Reordering All Agents in Asynchronous Backtracking for Distributed Constraint Satisfaction Problems

Younes Mechqrane[a], Mohamed Wahbi[b,*], Christian Bessiere[c], Kenneth N. Brown[b]

[a] *Mohammed V University, Rabat, Morocco*
[b] *Insight Centre for Data Analytics, University College Cork, Cork, Ireland*
[c] *CNRS, University of Montpellier, Montpellier, France*

Abstract

Distributed constraint satisfaction problems (DisCSPs) can express decision problems where physically distributed agents control different decision variables, but must coordinate with each other to agree on a global solution. Asynchronous Backtracking (ABT) is a pivotal search procedure for DisCSPs. ABT requires a static total ordering on the agents. However, reordering agents during search is an essential component for efficiently solving a DisCSP. All polynomial space algorithms proposed so far to improve ABT by reordering agents during search only allow a limited amount of reordering. In this paper, we propose AgileABT, a general framework for reordering agents asynchronously that is able to change the ordering of *all* agents. This is done via the original notion of *termination value*, a label attached to the orders exchanged by agents during search. We prove that AgileABT is sound and complete. We show that, thanks to termination values, our framework allows us to implement the main variable ordering heuristics from centralized CSPs, which until now could not be applied to the distributed setting. We prove that AgileABT terminates and has a polynomial space complexity in all these cases. Our empirical study shows the significance of our framework compared to state-of-the-art asynchronous dynamic ordering algorithms for solving distributed CSP.

Keywords: Distributed Constraint Reasoning, Asynchronous Backtracking, Dynamic Variable Ordering.

1. Introduction

Distributed artificial intelligence involves numerous combinatorial problems where multiple physically distributed entities, called agents, need to cooperate

*Corresponding author
Email addresses: ymechqrane@gmail.com (Younes Mechqrane), mohamed.wahbi@insight-centre.org (Mohamed Wahbi), bessiere@lirmm.fr (Christian Bessiere), ken.brown@insight-centre.org (Kenneth N. Brown)

in order to find a consistent combination of actions. Examples of such problems are: traffic light synchronization [1], truck task coordination [2], target tracking in distributed sensor networks [3, 4, 5], distributed scheduling [6], distributed planning [7], nurse shift assignment problem [8], distributed resource allocation [9], distributed vehicle routing [10], etc. In these problems agents have to achieve the combination in a distributed way and without centralization. In general, this condition is mainly motivated by privacy and/or security requirements: constraints or possible values may be strategic information that should not be revealed to other agents that can be seen as competitors. In addition, in many distributed settings, gathering the whole knowledge into a centralized agent may be impractical or its cost may be intolerable. In the field of multi-agent coordination, the above-mentioned problems were formalized using the distributed constraint satisfaction problem (DisCSP) paradigm that allows a distributed solving process.

A DisCSP is composed of multiple agents, each owning its local constraint network. Variables in different agents are connected by constraints. The agent community must assign a value to each variable so that all constraints are satisfied. To achieve this, agents assign values to their own variables that satisfy their own constraints; to satisfy constraints involving variables with other agents, they must exchange messages about their decisions and revise those decisions accordingly.

During the last two decades, many distributed algorithms have been designed for solving DisCSPs, among which Asynchronous Backtracking (ABT) is the central one [11, 12]. ABT is an asynchronous algorithm executed concurrently and autonomously by all agents in the distributed problem. Agents are not required to wait for the decisions of other agents. ABT assumes a single static total priority order on the agents. For a given agent, those agents that appear before it in the order are higher priority, while those that appear after it are lower priority. When an agent performs an assignment, it sends out messages to lower priority neighbors informing them about its new assignment. Each agent tries to find an assignment satisfying the constraints with what is currently known from higher priority neighbors, and whenever an agent detects a dead end, it determines a conflict set of assignments (called a no-good) that is responsible for the inconsistency. Because a superset of a no-good cannot be a solution, the generator of the no-good sends a message to the lowest priority agent involved in the conflict set (i.e., the *backtracking target*) in order to revise its current assignment.

A strong weakness in ABT is the static order on the agents. It is known from centralized CSPs that reordering variables dynamically during search dramatically improves the efficiency of the search procedure [13, 14, 15]. Hence, several extensions of ABT have been proposed to dynamically reorder variables during search, leading to a more flexible exploration of the search space. Silaghi et al. [16] proposed *asynchronous backtracking with reordering* (ABTR). ABTR is an asynchronous complete algorithm with polynomial space requirements where abstract agents fulfill the reordering operation. Zivan and Meisels [17] proposed *dynamic ordering for asynchronous backtracking* (ABT_DO). Three different

ordering heuristics were proposed to reorder lower priority agents in ABT_DO: random, min-domain [18] and no-good-triggered. The experimental results in [17] show that no-good-triggered, where the generator of the no-good is placed just after the target of the backtrack, is the best. Silaghi [19] has shown that ABT_DO, when used with no-good-triggered, is equivalent to ABTR when used with the ABTR-db dynamic-backtracking self redelegation heuristic.

While the above-mentioned algorithms have led to great improvement in performance compared to ABT, they all share the same weakness. Whenever a no-good is discovered, the agent that must change its assignment is that with the lowest priority among the conflicting set, and no lower agent can be moved to a position higher than this target of the backtrack. This restriction is a major source of inefficiency for these algorithms. If a bad variable assignment is made high in the agents order, an exhaustive search of lower priority agents must be performed before being able to reach back to the culprit variable.

A new kind of ordering heuristics for ABT_DO is presented in [20, 21]. These heuristics enable the reordering of agents that are higher than the backtracking target. The resulting algorithm is called ABT_DO_Retro. The degree of flexibility of the heuristics ABT_DO_Retro can implement depends on the value of a predefined parameter K that determines the no-good storage capacity. Agents are limited to store no-goods with a size equal to or smaller than K. The space complexity of ABT_DO_Retro agents is thus exponential in K.

Finally, *asynchronous weak commitment* (AWC), proposed by Yokoo [22], is the algorithm that has the highest degree of flexibility to reorder agents during search. AWC dynamically reorders agents by moving the sender of a no-good higher in the order than the other agents in the no-good. Thus, when a dead end occurs, AWC is not committed to the current partial assignment of agents. It starts constructing a new partial assignment from scratch. AWC stores all the abandoned partial assignments in order to ensure termination. AWC can be seen as the special case of ABT_DO_Retro when $K = n$. AWC was shown to outperform ABT on small problems [23]. However, AWC requires an exponential space for storing all generated no-goods. This high space complexity prevents its use on larger problems.

All algorithms we discussed have shown empirically the benefit of reordering agents during distributed asynchronous search. However, we observe that those that allow the greatest flexibility (i.e., AWC and ABT_DO_Retro) pay it at the cost of an exponential space complexity for storing no-goods.

In this paper, we propose agile asynchronous backtracking, AgileABT, a distributed constraint satisfaction framework that allows reordering of all agents during search without requiring exponential space. Agents operate asynchronously, and at any stage an agent may propose a reordering. Each proposal must be associated with some auxiliary information, which we call a termination value. Agents accept or reject the suggested reordering using a priority relation over termination values. The termination values are mathematical objects that could be simple scalar values, or could be more complex structures based on the intrinsic properties of the proposed reordering. The general framework AgileABT is instantiated by specifying a function used to compute new orders

and their associated termination values together with the priority relation over the computed termination values. If the priority relation is a well ordering, then AgileABT is guaranteed to terminate; if the function used to compute new orders and their associated termination values has polynomial space complexity, then AgileABT has polynomial space complexity as well.

The general framework AgileABT can be instantiated in several possible ways. We only present some examples in this paper. In our examples, any agent can propose a reordering of all other agents, including those appearing before the backtrack target, provided that the termination value is improved w.r.t. the priority relation over the termination values. As a consequence of this agile reordering capability, an agent is able to propose any other conflicting agent as a backtracking target, provided that the target agent is the last among the conflicting ones in the new ordering. These features are unique for a DisCSP algorithm with polynomial space complexity. Our motivation was to study in a distributed setting some of the most effective DVOs heuristics from centralized CSP. We show how to implement the most common DVOs from CSP, and we evaluate their performance on DisCSP benchmarks. Our empirical results show that the DVOs implemented in AgileABT can offer orders of magnitude improvement in both computation and messaging costs compared to the original static ABT, and that they consistently outperform previous proposals for dynamic ordering in ABT.

The reminder of the paper is organized as follows. We give the necessary background on distributed CSP and dynamic reordering materials in Section 2. Section 3 introduces the general framework of Agile Asynchronous Backtracking, AgileABT and analyses its theoretical properties. Section 4 presents some instantiation examples of AgileABT and their proof of correctness. We report our empirical results in Section 6. Finally, we conclude in Section 7.

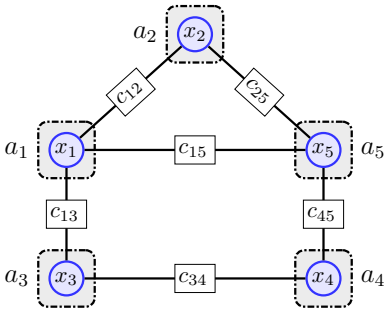
2. Background

Constraint programming is one of the main fields in artificial intelligence for studying combinatorial problems. Constraint Programming is based on approaches to solving a generic problem definition (*constraint satisfaction problem*) over a *constraint network*.

A constraint network is defined by a set of decision variables, their domain of possible values, and a set of constraints. Constraints represent restrictions on value combinations allowed for constrained variables. A solution is an assignment of values to decision variables that satisfies all the constraints. CSP is a general framework that can formalize many real world combinatorial problems whenever the knowledge about the whole problem is available for a (centralized) solver.

2.1. Basic definitions and notations

The *distributed constraint satisfaction problem* (DisCSP) consists in looking for solutions to a distributed constraint network. A distributed constraint



$\mathcal{A} = \{a_1, \ldots, a_5\}$
$\mathcal{X} = \{x_1, \ldots, x_5\}$
$\mathcal{D} = \{D_1, \ldots, D_5\}$,
where $D_i = \{1, 2, 3, 4\}$
$\mathcal{C} = \{c_{12}, c_{13}, c_{15}, c_{25}, c_{34}, c_{45}\}$

$c_{12} : x_1 \neq x_2$
$c_{13} : x_1 \neq x_3$
$c_{15} : x_1 \neq |x_5 - 2|$
$c_{25} : x_2 \neq x_5$
$c_{34} : x_3 < x_4$
$c_{45} : x_4 \geq x_5$

Figure 1: The constraint graph of a DisCSP instance of 5 agents/variables and 6 constraints.

network has been defined in [11] as a tuple $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{A}$ is a set of m agents $\{a_1, \ldots, a_m\}$, $\mathcal{X}$ is a set of n 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 n domains, where D_i is the initial set of possible values to which variable x_i may be assigned. During search, values may be pruned from the domain. At any node, the set of possible values for variable x_i is denoted by D_i^c and is called the *current domain* of x_i. Only the agent which controls a variable has knowledge of its current domain and can assign it a value. $\mathcal{C}$ is a set of constraints that specify the combinations of values which may be assigned simultaneously for the variables they involve. A constraint may involve variables from several agents. For this paper, we restrict our attention to binary constraints (i. e., constraints that involve two variables). A constraint $c_{ij} \in \mathcal{C}$ between two variables x_i and x_j is a subset of the Cartesian product of their domains ($c_{ij} \subseteq D_i \times D_j$). Each agent a_i only knows constraints involving its variables, denoted by $\mathcal{C}_i$. When there exists a constraint between two variables x_i and x_j, these variables are called ***neighbors***. The set of neighbors of a variable x_i is denoted by Γ_i. The connectivity between the variables can be represented with a constraint graph, where vertices represent the variables and edges represent the constraints [24]. A *solution* is an assignment to each variable of a value from its domain, satisfying all constraints. In order to propagate constraints locally, we assume that each agent in the system knows the initial domain of each neighbor and keeps a local copy of that domain.

For simplicity purposes, we assume each agent controls exactly one variable ($m = n$), so we use the terms agent and variable interchangeably and do not distinguish between a_i and x_i. For the rest of the paper we consider a generic agent $a_i \in \mathcal{A}$. Agent a_i stores a unique total ***order*** on agents, i.e. a vector of n agents IDs, denoted by λ_i. λ_i is called the current order of a_i. We denote by $\lambda_i[k]$ ($\forall k \in 1..n$) the ID of the agent located at position k in λ_i. Agents appearing before agent a_i in λ_i are the higher priority agents denoted by λ_i^- and conversely the lower priority agents λ_i^+ are agents appearing after a_i in λ_i. The order λ_i divides the set Γ_i of neighbors of a_i into higher priority neighbors Γ_i^-, and lower priority neighbors Γ_i^+. Figure 1 presents an example

of a DisCSP instance. This problem consists of 5 agents with the following domains $\forall i \in 1..5$, $D_i = \{1, 2, 3, 4\}$ and 6 constraints among these agents c_{12}: $x_1 \neq x_2$, c_{13}: $x_1 \neq x_3$, c_{15}: $x_1 \neq |x_5 - 2|$, c_{25}: $x_2 \neq x_5$, c_{34}: $x_3 < x_4$, and c_{45}: $x_4 \geq x_5$. Figure 1(left) shows the constraint graph representation of this instance. Agent a_1 knows three constraints $\mathcal{C}_1 = \{c_{12}, c_{13}, c_{15}\}$ and ignores the other constraints.

To solve a DisCSP, agents assign values to their variables and exchange messages to satisfy constraints with variables owned by other agents. Each agent maintains a counter, and increments it whenever it changes its value. The current value of the counter *tags* each generated assignment.

Definition 1. *Given a DisCSP defined by the network* $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$*, an* ***assignment*** *for an agent* $a_i \in \mathcal{A}$ *is a tuple* (x_i, v_i, t_i)*, where* $v_i \in D_i$ *is the value to which variable* x_i *is assigned, and* $t_i \in \mathbb{N}$ *is the timestamp tagging the assignement. Given two assignments* (x_i, v_i, t_i) *and* (x_i, v'_i, t'_i)*, if* $v_i \neq v'_i$ *then* $t_i \neq t'_i$ *by construction. If* $t_i > t'_i$*,* (x_i, v_i, t_i) *is said to be* ***more up to date*** *than* (x_i, v'_i, t'_i)*. Two sets of assignments are* ***compatible*** *if every common variable is assigned the same value in both sets.*

Definition 2. *The* ***agent-view*** *of an agent* a_i*,* AV*, stores the most up to date assignments received from other agents. It is initialized to the set of empty assignments.* AV^- *denotes the locally stored assignments of higher agents w.r.t. the current order* λ_i *stored by agent* a_i*.*

During search agents can infer inconsistent sets of assignments called no-goods. Agents use these no-goods to justify value removals.

Definition 3. *A* ***no-good*** *ruling out value* v_i *from the initial domain of a variable* x_i*,* $ngd[x_i \neq v_i]$*, is a clause of the form* $[x_j = v_j \wedge \ldots \wedge x_k = v_k] \rightarrow x_i \neq v_i$*, meaning that the assignment* $x_i = v_i$ *is inconsistent with the assignments* $x_j = v_j \wedge \ldots \wedge x_k = v_k$*. The left hand side* (lhs) *and the right hand side* (rhs) *are defined from the position of* $\rightarrow$*. We say that a no-good is* ***compatible*** *with an agent-view* AV *if its lhs is compatible with* AV*.*

Each value removal from D_i is justified by a no-good. The current domain D_i^c of a variable x_i contains all values from the initial domain D_i that are not ruled out by a no-good. The initial domain size of a_i is denoted by d_i while its current domain size is denoted by d_i^c (i. e., $d_i^c = |D_i^c|$ and $d_i = |D_i|$).

Let Λ_i be the conjunction of the left-hand sides of all no-goods ruling out values from D_i, i.e. $\Lambda_i = \bigwedge_{v_i \in \{D_i \setminus D_i^c\}} lhs(ngd[x_i \neq v_i])$. When all values of a variable x_i are ruled out by no-goods ($D_i^c = \emptyset$), these no-goods are resolved, producing a new no-good from the conjunction of their left-hand sides (Λ_i) meaning that at least one of the variables in Λ_i needs to change its value. There are clearly many different ways of representing Λ_i as a no-good. In standard backtracking search algorithms (like ABT), the variable that has the lowest priority in the current order among the conflicting variables must change its value. We will see later how our framework relaxes this restriction. Let x_t

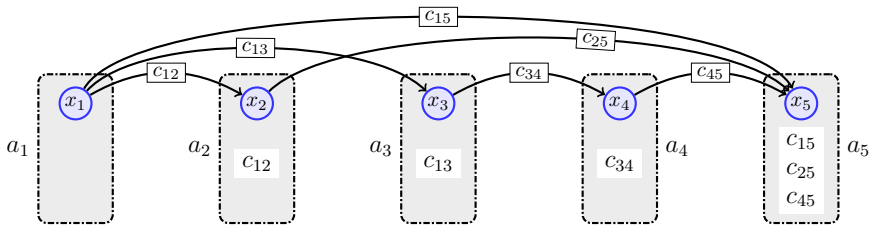


Figure 2: The directed acyclic constraint graph of the DisCSP instance in Figure 1 induced by $\lambda = [1, 2, 3, 4, 5]$.

be that variable that must change its value, i.e. to be put in the rhs of the new no-good. The variable x_t is called the *backtracking target*. The new no-good, $ngd[x_t \neq v_t]$, is obtained from Λ_i by setting $x_t \neq v_t$ in the rhs and all other assignments in Λ_i in the lhs, i.e. $ngd[x_t \neq v_t] : [\Lambda_i \setminus (x_t = v_t)] \rightarrow x_t \neq v_t$. The new generated no-good will be used as justification for removing the value v_t of the variable x_t once sent to agent a_t.

The variables in the lhs of a no-good must precede the variable on its rhs in the current order because the assignments of these variables have been used to filter the domain of the variable in its rhs. These ordering constraints induced by a no-good are called safety conditions in [25].

