

# Min-domain retroactive ordering for Asynchronous Backtracking

Roie Zivan · Moshe Zazone · Amnon Meisels

Published online: 7 May 2008  
© Springer Science + Business Media, LLC 2008

**Abstract** Ordering heuristics are a powerful tool in CSP search algorithms. Among the most successful ordering heuristics are heuristics which enforce a *fail first* strategy by using the Min-domain property (Haralick and Elliott, *Artif Intel* 14:263–313, 1980; Bessiere and Regin, *Mac and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems*. In *Proc. CP 96*, pp. 61–75, Cambridge, MA, 1996; Smith and Grant, *Trying harder to fail first*. In *European Conference on Artificial Intelligence*, pp. 249–253, 1998; Dechter, *Constraint Processing*. Morgan Kaufman, 2003). Ordering heuristics have been introduced recently to *asynchronous backtracking (ABT)*, for distributed constraints satisfaction (*DisCSP*) (Zivan and Meisels, *Dynamic ordering for asynchronous backtracking on discsps*. In *CP-2005*, pp. 32–46, Sigtes (Barcelona), Spain, 2005). However, the pioneering study of dynamically ordered ABT, *ABT\_DO*, has shown that a straightforward implementation of the Min-domain heuristic does not produce the expected improvement over a static ordering. The present paper proposes an asynchronous dynamic ordering which does not follow the standard restrictions on the position of reordered agents in *ABT\_DO*. Agents can be moved to a position that is higher than that of the target of the backtrack. Combining the Nogood-triggered heuristic and the Min-domain property in this new class of heuristics results in the best performing version of *ABT\_DO*.

---

Supported by the Lynn and William Frankel center for Computer Sciences and the Paul Ivanier Center for Robotics and Production Management.

R. Zivan (✉) · M. Zazone · A. Meisels  
Department of Industrial Engineering and Management,  
Ben-Gurion University of the Negev,  
Beer-Sheva, 84-105, Israel  
e-mail: zivanr@bgu.ac.il

M. Zazone  
e-mail: moshezaz@cs.bgu.ac.il

A. Meisels  
e-mail: am@cs.bgu.ac.il

The new version of retroactively ordered ABT is faster by a large factor than the best form of ABT.

**Keywords** Distributed constraints satisfaction · Search · Ordering heuristics

## 1 Introduction

Distributed constraint satisfaction problems (*DisCSPs*) are composed of agents, each holding its local constraints network, that are connected by constraints among variables of different agents. Agents assign values to variables, attempting to generate a locally consistent assignment that is also consistent with all constraints between agents [cf. 22, 23]. To achieve this goal, agents check the value assignments to their variables for local consistency and exchange messages with other agents, to check consistency of their proposed assignments against constraints with variables owned by different agents [4, 23].

A search procedure for a consistent assignment of all agents in a distributed CSP (*DisCSP*), is a distributed algorithm. All agents cooperate in search for a globally consistent solution. The solution involves assignments of all agents to all their variables and exchange of information among all agents, to check the consistency of assignments with constraints among agents.

*Asynchronous backtracking (ABT)* is one of the most efficient and robust algorithms for solving distributed constraints satisfaction problems. Asynchronous backtracking was first presented by Yokoo [23, 24] and was developed further and studied in Hamadi [9], Bessiere et al. [3], Silaghi and Faltings [17], Bessiere et al. [4]. Agents in the *ABT* algorithms perform assignments asynchronously according to their current view of the system's state. The method performed by each agent is in general simple. Later versions of *ABT* use polynomial space memory and perform dynamic backtracking [3, 4]. The versions of asynchronous backtracking in all of the above studies use a static priority order among all agents.

An asynchronous algorithm with dynamic ordering was proposed in Yokoo [25], asynchronous weak commitment (*AWC*). According to [23], *AWC* outperforms *ABT* on specific applications (N-queens, Graph-coloring). The heuristic used by *AWC* is very specific. Move any agent that sends back a *Nogood* to be first in the order of all agents [23]. However, in order to be complete, *AWC* uses exponential space for storing *Nogoods* [23]. This can be a problem when solving hard instances of *DisCSPs*.

An attempt to combine *ABT* with *AWC* was reported by Silaghi et al. [18]. In order to perform asynchronous finite reordering operations [18] propose that the reordering operation will be performed by abstract agents. In a later study the exact heuristic of *Dynamic Backtracking* [8] was proposed for *ABT* with dynamic ordering [19]. The results presented in both studies [18, 19] show minor improvements to *ABT* with static ordering.

A general algorithm for dynamic ordering in asynchronous backtracking, *ABT\_DO*, was presented in Zivan Meisels [28]. The *ABT\_DO* algorithm uses polynomial space, similarly to standard *ABT*. In the *ABT\_DO* algorithm the agents of the *DisCSP* choose orders dynamically and asynchronously. Agents in *ABT\_DO* perform computation according to the current, most updated order they hold. There

are three rules on the changes of orderings of agents in *ABT\_DO*. Each agent can change the order of agents who have a lower priority than its own. An agent can propose an order change each time it replaces its assignment. Each order is time-stamped according to the assignments performed by agents [29]. The method of time-stamping for defining the most updated order is the same that was used in Nguyen et al. [15] and Meisels and Zivan [13] for choosing the most updated partial assignment. A simple array of counters represents the priority of a proposed order, according to the global search tree.

The results presented in Zivan and Meisels [29] show that the performance of *ABT\_DO* is highly dependent on the selected heuristic. The classic *Min-domain* heuristic was implemented by including the current domain size of agents in the messages they send. Although this heuristic has privacy consequences it was expected to improve the runtime by a large factor as in the case of sequential algorithms. Surprisingly, this heuristic which in centralized algorithms and in distributed algorithms using a sequential assignment protocol produces a large improvement over static order, was found not to be efficient for asynchronous backtracking. A heuristic which achieved a significant improvement was inspired by *Dynamic Backtracking* [1, 8] in which the agent which sends a *Nogood* is advanced in the new order to be immediately after the agent to whom the *Nogood* was sent. The explanation for the success of this heuristic is that it does not cause the removal of relevant *Nogoods* as do other heuristics [29].

The present paper investigates the relation between the success of this heuristic and the *Min-domain* heuristic which was found to be successful for sequential assignments algorithms on *DisCSPs* [5, 13]. We demonstrate the effect of *Nogood* loss as a result of reordering on the failure of the *Min-domain* heuristic. Removal of *Nogoods* cause the return of values to the domains of agents. This harms the accuracy of the information that agents hold on the domain size of other agents. On the other hand, the *Nogood-triggered* heuristic of Zivan and Meisels [29] does not lose valid information and moves agents with a potential of having a smaller domain to a higher position.

In order to maximize the *Min-domain* property, a more flexible heuristic is proposed, which violates the restrictions on the ordering of agents in Zivan and Meisels [29]. We study changes of order that move agents to a higher position, replacing agents that were ahead of them including the first agent. This new type of heuristics is termed *Retroactive ordering* and is based on a slightly modified version of *ABT\_DO*. The studied scheme of dynamic variable ordering is more flexible than that of any centralized algorithm. As in *ABT\_DO*, agents change order only when an assignment is replaced. However, agents can be moved to higher priority positions than the agent which changes the assignment. The degree of flexibility of the heuristic is dependent upon the size of *Nogood* storage which is predefined. Agents are limited to store *Nogoods* equal or smaller than a predefined size  $k$ . When the *Nogood* is smaller or equal to  $k$  the agent which found the *Nogood* can be moved to a position in front of the agents included in the *Nogood*. These agents are required to store the *Nogood*. A specific case of this general definition is the *AWC* algorithm [23]. In the case of *AWC* one has  $k = n$  and all *Nogoods* are stored. The other extreme is to not allow the move of agents in front of the last agent in the *Nogood* which is not the culprit.

