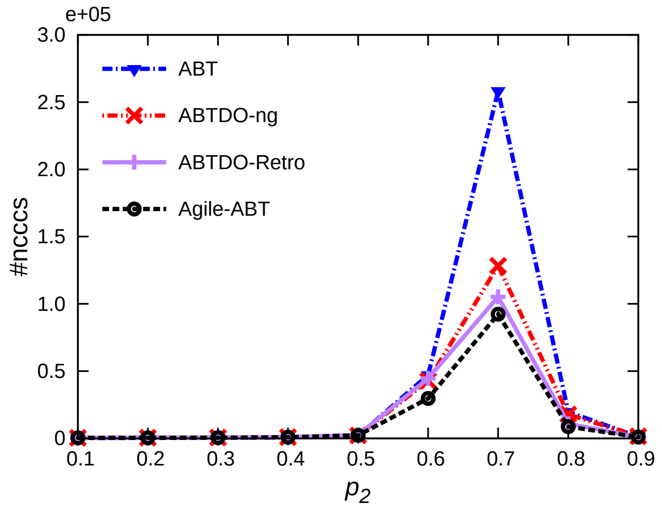
For ABT, we implemented the standard version where we use counters for tagging assignments. For ABTDO [12], we implemented the best version, using the *nogood-triggered* heuristic where the receiver of a nogood moves the sender to be in front of all other lower priority agents (denoted by ABTDO-ng). For ABTDO with retroactive heuristics [13], we implemented the best version, in which a nogood generator moves itself to be in a higher position between the last and the second last agents in the generated nogood.[2] However, it moves before an agent only if its current domain is smaller than the domain of that agent (denoted by ABTDO-Retro).

Uniform binary random DisCSPs

The algorithms are tested on uniform binary random DisCSPs that are characterized by $\langle n,\, d,\, p_1,\, p_2\rangle$, where n is the number of agents/variables, d the number of values per variable, p_1 the network connectivity defined as the ratio of existing binary constraints, and p_2 the constraint tightness defined as the ratio of forbidden value pairs. We solved instances of two classes of problems: sparse problems $\langle 20, 10, 0.2, p_2\rangle$ and dense problems $\langle 20, 10, 0.7, p_2\rangle$. We vary the tightness p_2 from 0.1 to 0.9 by steps of 0.1. For each pair of fixed density and tightness (p_1, p_2) we generated 25 instances, solved 4 times each. We report average over the 100 runs.

[1] `http://www.lirmm.fr/coconut/dischoco/`

[2] There are some discrepancies between the results reported in [13] and our version. This could be due to a bug that we fixed to ensure that ABTDO-ng and ABTDO-Retro actually terminate. You can see Appendix A and Appendix Bfor more details.



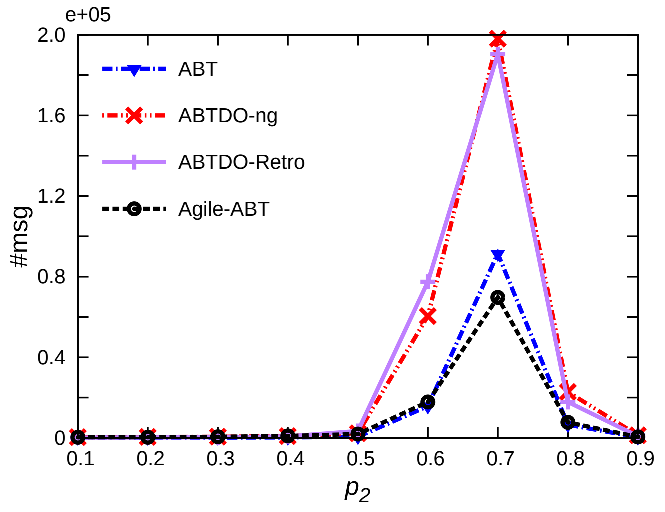
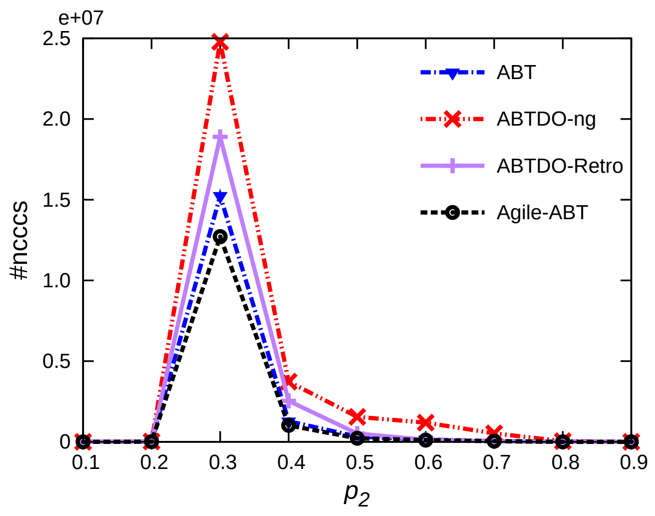


Figure 3: Total $\#msg$ exchanged and $\#ncccs$ performed on sparse problems ($p_1 = 0.2$).