Definition 4. *A **safety condition** is an assertion $x_j \prec x_k$ meaning that x_j must precede x_k in the ordering. We say that a no-good is **coherent** with an order λ_i if all agents in its lhs appear before its rhs in λ_i.*

For example, the no-good $[x_j = v_j \wedge x_k = v_k] \rightarrow x_i \neq v_i$ implies that $x_j \prec x_i$ and $x_k \prec x_i$, that is x_j and x_k must precede x_i in the variable ordering (i. e., $x_j, x_k \in \lambda_i^-$).

2.2. Asynchronous Backtracking - ABT

The first complete asynchronous search algorithm for solving DisCSPs is *asynchronous backtracking* (ABT) [26, 12]. In ABT, agents act concurrently and asynchronously, and do not have to wait for decisions of others. However, to be complete, ABT requires a total priority ordering on agents (λ_i). The priority order of agents is static and uniform across the agents. The required total ordering on agents in ABT induces a directed acyclic constraint graph where constraints are represented by directed links according to the total order among agents. Hence, a directed link between each two constrained agents is established. ABT performs asynchronous search based on this structure. The agent to which the direct link arrives is the agent evaluating the constraint represented by that link. Consider the instance of Figure 1 and a lexicographic ordering on agents $\lambda = [1, 2, 3, 4, 5]$. The ordering λ induces the acyclic constraint graph shown in Figure 2. Constraints are represented by directed links from higher to lower priority neighbors. Agent a_2 evaluates constraint c_{12} because

it has a lower priority than a_1 in λ, a_3 evaluates c_{13}, a_4 evaluates c_{34}, and a_5 evaluates constraints c_{15}, c_{25}, and c_{45}.

In ABT, each agent a_i tries to find an assignment satisfying the constraints with what is currently known from higher priority neighbors Γ_i^-. When an agent a_i assigns a value to its variable, it sends out messages to lower priority neighbors Γ_i^+ informing them about its assignment. When no value is possible for a variable x_i, a_i resolves its no-goods producing a new no-good $ngd[x_t \neq v_t]$ from Λ_i. Next, agent a_i reports the resolved no-good $ngd[x_t \neq v_t]$ to agent a_t. ABT computes a solution (or detects that no solution exists) in a finite time.

Furthermore, in ABT, each agent a_i, stores in its agent-view, AV, the most up to date assignments that it believes are assigned to higher priority neighbors. To stay polynomial in space, agent a_i only keeps one no-good per removed value. When two no-goods eliminating the same value are possible, the no-good with the *highest possible lowest variable* involved is selected (*HPLV* heuristic) [27]. By doing so, when an empty domain is found, the resolved no-good contains variables as high as possible in the ordering, so that the backtrack message is sent as high as possible, thus saving unnecessary search effort [12]. In the following, v_i will represent the current value assigned to x_i and t_i the counter tagging v_i. t_i is used for the timestamp mechanism of generated assignments. ABT agents exchange the following types of messages:

- **ok?** message used to notify its recipients of a new assignment of the sender.
- **ngd** message used to report a no-good to its receiver agent, requesting the removal of its value.
- **adl** message used to request the addition of a link to the receiver.
- **stp** message used to inform all agents to stop the search meaning that the problem is unsolvable because an empty no-good has been generated.

The pseudo-code of ABT executed by every agent $a_i \in \mathcal{A}$ is presented in Figure 3. In the main procedure `ABT()`, each agent assigns a value to its variable and informs its lower priority neighbors Γ_i^+ (`assignVariable()` call, line 2). Then, it loops for processing the received messages (lines 3 to 9).

Procedure `assignVariable` is used by agent a_i to select a consistent value for x_i from its current domain D_i^c (`chooseValue` call, line 15). During this process, some values from D_i^c may appear as inconsistent. Thus, a_i removes inconsistent values from D_i^c and stores the no-goods justifying their removals (line 18). To ensure a polynomial space complexity, agents keep only one no-good per removed value. If `chooseValue` finds a consistent value, it is assigned to x_i and agent a_i notifies all its lower priority neighbors (Γ_i^+) about its new assignment through **ok?** messages (line 22) after incrementing the tag counter t_i. Otherwise, agent a_i must identify the subset of higher agents' assignments responsible for the failure, i.e. its domain wipe-out (procedure `backtrack()` call, line 23).

```
procedure ABT()
01. initialize();
02. assignVariable();
03. while ( ¬end ) do
04.     msg ← getMsg();
05.     switch ( msg.type ) do
06.         ok?   : processOk(x_s, v'_s, t'_s);
07.         ngd   : processNgd(a_s, ngd[x_i≠v'_i]);
08.         adl   : processAdl(a_s, v'_i);
09.         stp   : end ← true;

procedure initialize()
10. v_i ← nil; t_i ← 0; end ← false;
11. Γ⁻_i ← Γ_i ∩ λ⁻_i;
12. Γ⁺_i ← Γ_i ∩ λ⁺_i;

function chooseValue()
13. foreach ( v'_i ∈ D^c_i ) do
14.     if ( isConsistent(v'_i, AV⁻) ) then
15.         x_i ← v'_i;
16.         return(true);
17.     else
18.         store ngd[x_i≠v'_i] from constraints;
19. return(false);

procedure assignVariable()
20. if ( chooseValue() ) then
21.     t_i ← t_i+1;
22.     sendMsg:ok?⟨x_i, v_i, t_i⟩ to Γ⁺_i;
23. else backtrack();

procedure checkAgentView()
24. if ( ¬isConsistent(v_i, AV⁻) ) then
25.     assignVariable();

procedure updateAgentView(S)
26. foreach ( ⟨x_j, v'_j, t'_j⟩ ∈ S ) do
27.     if ( t'_j ≥ t_j ) then
28.         AV[j] ← ⟨x_j, v'_j, t'_j⟩;
29. remove incompatible no-goods;

procedure processOk(⟨x_s, v'_s, t'_s⟩)
30. updateAgentView(⟨x_s, v'_s, t'_s⟩);
31. checkAgentView();

procedure backtrack()
32. Λ_i ← resolveNogoods();
33. if ( Λ_i = ∅ ) then
34.     end ← true;
35.     sendMsg:stp⟨⟩ to {A \ a_i};
36. else
37.     Let x_t be the lowest agent in Λ_i;
38.     ngd[x_t≠v_t] ← {Λ_i \ x_t} → x_t≠v_t;
39.     sendMsg:ngd⟨ngd[x_t≠v_t]⟩ to a_t;
40.     updateAgentView(⟨x_t, nil, t_t⟩);
41.     checkAgentView();

procedure processNgd(a_s, ngd[x_i≠v'_i])
42. checkAddLink(lhs(ngd[x_i≠v'_i]));
43. if (compatible(ngd[x_i≠v'_i], AV⁻)) then
44.     store ngd[x_i≠v'_i];
45.     if ( v_i = v'_i ) then
46.         checkAgentView();
47. else if ( v_i = v'_i ) then
48.     sendMsg:ok?⟨x_i, v_i, t_i⟩ to a_s;

procedure processAdl(a_s, v'_i)
49. Γ_i ← Γ_i ∪ {a_s};
50. if ( v_i ≠ v'_i ) then
51.     sendMsg:ok?⟨x_i, v_i, t_i⟩ to a_s;

procedure checkAddLink(S)
52. updateAgentView(S);
53. Δ ← {j | ⟨x_j, v_j, t_j⟩ ∈ S ∧ j ∉ Γ_i};
54. Γ_i ← Γ_i ∪ Δ;
55. foreach ( ⟨x_j, v_j, t_j⟩ ∈ S s.t. j ∈ Δ ) do
56.     sendMsg:adl⟨x_j, v_j, t_j⟩ to a_j;
```

Figure 3: The ABT algorithm running by agent a_i.

Whenever agent a_i receives an **ok?** message from a higher agent a_s, it processes it by calling procedure `processOk`$(\langle x_s, v'_s, t'_s\rangle)$, line 6. The agent-view of a_i is updated (`updateAgentView` call, lines 28 and 30) only if the received message contains an assignment more up to date than that already stored for the sender a_s (lines 27 to 28). Next, all no-goods that become incompatible with the agent-view of a_i are removed (line 29). Then, a consistent value for a_i is searched if necessary after the change in the agent-view (`checkAgentView` call, line 31). In procedure `checkAgentView()`, agent a_i checks whether its current assignment (v_i) is consistent with assignments of higher priority neighbors, i.e. AV^-. If it is not the case, agent a_i seeks a new consistent value (`assignVariable` call,

line 25).

When every value of variable x_i is forbidden by a stored no-good, procedure `backtrack()` is called. The first step of this procedure is to determine the set Λ_i of variable assignments responsible for the failure. If Λ_i is empty, then the DisCSP has no solution. Agent a_i sends **stp** messages to all agents and terminates its execution (lines 33 to 35). Otherwise, agent a_i selects the variable x_t that has the lowest priority among the variables in the conflicting set Λ_i to be the backtracking target, generating a new no-good, $ngd[x_t \neq v_t]$ (lines 37 to 38). The generated no-good is sent in a **ngd** message to the agent a_t owning the variable x_t, line 39. Then, the assignment of x_t is deleted from the agent-view (`updateAgentView` call, line 40). Finally, a new consistent value is selected (`checkAgentView` call, line 41).

Whenever agent a_i receives a **ngd** message, procedure `processNgd` is called, line 7. In this procedure, agent a_i calls procedure `checkAddLink` (line 42) in order to update its agent-view with the newer assignments contained in the left hand side of the received no-good (`updateAgentView` call, line 52) and to request the establishment of new links with non-neighbors agents owning variables on the the left hand side of the received no-good (lines 53 to 56). Then, agent a_i checks if the received no-good is compatible with its agent-view (Definition 3). The no-good is accepted only if it is compatible with the assignments of higher priority agents, i.e. AV^- (line 43). An accepted no-good is stored in order to justify the removal of the value on its *rhs* (line 44). If the current value of a_i is the same as the *rhs* of the accepted no-good then a new consistent value for a_i is searched (`checkAgentView` call, line 46). If the no-good is not accepted, it is discarded, but if the value in its *rhs* was correct, a_i re-sends its assignment to the no-good sender (a_s) through an **ok?** message (lines 47 and 48).

When a link request is received, agent a_i calls procedure `processAdl` in order to include the sender in Γ_i (line 49). Afterwards, agent a_i sends its assignment through an **ok?** message to the sender of the request if its value is different from that included in the received **adl** message (lines 50 and 51).

It has been proven in [28, 12] that ABT is sound, complete and terminates.

2.3. Set Theory

Let S be a set and $\prec$ be a relation on S. $(S, \prec)$ is a *total ordering* if and only if for all a, b in S, either $a \prec b$ or $b \prec a$ or $a = b$.

Definition 5. *We say that $\prec$ well-orders S, or $(S, \prec)$ is a **well-ordering**, iff $(S, \prec)$ is a total ordering and every non-empty subset of S has a $\prec$-least element, i.e. for every $B \subseteq S$, $B \neq \emptyset$, there exists $a \in B$ such that for every $b \in B$ we have $a \preceq b$.*

Well-orders have the following interesting property.

Proposition 1 ([29])**.** *An ordered set is well ordered if and only if it does not include an infinite decreasing sequence.*

3. The General Framework AgileABT

In this section, we propose *agile asynchronous backtracking* (AgileABT), a general framework for reordering agents asynchronously. In AgileABT, all agents start with the same order. Then, agents are allowed to change the order asynchronously. 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 among different orders to select. We propose to establish a priority relation between orders via their *termination value*, a label attached to the orders exchanged by agents during search. Termination values are auxiliary information associated with each suggested reordering. New orders and their termination values are computed by a function, denoted by `f()` in the rest of the paper. Agents accept or reject a suggested reordering using the priority relation over termination values. Termination values can be simple scalar values, or more complex structures based on some features of the agents. As in AgileABT every agent can change the order without any global control, whenever changing the order, agent a_i informs other agents of the new order by sending them its new order λ_i and its associated termination value, denoted by τ_i. When an agent compares two pairs order/termination value, the *strongest* pair is chosen.

Definition 6. *Let λ_i and λ_j be two total agent orderings and τ_i and τ_j their associated termination values. The pair (λ_i, τ_i) is **stronger** than the pair (λ_j, τ_j) if and only if the termination value τ_i is smaller than τ_j w.r.t. the priority relation $\prec_\tau$ over the range of termination values, or $\tau_i = \tau_j$ and the vector of agents IDs in λ_i is smaller than the vector of agents IDs in λ_j w.r.t. the lexicographic order $<_{lex}$.*

Consider for instance the three orders on five agents $\lambda_i = [1, 2, 3, 4, 5]$, $\lambda'_i = [1, 2, 5, 4, 3]$ and $\lambda''_i = [1, 2, 4, 5, 3]$. If the termination value τ'_i associated with λ'_i is smaller than the termination value τ_i associated with λ_i, the pair (λ'_i, τ'_i) is stronger than the pair (λ_i, τ_i). If the termination value τ''_i associated with λ''_i is equal to the termination value associated with λ'_i, the pair (λ''_i, τ''_i) is stronger than the pair (λ'_i, τ'_i) because the vector $[1, 2, 4, 5, 3]$ of IDs in λ''_i is lexicographically smaller than the vector $[1, 2, 5, 4, 3]$ of IDs in λ'_i.

3.1. The algorithm AgileABT

In AgileABT, agents exchange the following types of messages:

- **ok?** message used to notify its recipients of a new assignment of the sender.
- **ngd** message used to report a no-good to its receiver agent, requesting the removal of its value.
- **adl** message used to request the addition of a link to the receiver.
- **order** message is sent to propose a new order. This message includes the proposed order together with its associated termination value $\langle \lambda, \tau \rangle$.

- **stp** message used to inform all agents to stop the search meaning that the problem is unsolvable because an empty no-good has been generated.

The pseudo-code of AgileABT executed by every agent $a_i \in \mathcal{A}$ is presented in Figure 4. In the following, v_i will represent the current value assigned to x_i and t_i the counter tagging v_i. t_i is used for the timestamp mechanism of generated assignments. In the main procedure `AgileABT`, after initialization of the data structures (line 1) each agent assigns a value to its variable and informs its lower priority neighbors Γ_i^+ through **ok?** messages (`assignVariable` call, line 2). Then, it loops for processing the received messages (lines 3 to 10).

Procedures `initialize`, `chooseValue`, `assignVariable`, `checkAgentView`, `updateAgentView`, `processOk`, `processAdl`, and `checkAddLink` are exactly the same as in ABT. We reproduce them in Figure 4 but we do not repeat the description.

The procedure `backtrack()` is called when a dead end occurs (i.e., when every value of variable x_i is forbidden by a stored no-good). The first step of this procedure is to determine the set Λ_i of variable assignments responsible for the failure (`resolveNogoods()` call, line 37). If Λ_i is empty, then the DisCSP has no solution. Agent a_i sends **stp** messages to all agents and terminates its execution (lines 38 to 40). Otherwise, agent a_i is allowed to propose a new order (`proposeOrder()` call, line 43). After the reordering, agent a_i selects the variable x_t that has the lowest priority among the variables in the conflicting set Λ_i to be the backtracking target, generating a new no-good, $ngd[x_t \neq v_t]$ (lines 44 to 45). The generated no-good is sent in a **ngd** message to agent a_t owning the variable x_t, line 46. It is important to note that as a consequence of changing ordering, x_t was not necessarily the lowest variable in $ngd[x_t \neq v_t]$ on the agent ordering when calling procedure `backtrack()`. The assignment of x_t is deleted from the agent-view (`updateAgentView` call, line 47). Finally, agent a_i checks if its current assignment is consistent with assignments of higher agents AV^-. If it is the case, agent a_i has to send its assignment through **ok?** messages to its lower neighbors who did not receive it beforehand because they had a higher priority in previous agent ordering, line 49. If v_i is inconsistent with AV^-, agent a_i tries to select a new consistent value (`assignVariable()` call, line 50).

The procedure `proposeOrder()` calls function `f()` to generate a new order λ' and a new termination value τ', line 31. If the termination value τ' associated with the newly generated order λ' is smaller (w.r.t. $\prec_\tau$) than the termination value τ_i associated to the current order λ_i, agent a_i sends $\langle \lambda', \tau' \rangle$ to all other agents through **order** messages, line 33, and calls procedure `changeOrder`, line 34, to update its current order, termination value, according to the newly generated ones. Otherwise, a_i keeps its current order and termination value unchanged. In procedure `changeOrder()`, agent a_i replaces its current order λ_i and its associated termination value τ_i by the new ones (line 29). Then, a_i removes all no-goods that become incoherent with the new order (line 30).