The results presented in the present paper show that moving the *Nogood* sender as high as possible in the priority order is successful only if the domain size of agents

is taken into consideration. Our experiments show that the successful heuristics are those that support a *Min-domain* scheme in which agents are moved to a higher position only if their current domain size is smaller than the current domain of agents they are moved in front of. Moving an agent before the agents which are included in the *Nogood* actually *enlarges* its domain. The best heuristic in the present paper is that agents which generate a *Nogood* are placed in the new order between the last and the second last agents in the generated *Nogood*. This heuristic is the asynchronous form of the Min-Domain heuristic and *does not require any additional storage of Nogoods*. Agents are moved to a higher position only if their domain is smaller than the agents they pass on the way up. Our results on both random *DisCSPs* and on structured *DisCSPs* show that the proposed heuristic improves the best results to date by a large factor.

Distributed *CSPs* are presented in Section 2. A description of asynchronous backtracking with dynamic ordering (*ABT\_DO*) and its best heuristic is presented in Section 4. Section 5 presents an investigation of the existing heuristics and offers reasons for their performance in previous papers. Section 6 present the general scheme of retroactive heuristics for *ABT\_DO* and a correctness proof. An extensive experimental evaluation, which compares standard and retroactive heuristics of *ABT\_DO* is in Section 8. The experiments were conducted on randomly generated *DisCSPs* and on Course Scheduling problems. Section 9 presents a discussion of the relation between the experimental results and the *Min-domain* heuristic.

## 2 Distributed Constraint Satisfaction

A distributed constraint satisfaction problem - *DisCSP* is composed of a set of  $k$  agents  $A_1, A_2, \dots, A_k$ . Each agent  $A_i$  contains a set of constrained variables  $X_{i_1}, X_{i_2}, \dots, X_{i_{n_i}}$ . Constraints or **relations**  $R$  are subsets of the Cartesian product of the domains of the constrained variables. For a set of constrained variables  $X_{i_k}, X_{j_l}, \dots, X_{m_n}$ , with domains of values for each variable  $D_{i_k}, D_{j_l}, \dots, D_{m_n}$ , the constraint is defined as  $R \subseteq D_{i_k} \times D_{j_l} \times \dots \times D_{m_n}$ . A **binary constraint**  $R_{ij}$  between any two variables  $X_j$  and  $X_i$  is a subset of the Cartesian product of their domains;  $R_{ij} \subseteq D_j \times D_i$ . In a distributed constraint satisfaction problem *DisCSP*, constrained variables can belong to different agents [24]. Each agent has a set of constrained variables, i.e. a *local constraint network*.

An assignment (or a label) is a pair  $\langle var, val \rangle$ , where  $var$  is a variable of some agent and  $val$  is a value from  $var$ 's domain that is assigned to it. A *compound label* (or a partial solution) is a set of assignments of values to a set of variables. A **solution**  $P$  to a *DisCSP* is a compound label that includes all variables of all agents, that satisfies all the constraints. Agents check assignments of values against non-local constraints by communicating with other agents through sending and receiving messages. Agents exchange messages with agents whose assignments may be in conflict [4]. Agents connected by constraints are therefore called *neighbors*. The ordering of agents is termed *priority*, so that agents that are later in the order are termed "lower priority agents" [4, 23].

The following assumptions are routinely made in studies of *DisCSPs* and are assumed to hold in the present study [4, 23].

1. All agents hold exactly one variable.
2. Messages arrive at their destination in finite time.
3. Messages sent by agent  $A_i$  to agent  $A_j$  are received by  $A_j$  in the order they were sent.

### 3 Asynchronous Backtracking (*ABT*)

The *asynchronous backtracking* algorithm, was presented in several versions over the last decade and is described here in the form of the more recent papers [4, 23]. In the *ABT* algorithm, agents hold an assignment for their variables at all times, which is consistent with their view of the state of the system (i.e. their *Agent\_view*). When the agent cannot find an assignment which is consistent with its *Agent\_view*, it changes its view by eliminating a conflicting assignment from its *Agent\_view* data structure. It then sends back a *Nogood* which is based on its former inconsistent *Agent\_view* and makes another attempt to assign its variable [4, 23].

The code of the asynchronous backtracking algorithm (*ABT*) is presented in Fig. 1. *ABT* has a total order of priorities among agents. Agents hold a data structure called *Agent\_view* which contains the most recent assignments received from agents with higher priority. The algorithm starts by each agent assigning its variable, and sending the assignment to neighboring agents with lower priority. When an agent receives a message containing an assignment ([23] an **ok?** message; [23]), it updates its *Agent\_view* with the received assignment and if needed replaces its own assignment, to achieve consistency (first procedure in Fig. 1). Agents that reassign their variable, inform their lower priority neighbors by sending them **ok?** messages (Procedure **check\_agent\_view**, lines 3–5). Agents that cannot find a consistent assignment, send the inconsistent tuple in their *Agent\_view* in a backtrack message (a *Nogood* message; [23]) and remove from their *Agent\_view* the assignment of the lowest priority agent in the inconsistent tuple. In the simplest form of the *ABT* algorithm, the complete *Agent\_view* is sent as a *Nogood* [23]. The *Nogood* is sent to the lowest priority agent whose assignment is included in the *Nogood*. After the culprit assignment is removed from the *AgentView* the agent makes another attempt to assign its variable by calling procedure **check\_agent\_view** (procedure **backtrack** in Fig. 1).

Agents that receive a *Nogood*, check its relevance against the content of their *Agent\_view*. If the *Nogood* is relevant the agent stores it, and tries to find a consistent assignment. If the agent receiving the *Nogood* keeps its assignment, it informs the *Nogood* sender by resending it an **ok?** message with its assignment. An agent  $A_i$  which receives a *Nogood* containing an assignment of agent  $A_j$  which is not included in its *Agent\_view*, adds the assignment of  $A_j$  to its *Agent\_view* and sends a message to  $A_j$  asking it to add a link between them, i.e. inform  $A_i$  about all assignment changes it performs in the future (second procedure in Fig. 1).

**when received** ( $ok?, (x_j, d_j)$ ) **do**

1. add  $(x_j, d_j)$  to *agent\_view*;
2. **check\_agent\_view**; **end\_do**;

**when received** ( $nogood, x_j, nogood$ ) **do**

1. add *nogood* to *nogood* list;
2. **when** *nogood* contains an agent  $x_k$  that is not a neighbor **do**
3. request  $x_k$  to add  $x_i$  as a neighbor,
4. and add  $(x_k, d_k)$  to *agent\_view*; **end\_do**;
5.  $old\_value \leftarrow current\_value$ ; **check\_agent\_view**;
6. **when**  $old\_value = current\_value$  **do**
7. send ( $ok?, (x_i, current\_value)$ ) to  $x_j$ ; **end\_do**; **end\_do**;

procedure **check\_agent\_view**

1. **when** *agent\_view* and *current\_value* are not consistent **do**
2. **if** no value in  $D_i$  is consistent with *agent\_view* **then backtrack**;
3. **else** select  $d \in D_i$  where *agent\_view* and  $d$  are consistent;
4.  $current\_value \leftarrow d$ ;
5. send ( $ok?, (x_i, d)$ ) to *low\_priority\_neighbors*; **end\_if**; **end\_do**;

procedure **backtrack**

1.  $nogood \leftarrow resolve\_Nogoods$ ;
2. **if** *nogood* is an empty set **do**
3. broadcast to other agents that there is no solution;
4. terminate this algorithm; **end\_do**;
5. select  $(x_j, d_j)$  where  $x_j$  has the lowest priority in *nogood*;
6. send (**nogood**,  $x_i, nogood$ ) to  $x_j$ ;
7. remove  $(x_j, d_j)$  from *agent\_view*; **end\_do**;
8. **check\_agent\_view**

**Fig. 1** Standard ABT algorithm

The performance of *ABT* can be improved immensely by requiring agents to read all messages they receive before performing computation [4, 23]. This technique was found to improve the performance of *asynchronous backtracking* on the harder instances of randomly generated Distributed CSPs by a large factor [5, 27].