Figure 3 presents the results on the sparse instances ($p_1 = 0.2$). In terms of computational effort ($\#ncccs$) (left of Figure 3), ABT is the less efficient algorithm. ABTDO-ng improves ABT by a large scale and ABTDO-Retro is more efficient than ABTDO-ng. These findings are similar to those reported in [13]. Agile-ABT outperforms all these algorithms, suggesting that on sparse problems, the more sophisticated the algorithm is, the better it is. Regarding the number of exchanged messages ($\#msg$) (right of Figure 3), the faster resolution may not translate in an overall communication load reduction. ABT requires less messages than ABTDO-ng and ABTDO-Retro. On the contrary, Agile-ABT is the algorithm that requires the smallest number of messages despite its extra messages sent by agents to notify the others of a new ordering. This is not only because Agile-ABT terminates faster than the other algorithms (see $\#ncccs$).

A second reason is that Agile-ABT is more parsimonious than ABTDO algorithms in proposing new orders. Termination values seem to focus changes on those which will pay off.



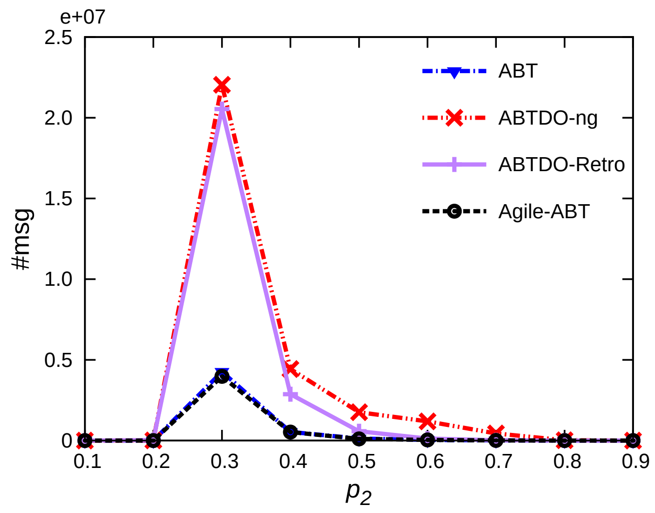


Figure 4: Total $\#msg$ exchanged and $\#ncccs$ performed on dense problems ($p_1 = 0.7$).

Figure 4 presents the results on the dense instances ($p_1 = 0.7$). Some differences appear compared to sparse problems. Concerning $\#ncccs$ (left of Figure 4), ABTDO algorithms deteriorate compared to ABT. However, Agile-ABT still outperforms all these algorithms. Regarding communication load ($\#msg$) (right of Figure 4), ABTDO-ng and ABTDO-Retro show the same bad performance as in sparse problems. Agile-ABT shows similar communication load as ABT. This confirms its good behavior observed on sparse problems.

Distributed sensor-mobile problems

The distributed sensor-mobile problem [1] is a benchmark based on a real distributed problem. It consists of n sensors that track m mobiles. Each mobile must be tracked by 3 sensors. Each sensor can track at most one mobile. A solution must satisfy visibility and compatibility constraints. The visibility constraint defines the set of sensors that are visible to each mobile. The compatibility constraint defines the compatibility among sensors. We encode SensorDCSP in DisCSP as follows. Each agent represents one mobile. There are three different variables per agent, one for each sensor that we need to allocate to the corresponding mobile. The value domain of each variable is the set of sensors that can detect the corresponding mobile. The intra-agent constraints between the variables of one agent (mobile) specify that the three sensors assigned to the mobile must be distinct and pair-wise compatible. The inter-agent constraints between the variables of different agents specify that a given sensor can be selected by at most one agent. In our implementation of the DisCSP algorithms, this encoding is translated to an equivalent formulation where we have three virtual agents for every real agent, each virtual agent handling a single variable. Problems are characterized by $\langle n,\, m,\, p_c,\, p_v\rangle$, where n is the number of sensors, m is the number of mobiles, p_c is the probability that two sensors are compatible and p_v is the probability that a sensor is visible to a mobile. We present results for class $\langle 25, 5, 0.4, p_v\rangle$ where we vary p_v from 0.1 to 0.9 by steps of 0.1 Again, for each p_v we generated 25 instances, solved 4 times each one and averaged over the 100 runs. The results are shown in Figure 5.

When comparing the speed-up of algorithms (left of Figure 5), Agile-ABT is slightly dominated by ABT and ABTDO-ng in the interval $[0.3\ 0.5]$, while outside of this interval, Agile-ABT outperforms all the algorithms. Nonetheless, the performance of ABT and ABTDO-ng dramatically deteriorate in the interval $[0.1\ 0.3]$. Concerning communication load (right of Figure 5), as opposed to other dynamic ordering algorithm, Agile-ABT is always better than or as good as standard ABT.