Whenever agent a_i receives an **order** message (procedure `processOrder`, line 9), it checks if the pair (λ_s, τ_s) included in the received message is stronger than its current pair (λ_i, τ_i) (line 51). If it is the case, a_i calls procedure

```
procedure AgileABT()
01. initialize();
02. assignVariable();
03. while ( ¬end ) do
04.     msg ← getMsg();
05.     switch ( msg.type ) do
06.         ok?   : processOk(x_s, v_s, t_s);
07.         ngd   : processNgd(a_s, ngd[x_i≠v'_i]);
08.         adl   : processAdl(a_s, v'_i);
09.         order : processOrder(λ_s, τ_s);
10.         stp   : end ← true;

procedure initialize()
11. v_i ← nil; t_i ← 0; end ← false;
12. Γ_i^- ← Γ_i ∩ λ_i^-;
13. Γ_i^+ ← Γ_i ∩ λ_i^+;

function chooseValue()
14. foreach ( v'_i ∈ D_i^c ) do
15.     if ( isConsistent(v'_i, AV^-) ) then
16.         x_i ← v'_i;
17.         return(true);
18.     else store ngd[x_i≠v'_i] from constraints;
19. return(false);

procedure assignVariable()
20. if ( chooseValue() ) then
21.     t_i ← t_i+1;
22.     sendMsg:ok?⟨x_i, v_i, t_i⟩ to Γ_i^+;
23. else backtrack();

procedure checkAgentView()
24. if ( ¬isConsistent(v_i, AV^-) ) then
25.     assignVariable();

procedure updateAgentView(S)
26. foreach ( ⟨x_j, v'_j, t'_j⟩ ∈ S ) do
27.     if ( t'_j ≥ t_j ) then AV[j] ← ⟨x_j, v'_j, t'_j⟩;
28. remove incompatible no-goods;

procedure changeOrder(λ', τ')
29. ⟨λ_i, τ_i⟩ ← ⟨λ', τ'⟩;
30. remove incoherent no-goods;

procedure proposeOrder()
31. ⟨λ', τ'⟩ ← f();
32. if ( τ' ≺_τ τ_i ) then
33.     sendMsg:order⟨λ', τ'⟩ to {A \ a_i};
34.     changeOrder(λ', τ');

procedure processOk(⟨x_s, v'_s, t'_s⟩)
35. updateAgentView(⟨x_s, v'_s, t'_s⟩);
36. checkAgentView();

procedure backtrack()
37. Λ_i ← resolveNogoods();
38. if ( Λ_i = ∅ ) then
39.     end ← true;
40.     sendMsg:stp⟨⟩ to {A \ a_i};
41. else
42.     LN ← Γ_i^+;
43.     proposeOrder() ;
44.     Let x_t be the lowest agent in Λ_i;
45.     ngd[x_t≠v_t] ← ({Λ_i \ x_t} → x_t≠v_t);
46.     sendMsg:ngd⟨ngd[x_t≠v_t]⟩ to a_t;
47.     updateAgentView(⟨x_t, nil, t_t⟩);
48.     if ( isConsistent(v_i, AV^-) ) then
49.         sendMsg:ok?⟨x_i, v_i, t_i⟩ to {Γ_i^+ \ LN};
50.     else assignVariable();

procedure processOrder(λ_s, τ_s)
51. if ( (λ_s, τ_s) is stronger than (λ_i, τ_i) ) then
52.     LN ← Γ_i^+;
53.     changeOrder(λ_s, τ_s);
54.     if ( isConsistent(v_i, AV^-) ) then
55.         sendMsg:ok?⟨x_i, v_i, t_i⟩ to {Γ_i^+ \ LN};
56.     else assignVariable();

procedure processNgd(a_s, ngd[x_i≠v'_i])
57. checkAddLink(lhs(ngd[x_i≠v'_i]));
58. if (compatible(ngd[x_i≠v'_i], AV^-) ∧
        coherent(ngd[x_i≠v_i], λ_i)) then
59.     store ngd[x_i≠v'_i];
60.     if ( v_i = v'_i ) then checkAgentView();
61. else if ( v_i = v'_i ) then
62.     sendMsg:ok?⟨x_i, v_i, t_i⟩ to a_s;

procedure processAdl(a_s, v'_i)
63. Γ_i ← Γ_i ∪ {a_s};
64. if ( v_i ≠ v'_i ) then
65.     sendMsg:ok?⟨x_i, v_i, t_i⟩ to a_s;

procedure checkAddLink(S)
66. updateAgentView(S);
67. Δ ← {j | ⟨x_j, v_j, t_j⟩ ∈ S ∧ j ∉ Γ_i};
68. Γ_i ← Γ_i ∪ Δ;
69. foreach ( ⟨x_j, v_j, t_j⟩ ∈ S s.t. j ∈ Δ ) do
70.     sendMsg:adl⟨x_j, v_j, t_j⟩ to a_j;
```

Figure 4: The AgileABT algorithm run by agent a_i.

`changeOrder` to change its current order and its associated termination value to the newly received ones $\langle \lambda_s, \tau_s \rangle$ (line 53). Next, agent a_i checks if v_i is consistent with assignments of higher agents AV^-. If it is the case, agent a_i has to send x_i's assignment through **ok?** messages to its lower neighbors who did not receive it beforehand because they had a higher priority in previous agent ordering, line 55. If v_i is inconsistent with AV^-, agent a_i tries to select a new consistent value (`assignVariable()` call, line 56).

The procedure `processNgd()` is called whenever agent a_i receives a **ngd** message, line 7. In this procedure, agent a_i calls procedure `checkAddLink` (line 57) in order to update its agent-view with the newer assignments contained in the left hand side of the received no-good (`updateAgentView` call, line 66) and to request the establishment of new links with non-neighbors agents owning variables on the left hand side of the received no-good (lines 67 to 70). Then, the compatibility (Definition 3) and coherence (Definition 4) of the received no-good are checked (line 58) and the no-good is accepted only if it is compatible with the assignments of higher priority agents, i.e. AV^- and coherent with the current order λ_i of agent a_i. An accepted no-good is stored in order to justify the removal of the value on its rhs (line 59). If the current value of a_i is the same as the rhs of the accepted no-good then a new consistent value for a_i is searched (`checkAgentView` call, line 60). If the no-good is not accepted, it is discarded, but if the value in its rhs was correct, a_i re-sends its assignment to the no-good sender (a_s) through an **ok?** message (lines 61 and 62).

There are several ways to instantiate the general framework AgileABT. The instantiation is made by defining the termination values and the priority relation over those termination values and by specifying the function `f()` that computes new orders associated with those termination values.

3.2. Correctness and complexity

In this section we prove that AgileABT is sound, complete and terminates provided that $(\mathcal{T}, \prec_\tau)$ is a well-ordering, where $\mathcal{T}$ denotes the range of termination values computed by function `f()` and $\prec_\tau$ is the priority relation on $\mathcal{T}$. In addition, if the function `f()` has polynomial space complexity, then AgileABT has polynomial space complexity.

AgileABT always stops its execution in one of the two cases: when an empty no-good has been generated, meaning that there is no solution, or when the network reaches a quiescence state reporting a solution. To prove that AgileABT is *sound*, one needs to prove that the reported solution is a correct one and to prove that it is *complete* one needs to prove that it cannot infer an empty no-good if a solution exists.

Proposition 2. *The AgileABT algorithm is sound.*

Proof. Let us assume that the state of quiescence is reached. The order (say λ^*) known by all agents is the same because when an agent proposes a new order, it sends it to all other agents. Obviously, (λ^*, τ_i^*) is the strongest pair 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 exists two agents a_i and a_k that do not satisfy the constraint between them (i. e., c_{ik}). Let a_i be the agent which has the highest priority between the two agents according to λ^* (a_k is the agent evaluating c_{ik}). Let v_i be the current value of a_i when the state of quiescence is reached (i. e., $\langle x_i, v_i, t_i \rangle$ is the most up to date assignment of a_i). Let msg be the last **ok?** message sent by a_i to a_k before the state of quiescence is reached. Clearly, msg contains v_i, otherwise, a_i would have sent another **ok?** message when it chose v_i, i.e. $msg = \underset{i \to k}{\textbf{ok?}}:\langle x_i, v_i, t_i \rangle$. If v_i was assigned after a_i changed its current order to λ^*, agent a_i sent msg to all its lower priority neighbors according to λ^* (including a_k, procedure `assignVariable()`). If v_i was assigned before a_i changed its previous order λ_i to λ^*, agent a_k received msg if $a_k \in \lambda_i^+$ when v_i was assigned to a_i, otherwise ($a_k \in \lambda_i^-$), a_k receives a copy of msg when a_i changed λ_i to λ^* (line 49 or line 55). The only case where a_k can forget v_i after receiving it is the case where a_k derives a no-good proving that v_i is not feasible. In this case, a_k should send a no-good message to a_i. If the no-good message is accepted by a_i, a_i must send an **ok?** message to its lower neighbors (and therefore msg is not the last one). Similarly, if the no-good message is discarded, a_i has to re-send an **ok?** message to a_k, lines 61 and 62 (and therefore msg is not the last one). So the state of quiescence implies that a_k knows both λ^* and v_i. Upon receiving an **ok?** message, agents in AgileABT call procedure `checkAgentView()` to ensure their current value is consistent with the assignments of higher neighbors. Therefore, the state of quiescence implies that the current value of a_k is consistent with value v_i. □

Proposition 3. *If the function* `f()` *has a polynomial space complexity, AgileABT has also a polynomial space complexity.*

Proof. In addition to the termination values, each agent in AgileABT stores one no-good per removed value and one current order that is bounded by n, the total number of agents. Thus, the space complexity of those data structures is in $\mathcal{O}(nd+n) = \mathcal{O}(nd)$ on each agent. Hence, the space complexity of AgileABT is polynomial. □

We now show that if the function `f()` always returns the same order, then AgileABT reduces to ABT.

Lemma 1. *If all agents hold the same order λ at the start of the solving process and the function* `f()` *always returns λ, then AgileABT behaves like ABT called with the order λ on all the agents.*

Proof. Agents in AgileABT exchange **ok?**, **ngd**, **adl**, **order**, and **stp** messages. When an AgileABT agent a_i assigns a value to its variable it sends **ok?** messages to its lower neighbors (Line 22) w.r.t. λ, exactly like ABT. When the AgileABT agent a_i detects a dead-end, it is allowed to propose a new order through **order** messages (Line 43). But this step has no effect because by assumption the function `f()` used to compute new orders always returns λ. Hence, when a dead-end

is detected, the agent a_i resolves its no-goods and reports the resolved no-good $ngd[x_t{\neq}v_t]$ to the lowest agent a_t w.r.t. λ through a **ngd** message (Line 46), exactly like ABT. Since λ is static, $lhs(ngd[x_t{\neq}v_t])$ exclusively contains assignments of agents higher than a_t in λ. Thus, the add-link requests in line 70 can only be sent to higher agents, again exactly as in ABT. Finally, an **stp** message is sent by an AgileABT agent a_i when an empty no-good is detected (Line 40), which is the same condition as in ABT. Therefore, AgileABT behaves exactly like ABT if all agents hold the same order λ at the start of the solving process and the function `f()` always returns λ. □

Proposition 4. *AgileABT terminates if $(\mathcal{T}, \prec_\tau)$ is a well-ordering.*

Proof. By the assumption that messages are delivered in finite time, a pair (λ, τ) sent by an agent is known by all other agents after a finite amount of time. After this time, an agent can no longer generate and send a new order that has a termination value equal to τ, even if the new order is lexicographically smaller than the one to which τ was attached (Figure 4, line 32). Hence, if no agent improves the termination value τ, all agents will know the same pair (λ', τ) after a finite amount time, where λ' is the lexicographically smallest order generated with termination value τ during the time (λ, τ) was traveling through the system. Furthermore, by Proposition 1, the termination values cannot decrease indefinitely w.r.t. $\prec_\tau$ if $(\mathcal{T}, \prec_\tau)$ is a well-ordering. Thus, AgileABT cannot change the pairs (λ, τ) indefinitely. As a result, after a finite amount of time, either a quiescent state has been reached, or all agents own the strongest pair computed in the system and follow the same static order. In that last case, AgileABT then behaves exactly like ABT (Lemma 1), which terminates. □

Proposition 5. *The AgileABT algorithm is complete.*

Proof. Suppose a solution to the DisCSP exists. By Proposition 4, AgileABT must terminate. As noted previously, AgileABT can only terminate if it generates an empty no-good, or reaches a quiescent state. All no-goods used as justification of removing inconsistent values are induced by the constraints of the problem (Line 18, Figure 4). Thus, these no-goods are redundant regarding the problem to solve. All additional no-goods produced when dead-ends occur are generated by logical inference from existing ones (Line 37, Figure 4). As a result, an empty no-good cannot be inferred if a solution exists. Therefore AgileABT must reach a quiescent state, and by Proposition 2, this must represent a solution. □

3.3. Geometric interpretation of the termination of AgileABT

In this Section we present a geometric interpretation of the termination proof of Proposition 4. Given the set $\mathcal{T}$ of termination values that can be computed by function `f()`, Proposition 4 uses the assumption that $(\mathcal{T}, \prec_\tau)$ is a well ordering, that is, $\mathcal{T} = (\tau^*, \dots, \tau^t, \dots, \tau^0, \dots)$ has a least element $\tau^* = \min_{\prec_\tau}(\mathcal{T})$. In AgileABT, the initial termination value is the same for all agents and is denoted

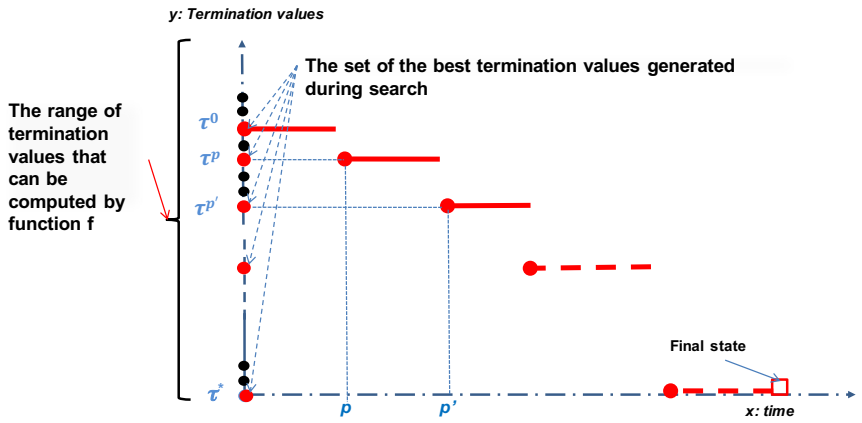


Figure 5: The variation of the best termination value of the system over time.

by τ^0. A final state is a state where a solution is found or the inconsistency is proved.

The function $\tau(x)$ returning the best termination value computed by all agents until time x is a function of time that has the shape of a decreasing sequence of horizontal segments (Figure 5). Given a termination value τ^p computed at time p, the segment where $\tau(x) = \tau^p$ is an interval of time that starts at the time p where the best termination value in the system was improved to τ^p and ends at the time p' where an agent has succeeded to propose a new termination value $\tau^{p'}$ smaller than τ^p, or the final state was reached. The key points of the proof are that the length of each segment is finite as well as the number of segments.

Regarding the length of each segment, assume a_i has proposed a new order λ^p that improves the current best termination value of the system to value τ^p. Once another agent a_j knows τ^p, it can no longer propose an order with the same termination value even if the proposed order is lexicographically smaller than λ^p (Figure 4, line 32). Hence, the time during which an agent a_j can change the order without improving τ^p is bounded by the finite time it takes to receive the **order** message sent by a_i and containing (λ^p, τ^p). Thus, after a finite time, τ^p is known by all agents and no order lexicographically smaller than λ^p can be generated. Again because messages are delivered in finite time, the lexicographically smallest order that could be generated with termination value equal to τ^p is known by all agents after a finite time. At this point, the order is the same for all agents and AgileABT starts behaving like ABT (Lemma 1). ABT is correct and terminates. As a result, either a new smaller termination value is discovered or a final state is reached after a finite time. Therefore, the length of each segment is finite.

As for the number of segments, $\tau(x)$ can only decrease or remain unchanged because agents reject termination values that are greater than their current termination value. Hence, as long as agents can improve the termination value,

$\tau(x)$ will continue to fall down along the y-axis. The fact that $(\mathcal{T}, \prec_\tau)$ is a well ordering (assumption in Proposition 4) implies that the number of times $\tau(x)$ can fall down along the y-axis is finite (Proposition 1). As every segment corresponds to a best termination value over an interval of time (Figure 5), we conclude that the number of segments that make the shape of $\tau(x)$ is finite.

To summarize, AgileABT has two main features that guarantee termination. When the best termination value stops changing, AgileABT ends up behaving like ABT and this ensures that the segments have a finite size. And if the best termination value continues to be improved, we are sure that it cannot indefinitely decrease because $(\mathcal{T}, \prec_\tau)$ is a well ordering. So the number of segments is bounded.

3.4. Example of Running AgileABT with a trivial function f

We now give a simple example of running AgileABT with a trivial function `f()`. Given that $\mathcal{T}$ is the range of termination values computed by `f()`, we have seen in Section 3.2 that the only restriction on `f()` to ensure that AgileABT terminates is that $(\mathcal{T}, \prec_\tau)$ is a well-ordering (Proposition 4). We thus choose a trivial function `f()` that simply returns a random reordering of agents associated with a termination value that is a random integer in the range $\mathcal{T} = \{0, \ldots, 100\}$ (Figure 6b).[1] The relation $\prec_\tau$ is the standard $<$ ordering of the integers. Clearly $(\mathcal{T}, \prec_\tau)$ is a well ordering.

We give a sketch of a possible execution of AgileABT on a small problem with four variables and five constraints (Figure 6a). The initial order is $\lambda^0 = [1, 2, 3, 4]$ and the initial termination value τ^0 is 100.

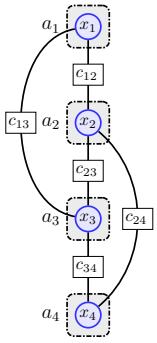
- t_0: All agents assign value 1 to their variables and send **ok?** messages to their lower priority neighbors. Observe that apart from the message sent by agent a_1 to agent a_2, no other message will cause a conflict with the current value of the recipient.

- t_1: Agent a_2 receives the assignment of a_1 ($x_1 = 1$) and removes value 1 from its domain because of the constraint $(x_1 + x_2)\ div\ 3 = 0$. This removal is justified by the no-good ngd_1 (Figure 6c). a_2 replaces its current assignment by $x_2 = 2$ and sends **ok?** messages to its lower priority neighbors (i. e., a_3 and a_4 (Figure 6a).

- t_2: Agent a_3 receives the new assignment of a_2 ($x_2 = 2$) and removes value 1 from its domain because of the constraint $x_2 \leq x_3$. This removal is justified by the no-good ngd_2 (Figure 6c). Next, a_3 replaces its current assignment by $x_3 = 2$ and sends an **ok?** message to a_4.

- t_3: Agent a_4 has received the **ok?** messages sent by a_2 and a_3 ($(x_2 = 2)$ and $(x_3 = 2)$). All of the values in its domain are now eliminated because

[1] In practice it is of course desirable to choose a function `f()` such that the smaller the termination value, the better the associated ordering.



