Agile Asynchronous Backtracking for Distributed Constraint Satisfaction Problems

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

[1] LIRMM/CNRS, University Montpellier II, France
[2] LIMIARF/FSR, University Mohammed V Agdal, Morroco
{bessiere,wahbi}@lirmm.fr, bouyakhf@fsr.ac.ma
ymechqrane@spsm.ma

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 is **compatible** with an order o_i if all agents in $lhs(nogood)$ appear before $rhs(nogood)$ 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 $[tv^1, \ldots, tv^n]$, 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 smaller being the stronger.

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(`E_i, ng, x_k`)`

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

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 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 2. 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

function `computeOrder(`up_E`)`

7. $G = (N, U)$ is the acyclic graph associated to up_E;
8. $p \leftarrow 1$; o is an array of length n;
9. **while** $G \neq \emptyset$ **do**
10. $Roots \leftarrow \{x_j \in N \mid x_j$ has no incoming edges$\}$;
11. $o[p] \leftarrow x_j$ such that $d_j = min\{d_k \mid x_k \in Roots\}$;
12. `remove` x_j from G; $p \leftarrow p + 1$;
13. **return** o;

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 16–17), and stores the target and the associated order if it is the strongest order found so far (lines 18–19). Finally, the information corresponding to the strongest order is returned.

function `chooseVariableOrder(`E_i, ng`)`

14. $o' \leftarrow o_i$; $TV' \leftarrow TV_i$; $E' \leftarrow nil$; $x' \leftarrow nil$;
15. **for each** $x_k \in ng$ **do**
16. $up\_E \leftarrow$ `updateExplanations(`E_i, ng, x_k`)` ;
17. $up\_o \leftarrow$`computeOrder(`up_E`)`; $up\_TV \leftarrow$ TV(up_E, up_o) ;
18. **if** (up_TV *is* `smaller than` TV') **then**
19. $x' \leftarrow x_k$; $o' \leftarrow up\_o$; $TV' \leftarrow up\_TV$; $E' \leftarrow up\_E$;
20. **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 (Figure 1 and 2) is executed on every agent. After initialization, each agent assigns a value and informs lower priority agents of its decision (`CheckAgentView` call, line 22) by sending **ok?** messages. Then, a loop considers the

procedure `Agile-ABT ( )`

21. $t_i \leftarrow 0$; $TV_i \leftarrow [\infty, \infty, \ldots, \infty]$; $end \leftarrow$ false; $v_i \leftarrow$ empty;
22. `CheckAgentView( )` ;
23. **while** $(\neg end)$ **do**
24. $msg \leftarrow$ `getMsg( )`;
25. **switch** $(msg.type)$ **do**
26. **ok?** : `ProcessInfo(`*msg*`)`;
27. **order**: `ProcessOrder(`*msg*`)`;
28. **ngd** : `ResolveConflict(`*msg*`)`;
29. **stp** : $end \leftarrow$ true;

procedure `ProcessInfo(`msg`)`

30. `CheckOrder(`*msg.Order, msg.TV*`)` ;
31. `UpdateAgentView(`*msg.Assig* $\cup$ `lhs(`*msg.Exp*`))` ;
32. **if** *msg.Exp is valid* **then** `add(`*msg.Exp, E*`)`;
33. `CheckAgentView( )` ;

procedure `ProcessOrder(`msg`)`

34. `CheckOrder(`*msg.Order,msg.TV*`)` ;
35. `CheckAgentView( )` ;

procedure `ResolveConflict(`msg`)`

36. `CheckOrder(`*msg.Order,msg.TV*`)` ;
37. `UpdateAgentView(`*msg.Assig* $\cup$ `lhs(`*msg.Nogood*`))` ;
38. **if** `Coherent(`*msg.Nogood, AgentView* $\cup$ x_i=v_i`)` **and** `Compatible(`*msg.Nogood,* o_i`)` **then**
39. `add(`*msg.Nogood,NogoodStore*`)`; $v_i \leftarrow$ empty;
40. `CheckAgentView( )`;
41. **else if** `rhs(`*msg.Nogood*`)`=v_i **then**
42. `sendMsg:` **ok?**(v_i, e_i, o_i, TV_i) to $msg.Sender$;