Discussion

From the experiments above we can conclude that Agile-ABT outperforms other algorithms in terms of computation effort ($\#ncccs$) when solving random DisCSP problem. On structured problems (SensorDCSP), our results suggest that Agile-ABT is more robust than other algorithms whose performance is sensitive to the type of problems solved. Concerning communication load ($\#msg$), Agile-ABT is more robust than other versions of ABT with dynamic agent ordering. As opposed to them, it is always better than or as good as standard ABT on difficult problems.

At first sight, Agile-ABT seems to need less messages than other algorithms but these messages are longer than messages sent by other algorithms. One could object that for Agile-ABT, counting the number of exchanged messages is biased. However, counting the number of exchanged messages would be biased only if $\#msg$ was smaller than the number of *physically* exchanged messages (going out from the network card). Now, in our experiments, they are the same. The International Organization for Standardization (ISO) has designed the Open Systems Interconnection (OSI) model to standardize networking. TCP and UDP are the principal Transport Layer protocols

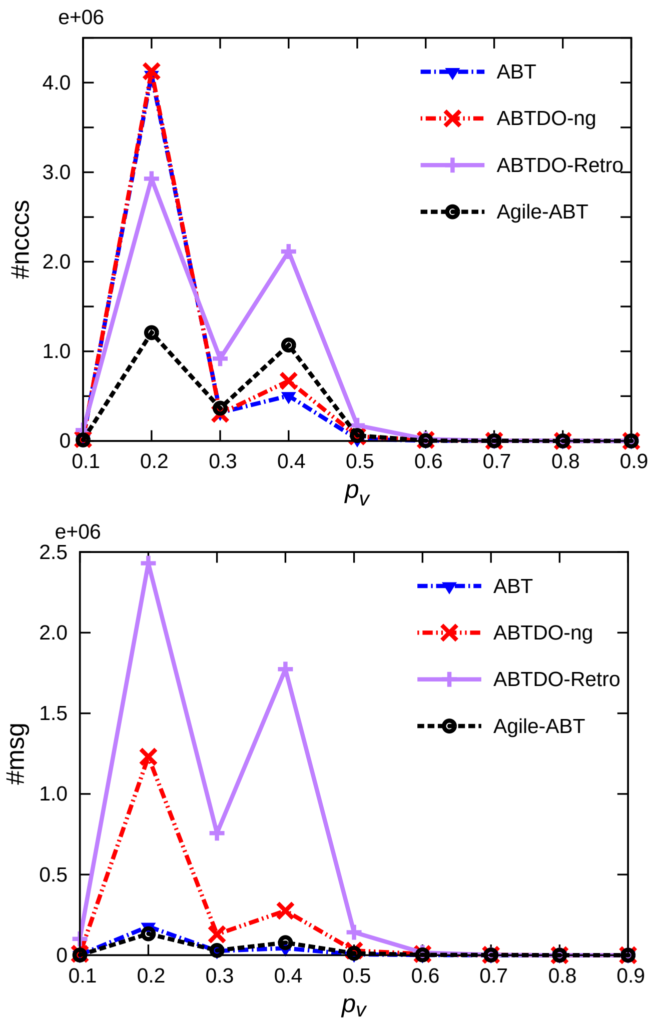
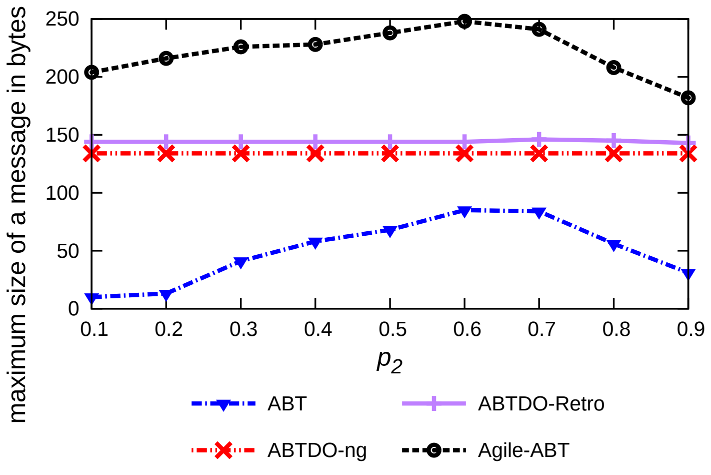


