

# Retroactive ordering heuristics for Asynchronous Backtracking on DisCSPs\*

Roie Zivan, Moshe Zazone and Amnon Meisels

Department of Computer Science,  
Ben-Gurion University of the Negev,  
Beer-Sheva, 84-105, Israel  
{zivanr,moshezaz,am}@cs.bgu.ac.il

**Abstract.** A new type of ordering heuristics for dynamic ordering asynchronous backtracking (*ABT\_DO*) on *DisCSPs* is presented. Agents can be moved to a position that is higher than that of the target of the backtrack (*Nogood*). This new type of heuristics do not follow the restrictions on the heuristics of previous versions of *ABT\_DO*.

The flexibility of the new type of heuristics is dependent on the size of additional *Nogood* storage agents are allowed to keep. This size is defined by a parameter  $k$  which limits the size of stored *Nogoods*.

The performance of the retroactive ordered *ABT* is found to be *worse* when larger *Nogood* storage is used. The best performing version is the one that uses a specific *min-domain* heuristic with *no additional* storage of *Nogoods*.

## 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. [Yok00,SGM96]). 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 [BMBM05].

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 [YDIK98,Yok00] and was developed further and studied in [Ham01,BMM01,SF05,BMBM05]. Agents in the *ABT* algorithms perform assignments asynchronously against their current view of the system's state. The

---

\* The research was supported by the Lynn and William Frankel Center for Computer Science, and by the Paul Ivanier Center for Robotics.

method performed by each agent is in general simple. Later versions of *ABT* use polynomial space memory and perform dynamic backtracking [BMM01,BMBM05]. 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 by [Yok95], Asynchronous Weak Commitment (*AWC*). According to [Yok00], *AWC* outperforms *ABT* on specific applications (N-queens, Graph-coloring). However, in order to be complete, *AWC* uses exponential space for storing *Nogoods* which makes it impractical for solving hard instances of even small *DisCSPs*.

An attempt to combine *ABT* with *AWC* was reported by [SSHF01]. In order to perform asynchronous finite reordering operations [SSHF01] suggest that the reordering operation will be performed by abstract agents. The results presented in [SSHF01] show minor improvements to static order *ABT*.

A simple algorithm for dynamic ordering in asynchronous backtracking, *ABT\_DO* was presented in [ZM05]. 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 according to the current, most updated order they hold. The restrictions on the changes of orderings of agents are very tight in *ABT\_DO*. Each agent can change the order of agents with 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. The method of time-stamping for defining the most updated order is the same that was used in [NSHF04] for choosing the most updated partial assignment.

The results presented in [ZM05] show that the performance of *ABT\_DO* is highly dependent on the selected heuristic. Classic Heuristics which in centralized algorithms and in distributed algorithms using a sequential assignment protocol show large improvement over the static order case, were found to only slightly improve the run of Asynchronous Backtracking. The only heuristic which achieved a significant improvement was a heuristic inspired by *Dynamic Backtracking* [Gin93] in which the agent which sends a *Nogood* is advanced in the new order to be right after the agent to whom the *Nogood* was sent.

The present paper focuses on more flexible heuristics, which violate the restrictions on the ordering of agents in [ZM05]. We study changes of order that move agents *backwards*, replacing agents that were ahead of them including the first agent. This new type of heuristics we term *retroactive heuristics*. 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 on the size of *Nogood* storage which is predefined for agents. Agents are limited to store *Nogoods* equal or smaller than a predefined size  $k$ . When the *Nogood* found 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 [Yok00]. In the case of *AWC* one has  $k = n$  and all *Nogoods* are stored. In addition, the *AWC* algorithm uses specific heuristic that is different than

the best heuristic suggested in the present study. In *AWC* the *Nogood* generator is always moved to the first place in the algorithm.

The results presented in this paper show that moving the *Nogood* sender as high as possible in the priority order is not always desirable. In Section 6 we 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 it is considered to be moved in front of. Moving an agent before the agents which are included in the *Nogood* actually *enlarges* its domain. Therefore, the best heuristic suggested 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*. The agents are moved to a higher position only if their domain is smaller than the agents they pass on the way up. Preliminary results show that it is enough to place the *Nogood* generator right before the *Nogood* receiver in case it has a smaller domain, in order to improve the best results of [ZM05] 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 3. Section 4 present the general scheme of retroactive heuristics for *ABT\_DO* and an outline of a correctness proof. An extensive experimental evaluation, which compares standard and retroactive heuristics of *ABT\_DO* is in Section 6. The experiments were conducted on randomly generated *DisCSPs*. Section 7 presents a discussion of the relation between the 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 [YDIK98,SGM96]. 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 [BMBM05]. 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" [Yok00,BMBM05].