$\mathcal{A} = \{a_1, \ldots, a_4\}$
$\mathcal{X} = \{x_1, \ldots, x_4\}$
$\mathcal{D} = \{D_1, \ldots, D_4\}$,
where $D_i = \{1, 2, 3\}$
$\mathcal{C} = \{c_{12}, c_{13}, c_{23}, c_{24}, c_{34}\}$

c_{12}: $(x_1 + x_2)$ *div* $3 = 0$
c_{13}: $x_1 + x_3 \neq 5$
c_{23}: $x_2 \leq x_3$
c_{24}: $x_2 + x_4 \neq 3$
c_{34}: $x_3 \cdot x_4 \leq 3$

(a) The constraint graph of a DisCSP instance of 4 agents/variables and 4 constraints.

function `f()`
01. $\lambda \leftarrow$ random permutation of $[1, \ldots, n]$;
02. $\tau \leftarrow$ random integer in $[0, \ldots, 100]$;
03. `return`$(\langle \lambda, \tau \rangle)$;

(b) A trivial function `f()` used by agents to propose new orders.

Step	Orders / Nogoods	Decisions
t_0	$\lambda^0 = [1,2,3,4], \tau^0 = 100$	$x_i = 1, \forall i$
t_1	ngd_1: $x_1 = 1 \rightarrow x_2 \neq 1$	$x_2 = 2$
t_2	ngd_2: $x_2 = 2 \rightarrow x_3 \neq 1$	$x_3 = 2$
t_3	ngd_3: $x_2 = 2 \rightarrow x_4 \neq 1$ ngd_4: $x_3 = 2 \rightarrow x_4 \neq 2$ ngd_5: $x_3 = 2 \rightarrow x_4 \neq 3$ ngd_6: $\neg[x_2 = 2 \wedge x_3 = 2]$	$x_4 = 1$
t_4	$\lambda^1 = [3,2,4,1], \tau^1 = 50$	

(c) Agent a_4 facing a dead-end.

Step	Orders / Nogoods	Decisions
t_5	**ok?** messages sent to new Γ_i^+	
t_6	ngd_6: $x_3 = 2 \rightarrow x_2 \neq 2$	$x_2 = 3$
t_7	ngd_7: $x_2 = 3 \rightarrow x_1 \neq 1$ ngd_8: $x_2 = 3 \rightarrow x_1 \neq 2$ ngd_9: $x_3 = 2 \rightarrow x_1 \neq 3$ ngd_{10}: $\neg[x_2 = 3 \wedge x_3 = 2]$	$x_1 = 1$
t_8	ngd_{10}: $x_3 = 2 \rightarrow x_2 \neq 3$	$x_2 = 1$
t_9	ngd_{11}: $x_2 = 1 \rightarrow x_1 \neq 1$	$x_1 = 2$

(d) Agent a_1 facing a dead-end.

Figure 6: An example of AgileABT running with a trivial function `f()`

of the constraints $x_2 + x_4 \neq 3$ and $x_3 \cdot x_4 \leq 3$. These deletions are justified by the no-goods ngd_3, ngd_4 and ngd_5 (Figure 6c). Then, agent a_4 proceeds to the resolution of its no-goods and derives the new no-good $ngd_6 : \neg[(x_2 = 2) \wedge (x_3 = 2)]$ (Figure 6c). Agent a_4 calls function `f()` to see whether it can propose a new order. Suppose `f()` returns $\langle \lambda^1 = [3,2,4,1], \tau^1 = 50 \rangle$. Because $\tau^1 = 50$ is smaller than $\tau^0 = 100$, the new order λ^1 is accepted by a_4 and sent to all other agents. The conclusion of the no-good ngd_6 is selected such that the ordering constraints induced by ngd_6 are coherent with the new order λ^1. That is, x_2 is moved down to the conclusion of ngd_6 ($ngd_6 : x_3 = 2 \rightarrow x_2 \neq 2$) and a **ngd** message is sent to the owner of x_2 (i. e., a_2). Afterwards, the no-good ngd_3 is removed from the set of no-goods stored by agent a_4 because it becomes incompatible. After that, agent a_4 assigns its variable to value 1, the unique value left in its domain. Agent a_4 does not have any lower priority

neighbor to inform, so it does not send any **ok?** message.

- t_4: Agents a_1, a_2 and a_3 receive the new order sent by a_4. All of them accept the new order because it is associated with a termination value smaller than τ^0. Agents a_2 and a_3 remove respectively the no-goods ngd_1 and ngd_2 because they become incoherent with the new order.

- t_5: After accepting the new order, agents a_1, a_2 and a_3 check their current values. Because their current values are still consistent, each of them has to send **ok?** messages to its new lower priority neighbors that had higher priority than its own before the order changed. That is, agent a_2 sends an **ok?** $(x_2 = 2)$ message to a_1 and agent a_3 sends an **ok?** $(x_3 = 2)$ message to a_2 and a_1. These messages do not cause any conflict with the current assignments of their recipients a_2 and a_1. Agent a_1 will not send any **ok?** message because it is now the last agent in the order.

- t_6: Agent a_2 receives the no-good sent by a_4 and replaces its assignment by $x_2 = 3$. Next, it sends **ok?** $(x_2 = 3)$ messages to a_1 and a_4.

- t_7: The **ok?** message sent by a_2 is received by a_4 and a_1. This message causes agent a_1 to face a dead end because of the constraints (x_1+x_2) div $3 = 0$ and $x_1+x_3 \neq 5$. Therefore, a_1 resolves its no-goods and derives the new no-good $ngd_{10} : \neg[(x_2 = 3) \wedge (x_3 = 2)]$ (Figure 6d). Then, agent a_1 calls function `f()` in the hope of proposing a new order. Suppose `f()` returns $\langle \lambda^2 = [4, 1, 2, 3], \tau^2 = 90 \rangle$. The new order λ^2 is not accepted by a_1 and is not sent to the other agents because $\tau^2 = 90 \geq \tau^1 = 50$. The no-good ngd_{10} is sent to a_2 and the no-goods ngd_7 and ngd_8 are removed by a_1 because they become incompatible. Agent a_1 reassigns its variable to value 1.

- t_8: Agent a_2 receives the no-good sent by a_1, instantiates x_2 to 1 and informs its lower priority neighbors.

- t_9: Agents a_1 and a_4 receive the **ok?** $(x_2 = 1)$ message sent by a_2. a_1 removes its value 1, stores the no-good $ngd_{11} : x_2 = 1 \rightarrow x_1 \neq 1$ and assigns its variable to value 2. The current value 1 of agent a_4 is not in conflict with the new assignment of a_2. A solution has been found.

4. Instantiation of the General Framework

In this section, we propose AgileABT($[\alpha], \prec_\alpha$), an instance of AgileABT in which the role of the termination values is not only to establish priority between the different orders proposed by agents but also to simulate a dynamic variable ordering (DVO) heuristic, since DVOs significantly speed up search in CSPs. To be able to simulate a given DVO, we first need to define a measure α that together with a preference relation $\prec_\alpha$ over the range of measure α (i. e., $\langle \alpha, \prec_\alpha \rangle$) capture the DVO heuristic. In other words, α needs to be smaller w.r.t. $\prec_\alpha$ when a variable is better for the DVO. For example to represent the

min-domain heuristic, α is defined by the domain size and $\prec_\alpha$ is defined by the standard ordering $<$ on numbers.

Furthermore, to be able to properly reorder all agents in AgileABT($[\alpha], \prec_\alpha$), a termination value needs to express information about every agent. Thus, in AgileABT($[\alpha], \prec_\alpha$), termination values are vectors of size n, where each element in the vector represents a measure α used to implement a DVO of the agent in that position in the order. More formally, given an ordering λ, its associated termination value τ is built in such a way that the k^{th} element $\tau[k]$ of τ depends on the agent at position k in λ. Specifically, we have $\tau[k] = \alpha(\lambda[k])$, where α is a generic measure that uses information about agents in the system.

In the general framework of AgileABT, we proposed to establish priority between orders using the priority relation $\prec_\tau$ over termination values. Thus, in AgileABT($[\alpha], \prec_\alpha$) we first need to define the priority relation over termination values in the form of a vector of measures α. AgileABT($[\alpha], \prec_\alpha$) uses termination values to implement the DVO and, as a side effect, to provide more flexibility in the choice of the backtracking target. We will show later how to simulate different DVO heuristics by specifying the generic measure α together with the preference relation $\prec_\alpha$. We then need to specify the function `f()` that computes new orders associated with their termination values in form of vector of measures α. We will do it in such a way that the smaller the value returned by α (w.r.t. , $\prec_\alpha$), the more preferred the corresponding variable for the DVO heuristic represented by α. In the following we discuss theses points together with necessary materials before formally presenting AgileABT($[\alpha], \prec_\alpha$) algorithm.

4.1. Priority between orders

In AgileABT, agents accept or reject the suggested reordering using a priority relation over termination values. Thus, we need to specify a priority relation on termination values defined by vector of measures α. Let α and $\prec_\alpha$ be a measure on agents and a total preference order that together capture the DVO to simulate and let $\mathcal{R}^\alpha$ be the range of measure α. We propose that the priority between the different orders is based on the lexicographic comparison of termination values (vector of measure α) using $\prec_\alpha$ as preference order on the elements of the vector:

Definition 7 ($\prec_\alpha^{lex}$)**.** *Let λ_i and λ_j be two total agent orderings and τ_i and τ_j their associated termination values. The termination value τ_i is smaller than τ_j ($\tau_i \prec_\alpha^{lex} \tau_j$) if and only if τ_i is lexicographically smaller than τ_j w.r.t. $\prec_\alpha$. In other words, $\tau_i \prec_\alpha^{lex} \tau_j$ if and only if $\exists\ k \in 1..n$ such that $\forall p \in 1..k-1,\ \tau_i[p] = \tau_j[p]$ and $\tau_i[k] \prec_\alpha \tau_j[k]$.*

Now we shall give a small example to illustrate how $\prec_\alpha^{lex}$ is used to compare termination values and orders. Consider for instance three orders on five agents $\lambda = [1, 2, 3, 4, 5]$, $\lambda' = [1, 2, 5, 4, 3]$, and $\lambda'' = [1, 2, 4, 5, 3]$, associated respectively with termination values $\tau = [2, 2, 3, 4, 2]$, $\tau' = [1, 2, 2, 2, 2]$, and $\tau'' = [1, 2, 2, 2, 2]$. The pair (λ', τ') is stronger than the pair (λ, τ) because the termination value

associated with λ' is smaller than the termination value associated with λ (i. e., $\tau' \prec_\alpha^{lex} \tau$). However, the pair (λ'', τ'') is stronger than the pair (λ', τ') because termination values associated with λ'' and λ' are equal, but we broke the tie by comparing lexicographically the vector of IDs as stated in Definition 6 (i. e., the vector $[1, 2, 4, 5, 3]$ of IDs in λ'' is lexicographically smaller than ($<_{lex}$) the vector $[1, 2, 5, 4, 3]$ of IDs in λ').

The termination of the general framework of AgileABT is based on Proposition 4 that requires that the priority relation on termination values $\prec_\alpha^{lex}$ well-orders the range $\mathcal{T}$ of terminations values. In the following, we define a simple condition on the total preference order $\prec_\alpha$ which guarantees that $(\mathcal{T}, \prec_\alpha^{lex})$ is a well-ordering and therefore AgileABT$([\alpha], \prec_\alpha)$ terminates.

Proposition 6. *If $\prec_\alpha$ well-orders the range $\mathcal{R}^\alpha$ of measure α then the lexicographic comparison $\prec_\alpha^{lex}$ well-orders the range $\mathcal{T}$ of termination values.*

Proof. We proceed by contradiction. Suppose $(\mathcal{T}, \prec_\alpha^{lex})$ is not a well-ordering, that is, we can obtain an infinite decreasing sequence of termination values using a lexicographic comparison w.r.t. $\prec_\alpha$. We thus must have an infinite decreasing sequence of $\tau[k] = \alpha(\lambda[k])$ measures, for some $k \in 1..n$. But, following Proposition 1, it would mean that $\prec_\alpha$ does not well-order $\mathcal{R}^\alpha$, which contradicts the assumption that $\prec_\alpha$ well-orders $\mathcal{R}^\alpha$. Therefore, the lexicographic comparison w.r.t. $\prec_\alpha$ well-orders $\mathcal{T}$ if the priority ordering $\prec_\alpha$ well-orders $\mathcal{R}^\alpha$. □

Corollary 1. *AgileABT$([\alpha], \prec_\alpha)$ terminates if $\prec_\alpha$ well-orders $\mathcal{R}^\alpha$.*

Proof. Direct from Proposition 4 and Proposition 6. □

4.2. Implementing DVO heuristics

The termination values defined by vector of measures α allow us to mimic a wide variety of DVOs. To mimic them in AgileABT$([\alpha], \prec_\alpha)$, it is sufficient to define the measure α in such a way that the smaller (w.r.t. $\prec_\alpha$) $\alpha(k)$, the better variable x_k for the heuristic. Once we have decided which DVO we want AgileABT$([\alpha], \prec_\alpha)$ to mimic, the only thing to do is to design the right measure α and the right total preference relation $\prec_\alpha$ over the range of α. The only condition required for AgileABT$([\alpha], \prec_\alpha)$ to terminate is that the total preference relation $\prec_\alpha$ well-orders $\mathcal{R}^\alpha$ (see Corollary 1). In the following, we show how AgileABT$([\alpha], \prec_\alpha)$ is able to capture a number of different DVO heuristics that are known to be effective in reducing search in centralized CSP.

The DVO heuristics we consider in this work can be divided in three categories: (i) min-domain size [13] (i. e., *dom*), (ii) neighborhood based DVOs [14] (i. e., *dom/deg*, *dom/fdeg*, and *dom/pdeg*), and (iii) conflict-directed DVO (i. e., *dom/wdeg*) [30]. Domain size is ubiquitous in all those DVOs. As AgileABT$([\alpha], \prec_\alpha)$ is used in a distributed environment, agents must be able to recognize the domain size of other agents. To this end, agents exchange *explanations* of the domain sizes.

Definition 8. *An **explanation** e_i is an expression $e_i\colon\ \Lambda_i \rightarrow d_i^c$, where Λ_i is the conjunction of the left hand sides of all no-goods stored by a_i as justifications of value removals for x_i, and d_i^c is the size of the current domain of x_i. (That is, Λ_i explains the removal of $|D_i \setminus D_i^c|$ values.)*

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 be able to infer its domain size. For an explanation e_i to be correct, the variables in the left hand side Λ_i of e_i must precede the variable x_i in the agent order because the assignments of these variables have been used to remove values from the current domain of x_i. Hence, every explanation induces some ordering constraints, called *safety conditions* in [25] (see Definition 4).

Definition 9. *An explanation $e_k\colon\ \Lambda_k \rightarrow d_k^c$ is **coherent** with an order λ if all variables in Λ_k appear before x_k in λ. Given an explanation e_k, $S(e_k)$ is the set of safety conditions induced by e_k, that is, $S(e_k) = \{x_j \prec x_k \mid x_j \in \Lambda_k\}$. Given a set E_i of explanations, $S(E_i) = \bigcup_{e_k \in E_i} S(e_k)$.*

Each agent a_i stores a set E_i of explanations sent by other agents. During search, a_i updates E_i to store newly received explanations. Furthermore, each time an agent computes a new order, E_i is updated to remove explanations that are no longer *valid*.

Definition 10. *An explanation $e_k\colon \Lambda_k \rightarrow d_k^c$ in E_i is **valid** on agent a_i if it is* coherent *with the current order λ_i and Λ_k is* compatible *with the agent-view of agent a_i.*

When E_i contains a valid explanation e_k associated with agent a_k, agent a_i can infer the size of the current domain of x_k. Otherwise, a_i can assume that the size of the current domain of x_k is equal to its initial domain size d_k.[2] This gives the following function:

$$dom(k) = \begin{cases} d_k^c & \text{if } (e_k\colon \Lambda_k \rightarrow d_k^c) \in E_i, \\ d_k & \text{otherwise (i.e., } e_k\colon \emptyset \rightarrow d_k) \end{cases} \tag{1}$$

4.2.1. Simulating Min-Domain DVO Heuristic

The most well-known DVO heuristic from the centralized CSP is the ubiquitous min-domain [13]. This heuristic selects first the variable with smallest current domain. To mimic min-domain in AgileABT($[\alpha], \prec_\alpha$), an agent a_i simply needs to select an order λ that minimizes the termination value $\tau = [\alpha(\lambda[1]), \ldots, \alpha(\lambda[n])]$ w.r.t. $\prec_\alpha^{lex}$, where $\alpha(k) = dom(k)$ and $\prec_\alpha$ is the standard ordering $<$ on integers.

[2] The initial domain size of each agent can be known in a preprocessing step before search starts.

procedure `propagate`(c_{ij})
01. **foreach** ($v_i \in D_i^c$) **do**
02. **if** ($\neg hasSupport(c_{ij}, x_i), v_i$) **then**
03. $D_i^c \leftarrow D_i^c \setminus \{v_i\}$;
04. **foreach** ($v_j \in D_j^c$) **do**
05. **if** ($\neg hasSupport(c_{ij}, x_j), v_j$) **then**
06. $D_j^c \leftarrow D_j^c \setminus \{v_j\}$;
07. **if** ($D_i^c = \emptyset \vee D_j^c = \emptyset$) **then** $w_j \leftarrow w_j + 1$;

procedure `computeWeight()`
08. $wdeg_i \leftarrow 1$;
09. **foreach** ($c_{ij} \in \mathcal{C}_i$) **do**
10. **if** ($x_j \in \Gamma_i^+ \vee \langle x_j, nil, t_j \rangle \in AV^-$) **then**
11. $wdeg_i \leftarrow wdeg_i + w_j$;
12. $wdeg_i \leftarrow \min(wdeg_i, W)$;

Figure 7: Compute $wdeg(i)$, the weighted degree of agent a_i.