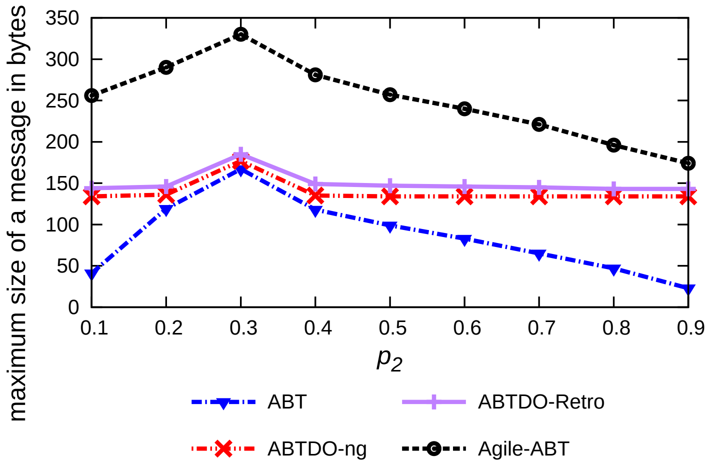
Figure 5: Total $\#msg$ exchanged and $\#ncccs$ performed on sensor-mobile problems.

using OSI model. The internet protocols IPv4 (http://tools.ietf.org/html/rfc791) and IPv6 (http://tools.ietf.org/html/rfc2460) specify the minimum datagram size that we are guaranteed to send without fragmentation of a message (in one physical message). This is 568 bytes for IPv4 and 1,272 bytes for IPv6 when using either TCP or UDP (UDP is 8 bytes less than TCP, see RFC-768 –http://tools.ietf.org/html/rfc768).

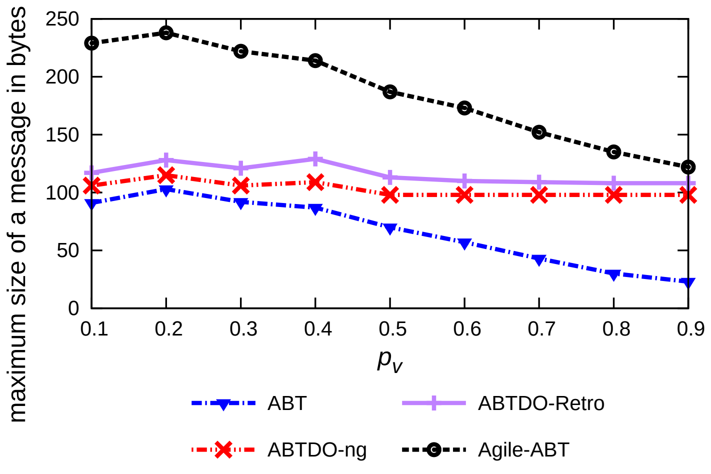
Figure 6 shows the size of the longest message sent by each algorithm on our random and sensor problems. It is clear that Agile-ABT requires lengthy messages compared to other algorithms. However, the longest message sent is always less than 568 bytes (in the worst case it is less than 350, see Figure 6(b)). In our implementation we do not proceed any message compression that would be a solution if the number of variables (n) was very large. Still, if n was so large that even the compression pro-



(a) sparse problems ($p_1 = 0.2$)



(b) dense problems ($p_1 = 0.7$)



(c) sensor-mobile problems where $p_c = 0.4$

Figure 6: Maximum message size in bytes.

tocol in the OSI model is not sufficient to fit one Agile-ABT message in one physical message, we believe that on such large problems, the exponential trend of the improvement of Agile-ABT would compensate by far for the linear overhead due to message splitting.

7 Conclusion

We have proposed Agile-ABT, an algorithm that is able to change the ordering of agents more agilely than all previous approaches. Thanks to the original concept of termination value, Agile-ABT is able to choose a backtracking target that is not necessarily the agent with the current lowest priority within the conflicting agents. Furthermore, the ordering of agents appearing before the backtracking target can be changed. These interesting features are unusual for an algorithm with polynomial space complexity. Our experiments confirm the significance of these features.

References

[1] R. Bejar, C. Domshlak, C. Fernandez, K. Gomes, B. Krishnamachari, B.Selman, and M.Valls. Sensor networks and distributed CSP: communication, computation and complexity. *Artificial Intelligence*, 161:1-2:117–148, 2005.

[2] C. Bessiere, A. Maestre, I. Brito, and P. Meseguer. Asynchronous backtracking without adding links: a new member in the ABT family. *Artificial Intelligence*, 161:1-2:7–24, 2005.

[3] R. Ezzahir, C. Bessiere, M. Belaissaoui, and E. H. Bouyakhf. Dischoco: a platform for distributed constraint programming. In *Proceeding of Workshop on DCR of IJCAI-07*, pages 16–21, 2007.

[4] Matthew L. Ginsberg and David A. McAllester. Gsat and dynamic backtracking. In *KR*, pages 226–237, 1994.

[5] K. Chandy Mani and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985. ISSN 0734-2071.

[6] A. Petcu and B. Faltings. A value ordering heuristic for distributed resource allocation. In *Proceeding of CSCLP04*, Lausanne, Switzerland, 2004.

