Dynamic Ordering for Asynchronous Backtracking on DisCSPs

Roie Zivan and Amnon Meisels[⋆]
{zivanr,am}@cs.bgu.ac.il

Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel

Abstract. An algorithm that performs asynchronous backtracking on distributed $CSPs$, with dynamic ordering of agents is proposed, ABT_DO. Agents propose reorderings of lower priority agents and send these proposals whenever they send assignment messages. Changes of ordering triggers a different computation of $Nogoods$. The dynamic ordered asynchronous backtracking algorithm uses polynomial space, similarly to standard ABT.
The ABT_DO algorithm with three different ordering heuristics is compared to standard ABT on randomly generated $DisCSPs$. A *Nogood-triggered* heuristic, inspired by dynamic backtracking, is found to outperform static order ABT by a large factor in run-time and improve the network load.

1 Introduction

Distributed constraint satisfaction problems (*DisCSP*s) 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. [Yokoo, 2000,Solotorevsky *et al.*, 1996]). 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 [Bessiere *et al.*, 2005].

Distributed CSPs are an elegant model for many every day combinatorial problems that are distributed by nature. Take for example a large hospital that is composed of many wards. Each ward constructs a weekly timetable assigning its nurses to shifts. The construction of a weekly timetable involves solving a constraint satisfaction problem for each ward. Some of the nurses in every ward are qualified to work in the *Emergency Room*. Hospital regulations require a certain number of qualified nurses (e.g. for Emergency Room) in each shift. This imposes constraints among the timetables of different wards and generates a complex Distributed CSP [Solotorevsky *et al.*, 1996].

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

---

[⋆] Supported by the Lynn and William Frankel center for Computer Sciences.

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. ABT was first presented by $Yokoo$ [Yokoo *et al.*, 1998,Yokoo, 2000] and was developed further and studied in [Hamadi, 2001,Bessiere *et al.*, 2001,Silaghi and Faltings, 2005,Bessiere *et al.*, 2005]. Agents in the ABT algorithms perform assignments asynchronously against their current view of the systems state. The method performed by each agent is in general simple. Later versions of ABT use polynomial space memory and perform dynamic backtracking [Bessiere *et al.*, 2001,Bessiere *et al.*, 2005]. The versions of asynchronous backtracking presented in all of the above studies use a static priority order among all agents.

In centralized $CSPs$, dynamic variable ordering is known to be an effective heuristic for gaining efficiency [Dechter, 2003]. Recent studies have shown that the same is true for algorithms which perform sequential (synchronous) assignments in Distributed $CSPs$ [Nguyen *et al.*, 2004,Brito and Meseguer, 2004]. These studies suggest heuristics of agent/variable ordering and empirically show large gains in efficiency over the same algorithms performing with static order. These results are the basic motivation for exploring the possibilities for dynamic reordering of asynchronous backtracking.

In [Hamadi, 2001] the authors present a distributed ordering algorithm, according to the properties of the constraints graph. Once the order is determined, the asynchronous backtracking algorithm uses this fixed order.

An asynchronous algorithm with dynamic ordering was proposed by [Yokoo, 1995], Asynchronous Weak Commitment (AWC). According to [Yokoo, 2000], AWC outperforms ABT. However, in order to be complete, AWC uses exponential space which makes it impractical for solving hard instances of even small $DisCSPs$.

An attempt to combine ABT with AWC was reported by [Silaghi *et al.*, 2001]. In order to perform asynchronous finite reordering operations [Silaghi *et al.*, 2001] suggest that the reordering operation will be performed by abstract agents. The results presented in [Silaghi *et al.*, 2001] show minor improvements to static order ABT.

The present paper proposes a simple algorithm for dynamic ordering in asynchronous backtracking, ABT_DO that uses polynomial space, as standard ABT. In the proposed 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. Each order is time-stamped according to agents assignment. The method of time-stamp for defining the most updated order is the same that is used in [Nguyen *et al.*, 2004] 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. Each agent can change the order of all agents with lower priority. An agent can propose an order change each time it replaces its assignment.

