A General Framework for Reordering Agents Asynchronously in Distributed CSP

Mohamed Wahbi, Younes Mechqrane, Christian Bessiere, Kenneth N. Brown

A General Framework for Reordering Agents Asynchronously in Distributed CSP

Mohamed Wahbi[1], Younes Mechqrane[2], Christian Bessiere[3], Kenneth N. Brown[1]
{mohamed.wahbi,ken.brown}@insight-centre.org,
ymechqrane@gmail.com, bessiere@lirmm.fr

[1] Insight Centre for Data Analytics, University College Cork, Ireland
[2] Mohammed V University-Agdal Rabat, Morocco
[3] University of Montpellier, France

Abstract. Reordering agents during search is an essential component of the efficiency of solving a distributed constraint satisfaction problem. Termination values have been recently proposed as a way to simulate the min-domain dynamic variable ordering heuristic. The use of termination values allows the greatest flexibility in reordering agents dynamically while keeping polynomial space. In this paper, we propose a general framework based on termination values for reordering agents asynchronously. The termination values are generalized to represent various heuristics other than min-domain. Our general framework is sound, complete, terminates and has a polynomial space complexity. We implemented several variable ordering heuristics that are well-known in centralized CSPs but could not until now be applied to the distributed setting. Our empirical study shows the significance of our framework compared to state-of-the-art asynchronous dynamic ordering algorithms for solving distributed CSP.

1 Introduction

Distributed artificial intelligence involves numerous combinatorial problems where multiple entities, called agents, need to cooperate in order to find a consistent combination of actions. Agents have to achieve the combination in a distributed manner and without any centralization. Examples of such problems are: traffic light synchronization [12], truck task coordination [18], target tracking in distributed sensor networks [11], distributed scheduling [16], distributed planning [6], distributed resource allocation [19], distributed vehicle routing [13], etc. These problems were successfully formalized using the distributed constraint satisfaction problem (DisCSP) paradigm.

DisCSP are composed of multiple agents, each owning its local constraint network. Variables in different agents are connected by constraints. Agents must assign values to their variables distributively so that all constraints are satisfied. To achieve this goal, agents assign values to their variables that satisfy their own constraints, and exchange messages to satisfy constraints with variables

owned by other agents. During the last two decades, many algorithms have been designed for solving DisCSP. Most of these algorithms assume a total (priority) order on agents that is static. However, it is known from centralized CSPs that reordering variables dynamically during search improves the efficiency of the search procedure. Moreover, reordering agents in DisCSP may be required in various applications (e.g., security [24]).

Agile Asynchronous Backtracking (AgileABT) [1] is a polynomial space asynchronous algorithm that is able to reorder *all* agents in the problem. AgileABT is based on the notion of *termination value*, a tuple of agents' domain sizes. Besides implementing the min-domain ([10]) dynamic variable ordering heuristic (DVO), the termination value acts as a timestamp for the orders exchanged by agents during search. Since the termination of AgileABT depends on the form of the termination value, the use of other forms of termination values (implementing other heuristics) may directly affect the termination of the algorithm.

In this paper, we generalize AgileABT to get a new framework AgileABT(α), in which α represents any measure used to implement a DVO (e.g., domain size in min-domain). AgileABT(α) is sound and complete. We define a simple condition on the measure α which guarantees that AgileABT(α) terminates. If the computation of the measure α also has polynomial space complexity, then AgileABT(α) has polynomial space complexity. This allows us to implement for the first time in Distributed CSP a wide variety of DVOs that have been studied in centralized CSP. To illustrate this, we implement a number of DVOs, including dom/deg [3] and dom/wdeg [14], and evaluate their performance on benchmark DisCSP problems.

The paper is organized as follows. Section 2 gives the necessary background on distributed CSP and dynamic reordering materials. It then discusses the Agile Asynchronous Backtracking algorithm for solving distributed CSP. Our general framework is presented and analyzed in Section 3. We show our empirical results in Section 4 and report related work in Section 5. Finally, we conclude in Section 6.

2 Background

2.1 Distributed Constraint Satisfaction Problem

The Distributed Constraint Satisfaction Problem (DisCSP) is a 5-tuple $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \varphi)$, where $\mathcal{A}$ is a set of agents $\{A_1, \ldots, A_p\}$, $\mathcal{X}$ is a set of variables $\{x_1, \ldots, x_n\}$, $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of domains, where D_i is the initial set of possible values which may be assigned to variable x_i, $\mathcal{C}$ is a set of constraints, and $\varphi : \mathcal{X} \rightarrow \mathcal{A}$ is a function specifying an agent to control each variable. During a solution process, only the agent which controls a variable can assign it a value. A constraint $C(X) \in \mathcal{C}$, on the ordered subset of variables $X = (x_{j_1}, \ldots, x_{j_k})$, is $C(X) \subseteq D_{j_1} \times \cdots \times D_{j_k}$, and specifies the tuples of values which may be assigned simultaneously to the variables in X. For this paper, we restrict attention to binary constraints. We denote by $\mathcal{C}_i \subseteq \mathcal{C}$ all constraints that involve

x_i. A *solution* is an assignment to each variable of a value from its domain, satisfying all constraints. Each agent A_i only knows constraints relevant to its variables ($\mathcal{C}_i$) and the existence of other variables involved in these constraints (its *neighbors*). Without loss of generality, we assume each agent controls exactly one variable ($p = n$), so we use the terms agent and variable interchangeably and do not distinguish between A_i and x_i.

Each agent A_i stores a unique order, an ordered tuple of agents IDs, denoted by λ_i. λ_i is called the current order of A_i. Agents appearing before A_i in λ_i are the higher priority agents (predecessors) denoted by λ_i^- and conversely the lower priority agents (successors) λ_i^+ are agents appearing after A_i in λ_i. We denote by $\lambda_i[k]$ ($\forall k \in 1..n$) the ID of the agent located at position k in λ_i.