[7] M. C. Silaghi. Generalized dynamic ordering for asynchronous backtracking on DisCSPs. In *DCR workshop, AAMAS-06*, Hakodate, Japan, 2006.

[8] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Technical report, 2001.

[9] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceeding of CP*, pages 88–102, Cassis, France, 1995.

[10] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998.

[11] R. Zivan and A. Meisels. Message delay and discsp search algorithms. *Annals of Mathematics and Artificial Intelligence*, 46(4):415–439, 2006. ISSN 1012-2443.

[12] R. Zivan and A. Meisels. Dynamic ordering for asynchronous backtracking on discsps. *Constraints*, 11(2-3):179–197, 2006. ISSN 1383-7133.

[13] R. Zivan, M. Zazone, and A.Meisels. Min-domain retroactive ordering for asynchronous backtracking. *Constraints*, 14(2):177–198, 2009. ISSN 1383-7133.

Appendices

A Appendix A: Why ABTDO is not correct

A.1 Preliminaries

We will show that ABTDO [12] is not correct. To this end, we first recall some features of this algorithm that we will need to build a counterexample:

- An agent A_i can propose a new order each time it replaces its assignment. In the new order, only the positions of the agents that have lower priority than A_i can be modified. However, these agents can not become higher than A_i.
- A nogood is a tuple of inconsistent assignments. An agent accepts a nogood if it contains no obsolete assignment. When an agent receives a new order that is more up to date than its current order, it removes the nogoods that are no longer valid according to the received order. A nogood message, contains in addition of the nogood itself, the identity of the agent that sent the nogood.
- An agent A_i can sometimes *mistakenly* receive a nogood that contains some agents that have lower priority than itself. When this occurs, A_i sends this nogood to the agent that has the lowest priority according to its current order. It will be specified in this message that the sender is A_i.

For ordering variables in ABTDO, Zivan and Meisels introduced in [12] three different heuristics. Among them, the Nogood-Triggered is the most efficient. In this heuristic, an agent A_i can change the order only after receiving a nogood eliminating its current assignment. When this occurs, A_i changes the order by placing the sender just after itself. In this note, we focus on ABTDO combined with the Nogood-Triggered heuristic (noted by ABTDO-ng).

A.2 Why the proof of ABTDO does not hold

To prove that ABTDO is correct, one needs to prove that it is sound, complete and that it terminates. The soundness of ABTDO is inherited from ABT. ABTDO is complete since an empty nogood can not be derived if a solution exists. To prove that ABTDO terminates, Zivan and Meisels first established two facts:

Lemma 3 *The highest priority agent in the initial order remains the highest priority agent in all proposed orders.*

Lemma 4 *When the highest priority agent proposes a new order, it is more up to date than all previous orders.*

Given these facts, Zivan and Meisels use induction on the number of agents in the DisCSP to prove that the algorithm terminates. For a single agent this property is verified. Assume now that this property is satisfied for any $k < n$. So this property is satisfied for $n-1$. Now, consider a DisCSP with n agents where A_1 is the highest priority agent. The position of A_1 will never be changed (lemma 3). When A_1 instantiates its value, it chooses a new order and sends it to all other agents. Now consider the DisCSP formed by the $n-1$ lower priority agents. The initial order of this DisCSP is the order sent by A_1. According to Zivan and Meisels (2006), since the algorithm terminates for $n-1$ agents, A_1 will continue to receive nogoods until the inconsistency is proved (the domain of A_1 is exhausted, for example), or a solution is found. However, to state that the DisCSP formed by the $(n-1)$ agents terminates, lemma 3 must hold for this DisCSP. In other words, while A_1 did not replace its assignment, the position of the agent that has the highest priority in the DisCSP formed by the $(n-1)$ lower priority agents can be changed only a finite number of times. Nonetheless, this is not verified by ABTDO-ng as it was presented in [12] and the algorithm may fall into an infinite loop.

Note: In the following counterexample, any agent that proposes a new order already knows the most up-to-date order generated so far. Therefore, when an agent proposes a new order, this order is more up to date than all the orders generated so far.

A.3 A counterexample

Let us consider a DisCSP instance with 5 agents $\mathcal{A} = \{A_1, A_2, A_3, A_4, A_5\}$ where $D(x_1) = D(x_4) = D(x_5) = \{1, 2\}$ and $D(x_2) = D(x_3) = \{1, 2, 3\}$. The constraints of the instance are:
$c_{12} : (x_1, x_3) \notin \{(1,1),(1,2),(1,3)\}$;
$c_{23} : (x_2, x_3) \notin \{(3,3)\}$;
$c_{24} : (x_2, x_4) \notin \{(1,1),(2,1)\}$;
$c_{25} : (x_2, x_5) \notin \{(1,1),(2,1)\}$;
$c_{34} : (x_3, x_4) \notin \{(1,2),(2,2)\}$;
$c_{35} : (x_3, x_5) \notin \{(1,2),(2,2)\}$.