4.2.2. Simulating Neighborhood Based Variable Ordering Heuristics

In the second category we simulate three DVO heuristics from centralized CSP that take into account the neighborhood of each agent ([14, 31]): dom/deg, $dom/pdeg$ and $dom/fdeg$. Each of these DVOs prefers the variable minimizing a ratio where the numerator is the size of the domain of that variable and the denominator is some information about its neighborhood. To obtain measures that mimic these DVOs, agents use Equation (1) for the numerator, but they require information about the neighborhood of other agents to infer the denominator. For dom/deg, each agent a_i requires to know the degree $deg(k)$ of each agent a_k in the problem (i. e., the number of neighbors of a_k).[3] Agent a_i computes $\tau = [\alpha(\lambda[1]), \ldots, \alpha(\lambda[n])]$ using $\alpha(k) = \frac{dom(k)}{1 + deg(k)}$. For $dom/pdeg$ (resp. $dom/fdeg$) each agent a_i requires to know the set of neighbors of each other agent a_k because it will need to compute the incoming degree $pdeg(k)$ (resp. the outgoing degree $fdeg(k)$) of a_k for any proposed order.[4] Agent a_i computes τ using $\alpha(k) = \frac{dom(k)}{1 + pdeg(k)}$ (resp. $\alpha(k) = \frac{dom(k)}{1 + fdeg(k)}$) where the incoming degree $pdeg(k)$ in λ is the number of neighbors of a_k that appear before k in λ and the outgoing degree $fdeg(k)$ in λ is the number of neighbors of a_k that appear after k in λ. As these DVO heuristics prefer the variable minimizing the measure, they are simulated by calling AgileABT($[dom/deg], <$), AgileABT($[dom/pdeg], <$), and AgileABT($[dom/fdeg], <$), respectively.

4.2.3. Simulating Conflict-Directed Variable Ordering Heuristic

The third category covers the popular conflict-directed variable ordering heuristic $dom/wdeg$ from the centralized CSP [15]. The conflict-driven heuristic $dom/wdeg$ associates a weight with each constraint to record conflicts during search. The weight of a constraint is increased each time the constraint fails. This heuristic selects the variable that minimizes the ratio $dom/wdeg$, where dom denotes the current size of the domain of the variable and $wdeg$ the weighted degree of the variable defined as the sum of the weights of the constraints involving that variable and at least another uninstantiated variable.

[3] $deg(k)$ can be known before the search starts as is the case for the d_k of each agent.
[4] Again this information can be known in a preprocessing step.

AgileABT($[dom/wdeg], <$) is the algorithm that simulates the conflict-driven heuristic in AgileABT($[\alpha], \prec_\alpha$). Agents in AgileABT($[dom/wdeg], <$) store and exploit information about failures in the form of constraint weights to compute measure $\alpha(k)$. Each agent a_i maintains the weight of each constraint c_{ij} in $\mathcal{C}_i$, denoted by w_j, and the weighted degree of each agent a_k denoted by $wdeg_k$. Each agent a_i in AgileABT($[dom/wdeg], <$) computes τ using $\alpha(k) = \frac{dom(k)}{wdeg_k}$.

In order to check consistency (`isConsistent()` calls, Figure 4), agents perform successive revisions of their constraints using a procedure `propagate()`. A general description of procedure `propagate(`c_{ij}`)` called to revise a constraint $c_{ij} \in \mathcal{C}_i$ is presented in Figure 7. Whenever a constraint c_{ij} fails (i.e., $D_i^c = \emptyset \vee D_j^c = \emptyset$) when calling procedure `propagate`, w_j is incremented (line 7, Figure 7). Each time a_i assigns a new value to its variable (procedure `assignVariable()`, Figure 4), a_i computes its own counter $wdeg_i$ using procedure `computeWeight()`, Figure 7. The weighted degree $wdeg_i$ is the sum of the weights (w_j) of all constraints (c_{ij}) in $\mathcal{C}_i$ involving a variable having a lower priority ($x_j \in \Gamma_i^+$) or not assigned in AV^- (lines 8 to 11, Figure 7). However, to guarantee that AgileABT($[dom/wdeg], <$) terminates (see Section 4.4), we require that the new computed weighted degree does not exceed a limit W (line 12, Figure 7). Next, agent a_i attaches its counter $wdeg_i$ to each **ok?** message it sends out. Upon receiving an **ok?** message from an other agent a_s, agent a_i updates $wdeg_s$ to the weight included in the received **ok?** message.

4.3. AgileABT($[\alpha], \prec_\alpha$) algorithm

Once we have decided which DVO we want to mimic and designed the right measure α with the right total preference relation $\prec_\alpha$ over the range of α, if we want to mimic the given DVO as closely as possible, AgileABT($[\alpha], \prec_\alpha$) should compute orders with the smallest possible termination values (w.r.t. , $\prec_\alpha^{lex}$).

Each time an agent faces a dead-end, it resolves its no-goods and then it is allowed to propose a new order. Unlike previous polynomial reordering approaches, AgileABT($[\alpha], \prec_\alpha$) relaxes the restriction of selecting the variable that has the lowest priority in the current order among conflicting variables as backtracking target. AgileABT($[\alpha], \prec_\alpha$) allows agent a_i to select the backtracking target among conflicting variables Λ_i. The only restriction for selecting x_t as a backtracking target is to find an order λ associated with a termination value $\tau = [\alpha(\lambda[1]), \ldots, \alpha(\lambda[n])]$ that is smaller (w.r.t. $\prec_\alpha^{lex}$) than the termination value associated with the current order λ_i of agent a_i and x_t is the lowest among variables in Λ_i w.r.t. the freshly computed order λ'.

To deal with explanations, an AgileABT($[\alpha], \prec_\alpha$) agent a_i keeps a set of explanations E_i, and some of AgileABT procedures in (Figure 4) are slightly modified. The new lines/procedures of AgileABT($[\alpha], \prec_\alpha$) with respect to AgileABT are presented in Figure 8. Agents in AgileABT($[\alpha], \prec_\alpha$) exchange the same types of messages as AgileABT, however, each agent sends an explanation of its new size in each of its **ok?** messages.

As in AgileABT, whenever agent a_i receives an **ok?** message from an agent a_s, it processes it by calling procedure `processOk(`$\langle x_s, v'_s, t'_s \rangle, e_s$`)`. Agent a_i

```
procedure AgileABT([α], ≺_α)
01. initialize();
02. assignVariable();
03. while ( ¬end ) do
04.     msg ← getMsg();
05.     switch ( msg.type ) do
06.         ok?   : processOk(⟨x_s, v_s, t_s⟩, e_s);
07.         ngd   : processNgd(a_s, ngd[x_i≠v'_i]);
08.         adl   : processAdl(a_s, v'_i);
09.         order : processOrder(λ_s, τ_s);
10.         stp   : end ← true;

procedure processOk(⟨x_s, v'_s, t'_s⟩, e_s)
11. add(E_i, e_s);                    /* e_s: Λ_s → d^c_s */
12. updateAgentView(Λ_s ∪ ⟨x_s, v'_s, t'_s⟩);
13. checkAgentView();

function simulateExplanations(x_t)
14. E'_i ← E_i;
15. foreach ( e_k ∈ E'_i ) do
16.     if ( x_t ∈ Λ_k ) then     /* e_k: Λ_k → d^c_k */
17.         E'_i ← E'_i \ e_k ;
18. if ( e_t ∉ E'_i ) then
19.     add(E'_i, e_t : ∅ → d_t);
20. add(E'_i, e_t ← [Λ_t ∪ Λ_i \ (x_t = v_t)] → d^c_t − 1);
21. return(E'_i);

function computeOrder(E'_i)
22. λ and τ are arrays of length n;
23. p ← 1;
24. G ← {(j, k) | x_j ∈ Λ_k, e_k ∈ E'_i};
25. while ( p ≤ n ) do
26.     R ← {r | ∄(s, r) ∈ G};
27.     r ← arg min_{k∈R} {α(k)};   /* using ≺_α */
28.     λ[p] ← r;
29.     τ[p] ← α(r);
30.     p ← p + 1;
31.     foreach ( (r, s) ∈ G ) do
32.         G ← G \ (r, s);
33. return(⟨λ, τ⟩);

function f()
34. foreach ( e_k ∈ E_i ) do
35.     if ( ¬valid(e_k) ) then
36.         E_i ← E_i \ e_k ;
37. λ ← nil;  τ ← nil;
38. foreach ( x_t ∈ Λ_i ) do
39.     E'_i ← simulateExplanations(x_t);
40.     ⟨λ', τ'⟩ ← computeOrder(E'_i);
41.     if ( τ' ≺^{lex}_α τ ) then
42.         λ ← λ';
43.         τ ← τ';
44. return(⟨λ, τ⟩);
```

Figure 8: New lines/procedures of AgileABT($[\alpha], \prec_\alpha$) with respect to AgileABT in Figure 4.

updates its set of explanations E_i by storing the newly received explanation e_s (line 11, Figure 8). As usual the agent-view of a_i is updated, but in addition to the assignment of the sender (a_s), agent a_i takes newer assignments contained in the left hand side of the explanation (i.e., Λ_s) included in the received **ok?** message to update its agent-view (line 12, Figure 8). Then, procedure `checkAgentView` is called (line 13) to search a new consistent value for a_i if necessary.

To compute a new reordering, an AgileABT($[\alpha], \prec_\alpha$) agent a_i makes use of the function `f()` presented in Figure 8. First, a_i updates its set of explanations E_i to remove all explanations that are no longer valid (lines 34 to 36, Figure 8). As a result, all explanations in E_i are coherent with λ_i. Therefore, the set of safety conditions $S(E_i)$ does not contain cycles. Next, agent a_i iterates through all variables x_t in Λ_i, considering x_t as a potential backtracking target, (i. e., the directed no-good is $ngd[x_t \neq v_t] : [\Lambda_i \setminus (x_t = v_t) \rightarrow x_t \neq v_t]$) line 38. Agent a_i then predicts the set of explanations E_i after backtracking (function `simulateExplanations` call, line 39). E_i is updated to remove all explanations containing x_t (after backtracking x_t assignment will be changed), lines 14 to 17. Next, agent a_i updates the explanation of x_t by considering the new generated no-good $ngd[x_t \neq v_t]$ (lines 19 to 20). Finally, agent a_i calls

`computeOrder` to compute a new order with small termination value (line 40). Once all the potential backtracking targets have been tried, the function `f()` returns the computed order associated with the smallest computed termination value (line 44). Unfortunately, given a backtracking target, computing a total order λ that respects the ordering constraints induced by the set $S(E_i)$ while minimizing $[\alpha(\lambda(1)), \ldots, \alpha(\lambda(n))]$ is NP-hard (see Proposition 9 in Section 4.4).

As it is NP-hard to compute an order λ satisfying a set of safety conditions and minimizing termination value w.r.t. $\prec_\alpha^{lex}$ (Definition 7), AgileABT($[\alpha], \prec_\alpha$) uses a greedy function `computeOrder` to find a small termination value for $\prec_\alpha^{lex}$ while respecting the safety conditions. The basic idea of the function `computeOrder` is to perform a topological sort of the acyclic graph representing the ordering constraints with the objective of minimizing τ. The function `computeOrder` performs n iterations ranging from 1 to n (lines 25 to 32). At the p^{th} iteration, a variable x_r is selected such that $\alpha(r)$ is minimal among the set of variables R that have no predecessors, lines 26 to 27. Ties are broken lexicographically. The selected variable x_r is put in p^{th} position in the order, and $\tau[p]$ is assigned value $\alpha(r)$, lines 28 to 29. Next, x_r is removed from the acyclic graph, lines 31 to 32. Then, the next iteration is executed.

Correctness of function `computeOrder` is based on the fact that the set R computed at line 26 is never empty. This is true if `computeOrder` is called with a set E_i' of explanations such that $S(E_i')$ does not contain cycles. This is what Lemma 2 tells us.

Lemma 2. *When calling function* `computeOrder(`E_i'`)`, $S(E_i')$ *does not contain cycles.*

Proof. Function `computeOrder(`E_i'`)` is called on function `f()`. In function `f()`, all explanations in E_i that are no longer valid are removed (lines 34 to 36, Figure 8). As a result, all explanations in E_i are coherent with λ_i. Therefore, the set of safety conditions $S(E_i)$ does not contain cycles. We need now to prove that function `simulateExplanations` does not create cycles in the set of safety conditions $S(E_i')$ if $S(E_i)$ is acyclic. The first step is to copy E_i into E_i' (line 14, Figure 8). Then all explanations containing x_t are removed from E_i' (lines 15 to 17, Figure 8). By removing these explanations, all safety conditions of the form $x_t \prec x_j$ are removed from $S(E_i')$. Until this point, if $S(E_i)$ is acyclic, then $S(E_i')$ is acyclic because $S(E_i') \subseteq S(E_i)$. Then, the explanation of e_t is updated by taking the generated no-good into account (lines 19 and 20). All the safety conditions added to $S(E_t')$ after this step are of the form $x_l \prec x_t$. Suppose that once e_t is updated $S(E_i')$ becomes cyclic. Hence, if $S(E_i')$ contains cycles, all these cycles should contain x_t and there is a sequence of safety conditions in $S(E_i')$ such that $x_t \prec x_j \prec \cdots \prec x_l \prec x_t$. However, all safety conditions of the form $x_t \prec x_j$ were removed, and all the safety conditions in $S(e_t')$ are of the form $x_l \prec x_t$. Therefore, $S(E_i')$ cannot be cyclic. □

4.4. Theoretical Analysis

We first show that all measures presented in Section 4.2 satisfy the condition in Corollary 1. This is sufficient to ensure termination of AgileABT($[\alpha], \prec_\alpha$).

Proposition 7. *For all measures α in $\{dom,\ dom/deg,\ dom/pdeg,\ dom/fdeg,\ dom/wdeg\}$, $(\mathcal{R}^{\alpha}, <)$ is a well-ordering.*

Proof. We proceed by contradiction. Suppose there is an infinite decreasing sequence of values of $\alpha(k)$. In all measures above, $\alpha(k) = \frac{dom(k)}{\rho(k)}$, for some $\rho(k)$. $dom(k)$ is the expected domain size of the agent a_k. It is obvious that domain size cannot be negative, that is, $\forall k \in 1..n,\ 0 \leq dom(k) \leq d_k$. So $dom(k)$ cannot decrease indefinitely. Therefore, $\rho(k)$ must increase infinitely. $\rho(k)$ is a positive integer whose value depends on the measure used. Three cases were explored in this paper. The first case concerns min-domain (dom) where $\rho(k)$ equals 1. The second case concerns the family of degree-based heuristics (dom/deg, dom/pdeg, dom/fdeg). In this case, all of the $\rho(k)$ are greater than or equal to 1 and smaller than or equal to n the number of agents in the system because an agent is at most constrained to $n-1$ other agents. Thus, $1 \leq \rho(k) \leq n$. The third case is related to the heuristic dom/wdeg. We have outlined that an agent is not allowed to increment its weight when it has reached the limit W set beforehand (Figure 7, line 12). Thus, $1 \leq \rho(k) \leq W$. In all three cases $\rho(k)$ is an integer that cannot increase infinitely. Therefore, for all measures presented above, $\alpha(k) = \frac{dom(k)}{\rho(k)}$ cannot decrease infinitely, and so $<$ well-orders $\mathcal{R}^{\alpha}$ the range of measure α. □

Corollary 2. *AgileABT($[\alpha], \prec_{\alpha}$) is guaranteed to terminate if $\alpha \in \{dom,\ dom/deg,\ dom/pdeg,\ dom/fdeg,\ dom/wdeg\}$.*

Proof. From Corollary 1 and Proposition 7. □

Hence, AgileABT($[\alpha], \prec_{\alpha}$) has all the good properties we can expect from an algorithm for solving DisCSPs.

Corollary 3. *AgileABT($[\alpha], \prec_{\alpha}$) is sound, complete, and terminates if $\alpha \in \{dom,\ dom/deg,\ dom/pdeg,\ dom/fdeg,\ dom/wdeg\}$.*

Proof. From Proposition 2, Proposition 5, and Corollary 2. □

We demonstrate that the space complexity of AgileABT($[\alpha], \prec_{\alpha}$) is polynomially bounded.

Proposition 8. *The spatial complexity of AgileABT($[\alpha], \prec_{\alpha}$) is polynomial.*

Proof. The size of no-goods, explanations, termination values, and orderings, is bounded by n, the total number of variables. Now, on each agent, AgileABT($[\alpha], \prec_{\alpha}$) only stores one no-good per value, one explanation per agent, one termination value and one ordering. Thus, the space complexity of AgileABT($[\alpha], \prec_{\alpha}$) is in $\mathcal{O}(nd + n^2 + n + n) = \mathcal{O}(nd + n^2)$ on each agent. □

We prove that it is difficult for an agent a_i to compute an order with the smallest possible termination value.