2.2 Asynchronous Backtracking - ABT

The first complete asynchronous search algorithm for solving DisCSP is *Asynchronous Backtracking* (ABT) [27,2]. In ABT, agents act concurrently and asynchronously, and do not have to wait for decisions of others. However, ABT requires a total priority order among agents. Each agent tries to find an assignment satisfying the constraints with what is currently known from higher priority agents. When an agent assigns a value to its variable, it sends out messages to lower priority agents, with whom it is constrained, informing them about its assignment. When no value is possible for a variable, the inconsistency is reported to higher agents in the form of a no-good (an unfruitful value combination). ABT computes a solution (or detects that no solution exists) in a finite time. However, the priority order of agents is static and uniform across the agents.

2.3 No-goods and Explanations

During a solution process, agents can infer inconsistent sets of assignments, called no-goods. No-goods are used to justify value removals. A no-good ruling out value v_i from the domain of a variable x_i is a clause of the form $x_j = v_j \wedge \ldots \wedge x_k = v_k \rightarrow x_i \neq v_i$. This no-good, which means that the assignment $x_i = v_i$ is inconsistent with the assignments $x_j = v_j \wedge \ldots \wedge x_k = v_k$, is used by agent A_i to justify removal of v_i from its domain D_i. The *left hand side* (lhs) and the *right hand side* (rhs) of a no-good are defined from the position of $\rightarrow$. 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 [9]. 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., $x_j, x_k \in \lambda_i^-$). We say that a no-good is *compatible* with an order λ_i if all agents in its lhs appear before its rhs in λ_i.

The current domain of a variable x_i, maintained by A_i, is composed by values not ruled out by a no-good.[4] The initial domain size (before search starts) of A_i

[4] To stay polynomial, A_i keeps only one no-good per removed value.

is denoted by d_i^0 while its current domain size is denoted by d_i. Let Σ_i be the conjunction of the left hand sides of all no-goods ruling out values from D_i. We explain the current domain size of D_i by the following expression $e_i : \Sigma_i \rightarrow d_i$, called explanation of x_i (e_i). Every explanation e_i induces safety conditions: $\{\forall x_m \in \Sigma_i,\ x_m \prec x_i\}$. When all values of a variable x_i are ruled out by some no-goods ($\Sigma_i \rightarrow 0$), these no-goods are resolved, producing a new no-good from Σ_i. There are clearly many different ways of representing Σ_i as a directed no-good (an implication). In standard backtracking search algorithms (like ABT), the variable, say x_t, that has the lowest priority in the current order (among variables in Σ_i) must change its value. x_t is called the backtracking target and the directed no-good is $ng_t : \Sigma_i \setminus x_t \rightarrow x_t \neq v_t$. In AgileABT the backtracking target is not necessarily the variable with the lowest priority within the conflicting variables in the current order.

2.4 Agile Asynchronous Backtracking

In AgileABT, an order λ is always associated with a termination value τ. A termination value is a tuple of positive integers (representing the sizes of the domains of other agents seen from A_i). When comparing two orders the strongest order is that associated with the lexicographically smallest termination value. The lexicographic order on agents IDs ($<_{lex}$) is used to break ties, the smallest being the strongest.

In AgileABT, all agents start with the same order. Then, every agent A_i is allowed to change the order asynchronously. In the following we describe AgileABT by illustrating the computation performed within agent A_i. A_i can change its current order λ_i only if it receives a stronger one from another agent or if itself proposes a new order (λ_i') stronger than its current order λ_i. A_i can only propose new orders (λ_i') when it tries to backtrack after detecting a dead-end ($\Sigma_i \rightarrow 0$).

In AgileABT, agents exchange the following types of messages to coordinate the search (where A_i is the sender):

- ***ok?*** message is sent by A_i to all lower agents (λ_i^+) to ask whether its assignment is acceptable. Besides the assignment, the ***ok?*** message contains an explanation e_i which communicates the current domain size of x_i, the current order λ_i, and the current termination value τ_i stored by A_i.
- ***ngd*** message is sent by A_i when all its values are ruled out by Σ_i. This message contains a directed no-good, as well as λ_i and τ_i.
- ***order*** message is sent to propose a new order. This message includes the order λ_i proposed by A_i accompanied by the termination value τ_i.

Each agent needs to compute the size of the domain of other variables to build its termination value. Hence, each agent A_i stores a set E_i of explanations sent by other agents. During search, A_i updates E_i to store new received explanations and to remove those that are no more relevant to the search state or not compatible with its current order λ_i. If $e_k \in E_i$, A_i uses this explanation to justify the size $dom(k)$ of the current domain of x_k, i.e., d_k. Otherwise, A_i assumes that the size of the current domain of x_k is equal to d_k^0.

Algorithm 1: Computing termination value using heuristic α.

function $TV^{\alpha}(\lambda)$

1. τ is an array of length n;
2. **for** ($j \leftarrow 1$ **to** n) **do** $\tau[j] \leftarrow \alpha(\lambda[j])$;
3. **return** τ;

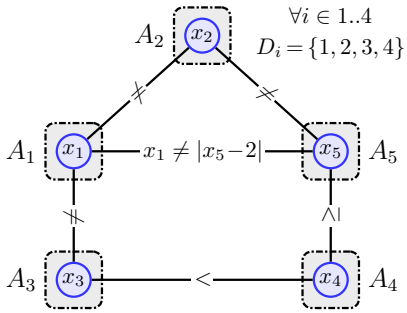
In AgileABT, the termination value $\tau_i = [tv_i^1, \ldots, tv_i^n]$ computed by agent A_i is such that $tv_i^k = dom(\lambda_i[k]), \forall k \in 1..n$. τ_i depends on the order λ_i and the domain sizes of agents given by the set of explanations E_i (Algorithm 1, using $TV^{\alpha}(\lambda_i)$ with $\alpha = dom$).