procedure `CheckOrder(`o, TV`)`

43. **if** *o is stronger than* o_i **then** $o_i \leftarrow o$; $TV_i \leftarrow TV$;
44. remove nogoods and explanations incompatible with o_i;

procedure `CheckAgentView( )`

45. **if** $\neg$`Consistent(`v_i*,AgentView*`)` **then**
46. $v_i \leftarrow$ `ChooseValue( )` ;
47. **if** (v_i) **then** `sendMsg:` **ok?**(v_i, e_i, o_i, TV_i) to $Succ(A_i)$;
48. **else** `Backtrack( )`;
49. **else if** $(o_i$ *was modified*$)$ **then**
50. `sendMsg:` **ok?**(v_i, e_i, o_i, TV_i) to $Succ(A_i)$;

procedure `UpdateAgentView(`$Assignments$`)`

51. **for each** *var* $\in$ *Assignments* **do**
52. **if** Assignments[var].c $>$ AgentView [var].c **then**
53. AgentView [var] $\leftarrow$ Assignments[var];
54. remove nogoods and explanations incoherent with AgentView;

Fig. 1. The Agile-ABT algorithm (Part 1).

```
procedure Backtrack( )
55.  ng ← solve(NogoodStore) ;
56.  if (ng = empty) then  end ← true; sendMsg:stp(system) ;
57.  ⟨x_k, o', TV', E'⟩ ←chooseVariableOrder(E_i, ng) ;
58.  if (TV' is smaller than TV_i ) then
59.     TV_i ← TV'; o_i ← o'; E_i ← E' ;
60.     setRhs(ng, x_k);
61.     sendMsg:ngd(ng, o_i, TV_i) to A_k ;
62.     remove e_k from E_i ;
63.     broadcastMsg:order(o_i, TV_i) ;
64.  else
65.     setRhs(ng, x_k);
66.     sendMsg:ngd(ng, o_i, TV_i) to A_k ;
67.  UpdateAgentView (x_k ← unknown) ;
68.  CheckAgentView( ) ;

function ChooseValue( )
69.  for each (v ∈ D_i not eliminated by NogoodStore ) do
70.     if (∃ x_j ∈ AgentView such that ¬Consistent(v, x_j )) then
71.        add(x_j=v_j ⇒ x_i ≠ v, NogoodStore) ;
72.  if (D_i = ∅) then  return (empty);
73.  else  t_i ← t_i+1 return (v);                          /* v ∈ D_i */
```

Fig. 2. The Agile-ABT algorithm (Part 2).

reception of the possible message types. If no message is traveling through the 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 30, 34 and 36). If it is the case, A_i replaces o_i and TV_i by those newly received (line 43). The nogoods and explanations that are no longer compatible with o_i are removed to ensure that $S(E_i)$ remains acyclic (line 44).

If the message was an **ok?** message, the AgentView of A_i is updated to include the new assignments (`UpdateAgentView` call, line 31). 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 54). 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 33).

When receiving an **order** message, A_i processes the new order (`CheckOrder`) and calls `CheckAgentView` (line 35).

When A_i receives a **ngd** message, it calls `CheckOrder` and `UpdateAgentView` (lines 36 and 37). 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 41 and 42). When the nogood is accepted, it is stored, acting as justification for removing the current value of A_i (line 39). A new value consistent with the AgentView is searched (`CheckAgentView` call, line 40).

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 49). 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 46). 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 71 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 47). Otherwise, every value of A_i is forbidden by the NogoodStore and A_i has to backtrack (`Backtrack` call, line 48).

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 56). 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 59). Then, a **ngd** message is sent to the agent A_k owner of x_k (line 61). Then, e_k is removed from E_i since A_k will probably change its explanation after receiving the nogood (line 62). Afterwards, A_i sends an **order** message to all other agents (line 63). 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 65 and 66). Next, the assignment of x_k (the target of the backtrack) is removed from the AgentView of A_i (line 67). Finally, the search is continued by calling the procedure `CheckAgentView` (line 68).

5 Correctness and Complexity

We demonstrate that Agile-ABT is sound, complete and terminates, with a polynomial space complexity.