Another improvement to the performance of *ABT* can be achieved by using the method for resolving inconsistent subsets of the *Agent\_view*, based on methods of dynamic backtrack. A version of *ABT* that uses this method was presented in Bessiere et al. [4]. In all the experiments in this paper, a version of *ABT* which includes both of the above improvements is used. Agents read all incoming messages that were received before performing computation and *Nogoods* are resolved, using the dynamic backtracking method.

#### 4 ABT with Dynamic Ordering

Each agent in *ABT\_DO* holds a *Current\_order* which is an ordered list of pairs. Every pair includes the ID of one of the agents and a counter. Each agent can propose a new order for agents that have lower priority (i.e. are in a lower position in the current order), each time it replaces its assignment. This way the sending of an ordering proposal message always coincides with an assignment message (an **ok?** message; [26]). An agent  $A_i$  can propose an order according to the following rules:

1. Agents with higher priority than  $A_i$  and  $A_i$  itself, do not change priorities in the new order.
2. Agents with lower priority than  $A_i$ , in the current order, can change their priorities in the new order *but not to a higher priority than  $A_i$  itself* (This rule enables a more flexible order than in the centralized case).

The counters attached to each agent ID in the *order* list form a time-stamp. Initially, all time-stamp counters are set to zero and all agents start with the same *Current\_order*. Each agent  $A_i$  that proposes a new order, changes the order of the pairs in its own ordered list and updates the counters as follows:

1. The counters of agents with higher priority than  $A_i$ , according to the *Current\_order*, are not changed.
2. The counter of  $A_i$  is incremented by one.
3. The counters of agents with lower priority than  $A_i$  in the *Current\_order* are set to zero.

In *ABT*, agents send **ok?** messages to their neighbors whenever they perform an assignment. In *ABT\_DO*, an agent can choose to change its *Current\_order* after changing its assignment. If that is the case, besides sending **ok?** messages an agent sends **order** messages to all lower priority agents. The **order** message includes the agent's new *Current\_order*.

An agent which receives an **order** message must determine if the received order is more updated than its own *Current\_order*. It decides by comparing the time-stamps lexicographically. Since orders are changed according to the above rules, every two orders must have a common prefix of agents' IDs. The agent that performs the change does not change its own position and the positions of higher priority agents.

When an agent  $A_i$  receives an order which is more up to date than its *Current\_order*, it replaces its *Current\_order* by the received order. The new order might change the location of the receiving agent with respect to other agents (in the new *Current\_order*). In other words, one of the agents that had higher priority than  $A_i$  according to the old order, now has a lower priority than  $A_i$  or vice versa. Therefore,  $A_i$  rechecks the consistency of its current assignment and the validity of its stored *Nogoods* (the explanations for removing values from its domain; [26, 29]) according to the new order. If the current assignment is inconsistent according to the new order, the agent makes a new attempt to assign its variable. In *ABT\_DO* agents send **ok?** messages to all constraining agents (i.e. their neighbors in the constraints graph). Although agents might hold in their *Agent\_views* assignments of agents with

lower priorities, according to their *Current\_order*, they eliminate values from their domain *only if they violate constraints with higher priority agents*.

A *Nogood* message (i.e. a message carrying a partial assignment which was found to be inconsistent; [26, 29]) is always checked according to the *Current\_order* of the receiving agent. If the receiving agent is not the lowest priority agent in the *Nogood* according to its *Current\_order*, it sends the *Nogood* to the lowest priority agent and sends an **ok?** message to the sender of the *Nogood*. This is a similar operation to that performed in standard *ABT* for any unaccepted (inconsistent) *Nogood* [4].

Figures 2 and 3 present the code of asynchronous backtracking with dynamic ordering (*ABT\_DO*).

When an **ok?** message is received (first procedure in Fig. 2), the agent updates the *Agent\_view* and removes inconsistent *Nogoods*. Then it calls **check\_agent\_view** to make sure its assignment is still consistent.

A new order received in an order message is accepted only if it is more up to date than the *Current\_order* (second procedure of Fig. 2). If so, the received order is stored and **check\_agent\_view** is called to make sure the current assignment is consistent with the higher priority assignments in the *Agent\_view*.

When a *Nogood* is received (third procedure in Fig. 2) the agent first checks if it is the lowest priority agent in the received *Nogood*, according to the *Current\_order*. If not, it sends the *Nogood* to the lowest priority agent and an **ok?** message to

**when received (ok?,  $(x_j, d_j)$ ) do:**

1. add  $(x_j, d_j)$  to *agent\_view*;
2. remove inconsistent *nogoods*;
3. **check\_agent\_view**;

**when received (order, *received\_order*) do:**

1. **if** (*received\_order* is more updated than *Current\_order*)
2.     *Current\_order*  $\leftarrow$  *received\_order*;
3.     remove inconsistent *nogoods*;
4.     **check\_agent\_view**;

**when received (nogood,  $x_j$ , *nogood*) do**

1. **if** (*nogood* contains an agent  $x_k$  with lower priority than  $x_i$ )
2.     send (**nogood**,  $(x_i, \textit{nogood})$ ) to  $x_k$ ;
3.     send (**ok?**,  $(x_i, \textit{current\_value})$ ) to  $x_j$ ;
4. **else**
5.     **if** (*nogood* consistent with  $\{\textit{Agent\_view} \cup \textit{current\_assignment}\}$ )
6.         store *nogood*;
7.         **if** (*nogood* contains an agent  $x_k$  that is not its neighbor)
8.             request  $x_k$  to add  $x_i$  as a neighbor;
9.             add  $(x_k, d_k)$  to *agent\_view*;
10.         **check\_agent\_view**;
11.     **else**
12.         send (**ok?**,  $(x_i, \textit{current\_value})$ ) to  $x_j$ ;

**Fig. 2** The *ABT\_DO* algorithm (first part)

procedure **check\_agent\_view**

1. **if**(*current\_assignment* is not consistent with all higher priority assignments in *agent\_view*)
2.   **if**(no value in  $D_i$  is consistent with all higher priority assignments in *agent\_view*)
3.     **backtrack**;
4.   **else**
5.     select  $d \in D_i$  where *agent\_view* and  $d$  are consistent;
6.     *current\_value*  $\leftarrow d$ ;
7.     *Current\_order*  $\leftarrow$  **choose\_new\_order**
8.     send (**ok?**,( $x_i, d$ )) to *neighbors*;
9.     send (**order**,*Current\_order*) to lower priority agents;

procedure **backtrack**

1. *nogood*  $\leftarrow$  **resolve\_inconsistent\_subset**;
2. **if** (*nogood* is empty)
3.   broadcast to other agents that there is no solution;
4.   **stop**;
5.   select ( $x_j, d_j$ ) where  $x_j$  has the lowest priority in *nogood*;
6.   send (**nogood**,  $x_i, nogood$ ) to  $x_j$ ;
7.   remove ( $x_j, d_j$ ) from *agent\_view*;
8.   remove all *Nogoods* containing ( $x_j, d_j$ );
9.   **check\_agent\_view**;

**Fig. 3** The ABT\_DO algorithm (second part)

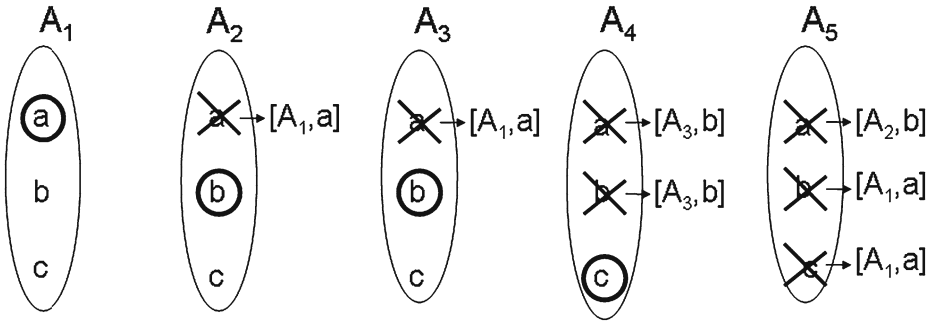
the *Nogood* sender (lines 1-3). If the receiving agent is the lowest priority agent it performs the same operations as in the standard *ABT* algorithm (lines 4-12).