In standard backtracking search algorithms, the backtracking target is always the variable that has the lowest priority among the variables in the detected conflict (i.e., Σ_i). AgileABT relaxes this restriction by allowing A_i to select the target of backtracking x_t among conflicting variables Σ_i. The only restriction for selecting x_t as a backtracking target is to find an order λ'_i such that $\tau'_i = TV^{\alpha}(\lambda'_i)$ with $\alpha = dom$ (Algorithm 1) is lexicographically smaller than the termination value associated with the current order λ_i and x_t is the lowest among variables in Σ_i w.r.t. λ'_i.

When a dead-end occurs, AgileABT iterates through all variables $x_t \in \Sigma_i$, considering x_t as the target of the backtracking, i.e., the directed no-good is ng_t: $\Sigma_i \setminus x_t \rightarrow x_t \neq v_t$. A_i then updates E_i to remove all explanations containing x_t (after backtracking x_t assignment will be changed). Next, it updates the explanation of x_t by considering the new generated no-good ng_t (i.e., $e_t \leftarrow [\Sigma_t \cup lhs(ng_t) \rightarrow d_t - 1]$). Finally, A_i computes a new order (λ'_i) and its associated termination value (τ'_i) from the updated explanations E_i. λ'_i is obtained by performing a topological sort on the directed acyclic graph (G) formed by safety conditions induced by the updated explanations E_i ($\forall x_m \in \Sigma_k | e_k \in E_i,\ (x_m, x_k) \in G$) and τ'_i is obtained from $TV^{\alpha}(\lambda'_i)$ with $\alpha = dom$ (Algorithm 1). Let λ'_i be the strongest computed order over all possible targets in Σ_i. If the termination value τ'_i associated to λ'_i is lexicographically smaller than the τ_i associated to the current order λ_i, A_i reorders agents according to λ'_i and informs all agents about the new order λ'_i and its associated termination value τ'_i. The backtracking target is that used when A_i computed λ'_i. If no λ'_i stronger than λ_i exists, the backtracking target x_t is the variable that has the lowest priority among Σ_i in the current order λ_i. For more details we refer the reader to [1,25].

Example of running AgileABT

Figure 1 presents an example of a possible execution of AgileABT on a simple problem. This problem (fig. 1a) 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}: $x_4 \geq x_5$. All agents start with the same initial ordering $\lambda_i = [1, 2, 3, 4, 5]$ associated with



(a) The constraint graph

Data structures maintained by A_5

incoming messages	explanations (E_5)	no-goods ($D_5 = \emptyset$)
ok?$_{1\to5}$:$\langle e_1, x_1=1, \lambda_1, \tau_1\rangle$	e_1: [] $\to d_1=4$	$[x_1=1] \to x_5 \neq 1$
ok?$_{2\to5}$:$\langle e_2, x_2=2, \lambda_2, \tau_2\rangle$	e_2: $[x_1=1] \to d_2=3$	$[x_2=2] \to x_5 \neq 2$
ok?$_{3\to5}$:$\langle e_3, x_3=2, \lambda_3, \tau_3\rangle$	e_3: $[x_1=1] \to d_3=3$	$[x_1=1] \to x_5 \neq 3$
ok?$_{4\to5}$:$\langle e_4, x_4=3, \lambda_4, \tau_4\rangle$	e_4: $[x_3=2] \to d_4=2$	$[x_4=3] \to x_5 \neq 4$

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

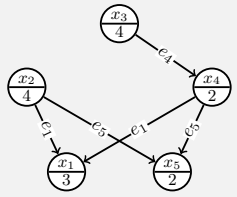
Σ_5: $x_1=1 \wedge x_2=2 \wedge x_4=3$

(b) Explanations and no-goods maintained by A_5

A_5: when choosing x_1 as backtracking target for Σ_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



Directed graph

(c) A_5: updated explanations with x_1 as target

A_5: target selection

	$\lambda_5 = [1,2,3,4,5]$	$\tau_5 = [4,4,4,4,4]$
target	**agent ordering**	**termination value**
$x_1 \neq 1$	$\lambda'_5 = [3,4,2,5,1]$	$\tau'_5 = [4,2,4,2,3]$
$x_2 \neq 2$	$\lambda'_5 = [1,3,4,5,2]$	$\tau'_5 = [4,3,2,1,2]$
$x_4 \neq 3$	$\lambda'_5 = [1,2,5,3,4]$	$\tau'_5 = [4,3,1,3,1]$

Selected no-good: ng_1: $[x_4=3 \wedge x_2=2] \to x_1 \neq 1$

(d) A_5: selection of target (x_1)

Fig. 1: An example of a possible execution of AgileABT on a simple problem.

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 (fig. 1b). 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 (c_{15}, c_{25}, and c_{45}), A_5 detects a dead-end ($D_5=\emptyset$) where Σ_5: $\{x_1=1 \wedge x_2=2 \wedge x_4=3\}$. A_5 iterates through all variables $x_t \in \Sigma_5$, considering x_t as the target of the backtracking. Figure 1c 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., $x_t = x_1$). 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 ng_1 in the explanation of x_1, i.e., e_1 (fig. 1c, left). Finally, A_5 computes a new order (λ'_5) and its associated termination value (τ'_5) from the updated explanations E_5. λ'_5 is obtained by performing a topological sort on the directed acyclic graph formed by safety conditions induced by the updated explanations E_5 (fig. 1c, right). Figure 1d presents the computed orderings and their associated termination values (by topological sort) when considering each $x_t \in \Sigma_5$ as backtracking target. The strongest computed order (e.g, $\lambda'_5 = [3,4,2,5,1]$, $\tau'_5 = [4,2,4,2,3]$) is that computed when considering x_1 as backtracking target. Since λ'_5 is stronger than λ_5, A_5 changes its current order to λ'_5 and proposes this ordering to all other agents through ***order*** messages (i.e., ***order***:$\langle \lambda'_5, \tau'_5 \rangle$). Then, A_5 sends the no-good ng_1 to agent x_1.