Theorem 1. *Agile-ABT requires $O(nd + n^2)$ space per agent.*

Theorem 2. *Agile-ABT is sound.*

Proof. (Sketch) When the state of quiescence is reached, all agents necessarily know the order o that is the strongest ever computed. In addition, all agents know the most up to date assignments of all their predecessors in o. Thus, any constraint c_{ik} between

agents A_i and A_k has been successfully checked by the agent with lowest priority in o. Otherwise that agent would have tried to change its value and would have either sent an ok? or a ngd message, breaking the quiescence. □

Theorem 3. *Agile-ABT is complete.*

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

The proof of termination is built on lemmas 1 and 2.

Lemma 1. *For every agent A_i, while no solution is 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. (Sketch) If an agent gets stuck a sufficiently long time with the same termination value, all agents will eventually have that same termination value and the order o to which it was attached. At this point, Agile-ABT works exactly like ABT, which is complete and terminates. Thus, either a solution is found or the first agent in the current order, $A_{o(1)}$, will receive a nogood with empty lhs that prunes one of its remaining values. As soon as $A_{o(1)}$ has sent its new domain size (smaller than tv^1) to the lower agents in o, any backtracking agent will generate a smaller termination value. □

Lemma 2. *For any termination value $TV = [tv^1, \ldots, tv^n]$ generated by an agent, we have $tv^j \geq 0, \forall j \in 1..n$*

Proof. (Sketch) All explanations e_k stored by an agent A_i have $rhs(e_k) \geq 1$ because it represents the current domain size of A_k. Now, termination values are built with rhs of explanations. The only case where a tv^j can be zero (line 4) is when $A_{o(j)}$ is selected by A_i to be the backtracking target, and in such a case, the explanation $e_{o(j)}$ is removed just after sending the nogood to $A_{o(j)}$ (line 62). Hence, A_i never stores an explanation e_k with $rhs(e_k) = 0$ and cannot produce a termination value with a negative element. □

Theorem 4. *Agile-ABT terminates.*

Proof. Direct from Lemmas 1 and 2. □

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

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

messages). Computation effort is measured by an adaptation of the number of non-concurrent constraint checks ($\#nccc$) [12] 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 [11], 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. However, it moves before an agent only if its current domain is smaller than the domain of that agent (denoted by ABTDO-Retro).

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 $\langle 20, 10, 0.20, p_2 \rangle$ and dense problems $\langle 20, 10, 0.70, p_2 \rangle$. We vary the tightness p_2 from 0.10 to 0.90 by steps of 0.10. 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.

Figure 3 presents the results on the sparse instances ($p_1 = 0.20$). In terms of computational effort ($\#ncccs$) (top 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 ($\#msg$) (bottom 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, even compared to ABT. 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 presents the results on the dense instances ($p_1 = 0.70$). Some differences appear compared to sparse problems. Concerning $\#ncccs$ (top of figure 4), ABTDO algorithms deteriorate compared to ABT. However, Agile-ABT still outperforms all these algorithms. Regarding communication load ($\#msg$) (bottom of figure 4), ABTDO-ng and ABTDO-Retro show the same bad performance as in sparse problems. On the contrary, Agile-ABT is approximately as good as ABT. This confirms its good behavior observed on sparse problems.

From these experiments we can conclude that Agile-ABT outperforms other algorithms in terms of computation load ($\#ncccs$) on all types of problems tested. 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.