Procedure **backtrack** (Fig. 3) is the same as in standard *ABT*. The *Nogood* is resolved and the result is sent to the lower priority agent in the *Nogood*, according to the *Current\_order*.

Procedure **check\_agent\_view** (Fig. 3) is very similar to standard *ABT* but the difference is important (lines 5-9). If the current assignment is not consistent and must be replaced and a new consistent assignment is found, the agent chooses a new order as its *Current\_order* (line 7) and updates the corresponding time-stamp. Next, **ok?** messages are sent to all neighboring agents. The new order and its time-stamp counters are sent to all lower priority agents.

## 5 Investigation of asynchronous heuristics

In this section we offer explanations for the failure of the *Min-domain* heuristic and the success of the *Nogood-triggered* heuristic when used in asynchronous backtracking [28, 29]. Consider the example in Fig. 4. The agents are ordered by their indices. Each agent has a single variable and three values,  $a$ ,  $b$  and  $c$ , in its domain. The eliminated values are crossed and each points to its eliminating explanation (i.e. the assignment which caused its removal). The circled values represent the current

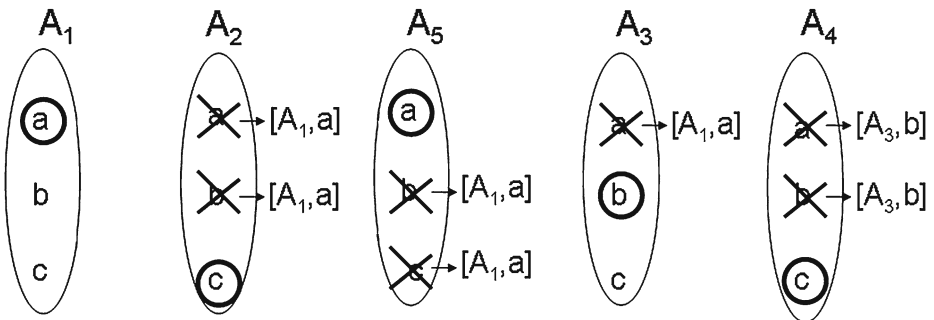


**Fig. 4** Heuristics example before backtrack

assignments. In this example, agent  $A_5$  has exhausted its domain and must create a *Nogood*. The *Nogood* it generates includes the assignments of  $A_1$  and  $A_2$  therefore the *Nogood* is sent to  $A_2$ . According to the rules of the *ABT\_DO* algorithm agent  $A_2$  can reorder agents  $A_3$ ,  $A_4$  and  $A_5$ . Now, if it will reorder them according to their current domain sizes then  $A_3$  and  $A_4$  will switch places. But, since both of the values eliminated from the domain of  $A_4$  are in conflict with the assignment of  $A_3$  then after they change places, these values will be returned to the domain of  $A_4$  and its domain size will be larger than the domain of  $A_3$ .

In contrast, if  $A_2$  reorders according to the *Nogood-triggered* heuristic then the only agent to change places is  $A_5$  which is moved to be after  $A_2$  and before  $A_3$ . Now, after  $A_2$  replaces its assignment we get the situation in Fig. 5. We can see that an agent with a small domain was moved forward while the others kept their domain sizes and places.

The example demonstrates why the *Min-domain* heuristic fails when used in asynchronous backtracking. In asynchronous backtracking, all agents hold an assignment throughout the search. Conflicts with these assignments effect the size of domains of other agents. For each value which is removed from an agent's domain an explanation *Nogood* is stored. When an agent is moved in front of an agent whose assignment is included in one of its *Nogoods*, this *Nogood* must be eliminated and the corresponding value is returned to the domain. Thus, in contrast to sequential



**Fig. 5** After reordering using the NG-triggered heuristic

ordering algorithms, in asynchronous backtracking the resulting domain sizes after reordering cannot be anticipated by the ordering agent. The example demonstrates how this phenomena does not affect the *Nogood-triggered* heuristic.

Following the example one can see that the *Nogood-triggered* heuristic is successful because in many cases it moves an agent with a small domain to a higher position. Only values whose *Nogood* explanation includes the assignment of the culprit agent are returned to the moving agent's domain. In fact, the agent can be moved up passed the culprit, and as long as it does not pass the second last assignment in the *Nogood* its domain size will stay the same. In Fig. 5, Agent  $A_5$  is moved right after agent  $A_2$ . Its domain size is one, since the *Nogoods* of its other two values are valid. If  $A_5$  is moved before  $A_2$  its domain size will stay the same as both eliminating *Nogoods* include only the assignment of  $A_1$ . However, if  $A_5$  will be moved in front of  $A_1$  then all its values will return to its domain. This possibility of moving an agent with a small domain beyond the culprit agent to a higher position is the basic motivation for retroactive ordering.

## 6 Retroactive ordering heuristics for ABT

In contrast to the rules of *ABT\_DO* of the previous section, the present paper proposes a new type of ordering. The new type of ordering can change the order of agents with higher priority than the agent which replaces its assignment. The best heuristic which was presented in [29] moved an agent which has detected a dead end and created a *Nogood*, to be right after the agent it has sent the *Nogood* to. A *retroactive* heuristic would enable moving the *Nogood* sender to a higher position than the *Nogood* receiver. In order to preserve the correctness of the algorithm, agents must be allowed to store *Nogoods*. In order to generate a general scheme for retroactive heuristics, one can define a global space limit for the storage of *Nogoods*. The specific realization is to limit the storage of *Nogoods* that are smaller or equal to some predefined size  $k$ . This makes the space complexity of the agents exponential in  $k$  so keeping  $k$  small is important.

As in standard *ABT\_DO*, the proposed ordering heuristic is triggered by the sending of a *Nogood*. The reordering operation can be generated by either the *Nogood* generator or by the *Nogood* receiver (but not by both). In contrast to Zivan and Meisels [28] and [29] we choose the *Nogood* sender to be the one to reorder. This is since the only agent which can lose a relevant *Nogood* as a result of the reordering is the *Nogood* sender (the only one moving to a higher position). Therefore, since it is aware of its own state and the others do not lose information, the *Nogood* sender is the best candidate for selecting the new order.

The new order is selected according to the following rules:

1. The *Nogood* generator can be moved to any position in the new order.
2. If the *Nogood* generator is moved to a position which is before the second last in the *Nogood* (the one before the culprit) all the agents included in the *Nogood* must hold the *Nogood* until the search is terminated.
3. Agents with lower priority than the *Nogood* receiver can change order but not move in front of it (as in standard *ABT\_DO*).

According to the above rules, agents which detect a dead end are moved to a higher position in the priority order. If the length of the created *Nogood* is larger than  $k$ , they can be moved up to the place that is right after the agent which is the last to be included in the *Nogood* according to the current order and is not the culprit (i.e. second last in the *Nogood*).

If the length of the created *Nogood* is smaller or equal to  $k$ , the sending agent can be moved to a position before all the participants in the *Nogood* and the *Nogood* is sent and saved by all of them. In the extreme case where  $k$  is equal to the number of agents in the *DisCSP* (i.e.  $k = N$ ), the *Nogood* sender can always move to be first in the priority order and the resulting algorithm is a generalization of *AWC* [23].