The following assumptions are routinely made in studies of *DisCSPs* and are assumed to hold in the present study [Yok00,BMBM05].

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 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, each time it replaces its assignment. This way the sending of an ordering proposal message always coincides with an **ok?** message. 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

```

when received (ok?, (xj, dj):
1. add (xj, dj) to agent_view;
2. remove inconsistent nogoods;
3. check_agent_view;

when received (order, received_order):
1. if (received_order is more updated than Current_order)
2.   Current_order ← received_order;
3.   remove inconsistent nogoods;
4.   check_agent_view;

when received (nogood, xj, nogood)
1. if (nogood contains an agent xk with lower priority than xi)
2.   send (nogood, (xi, nogood)) to xk;
3.   send (ok?, (xi, current_value) to xj;
4. else
5.   if (nogood consistent with
        {Agent_view ∪ current_assignment})
6.     store nogood;
7.     if (nogood contains an agent xk that is not its neighbor)
8.       request xk to add xi as a neighbor;
9.       add (xk, dk) to agent_view;
10.    check_agent_view;
11.  else
12.    send (ok?, (xi, current_value)) to xj;

```

**Fig. 1.** The ABT\_DO algorithm (first part)

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* 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 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* [BMBM05].

Figures 1 and 2 present the code of asynchronous backtracking with dynamic ordering (*ABT\_DO*).

When an **ok?** message is received (first procedure in Figure 1), 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 Figure 1). If so, the received order is

```

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. 2.** The two procedures of the ABT\_DO algorithm

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 Figure 1) 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 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** (Figure 2) is the same as in standard *ABT*. The *Nogood* is resolved and the result is sent to the lowest priority agent in the *Nogood*, according to the *Current\_order*.

Procedure **check\_agent\_view** (Figure 2) is very similar to the same procedure in 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, according to the algorithms rules and the heuristic used, 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.

## 4 Retroactive ordering heuristics for Dynamic Ordered 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

```

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, ( $x_i$ , 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?, ( $x_i$ , current\_value)) to  $x_j$ ;

```

**Fig. 3.** Retroactive ABT.DO algorithm (first part)

higher priority than the agent which replaces its assignment. The best heuristic which was presented in [ZM05] 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*, a detection and sending of a *Nogood* triggers reordering operation. In contrast to *ABT.DO*, the new ordering can be suggested either by the *Nogood* generator or by the *Nogood* receiver (but not by both). 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. Every agent whose assignment was included in the *Nogood* and the *Nogood* generator is moved in front of, 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 higher in the priority order. If the *Nogood* created is larger than  $k$ , they can be moved up to the place that is right after the last agent in the created *Nogood* that will not change its assignment i.e. the agent whose place on the *Nogood* is before the *Nogood* receiver. If the *Nogood* is smaller or equal to  $k$ , the sending agent can be moved to be before all the participants in the *Nogood* and the *Nogood* is sent to 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*.