Step 1:

$o_1 = [A_1, A_2, A_3, A_4, A_5]$ where $x_1 = 1$, $x_2 = 1$, $x_3 = 1$, $x_4 = 1$ and $x_5 = 1$.
All agents instantiate theirs variables to the first value in their domains and send ***ok?*** messages to their neighbours.
After receiving the assignment $(x_1 = 1)$, A_3 generates the nogood $ng_1 : \neg(x_1 = 1)$ and sends it to A_1. A_3 removes $(x_1 = 1)$ from its AgentView and instantiates its variable to 1. It sends its current value in ***ok?*** $(x_3 = 1)$ messages to all its neighbours including A_2.
After receiving ng_1, A_1 replaces its assignment and proposes a new order o_2 where A_3 is placed in the second position: $o_2 = [A_1, A_3, A_2, A_4, A_5]$. A_1 sends o_2 to all other agents A_3, A_2, A_4 and A_5. The current assignment of A_1 is $x_1 = 2$.

Step 2:

A_2 and A_3 have received $o_2 = [A_1, A_3, A_2, A_4, A_5]$.
A_2 receives the new assignment of A_3 $(x_3 = 1)$.

A_4 has received the new assignment of A_3 but it has not yet received o_2. A_4 generates $ng_2 : \neg(x_2 = 1 \wedge x_3 = 1)$ and sends it to A_3. Since A_2 has lower priority than A_3 in o_2, A_3 sends ng_2 to A_2.
A_2 accepts ng_2 since it is coherent with its AgentView. A_2 replaces its current value with 2 and proposes a new order o_3 where A_3 is placed immediately after A_2: $o_3 = [A_1, A_2, A_3, A_4, A_5]$.
The order o_3 is sent to all agents that has lower priority than A_2 in o_3 (A_3, A_4 and A_5).
A_2 removes ng_2 from its nogood-Store since it is no longer valid according to o_3.
A_2 sends an ***ok?*** $(x_2 = 1)$ message to all its neighbours including A_3.

Step 3:

A_3 has received $o_3 = [A_1, A_2, A_3, A_4, A_5]$ and the ***ok?*** $(x_2 = 1)$ message sent by A_2.
A_5 has received the new assignment of A_2.
A_5 has not yet received o_3. Until now, the order known by A_5 is $o_2 = [A_1, A_3, A_2, A_4, A_5]$.
A_5 generates a new nogood $ng_3 : \neg(x_3 = 1 \wedge x_2 = 2)$ and sends it to A_2.
When receiving ng_3, A_2 sends it to A_3 since it has the lowest priority according to o_3.
A_3 accepts ng_3 since it is coherent with its AgentView. Because of ng_3, A_3 replaces its current assignment with 2 and proposes a new order o_4 where A_2 is placed immediately after A_3: $o_4 = [A_1, A_3, A_2, A_4, A_5]$. The order o_4 is sent to all agents that has lower priority than A_3 in o_4 (A_2, A_4 and A_5). Now ng_3 is removed from the nogood-Store of A_3 since it is no longer valid according to o_4.
A_3 sends an ***ok?*** $(x_3 = 2)$ message to all its neighbours.

Step 4:

A_2 has received $o_4 = [A_1, A_3, A_2, A_4, A_5]$ and the ***ok?*** $(x_3 = 2)$ message sent by A_3.
A_4 has received the new assignment of A_3.
A_4 has not yet received o_4. Until now, the order known by A_4 is $o_3 = [A_1, A_2, A_3, A_4, A_5]$.
A_4 generates a new nogood $ng_4 :\neq (x_3 = 2 \wedge x_2 = 2)$ and sends it to A_3.
When receiving ng_4, A_3 sends it to A_2 since A_2 the lowest priority according to o_4.
A_2 accepts ng_4 since it is coherent with its AgentView. Because of ng_4, A_2 replaces its current assignment with 1 and proposes a new order o_5 where A_3 is placed immediately after A_2: $o_5 = [A_1, A_2, A_3, A_4, A_5]$. The order o_5 is sent to all agents that has lower priority than A_2 in o_5,i.e A_3, A_4 and A_5.
A_2 removes ng_4 from its nogood-Store since it is no longer valid according to o_5.
A_2 sends an ***ok?*** $(x_2 = 1)$ message to all its neighbours.

Step 5

A_3 has received $o_5 = [A_1, A_2, A_3, A_4, A_5]$ and the ***ok?*** $(x_2 = 1)$ message sent by A_2.
A_5 has received the new assignment of A_2.
A_5 has not yet received o_5. Until now, the order known by A_5 is o_4.
A_5 generates a new nogood $ng_5 : \neg(x_3 = 2 \wedge x_2 = 1)$ and sends it to A_2.