Figures 6 and 7 present the code of *Retroactive ABT\_DO*. The difference from standard *ABT\_DO* in the code performed when a *Nogood* is received (Fig. 6) derives from the different possible types of *Nogoods*. A *Nogood* smaller or equal to  $k$  is actually a constraint that will be stored by the agent until the search is terminated. In the case of *Nogoods* which are longer than  $k$ , the algorithm treats them as in standard *ABT\_DO* i.e. accepts them only if the receiver is the lowest priority agent in the *Nogood* and the *Nogood* is consistent with the *Agent\_view* and

**when received (ok?,  $(x_j, d_j)$ ) do:**

1. add  $(x_j, d_j)$  to *agent\_view*;
2. remove inconsistent *nogoods*;
3. **check\_agent\_view**;

**when received (order, *received\_order*) do:**

1. **if** (*received\_order* is more updated than *Current\_order*)
2.     *Current\_order*  $\leftarrow$  *received\_order*;
3.     remove inconsistent *nogoods*;
4.     **check\_agent\_view**;

**when received (nogood,  $x_j$ , *nogood*)**

1. *old\_value*  $\leftarrow$  *current\_value*
2. **if** (*nogood* contains an agent  $x_k$   
    with lower priority than  $x_i$  and *nogood.size*  $>$   $K$ )
3.     send (**nogood**,  $(x_i, \textit{nogood})$ ) to  $x_k$ ;
4. **else**
5.     **if** (*nogood* consistent with  $\{Agent\_view \cup$   
    *current\_assignment* $\}$  or *nogood.size*  $\leq$   $K$ )
6.     store *nogood*;
7.     **if** (*nogood* contains an agent  $x_k$  that is not its neighbor)
8.     request  $x_k$  to add  $x_i$  as a neighbor;
9.     add  $(x_k, d_k)$  to *agent\_view*;
10.    **if** ( $x_i$  is with lowest priority in *nogood*)
11.    **check\_agent\_view**;
12. **if** (*old\_value* = *current\_value*)
13.    send (**ok?**,  $(x_i, \textit{current\_value})$ ) to  $x_j$ ;

**Fig. 6** Retroactive ABT\_DO algorithm (first part)

```

procedure backtrack
1. nogood ← resolve_inconsistent_subset;
2. if (nogood is empty)
3.   broadcast to other agents that there is no solution;
4.   stop;
5. select  $(x_j, d_j)$  where  $x_j$  has the lowest priority in nogood;
6. if(nogood.size > K)
7.   Current_order ← choose_new_order( $x_i$ )
       where  $x_i$  has the second lowest priority in nogood;
8.   send (nogood,  $x_i$ , nogood) to  $x_j$ ;
9. else if(is_new(nogood))
10.  new_position ← unlimited
11.  send (nogood,  $x_i$ , nogood) to all agents in nogood;
12.  store sent nogood;
13. Current_order ← choose_new_order(null)
14. send (order, Current_order) to lower priority agents;
15. remove  $(x_j, d_j)$  from agent_view;
16. remove all nogoods containing  $(x_j, d_j)$ ;
17. check_agent_view;

procedure check_agent_view
1. if(current_assignment is not consistent with all
   higher priority assignments in Agent_view)
2.   if(no value in  $D_i$  is consistent with all higher priority
   assignments in Agent_view)
3.     backtrack;
4.   else
5.     select  $d \in D_i$  where Agent_view and  $d$  are consistent;
6.     current_value ←  $d$ ;
7.     send (ok?, ( $x_i, d$ )) to neighbors;

```

**Fig. 7** The retroactive ABT\_DO algorithm (second part)

*current\_assignment* of the receiver. In any case of acceptance of a *Nogood*, the agent searches for a new assignment only if it happens to be the lowest priority agent in the *Nogood*. As stated above, our choice is that only the *Nogood* generator is allowed to change order.

Procedure **backtrack** (Fig. 7) is largely changed in the retroactive heuristic version of *ABT\_DO*. When an agent creates a *Nogood* it determines whether it is larger than  $k$  or not. If it is larger then a single *Nogood* is sent to the lowest priority agent in the *Nogood* in the same way as in *ABT\_DO*. Consequently, the agent selects a new order in which it puts itself not higher than the second lowest priority agent in the *Nogood*. When the *Nogood* is smaller or equal to  $k$ , if it is the first time this *Nogood* is generated, the *Nogood* is sent to all the agents included in the *Nogood* and the agent moves itself to an unlimited position in the new order (In this case the function **choose\_new\_order** is called with no limitations). In both cases, order messages are sent to all the lower priority agents in the new order. The assignment

of the lowest priority agent in the *Nogood* is removed from the *Agent\_view*, the relevant *Nogoods* are removed and the agent attempts to re-assign its variable by calling **check\_agent\_view**.

Procedure **check\_agent\_view** (Fig. 6) is slightly changed from that of standard *ABT\_DO* since the change of order in the new scheme is performed by the *Nogood* sender and not by its receiver.

## 7 Correctness of Retroactive *ABT\_DO*

In order to prove the correctness of *Retroactive ABT\_DO* we assume the correctness of the standard *ABT\_DO* algorithm (see proof in [29]) and prove that the changes made for retroactive heuristics do not damage its correctness. We first prove the case for no *Nogood* storage ( $k = 0$ ):

**Theorem 1** *Retroactive ABT\_DO is correct when  $k = 0$ .*

There are two differences between standard *ABT\_DO* and *Retroactive ABT\_DO* with  $k = 0$ . First, order is changed whenever a *Nogood* is sent and not when an assignment is replaced. This change does not make a difference in the correctness since when a *Nogood* is sent there are two possible outcomes. Either the *Nogood* receiver replaces its assignment, which makes it effectively the same as in standard *ABT\_DO*, or the *Nogood* is rejected. A rejected *Nogood* can only be caused by a change of assignment either of the receiving agent or of an agent with higher priority. In all of these cases, the most relevant order is determined lexicographically. Ties which could not have been generated in standard *ABT\_DO*, are broken using the agents indexes.

The second change in the code for  $k = 0$  is that in *Retroactive ABT\_DO* a *Nogood* sender can move to a position in front of the agent that receives the *Nogood*. Since the *Nogood* sender is the only agent moving to a higher position, it is the only one that can lose a *Nogood* as a result. However, the *Nogood* sender removes all *Nogoods* containing the assignment of the *Nogood* receiver and it does not pass any other agent contained in the *Nogood*. Thus, no information is actually lost by this change. Moreover, the number of times two agents can move in front of one another without a higher priority agent changing its assignment is bounded by their domain sizes.

**Theorem 2** *Retroactive ABT\_DO is correct when  $n \geq k > 0$ .*

In order to prove that *Retroactive ABT\_DO* is correct for the case that  $n \geq k > 0$  we need to show that infinite loops cannot occur. In the case of *Nogoods* which are smaller or equal to  $k$  the case is very simple. All agents involved in the *Nogood* continue to hold it, therefore the same assignment can never be produced again. The number of these *Nogoods* with a limited length is finite. In finite time the algorithm reaches a state in which no permanent *Nogoods* are added. In this state, agents do not move in front of the second last in the *Nogoods* generated and the previous proof holds.

### 8 Experimental evaluation

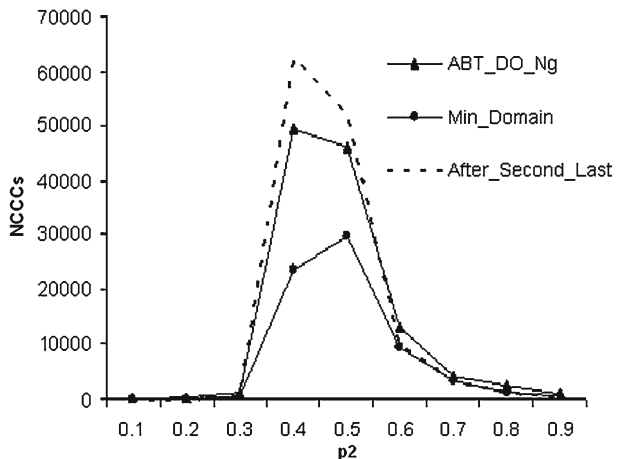
The common approach in evaluating the performance of distributed algorithms is to compare two independent measures of performance – time, in the form of steps of computation [12, 23], and communication load, in the form of the total number of messages sent [12].