Algorithm 2: AgileABT(α): Procedures for changing the order by A_i.

procedure `processOrder`(λ_j, τ_j)

1. **if (** $\tau_j \prec_{tv} \tau_i \vee (\tau_j \overset{tv}{=} \tau_i \wedge \lambda_j <_{lex} \lambda_i)$ **) then** `changeOrder`(λ_j, τ_j) ;

procedure `proposeOrder`$(args)$

2. $\langle \lambda'_i, \tau'_i \rangle \leftarrow$ `computeNewOrder`$(args)$;
3. **if (** $\tau'_i \prec_{tv} \tau_i$ **) then**
4. `changeOrder`(λ'_i, τ'_i) ;
5. `sendMsg`: ***order***$\langle \lambda_i, \tau_i \rangle$ **to** all agents in $\mathcal{A}$;

procedure `changeOrder`(λ', τ')

6. $\lambda_i \leftarrow \lambda'$; $\tau_i \leftarrow \tau'$;
7. remove no-goods and explanations incompatible with τ_i ;

It has been proved that AgileABT is sound, complete and terminates [1,25]. The termination proofs of AgileABT are based on the fact that the termination value is a tuple of positive integers (representing the expected sizes of the domains of other agents) and, as search progresses, these tuples can only decrease lexicographically. Thus, any change to the form of the termination values (i.e. implemented heuristic) may directly affect the termination of AgileABT.

3 Generalized AgileABT

In AgileABT, the termination value can be seen as an implementation of the *dom* dynamic variable ordering heuristic. In this section, we generalize AgileABT to get a new framework AgileABT(α), in which α represents any measure used to implement a DVO. The original AgileABT [1] is then equivalent to AgileABT(*dom*).

Due to space constraints, we only present in Algorithm 2 the pseudo-code of AgileABT(α) related to the cases where an agent (A_i) may change its current order λ_i (i.e., calling procedure `changeOrder`) where $\prec_{tv}$ is an ordering on the termination values and $\overset{tv}{=}$ represents the equality. A_i can change its current order λ_i and its associated termination value τ_i (procedure `changeOrder`, line 6) in two cases. The first case is when A_i receives a stronger order λ_j associated with the termination value τ_j from another agent A_j (`processOrder`, line 1). The second case occurs when A_i itself proposes a new order (λ'_i) associated with a termination value τ'_i that is preferred (w.r.t. $\prec_{tv}$) to the termination value τ_i associated to its current order λ_i (procedure `proposeOrder`, lines 3 and 4).

The soundness, completeness and polynomial space complexity[5] of AgileABT(α) are directly inherited from original AgileABT, i.e., AgileABT(*dom*). The only property that could be jeopardized is the termination of the algorithm. In the following we define a sufficient condition on

[5] If the computation of the measure α also has polynomial space complexity.

termination values and the ordering $\prec_{tv}$ which guarantees that AgileABT(α) (Algorithm 2) terminates. Next, we will discuss a condition on the measure α that allows the termination values to obey the required condition.

Condition 1 *The priority ordering $\prec_{tv}$ is a well-ordering on the range of function TV^{α}.*

Proposition 1. *AgileABT(α) terminates if $\prec_{tv}$ obeys condition 1.*

Proof. Following the pseudo-code of AgileABT(α), an agent A_i can only change its current order in two cases (lines 1 and 3). The termination values can only decrease w.r.t. the well-ordering $\prec_{tv}$, or remain the same and have a lexicographically decreasing agent order (line 1). The agent order cannot decrease lexicographically indefinitely and by condition 1 the termination values cannot decrease indefinitely w.r.t. $\prec_{tv}$. Therefore, AgileABT(α) cannot change the order indefinitely. Once the order stops changing, all agents will eventually have the same termination value and the same order to which it is attached (line 5, Algorithm 2). This order corresponds to the strongest order computed in the system so far. Since the agent order is now common and static, AgileABT(α) will behave exactly like ABT, which terminates. □

To guarantee that AgileABT(α) terminates we need to define a well-ordering $\prec_{tv}$ on the termination values. Let α be the measure applied to the agents, and let the function TV^{α} be as defined in Algorithm 1. Let $\mathcal{S}$ be the range of α and let $\prec_{\alpha}$ be a total preference order on $\mathcal{S}$.

Definition 1 *Let λ_i and λ_j be two total agent orderings, $\tau_i = TV^{\alpha}(\lambda_i)$ and $\tau_j = TV^{\alpha}(\lambda_j)$. The termination value τ_i is preferred (w.r.t. $\prec_{tv}$) to τ_j (i.e., $\tau_i \prec_{tv} \tau_j$) if and only if τ_i is lexicographically less than τ_j (w.r.t. $\prec_{\alpha}$). In other words, $\tau_i \prec_{tv} \tau_j$ iff $\exists\, k \in 1..n$ such that $\alpha(\lambda_i[k]) \prec_{\alpha} \alpha(\lambda_j[k])$ and $\forall p \in 1..k-1$ $\alpha(\lambda_i[p]) = \alpha(\lambda_j[p])$.*

Condition 2 *The priority ordering $\prec_{\alpha}$ is a well-ordering on the range of measure α.*

Proposition 2. *AgileABT(α) terminates if $\prec_{\alpha}$ obeys condition 2.*

Proof. Suppose condition 2 is satisfied but AgileABT(α) does not terminate. Then, by proposition 1, condition 1 is not satisfied. Therefore, $\prec_{tv}$ is not a well-ordering on the range of TV^{α}. In other words, we can obtain an infinite decreasing sequence of termination values using a lexicographic comparison w.r.t. $\prec_{\alpha}$. Therefore we must have an infinite decreasing sequence of $\alpha(\lambda[k])$ values, for some $k \in 1..n$. But this contradicts condition 2 that forces $\prec_{\alpha}$ to be a well-ordering on $\mathcal{S}$. Therefore AgileABT(α) must terminate. □

In the following, we consider a number of different heuristics that are known to be effective in reducing search in centralized CSP, but which could not before now be applied to distributed CSP. We show how the measures that inform these heuristics can obey condition 2, and thus can be applied in the general AgileABT(α) framework.