```

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. if( $nogood.size > K$ )
7.    $new\_position \leftarrow$  choose position lower than  $x_l$ 
       where  $x_l$  has the second lowest priority in  $nogood$ ;
8.   send (nogood,  $x_i$ ,  $nogood$ ) to  $x_j$ ;
9. else if( $is\_new(nogood)$ )
10.   $new\_position \leftarrow unlimited$ 
11.  send (nogood,  $x_i$ ,  $nogood$ ) to all agents in  $nogood$ ;
12.  store sent  $nogood$ ;
13.  $Current\_order \leftarrow$  choose_new_order( $x_i$ )
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.      $current\_value \leftarrow d$ ;
7.     send (ok?, ( $x_i, d$ )) to  $neighbors$ ;

```

**Fig. 4.** Retroactive ABT\_DO algorithm (first part)

Figures 3 and 4 present changes in the code from standard *ABT\_DO*, that are needed in order to implement retroactive heuristics. The difference in the code performed when a *Nogood* is received 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 larger 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 *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*. For simplicity of presentation we assume that only the *Nogood* generator is allowed to change order.

Procedure **backtrack** is largely changed in the retroactive heuristic version of *ABT\_DO* (Figure 4). 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 its self to an unlimited position in the new order. 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 reattempts to assign its variable by calling **check\_agent\_view**.

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

## 5 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 (as proven in [ZM05]) 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 both of these cases the sent order is legal in the updated state.

The second change in the code for  $k = 0$  is that in *Retroactive ABT\_DO* a *Nogood* sender can move 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.  $\square$

**Theorem 2.** *RetroactiveABT.DO is correct when  $n \geq k > 0$ .*

In order to prove that *RetroactiveABT.DO* is correct for the case that  $n \geq k > 0$  we need to show that infinite loops cannot occur. That a *Nogood* cannot be sent twice unless there are higher priority assignment changes. 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 case of *Nogoods* larger than  $k$ , are similar to the case when  $k = 0$  until a shorter *Nogood*, smaller or equal to  $k$ , is generated. This *Nogood* can cause an assignment replacement which will cause the long *Nogood* to be discarded. However, the shorter *Nogood* is kept indefinitely, therefore the same assignment which caused the creation of the long *Nogood* cannot be duplicated.  $\square$

## 6 Experimental Evaluation

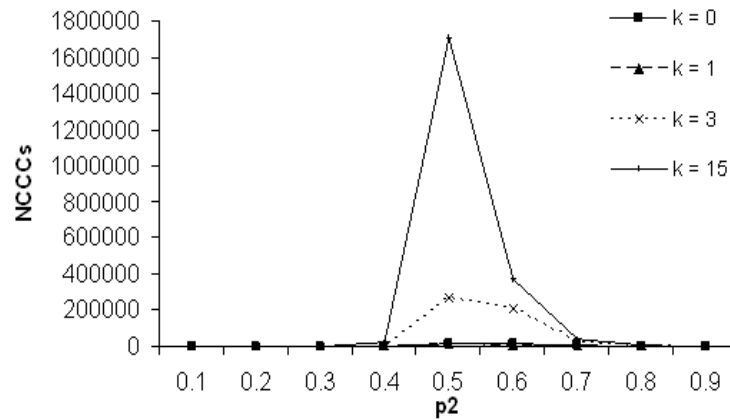
The common approach in evaluating the performance of distributed algorithms is to compare two independent measures of performance - time, in the form of steps of computation [Lyn97,Yok00], and communication load, in the form of the total number of messages sent [Lyn97].