Proposition 9. *Given a DisCSP defined by the network* $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$*, with* $\mathcal{X} = \{x_1, \ldots, x_n\}$*, given an agent* a_i*, and given a measure* α *from* $[1..n]$ *to* $\mathbb{Q}$*, it is NP-hard to compute a total order* λ *on* $\mathcal{X}$ *that satisfies the safety conditions in* $S(E_i)$ *and such that* $[\alpha(\lambda[1]), \ldots, \alpha(\lambda[n])]$ *is lexicographically minimum, even if* $S(E_i)$ *is acyclic.*

Proof. Our proof is inspired from a proof of hardness of finding minimal topological sorts of a graph [32]. We reduce the CLIQUE problem (one of Karp's 21 NP-complete problems), to our problem of finding a total order λ on $\mathcal{X}$ that satisfies the safety conditions in $S(E_i)$ and such that $[\alpha(\lambda[1]), \ldots, \alpha(\lambda[n])]$ is lexicographically minimum. Let $G = (X, H)$ be an undirected graph in which we want to decide if there exists a clique of size k. We build a DisCSP with network $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$. $\mathcal{X}$ contains $|X| + |H| + 1$ variables, where each node i in X is associated with a variable u_i, each edge (i, j) is associated with a variable w_{ij}, plus a variable z. $\mathcal{A}$ contains an agent per variable of $\mathcal{X}$. The variables u_i have two values $\{i, i+1\}$ in their domain. The variables w_{ij} have three values $\{i, j, i + j \cdot |X|\}$ in their domain, so that value $i + j \cdot |X|$ only belongs to one w_{ij} variable. The variable z has domain $\{i + j \cdot |X| \mid (i, j) \in H\}$ that contains $|H|$ values, each value corresponding to an edge $(i, j) \in H$. $\mathcal{C}$ is composed of $3 \cdot |H|$ constraints: $u_i \neq w_{ij}$, $u_j \neq w_{ij}$, and $w_{ij} \neq z$, for each edge $(i, j) \in H$. The measure α is defined by $\alpha(x) = dom(x)$, for any variable x in $\mathcal{X}$. The initial order among agents is such that all u_i appear before all w_{ij}, and z is the last. Finally suppose all agents instantiate their variables in lexicographic order of their values. Once a variable w_{ij} has received **ok?** messages from its two predecessors u_i and u_j, it eliminates values i and j from its domain and assigns value $i + j \cdot |X|$ to its variable. Afterwards, it sends a new **ok?** message to the variable z. In addition to the new value of the variable w_{ij}, this **ok?** message contains the explanation $u_i = i \wedge u_j = j \rightarrow 1$. Once the variable z has received all the **ok?** messages from variables w_{ij}, it has to backtrack because all values in its domain become forbidden. It has to compute a new order minimizing the termination value, that is, an order λ such that $[\alpha(\lambda[1]), \ldots, \alpha(\lambda[n])]$ is lexicographically minimum. For each $w_{ij} \in \mathcal{X}$, there are two safety conditions $(u_i \prec w_{ij})$ and $(u_j \prec w_{ij})$ in $S(E_z)$ induced by the explanation sent by w_{ij}. Observe that $S(E_z)$ is thus acyclic. In addition, for each $i \in X, \alpha(u_i) = 2$, and for each $(i, j) \in H, \alpha(w_{ij}) = 1$ (because no-goods have removed values i and j from the domain of w_{ij}). As a result, G contains a clique of size k if and only if the lexicographically minimum vector $[\alpha(\lambda[1]), \ldots, \alpha(\lambda[n])]$ satisfying the acyclic safety conditions accepts the word $22121^2 21^3 ... 21^{k-1}$ as prefix, where 2^i denotes the sequence of i consecutive 2.

To prove this claim we first observe that an order λ satisfying the safety conditions cannot put a variable w_{ij} before u_i and u_j. We now analyze the two directions of this claim. If G contains a clique of size k, we generate a sequence of variables the following way. Start from the empty sequence. Select each node j in turn in the clique of size k. Add u_j to the sequence, followed by all w_{ij} such that u_i is already in the sequence. At the end of this process we have built a prefix of an order λ such that $22121^2 21^3 ... 21^{k-1}$ is a prefix

of $[\alpha(\lambda[1]), \ldots, \alpha(\lambda[n])]$. By construction, any lexicographically smaller prefix would put a w_{ij} before u_i or u_j in λ , which would break the safety conditions. Suppose now that we have an order λ such that $22121^2 21^3 ... 21^{k-1}$ is a prefix of $[\alpha(\lambda[1]), \ldots, \alpha(\lambda[n])]$. By construction, this means that there are k nodes in G that are linked by $[1 + 2 + \cdots + (k-1)] = k(k-1)/2$ edges. In other words, these nodes form a clique in G.

We have proved that the existence of a clique of size k is equivalent to the existence of the prefix $22121^2 21^3 ... 21^{k-1}$ in $[\alpha(\lambda[1]), \ldots, \alpha(\lambda[n])]$. Checking this prefix is polynomial, therefore we have a polynomial-time reduction, and our problem is NP-hard. □

4.5. Example of Running AgileABT($[dom], <$)

Figure 9 presents an example of a possible execution of AgileABT($[dom], <$) on the simple problem presented in Figure 1. Assume all 5 agents start with the same initial ordering $\lambda_i = [1, 2, 3, 4, 5]$ associated with the termination value $\tau_i = [4, 4, 4, 4, 4]$ and values are chosen lexicographically. Consider the situation in a_5 after receiving **ok?** messages from other agents (Figure 9a). On receipt, explanations e_1, e_2, e_3, and e_4 are stored in E_5, and assignments $x_1 = 1$, $x_2 = 2$, $x_3 = 2$, and $x_4 = 3$ are stored in a_5 agent-view. After checking its constraints with higher neighbors (c_{15}, c_{25}, and c_{45}), a_5 detects a dead end ($\Lambda_5 \rightarrow 0$) where Λ_5: $\{x_1 = 1 \wedge x_2 = 2 \wedge x_4 = 3\}$. All explanations stored in E_5 are valid: They are compatible with the agent-view of a_5 and coherent with λ_5. Agent a_5 iterates through all variables $x_t \in \Lambda_5$, considering x_t as the target of the backtracking. Figure 9b shows the updates on the explanations stored in a_5 (E_5) when it considers x_1 as the target of the backtracking (i. e., $x_t = x_1$). Agent a_5 updates E_5 to remove all explanations containing x_1 (i. e., e_2 and e_3) and considering the new generated no-good $ngd[x_1 \neq 1] : [x_2 = 2 \wedge x_4 = 3] \rightarrow x_1 \neq 1$ in the explanation of x_1, i.e. e_1 (Figure 9b, left). Finally, a_5 computes a new order (λ') and its associated termination value (τ') from the updated explanations E_5. λ' is obtained by performing a topological sort on the directed acyclic graph formed by safety conditions induced by the updated explanations E_5 (Figure 9b, right). Figure 9c presents the computed orderings and their associated termination values (by topological sort) when considering each $x_t \in \Lambda_5$ as backtracking target. The strongest computed order (i.e., $\lambda' = [3, 4, 2, 5, 1]$, $\tau' = [4, 2, 4, 2, 3]$) is that computed when considering x_1 as backtracking target. Since τ' is smaller than τ_5 (i.e., $\tau' \prec_{\alpha}^{lex} \tau_5$) agent a_5 changes its current order to λ' and proposes this ordering to all other agents through **order** messages (i. e., **order**:$\langle \lambda', \tau' \rangle$). Then, a_5 sends the no-good $ngd[x_1 \neq 1] : [x_4 = 3 \wedge x_2 = 2] \rightarrow x_1 \neq v_1$ to agent x_1.

5. AgileABT versus retroactive ABT_DO

As mentioned in the introduction, several algorithms have been proposed for reordering agents dynamically during search. AgileABT and retroactive ABT_DO (i. e., ABT_DO_Retro [20]) are the only ones which are able to reorder

Data structures maintained by A_5

incoming messages	**explanations** (E_5)	**no-goods** ($D_5 = \emptyset$)
ok?$_{1\to5}$:$\langle(x_1=1), e_1$: $[]\to 4\rangle$	e_1: $[] \to d_1=4$	$ngd(x_5{\neq}1)$: $[x_1=1]\to x_5{\neq}1$
ok?$_{2\to5}$:$\langle(x_2=2), e_2$: $[x_1=1]\to 3\rangle$	e_2: $[x_1=1]\to d_2=3$	$ngd(x_5{\neq}2)$: $[x_2=2]\to x_5{\neq}2$
ok?$_{3\to5}$:$\langle(x_3=2), e_3$: $[x_1=1]\to 3\rangle$	e_3: $[x_1=1]\to d_3=3$	$ngd(x_5{\neq}3)$: $[x_1=1]\to x_5{\neq}3$
ok?$_{4\to5}$:$\langle(x_4=3), e_4$: $[x_3=2]\to 2\rangle$	e_4: $[x_3=2]\to d_4=2$	$ngd(x_5{\neq}4)$: $[x_4=3]\to x_5{\neq}4$

$\forall i \in 1..5,\ \lambda_i = [1,2,3,4,5],\ \tau_i = [4,4,4,4,4]$

A_5: $x_1=1 \wedge x_2=2 \wedge x_4=3$

(a) Explanations and no-goods maintained by a_5

A_5: when choosing x_1 as backtracking target for A_5

e_1: $[x_2=2 \wedge x_4=3] \to d_1=3$

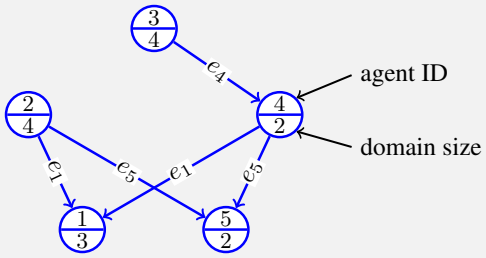
e_2: $[] \to d_2=4$

e_3: $[] \to d_3=4$

e_4: $[x_3=2] \to d_4=2$

e_5: $[x_2=2 \wedge x_4=3] \to d_5=2$

Updated explanations (E'_5)



Directed graph

(b) a_5: updated explanations with x_1 as target.

A_5: target selection

	$\lambda_5 = [1,2,3,4,5]$	$\tau_5 = [4,4,4,4,4]$
directed nogood	**agent ordering**	**termination value**
$ngd[x_1{\neq}1]$: $[x_4=3 \wedge x_2=2] \to x_1{\neq}1$	$\lambda' = [3,4,2,5,1]$	$\tau' = [4,2,4,2,3]$
$ngd[x_2{\neq}2]$: $[x_1=1 \wedge x_4=3] \to x_2{\neq}2$	$\lambda' = [1,3,4,5,2]$	$\tau' = [4,3,2,1,2]$
$ngd[x_4{\neq}3]$: $[x_1=1 \wedge x_2=2] \to x_4{\neq}3$	$\lambda' = [1,2,5,3,4]$	$\tau' = [4,3,1,3,1]$

Selected no-good: $ngd[x_1{\neq}1]$: $[x_4=3 \wedge x_2=2] \to x_1{\neq}1$

(c) a_5: selection of target (x_1).

Figure 9: An example of a possible execution of AgileABT($[dom], <$) on the problem of Figure 1.

the agents with priority higher than the no-good receiver.[5] Therefore the main differences between these two algorithms deserve to be explored in more depth

5.1. Reordering

5.1.1. ABT_DO_Retro

The algorithm ABT_DO was first introduced in [17]. When an ABT_DO agent assigns a value to its variable, it can only change the order of agents that have a lower priority than its own. ABT_DO_Retro, presented in [20], is a slightly modified version of ABT_DO where the reordering operation is generated by the no-good generator instead of the no-good receiver. Furthermore, ABT_DO_Retro allows for retroactive heuristics. A retroactive heuristic enables moving the no-good sender to a higher position than the no-good receiver. The more the no-good sender is moved higher in the order, the more the heuristic is flexible.

However, the flexibility of the retroactive heuristics enabled by ABT_DO_Retro comes at the price of keeping no-goods in memory. To monitor the storage of the no-goods, ABT_DO_Retro uses a parameter K and only allows the storage of no-goods that are smaller than or equal to K in size. Hence, ABT_DO_Retro has a space complexity exponential in K.

When K is equal to the number of agents in the system, the no-good sender can always be moved to be before all the participants in the no-good. The resulting algorithm is a generalization of AWC [20]. However, agents must store all the no-goods until the end of the search, which gives a space complexity exponential in the number of agents.

When K equals 0, the spatial complexity of the algorithm becomes polynomial, but the flexibility becomes limited because the sender of the no-good, can no longer be moved to a position higher than the second last in the no-good [20].

An intermediate version between these two extremes consists in setting the parameter K to a value between 0 and the number of agents in the system. In this case, if the no-good created has size larger than K, the sender of the no-good cannot be moved to a position higher than the second last in the no-good. If the no-good has size smaller than or equal to K, the sending agent can be moved to be higher than all the participants in the no-good and the no-good is sent to and stored by all of them.

Despite allowing this space consumption, the extra flexibility given to ABT_DO_Retro had no positive effects on its performance. Indeed, "The fact that a larger storage, which enables more flexibility of the heuristic, actually causes a deterioration of the performance might come as a surprise. " [quoted from [20, p.193]]. Furthermore, "Larger storage for Nogoods (even exponential in the extreme case) was found to produce worse efficiency for search on random problems." [quoted from [20, p.197]]. The only heuristic that performed well

[5] AWC also allows retroactive reordering, but ABT_DO_Retro is a generalization of AWC [20].

was the min-domain retroactive heuristic that works with K set to 0: "In our best performing heuristic, agents are moved higher in the priority order as long as their domain size is smaller than the domains of the agents before them and as long as they do not pass the second last in the no-good." [quoted from Zivan et al. [20, p.197]].

5.1.2. AgileABT

To increase flexibility, AgileABT uses a totally different and innovative strategy based on termination values. To be able to reorder agents that are before the receiver of the no-good, agents in AgileABT do not need to keep no-goods in memory. They only need to improve the termination value. An agent that can improve the termination value can propose a new order where the positions of all agents are changed, including the agents before the receiver of the no-good.

5.2. Selection of the Backtracking target

5.2.1. ABT_DO_Retro

In all previous asynchronous backtracking algorithms, including ABT_DO [17] and ABT_DO_Retro [20], whenever a dead end occurs, the backtracking target is determined by the order known by the generator of the no-good before the dead end. That is, the backtracking target always corresponds to the agent with the lowest priority among those participating in the no-good.

5.2.2. AgileABT

In order to ensure greater flexibility in exploring the search space, the restriction of selecting the lowest agent in the current order as backtracking target has been relaxed in AgileABT. Instead of being determined by the order known by the generator of the no-good before the dead end, the backtracking target is determined by the order newly proposed by this agent. By proposing an appropriate new order, any variable participating in the generated no-good can be moved down to the lowest position in that generated no-good and then be selected as the backtracking target (Figure 4, lines 43 to 46).

The example of Section 4.5 illustrates this feature. When the no-good ngd_6 : $\neg[(x_2 = 2) \wedge (x_3 = 2)]$ was generated by a_4, the current order that was known by this agent was $[1, 2, 3, 4]$. In all previous asynchronous backtracking algorithms, including ABT_DO [17] and ABT_DO_Retro [20], the conclusion of the no-good ngd_6 must be x_3 because it is owned by the agent that has the lowest priority according to the current order. In AgileABT, this restriction can be relaxed. Indeed, by choosing the new order $[3, 2, 4, 1]$, the variable x_2 was dragged down to the lowest position among the variables participating in ngd_6 and the no-good $x_3 = 2 \rightarrow x_2 \neq 2$ was sent to a_2 instead of a_3.

5.3. Timestamps versus termination values

5.3.1. ABT_DO_Retro

The method of timestamping for defining the most updated order in ABT_DO_Retro is the same as that used in [17]. In these algorithms, an order

is an ordered list of pairs where every pair includes the ID of one of the agents and a counter attached to it. Initially, all counters are set to zero, and each agent a_i that proposes a new order, updates the counters as follows:

- The counters of agents with priority higher than a_i are not changed.
- The counter of a_i is incremented by one.
- The counters of agents with priority lower than a_i are set to zero.

The counters attached to the agents ID form a timestamp. Agents decide which order is more up-to-date by comparing the timestamps lexicographically [21]. Therefore, the timestamps used by ABT_DO_Retro are arrays of integers whose sole purpose is to indicate that a given order is more up-to-date than another one.

5.3.2. AgileABT

Termination values used by AgileABT are fundamentally different from the timestamps of ABT_DO_Retro. First, termination values do not need to be arrays of integers. They may be numbers, arrays or any other mathematical object. The only requirement is that $(\mathcal{T}, \prec_\tau)$ is a well ordering (Proposition 4). The example of Section 4.5 illustrates a case where termination values are positive integers, and the heuristics introduced in Section 4.2 illustrate cases where termination values are arrays of integers and arrays of rational numbers.

Second, termination values can be used to mimic a wide variety of DVOs. The termination values used by the instantiations of AgileABT presented in Section 4 are arrays of measures α defined such that the smaller the value returned by α w.r.t. $\prec_\alpha$, the more preferred the corresponding variable for the DVO heuristic represented by α. We showed in Section 4.2 how AgileABT$([\alpha], \prec_\alpha)$ is able, thanks to the termination values, to capture a number of different DVO heuristics (*dom*, *dom/deg* and *dom/wdeg*) that are known to be effective in reducing search in centralized CSP.

6. Empirical Analysis

In this section we experimentally compare AgileABT$([\alpha], \prec_\alpha)$[6] using different DVO heuristics to three other algorithms: ABT, ABT_DO with nogood-triggered heuristic (ABT_DO-ng) [17] and ABT_DO with min-domain retroactive heuristic (ABT_DO_Retro(mindom)) [20, 21]. All experiments were performed on the DisChoco 2.0 platform [33].[7] DisChoco implements a model to solve Distributed Constraint Reasoning problems. Communication is performed via the Simple Agent Communication Infrastructure (SACI) if agents are implemented on different Java Virtual Machine (JVM). Otherwise, if agents are

[6] For AgileABT$([dom/wdeg], <)$, we fixed $W = 10^3$. But, varying W (to 10^4 or 10^5) made negligible difference to the results.