Algorithm 3: Compute the $wdeg$ of agent A_i ($wdeg(i)$).

// Filtering $x_j \in X$ by propagating $C(X)$

function `revise`$(C(X), x_j)$

1. **foreach** ($v_j \in D_j$) **do**
2. **if** ($\neg hasSupport(C(X), v_j, x_j)$) **then** $D_j \leftarrow D_j \setminus v_j$;
3. **if** ($D_j = \emptyset$) **then** $weight[C] \leftarrow weight[C] + 1$;
4. **return** $D_j \neq \emptyset$;

procedure `computeWeight()`

5. $wdeg \leftarrow 1$;
6. **foreach** ($C(X) \in \mathcal{C}_i \mid nbUnassigned(X) > 1$) **do** $wdeg \leftarrow wdeg + weight[C]$;
7. $wdeg(i) \leftarrow \min(wdeg, W)$;

3.1 Neighborhood based variable ordering heuristics

In the first category we try to take into account the neighborhood of each agent. We implemented three DVOs (dom/deg, $dom/fdeg$ and $dom/pdeg$) to obtain respectively AgileABT(dom/deg), AgileABT($dom/fdeg$), and AgileABT($dom/pdeg$). In AgileABT(dom/deg), each agent A_i only requires to know the degree $deg(k)$ of each agent A_k in the problem. $deg(k)$ (i.e., the number of neighbors of A_k) can be obtained before the search starts as is the case for d_k^0 of each agent. Afterwards, A_i computes $\tau_i' = TV^{dom/deg}(\lambda_i')$ using $\alpha(k) = \frac{dom(k)}{deg(k)}$ (Algorithm 1, line 2). In AgileABT($dom/fdeg$) and AgileABT($dom/pdeg$) each agent A_i is required to know the set of neighbors of each agent A_k because it will need to compute the incoming degree $pdeg(k)$ and the outgoing degree $fdeg(k)$ of A_k for any proposed order. Again this information can be known in a preprocessing step before running AgileABT(α). Afterwards, A_i computes τ_i' from $TV^{dom/fdeg}(\lambda_i')$ (resp. $TV^{dom/pdeg}(\lambda_i')$) using $\alpha(k) = \frac{dom(k)}{fdeg(k)}$ (resp. $\alpha(k) = \frac{dom(k)}{pdeg(k)}$) where the incoming degree $pdeg(k)$ in λ_i' is the number of neighbors of A_k that appear before k in λ_i' and the outgoing degree $fdeg(k)$ in λ_i' is the number of neighbors of A_k that appear after k in λ_i'.

3.2 Conflict-directed variable ordering heuristic

The second category covers the conflict-directed variable ordering heuristic: $dom/wdeg$. In order to compute τ_i' using $\alpha(k) = \frac{dom(k)}{wdeg(k)}$, each agent A_i in AgileABT($dom/wdeg$) requires to know the weighted degree $wdeg(k)$ of each other agent A_k. A_i maintains its weighted degree, $wdeg(i)$, that it computes and the weighted degrees received from other agents, $wdeg(k)$. In order to compute the weighted degree, $wdeg$, (Algorithm 3) each agent A_i maintains a counter $weight[C]$ for each constraint $C(X)$ in $\mathcal{C}_i$. Whenever a domain (D_j where $x_j \in X$) is wiped-out while propagating $C(X)$ (line 3, Algorithm 3), $weight[C]$ is incremented. Before assigning its variable and sending an ***ok?*** message to lower priority agents, A_i computes its weighted degree, $wdeg(i)$, by summing up (line 6, Algorithm 3) the weights of all constraints in $\mathcal{C}_i$ having at least two

unassigned variables ([14]). However, to guarantee that AgileABT($dom/wdeg$) terminates we only update $wdeg(i)$ if the new computed weighted degree ($wdeg$) does not exceed a limit W on which all agents agree beforehand (line 7, Algorithm 3). In AgileABT($dom/wdeg$), whenever A_i sends an ***ok?*** message it attaches to this message the largest weighted degree computed so far $wdeg(i)$.

3.3 Theoretical Analysis

Lemma 1. *All measures α above are a well-ordering on a subset of $\mathbb{Q}$ w.r.t. $<$.*

Proof. We proceed by contradiction. Suppose there is an infinite decreasing sequence of values of $\alpha(k)$. In all measures above, $\alpha(k) = \frac{dom(k)}{\omega(k)}$, for some $\omega(k)$. $dom(k)$ is the expected domain size of the agent A_k. It is obvious that $dom(k)$ is a well-ordering on $\mathbb{N}$ w.r.t. $<$, and so cannot decrease indefinitely. Therefore, $\omega(k)$ must increase indefinitely. $\omega(k)$ is a positive integer whose value depends on the measure used. Two cases were explored in this paper. The first case concerns the family of degree-based heuristics (deg, $pdeg$, $fdeg$). In this case, all of the $\omega(k)$ are greater than or equal to 1 and smaller than the number of agents in the system (i.e., n) because an agent is at most constrained to $n-1$ other agents. Thus, $1 \leq \omega(k) \leq n-1$. The second case is related to the heuristic $wdeg$. We have outlined in section 3.2 that an agent is not allowed to increment its weight when it has reached the limit W set beforehand (Algorithm 3, line 7). Thus, $1 \leq \omega(k) \leq W$. In both cases $\omega(k)$ cannot increase indefinitely. Therefore for all measures presented above, of the form $\alpha(k) = \frac{dom(k)}{\omega(k)}$, cannot decrease indefinitely, and so $\alpha(k)$ is a well-ordering w.r.t. $<$. □

4 Empirical Analysis

In this section we experimentally compare AgileABT(α)[6] using different DVO heuristics to three other algorithms: ABT, ABT_DO with nogood-triggered heuristic (ABT_DO-ng) [29] and ABT_DO with min-domain retroactive heuristic (ABT_DO_Retro(mindom)) [31]. All experiments were performed on the DisChoco 2.0 platform [26],[7] in which agents are simulated by Java threads that communicate only through message passing.