Having established a correct algorithm for dynamic variable ordering in ABT, one needs to investigate ordering heuristics. Surprisingly, some of the heuristics which are very effective for sequential assignments distributed algorithms, do not improve the run-time of ABT. It turns out that an ordering heuristic, based on *Dynamic Backtracking* [Ginsberg, 1993], is very successful (see Section 6).

Distributed $CSPs$ are presented in Section 2. A description of the standard ABT algorithm is presented in Section 3. Asynchronous backtracking with dynamic ordering (ABT_DO) is presented in Section 4. Section 5 introduces a correctness and completeness proof for ABT_DO. An extensive experimental evaluation, which compares ABT to ABT_DO with several ordering heuristics is in Section 6. The experiments were conducted on randomly generated $DisCSPs$.

2 Distributed Constraint Satisfaction

A distributed constraints network (or a distributed constraints satisfaction problem - *DisCSP*) is composed of a set of k agents $A_1, A_2, ..., A_k$. Each agent A_i contains a set of constrained variables $X_{i_1}, X_{i_2}, ..., X_{i_{n_i}}$. Constraints or **relations** R are subsets of the Cartesian product of the domains of the constrained variables. For a set of constrained variables $X_{i_k}, X_{j_l}, ..., X_{m_n}$, with domains of values for each variable $D_{i_k}, D_{j_l}, ..., D_{m_n}$, the constraint is defined as $R \subseteq D_{i_k} \times D_{j_l} \times ... \times D_{m_n}$. 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* , the agents are connected by constraints between variables that belong to different agents [Yokoo *et al.*, 1998,Solotorevsky *et al.*, 1996]. In addition, each agent has a set of constrained variables, i.e. a *local constraint network*.

An assignment (or a label) is a pair $< var, val >$, 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* 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.

The following assumptions are routinely made in studies of $DisCSP$s and are assumed to hold in the present study [Yokoo, 2000,Bessiere *et al.*, 2005].

1. All agents hold exactly one variable.
2. The amount of time that passes between the sending and the receiving of a message is finite.
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 [Yokoo, 2000,Bessiere *et al.*, 2005]. 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

```
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

on its former inconsistent $Agent\_view$ and makes another attempt to assign its variable [Yokoo, 2000,Bessiere *et al.*, 2005].

The code of the Asynchronous Backtracking algorithm (ABT) is presented in figure 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 (an **ok?** message [Yokoo, 2000]), it updates its $Agent\_view$ with the received assignment and if needed replaces its own assignment, to achieve consistency (first procedure in Figure 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 [Yokoo, 2000]) 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 $Nogood$ [Yokoo, 2000]. 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 Figure 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 Figure 1).

The performance of ABT can be improved immensely by requiring agents to read all messages they receive before performing computation [Yokoo, 2000,Bessiere *et al.*, 2005]. This technique was found to improve the performance of *Asynchronous Backtracking* on the harder instances of randomly generated Distributed CSPs by a large factor [Zivan and Meisels, 2003,Brito and Meseguer, 2004].

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.*, 2005]. 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

For simplicity of presentation we assume that agents send **order** messages to all lower priority agents. In the more realistic form of the algorithm, agents send **order** messages only to their lower priority *neighbors*. Both versions are proven correct in section 5.

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. 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.

The counters attached to each agent ID in the $order$ list form a time-stamp. Initially, all time-stamp counters are zero and all agents start with the same $Current\_Order$. Each agent that proposes a new order changes the order of the pairs in its 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.

Consider an example in which agent A_2 holds the following $Current\_order$: $(1,4)(2,3)(3,1)(4,0)(5,1)$. There are 5 agents $A_1...A_5$ and they are ordered according to their IDs from left to right. After replacing its assignment it changes the order to: $(1,4)(2,4)(4,0)(5,0)(3,0)$. In the new order, agent A_1 which had higher priority than A_2 in the previous order keeps its place and the value of its counter does not change. A_2 also keeps its place and the value of its counter is incremented by one. The rest of the agents, which have lower priority than A_2 in the previous order, change places as long as they are still located lower than A_2. The new order for these agents is A_4, A_5, A_3 and their counters 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, beside 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 the agents IDs since the agent that performs the change does not change its own position and the positions of higher priority agents. In the above example the common prefix includes agents A_1 and A_2. Since the agent proposing the new order increases its own counter, when two different orders are compared, at lease one of the time-stamp counters in the common prefix is different between the two orders. The more updated order is the one for which the first different counter in the common prefix is larger. In the example above, any agent which will receive the new order will know it is more updated than the previous order since the first pair is identical, but the counter of the second pair is larger.

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 vise 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