[7] http://dischoco.sourceforge.net/

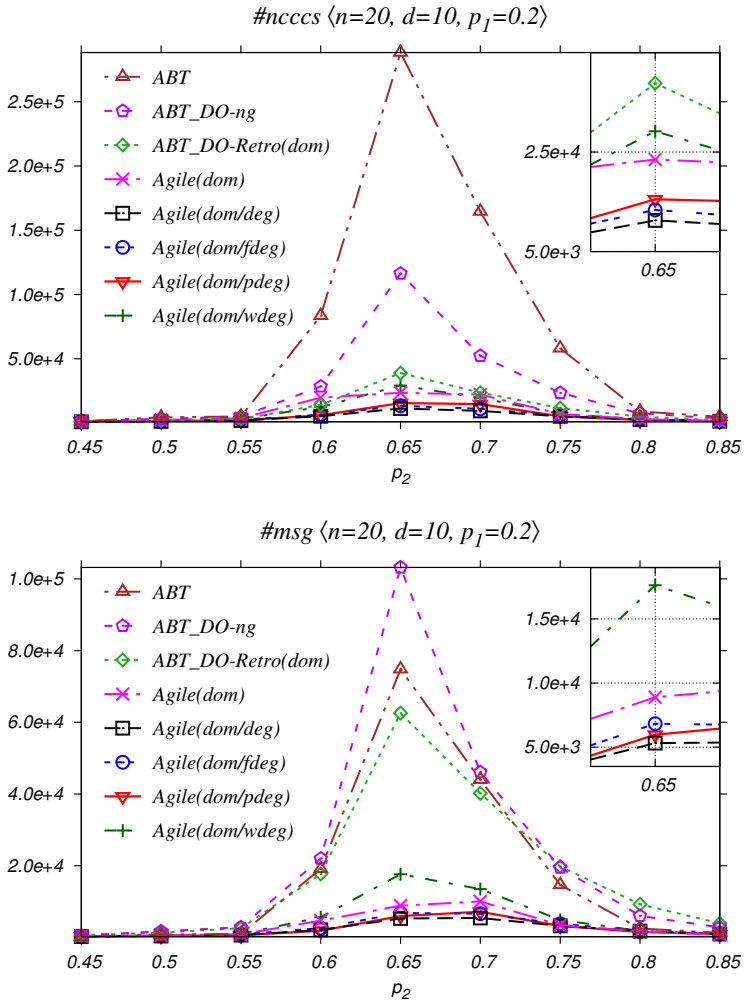


Figure 10: Sparse uniform binary random DisCSPs

simulated by Java threads on a single JVM, the communication is performed only through message passing via a local communication simulator. This is this second option that we have used in our experiments.

When comparing distributed algorithms, the performance is evaluated using two common metrics: the communication load and computation effort. Communication load is measured by the total number of exchanged messages among agents during algorithm execution ($\#msg$) [34]. Computation effort is measured by the number of non-concurrent constraint checks ($\#ncccs$) [35]. $\#ncccs$ is the metric used in distributed constraint solving to simulate computation time, but for dynamic reordering algorithms its variant generic $\#ncccs$ is used [36]. Al-

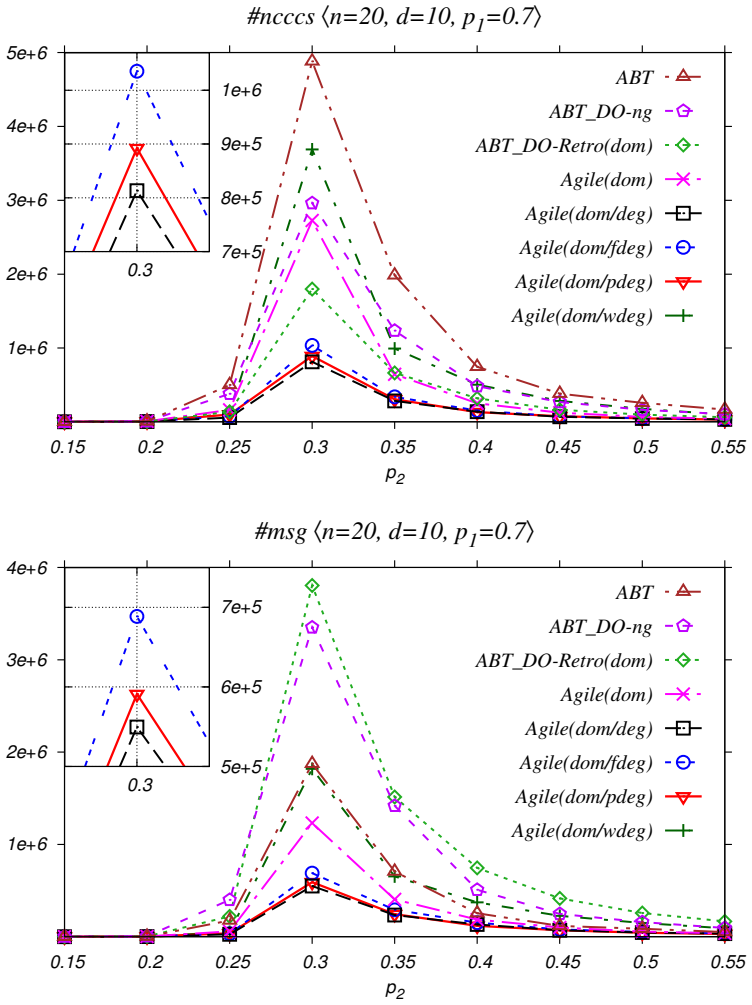


Figure 11: Dense uniform binary random DisCSPs

gorithms are evaluated on four benchmarks: uniform binary random DisCSPs, distributed graph coloring problems, composed random instances and target tracking in distributed sensor networks.

6.1. Uniform binary random DisCSPs

Uniform binary random DisCSPs are characterized by $\langle n, d, p_1, p_2 \rangle$, where n is the number of agents/variables, d is the number of values in each domain, p_1 is the network connectivity defined as the ratio of existing binary constraints to possible binary constraints, and p_2 is the constraint tightness defined as the ratio of forbidden value pairs to all possible pairs. We solved instances of two

classes of random DisCSPs: sparse problems $\langle 20, 10, 0.2, p_2 \rangle$ and dense problems $\langle 20, 10, 0.7, p_2 \rangle$. We varied the tightness from 0.1 to 0.9 by steps of 0.05. For each pair of fixed density and tightness (p_1, p_2), we generated 20 instances, solved 5 times each. We report average over the 100 executions.

Figures 10 and 11 show the results on sparse respectively dense uniform binary random DisCSPs. In sparse problems (Figure 10), ABT is significantly the slowest algorithm but it requires fewer messages than ABT_DO-ng. AgileABT($[\alpha], \prec_\alpha$) outperforms all other algorithms on both $\#msg$ and $\#ncccs$. AgileABT($[dom/wdeg], <$) is slower and requires more messages compared to other heuristics of AgileABT($[\alpha], \prec_\alpha$). Regarding the messaging, the neighborhood based heuristics (i. e., dom/deg, $dom/pdeg$ and $dom/fdeg$) perform well. They show more than an order of magnitude improvement compared to ABT, ABT_DO-ng and ABT_DO_Retro(mindom) and an almost two-fold improvement over $dom/wdeg$. Regarding the speedup, AgileABT($[\alpha], \prec_\alpha$) shows at least an order of magnitude improvement compared to ABT. The neighborhood based heuristics (i. e., dom/deg, $dom/pdeg$ and $dom/fdeg$) show an order of magnitude improvement compared to ABT_DO-ng. Comparing AgileABT($[\alpha], \prec_\alpha$) algorithms, neighborhood based heuristics (i. e., dom/deg, $dom/pdeg$ and $dom/fdeg$) show an almost two-fold improvement over dom and $dom/wdeg$ on $\#ncccs$.

In dense problems (Figure 11), AgileABT($[\alpha], \prec_\alpha$) with neighborhood based heuristics outperforms all other algorithms both on $\#msg$ and $\#ncccs$. The improvement factor is almost 6 over ABT on $\#ncccs$ and almost 7 in $\#msg$. Only neighborhood based heuristics are faster than ABT_DO_Retro(mindom). ABT and AgileABT($[dom/wdeg], <$) are the slowest algorithms but they require almost half the $\#msg$ of ABT_DO-ng and ABT_DO_Retro(mindom). AgileABT($[dom/wdeg], <$) shows poor performance compared to other heuristics of AgileABT($[\alpha], \prec_\alpha$).

6.2. Distributed graph coloring problems

Distributed graph coloring problems are characterized by $\langle n, d, p_1 \rangle$, where n, d and p_1 are as above and all constraints are binary difference constraints. We report the average on 100 instances of two classes $\langle n = 15, d = 5, p_1 = 0.65 \rangle$ and $\langle n = 25, d = 5, p_1 = 0.45 \rangle$ in Table 1. Again, AgileABT($[\alpha], \prec_\alpha$) using neighborhood based DVO are by far the best algorithms for solving both classes. The results show that AgileABT($[dom/pdeg], <$) outperforms all other algorithms in both classes. ABT_DO-ng shows poor performance on solving those problems. ABT_DO_Retro(mindom) outperforms AgileABT($[dom], <$) and AgileABT($[dom/wdeg], <$) in both classes when comparing $\#ncccs$, but require more $\#msg$ than them. Comparing AgileABT($[dom], <$) to AgileABT($[dom/wdeg], <$), dom requires fewer messages compared to $dom/wdeg$ but it is slower than AgileABT($[dom/wdeg], <$) on large problems.

Table 1: Distributed graph coloring problems

Algorithm	$\langle 15, 5, 0.65 \rangle$		$\langle 25, 5, 0.45 \rangle$	
	$\#msg$	$\#ncccs$	$\#msg$	$\#ncccs$
AgileABT($[dom/wdeg], <$)	64,587	184,641	1,155,373	2,068,677
AgileABT($[dom/fdeg], <$)	42,305	109,348	453,888	821,397
AgileABT($[dom/pdeg], <$)	24,174	67,829	197,877	396,320
AgileABT($[dom/deg], <$)	29,688	78,200	255,434	504,600
AgileABT($[dom], <$)	48,095	166,564	1,054,793	2,386,179
ABT_DO_Retro(mindom)	76,228	87,480	1,208,224	678,424
ABT_DO-ng	139,866	239,424	4,110,364	3,131,566
ABT	77,536	133,354	1,531,776	3,413,261

Table 2: Composed random instances

Instances	**25-1-25**		**25-1-40**	
Algorithm	$\#msg$	$\#ncccs$	$\#msg$	$\#ncccs$
AgileABT($[dom/wdeg], <$)	19,133	22,205	23,313	23,728
AgileABT($[dom/fdeg], <$)	56,158	139,106	112,389	316,381
AgileABT($[dom/pdeg], <$)	9,511	10,786	10,911	11,512
AgileABT($[dom/deg], <$)	38,026	150,880	156,164	763,136
AgileABT($[dom], <$)	9,871	14,328	10,920	14,286
ABT_DO_Retro(mindom)	53,566	27,507	69,627	37,049
ABT_DO-ng	1,045,077	1,166,210	14,400,090	13,017,189
ABT	1,327,065	7,883,914	10,219,262	64,570,955

6.3. Composed random instances

We also evaluate all algorithms on two sets of unsatisfiable composed random instances used to evaluate the conflict-directed variable ordering heuristic in centralized CSP [37, 15].[8] Each set contains 10 different instances where each instance is composed of a main (under-constrained) fragment and some auxiliary fragments, each of which being grafted to the main one by introducing some binary constraints. Each instance contains 33 variables and 10 values per variable, and as before, each variable is controlled by a different agent. We solved each instance 5 times and present the average over 50 executions in Table 2.

The results (Table 2) show that AgileABT($[dom/pdeg], <$) outperforms

[8] http://www.cril.univ-artois.fr/∼lecoutre/benchmarks.html

all other algorithms in both sets. The second best algorithm for solving these instances is AgileABT($[dom], <$). ABT_DO_Retro(mindom) outperforms AgileABT($[dom/deg], <$) and AgileABT($[dom/fdeg], <$) but requires more $\#msg$ than AgileABT($[dom/deg], <$) in the set 25-1-25. ABT shows very poor performance on solving these problems followed by ABT_DO-ng. AgileABT($[dom/pdeg], <$) shows 3 orders of magnitude improvement compared to ABT and ABT_DO-ng. Regarding AgileABT($[\alpha], \prec_\alpha$) DVOs, $dom/wdeg$ pays off on these instances compared to dom/deg and $dom/fdeg$.

6.4. Target tracking in distributed sensor networks

The *Target Tracking in Distributed Sensor Network* (SensorDisCSP) [4, 38] is a benchmark based on a real distributed problem. This problem consists of a set of n stationary sensors, and a set of m mobile targets, moving through their sensing range. The objective is to track each target by sensors. Thus, sensors have to cooperate for tracking all targets. In order for a target to be tracked accurately, at least three sensors must concurrently turn on overlapping sectors. This allows the target's position to be triangulated. However, each sensor can track at most one target. Hence, a solution is an assignment of three distinct sensors to each target. A solution must satisfy visibility and compatibility constraints. The visibility constraint defines the set of sensors to which a mobile is visible. The compatibility constraint defines a compatibility relation among sensors regarding the physical limitations of the sensors and the properties of the terrain on which the sensors are located.

We encode SensorDisCSP in DisCSP as follows. Each agent represents one mobile. There are three variables per agent, one for each sensor that we need to allocate to the corresponding mobile. The 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 pairwise 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 into an equivalent formulation where we have three virtual agents for each real agent. Each virtual agent handles a single variable but $\#msg$ does not take into account messages exchanged between virtual agents belonging to the same real agent.

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. Each sensor can communicate with a fraction p_c of the sensors that are in its sensing range, and each mobile can be tracked by a fraction p_v of the sensors having the mobile in their sensing range. We solved instances of class $\langle 25, 6, 0.15, p_v \in \{.1, \ldots, .65\} \rangle$, where we vary p_v by steps of 0.05. Again, for each pair $(p_c,\ p_v)$ we generated 20 instances, solved 5 times each, and averaged over the 100 runs. The results are shown in Figure 12.

When comparing the speed-up of algorithms (top of Figure 12), AgileABT($[dom/wdeg], <$) is the fastest algorithm and it shows more than an order of magnitude improvement compared to ABT that shows very

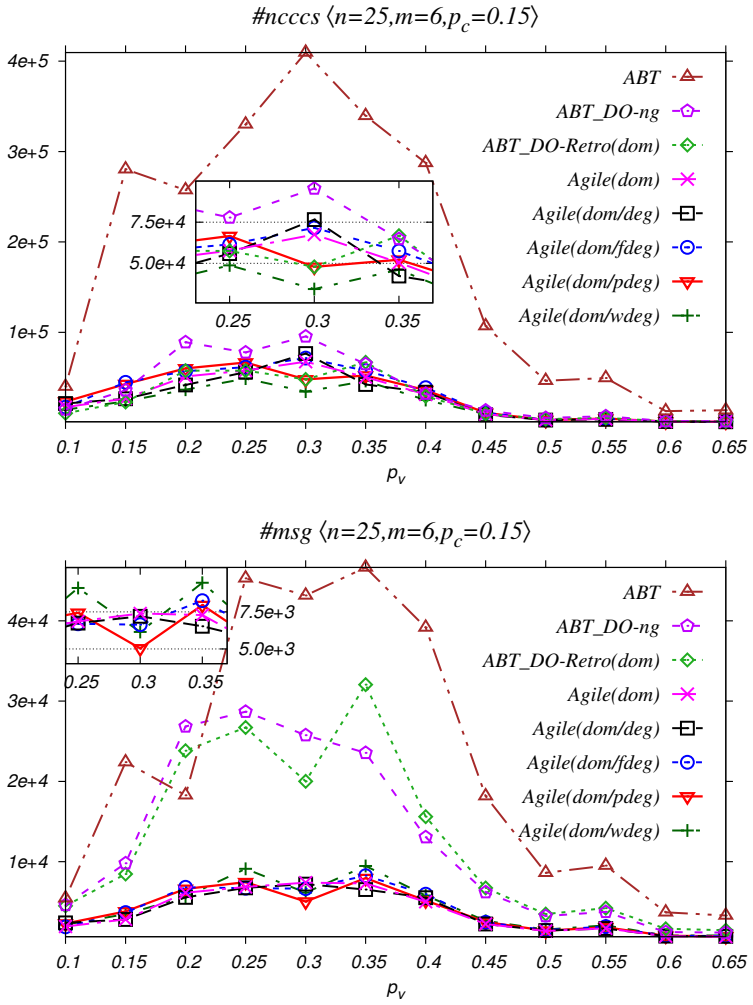


Figure 12: Distributed sensor networks

poor performance compared to dynamic asynchronous algorithms. Comparing dynamic asynchronous algorithms, ABT_DO-ng is outperformed by AgileABT($[\alpha], \prec_\alpha$) and ABT_DO_Retro(mindom). ABT_DO_Retro(mindom) and AgileABT($[dom/pdeg], <$) show the same performance and they slightly outperform AgileABT($[\alpha], \prec_\alpha$) with dom, dom/deg and $dom/fdeg$. Concerning communication load (bottom of Figure 12), AgileABT($[\alpha], \prec_\alpha$) heuristics outperform all other algorithms. ABT is the algorithm that requires the most messages to solve SensorDisCSP instances. ABT_DO-ng and ABT_DO_Retro(mindom) require almost half the $\#msg$ of ABT. All AgileABT($[\alpha], \prec_\alpha$) heuristics require almost the same $\#msg$ to solve Sensor-

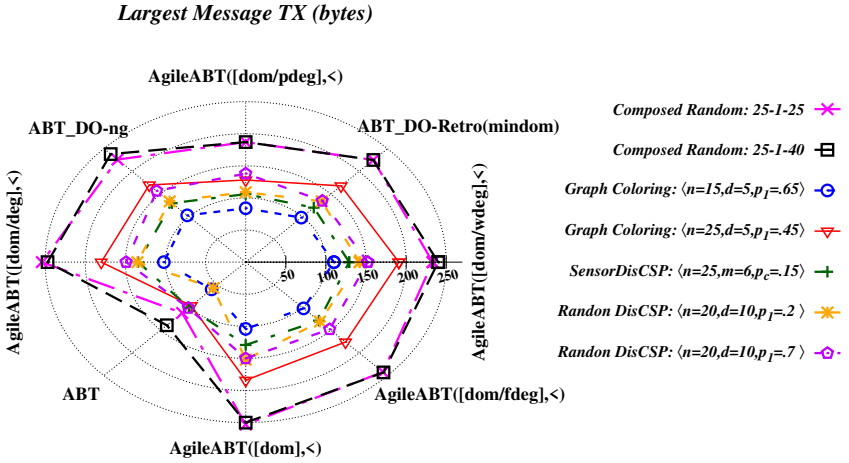


Figure 13: The longest message transmitted in bytes.

DisCSP instances. They show an improvement factor of 3.5 over ABT_DO-ng and ABT_DO_Retro(mindom) and an improvement factor of 7 over ABT.