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 ($\#msg$) [15]. Computation effort is measured by the number of non-concurrent constraint checks ($\#ncccs$) [28]. $\#ncccs$ is the metric used in distributed constraint solving to simulate computation time, but for dynamic reordering algorithms its variant generic $\#ncccs$ is used[30]. Algorithms are evaluated on three benchmarks: uniform binary random DisCSPs,

[6] For AgileABT($dom/wdeg$), we fixed $W = 1,000$. But, varying W made negligible difference to the results.

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

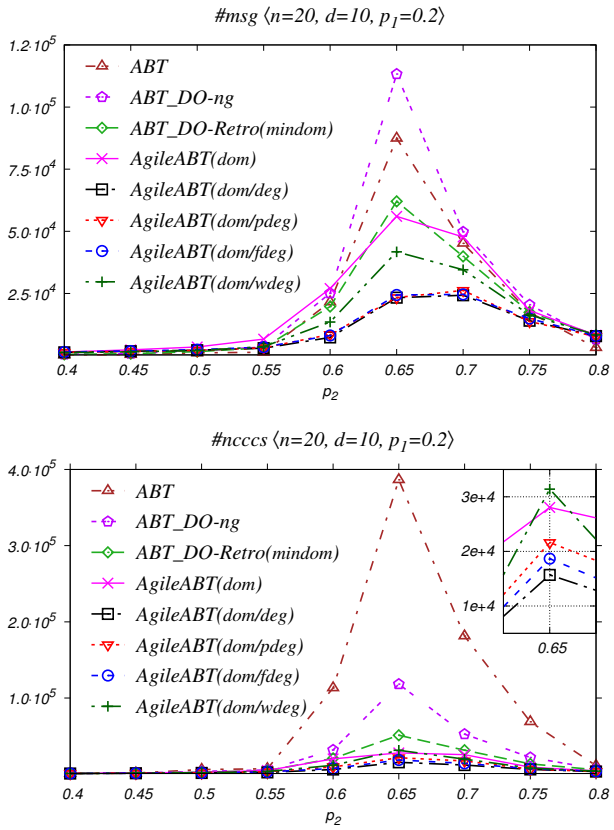


Fig. 2: Sparse uniform binary random DisCSPs

distributed graph coloring problems, and composed random instances. All binary table constraints in these problems are implemented using *AC-2001* [4].

4.1 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 execution.

Figures 2 and 3 show the results on sparse respectively dense uniform binary random DisCSPs. In sparse problems (Figure 2), AgileABT(α) outperforms all other algorithms on both *#msg* and *#ncccs*. ABT with AC-2001 is

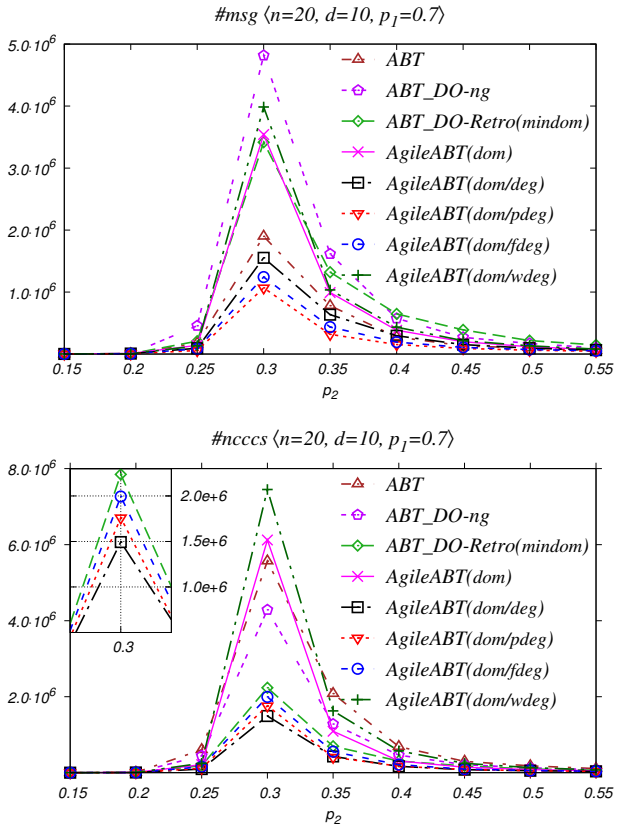


Fig. 3: Dense uniform binary random DisCSPs

significantly the slowest algorithm but it requires fewer messages than ABT_DO-ng. Regarding the speedup, AgileABT(α) shows almost an order of magnitude improvement compared to ABT and they all improve on ABT_DO-ng and ABT_DO_Retro(mindom). Comparing AgileABT(α) algorithms, neighborhood based heuristics (i.e., deg, $pdeg$ and $fdeg$) show an almost two-fold improvement over dom and $wdeg$ on $\#ncccs$. This improvement is even more significant on $\#msg$. $wdeg$ requires fewer messages than dom. In dense problems (Figure 3), neighborhood based heuristics outperform all other algorithms both on $\#msg$ and $\#ncccs$. ABT requires almost half the $\#msg$ of ABT_DO_Retro(mindom), AgileABT(dom), and AgileABT($dom/wdeg$). ABT_DO-ng is always the algorithm that requires more messages. AgileABT($dom/wdeg$) shows poor performance. It is slower and requires more messages than AgileABT(dom).