```
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, $(x_i, nogood)$) to $x_k$;
3.      send (ok?, $(x_i, current\_value)$ to $x_j$;
4.  else
5.      if ($nogood$ consistent with $\{Agent\_view \cup current\_assignment\}$ )
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?, $(x_i, current\_value)$) to $x_j$;
```

Fig. 2. The ABT_DO algorithm (first part)

sends an **ok?** message to the sender of the $Nogood$. This is a similar operation to that performed in standard ABT for any unaccepted $Nogood$.

Figures 2 and 3 present the code of asynchronous backtracking with dynamic ordering (ABT_DO).

When an **ok?** message is received (first procedure in Figure 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 Figure 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 Figure 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 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 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)

Procedure **backtrack** (Figure 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** (Figure 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 Correctness of ABT_DO

In order to prove the correctness of the ABT_DO algorithm we first establish two facts by proving the following lemmas:

Lemma 1 *The highest priority agent in the initial order remains the highest priority agent in all proposed orders.*

The proof for Lemma 1 is immediate from the two rules of reordering. Since no agent can propose a new order which changes the priority of higher priority agents and its own priority, no agent including the first can move the highest priority agent to a lower position. □

Lemma 2 *When the highest priority agent proposes a new order, it is more up to date than all previous orders.*

This proof is again immediate. In all previous orders the time-stamp counter of the first agent is smaller than the counter of the time-stamp counter of the first agent in the new proposed order. □

To prove correctness of a search algorithm for $DisCSPs$ one needs to prove that it is sound, complete and that it terminates. ABT_DO, like ABT, reports a solution when all agents are idle and no messages are sent. Its soundness follows from the soundness of ABT [Bessiere *et al.*, 2005]. One point needs mentioning. Since no messages are traveling in the system in the idle state, all overriding messages have arrived at their destinations. This means that for every pair of constraining agents an agreement about their pairwise order has been achieved. One of each pair of constraining agents checks their constraint and no messages mean no violations, as in the proof for ABT [Bessiere *et al.*, 2005].

To prove the completeness and termination of ABT_DO we use induction on the number of agents (i.e. number of variables) in the $DisCSP$. For a single agent $DisCSP$ the order is static therefore the completeness and termination of ABT implies the same for ABT_DO. Assume ABT_DO is complete and terminates for every $DisCSP$ with k agents where $k < n$. Consider a $DisCSP$ with n agents. According to Lemma 1 the agent with the highest priority in the initial order will not change its place. The highest priority agent assigns its variable for the first time and sends it along with its order proposal to other agents. The remaining $DisCSP$ has $n - 1$ agents and its initial order is that proposed by the first