When receiving ng_5, A_2 sends it to A_3 since A_3 the lowest priority according to o_5.
A_3 accepts ng_5 since it is coherent with its AgentView. Because of ng_5, A_3 replaces its current assignment with 1 and proposes a new order o_6 where A_2 is placed immediately after A_3: $o_6 = [A_1, A_3, A_2, A_4, A_5]$. The order o_6 is sent to all agents that has lower priority than A_3 in o_6 (A_2, A_4 and A_5).
A_3 removes ng_5 from its nogood-Store since it is no longer valid according to o_6.
A_3 sends an ***ok?*** $(x_3 = 1)$ message to all its neighbours.
Hence, we come back to the order o_2 ($o_6 = o_2$) without performing any progress since all nogoods have been removed. Thus, ABTDO-ng may not terminate.

A.4 How to ensure that ABTDO actually terminate

The concern with ABTDO-ng is that an agent A_i that replaces its assignment sends ***ok?*** messages before sending ***order*** messages. As a result, another agent A_k that has not yet received the ***order*** message, can use the assignment contained in the ***ok?*** message to generate a nogood and therefore to send this nogood to the *wrong* agent. To remedy this, in our implementation, in addition to the value taken by the agent's variable, an ***ok?*** message also contains the order.

B Appendix B: Retroactive Dynamic Ordering for Asynchronous Backtracking algorithm may not terminate

B.1 Preliminaries

Zivan and Meisels (2006) proposed Dynamic Ordering for Asynchronous Backtracking (ABTDO). In this algorithm, when an agent assigns a value to its variable, it can reorder lower priority agents. Each agent in ABTDO holds a current order which is an ordered list of pairs. Every pair includes the ID of one of the agents and a counter. The counters attached to each agent ID in the order list form a time-stamp. Initially, all time-stamp counters are zero and all agents start with the same order. Each agent that proposes a new order increments its counter by one and sets to zero counters of all lower priority agents (the counters of higher priority agents are not modified). When comparing two orders, the most up-to-date is the one with the lexicographically *larger* time-stamp. In other words, the most up-to-date order is the one for which the first different counter is larger.

A new kind of ordering heuristics for ABTDO is presented in [13]. These heuristics, called retroactive heuristics, enable the generator of the nogood to be moved to a higher position than that of the target of the backtrack. In [13], ties which could not have been generated in standard ABTDO, are broken using the agents indexes. In other words, when two contradictory orders have the same time-stamp, the most up-to-date order is the one for which the index of the first different agent is smaller. The degree of flexibility of these heuristics is dependent on the size of the nogood storage capacity, which is predefined. Agents are limited to store nogoods smaller or equal to a predefined size K. The space complexity of the agents is thus exponential in K.

However, the best heuristic proposed in [13] does not require this exponential storage of nogoods. In this heuristic called ABTDO-Retro-MinDom, agents that generate a nogood are placed in the new order between the last and the second last agents in the generated nogood. However, agents are moved to a higher position only if their domain is smaller than that of the agents they pass on the way up. Otherwise, the generator of the nogood is placed right after backtracking target.

B.2 ABTDO-Retro-MinDom may not terminate: a counterexample

We will now show that ABTDO-Retro-MinDom may not terminate. To this end, consider a DisCSP of 5 agents $\{A_1, A_2, A_3, A_4, A_5\}$. We assume that, initially, all agents store the same order $o_1 = [A_1, A_5, A_4, A_2, A_3]$ with $s_1 = [0, 0, 0, 0, 0]$. We assume that: $D(x_2) = D(x_3) = D(x_4) = \{6, 7\}$ and $D(x_1) = D(x_5) = \{1, 2, 3, 4, 5\}$.
The constraints are:
$c_{12} : (x_1, x_2) \notin \{(1, 6), (1, 7)\}$;
$c_{13} : (x_1, x_3) \notin \{(2, 6), (2, 7)\}$;
$c_{14} : (x_1, x_4) \notin \{(1, 6), (1, 7)\}$;
$c_{24} : (x_2, x_4) \notin \{(6, 6), (7, 7)\}$.

$c_{35} : (x_1, x_5) \notin \{(6,4),(6,3),(7,5)\}$.

In the following we give a possible execution of ABTDO-Retro-MinDom.

t_0: All agents assigns their variables to the first values in their domains and send ***ok?*** messages to their neighbours.

t_1: A_4 receives the first ***ok?*** ($x_1 = 1$) message sent by A_1 and generates a nogood $ng_1 : \neg(x_1 = 1)$. Then, it proposes a new order $o_2 = [A_4, A_1, A_5, A_2, A_3]$ with $s_2 = [1,0,0,0,0]$. Afterwards, it assigns the value 6 to its variable and sends ***ok?*** ($x_4 = 6$) message to all its neighbours (including A_2).

t_2: A_3 receives $o_2 = [A_4, A_1, A_5, A_2, A_3]$ and deletes o_1 since o_2 is more up-to-date;
A_1 receives the nogood sent by A_4, it replaces its assignment by 2 and sends an ***ok?*** ($x_1 = 2$) message to all its neighbours.