4.2 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

Table 1: Distributed graph coloring problems

Algorithm	$\langle \mathbf{15}, \mathbf{5}, \mathbf{0.65} \rangle$		$\langle \mathbf{25}, \mathbf{5}, \mathbf{0.45} \rangle$	
	$\#msg$	$\#ncccs$	$\#msg$	$\#ncccs$
AgileABT($dom/wdeg$)	90,630	188,991	2,600,016	2,783,132
AgileABT($dom/fdeg$)	51,820	104,517	940,481	937,861
AgileABT($dom/pdeg$)	47,949	89,514	**454,998**	**434,540**
AgileABT(dom/deg)	**44,083**	**78,050**	607,927	505,140
AgileABT(dom)	79,518	204,012	3,001,538	3,836,301
ABT_DO_Retro(mindom)	73,278	115,850	1,089,024	830,423
ABT_DO-ng	157,873	282,737	4,547,565	3,639,791
ABT	58,817	288,803	1,626,901	3,836,391

Table 2: Composed random instances

Instances	**25-1-25**		**25-1-40**	
Algorithm	$\#msg$	$\#ncccs$	$\#msg$	$\#ncccs$
AgileABT($dom/wdeg$)	85,521	30,064	89,804	33,461
AgileABT($dom/fdeg$)	146,668	219,980	1,337,552	2,830,906
AgileABT($dom/pdeg$)	**57,079**	**16,043**	**54,667**	**17,704**
AgileABT(dom/deg)	122,735	309,064	740,669	2,793,670
AgileABT(dom)	57,451	20,944	59,859	23,405
ABT_DO_Retro(mindom)	67,022	41,401	96,783	59,980
ABT_DO-ng	1,329,257	1,614,960	$> 10^8$	$> 10^9$
ABT	2,850,137	22,042,094	9,429,088	72,524,742

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(α) using neighborhood based DVO are by far the best algorithms for solving both classes. ABT_DO-ng shows poor performance on solving those problems. ABT_DO_Retro(mindom) outperforms AgileABT(dom) in both classes. Comparing AgileABT(dom) to AgileABT($dom/wdeg$), dom is slower than $wdeg$ but it requires fewer messages. In $\langle n = 15, d = 5, p_1 = 0.65 \rangle$, only AgileABT($\alpha$) using neighborhood based DVO outperform ABT on messages while other asynchronous dynamic ordering algorithms require more messages.

4.3 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 cen-

tralized CSP [20,7].[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 show that AgileABT($dom/pdeg$) outperforms all other algorithms in both classes. The second best algorithm for solving these instances is AgileABT(dom). ABT shows very poor performance on solving these problems followed by ABT_DO-ng that cannot solve instances in the second class (25-1-40) within the limits we fixed for all algorithms (10^8 $\#msg$ and 10^9 $\#ncccs$). Regarding AgileABT(α) DVOs, $wdeg$ seems to pay off on these instances compared to dom/deg and $dom/fdeg$. In 25-1-40, AgileABT(dom/deg) outperforms ABT_DO_Retro(mindom), but the opposite happens for 25-1-25.

4.4 Discussion

Looking at all results together, we come to the straightforward conclusion that AgileABT(α) with neighbourhood-based heuristics, namely deg, $fdeg$ and $pdeg$ 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 [3]. Distinctly, among these three heuristics $dom/pdeg$ seems to be the best one because of the limited changes on the agent at the first position. In $dom/pdeg$ $\tau[1] = dom(\lambda[1])$ because $pdeg(\lambda[1]) = 1$. Thus, the number of order changes (cost on messages) in AgileABT($dom/pdeg$) is reduced. Note that the strength of AgileABT(α) is that it enables any ordering to be identified and executed as the algorithm runs. However, each change invokes a series of coordination messages, and so too many changes of order will have a negative impact.

On the other hand the AgileABT(α) with the conflict-directed variable ordering heuristic, namely $wdeg$, shows a relatively poor performance. 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 some instances of the coloring problem, the performance of the conflict-directed heuristic varies significantly from one execution to another, indicating it is more sensitive to the asynchrony than the other heuristics.

5 Related Work

Bliek ([5]) proposed *Generalized Partial Order Dynamic Backtracking* (GPODB), an algorithm that generalizes both *Partial Order Dynamic Backtracking* (PODB) [9] and *Dynamic Backtracking* (DBT) [8]. This generalization

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

was obtained by relaxing the safety conditions of PODB which results in additional flexibility. AgileABT has some similarities with PODB and GPODB because AgileABT also maintains a set of safety conditions. However, both PODB and GPODB are subject to the same restriction: when a dead end occurs, the backtracking target of the generated no-good must be selected such that the safety conditions induced by the new no-good satisfy all existing safety conditions. By contrast, AgileABT overcomes this restriction by allowing the violation of existing safety conditions by relaxing some explanations.

Silaghi et al. proposed Asynchronous Backtracking with Reordering (ABTR) [21,22,23]. In ABTR, a reordering heuristic identical to the centralized dynamic backtracking [8] was applied to ABT. The authors in [29] proposed Dynamic Ordering for Asynchronous Backtracking (ABT_DO) [29]. When an ABT_DO agent assigns a value to its variable, it can reorder lower priority agents. In other words, an agent on a position j can change order of the agents on positions $j+1$ through n. The authors in [29] proposed three different ordering heuristics to reorder lower priority agents: random, min-domain and nogood-triggered inspired by dynamic backtracking [8]. Their experimental results show that nogood-triggered (ABT_DO-ng), where the generator of the no-good is placed just after the target of the backtrack, is best.

A new kind of ordering heuristics for ABT_DO is presented in [31] for reordering higher agents. These *retroactive* heuristics enable the generator of a no-good (backtrack) to be moved to a higher position than that of the target of the backtrack. The resulting algorithm is called ABT_DO_Retro. The degree of flexibility of these heuristics depends on the size of the no-good storage capacity K, which is predefined. Agents are limited to store no-goods with a size equal to or smaller than K. The space complexity of the agents is thus exponential in K. However, the best of those heuristics, ABT_DO_Retro(mindom) [31,17], does not need any no-good storage. In ABT_DO_Retro(mindom), the agent that generates a no-good is placed between the last and the second last agents in the no-good if its domain size is smaller than that of the agents it passes on the way up.