agent (all other orders are discarded according to Lemma 2). By the induction assumption the remaining $DisCSP$ is complete and terminates. If a solution to the induced $DisCSP$ is found, this means that the lower priority $n - 1$ agents are idle. So is the first (highest priority) agent since none of the others sends it any message. If a solution is not found, by the $n - 1$ lower priority agents, a single assignment $Nogood$ will be sent to the highest priority agent which will cause it to replace its assignment. The new assignment of the first agent and the new order proposed will induce a new $DisCSP$ of size $n-1$. The search on this new $DisCSP$ of size $n-1$ is also complete and terminates according to the induction assumption. The number of induced $DisCSPs$, created by the assignments of the highest priority agent is bound by the size of its domain. Therefore, the algorithm will terminate in a finite time.

The algorithm is complete since a solution to the $DisCSP$ must include one of the highest priority agent value assignments, which means that one of the induced $DisCSPs$ includes a solution iff the original $DisCSP$ includes a solution. This completes the correctness proof of ABT_DO □

If the network model, or privacy restraints, enable agents to communicate only with their neighbors in the constraints network, some small changes are needed in order to keep the algorithm correct. First, agents must be allowed to change only the order of lower priority $neighbors$. This means that the method **choose_new_order**, called in line 7 of procedure **check_agent_view**, changes the order by switching between the position of lower priority neighbors and leaving other lower priority agent at their current position. Second, whenever an updated order message is received, an agent informs its neighbors of its new $Current\_order$.

In order to prove that the above two changes do not affect the correctness of the algorithm we first establish the correctness of Lemmas 1 and 2 under these changes. Lemma 1 is not affected by the change since the rules for changing agents positions have become more strict, and still do not allow to change the position of higher priority agents. Lemma 2 holds because the time-stamp mechanism which promises its correctness has not changed. These Lemmas are the basis for the correctness of the induction which proves the algorithm is complete and terminates. However, we still need to prove the algorithm is sound. One of the assumptions that our soundness proof dependent on was that an idle state of the system would mean that every constrained couple of agents agrees on the order between them. This claim might not hold since the most up to date order is not sent to all agents. The following Lemma proves this claim is still true after the changes in the algorithm:

Lemma 3 *When the system reaches an idle state, every pair of constrained agents hold the same order.*

According to the changes described above, whenever one of the constrained agents receives an updated order message, it informs its neighbors. Therefore, all agents which have constraints with it will be notified and hold the updated order. If two agents are not informed with the most updated order, this would mean both of them are not lower priority neighbors of the reordering agent and as a result their current position in the order stays the same.