t_3: A_2 has not yet received o_2 and the new assignment of A_1. A_2 generates the same nogood us ng_1, $ng_2 : \neg(x_1 = 1$ and proposes a new order $o_3 = [A_2, A_1, A_5, A_4, A_3]$ with $s_3 = [1,0,0,0,0]$;
Afterwards, it assigns the value 6 to its variable and sends ***ok?*** ($x_2 = 6$) message to all its neighbours (including A_4).

t_4: A_4 receives the new assignment of A_2 (i.e. $x_4 = 6$) and $o_3 = [A_2, A_1, A_5, A_4, A_3]$. Afterwards, it discards o_2 since o_3 is more up-to-date;
Then, A_4 tries to satisfy c_{24} because A_2 has a higher priority according to o_3. Hence, A_4 replaces its current assignment (i.e $x_4 = 6$) by $x_4 = 7$ and sends an ***ok?*** ($x_4 = 7$) message to all its neighbours (including A_2).

t_6: After receiving the new assignment of A_1 (i.e $x_1 = 2$) and before receiving $o_3 = [A_2, A_1, A_5, A_4, A_3]$, A_3 generates a nogood $ng_3 : \neg(x_1 = 2)$ and proposes a new order $o_4 = [A_4, A_3, A_1, A_5, A_2]$ with $s_4 = [1,1,0,0,0]$;
The order o_4 is more up-to-date than o_3.
Since in ABTDO, an agent sends the new order only to lower priority agents, A_3 will not send o_4 to A_4 because it is a higher priority agent.

t_8: A_2 receives o_4 but it has not yet received the new assignment of A_4. Then, it tries to satisfy c_{24} because A_4 has a higher priority according to its current order o_4. Hence, A_2 replaces its current assignment (i.e $x_2 = 6$) by $x_2 = 7$ and sends an ***ok?*** ($x_2 = 7$) message to all its neighbours (including A_4).

$t > t_8$: ABTDO-Retro-MinDom will not terminate if A_2 and A_4 always change their values simultaneously.

$$
\begin{aligned}
o_1 &= [\, A_1\,, A_5\,, A_4\,, A_2\,, A_3 \,] & s_1 &= [\, 0\,, 0\,, 0\,, 0\,, 0 \,] \\
o_2 &= [\, A_4\,, A_1\,, A_5\,, A_2\,, A_3 \,] & s_2 &= [\, 1\,, 0\,, 0\,, 0\,, 0 \,] \\
o_3 &= [\, A_2\,, A_1\,, A_5\,, A_4\,, A_3 \,] & s_3 &= [\, 1\,, 0\,, 0\,, 0\,, 0 \,] \\
o_4 &= [\, A_4\,, A_3\,, A_1\,, A_5\,, A_2 \,] & s_4 &= [\, 1\,, 1\,, 0\,, 0\,, 0 \,]
\end{aligned}
$$

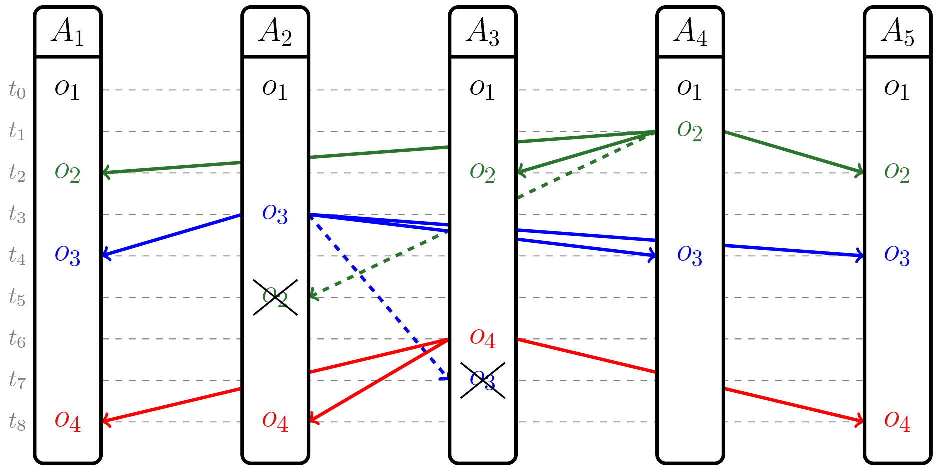


Figure 7: The schema of exchanging ***order*** messages by ABTDO-Retro

B.3 How to ensure that ABTDO-Retro-MinDom actually terminate

To ensure that ABTDO-Retro-MinDom actually terminate, we have to simply make sure that after a finite time, all agents that share a constraint agree on the order. A simple way to this is to send the order in the ***ok?*** messages (***ok?*** messages are sent to all neighbours). Of course, an agent that proposes a new order should also send order messages to lower priority agents that not share a constraint with it.