Non concurrent steps of computation, are counted by a method similar to the clock synchronization algorithm of Lamport [11]. Every agent holds a counter of computation steps. Every message carries the value of the sending agent’s counter. When an agent receives a message it stores the data received together with the corresponding counter. When the agent first uses the received counter it updates its counter to the largest value between its own counter and the stored counter value which was carried by the message [30]. By reporting the cost of the search as the largest counter held by some agent at the end of the search, a measure of non-concurrent search effort that is close to Lamports logical time is achieved [11]. If instead of steps of computation, the number of non concurrent constraint checks is counted (NCCCs), then the local computational effort of agents in each step is measured [14, 30].

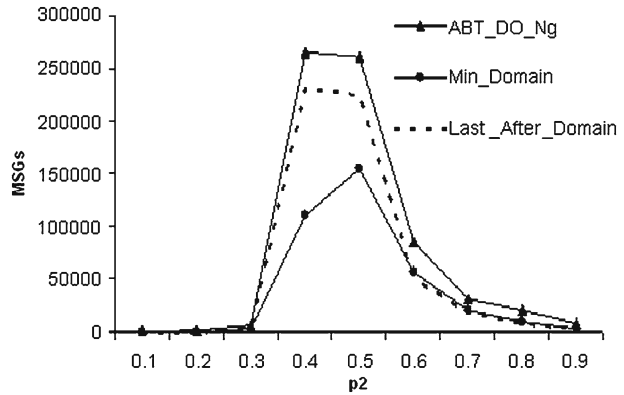
The first set of experiments was conducted on random networks of constraints of  $n$  variables,  $k$  values in each domain, a constraints density of  $p_1$  and tightness  $p_2$  (which are commonly used in experimental evaluations of CSP algorithms [16, 21]). The constraint networks were generated with 20 agents ( $n = 20$ ) each holding exactly one variable, 10 values for each variable ( $k = 10$ ) with two different constraints densities  $p_1 = 0.4$  and  $p_1 = 0.7$ . The tightness value  $p_2$ , is varied between 0.1 and 0.9, to cover all ranges of problem difficulty. For each pair of fixed density and tightness ( $p_1, p_2$ ) 50 different random problems were solved by each algorithm and the results presented are an average of these 50 runs.

In order to confirm the dependency of the performance on the size of the current domain of the moved agents, we compared *ABT\_DO* with *ABT\_DO* with a retroactive heuristic in which agents are not allocated any additional *Nogood* storage. Agents include in their messages the size of their current domains. This

**Fig. 8** Non concurrent constraints checks performed by retroactive *ABT\_DO* and *ABT\_DO* on low density DisCSPs ( $p_1 = 0.4$ )



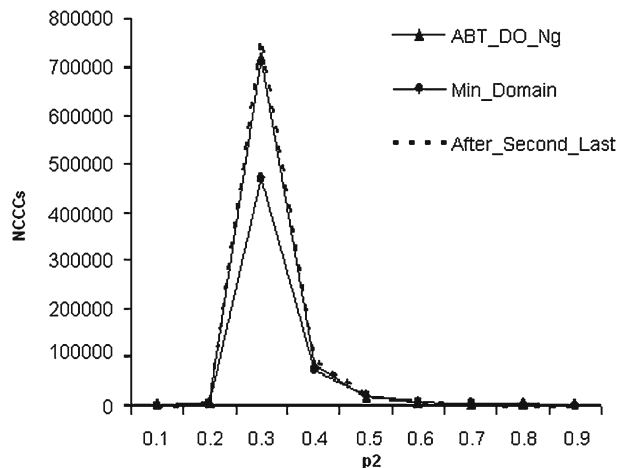
**Fig. 9** Number of messages sent by retroactive *ABT\_DO* and *ABT\_DO* on low density DisCSPs ( $p_1 = 0.4$ )



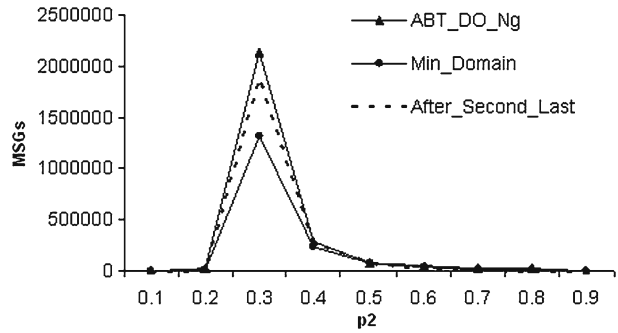
information is stored in the agent’s *Agent\_views*. A *Nogood* generator moves itself to be in a higher position than the culprit agent but it moves in front of an agent *only if its current domain is smaller* than the domain of that agent. Otherwise, it places itself right after the culprit agent as in standard *ABT\_DO*.

Figure 8 presents the results in *NCCCs* for *ABT\_DO* and Retroactive *ABT\_DO* with the above heuristic. The retroactive version of *ABT\_DO* (depicted in the figures as *Min-domain*) improves the run-time performance of *ABT\_DO* (depicted as *ABT\_DO\_NG*). In order to emphasize the relation to the *Min-domain* property, a third line in Figs. 8, 9, 10, 11, 16 and 17 represents retroactive *ABT\_DO* without checking the domain sizes (depicted in the figures as *After Second Last*). This version of retroactive *ABT\_DO* was the slowest among the three. Similar results for the number of messages sent are presented in Fig. 9. In the case of network load, both versions of *Retroactive ABT\_DO* send less messages than standard *ABT\_DO*. For high density problems the difference between the algorithms is similar but smaller (Figs. 10 and 11).

**Fig. 10** Non concurrent constraints checks performed by retroactive *ABT\_DO* and *ABT\_DO* on high density DisCSPs ( $p_1 = 0.7$ )



**Fig. 11** Number of messages sent by retroactive *ABT\_DO* and *ABT\_DO* on high density DisCSPs ( $p_1 = 0.7$ )



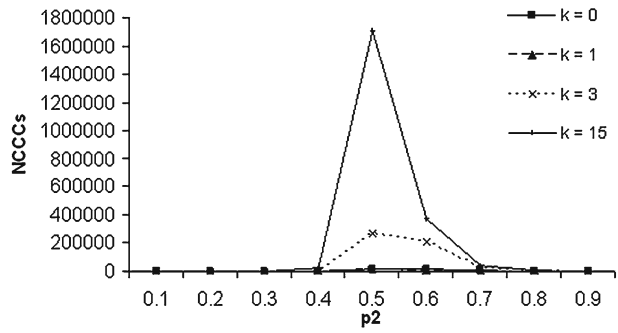
In order to further demonstrate the dependency of the domain size of agents on the success of the selected heuristic we performed an additional experiment on random problems in which the size limit for keeping *Nogoods* ( $k$ ) is varied<sup>1</sup>. A *Nogood* generator which created a *Nogood* of length larger than  $k$  places itself right after the *Nogood* receiver as in standard *ABT\_DO*. When the *Nogood* generator creates a *Nogood* smaller or equal to  $k$ , it places itself first in the priority order and sends the generated *Nogood* to all the participating agents. In the case of  $k = n$  the resulting algorithm is exactly *AWC* (in *AWC* an agent which generates a *Nogood* actually places it self in front of all its neighboring agents). In the case of  $k = 0$  the resulting algorithm is standard *ABT\_DO*. Figure 12 presents the number of *NCCCs* performed by the algorithm with  $k$  equal to 0, 1, 3 and  $n$  ( $n = 15$ ). The results show similar performance when  $k$  is small. The performance of the algorithm deteriorates when  $k = 3$  and the slowest performance is when  $k = n$ . Similar results in the number of messages are presented in Fig. 13.

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. However, one must examine the effect of the specific heuristic used on the size of the domains of the agents which are moved up in the order of priorities. An agent creates a *Nogood* when its domain empties. After sending the *Nogood*, it removes the assignment of the culprit agent from its *Agent\_view* and returns to the domain only values whose eliminating *Nogood* included the removed assignment. When the agent is moved in front of other agents whose assignments were included in the generated *Nogood* it must return more values to its domain (the values whose explanation *Nogood* included the assignment of the agent which was passed). This of course does not happen for the case of a *Nogood* of size one and that is why for  $k = 1$  we get better results. Thus, moving an agent as high as possible in the priority order actually results in moving upwards an agent with a larger domain.