Lemma 3 implies that the algorithm is still sound according to our previous proof.□

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 [Lynch, 1997,Yokoo, 2000], and communication load, in the form of the total number of messages sent [Lynch, 1997].

Non concurrent steps of computation, are counted by a method similar to that of [Lamport, 1978,Meisels *et al.*, 2002]. 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 updates its counter to the largest value between its own counter and the counter value carried by the message. 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 [Lamport, 1978]. If instead of steps of computation, the number of non concurrent constraints check is counted ($NCCC$s), then the local computational effort of agents in each step is measured [Meisels *et al.*, 2002].

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 [Smith, 1996]). All three sets of experiments were conducted on networks with 20 agents ($n = 20$) each holding

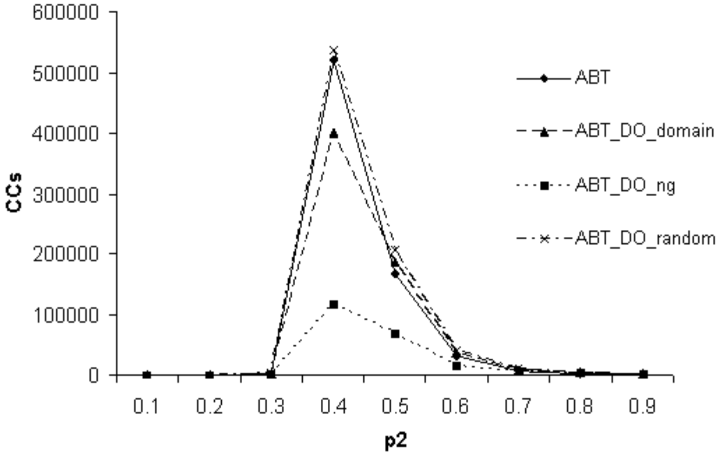

Fig. 4. Non concurrent constraints checks performed by ABT and ABT_DO using different order heuristics on low density DisCSPs ($p_1 = 0.4$).

exactly one variable, 10 values for each variable ($k = 10$) and two values of constraints density $p_1 = 0.4$ 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 ($p1$, $p2$) 50 different random problems were solved by each algorithm and the results presented are an average of these 50 runs.

ABT_DO is compared to the run of standard ABT. For ordering variables in ABT_DO three different heuristics were used.

1. Random: each time an agent changes its assignment it randomly orders all agents with lower priorities in its $Current\_order$.
2. Domain-Size: This heuristic is inspired by the heuristics used for sequential assigning algorithms in [Brito and Meseguer, 2004]. Domain sizes are calculated based on the fact that each agent that performs an assignment sends its current domain size to all other agents. Every agent that replaces an assignment, orders the lower priority agents according to their domain size from the smallest to the largest.
3. Nogood-Triggered: Agents change the order of the lower priority agents only when they receive a $Nogood$ which eliminates their current assignment. In this case the agent moves the sender of the $Nogood$ to be in front of all other lower priority agents. This heuristic was first used for dynamic backtracking in centralized $CSPs$ [Ginsberg, 1993].

Figure 4 presents the computational effort in number of non concurrent constraints checks to find a solution, performed by ABT and ABT_DO using the above three heuristics. The algorithms solve low density $DisCSPs$ with $p_1 = 0.4$. ABT_DO with random ordering does not improve the results of standard ABT. ABT_DO which uses domain sizes to order the lower priority agents performs slightly better than ABT.

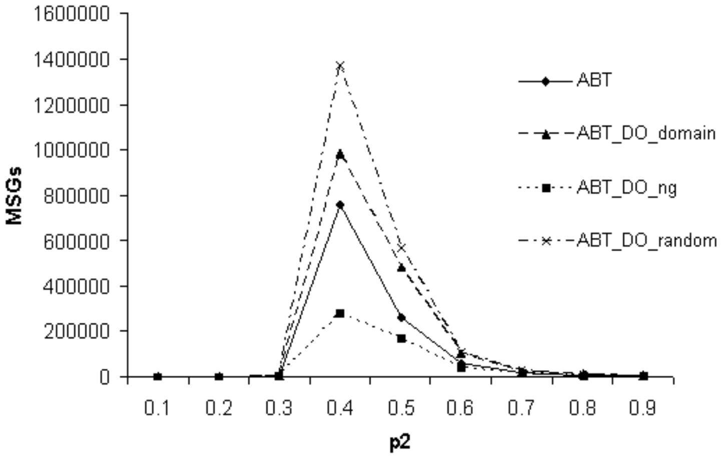

Fig. 5. Total number of messages sent by ABT and ABT_DO on low density DisCSPs ($p_1 = 0.4$).

The largest improvement is gained by using the *Nogood-trigerred* heuristic. For the hardest $DisCSP$ instances, ABT_DO with the *Nogood-trigerred* heuristic improves the performance of standard ABT by a factor of 5.

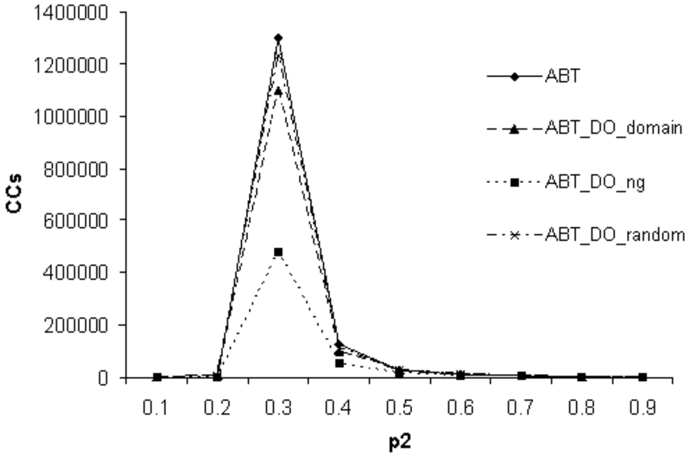

Fig. 6. Non concurrent constraints checks performed by ABT and ABT_DO using different order heuristics on high density DisCSPs ($p_1 = 0.7$).