Non concurrent steps of computation, are counted by a method similar to the clock synchronization algorithm of [Lam78]. 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 [ZM06a]. 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 [Lam78]. If instead of steps of computation, the number of non concurrent constraints check is counted (*NCCCs*), then the local computational effort of agents in each step is measured [MRKZ02,ZM06b].

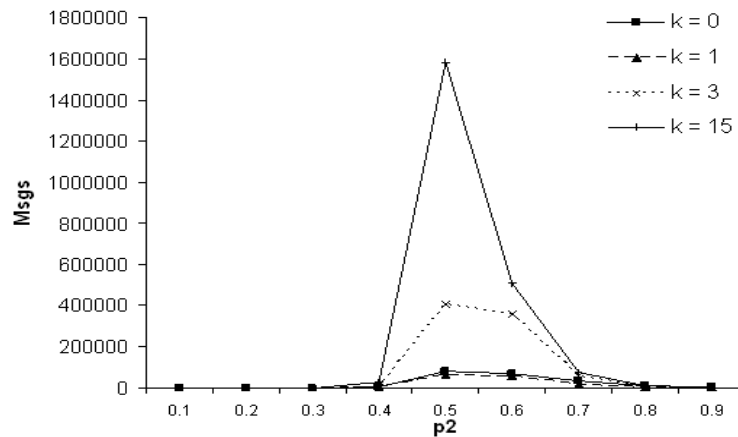
Experiments were conducted on random networks of constraints of  $n$  variables,  $k$  values in each domain, a constraints density of  $p_1$  and tightness  $p_2$  (which are commonly used in experimental evaluations of CSP algorithms [Smi96,Pro96]). The first set of experiments was conducted on networks with 15 agents ( $n = 15$ ) each holding exactly one variable, 10 values for each variable ( $k = 10$ ) constraints density  $p_1 = 0.4$ . 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. The second set of experiments was conducted on larger problems ( $n = 20$ ) with two different constraints densities  $p_1 = 0.4$  and  $p_1 = 0.7$ .

In the first set of experiments the size limit for keeping *Nogoods* is varied. A *Nogood* generator which created a *Nogood* 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 the case of  $k = 0$  the resulting algorithm is standard *ABT\_DO*. Figure 5 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 Figure 6. A closer look at the results of the smaller values of  $k$  show a small improvement of the  $k = 1$  version over the version with  $k = 0$ .

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 domain of the agents which are moved forward 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



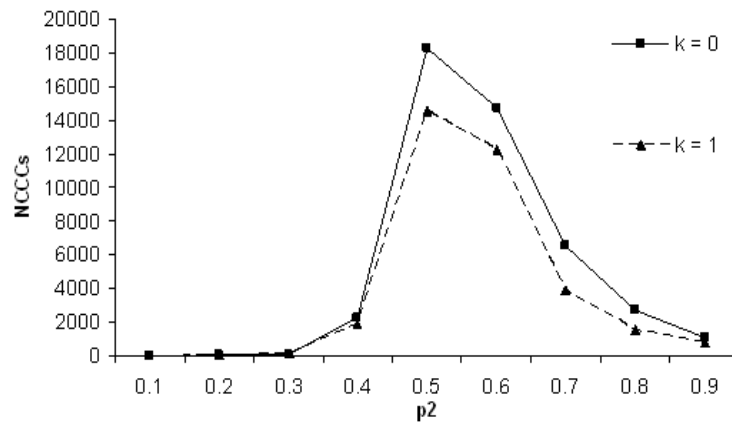
**Fig. 5.** Non concurrent constraints checks performed by Retroactive *ABT\_DO* with different limits on Nogood size ( $p_1 = 0.4$ ).



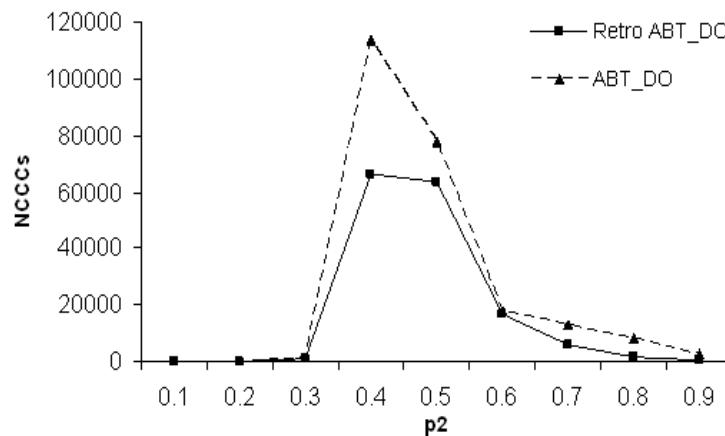
**Fig. 6.** Number of messages sent by Retroactive *ABT\_DO* with different limits on Nogood size ( $p_1 = 0.4$ ).

agents whose assignment was included in the generated *Nogood* it must return more values to its domain (the values included in the assignment of the *Nogood*). This of course did not happen in 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 actually results in moving forward an agent with a larger domain.

In order to confirm the dependency of the performance the size of the current domain of the moved agents, a second set of experiments was performed. In the second set we compared *ABT\_DO* with *ABT\_DO* with a retroactive heuristic in which agents are not allocated any additional *Nogood* storage. A *Nogood* generator moves itself to be right before the culprit agent *only if its current domain is smaller* than that of the culprit agent. Otherwise, it places itself right after the culprit agent.



**Fig. 7.** Non concurrent constraints checks performed by Retroactive *ABT\_DO* with small size of Nogood storage ( $p_1 = 0.4$ ).

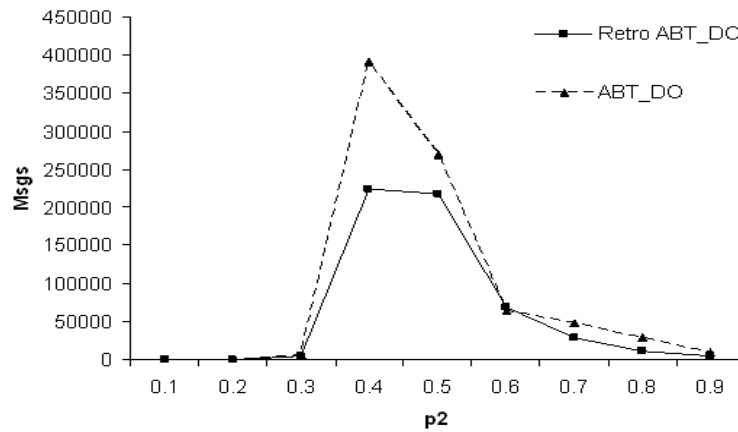