6 Conclusion

We proposed a general framework for reordering agents asynchronously in DisCSPs which can implement many different dynamic variable ordering heuristics. Our general framework is sound, complete, and has a polynomial space complexity. We proved that a simple condition on the measure used in the heuristics is enough to guarantee termination. We implemented several DVOs heuristics that are well-known in centralized CSP but were not able to be used before in the distributed CSP. Our empirical study shows the significance of these DVOs on a distributed setting. In particular, it highlights the good performance of neighborhood based heuristics. Future work will focus on designing new DVOs that can be used in AgileABT(α).

References

1. Bessiere, C., Bouyakhf, E.H., Mechqrane, Y., Wahbi, M.: Agile asynchronous backtracking for distributed constraint satisfaction problems. In: Proceedings of ICTAI'11. pp. 777–784. Boca Raton, Florida, USA (November 2011)
2. Bessiere, C., Maestre, A., Brito, I., Meseguer, P.: Asynchronous backtracking without adding links: a new member in the ABT family. Artif. Intel. 161, 7–24 (2005)
3. Bessiere, C., Régin, J.C.: MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In: Proceedings of CP'96. pp. 61–75 (1996)
4. Bessiere, C., Régin, J.C.: Refining the basic constraint propagation algorithm. In: Proceedings of IJCAI'01. pp. 309–315. San Francisco, CA, USA (2001)
5. Bliek, C.: Generalizing partial order and dynamic backtracking. In: Proceedings of AAAI'98/IAAI'98. pp. 319–325. Menlo Park, CA, USA (1998)
6. Bonnet-Torrés, O., Tessier, C.: Multiply-constrained dcop for distributed planning and scheduling. In: AAAI SSS: Distributed Plan and Schedule Management. pp. 17–24 (2006)
7. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of ECAI'04. pp. 146–150 (2004)
8. Ginsberg, M.L.: Dynamic Backtracking. JAIR 1, 25–46 (1993)
9. Ginsberg, M.L., McAllester, D.A.: GSAT and dynamic backtracking. In: KR. pp. 226–237 (1994)
10. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artif. Intel. 14(3), 263–313 (1980)
11. Jung, H., Tambe, M., Kulkarni, S.: Argumentation as Distributed Constraint Satisfaction: Applications and Results. In: Proceedings of AGENTS'01. pp. 324–331 (2001)
12. Junges, R., Bazzan, A.L.C.: Evaluating the performance of dcop algorithms in a real world, dynamic problem. In: Proceedings of AAMAS'08. pp. 599–606. Richland, SC (2008)
13. Léauté, T., Faltings, B.: Coordinating Logistics Operations with Privacy Guarantees. In: Proceedings of the IJCAI'11. pp. 2482–2487 (2011)
14. Lecoutre, C., Boussemart, F., Hemery, F.: Backjump-Based Techniques versus Conflict-Directed Heuristics. In: Proceedings of IEEE ICTAI'04. pp. 549–557 (2004)
15. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Series (1997)
16. Maheswaran, R.T., Tambe, M., Bowring, E., Pearce, J.P., Varakantham, P.: Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In: Proceedings of AAMAS'04 (2004)
17. Mechqrane, Y., Wahbi, M., Bessiere, C., Bouyakhf, E.H., Meisels, A., Zivan, R.: Corrigendum to "Min-Domain Retroactive Ordering for Asynchronous Backtracking". Constraints 17, 348–355 (2012)
18. Ottens, B., Faltings, B.: Coordination agent plans trough distributed constraint optimization. In: Proceedings of MASPLAN'08. Sydney Australia (2008)
19. Petcu, A., Faltings, B.: A Value Ordering Heuristic for Distributed Resource Allocation. In: Proceedings of Joint Annual Workshop of ERCIM/CoLogNet on CSCLP'04. pp. 86–97 (2004)
20. Roussel, O., Lecoutre, C.: Xml representation of constraint networks: Format XCSP 2.1. CoRR (2009)
21. Silaghi, M.C.: Framework for modeling reordering heuristics for asynchronous backtracking. In: Intelligent Agent Technology, 2006. IAT'06. IEEE/WIC/ACM International Conference on. pp. 529–536 (Dec 2006)

22. Silaghi, M.C.: Generalized Dynamic Ordering for Asynchronous Backtracking on DisCSPs. In: Proceedings of DCR'06 (2006)
23. Silaghi, M.C., Sam-Haroud, D., B.Faltings: ABT with Asynchronous Reordering. In: 2nd Asia-Pacific IAT (2001)
24. Silaghi, M.C., Sam-Haroud, D., Calisti, M., Faltings, B.: Generalized English Auctions by Relaxation in Dynamic Distributed CSPs with Private Constraints. In: Proceedings of DCR'01. pp. 45–54 (2001)
25. Wahbi, M.: Algorithms and Ordering Heuristics for Distributed Constraint Satisfaction Problems. John Wiley & Sons, Inc. (2013)
26. Wahbi, M., Ezzahir, R., Bessiere, C., Bouyakhf, E.H.: DisChoco 2: A Platform for Distributed Constraint Reasoning. In: Proceedings of workshop on DCR'11. pp. 112–121 (2011), http://dischoco.sourceforge.net/
27. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: Distributed constraint satisfaction for formalizing distributed problem solving. In: Proceedings of ICDCS. pp. 614–621 (1992)
28. Zivan, R., Meisels, A.: Parallel Backtrack search on DisCSPs. In: Proceedings of DCR'02 (2002)
29. Zivan, R., Meisels, A.: Dynamic Ordering for Asynchronous Backtracking on DisCSPs. Constraints 11(2-3), 179–197 (2006)
30. Zivan, R., Meisels, A.: Message delay and DisCSP search algorithms. AMAI 46(4), 415–439 (2006)
31. Zivan, R., Zazone, M., Meisels, A.: Min-Domain Retroactive Ordering for Asynchronous Backtracking. Constraints 14(2), 177–198 (2009)