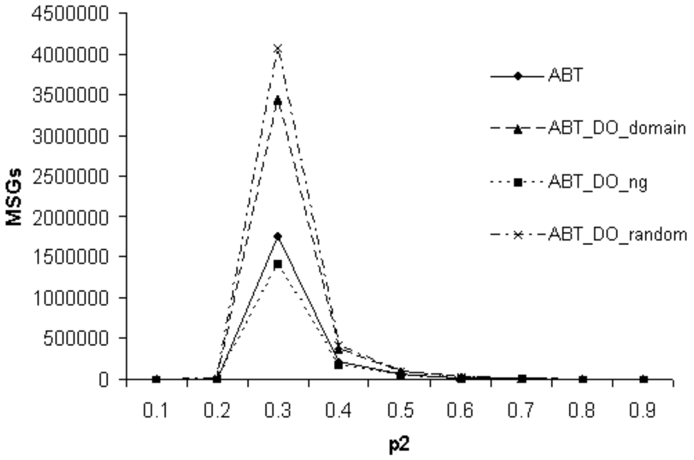

Fig. 7. Total number of messages sent by ABT and ABT_DO on high density DisCSPs ($p_1 = 0.7$).

Figure 5 presents the total number of messages sent by the algorithms for the same problems. While ABT_DO with random ordering heuristic shows similar run time results to standard ABT it sends almost twice as many messages. This can be expected since in ABT_DO agents send additional **order** messages and **ok?** messages to all their neighbors while in standard ABT, **ok?** messages are sent only to lower priority agents. ABT_DO with domain size ordering sends more messages than standard ABT but less than the random ordering version. The really interesting result is that ABT_DO with the $Nogood$-$triggered$ heuristic sends $less$ messages than ABT. Counting the additional **ok?** messages (sent to higher priority agents) and the **order** messages, it still sends less messages than standard ABT on the hardest $DisCSP$ instances.

Figures 6 and 7 present similar results in runtime, for high density *DisCSPs* with $p_1 = 0.7$. Clearly, the influence of good ordering heuristics on the performance of the algorithm is independent of network density. The results in total communication are closer than in the low density case.

7 Discussion

Dynamic ordering is a powerful heuristic used to improve the run-time of centralized CSP algorithms [Dechter, 2003] and of distributed CSP algorithms with sequential assignments [Brito and Meseguer, 2004,Nguyen *et al.*, 2004]. The results in the previous section, show that dynamic ordering must be combined with the right heuristic in order to improve the run-time and justify the overhead in message load of asynchronous backtracking. A random order heuristic does not improve the run-time of standard ABT and sends many more messages. Surprisingly, ordering the agents according to their domain size does not gain a large improvement as reported for sequential (synchronous) assignments algorithms by [Brito and Meseguer, 2004]. This can be explained by the fact that asynchronous backtracking prunes the $DisCSP$ search tree by storing $Nogoods$ which prevent it from trying to extend inconsistent tuples. $Nogoods$ are discarded in standard ABT whenever they become irrelevant [Bessiere *et al.*, 2005]. In ABT_DO, this can happen when an agent holding a $Nogood$ changes places with one of the other agents whose assignment appears in the $Nogood$. This generates the need for additional (redundent) messages reporting the same $Nogoods$.

On the other hand, the $Nogood$ triggered heuristic, inspired by *Dynamic Backtracking* [Ginsberg, 1993] was found to be very effective. In this heuristic, the above example of losing useful information cannot occur. $Nogoods$ are resolved and created according to dynamic backtracking. They include all the conflicting assignments held by the $Nogood$ sender. An agent that is moved to a higher priority position in the order, is moved lower than all the agents with conflicting assignments, therefore no $Nogoods$ are discarded. The results show that this heuristic is very effective in both measures, run-time and network load. The improvement in network load is particularly striking in view of the additional ordering messages of ABT_DO.

8 Conclusions