**Fig. 8.** Non concurrent constraints checks performed by Retroactive *ABT\_DO* and *ABT\_DO* on low density DisCSPs ( $p_1 = 0.4$ ).

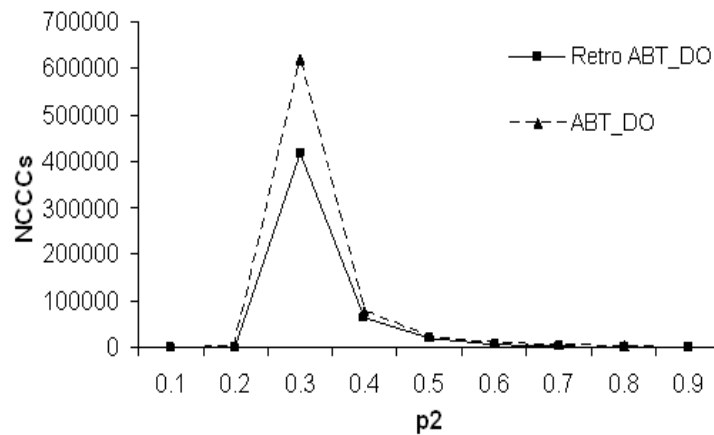
Figure 8 presents the results in *NCCCs* for *ABT\_DO* and Retroactive *ABT\_DO* with the above heuristic. The retroactive version of *ABT\_DO* improves the run-time performance of *ABT\_DO*. Similar results for different measures and two different densities are presented in Figures 8, 9, 10 and 11.

## 7 Discussion

The results in the previous section show clearly that the property of the heuristic to move an agent *backwards* is not enough to make it successful. While moving agents to the highest priority showed a minor improvement in performance over standard *ABT\_DO* when  $k = 1$ , it actually deteriorates for larger  $k$  values. A well known fact from cen-

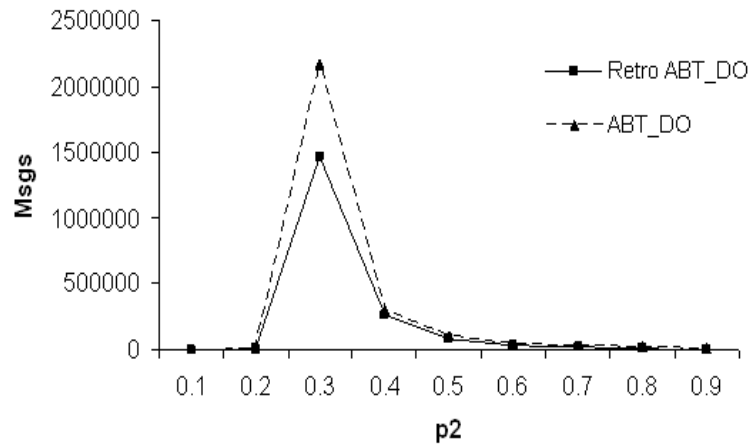


**Fig. 9.** Number of messages sent by Retroactive *ABT\_DO* and *ABT\_DO* on low density DisCSPs ( $p_1 = 0.4$ ).



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

tralized *CSP* algorithms [HE80,Dec03] and from *DisCSP* algorithms with a sequential assignment protocol [BM04] is that the min-domain heuristic is very powerful and improves by a large factor the run of the same algorithms using a static order. If we look at the *Nogood-triggered* heuristic of [ZM05] we can see that this heuristic actually moves to higher priority, agents which in most cases have smaller domains. Since a sender of a *Nogood* is an agent whose domain is empty, removing one assignment from its *Agent\_view* will in many cases return only a small number of values to its domain. Therefore, it is not surprising that this heuristic was found to be very successful in [ZM05] (although this explanation was not given there). On the other hand, when an agent is moved to a higher priority than the agents in the *Nogood* it discovered, it actually ignores the small domain it created and must return to its domain all the val-



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

ues which were not eliminated yet. 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 which found the *Nogood* anyway. Therefore, moving to a higher priority would not change the domain size. The preliminary results presented in the previous section show the potential of moving the *Nogood* sender to the highest priority in which it does not need to restore any additional values to its domain. Simply by moving the agent to a higher position than the culprit agent in the cases in which its domain is smaller has improved the results of the best heuristic of standard *ABT\_DO* by a factor of 2.

## 8 Conclusion

A general scheme for introducing retroactive heuristics into Asynchronous Backtracking with dynamic agent ordering was presented. The flexibility of the heuristic is dependent on the amount of memory that agents are allowed to use. However, moving agents to the most highest position possible was found to deteriorate the performance of the algorithm. The best heuristic which exploits the *min-domain* property improves the run of the best heuristic reported in [ZM05].