6.5. Evaluation of messages size

In order to assess the communication load we measured $\#msg$, the total number of exchanged messages among agents during algorithm execution [34]. Our experiments show that AgileABT($[\alpha], \prec_\alpha$) generally needs fewer messages than other algorithms. However, AgileABT($[\alpha], \prec_\alpha$) messages can be longer than those sent by other algorithms. One could object that for AgileABT($[\alpha], \prec_\alpha$), 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 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 $\mathbf{1,272}$ bytes for IPv6 when using either TCP or UDP (UDP is 8 bytes smaller than TCP, see RFC-768 –http://tools.ietf.org/html/rfc768).

Figure 13 presents the size of the longest message sent by each algorithm on all our experiments. The results show that in all the compared algorithms the size of the longest message sent is larger when solving composed random instances (it is between 147 bytes for ABT and 253 bytes for AgileABT($[dom/pdeg], <$)). Solving graph coloring instances requires the

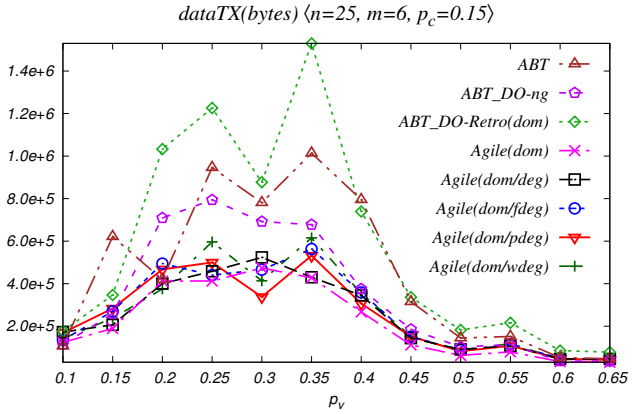
smaller longest messages for all compared algorithms. The size of largest message size is 105 and it is exchanged by AgileABT($[dom/wdeg], <$). In ABT the size of longest message size is smaller than in all asynchronous dynamic ordering algorithms. AgileABT($[\alpha], \prec_\alpha$) requires larger longest messages compared to other algorithms. However, among all our experiments, the size of the longest message sent by AgileABT($[\alpha], \prec_\alpha$) was of size 253 bytes. We are able to send a single datagram of up to 568 bytes (IPv4) or $1,272$ bytes (IPv6) without fragmentation (i. e., send in one single physical message) in either TCP or UDP. Thus, counting the number of exchanged messages is equivalent to counting the number of physical messages. Therefore, in all our experiments assessing the communication load by $\#msg$ is not biased.

Figure 14 presents the total number of bytes exchanged on the uniform binary random DisCSPs and the target tracking in distributed sensor network benchmark. The obtained results for distributed sensor network instances (Figure 14a) show that ABT_DO_Retro(mindom) is the algorithm requiring the largest number of exchanged bytes to solve these problems followed by ABT. All AgileABT($[\alpha], \prec_\alpha$) heuristics require fewer exchanged bytes than all other algorithms. They show an improvement factor of almost 3 over ABT_DO_Retro(mindom) and of almost 2.5 over ABT. The improvement factor is almost 2 over ABT_DO. For sparse binary random DisCSPs (Figure 14b), the obtained results show that ABT_DO-ng and ABT_DO_Retro(mindom) are the algorithm that require the largest amount of data. Except for AgileABT($[dom/wdeg], <$) that shows a similar performance compared to ABT, all other AgileABT($[\alpha], \prec_\alpha$) heuristics outperform all other algorithms by a large scale. They show an improvement factor of almost 3 over ABT and an almost order of magnitude improvement over ABT_DO-ng and ABT_DO_Retro. For dense binary random DisCSPs (Figure 14c), ABT_DO_Retro(mindom) is again the algorithm requiring the largest amount of exchanged bytes. ABT_DO-ng shows better performance compared to sparse instances and it outperforms AgileABT($[dom/wdeg], <$). In these instances, ABT and AgileABT($[\alpha], \prec_\alpha$) with neighbourhood based heuristics are the algorithms that require the minimum amount of exchanged bytes with a slight improvement for ABT over AgileABT($[\alpha], \prec_\alpha$).

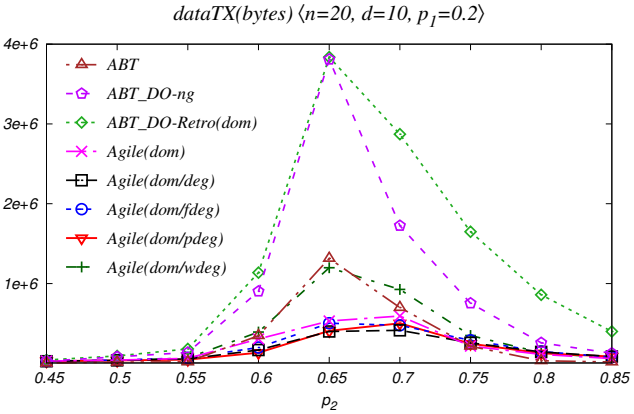
6.6. Discussion

Looking at all results together, we come to the straightforward conclusion that AgileABT($[\alpha], \prec_\alpha$) with neighbourhood-based heuristics, namely dom/deg, $dom/fdeg$ and $dom/pdeg$ perform very well compared to other techniques. We think that neighbourhood-based heuristics perform well thanks to their ability to take into account the structure of the problem [14]. Distinctly, among these three heuristics $dom/pdeg$ seems to be the best one.

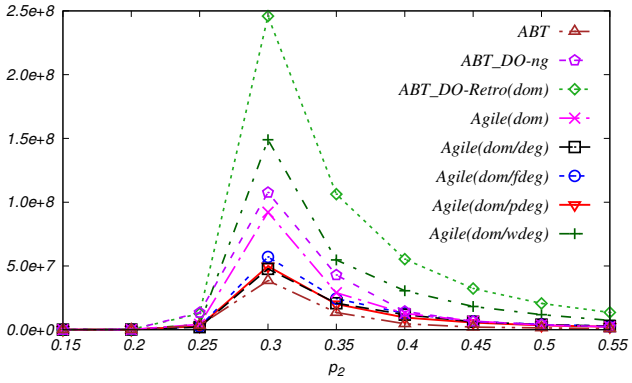
In distributed dynamic ordering algorithms, each change on the agents ordering invokes a series of coordination messages, and affects the search effort by wasting all incoherent no-goods computed so far. Therefore, too many changes of agents ordering will have a negative impact on the communication load and on the computational effort. In all our experiments, we counted



(a) The total number of bytes exchanged on SensorDisCSP instances.



(b) The total number of bytes exchanged on sparse random DisCSP instances.



(c) The total number of bytes exchanged on dense random DisCSP instances.

Figure 14: The total number of bytes exchanges to solve random DisCSP and SensorDisCSP instances.

Table 3: The number of **order** messages exchanged by each algorithm

Benchmark	Composed Random: 25-1-25	Composed Random: 25-1-40	Graph Coloring: $\langle 15, 5, .65\rangle$	Graph Coloring: $\langle 25, 5, .45\rangle$	Sensor DisCSP: $\langle 25, 6, .15, .3\rangle$	Randon DisCSP: $\langle 20, 10, .2, .65\rangle$	Randon DisCSP: $\langle 20, 10, .7, .3\rangle$
ABT_DO	84,709	842,538	24,962	708,482	2,710	19,331	392,376
ABT_DO_Retro(mindom)	30,357	35,550	27,450	423,198	5,470	24,795	1,559,112
AgileABT($[dom], <$)	5,095	5,182	7,935	20,315	1,022	2,206	171,374
AgileABT($[dom/deg], <$)	6,170	10,101	7,447	17,260	968	1,994	155,031
AgileABT($[dom/fdeg], <$)	10,175	10,428	10,171	30,746	837	2,622	174,801
AgileABT($[dom/pdeg], <$)	4,600	4,813	5,081	16,925	688	1,496	173,826
AgileABT($[dom/wdeg], <$)	6,591	7,104	13,285	67,197	1,265	5,187	240,000

the number of **order** messages exchanged by each distributed dynamic ordering algorithm. The results (presented in Table 3) show that ABT_DO-ng and ABT_DO_Retro(mindom) exchange a large number of **order** messages compared to AgileABT($[\alpha], \prec_\alpha$). Among AgileABT($[\alpha], \prec_\alpha$) algorithms, AgileABT($[dom/pdeg], <$) is the algorithm that exchanges the smallest number of **order** messages while AgileABT($[dom/wdeg], <$) is the one that exchanges the largest number of **order** messages. These empirical results suggest that to be efficient a distributed dynamic ordering algorithm needs to be able to revise the ordering of all agents. However, the fewer times the ordering is changed, the better the performance.

On the other hand, AgileABT($[\alpha], \prec_\alpha$) with the conflict-directed variable ordering heuristic $dom/wdeg$ shows a relatively poor performance on some instances. This fact can be explained by the limited amount of constraint propagation performed by DisCSP algorithms. Furthermore, asynchrony affects reception and treatment of **ok?** and **ngd** messages and has a direct impact on the computation of weights and new orders for the $dom/wdeg$ heuristic.

7. Conclusion

We proposed agile asynchronous backtracking (AgileABT), a distributed asynchronous constraint satisfaction framework which allows total reordering of agents during search without requiring exponential space. This is done via the original notion of termination values, labels attached to the orders exchanged by agents during search. Agents accept or reject the suggested reordering using a priority relation over termination values that can be simple scalar values, or could be more complex structures based on the intrinsic properties of the proposed reordering. We proved that AgileABT is guaranteed to terminate if the priority relation over the termination values is a well ordering and has a polynomial space complexity when the computation of the termination values has polynomial space complexity.

We then proposed AgileABT($[\alpha], \prec_\alpha$) an instance of AgileABT where arbitrary dynamic variable ordering heuristics defined by an order relation over

a measure, α, applied to each variable in the problem can be implemented using termination values in the form of vector of measures α. We proved that AgileABT($[\alpha], \prec_\alpha$) is guaranteed to terminate if the order relation over the measure α is a well ordering. Thanks to this original concept of termination values, any agent is now able to propose any other conflicting agent as a target to backtrack to, and can propose a reordering of all other agents, including those appearing before that backtrack target. These interesting features are totally new for a DisCSP algorithm with polynomial space complexity. AgileABT($[\alpha], \prec_\alpha$) allowed us to implement for the first time in DisCSPs a wide variety of the DVOs studied in centralized CSP research. Our experiments confirm the significance of these DVOs on a distributed setting. These experiments showed that AgileABT($[\alpha], \prec_\alpha$) offers orders of magnitude improvement in both computation and messaging costs compared to the original static ABT, and consistently outperforms previous proposals for dynamic ordering in ABT.

8. Acknowledgements

This material is based in part on work supported by Science Foundation Ireland under Grant No. 12/RC/2289 P2 which is co-funded under the European Regional Development Fund.

References

1. Junges R, Bazzan ALC. Evaluating the Performance of DCOP Algorithms in a Real World, Dynamic Problem. In: *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*. AAMAS'08; Richland, SC; 2008:599–606.

2. Ottens B, Faltings B. Coordination Agent Plans Trough Distributed Constraint Optimization. In: *Proceedings of the Multi Agent Planning Workshop*. MASPLAN'08; Sydney Australia; 2008:.

3. Jung H, Tambe M, Kulkarni S. Argumentation As Distributed Constraint Satisfaction: Applications and Results. In: *Proceedings of the Fifth International Conference on Autonomous Agents*. AGENTS'01; New York, NY, USA: ACM; 2001:324–31.

4. Béjar R, Domshlak C, Fernández C, Gomes C, Krishnamachari B, Selman B, Valls M. Sensor Networks and Distributed CSP: Communication, Computation and Complexity. *Artificial Intelligence* 2005;161(1-2):117–47.

5. Modi PJ, Shen WM, Tambe M, Yokoo M. Adopt: Asynchronous Distributed Constraint Optimization with Quality Guarantees. *Artificial Intelligence* 2005;161(1-2):149–80.

6. Maheswaran RT, Tambe M, Bowring E, Pearce JP, Varakantham P. Taking DCOP to the Real World: Efficient Complete Solutions for Distributed Multi-Event Scheduling. In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*. AAMAS'04; Washington, DC, USA: IEEE Computer Society; 2004:310–7.

7. Bonnet-Torrés O, Tessier C. Multiply-Constrained DCOP for Distributed Planning and Scheduling. In: *AAAI Spring Symposium: Distributed Plan and Schedule Management*. AAAI; 2006:17–24.

8. Kaplansky E, Meisels A. Distributed Personnel Scheduling–Negotiation Among Scheduling Agents. *Annals of Operations Research* 2007;155(1):227–55.

9. Petcu A, Faltings B. A Value Ordering Heuristic for Distributed Resource Allocation. In: *Proceedings of Joint Annual Workshop of ERCIM/CoLogNet on CSCLP'04*. 2004:86–97.

10. Léauté T, Faltings B. Coordinating Logistics Operations with Privacy Guarantees. In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume 3*. IJCAI'11; AAAI Press; 2011:2482–7.

11. Yokoo M, Durfee EH, Ishida T, Kuwabara K. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans on Knowledge and Data Engineering* 1998;10:673–85.

12. Bessière C, Maestre A, Brito I, Meseguer P. Asynchronous Backtracking Without Adding Links: A New Member in the ABT Family. *Artificial Intelligence* 2005;161(1-2):7–24.

13. Haralick RM, Elliott GL. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* 1980;14(3):263–313.

14. Bessière C, Régin J. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In: *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*. CP'96; Cambridge, Massachusetts, USA; 1996:61–75.

15. Boussemart F, Hemery F, Lecoutre C, Sais L. Boosting Systematic Search by Weighting Constraints. In: *Proceedings of the 16th European Conference on Artificial Intelligence*. ECAI'04; Amsterdam, The Netherlands, The Netherlands: IOS Press; 2004:146–50.

16. Silaghi MC, Sam-Haroud D, Faltings B. Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Tech. Rep. LIA-REPORT-2001-008; EPFL; 2001.

17. Zivan R, Meisels A. Dynamic Ordering for Asynchronous Backtracking on DisCSPs. *Constraints* 2006;11(2-3):179–97.

18. Brito I, Meseguer P. Synchronous, Asynchronous and Hybrid Algorithms for DisCSP. In: *Proceedings of the 5th International Workshop on Distributed Constraint Reasoning*. DCR'04; 2004:80–94.

19. Silaghi MC. Framework for Modeling Reordering Heuristics for Asynchronous Backtracking. In: *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*. IAT'06; Washington, DC, USA: IEEE Computer Society; 2006:529–36.

20. Zivan R, Zazone M, Meisels A. Min-Domain Retroactive Ordering for Asynchronous Backtracking. *Constraints* 2009;14(2):177–98.

21. Mechqrane Y, Wahbi M, Bessiere C, Bouyakhf EH, Meisels A, Zivan R. Corrigendum to "Min-Domain Retroactive Ordering for Asynchronous Backtracking". *Constraints* 2012;17:348–55.

22. Yokoo M. Asynchronous Weak-commitment Search for solving Distributed Constraint Satisfaction Problems. In: *Proceedings of the 1st International Conference on Principles and Practice of Constraint Programming*. CP'95; Cassis, France; 1995:88–102.

23. Hirayama K, Yokoo M. An Approach to Over-constrained Distributed Constraint Satisfaction Problems: Distributed Hierarchical Constraint Satisfaction. In: *Proceedings of the 4th International Conference on Multi-Agent Systems*. ICMAS'00; 2000:135–42.

24. Dechter R. Constraint Networks (Survey). *In Encyclopedia of Artificial Intelligence, 2nd edition* 1992;1:276–85.

25. Ginsberg ML, McAllester DA. GSAT and Dynamic Backtracking. In: *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*. KR'94; Bonn, Germany: Morgan Kaufmann Publishers Inc.; 1994:226–37.

26. Yokoo M, Durfee EH, Ishida T, Kuwabara K. Distributed constraint satisfaction for formalizing distributed problem solving. In: *Proceedings of the 12th International Conference on Distributed Computing Systems*. 1992:614–21.

27. Hirayama K, Yokoo M. The Effect of Nogood Learning in Distributed Constraint Satisfaction. In: *Proceedings of ICDCS'00*. 2000:169–77.

28. Yokoo M. Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-Agent Systems. London, UK: Springer-Verlag; 2000.

29. Colburn TR, Rankin TL, Fetzer JH, eds. Program Verification: Fundamental Issues in Computer Science. Norwell, MA, USA: Kluwer Academic Publishers; 1993. ISBN 0792319656.

30. Lecoutre C, Boussemart F, Hemery F. Backjump-Based Techniques versus Conflict-Directed Heuristics. In: *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*. ICTAI'04; 2004:549–57.

31. Smith BM, Grant SA. Trying Harder to Fail First. In: *Proceedings of 13th European Conference on Artificial Intelligence*. ECAI'98; Brighton, UK: John Wiley and Sons; 1998:249–53.

32. Eppstein D. Lexicographically Minimal Topological Sort of a Labeled DAG. Theoretical Computer Science Stack Exchange; 2015. URL: http://cstheory.stackexchange.com/q/31993.

33. Wahbi M, Ezzahir R, Bessiere C, Bouyakhf EH. Dischoco 2: A platform for distributed constraint reasoning. In: *Proceedings of the 13th International Workshop on Distributed Constraint Reasoning*. DCR'11; Barcelona, Catalonia, Spain; 2011:112–21. URL: http://dischoco.sourceforge.net.

34. Lynch NA. Distributed Algorithms. Morgan Kaufmann Series; 1997.

35. Meisels A, Razgon I, Kaplansky E, Zivan R. Comparing Performance of Distributed Constraints Processing Algorithms. In: *Proceedings of the 3rd International Workshop on Distributed Constraint Reasoning*. DCR'02; 2002:86–93.

36. Zivan R, Meisels A. Message delay and DisCSP search algorithms. *Annals of Mathematics and Artificial Intelligence* 2006;46(4):415–39.

37. Roussel O, Lecoutre C. XML Representation of Constraint Networks: Format XCSP 2.1. *CoRR* 2009;abs/0902.2362. arXiv:0902.2362.

38. Wahbi M. CSPLib problem 072: Target tracking in distributed sensor network. http://www.csplib.org/Problems/prob072; 2015.