The effect of uncertainty on the size of agents’ domains after reordering can be reduced. In the next set of experiments the best version of the retroactive *ABT\_DO* algorithm was compared with an additional version of the *ABT\_DO* algorithm which uses a new type of the *Min-domain* heuristic. In the new heuristic, beside their domain sizes, agents included in their *ok?* messages the union of all assignments that caused removal of assignments from its domain (i.e. the union of all its eliminating

<sup>1</sup>In this experiment the problems were smaller ( $n = 15$ ) since the algorithms run slower.

**Fig. 12** Non concurrent constraints checks performed by retroactive *ABT\_DO* with different limits on *Nogood* size ( $p_1 = 0.4$ )

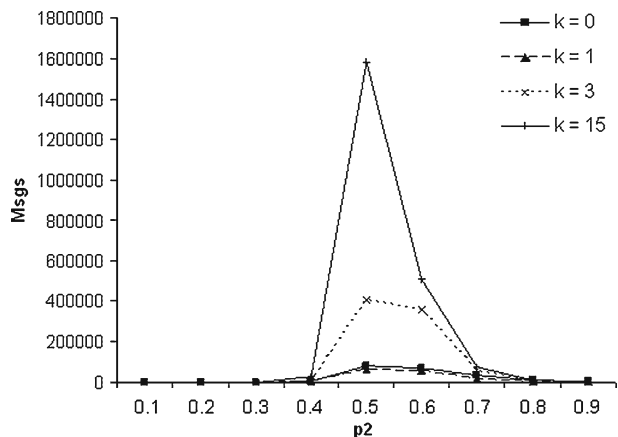


explanations or its *temporal Nogood*). Now the ordering agent knows what is the highest position it can move an agent in order to preserve its reported domain size. We have performed experiments in which the agents used this additional information in a number of ways:

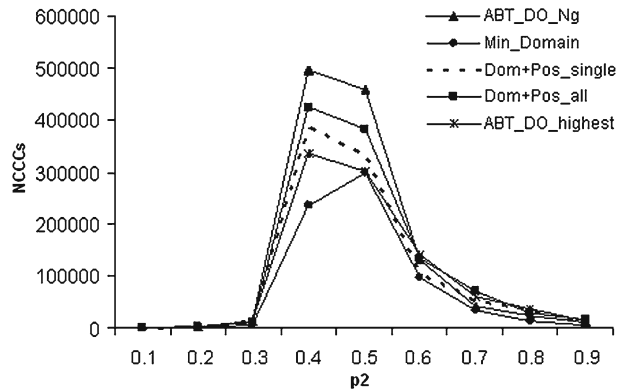
- The reordering agent moves the agent which the sum of its reported domain size and its new expected position (the highest place it can get according to the *Nogood* it sent with the same domain size) is the smallest, to the expected position (termed in the figure *Dom + Pos\_single*).
- The reordering agent *moves all lower priority agents* according to the sum of their new expected position and their reported domain size (termed in the figure *Dom + Pos\_all*).
- The reordering agent moves the agent which can move to the highest position according to the *nogood* it sent without moving higher than another agent with a smaller domain (termed in the figure *ABT\_DO\_highest*).

Figures 14 and 15 present the results for the three heuristics using temporal *Nogoods* and domain sizes with standard *ABT\_DO* and the combined version of Retroactive *ABT\_DO* with the *Min-domain* heuristic. The three proposed heuristics run faster than standard *ABT\_DO* with a *Nogood\_triggered* heuristic (which was an order of magnitude faster than *ABT\_DO* using standard *Min-domain* heuristic). This result confirms our analytic explanation for the failure of the standard *Min-*

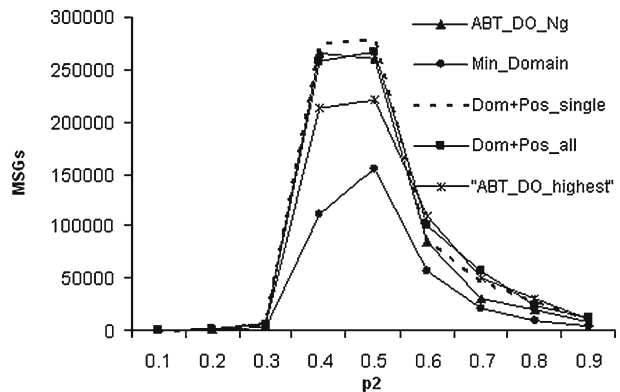
**Fig. 13** Number of messages sent by retroactive *ABT\_DO* with different limits on *Nogood* size ( $p_1 = 0.4$ )



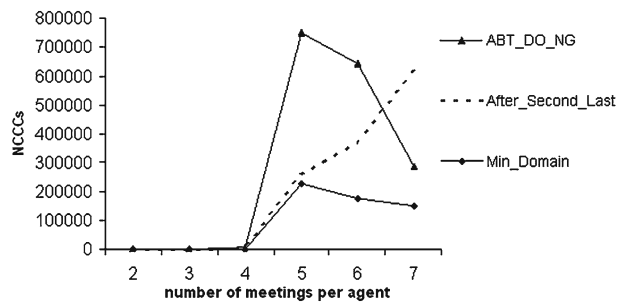
**Fig. 14** Non concurrent constraints checks performed by *ABT\_DO* with a min-domain heuristic using temporal Nogoods ( $p_1 = 0.4$ )



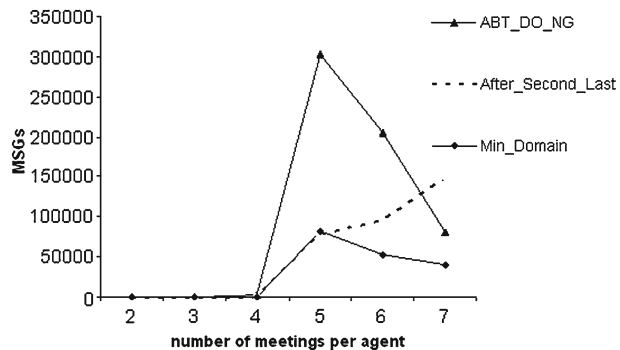
**Fig. 15** Number of messages sent by *ABT\_DO* with a min-domain heuristic using temporal Nogoods ( $p_1 = 0.4$ )



**Fig. 16** Non concurrent constraint checks performed by retroactive *ABT\_DO* and *ABT\_DO* on random course scheduling problems



**Fig. 17** Number of messages sent by retroactive *ABT\_DO* and *ABT\_DO* on random course scheduling problems



*domain* heuristic when used in *ABT\_DO* in Section 5. When agents can be moved to a higher position but not higher than an agent with assignments that caused removal of assignments from its domain, the heuristic is more successful by orders of magnitude. However, none of these heuristics run faster than the combined version of Retroactive *ABT\_DO* with the *Min-domain* heuristic. This can be explained by the amount of information which this heuristic is using which is much larger than in the retroactive version and thus has more chance to be invalid as a result of the asynchronicity of the algorithm.

When comparing between the three versions of this new type of heuristics, the fastest one is the one taking a single agent as high as possible according to the information it sent without passing agents with a smaller domain. The success of this heuristic is not surprising since it actually moves agents with a potential of a small domain as high as possible just like the Retroactive *Min-domain* heuristic. In fact, our statistics show that in one third of the cases, these two heuristics actually move the same agent (the *Nogood sending* agent).