## References

- [BM04] I. Brito and P. Meseguer. Synchronous, asynchronous and hybrid algorithms for discsp. In *Workshop on Distributed Constraints Reasoning(DCR-04) CP-2004*, Toronto, September 2004.
- [BMBM05] C. Bessiere, A. Maestre, I. Brito, and P. Meseguer. Asynchronous backtracking without adding links: a new member in the abt family. *Artificial Intelligence*, 161:1-2:7–24, January 2005.

- [BMM01] C. Bessiere, A. Maestre, and P. Messeguer. Distributed dynamic backtracking. In *Proc. Workshop on Distributed Constraint of IJCAI01*, 2001.
- [Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufman, 2003.
- [Gin93] M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.
- [Ham01] Y. Hamadi. Distributed interleaved parallel and cooperative search in constraint satisfaction networks. In *Proc. IAT-01*, Singapore, 2001.
- [HE80] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
- [Lam78] L. Lamport. Time, clocks, and the ordering of events in distributed system. *Communication of the ACM*, 2:95–114, April 1978.
- [Lyn97] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series, 1997.
- [MRKZ02] A. Meisels, I. Razgon, E. Kaplansky, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, pages 86–93, Bologna, July 2002.
- [NSHF04] T. Nguyen, D. Sam-Hroud, and B. Faltings. Dynamic distributed backjumping. In *Proc. 5th workshop on distributed constraints reasoning DCR-04*, Toronto, September 2004.
- [Pro96] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.
- [SF05] M. C. Silaghi and B. Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence*, 161:1-2:25–54, January 2005.
- [SGM96] G. Sotolorevsky, E. Gudes, and A. Meisels. Modeling and solving distributed constraint satisfaction problems (dcsp). In *Constraint Processing-96, (short paper)*, pages 561–2, Cambridge, Massachusetts, USA, October 1996.
- [Smi96] B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155 – 181, 1996.
- [SSHF01] M. C. Silaghi, D. Sam-Haroud, and B. Faltings. Hybridizing abt and awc into a polynomial space, complete protocol with reordering. Technical Report 01/#364, EPFL, May 2001.
- [YDIK98] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. on Data and Kn. Eng.*, 10:673–685, 1998.
- [Yok95] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proc. 1st Intrnat. Conf. on Const. Progr.*, pages 88 – 102, Cassis, France, 1995.
- [Yok00] M. Yokoo. Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents & Multi-Agent Sys.*, 3:198–212, 2000.
- [ZM05] R. Zivan and A. Meisels. Dynamic ordering for asynchronous backtracking on discsp. In *CP-2005*, pages 32–46, Sigtes (Barcelona), Spain, 2005.
- [ZM06a] R. Zivan and A. Meisels. Message delay and asynchronous discsp search. *Archives of Control*, (accepted for publication), 2006.
- [ZM06b] R. Zivan and A. Meisels. Message delay and discsp search algorithms. *Annals of Mathematics and Artificial Intelligence(AMAI)*, (accepted for publication), 2006.