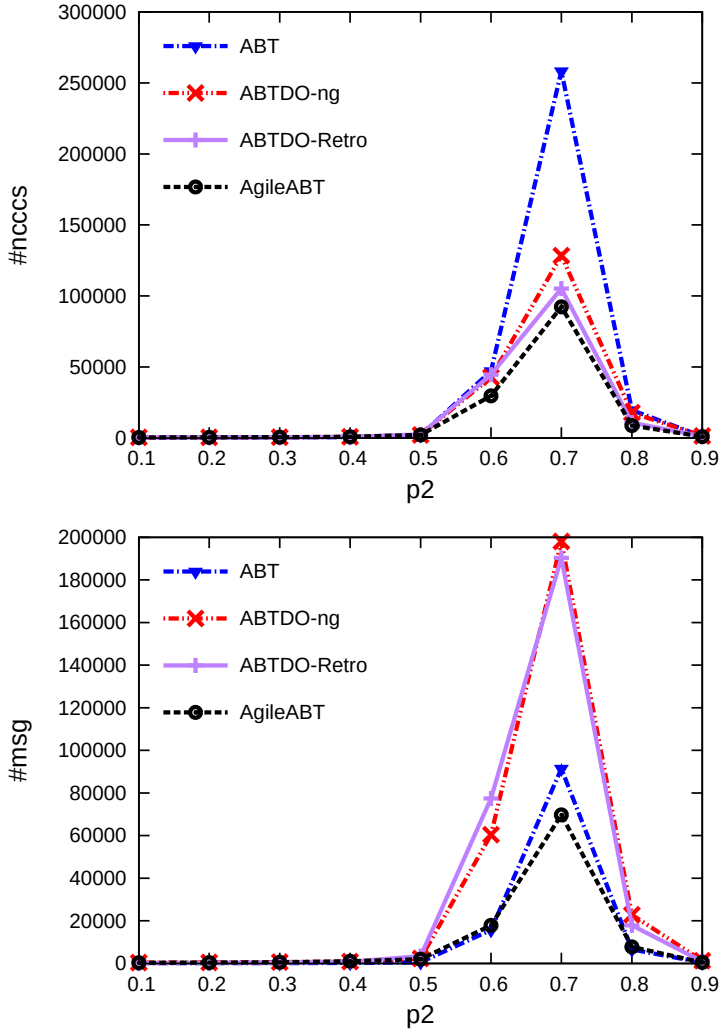

Fig. 3. Total $\#msg$ exchanged and $\#ncccs$ performed on sparse graphs ($p_1 = 0.20$).

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.

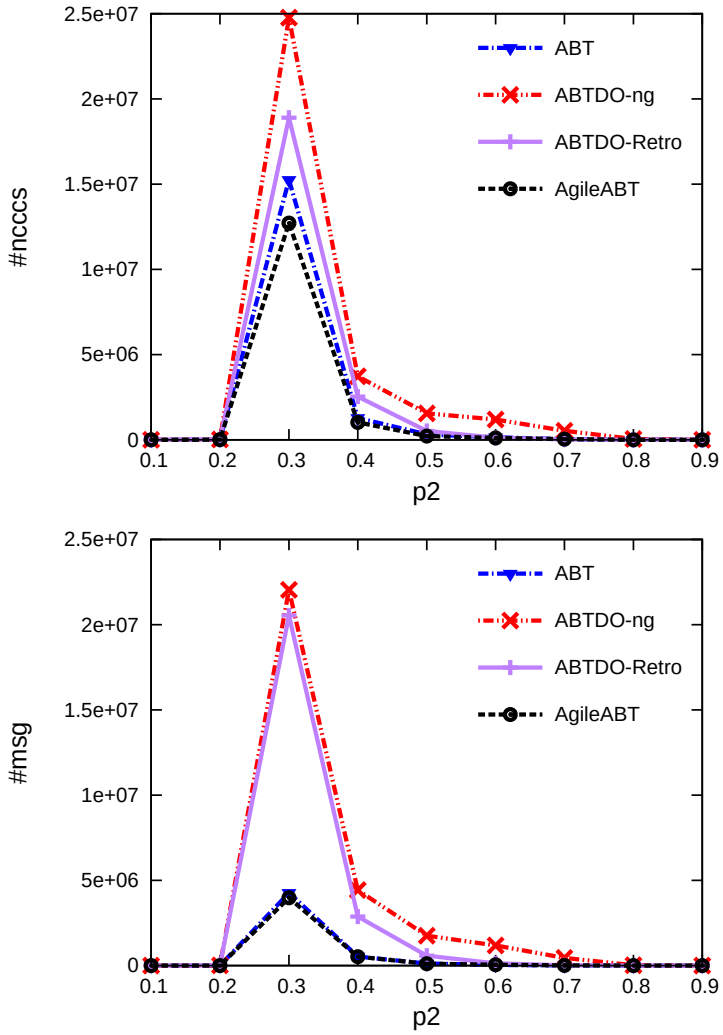

Fig. 4. Total $\#msg$ exchanged and $\#ncccs$ performed on dense graphs ($p_1 = 0.70$).

Bibliography

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

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

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

[4] Ginsberg, M.L., McAllester, D.A.: Gsat and dynamic backtracking. In: KR. pp. 226–237 (1994)

[5] Mani, K.C., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems 3(1), 63–75 (1985)

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

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

[8] Silaghi, M.C., Sam-Haroud, D., Faltings, B.: Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Tech. rep. (2001)

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

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

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

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

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