In the next set of experiments, the successful versions of ABT with retroactive dynamic ordering were compared on realistic structured problems. The generated problems were course scheduling in which each variable assigned to an agent represents a single course which is taken by a number of students. Two variables are constrained if there is a student attending both courses. The constraints are arrival constraints, i.e. if the length of a course is  $t_1$  and the time to get from one course to the other is  $t_2$ , then the beginnings of each two constrained courses must satisfy:  $ct_1 - ct_2 \geq t_1 + t_2$ . This problem is equivalent to the published *Meeting Scheduling Problems* in which each agent holds exactly one variable [7]. For a detailed description of how a large random benchmark of problems with these realistic properties can be produced the reader is referred to Gent and Walsh [7]. The results presented in Figs. 16 and 17 show clearly that the advantage of the retroactive heuristic which takes into account the domain sizes is more pronounced for structured DisCSPs. Furthermore, on tight problems, the performance of the version of retroactive *ABT\_DO* which does not take into account domain sizes deteriorates.

## 9 Discussion

The results in the previous section show clearly the relation between the examined heuristics and the *Min-domain* property of the generated search tree. A well known fact from centralized CSP algorithms [6, 10] and from *DisCSP* algorithms with a sequential assignment protocol [5] is that the *Min-domain* heuristic is very powerful and improves the run of the same algorithms using a static order. If we investigate the *Nogood-triggered* heuristic of Zivan and Meisels [29] we can see that in most cases this heuristic moves to higher priority, agents with smaller domains. This is because an agent whose domain was exhausted returns to its domain, after sending the *Nogood*, only the values in conflict with the assignment of the culprit agent. Thus, only a small number of values are returned to its domain. It is not surprising that this heuristic was found to be very successful in Zivan and Meisels [29]. On the other hand, when an agent is moved to a higher position than the agents in the *Nogood* it discovered, it must return additional values to its domain. This contradicts the

properties of the *Min-domain* heuristic and was found to perform poorly in practice. The case of  $k = 1$  did show an improvement since the last assignment in a detected *Nogood* is removed from the *Agent\_view* of the agent which found the *Nogood* anyway.

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 *Nogood* they have generated, which would result in returning more values to their domain. Since the agent moving to a higher position is not in conflict with the assignments of agents it has moved in front of, its move will not cause the loss of *Nogoods* and therefore the information it holds on the size of the current domains of these agents remains valid. The retroactive ordering version has improved the results of Zivan and Meisels [29] by a factor of 2. In the case of structured problems, this heuristic was found to improve the run of the standard *ABT\_DO* by an even larger factor.

## 10 Conclusion

A general scheme for ordering heuristics for asynchronous backtracking with dynamic agent ordering was presented. Within the general scheme an additional flexibility has been introduced. Moving of agents forward, which involves the use of larger storage for *Nogoods*. The flexibility of the heuristic is dependent upon the amount of memory that agents are allowed to use. However, moving agents to the highest position possible was found to deteriorate the performance of the algorithm. Larger storage for *Nogoods* (even exponential in the extreme case) was found to produce worse efficiency for search on random problems.

Our experimental study brings multiple evidence for the connection between the success of ordering heuristics in asynchronous backtracking, the validity of the information used by the heuristic and the *Min-domain* property. The validity of the information lies at the heart of the difference between heuristics for standard CSP search and for distributed constraints. It emphasizes the asynchronous nature of DisCSP search by the ABT algorithm. The best heuristic, moves to a higher priority only agents whose variable's domains are smaller than the agents whose priority they replace, but avoids the return of values to domains as a result of reordering. This brings to an extreme the exploitation of the *Min-domain* property and improves the run of the best heuristic reported so far [29] by a large factor.

## References

1. Baker, A. B. (1994). The hazards of fancy backtracking. In *Proceedings of the 12th national conference on artificial intelligence (AAAI '94)* (Vol. 1, pp. 288–293). Seattle, WA, USA: AAAI Press July 31–August 4.
2. Bessiere, C., & Regin, J. C. (1996). Mac and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Proc. CP 96* (pp. 61–75). Cambridge, MA.
3. Bessiere, C., Maestre, A., & Messeguer, P. (2001). Distributed dynamic backtracking. In *Proc. workshop on distributed constraint of IJCAI01*.
4. Bessiere, C., Maestre, A., Brito, I., & Messeguer, P. (2005). Asynchronous backtracking without adding links: A new member in the abt family. *Artificial Intelligence*, 161(1–2), 7–24, January.

5. Brito, I., & Meseguer, P. (2004). Synchronous, asynchronous and hybrid algorithms for discsp. In *Workshop on distributed constraints reasoning (DCR-04) CP-2004*. Toronto, September.
6. Dechter, R. (2003). *Constraint processing*. Morgan Kaufman.
7. Gent, I. P., & Walsh, T. (1999). *Csplib: A benchmark library for constraints*. Technical report, Technical report APES-09-1999. Available from <http://csplib.cs.strath.ac.uk/>. A shorter version appears in the Proceedings of the 5th international conference on principles and practices of constraint programming (CP-99).
8. Ginsberg, M. L. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1, 25–46.
9. Hamadi, Y. (2001). Distributed interleaved parallel and cooperative search in constraint satisfaction networks. In *Proc. intelligent agent technology, 2001 (IAT-01)*. Singapore.
10. Haralick, R. M., & Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14, 263–313.
11. Lamport, L. (1978). Time, clocks, and the ordering of events in distributed system. *Communication of the ACM*, 2, 95–114, April.
12. Lynch, N. A. (1997). *Distributed algorithms*. Morgan Kaufmann Series.
13. Meisels, A., & Zivan, R. (2007). Asynchronous forward-checking for DisCSPs. *Constraints*, 12(1), 131–150.
14. Meisels, A., Razgon, I., Kaplansky, E., & Zivan R. (2002). Comparing performance of distributed constraints processing algorithms. In *Proc. AAMAS-2002 workshop on distributed constraint reasoning DCR* (pp. 86–93). Bologna, July.
15. Nguyen, T., Sam-Hroud, D., & Faltings, B. (2004). Dynamic distributed backjumping. In *Proc. 5th workshop on distributed constraints reasoning DCR-04*. Toronto, September.
16. Prosser, P. (1996). An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81, 81–109.
17. Silaghi, M. C., & Faltings, B. (2005). Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence*, 161(1–2), 25–54, January.
18. Silaghi, M. C., Sam-Haroud, D., & Faltings, B. (2001). Hybridizing abt and awc into a polynomial space, complete protocol with reordering. Technical Report, 01/#364, EPFL, May.
19. Silaghi, M. C. (2006). Generalized dynamic ordering for asynchronous backtracking on discsp. In *AAMAS 2006, DCR workshop*. Hakodate, Japan.
20. Smith, B. M., & Grant, S. A. (1998). Trying harder to fail first. In *European conference on artificial intelligence* (pp. 249–253).
21. Smith, B. M. (1996). Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81, 155–181.
22. Solotorevsky, G., Gudes, E., & Meisels, A. (1996). Modeling and solving distributed constraint satisfaction problems (dcsp). In *Constraint Processing-96 (short paper)* (pp. 561–2). Cambridge, Massachusetts, USA, October.
23. Yokoo, M., & Hirayama, K. (2000). Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents & Multi-Agent System*, 3, 198–212.
24. Yokoo, M., Durfee, E. H., Ishida, T., & Kuwabara, K. (1998). Distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Data and Knowledge Engineering*, 10, 673–685.
25. Yokoo, M. (1995). Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceedings of the 1st international conference on constraint programming* (pp. 88 – 102). France: Cassis.
26. Yokoo, M. (2000). *Distributed constraint satisfaction: Foundation and cooperation in multi agent systems*. Springer.
27. Zivan, R., & Meisels, A. (2003). Synchronous vs asynchronous search on discsp. In *Proc. 1st European workshop on multi agent system, EUMAS*. Oxford, December.
28. Zivan, R., & Meisels, A. (2005). Dynamic ordering for asynchronous backtracking on DisCSPs. In *CP-2005* (pp. 32–46). Sigtes (Barcelona), Spain.
29. Zivan, R., & Meisels, A. (2006a). Dynamic ordering for asynchronous backtracking on DisCSPs. *Constraints*, 11(2,3), 179–197.
30. Zivan, R., & Meisels, A. (2006b). Message delay and asynchronous discsp search. *Archives of Control Sciences*, 16(2), 221–242.