Most of the studies of *Asynchronous Backtracking* used a static order of agents and variables [Hamadi, 2001,Yokoo, 2000,Bessiere *et al.*, 2005,Silaghi and Faltings, 2005]. An exponential space algorithm using dynamic ordering has shown improvement in run-time over ABT [Yokoo, 2000]. The only study that suggested dynamic ordering in ABT with polynomial space used a complex method including additional abstract agents [Silaghi *et al.*, 2001]. The results presented in [Silaghi *et al.*, 2001] show a minor improvement compared to standard, static order, ABT.

The present study proposes a simple way of performing dynamic ordering in ABT with polynomial space. The ordering is performed as in sequential assignment algorithms by each agent changing only the order of agents following it in the former order. A simple method of time-stamping [Nguyen *et al.*, 2004] is used to determine the most updated proposed order.

When a heuristic order inspired by dynamic backtracking [Ginsberg, 1993] is used to dynamically reorder agents, there is a significant improvement in run-time and network load over standard ABT.

References

[Bessiere *et al.*, 2001] C. Bessiere, A. Maestre, and P. Messeguer. Distributed dynamic backtracking. In *Proc. Workshop on Distributed Constraint of IJCAI01*, 2001.

[Bessiere *et al.*, 2005] 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.

[Brito and Meseguer, 2004] I. Brito and P. Meseguer. Synchronous,asnchronous and hybrid algorithms for discsp. In *Workshop on Distributed Constraints Reasoning(DCR-04) CP-2004*, Toronto, September 2004.

[Dechter, 2003] Rina Dechter. *Constraints Processing*. Morgan Kaufman, 2003.

[Ginsberg, 1993] M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.

[Hamadi, 2001] Y. Hamadi. Distributed interleaved parallel and cooperative search in constraint satisfaction networks. In *Proc. IAT-01*, Singappore, 2001.

[Lamport, 1978] L. Lamport. Time, clocks, and the ordering of events in distributed system. *Communication of the ACM*, 2:95–114, April 1978.

[Lynch, 1997] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series, 1997.

[Meisels *et al.*, 2002] 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.

[Nguyen *et al.*, 2004] T. Nguyen, D. Sam-Hroud, and B. Faltings. Dynamic distributed backjumping. In *Proc. 5th workshop on distributed constraints reasoning DCR-04*, Toronto, September 2004.

[Silaghi and Faltings, 2005] M. C. Silaghi and B. Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. *Artificial Intelligence*, 161:1-2:25–54, January 2005.

[Silaghi *et al.*, 2001] 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.

[Smith, 1996] B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155 – 181, 1996.

[Solotorevsky *et al.*, 1996] G. Solotorevsky, E. Gudes, and A. Meisels. Modeling and solving distributed constraint satisfaction problems (dcsps). In *Constraint Processing-96*, pages 561–2, New Hamphshire, October 1996.

[Yokoo *et al.*, 1998] 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.

[Yokoo, 1995] 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.

[Yokoo, 2000] M. Yokoo. Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents & Multi-Agent Sys.*, 3:198–212, 2000.

[Zivan and Meisels, 2003] R. Zivan and A. Meisels. Synchronous vs asynchronous search on discsps. In *Proc. 1st European Workshop on Multi Agent System, EUMAS*, Oxford, December 2003.